Data, Syntax and Semantics

An Introduction to Modelling Programming Languages

J V Tucker

Department of Computer Science
University of Wales Swansea
Singleton Park
Swansea  SA2 8PP
Wales

K Stephenson

QinetiQ
St Andrews Road
Malvern
Worcestershire  WR14 3PS
England

Copyright J V Tucker and K Stephenson ©2003

This is an incomplete draft of a text-book. It is the text for a second year undergraduate course on the Theory of Programming Languages at Swansea. Criticisms and suggestions are most welcome.

April 2003
### Contents

1 Introduction ................................................................. 1  
1.1 Science and the aims of modelling programming languages ........ 2  
1.2 Some Scientific Questions about Programming Languages ........ 3  
1.3 A Simple Imperative Programming Language and its Extensions ..... 5  
   1.3.1 What is a while Program? ........................................ 5  
   1.3.2 Core Constructs .................................................. 8  
   1.3.3 Additional Constructs of Interest ............................. 10  
1.4 Conclusion ............................................................. 13  
1.5 Exercises .............................................................. 14  

2 Development of Foundations for Programming .......................... 15  
2.1 Historical Pitfalls .................................................... 15  
2.2 The analytical engine and its programming .......................... 16  
2.3 Development of Universal Machines .................................. 20  
2.4 Programming in the Decade 1945-1954 ............................. 22  
2.5 The Decade 1955–1964 ............................................... 25  
2.6 The Decade 1965–1974 ............................................... 26  
2.7 1975-1984 ............................................................ 27  
2.8 Exercises ............................................................. 28  

I Data ................................................................................. 29  

### 3 Basic Data Types and Algebras ......................................... 33  
3.1 What is an Algebra? ................................................... 34  
3.2 Algebras of Booleans .................................................. 36  
   3.2.1 Standard Booleans ................................................. 36  
   3.2.2 Bits .................................................................. 37  
   3.2.3 Equivalence of Booleans and Bits .............................. 38  
   3.2.4 Three-Valued Booleans ......................................... 39  
3.3 Algebras of Natural Numbers .......................................... 41  
   3.3.1 Basic Arithmetic .................................................. 41  
   3.3.2 Tests .................................................................. 43  
   3.3.3 Decimal versus Binary ........................................... 44  
   3.3.4 Partial Functions .................................................. 45  
3.4 Algebras of Integers and Rationals .................................... 46
5.3 Axiomatic Specification of Commutative Rings .................................. 116
  5.3.1 The Specification ......................................................... 116
  5.3.2 Deducing Further Laws ................................................. 117
  5.3.3 Solving Quadratic Equations in a Commutative Ring ............. 120
5.4 Axiomatic Specification of Fields .................................................. 123
5.5 Axiomatic Specification of Groups and Abelian Groups .................. 126
  5.5.1 The Specification ......................................................... 126
  5.5.2 Groups of Transformations ......................................... 128
  5.5.3 Matrix Transformations .............................................. 129
  5.5.4 Reasoning with the Group Axioms .................................. 130
5.6 Current Position ............................................................................ 131

6 Examples: Data Structures, Files, Streams and Spatial Objects ......... 141
  6.1 Records ............................................................................. 144
    6.1.1 Signature/Interface of Records .................................. 144
    6.1.2 Algebra/Implementation of Records ......................... 146
  6.2 Dynamic Arrays ..................................................................... 147
    6.2.1 Signature/Interface of Dynamic Arrays ................... 148
    6.2.2 Algebra/Implementation of Dynamic Arrays ............ 150
  6.3 Algebras of Files .................................................................. 152
    6.3.1 A Simple Model of Files ............................................ 153
    6.3.2 Files with Identities and Simple Permissions .......... 156
    6.3.3 File System .................................................................. 160
  6.4 Time and Data: A Data Type of Infinite Streams ...................... 165
    6.4.1 What is a stream? ....................................................... 165
    6.4.2 Time .......................................................................... 166
    6.4.3 Streams of Elements ................................................ 168
    6.4.4 Algebra/Implementation .......................................... 168
    6.4.5 Finitely Determined Stream Transformations and Stream Predicates .......... 169
    6.4.6 Decimal Representation of Real Numbers .................. 172
  6.5 Space and Data: A Data Type of Spatial Objects ....................... 176
    6.5.1 What is space? ............................................................ 176
    6.5.2 Spatial objects .......................................................... 179
    6.5.3 Operations on Spatial Objects .................................... 181
    6.5.4 Volume Graphics and Constructive Volume Geometry ...... 183
  6.6 Historical Notes and Further Reading ........................................ 190

7 Abstract Data Types and Homomorphisms .................................... 197
  7.1 Comparing Decimal and Binary Algebras of Natural Numbers ...... 200
    7.1.1 Data Translation ......................................................... 201
    7.1.2 Operation Correspondence ....................................... 201
  7.2 Translations between Algebras of Data and Homomorphisms ........ 203
    7.2.1 Basic Concept ............................................................. 203
    7.2.2 Homomorphisms and Binary Operations .................. 206
    7.2.3 Homomorphisms and Number Systems ..................... 207
    7.2.4 Homomorphisms and Machines ................................. 212
II Syntax

9 Syntax and Grammars

9.1 Alphabets, Strings and Languages ................................................. 304
  9.1.1 Formal Languages .................................................................. 304
  9.1.2 Simple Examples .................................................................. 306
  9.1.3 Natural Language Examples .................................................. 306
  9.1.4 Languages of Addresses ....................................................... 309
  9.1.5 Programming Language Examples ........................................ 310

9.2 Grammars and Derivations .......................................................... 311
  9.2.1 Grammars ........................................................................... 312
  9.2.2 Examples ............................................................................. 313
  9.2.3 Derivations ......................................................................... 316
  9.2.4 Language Generation .......................................................... 317

9.3 Specifying Syntax using Grammars: Modularity and BNF Notation .......... 318
  9.3.1 Designing Syntax using Grammars ........................................ 318
  9.3.2 A Simple Example of Building a Language using Grammars ........ 318
  9.3.3 Modularity and the Import Construct ..................................... 322
  9.3.4 User Friendly Grammars and BNF Notation ............................ 323

9.4 What is an Address? ..................................................................... 324
  9.4.1 Postal Addresses .................................................................. 325
  9.4.2 World Wide Web Addresses .................................................. 333

10 Languages for Interfaces, Specifications and Programs ....................... 341

10.1 Interface Definition Languages ................................................... 342
  10.1.1 Mathematical Definition of Signatures .................................... 342
  10.1.2 A Simple Interface Definition Language for Data Types ............ 343
  10.1.3 Example ............................................................................. 346

10.2 A Modular Interface Definition Language for Data Types .................. 347
  10.2.1 Signatures with Imports ....................................................... 348

10.3 Flattening .................................................................................... 350
  10.3.1 Repositories ......................................................................... 351
  10.3.2 Dependency Graphs .............................................................. 352
  10.3.3 Flattening Algorithm ............................................................ 352
  10.3.4 Example ............................................................................. 353

10.4 Languages for Data Type Specifications .......................................... 355
  10.4.1 Specifications ...................................................................... 355
  10.4.2 Languages for First-Order Specifications ............................... 357
  10.4.3 Languages for Equational Specifications ................................ 361

10.5 A While Programming Language over the Natural Numbers ................ 364
  10.5.1 A Grammar for while Programs over Natural Numbers ............. 364
  10.5.2 Example ............................................................................. 370

10.6 A While Programming Language over a Data Type ............................ 370
  10.6.1 While Programs for an Arbitrary, Fixed Signature ..................... 371

10.7 A While Programming Language over all Data Types .......................... 375
11 Context-Free Grammars and Programming Languages 385

11.1 Chomsky Hierarchy and Context Free Grammars ............................. 385
   11.1.1 Example ........................................................................ 387
   11.1.2 Examples ....................................................................... 388
11.2 Regular Languages ........................................................................ 389
   11.2.1 Regular Grammars .......................................................... 389
   11.2.2 Languages Generated by Regular Grammars ...................... 389
11.3 Limitations of Regular Grammars .................................................... 389
   11.3.1 The Pumping Lemma for Regular Languages ..................... 390
   11.3.2 Constructs that are not Regular ..................................... 390
11.4 Derivation Trees for Context Free Grammars ............................... 390
   11.4.1 Examples ....................................................................... 391
   11.4.2 Ambiguity ...................................................................... 393
11.5 Normal Forms for Context Free Grammars .................................. 395
11.6 Parsing Algorithms for Context Free Grammars .......................... 397
   11.6.1 Grammars and Machines .............................................. 397
   11.6.3 A Recognition Algorithm for Context-Free Grammars ....... 399
11.7 The Pumping Lemma for Context Free Languages ........................ 400
   11.7.1 Pumping Lemma ............................................................ 400
   11.7.2 Applications of the Pumping Lemma ............................... 401
   11.7.3 Proof of the Pumping Lemma ....................................... 401
11.8 Limitations of context-free grammars ......................................... 403
   11.8.1 Variable Declarations and Floyd’s Theorem ...................... 403
   11.8.3 The Concurrent Assignment Construct ............................ 407
11.9 Historical Notes ........................................................................... 410

12 Abstract Syntax and Algebras of Terms 415

12.1 Terms ...................................................................................... 417
   12.1.1 Single-Sorted Terms ...................................................... 417
   12.1.2 Many-Sorted Terms ...................................................... 421
12.2 Induction on Terms ...................................................................... 423
   12.2.1 Principle of Induction for Single-Sorted Terms ................. 424
   12.2.2 Principle of Induction for Many-Sorted Terms ................. 424
   12.2.3 Functions on Terms ....................................................... 425
   12.2.4 Comparing Induction on Natural Numbers and Terms ....... 427
12.3 Terms and Trees .......................................................................... 428
   12.3.1 Single-Sorted Trees ...................................................... 428
   12.3.2 Many-Sorted Trees ...................................................... 429
12.4 Term Evaluation .......................................................................... 430
   12.4.1 Single-Sorted Term Evaluation ..................................... 430
   12.4.2 Many-Sorted Term Evaluation ..................................... 431
   12.4.3 Term Substitution ......................................................... 432
12.5 Term Algebras ............................................................................ 433
   12.5.1 Single-Sorted Term Algebras ....................................... 433
   12.5.2 Many-Sorted Term Algebras ....................................... 434
12.6 Homomorphisms ......................................................................... 436
12.6.1 Structural Induction, Term Evaluation and Homomorphisms 436
12.6.2 Representing Algebras using Terms 439
12.7 An Algebra of while Programs 439
12.7.1 Constructing an Algebra of while programs 440
12.7.2 Representing while programs as terms 444
12.8 Context free grammars and terms 447
12.8.1 Outline of Algorithm 447
12.8.2 Construction of a signature from a context free grammar 447
12.8.3 Algebra \( T(\Sigma^G) \) of language terms 449
12.8.4 Observation 450
12.8.5 Algebra of while programs revisited 453
12.9 Context Sensitive Languages 454

III Semantics 463

13 Input-Output Semantics 467
13.1 Some Simple Semantic Problems 469
13.1.1 Semantics of a Simple Data Type 469
13.1.2 Semantics of a Simple Construct 469
13.1.3 Semantics of a Simple Program 472
13.2 Overview 473
13.3 Data 475
13.3.1 Algebras of Naturals 476
13.3.2 Algebra of Reals 477
13.4 States 478
13.4.1 Example 479
13.4.2 Substitutions in States 480
13.5 Operations and tests on states 480
13.5.1 Expressions 480
13.5.2 Example 481
13.5.3 Tests 481
13.6 Statements and Commands: First Definition 483
13.6.1 First Definition 483
13.6.2 Examples 485
13.6.3 Non-Termination 486
13.7 Statements and Commands: Second Definition using Recursion 487
13.8 Adding Data Types to Programming Languages 488
13.8.1 Adding Dynamic Arrays 489
13.8.2 Adding Infinite Streams 489

14 Proving Properties of Programs 491
14.1 Principles of Structural Induction for Programming Language Syntax 492
14.1.1 Principle of Induction for Expressions 493
14.1.2 Principle of Structural Induction for Boolean Expressions 493
14.1.3 Principle of Structural Induction for Statements 494
17.2 Structuring the Compiler .................................................. 568
  17.2.1 Defining a Compiler ................................................... 569
  17.2.2 Algebraic Model of Compilation ....................................... 570
  17.2.3 Structuring the States ................................................. 570
17.3 Proof Techniques .......................................................... 570
  17.3.1 Other Formulations of Correctness .................................... 571
  17.3.2 Data Types ................................................................... 571
  17.3.3 Recursion and Structural Induction .................................... 571
17.4 Comparing while Statements and Register Machine Programs ........ 572
17.5 Memory Allocation ........................................................... 573
  17.5.1 Data Equivalence ......................................................... 573
  17.5.2 Memory Allocation and Structuring the Register Machine States .. 574
  17.5.3 State Equivalence ......................................................... 577
17.6 Defining the Compiler ....................................................... 578
  17.6.1 Compiling the Identity Statement ...................................... 579
  17.6.2 Compiling Assignment Statements .................................... 579
  17.6.3 Compiling Sequencing .................................................. 580
  17.6.4 Compiling Conditionals ................................................ 581
  17.6.5 Compiling Iterative Statements ....................................... 583
  17.6.6 Summary of the Compiler .............................................. 585
  17.6.7 Constructing Compiled Programs .................................... 587
  17.6.8 Compiled Statements Behaviour ..................................... 587
17.7 Correctness of Compilation: Statements .................................. 589
  17.7.1 Requirements for Compiled Expressions .............................. 589
  17.7.2 Requirements for Compiled Boolean Expressions ................. 590
  17.7.3 Correctness of Statements ............................................ 590
17.8 Compiling Expressions into Register Machine Programs ............... 600
  17.8.1 The Compiler ............................................................ 600
  17.8.2 Correctness of Expression Compilation ............................. 605
17.9 Compiling Boolean Expressions into Register Machine Programs ....... 608
  17.9.1 The Compiler ............................................................ 608
  17.9.2 Correctness of Boolean Expression Compilation ................... 612

18 Further Reading .................................................................. 615
18.1 Basic Topics ..................................................................... 615
18.2 Advanced Topics ............................................................. 616
  18.2.1 Abstract Data Types and Equational Specifications .............. 616
  18.2.2 Computable Data Types ............................................... 616
  18.2.3 Fixed Points and Domain Theory ...................................... 616
  18.2.4 $\lambda$-Calculus and Type Theory ...................................... 616
  18.2.5 Concurrency ............................................................... 616
References ............................................................................ 617
Preface

Data, syntax and semantics are among the Big Ideas of Computer Science. The concepts are extremely general and can be found throughout Computer Science and its applications. Wherever there are languages for specifying, programming or reasoning, one finds data, syntax and semantics.

A programming language is simply a notation for expressing algorithms and performing computations with the help of machines. There are many different designs for programming languages, tailored to the computational needs of many different types of users. Programming languages are a primary object of study in Computer Science, influencing most of the subject and its applications.

This book is an introduction to the mathematical theory of programming languages. It is intended to provide a first course, one that is suitable for all university students of Computer Science to take early in their education; for example, at the beginning of their second year or in the second half of their first year. The background knowledge needed is a first course in elementary set theory and logic, and in imperative programming. The theory will help develop their scientific maturity by asking simple and sometimes deep questions, and by weaving them off examples and giving them a taste for general ideas, principles and techniques, precisely expressed. We have picked a small number of topics, and attempted to make the book self-contained and relevant.

The book contains much basic mathematical material on data, syntax and semantics. There are some seemingly advanced features and contemporary topics that may not be common in the elementary text-book literature: data types and their algebraic theory, real numbers, interface definition languages, algebraic models of syntax, use of algebraic operational semantics, connections with computability theory, virtual machines and compiler correctness. Where our material is standard (e.g., grammars), we have tried to include new and interesting examples and case studies (e.g., internet addressing).

The book is also intended to create an interest and provide a foundation for the further study of the theory of programming languages, and related subjects in algebra and logic, such as: initial algebra semantics; term rewriting; process algebra; computability and definability theory; program correctness logic; λ-calculus and type theory; domains and fixed point theory etc. There are a number of sources available for these later stages, and the literature is discussed in a final chapter on further reading.

The book is based on the lectures of J V Tucker (JVT) to undergraduates at the University of Leeds and at the University of Wales Swansea. In particular, it has developed from the notes for a compulsory second year course on the theory of programming languages, established at Swansea in 1989. These notes began their journey from ring binder to book-shop in 1993, when Chris Tofts taught the course in JVT’s stead, and provided the students with a typescript of the lecture notes. Subsequently, as JVT continued to teach the course, Karen Stephenson (KS) maintained and improved the notes, and assisted greatly in the seemingly endless process of revision and improvement. She also contributed topics and conducted a number of successful experiments on algebraic methods with the students. KS became a co-author of the book in 2000. Together, we revised radically the text and added new chapters on virtual machines and compiler correctness.

JVT’s interest and views on the theory of programming languages owe much to J W de Bakker, J I Zucker and J A Bergstra. Ideas for this book have been influenced by continuous conversa-
tions and collaborations starting at the Mathematical Centre (now CWI), Amsterdam in 1979. However, its final contents and shape has been determined by our own discussions and our work with several generations of undergraduate students.

We also would like to thank the following colleagues at Swansea: Dafydd Rees, Chen Min, Jens Blanck, Andy Gimblett, ... for their suggestions and advice. We are grateful to the following students who formed a reading group and gave useful criticism of earlier drafts of the text: Carl Gilbert, Kevin Hicks, Rachel Holbeche, Tim Hutchison, Richard Knuszka, Paul Marden, Ivan Phillips and Stephan Reiff. Our colleagues and students have made Swansea a warm and inspiring environment in which to educate young people in Computer Science.

The ambiguities and errors that remain in the book are solely our responsibility.

J V Tucker
Perriswood, Gower, 2003

K Stephenson
Malvern, 2003
A Guide for the Reader

The educational objectives of the book are as follows:

**Educational Objectives**

1. To study some theoretical concepts and results concerning abstract data types; programming language syntax; and programming language semantics.
2. To develop mathematical knowledge and skills in mathematically modelling computing systems.
3. To introduce the intellectual history, development, organisation and styles of the scientific study of programming languages.

At the heart of the book is the idea that the theory presented is intended to answer some simple scientific questions about programming languages. A long list of scientific questions is given in Section 1.2. To answer these questions we must mathematically model data, syntax and semantics, and analyse the models in some mathematical depth. In addition to meeting the objectives above, we hope that our book will help the students learn early in their intellectual development the following:

**Intellectual Experiences**

1. That the mathematical theory gives interesting and definitive answers to essential questions about programming.
2. That the theory improves the capacity for practical problem solving.
3. That the answers to the scientific questions posed involve a wide range of technical material, that was invented and developed by many able people over most of the twentieth century, and has its roots in the nineteenth century.
4. That the theory contains technical ideas that may be expected to be useful for decades if not centuries.
5. That intellectual curiosity and the ability to satisfy it are ends in themselves.

It will be helpful to summarise the structure and contents of the book, part by part, to ease its use by the reader, whether student or teacher.

**Subjects**

Data, syntax and semantics are fundamental and ubiquitous in Computer Science. This book introduces these three subjects through the theoretical study of programming languages.
The subjects are intimately connected, of course, and our aim is to integrate their study through the course of the whole book. However, we have found it natural and practical to divide the text into three rather obvious parts, namely on data, syntax and semantics. We begin with a general overview of the whole subject, and a short history of the development of imperative programming languages.

Part I is on data. It is an introduction to abstract data types. It is based on the algebraic theory of data and complements their widespread use in practice. We cover many examples of basic data types, and model the interfaces and implementations of arbitrary data types, and their axiomatic specifications. We focus on the data types of the natural numbers and real numbers.

Part II is on syntax. It introduces the problem of designing and specifying syntax using formal languages and grammars, and it develops a little of the theory of context free grammars. Again, there are plenty of examples to motivate and illustrate the language definition methods and their mathematical theory. Our three main case studies are simple but influential: addresses, interface definition languages and imperative programming languages. We round off the topic by applying data type theory to the definition of languages.

Part III is on semantics. It introduces some methods for defining the semantics of imperative programs using states and their transformation. It also deals with proving properties of programs using structural induction. We conclude with a study of compiler correctness. We define a simple virtual machine and compiler from an imperative language into its virtual assembler and prove by structural induction that it is correct.

In the book explanations are detailed and examples are abundant. There are plenty of exercises, both easy and hard, and including some longer assignments, and occasionally we offer suggestions for projects. The final chapter of each part contains advanced undergraduate material that brings together ideas and results from earlier chapters.

**Options**

Rarely are the chapters of a scientific textbook read in order. Lecturers and students have their own ideas and needs that lead them to re-order, select or plunder the finely gauged writings of authors. This textbook presents a coherent and systematic account of the theory of programming languages, but it is flexible and can be used in several ways. We certainly hope the book will be plundered for its treasures.

The structure of the book is depicted in Figure ?, which describes the dependency of one chapter upon another. The chapters may be read in a number of different orders. For example, all of Part I is independent of Part II, but Chapter 13 of Part II depends on Chapters 3–6 of Part I.

The complete course can be given by reading all the chapters in order. As noted, it is easy to swap the order of Part I and Part II.

A concise course that covers the mathematical modelling of an imperative language with arbitrary data types, but without the advanced topics, can be based on this selection of chapters:

**Data**  Chapters 1, 3–7

**Syntax**  Chapters 9, 10 and 11

**Semantics**  Chapters 13 and 14
Shorter and easier courses are also practical:

- **Data** Chapters 1, 3–6
- **Syntax** Chapter 9 and 10
- **Semantics** Chapters 13 and 15

A set of lectures on *data types* can be based on:

- **Data** Chapters 3–8
- **Syntax** Chapter 12
- **Semantics** Chapter 13 and 14

A set of lectures on *syntax* can be based on:

- **Data** Chapters 3–4
- **Syntax** Chapters 9–12

A set of lectures on *hierarchical structure and correct compilation* can be based on:

- **Syntax** Chapter 9 and 10
- **Semantics** Chapter 13–14 and 16–17

### Prerequisites

Throughout, we assume that readers have a sound knowledge and experience of

1. programming with an imperative language;
2. sets, functions and relations;
3. number systems and induction;
4. propositional and predicate logic; and
5. axiomatic theories.

However, our chapters provide plenty of reminders and opportunities for revising these essential subjects. Students who take the trouble to revise these topics *thoroughly*, here at the start, or even as they are needed in the book, will progress smoothly and speedily. Occasionally, for certain topics, we will use or mention some more advanced concepts and results from abstract algebra, computability theory and logic. Hopefully, these ideas will be clear and only add to the richness of the subject.

### Notation
We will follow the discipline of mathematical writing, adopting its standard notations, forms of expression and practices. It should be an objective for the student of Computer Science to master the elements of mathematical notation and style.

Notation in Computer Science is very important and always complicated by the need for concepts to have three forms of notation, designed to facilitate

(i) mathematical expression and analysis of general ideas and small illustrative examples;
(ii) reading and comprehension of large examples; and
(iii) processing by machine.

Several of our key concepts (algebras, grammars, programs, etc.) will be equipped with two standard notations aimed at (i) and (ii).

**Exercises and Assignments**

At the end of each chapter there is a set of problems and assignments designed to improve and test the student’s understanding and technique. Some have been suggested by students. The problems illustrate, complete, explore, or generalise the concepts, examples and results of the chapter. The assignments contain families of exercises on a specific topic that extends the chapter. We end this Guide with an assignment that invites students to revise the prerequisites and a piece of advice: do the assignment, and do it well.

**Assignment: Revision**

Prepare a concise list of concepts, results and notations that you believe are the basics of the five topics mentioned in the Prerequisites. Add to this list as you study the following chapters.

Here is a start.

**Sets**

<table>
<thead>
<tr>
<th>( \emptyset )</th>
<th>( A \neq \emptyset )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x \in A )</td>
<td>( x \not\in A )</td>
</tr>
<tr>
<td>( A \subseteq B )</td>
<td>( A \nsubseteq B )</td>
</tr>
<tr>
<td>( A \cup B )</td>
<td>( A \cap B )</td>
</tr>
<tr>
<td>( A - B )</td>
<td>( \mathcal{P}(A) )</td>
</tr>
<tr>
<td>(</td>
<td>A</td>
</tr>
</tbody>
</table>

**Functions**

\( f : A \rightarrow B \)

\( f : A \rightarrow B \) and \( g : B \rightarrow C \) implies \( g \circ f : A \rightarrow C \).

\( f \) is injective, or one-to-one.

\( f \) is surjective, or onto.

\( f \) is bijective, or a one-to-one correspondence.
Relations

\[ R \subseteq A \times B \quad xRy \]
\[ \equiv \text{equivalent relation} \quad [a] \text{ equivalence class} \]
\[ \leq \text{partial order} \quad : \]
Chapter 1

Introduction

Data, syntax and semantics are three fundamental ideas in Computer Science. They are Big Ideas. That is: they are ideas that are general and widely applicable; they are deep and are made precise in different ways; they lead to beautiful and useful theories. The ideas are present in any computing application. As its contributions to Science and human affairs mature, Computer Science will influence profoundly many areas of thinking and making. Thus data, syntax and semantics are three ideas which are fundamental in Science and other intellectual fields.

Our introduction to these concepts of data, syntax and semantics is through the task of modelling and analysing programming languages. Programming languages abound in Computer Science and range from large general purpose languages to the small input languages of application packages. Some languages are well known and widely used, some belong to communities of specialists, some are purely experimental and do not have a user group. The number of programming languages cannot be counted!

In learning a programming language the aim is to read, write and execute programs. After gaining fluency in one or more programming languages it is both natural and necessary to reflect on the components that make up the languages and enquire about their individual roles and properties. A multitude of questions can be formulated about the capabilities and shortcomings of a language and its relation with other languages. For example, How is this programming language specified? and Is this language more powerful than that? To raise the reader's curiosity, we will ask several questions like these shortly, in Section 1.2. To understand something of the nature and essentials of programming languages, we will dissect languages, look at their separate components and overall structure rather abstractly and analytically, and answer some of these questions.

Now, this analysis, classification and comparison of language components requires a large scale scientific investigation. The investigation is organised as the study of three aspects of programs:

<table>
<thead>
<tr>
<th>Data</th>
<th>the information to be transformed by the program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>the text of the program</td>
</tr>
<tr>
<td>Semantics</td>
<td>the behaviour of the program</td>
</tr>
</tbody>
</table>
To create a theory of programming languages we need to discover fundamental concepts, methods and results that can reveal the essential structure of programming languages, and can answer our questions.

In this first chapter we will simply prepare for our theoretical investigations. We will explore the scientific view of programming languages (Section 1.1), raise plenty of questions about programming languages for which we need answers (Section 1.2), look at some raw material for modelling programming languages (Sections 1.3 and 13.1). In Chapter 2, we give a little of the history of programming languages.

1.1 Science and the aims of modelling programming languages

Computing systems are artificial. They are designed and developed by and for human use. They are superseded and are discarded, sometimes with uncomfortable speed. However, the scientific study of computing systems is remarkably similar to the scientific study of physical systems which are God given and timeless. Roughly speaking, scientific studies have theoretical and practical objectives, pursue answers to definite questions, and require mathematical models and experimental methods.

Theoretical Computer Science develops mathematical models and theories for the design and analysis of

- data;
- algorithms for transforming data;
- programs and programming languages for expressing algorithms;
- systems and machine architectures for implementing programming languages.

Simply put, the subject is aimed towards the discovery and application of fundamental principles, models, methods and results, which are intended

- to help software and hardware engineers make computing systems;
- to help understand the nature, scope and limits of computing systems.

Theories of programming languages and program construction are a fundamental area of Theoretical Computer Science. There are many programming constructs and program development techniques and tools, all of which are the fruits of, or require, theoretical investigation. In our time, it is believed that the development of theories is necessary for the practical craft of program construction to become a mathematically well-founded engineering science. Independently of such a goal, we believe that the development of theories is necessary to satisfy our curiosity and to understand what is being done.

The scientific approach of the theory of programming languages places three intellectual requirements on the reader:
1.2. SOME SCIENTIFIC QUESTIONS ABOUT PROGRAMMING LANGUAGES

Intellectual Aims

1. To ask simple questions.
2. To make and analyse simple mathematical models.
3. To think deeply and express ideas precisely.

Now we will formulate a number of questions and set the scene for some mathematical theories to answer them. Our theories will show how it is possible

- to model mathematically any kind of data;
- to model mathematically the syntax of a programming language;
- to model mathematically the semantics of a programming language and its programs.

1.2 Some Scientific Questions about Programming Languages

Let us begin by posing some simple general questions about a programming language. The questions will make us reflect on our ideas about programming languages and programs. They require us to think generally and abstractly. They also require us to explore our existing knowledge, experience and prejudices. Most of the questions are insufficiently precise to allow a definite answer. One of our most important tasks will be to make questions precise, using mathematical models, and hence turn them into technical problems that can be solved definitively.

By no means all of the questions will be given an answer in this book: readers are invited to return to this section from time to time to see which questions they can formulate and answer precisely.

Let $\mathcal{L}$ and $\mathcal{L}'$ be any programming languages. Try to answer the following rather vague questions.

Data

What exactly are data types? What is an interface to a data type? What is an implementation of a data type? How do we specify data types like integers, reals or strings, independently of programming languages? How do we model data types for users? To what extent is a data type independent of its implementation? How do we compare two implementations of a data type? Can any data type be implemented using an imperative language? Are the representations of the natural numbers based upon decimal, binary, octal and Roman notations equivalent? How accurate as approximations are implementations of the real numbers? What are the effects of approximating infinite sets of data by finite subsets? Are error messages necessary for data types? What are the base data types of $\mathcal{L}$? What data types can be implemented in $\mathcal{L}$? Can any data type be implemented in $\mathcal{L}$?
Syntax
How do we define exactly the syntax of \( \mathcal{L} \)? How do we specify the set of legal programs of \( \mathcal{L} \)? Is there an algorithm that checks that the syntax of a program of \( \mathcal{L} \) is correct?

Semantics
How do we define exactly the semantics of \( \mathcal{L} \)? How do we specify the meaning of the data types and commands of \( \mathcal{L} \)? How do we specify the operation or dynamic behaviour of programs of \( \mathcal{L} \)? How do we specify the input-output behaviour of programs required by users? What is a program trace? What is the relationship between the number of steps in a computation and its run time? What exactly does it mean for two programs of \( \mathcal{L} \) to be equivalent?

Expressiveness and Power
How expressive or powerful is \( \mathcal{L} \)? Can \( \mathcal{L} \) implement any desired data type, function or specification? Which specifications cannot be implemented in \( \mathcal{L} \)? Is \( \mathcal{L} \) equally expressive or more powerful as another programming language \( \mathcal{L}' \)? There are four possibilities:

- \( \mathcal{L} \) and \( \mathcal{L}' \) are equivalent;
- \( \mathcal{L} \) can accomplish all that \( \mathcal{L}' \) can and more;
- \( \mathcal{L}' \) can accomplish all that \( \mathcal{L} \) can and more; or
- \( \mathcal{L} \) can accomplish some tasks that \( \mathcal{L}' \) cannot, and \( \mathcal{L}' \) can accomplish some tasks that \( \mathcal{L} \) cannot.

What exactly does it mean for two languages \( \mathcal{L} \) and \( \mathcal{L}' \) to be equivalent in expressiveness or power? Is \( \mathcal{L} \) a universal language, i.e., can it implement any specification that can be implemented in some other programming language? Do universal languages exist? Which are the most expressive, imperative, functional or logic programming languages? What is the smallest set of imperative constructs necessary to make a universal language?

Program Properties
What properties of the programs of \( \mathcal{L} \) can be specified? What properties cannot be specified? What exactly are static and dynamic properties? What properties of the programs of \( \mathcal{L} \) can be checked by algorithms? What properties are decidable, or undecidable, when the program is being compiled? What properties are decidable, or undecidable, when the program is being executed? What is the relationship between the expressiveness or power of the language \( \mathcal{L} \) and its decidable properties? What properties of programs in \( \mathcal{L} \) can be proved? Given a program and an identifier, can we decide whether or not

\begin{itemize}
  \item[(i)] the identifier appears in the program?
  \item[(ii)] given an input, the identifier changes its value in the resulting computation?
  \item[(iii)] the identifier changes its value in some computation?
\end{itemize}

Given a program and an input, can we decide whether or not the program halts when executed on the input? Given two programs, can we decide whether or not they are equivalent?
1.3. A SIMPLE IMPERATIVE PROGRAMMING LANGUAGE AND ITS EXTENSIONS

Correctness

How do we specify the purpose of a program in $L$? To what extent can we test whether or not a program of $L$ meets its specification? How do we prove the correctness of a program with respect to its specification? Given a specification method, are there programs that are correct with respect to a specification but which cannot be proved to be correct? Is there an algorithm which, given a specification and a program, decides whether or not the program meets its specification?

Compilation

What exactly is compilation? How do we structure a compiler from the syntax of $L$ to the syntax of $L'$? What does it mean for a compiler from $L$ to $L'$ to be correct? Is correctness defined using a class of specifications, or by comparing the semantics of $L$ and $L'$? How do we prove that a compiler from $L$ to $L'$ is correct?

Efficiency

Are the programs of $L$ efficient for a class of specifications? If $L$ and $L'$ are equally expressive will the programs written in them execute equally efficiently? Can $L$ be compiled efficiently on a von Neumann machine and is the code generated efficient? Are imperative programming languages more efficient than logic and functional languages? Are programs only involving while and other so-called structured constructs, less efficient than those allowing gotos?

Attempting to answer the questions should reveal what the reader knows about programming languages.

1.3 A Simple Imperative Programming Language and its Extensions

Clearly, the questions of Section 1.2 are a starting point for a major scientific investigation, one that is ongoing in the Computer Science research community. We will examine only some of them by analysing in great detail an apparently simple imperative language. The language is called the

language $WP$ of while programs with user-defined data types.

The main features of $WP$ are its very general data types and very limited control constructs. The language can compute on any data but the most complicated control construct is the while loop. The language can be extended in many ways and is a valuable kernel language, from which many more expressive or powerful (in some sense) languages can be made.

1.3.1 What is a while Program?

In our language $WP$, a program is created using

(i) some data equipped with some atomic operations; and
(ii) some algorithm that uses these atomic operations.

A program is a text composed of atomic statements that are scheduled by control constructs. The atomic statements compute the values of expressions made from given atomic operations on data. The control constructs use tests made from given Boolean-valued operations on data. We collect the data and the given operations, and form a programming construct called a data type. The statements of a program invoke operations and tests via an interface for a data type, called a signature.

Thus, we have an apparently simple idea of a program, namely:

\[
\text{Program} = \text{Data Type} + \text{Algorithm}
\]

What do the programs of our language WP look like?

Here is an example of a while program expressing an ancient famous algorithm:

```
program Euclid(input : x, y; output : y);
signature Naturals for Euclidean Algorithm
sorts nat, bool
constants 0 : \to nat
true, false : \to bool
operations mod : nat \times nat \to nat
\neq: nat \times nat \to bool
endsig
body
var x, y, z : nat;
begin
z := x \mod y;
while z \neq 0 do
\xleftarrow{} x := y;
y := z;
z := x \mod y
od
end
```

Euclid’s algorithm computes the greatest common divisor of two natural numbers. Let us examine its form and constructs (we consider its computational behaviour later, in Chapter 13).
1.3. A SIMPLE IMPERATIVE PROGRAMMING LANGUAGE AND ITS EXTENSIONS

The form of the program is two parts with three components in total. The first part is an

*interface for a data type*

which is called a **signature**. Specifically, it is:

(i) a declaration listing names for the data and for the operations on data.

The second part is an

*algorithm that uses the data type*

which is called a **body**. Specifically, it is:

(ii) a declaration of variables and their types; and

(iii) a list of commands expressing the algorithm, which uses operations from the data type

interface (i) and variables from the declarations (ii).

The constructs of the program are of two kinds. First, there are commands that apply

(a) operations on data that compute new data from old.

For example, the assignments:

\[ z \leftarrow x \, \text{mod} \, y \quad \text{and} \quad x \leftarrow y \]

Secondly, there are commands that apply

(b) operations on data that test data and control the order of execution of commands.

For example,

\[ \textbf{while } z \neq 0 \textbf{ do } \ldots \textbf{ od} \]

The program is a list of instructions that we will call

*commands* or *statements*.

The list is built by placing the commands in order using the construct of

*sequencing* denoted by ;

and usually introducing a new line of text. However, the list has some *nesting*. Notice that the

**while** construct “contains” a block of three sequenced commands.

Finally, notice that the program obeys the conventions that the algorithm uses operations and variables that are declared inside the program.

These observations of Euclid’s algorithm are merely the beginning of our search for general principles, theories and tools for building programming languages. On the basis of this example, possibly complemented by a few more, is it possible to gain a clear picture of what our **while** programs look like and to describe what they do?

It is possible to recognise, adapt and create many new specific examples of **while** programs. This is how most people learn and hence “know” a programming language! It is not possible to even recognise all **while** programs since it is not possible to define exactly what is, and what is not, a **while** program. It is not possible to give exact rules, derived or at least supported by general principles, for the composition of its programs. It is not possible to write an algorithm that would check the correctness of the program syntax (as we find in compilers). Clearly, on the basis of examples, it is not possible to answer the demanding questions we posed in Section 1.2.
1.3.2 Core Constructs

Let us review and formulate in more general terms some of the observations we made in the previous section.

Program Structure

The overall form of a **while** program is summarised in Figure 1.1. We concentrate on the

![Figure 1.1: General form of a while program.](image)

elements of the components.

More precisely, the principal constructs of the **while** language **WP** are as follows:

Data

*Any* set of data, and any operations and tests on the data, can be defined. Data, and operations and tests on data together form a

*data type.*

The interface by which a **while** program invokes a data type is called a signature and has the form:
1.3. A SIMPLE IMPERATIVE PROGRAMMING LANGUAGE AND ITS EXTENSIONS 9

signature a data type
sorts ... , s, ...
constants ... , c : \rightarrow s, ...
operations ... , \ f : s_1 \times \cdots \times s_n \rightarrow s, ...
endsig

Here sorts are names for the sets of data, constants are names for special data and operations are names for functions and tests.

Variables and State
There are infinitely many variables
\[ x_0, x_1, x_2, \ldots \]
also denoted x, y, z, \ldots etc. The variables “store” data and give rise to a notion of state of a computation or memory.

Atomic Statements
An expression \( e \), built from the variables and operators of the data type, can be evaluated and taken to be the new value of any variable \( x \) by an

\[
\text{Assignment} \quad x := e.
\]

We will also often include a dummy atomic statement which does nothing:

\[
\text{Skip} \quad \text{skip}
\]

Control Constructs
The order of performing operations is controlled by three constructs: Given programs \( S_0, S_1, S_2 \) and test \( b \), we can construct the new programs:
### Sequencing

- **Sequencing**: $S_1; S_2$

### Conditional

- **Conditional**: if $b$ then $S_1$ else $S_2$ fi

### Iteration

- **Iteration**: while $b$ do $S_0$ od

---

**Statements**

A **while** program is built from atomic statements by the repeated application of control constructs.

The theory of this small set of programming constructs will provide answers to many of the questions, and be strong enough to found theories of much larger languages.

---

#### 1.3.3 Additional Constructs of Interest

There are many omissions from the rich collections of constructs used in everyday imperative programming. Hence, there are many possible additional constructs of interest, each giving rise to a language that is an enhancement of **WP**. Let us consider some missing constructs, most of which we will meet later. We will not stop to explain, even vaguely, what the constructs do. We simply want to review the scope of our language.

---

**Data**

Special data types can be added and exploited such as integers, reals, characters, etc.

Time can be made explicit and constructs that schedule data according to a clock can be added:

- **Clocks**:
  
  \[0, 1, 2, \ldots, t, \ldots\]

- **Streams**:
  
  \[a(0), a(1), a(2), \ldots\]

Structures for **storing data** can be added such as records, stacks, queues, files and, for example:

- **Arrays**:
  
  \[a[i] := e\]
1.3. A SIMPLE IMPERATIVE PROGRAMMING LANGUAGE AND ITS EXTENSIONS

**Atomic Statements**

The computation of *basic operations* can be performed simultaneously:

- Concurrent assignment:
  
  \[ x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n \]

  Rather like *skip*, special statements can be added that stop or terminate a computation:

  - **halt**

or force an endless continuation or non-termination of a computation:

  - **diverge**

**Control Constructs**

Other forms of *testing and control* can be added. Given a program \( S \) and test \( b \):

- Bounded iteration:

  ```
  do e times S od
  for i := 1 to e do S od
  ```

  where \( e \) is an expression of type natural number.

- Unbounded iteration:

  ```
  repeat S until b end
  repeat S forever
  ```

- Conditional:

  ```
  case statements
  ```

- Branching and jumps:

  ```
  goto label
  exit statements for loops
  ```

**Error handling**

Interrupts can be added in different ways as an:

- Atomic statement:

  ```
  error name
  ```

- Control construct:

  ```
  exception label
  ```
CHAPTER 1. INTRODUCTION

Abstractions

Abstractions from the state and operation of a program can be added:

- Functions can be abstracted by procedures such as:

  \[
  \text{function } y = F(x) \text{ is } S \text{ end}
  \]

  \[
  \text{func in } a \text{ out } b \text{ aux } c \text{ begin } S \text{ end}
  \]

Abstractions from the control and operation of a program can be added:

- Recursively defined procedures and functions.

Languages made by adding the above constructs are all \textit{deterministic}. Each construct determines one and only one operation or sequence of operations. Hopefully, most of them are instantly recognisable.

Now, we can consider constructs that introduce \textit{non-determinism} into computations. Perhaps these are new or less obvious to the reader. They each have valuable and essential roles in program design.

Non-deterministic Atomic Statements

There are constructs that choose data from specific sets such as natural, integer or real numbers, or from any set:

- Non-deterministic assignments:

  \[
  x := \text{random}(0, 1)
  \]

  \[
  x := ?
  \]

  \[
  x := \text{choose} \ (z : b(z))
  \]

  \[
  x := \text{some } y \text{ such that } P(y)
  \]

Non-deterministic Control Constructs

There are constructs that make a choice in passing control:

- Non-deterministic selection:

  \[
  \text{choose } S_1 \text{ or } S_2 \text{ ro}
  \]

- Guarded commands:

  \[
  \text{do } b_1 \rightarrow S_1 \text{ or } b_2 \rightarrow S_2 \text{ or } \ldots \text{or } b_n \rightarrow S_n \text{ od}
  \]

  \[
  \text{if } b_1 \rightarrow S_1 \text{ or } b_2 \rightarrow S_2 \text{ or } \ldots \text{or } b_n \rightarrow S_n \text{ fi}
  \]
1.4 Conclusion

We have introduced the theoretical investigations to come by asking questions and sketching some raw material in need of mathematical analysis. We will observe more aspects of data, syntax and semantics and build up a stock of examples that will serve as raw material for our theories.

At the heart of our analysis of the language WP and its extensions is its formal definition. The answers to the questions of Section 1.2, when WP is substituted for the arbitrary language \( \mathcal{L} \), depend on the answers to the questions:

- **Data**  
  *How do we define exactly the data types of WP, or of an extension?*

- **Syntax**  
  *How do we define exactly the legal program texts of WP, or of an extension?*

- **Semantics**  
  *How do we define exactly the computations by the programs in WP, or of an extension?*

As we gain a precise understanding of data, syntax and semantics, we can proceed to formulate the questions exactly and try to answer them.

The main objectives are to develop the elements of

- a *theory of data* that is a foundation for computation;
- a *theory of syntax* that allows the specification and processing of languages; and
- a *theory of semantics* that defines the meaning and operation of constructs and explains how a program produces computations.

These theories are the foundations for understanding extensions to the language and for the new objectives, such as creating a *theory of program specification*; and a *theory of compilation*. 
1.5 Exercises

1. Give three Big Ideas in Physics, Biology and Economics.

2. What is the point of making mathematical models of systems in Physics, Biology and Economics?

3. Write out in English the meaning of the following constructs:
   a. concurrent assignments \( x_1, \ldots, x_n := e_1, \ldots, e_m \);
   b. sequencing \( S_1; S_2 \);
   c. conditional \( \textbf{if} \ b \ \textbf{then} \ S_1 \ \textbf{else} \ S_2 \ \textbf{fi} \);
   d. iteration \( \textbf{while} \ b \ \textbf{do} \ S_0 \ \textbf{od} \); and
   e. infinite iteration \( \textbf{repeat} \ S_0 \ \textbf{forever} \);

   Where \( x_1, \ldots, x_n \) are variables, \( e_1, \ldots, e_m \) are expressions, \( n \) need not equal \( m \), \( b \) is a Boolean test and \( S_0, S_1, S_2 \) are programs. Try to make your description exact and complete. Note any possible curious variations.

4. Using the core constructs of the \textbf{while} language \( WP \) (as in Section 1.3.2) write programs for the following:
   a. sorting a list of integers using the bubble-sort algorithm;
   b. listing prime numbers using the sieve algorithm of Eratosthenes;
   c. finding an approximation to the square root of a real number; and
   d. drawing a straight line on the integer grid using Bresenham’s algorithm.

   Take care to define the interfaces to the underlying data types.

5. Show how to express the following constructs in terms of the core constructs of the \textbf{while} language \( WP \):
   a. the \textbf{repeat} statement;
   b. the \textbf{repeat-forever} statement;
   c. the \textbf{for} statement; and
   d. the \textbf{case} statement.

6. Use a non-deterministic assignment to construct a program that, given a real number \( x \) and a rational number \( \epsilon \), finds a rational number within a distance \( \epsilon \) from \( x \).

7. Investigate the use of guarded commands in programming methodology by reading Dijkstra [1976].
Chapter 2

Development of Foundations for Programming

We will continue our introduction by looking at the historical development of our present understanding of programming. We will give an impression of how practical problems in programming require conceptual insights and mathematical theories for their solution. Again and again difficult research problems are transferred into standard practices. We will focus on the technical themes and problems which have motivated and shaped the theory in this book, namely:

- Machine architectures and high-level programming constructs
- Machine independence of programming languages
- Unambiguous and complete definitions of programming languages
- Translation and compilation of programs
- User defined data types
- Specification and correctness of programs

We begin with the origins of imperative programming in the work of Charles Babbage on the Analytical Engine. Then we jump to some twentieth century problems, research and developments, associated with the above themes. These will be grouped into four decades, the first starting in 1945, the era of the first electronic stored-program universal computers.

2.1 Historical Pitfalls

Right from the start, one must beware that the historical analysis of technical subjects is beset by three temptations:
**Temptations**

1. The wish to reconstruct the past as a clearly recognisable path leading to the present.
2. The wish to find and create heroic figures whose achievements more or less define the story.
3. The wish to separate the work from the life and times of the society in which it took place.

The three temptations are well known to all historians. The first is called *teleological history*. The second and third are particular curses of writers on the history of technical subjects (like physics, mathematics and computing).

All three are temptations to be selective, to be blind to the complexity and messiness of scientific work, to the slowness of scientific understanding, to the immense amount of work, by many people, that is necessary to produce a high quality and useful subject in science and engineering. To simplify in this way can lead to a history that is so misleading as to be almost worthless. *A distorted history can distort our picture of the present and future.*

The early development of computers and programming languages illustrate a common state of affairs in scientific research: there are many objectives, problems, methods, projects, and people playing independent and significant roles over long periods of time. Gradually, some convergence and standardisation takes place, as problems, ideas and people are celebrated or forgotten. Histories of computers and programming languages must reflect the scale and diversity of activities, in order to be honest and useful to contemporary readers who may be caught up in similarly messy intellectual circumstances.

Of course the problem with history, as with software and much else, is that it is difficult to get big things "right"! In the sketch that follows, I can only hope to awaken curiosity and to give a health warning about the simplifications I am forced to make.

### 2.2 The analytical engine and its programming

Our contemporary ideas about programs and programming can clarify and organise our attempts to appreciate and reconstruct the ideas of Charles Babbage on programming. Equally, they may obscure or mislead us in reinterpreting and extending his achievements (compare Temptations 1 and 2 above!). There is a small number of draft programs and notes, in contrast with the large amount of information on the machine. There are no complete programs and there is no detailed specification of all the instructions. There are the descriptions of the machine, and the paper by L Menabrea and the important set of notes by Ada Lovelace of 1845.

The evolution of the Analytical Engine began in 1833 when work on the Difference Engine stopped. The first notes of 1834 are workable designs. In late 1837 there is a major revision which was the basis for the development until 1849, when Babbage ceased work until starting up again in 1856.

A clear and concise specification of an early form of the Analytical Engine is found in the
2.2. THE ANALYTICAL ENGINE AND ITS PROGRAMMING

letter Babbage wrote to the Belgian statistician Adolphe Quetelet, dated 27 April 1835:

It is intended to be capable of having 100 variables (or numbers susceptible of change) placed upon it; each number to consist of 25 figures.

The engine will then perform any of the following operations or any combination of them: \(v_1, v_2, \ldots, v_{100}\) being any number, it will

- add \(v_i\) to \(v_k\);
- subtract \(v_i\) from \(v_k\);
- multiply \(v_i\) by \(v_k\);
- divide \(v_i\) by \(v_k\);
- extract the square root of \(v_k\);
- reduce \(v_k\) to zero.

Hence if \(f(v_1, v_2, \ldots, v_n), n < 100\), be any given function which can be formed by addition, multiplication, subtraction, division, or extraction of square root the engine will calculate the numerical value of \(f\). It will then substitute this value instead of \(v_1\) or any other variable and calculate the second function with respect to \(v_1\).

It appears that it can tabulate almost any equation of finite differences.

Also suppose you had by observation a thousand values of \(a, b, c, d\), and wish to calculate by the formula

\[
p = \sqrt{(a + b)/cd}.
\]

The engine would be set to calculate the formula. It would then have the first set of the values of \(a, b, c, d\) put in. It would calculate and print them, then reduce them to zero and ring a bell to inform you that the next set of constants must be put into the engine.

Whenever a relation exists between any number of coefficients of a series (provided it can be expressed by addition, subtraction, multiplication, division, powers, or extraction root) the engine will successively calculate and print those terms and it may then be set to find the value of the series for any values of the variable.

Extract of a letter from C Babbage to A Quetelet, 27 April 1835.

*Works of Babbage. Volume III. Analytical Engine and Mechanical Notation.*


The idea of the machine contained in this letter can be explored by making a simple mathematical model based on a few features. In particular, one might first guess that:

(i) the machine simply computes arithmetic expressions based on the five operators and 100 variables mentioned;
(ii) the expression is changed;

(iii) each expression is repeatedly evaluated on a finite sequence or stream of values for the variables; and

(iv) loops do not occur.

What does the remark about tabulating all finite difference equations imply? The specification is too loose and vague for us to answer semantic questions with confidence.

We will develop a formal model based on these four assumptions in due course.

This description of 1835 predates the idea of using Jacquard cards (which has been dated to the night of 30 June 1836). The idea of the Jacquards, or punched cards, is fundamental. It enriches the conception of the machine and its use, making explicit the breaking down and sequencing of its actions. It is an essential feature of mechanisation to break down processes to actions and operations. A number of types of cards were envisaged, notably variable and operation cards. In a description of the engine in 1837, the operation cards were to be used for addition, subtraction, multiplication and division.

In the paper by L Menabrea, there is an emphasis on (what we may call) the algorithmic aspects of the Analytical Engine, and this is further developed in the Lovelace notes. A few “programs” are discussed and, to add to our stock of raw materials for our theories, we will look at the “program” for solving a pair of simultaneous equations.

Consider the pair

\[ mx + ny = d \]
\[ m'x + n'y = d' \]

of equations, where \( x \) and \( y \) are the unknowns. By eliminating the variable \( y \) in the usual way, we find

\[ x = \frac{dn' - d'n}{n'm' - nn'} \]

and, similarly eliminating \( x \), we find

\[ y = \frac{d'm' - d'n'}{m'n' - nn'} \]

The problem is to program the machine to calculate \( x \) and \( y \) according to these formulae.

The letters

\( V_0, V_1, V_2, V_3, V_4 \) and \( V_5 \)

denote the columns reserved for the coefficients and have values

\( m, n, d, m', n' \) and \( d' \)

respectively. A series of Jacquard cards provides the series of arithmetic operations that evaluate the expressions on the machine. They involve a number of working variables \( V_0, \ldots, V_4 \), and two output variables \( V_{15} \) and \( V_{16} \). The series of operations and their effects on all the columns of the machine is presented in a table given as Figure 2.1.
Figure 2.1: A program trace for solving simultaneous equations on the Analytical Engine
We leave the reader to ponder on the calculation and the underlying program from this trace. Notice that when the value of a variable column is no longer required, then it is re-initialised to the value 0.

From time to time we will look at the treatment of data, program control, and the methodology of programming the Analytical Engine. Our aim in looking at old ideas is not to write their history. We will pick on historical concepts and methods to suggest examples to illuminate and enrich contemporary ideas.

### 2.3 Development of Universal Machines

The current conception of programming is based on the ideas of a stored-program computer and, in particular, a universal computer. The concept and basic theory of the universal computer was developed in mathematical logic, starting with A Turing’s work in 1936. However, most of the early projects on computing machines did not use, nor were aware of, this theory, until J von Neumann entered the area.

The development of computers is an interesting subject to study for many obvious reasons. An unobvious reason is that the subject is so complicated. There are many contributions (concerning electronics, mathematics, computer architecture, programming and applications) to examine and classify. There are many historical problems in evidence (recall Section 2.1).

To complement our remarks on the Analytical Engine, we will present a very simplified picture of the development of universal machines based on a classification of their programming capabilities; it is summarised in Figure 2.2. We hope it is a useful reference for a proper study of the origins of programming.
2.3. DEVELOPMENT OF UNIVERSAL MACHINES

Manual Program Sequence
1938 G Stibitz
  Model I
  Completed 1940

Automatic Program Sequence

External Program
1936 K Zuse
  Z3
  Completed 1941
1939 H Aitken
  Mark I
  Completed 1944

Internal Program

Plugged Program
1943 J P Eckert &
  J W Mauchley
  ENIAC
  Completed 1946

Stored Program

Fixed Program Storage
1948 Clippinger
  ENIAC modification

Variable Program Storage
1944 J P Eckert & J W Mauchly
  Completed 1949
1945 J von Neumann & H Goldstine
1948 F C Williams & T Kilburn
  Completed 1949 M V Wilkes

Figure 2.2: Some stages in the development of electronic universal machines.
2.4 Programming in the Decade 1945-1954

- Stored program computers.
- Logical specification of computer architectures.
- Algorithmic and programming languages.
- Automatic programming seen as compilation or translation of programs.
- Problem of programming cost recognised.
- Problem of program correctness recognised.

Programming languages and programs emerged with computers. During the Second World War electro-mechanical and electronic digital computing machines were produced for special applications; for example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Responsible</th>
<th>Demonstration Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model I</td>
<td>G Stibitz</td>
<td>1940</td>
</tr>
<tr>
<td>Z3</td>
<td>K Zuse</td>
<td>1941</td>
</tr>
<tr>
<td>Differential Analyser</td>
<td>J Atanasoff &amp; C Berry</td>
<td>1942</td>
</tr>
<tr>
<td>ASCC (Harvard Mark I)</td>
<td>H Aiken, C D Lake, F E Hamilton &amp; B M Durfee</td>
<td>1943</td>
</tr>
<tr>
<td>COLOSSUS</td>
<td>T H Flowers &amp; M H A Newman</td>
<td>1943</td>
</tr>
<tr>
<td>ENIAC</td>
<td>J Mauchly &amp; J P Eckert</td>
<td>1945</td>
</tr>
</tbody>
</table>

Zuse’s Z3 is the earliest electro-mechanical general purpose program controlled computer. The ENIAC (Electronic Numerical Integrator and Computer) is well known as an early electronic computer that was close to being general purpose. It was programmed by setting switches and plugging in cables. Later it was redeveloped with stored program facilities (the EDVAC completed in 1952), and led to a commercially available machine the UNIVAC.

By 1949 some electronic universal stored program machines were operational; for example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Responsible</th>
<th>Demonstration Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype Manchester Mark I</td>
<td>F C Williams &amp; T Kilburn</td>
<td>1948</td>
</tr>
<tr>
<td>EDSAC</td>
<td>M Wilkes</td>
<td>1949</td>
</tr>
<tr>
<td>IAS Computer</td>
<td>J von Neumann &amp; H Goldstine</td>
<td>1951</td>
</tr>
</tbody>
</table>
2.4. PROGRAMMING IN THE DECADE 1945-1954

The Manchester Mark I was the first electronic stored program machine to operate, on 21 June 1948; it was developed into the first commercially available machine, the Ferranti Mark I, and sold to 11 customers. The EDSAC (Electronic Delay Store Automatic Calculator) which was successful in practical terms and played a significant rôle in the development of programming. The IAS Computer was developed 1946-52 and was influential from its inception. The designs of these and other machines were copied and refined by many groups and companies within the decade.

Like Babbage's creations, these early electrical machines are great achievements. For our purposes, that of reflecting on the origins of programming, the early machines can be classified by the physical properties of their programs. For example, the means for program sequencing and the storage of programs lead us to Figure 2.2.

The design of such machines combined engineering and mathematical ideas and, in the case of the earliest machines, they were programmed by the people who built them. To establish the independent status of programming, a logical view of a machine is essential, one that abstracts from the machine's physical implementation. Isolating the logical view of a machine and reflecting on programming structures was the work of scientists trained in mathematical logic.

Mathematical logic has played a completely fundamental rôle in computer science from its inception, especially in the field of programming languages. The most important contribution was Alan Turing's concept of a mathematical machine and the theorems about its scope and limits that he proved in 1936. The theorem on the existence of a universal Turing machine able to simulate all other Turing machines is the discovery of the universal stored program computer. Turing's discoveries led to the theory of computable functions which played a decisive role in the development of mathematical logic in the previous decade 1935-1944. Historically, the theory of computable functions is the start of the theory of programming languages.

As part of the ENIAC project, the logical idea of machine architecture was made explicit in John von Neumann's First draft of a report on the EDVAC of 1945. The independence of the architecture from technology was emphasised by describing components not in terms of current technologies but using logical components taken from McCulloch and Pitts pioneering work on logical models of the nervous system of 1943; the theory of neural networks was inspired by computability theory. A full account of the basic features of a computer architecture appears in A Burks, H Goldstine and John von Neumann's Preliminary discussion of the logical design of an electronic computing instrument. The report was written in 1946, at the start of the project to build the IAS Computer and was widely circulated. The report helped promote the term von Neumann architecture, though many of the ideas were known to others.

Programming these machines was difficult, partly because of the deficiencies of the hardware, and partly because of the codes used to communicate with the machine. To develop computing, the general problem of programming must be tackled:

General Question

How can programming be made less time consuming and error prone?

Plus ça change, plus c'est la même chose!

However, in the decade 1945-55 this meant more precisely,
Problem 1

How can the logical control and data structure of a program be modelled and understood?

In 1945-46, Konrad Zuse invented a general language for expressing algorithms called the Plankalkül. It was not intended to be machine implementable but to provide a framework for preparing programs. Appropriately, it is referred to as an algorithmic language rather than a programming language. It was based on the data type of booleans and had some type forming operations. Sample programs included numerical calculations and testing correctness of expressions. See Bauer and Wössner [1972] for a summary.

Herman Goldstine and John von Neumann wrote a series of papers called Planning and coding problems for an electronic computing instrument in 1947-48. (See Goldstine and von Neumann [1947].) As can be seen from their classification of the programming process into six stages, their view was that programs were needed to solve mainly applied mathematical problems. Among their ideas, they used flow diagrams as a pictorial way of representing the flow of control in a program containing jump instructions. These reports were essentially the only papers on programming widely available until the first text-book on programming, namely Wilkes, Wheeler and Gill [1951], based on the EDSAC.

In addition to understanding the logic of programs, there was the following:

Problem 2

How can we make programming notations that are more expressive, easier to understand and to translate into machine codes?

There was a need for high level notations and tools to translate higher level notations into machine codes. Hardware was limited and much needed to be programmed. Since programming meant writing programs in terms of primitive codes for specific machines, this led to the idea automatic programming. Early on some recognised that the inadequacies of programming notations and tools for programming were the main barriers to the development of computing. Linked to the ideas of programming methodologies and languages was the following:

Problem 3

How can we reason about the behaviour of programs?

Problems of reasoning about programs were noted in the Planning and coding reports of Goldstine and von Neumann. At the inaugural conference of the EDSAC in 1949, Alan Turing lectured on checking a large routine (see Morris and Jones [1984]).
2.5 The Decade 1955–1964

- Independent levels of computational abstraction.
- Compiler construction.
- Machine independence of programming languages.
- Formal definition of programming language syntax.
- Formal definition of operational semantics of programming languages.
- Mathematical theory of grammars and parsing.
- Functional languages.
2.6 The Decade 1965–1974

- Formal definition of programming language semantics.
- Vienna definition language for operational semantics.
- Denotational semantics of programming languages.
- Axiomatic semantics of programming languages.
- Specification and verification of programs.
- Formal logics for program correctness.
- Automatic verification.
- Data structures developed.
- Automatic programming as interpretation and compilation of specification.
- Modularisation of programs.
- Object-oriented languages.
- Logic programming languages.
- Interfaces with pull down menus, buttons and mice.
2.7 1975-1984

- Abstract data types.
- Algebraic theory of data types.
- Abstraction and modularisation principles for the formal definition of large systems.
- Domain theory.
- Algebraic description of concurrency.
- Specification languages.
- Modal and temporal logics and model checking.
- Programming environments and software tools.
- Concurrent programming languages.
2.8 Exercises

1. Describe the input-output behaviour of the Analytical Engine of 1835. List the programming constructs that are needed to formulate assumptions (i)–(iv).

2. Write a program for solving simultaneous equations, in a while language, that would generate a trace similar to that of Lovelace’s program trace in Section 2.2.

3. Examine Lovelace’s notes and
   a. explain the rôle of the superscript $i$ in the notation $iV_j$;
   b. describe what data could be processed;
   c. describe what control constructs were intended; and
   d. write an account of the calculation of the Bernoulli numbers.

4. Add the names of some of the originators of the above concepts, and the dates when they were exploited commercially.

5. Trace the history of the following over the same time-scale:
   a. graphics programming;
   b. machine translation of natural languages; and
   c. parallel computation, starting with the COLOSSUS, neural networks and cellular automata.

6. Was there a “Dark Age” for Computer Science from the time of Babbage to the 1930s?


8. Write a list of concepts for the decade 1985-94.

Assignment for Chapter 2

Write a list of concepts in mathematical logic in the decades 1925-34 and 1935-44 that are relevant for the history of computer science. Explain these ideas and their history. When and in what ways did they influence the development of programming languages?
Part I

Data
Introduction

Data is represented, stored, communicated, transformed and displayed throughout Computer Science. Data is one of the Big Ideas of Computer Science — it is as big as the ideas of energy in physics, molecules in chemistry and money in economics. Theories about data are fundamental to the subject. In Part I, we will introduce the study of data in general, using ideas from abstract algebra. The subject may be called the

*algebraic theory of data*

and it will be applied to the theory of programming languages in several ways.

Indeed, we begin the study of the syntax and semantics of programming languages by examining the idea of a *data type*. A data type is a programming construct for defining data. Its purpose is to name and organise data, and the basic operations and tests that we may apply to data, in a program. It consists of an interface and an implementation.

We will introduce the mathematical concepts of

*many-sorted signature* and *many-sorted algebra*

to model a data type. A signature contains names for types of data, particular data, operations and tests on data. It models the interface of the data type. An algebra contains sets of data, particular data, operations and tests on the sets of data. It models an implementation of the data type. The theory of many-sorted algebras is the basis of the general theory of data in Computer Science which is commonly called the

*theory of abstract data types.*

The many-sorted algebra is the single most important concept in this book.

To start the development of the theory of abstract data types, in Chapter 3 we give a working definition of an algebra and present several examples of algebras in order to familiarise the reader with some of the concepts and objectives of the theory. In Chapter 4, we replace the working definition of an algebra with a detailed formal definition of a signature and an algebra. We show how signatures and algebras model interfaces and implementations of data types.

Next, we meet general methods for constructing some new algebras from old; for example, in Chapter 6 we study algebras of records, arrays, streams and files.

The theory of abstract data types is introduced in Chapter 7.

To conclude Part I, we use many of the ideas we have introduced in an analysis of the data type of real numbers in Chapter 8.

In Chapter 12 of Part II, we study algebras of syntax, including terms, trees and programs.
Chapter 3

Basic Data Types and Algebras

In any computation there is some data. But to perform a computation with that data there must also be some primitive operations and tests on data. The algorithm or program that generates the computation uses the primitive operations and tests in producing some output from some input. The concept of data type brings together some data in a particular representation and some operations and tests to create a starting point for computation; roughly speaking:

\[ \text{Data Type} = \text{Data + Operations + Tests}. \]

To make a precise mathematical model of a data type is easy because we have the theory of sets, functions and relations from Mathematics.

The theory of sets was an advanced theory created by Georg Cantor (1845–1918) to sort out some problems in the foundations of mathematical practice, largely to do with conceptions of the infinite. Cantor’s later definition of a set, of 1895, reveals the abstract nature and wide scope of set theory:

“A set is a collection into a whole of definite distinct objects of our perception or thought. The objects are called the elements (members) of the set.”

This degree of abstraction is exactly what we need to create a general mathematical theory of data. First, data is collected into sets; indeed, for our purposes, the elements of a set can be simply called data. Next, the operations on data are modelled by functions on those sets, and tests are modelled by either relations on the sets or functions on the sets returning a Boolean truth-value. Choosing the latter option, we have a model of a data type called an algebra which is:

\[ \text{Algebra} = \text{Sets + Functions + Boolean-valued Functions}. \]

In this chapter we will begin to develop this model by examining in detail some basic examples of data, and of the operations and tests that algorithms use to compute with the data. The examples of basic data types we model as algebras simply equip the following sets of data with operations and tests:

- Boolean truth-values \( B = \{tt, ff\} \)
- Natural numbers \( N = \{0, 1, 2, \ldots\} \)
- Integer numbers \( Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \)
- Rational numbers \( Q = \left\{ \frac{p}{q} \mid p, q \in Z, q \neq 0 \right\} \)
- Real numbers \( R \)
- Characters and strings \( \{a, b, c, \ldots, x, y, z\} \) and \( \{a, b, c, \ldots, x, y, z\}^* \)
Most other data types in computing are implemented using these data and various data structures. The data structures are programming language constructs for storing and accessing data, such as records and arrays, or higher level data types, such as streams, graphs and files. We will study data structures in Chapter 6.

In Section 3.1 we introduce a provisional definition of an algebra. We illustrate this definition with examples of algebras of Booleans (Section 3.2), natural numbers (Section 3.3), integer and rational numbers (Section 3.4), real numbers (Section 3.5), machine data (Section 3.6) and strings (Section 3.7). In these examples, we meet basic theoretical questions about algebras which lead us, in Chapter 4, to reformulate our definitions.

### 3.1 What is an Algebra?

Let us begin with a provisional definition of the general idea of an algebra that will help us collect and reflect upon some basic examples. The provisional definition is inadequate because it lacks the important associated idea of signature; it will be refined to its final form in Section 3.2.

Simply said, an algebra consists of sets of data together with some functions on the sets of data. The functions provide some basic operations and tests for working with the data. An algebra can contain many different kinds of data and any number of functions. For example, to have tests on data the algebra must contain the set of Booleans. The general idea of an algebra does not assume there are tests.

**Provisional Definition (Many-Sorted Algebra)**

A *many-sorted algebra* $A$ consists of:

1. A family

   $$\ldots, A_s, \ldots$$

   of non-empty sets of *data* indexed by $s \in S$. The elements $\ldots, s, \ldots$ of the non-empty set $S$ that index or name the sets containing the data are called *sorts*. Each set $A_s$ is called a carrier set of sort $s \in S$ of the algebra.

2. A collection of elements from the carrier sets of the form

   $$\ldots, a \in A_s, \ldots$$

   called the constants of the algebra.

3. A collection of functions on the carrier sets of the form

   $$\ldots, f : A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n} \to A_s, \ldots$$

   for $s, s_1, s_2, \ldots, s_n \in S$, called the operations of the algebra.

Now, the algebra $A$ is called a

*many-sorted algebra*
3.1. WHAT IS AN ALGEBRA?

because there can be many carrier sets $A_s$ of data, named by the many sorts $s \in S$. When the set $S$ of sorts has only one element, $S = \{ s \}$, then $A$ is called a

\[ \text{single-sorted algebra} \]

because $A$ contains one carrier set of data, named by one sort.

The constants are sometimes treated as functions with no arguments and are called 0-\textit{ary functions} or \textit{operations}. In this case, instead of $a \in A_s$ we write

\[ a : \rightarrow A_s \quad \text{or} \quad a : A_s. \]

An algebra $A$ consists of these three components. It is written concisely as a list of the form

\[ A = ( \ldots, A_s, \ldots; \ldots, a, \ldots; \ldots, f, \ldots ). \]

This is convenient for mathematical reasoning about algebras in general.

Alternatively, an algebra may be displayed expansively in the form:

| algebra   | $A$ |
| carriers  | $\ldots, A_s, \ldots$ |
| constants | $\vdots$ |
| \hspace{1em} $a : \rightarrow A_s$ |
| \hspace{1em} $\vdots$ |
| operations | $\vdots$ |
| \hspace{1em} $f : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ |
| \hspace{1em} $\vdots$ |

This is convenient for introducing and reasoning about particular examples.

The general idea of an algebra is easy to grasp. Whenever there is a function $f : A \rightarrow B$ we can construct an algebra

\[ (A, B; f) \]

with carriers $A$, $B$ and operation $f$. Thus, we see that:

\[ \text{The concept of an algebra is as general as that of a function.} \]

Indeed, thinking of a collection of functions, we see that:

\[ \text{By simply grouping together the sets of data, elements and functions, we form an algebra.} \]

Thus, algebras are simply a way of organising a collection of elements and functions on sets. Wherever there are functions on sets, there are algebras.

We will give a series of examples of algebras by choosing a set of data and choosing some elements and functions on the data.
3.2 Algebras of Booleans

We begin with what is, perhaps, the simplest and most useful data type: the Booleans.

3.2.1 Standard Booleans

The set $B = \{tt, ff\}$ of truth values or Booleans, where

$$tt \text{ represents } true \text{ and } ff \text{ represents } false,$$

has associated with it many useful functions or operations; usually they are called

*logical or propositional connectives.*

For example, we assume the reader recognises the propositional connectives

\[
\begin{align*}
\text{Not} & : B \rightarrow B \\
\text{And} & : B \times B \rightarrow B \\
\text{Or} & : B \times B \rightarrow B \\
\text{Xor} & : B \times B \rightarrow B \\
\text{Implies} & : B \times B \rightarrow B \\
\text{Equiv} & : B \times B \rightarrow B \\
\text{Nand} & : B \times B \rightarrow B \\
\text{Nor} & : B \times B \rightarrow B
\end{align*}
\]

which are normally defined by truth tables. Here are the truth tables for $\text{Not}$ and $\text{And}$:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\text{Not}(x)$</th>
<th>$\text{And}$</th>
<th>$tt$</th>
<th>$ff$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tt$</td>
<td>$ff$</td>
<td>$tt$</td>
<td>$tt$</td>
<td>$ff$</td>
</tr>
<tr>
<td>$ff$</td>
<td>$tt$</td>
<td>$ff$</td>
<td>$ff$</td>
<td>$ff$</td>
</tr>
</tbody>
</table>

It is also worth noting logical equivalence $\text{Equiv}$ which defines equality on Booleans:

<table>
<thead>
<tr>
<th>$\text{Equiv}$</th>
<th>$tt$</th>
<th>$ff$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tt$</td>
<td>$tt$</td>
<td>$ff$</td>
</tr>
<tr>
<td>$ff$</td>
<td>$ff$</td>
<td>$tt$</td>
</tr>
</tbody>
</table>

By simply choosing constants and a set of connectives we can make various algebras of Booleans such as

- $(B; tt, ff; \text{Not}, \text{And})$
- $(B; tt, ff; \text{Not}, \text{Or})$
- $(B; tt, ff; \text{And}, \text{Or})$
- $(B; tt, ff; \text{And}, \text{Implies})$
- $(B; tt, ff; \text{Nand})$
- $(B; tt, ff; \text{Not}, \text{And}, \text{Or}, \text{Implies}, \text{Equiv})$
- $(B; tt, ff; \text{Not}, \text{And}, \text{Xor}, \text{Nand}, \text{Nor})$

Most of the functions listed are binary operations, i.e., they have two arguments. There are interesting functions of more arguments.
3.2. ALGEBRAS OF BOOLEANS

Lemma (Counting) The number of functions on \(B\) with \(k\) arguments is \(2^{2^k}\). The number of algebras on \(B\) of the form

\[
(B; F_k)
\]

where \(F_k\) is a set of \(k\)-ary operations, is \(2^{2^k}\).

Proof. Given two sets \(X\) and \(Y\) with cardinalities \(|X| = n\) and \(|Y| = m\), we may deduce that the number of maps \(X \to Y\) is \(m^n\).

Now \(|B| = 2\) and take \(X = B^k\) and \(Y = B\). Then \(|X| = 2^k\) and \(|Y| = 2\) and we may deduce that the number of \(k\)-ary functions \(B^k \to B\) is \(2^{2^k}\).

Given the set \(X\) with \(|X| = n\), there are \(2^n\) possible subsets of \(X\). Thus, the number of sets \(F_k\) of \(k\)-ary operations on \(B\), with which we can make algebras, is \(2^{2^{2^k}}\). \(\square\)

Over the set \(B\), in the case \(n = 2\), there are 16 possible binary connectives and hence at least \(2^{16}\) algebras to be made from them.

All the usual Boolean connectives can be constructed by composition from the functions \(\text{Not}\) and \(\text{And}\). So, for theoretical purposes, the most useful algebra is the first that we listed, namely:

<table>
<thead>
<tr>
<th>algebra</th>
<th>Booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>(B)</td>
</tr>
<tr>
<td>constants</td>
<td>(tt, ff : \to B)</td>
</tr>
<tr>
<td>operations</td>
<td>(\text{Not} : B \to B)</td>
</tr>
<tr>
<td></td>
<td>(\text{And} : B \times B \to B)</td>
</tr>
</tbody>
</table>

Notice that all two elements of these algebras are listed as constants. It is sufficient to pick one, say \(tt\), since \(\text{Not}\) creates the other element, say \(ff = \text{Not}(tt)\).

There are several common notations for propositional connectives, all of them infix: for example,

<table>
<thead>
<tr>
<th>Logical</th>
<th>Boolean</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{And}(x, y))</td>
<td>(x \wedge y)</td>
<td>(x &amp; y)</td>
</tr>
<tr>
<td>(\text{Or}(x, y))</td>
<td>(x \lor y)</td>
<td>(x + y)</td>
</tr>
<tr>
<td>(\text{Implies}(x, y))</td>
<td>(x \Rightarrow y)</td>
<td>(x \rightarrow y)</td>
</tr>
<tr>
<td>(\text{Not}(x, y))</td>
<td>(\neg x)</td>
<td>(\overline{x})</td>
</tr>
</tbody>
</table>

3.2.2 Bits

Algebras of Booleans are used everywhere in Computer Science, notably to define and evaluate tests on data in control constructs. They are also used to define and design digital hardware.
In this case of hardware design it is customary to use the set

\[ \text{Bit} = \{1, 0\} \]

with

1 representing high voltage and 0 representing low voltage.

Many simple designs for digital circuits can be developed from

\( (\text{Bit}; 1, 0; \text{Not}_\text{Bit}, \text{And}_\text{Bit}) \).

This algebra is equivalent to the algebra on B

\( (B; \text{tt}, \text{ff}; \text{Not}_B, \text{And}_B) \)

where

1 corresponds with \( \text{tt} \) and 0 with \( \text{ff} \).

Then there is an obvious practical sense in which these choices result in equivalent algebras. For instance, the tables defining the operations correspond:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \text{Not}_\text{Bit}(x) )</th>
<th>( \text{And}_\text{Bit} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{Not}_B(x) )</th>
<th>( \text{And}_B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3.2.3 Equivalence of Booleans and Bits

It seems clear that Booleans and Bits are essentially the same data, though they are different and have different roles in computing. It is tempting to say that they are the same data in different notations or representations. Other notations for truth and falsity are

\[ \{T, F\} \quad \{t, f\} \quad \{true, false\} \]

in a choice or type faces,

\[ \{T, F\} \quad \{t, f\} \quad \{true, false\} \quad \{T, F\} \quad \{t, f\} \quad \{true, false\} \]

What is going on exactly?

**Equivalence of Algebras** Can we formulate in what precise theoretical sense these different algebras are equivalent?

The correspondence is made precise by a function

\[ \phi : B \to \text{Bit} \]

defined by

\[ \phi(tt) = 1 \quad \text{and} \quad \phi(ff) = 0 \]

that converts truth values to bits.
3.2. ALGEBRAS OF BOOLEANS

Conversely, we have a function

$$\psi : \text{Bit} \to \mathbf{B}$$

defined by

$$\psi(1) = tt \quad \text{and} \quad \psi(0) = ff$$

that converts bits to truth values.

We note that these functions express equivalence because

$$\psi(\phi(tt)) = tt \quad \text{and} \quad \psi(\phi(ff)) = ff$$
$$\phi(\psi(1)) = 1 \quad \text{and} \quad \phi(\psi(0)) = 0.$$ 

Clearly $\phi$ is a bijection or one-to-one correspondence with inverse $\psi$.

However, the equivalence also depends upon relating the operations on truth values and bits. For instance, for any $b, b_1, b_2 \in \mathbf{B},$

$$\phi(\text{Not}_B(b)) = \text{Not}_B(\phi(b))$$
$$\phi(\text{And}_B(b_1, b_2)) = \text{And}_B(\phi(b_1), \phi(b_2)).$$

And, conversely, for any $x, x_1, x_2 \in \text{Bit},$

$$\psi(\text{Not}_B(x)) = \text{Not}_B(\psi(x))$$
$$\psi(\text{And}_B(b_1, b_2)) = \text{And}_B(\psi(x_1), \psi(x_2)).$$

These two sets of equations show that the conversion mappings $\phi$ and $\psi$ preserve the operations of the algebras. Thus, equivalence can be made precise by a mapping $\phi$ and its inverse $\psi$, both of which preserve operations.

This precise sense of equivalence is a fundamental notion, called the

isomorphism

of algebras: two algebras are equivalent when they are isomorphic. We will study isomorphism later (in Chapter 7).

3.2.4 Three-Valued Booleans

We can usefully extend the design of these algebras of Booleans by introducing a special value $u$ to model an unknown truth value in calculations. Let

$$\mathbf{B}^u = \mathbf{B} \cup \{u\}.$$ 

How do we incorporate the “don’t know” element $u$ in the definition of our logical connectives?

Strict Interpretation

Consider the simple principle that if one does not know the truth value of an input to a connective then one does not know the truth value of the output of the connective. This leads us to extend the definition of the connectives on $\mathbf{B}$ to connectives on $\mathbf{B}^u$ as follows.
Let $f : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ be a binary connective on the Booleans. We define its extension $f^u : \mathbb{B}^u \times \mathbb{B}^u \rightarrow \mathbb{B}^u$ on known and unknown truth values by

$$f^u(x, y) = \begin{cases} f(x, y) & \text{if } x, y \in \mathbb{B}; \\ u & \text{if } x = u \text{ or } y = u. \end{cases}$$

By this method the truth tables of $\text{Not}$ and $\text{And}$ are extended to:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\text{Not}(x)$</th>
<th>$\text{And}$</th>
<th>$tt$</th>
<th>$ff$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tt$</td>
<td>$ff$</td>
<td>$tt$</td>
<td>$tt$</td>
<td>$ff$</td>
<td>$u$</td>
</tr>
<tr>
<td>$ff$</td>
<td>$tt$</td>
<td>$ff$</td>
<td>$ff$</td>
<td>$ff$</td>
<td>$u$</td>
</tr>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
</tr>
</tbody>
</table>

This extension of Boolean logic is called $\text{Kleene 3-value logic}$. Applying this principle leads to adaptations of all the algebras given earlier. For example,

<table>
<thead>
<tr>
<th>algebra</th>
<th>3-valued logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$\mathbb{B}^u$</td>
</tr>
<tr>
<td>constants</td>
<td>$tt, ff, u : \rightarrow \mathbb{B}^u$</td>
</tr>
<tr>
<td>operations</td>
<td>$\text{Not} : \mathbb{B}^u \rightarrow \mathbb{B}^u$</td>
</tr>
<tr>
<td></td>
<td>$\text{And} : \mathbb{B}^u \times \mathbb{B}^u \rightarrow \mathbb{B}^u$</td>
</tr>
</tbody>
</table>

This algebra for evaluating Booleans may be used to raise errors and exceptions in programming constructs depending on tests. However a case can be made for other methods of extending the operations.

**Lazy Interpretation**

Consider the table for $\text{And}$. A reasonable alternate decision for the definition of this particular connective is given by the observation that in processing the conjunction $\text{And}(x, y)$ of two tests, the falsity of $x$ alone can determine the output, i.e., for any $y = tt, ff$ or $u$,

$$\text{And}(ff, y) = ff$$

(and similarly for $x = tt, ff$ or $u$). This results in a connective, well known to implementors of programming languages, sometimes called parallel or concurrent and, or simply $\text{Cand}$. Here is its truth table:

<table>
<thead>
<tr>
<th>Cand</th>
<th>$tt$</th>
<th>$ff$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tt$</td>
<td>$tt$</td>
<td>$ff$</td>
<td>$u$</td>
</tr>
<tr>
<td>$ff$</td>
<td>$ff$</td>
<td>$ff$</td>
<td>$ff$</td>
</tr>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>$ff$</td>
<td>$u$</td>
</tr>
</tbody>
</table>

Unlike the concurrent calculation of a truth value, we note that a sequential calculation that requires both inputs to be known for the output to be known would be described by the first truth table.
3.3 Algebras of Natural Numbers

The data type of natural numbers is profoundly important for all aspects of the theory of computing.

3.3.1 Basic Arithmetic

We can define many useful algebras by selecting a set of functions on the set

\[ \mathbb{N} = \{0, 1, 2, \ldots \} \]

of natural numbers. Consider the following functions:

- **Succ**: \( \mathbb{N} \to \mathbb{N} \)
  \[ \text{Succ}(x) = x + 1 \]

- **Pred**: \( \mathbb{N} \to \mathbb{N} \)
  \[ \text{Pred}(x) = \begin{cases} x - 1 & \text{if } x \geq 1; \\ 0 & \text{otherwise}. \end{cases} \]

- **Add**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Add}(x, y) = x + y \]

- **Sub**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Sub}(x, y) = \begin{cases} x - y & \text{if } x \geq y; \\ 0 & \text{otherwise}. \end{cases} \]

- **Mult**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Mult}(x, y) = x \cdot y \]

- **Fact**: \( \mathbb{N} \to \mathbb{N} \)
  \[ \text{Fact}(x) = \begin{cases} x(x - 1) \ldots 2 \cdot 1 & \text{if } x \geq 1; \\ 1 & \text{if } x = 0. \end{cases} \]

- **Quot**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Quot}(x, y) = \begin{cases} \text{largest } k : ky \leq x & \text{if } y \neq 0; \\ 0 & \text{if } y = 0. \end{cases} \]

- **Mod**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Mod}(x, y) = x \mod y \]

- **Exp**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Exp}(x, y) = x^y \]

- **Log**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Log}(x, y) = \begin{cases} \text{largest } k : x^k \leq y & \text{if } x > 1 \text{ and } y \neq 0; \\ 0 & \text{if } x = 0 \text{ or } y = 0; \\ 1 & \text{if } x = 1. \end{cases} \]

- **Max**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Max}(x, y) = \begin{cases} x & \text{if } x \geq y; \\ y & \text{otherwise}. \end{cases} \]

- **Min**: \( \mathbb{N} \times \mathbb{N} \to \mathbb{N} \)
  \[ \text{Min}(x, y) = \begin{cases} x & \text{if } x \leq y; \\ y & \text{otherwise}. \end{cases} \]

Selected in various combinations, these functions make the following algebras, each of which
has interesting properties and applications:

\[(\mathbb{N}; \text{Zero}; \text{Succ})\]
\[(\mathbb{N}; \text{Zero}; \text{Pred})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Pred})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Add})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Pred}; \text{Add}, \text{Sub})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Add}, \text{Mult})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Pred}; \text{Add}; \text{Sub}; \text{Mult}; \text{Quot}; \text{Mod})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}, \text{Add}; \text{Mult}; \text{Exp})\]
\[(\mathbb{N}; \text{Zero}; \text{Succ}; \text{Pred}; \text{Add}; \text{Sub}; \text{Mult}; \text{Quot}; \text{Mod}; \text{Exp}; \text{Log})\]

The first algebra in the list is called the standard model of Presburger arithmetic. It is important in computation because many sets and functions on \(\mathbb{N}\) concern counting and the operation \(\text{Succ}\) creates all natural numbers from 0. We say \(\text{Succ}\) is a constructor. We display it:

### algebra

**Standard Model of Presburger Arithmetic**

**carriers** \(\mathbb{N}\)

**constants** \(\text{Zero} : \to \mathbb{N}\)

**operations** \(\text{Succ} : \mathbb{N} \to \mathbb{N}\)

In fact, every function on \(\mathbb{N}\) that can be defined by an algorithm, can be programmed using a while program over this algebra.

The algebra has some delightful algebraic properties we will meet in Chapter 7.

The algebra of natural numbers with successor, addition and multiplication (the sixth in the list above) is called the standard model of Peano arithmetic. It is important in logical reasoning because many sets and functions are definable by first-order logical languages over these three operations. We display it:

### algebra

**Standard Model of Peano Arithmetic**

**carriers** \(\mathbb{N}\)

**constants** \(\text{Zero} : \to \mathbb{N}\)

**operations** \(\text{Succ} : \mathbb{N} \to \mathbb{N}\)

\(\text{Add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\)

\(\text{Mult} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\)
3.3. ALGEBRAS OF NATURAL NUMBERS

Again, we note that the standard notations for their operations are all infix: for example,

\[
\begin{align*}
Succ(x) & = x + 1 \\
Add(x, y) & = x + y \\
Mult(x, y) & = x \cdot y
\end{align*}
\]

There will be many occasions when we use the familiar notation

\[(\mathbb{N}; 0; x + 1, x + y, x \cdot y).\]

3.3.2 Tests

So far, the algebras we have presented have contained one carrier or data set, i.e., they are single-sorted algebras. Now we consider algebras with several carrier or data sets, i.e., many-sorted algebras.

To the operations on natural numbers, we may add the characteristic functions of basic relations such as:

\[
\begin{align*}
Eq : \mathbb{N} \times \mathbb{N} & \to \mathbb{B} \\
Eq(x, y) & = \begin{cases} 
    \text{tt} & \text{if } x = y; \\
    \text{ff} & \text{otherwise.}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
Lt : \mathbb{N} \times \mathbb{N} & \to \mathbb{B} \\
Lt(x, y) & = \begin{cases} 
    \text{tt} & \text{if } x < y; \\
    \text{ff} & \text{otherwise.}
\end{cases}
\end{align*}
\]

And we may add the characteristic functions of interesting sets of numbers:

\[
\begin{align*}
Prime : \mathbb{N} & \to \mathbb{B} \\
Prime(x) & = \begin{cases} 
    \text{tt} & \text{if } x \text{ is prime;}
\end{cases} \\
& \text{ff otherwise.}
\end{align*}
\]

There are other operations that involve both Booleans and natural numbers, such as:

\[
\begin{align*}
If : \mathbb{B} \times \mathbb{N} \times \mathbb{N} & \to \mathbb{N} \\
If(b, x, y) & = \begin{cases} 
    x & \text{if } b = \text{tt}; \\
    y & \text{otherwise.}
\end{cases}
\end{align*}
\]

These, and any other tests on \( \mathbb{N} \), require us to add the Booleans to our algebras. For example, we can define the following algebra to form \textit{Peano Arithmetic with the Booleans}:
### Algebra

*Standard Model of Peano Arithmetic with the Booleans*

<table>
<thead>
<tr>
<th>carriers</th>
<th>N, B</th>
</tr>
</thead>
</table>
| constants | Zero : \( \rightarrow N \)  
\( tt, ff : \rightarrow B \) |
| operations | \( Succ : N \rightarrow N \)  
\( Add : N \times N \rightarrow N \)  
\( Mult : N \times N \rightarrow N \)  
\( Eq : N \times N \rightarrow B \)  
\( Lt : N \times N \rightarrow B \)  
\( If : B \times N \times N \rightarrow N \)  
\( And : B \times B \rightarrow B \)  
\( Not : B \rightarrow B \) |

Note, however, if we replace \( \{ tt, ff \} \) by \( \{ 1, 0 \} \subseteq N \) then tests could be seen as functions on \( N \) and the addition of a second sort could be avoided.

#### 3.3.3 Decimal versus Binary

In these examples of algebras of natural numbers, we have not considered the precise details of the representation of natural numbers. We have assumed a standard decimal representation, i.e., one with radix \( b = 10 \). The functions and algebras described above can be developed for any number representation system, for example the binary, octal and hexadecimal representations with radix \( b \) systems for \( b = 2, 8 \) or 16.

**Equivalence of Algebras**  
*The algebras obtained from using radix \( b \) are all implementations of natural number arithmetic: in what precise theoretical sense are these different algebras equivalent?*

Consider the equivalence of two radix \( b \) representations, namely

the algebra \( A_{10} \) of decimal arithmetic (with \( b = 10 \))

and

the algebra \( A_2 \) of binary arithmetic (with \( b = 2 \)).

First, we have a transformation of decimal to binary, which is a function

\[ \phi : A_{10} \rightarrow A_2. \]

Second, we have a transformation of binary to decimal, which is a function

\[ \psi : A_2 \rightarrow A_{10}. \]
The transformations must be compatible, namely: for decimals $x \in A_{10}$
\[
\psi(\phi(x)) = x
\]
and for binaries $y \in A_2$
\[
\phi(\psi(y)) = y.
\]
The functions must preserve the operations of the algebras.
Again, as in the case of Booleans and bits, this basic idea of equivalence of algebras is formalised by the notions of special functions that preserve operations called

*homomorphisms,*

and the homomorphisms that are also bijections, called

*isomorphisms.*

Further algebras of natural numbers arise from the attempt to make finite counting systems based on sets of the form
\[
\{0, 1, 2, \ldots, n - 1\}
\]
for some $n \geq 1$. We will discuss finite number systems shortly in Section 5.1.

### 3.3.4 Partial Functions

Here are some further types of functions that can be used in algebras of natural numbers.

An observation made easily at this early stage is that operations of algebras could be *partial functions.*

Several functions on $\mathbb{N}$, including $\text{Pred}$, $\text{Sub}$ and $\text{Log}$, have odd-looking definitions. For example,

\[
\text{Pred} : \mathbb{N} \to \mathbb{N}
\]

is defined
\[
\text{Pred}(x) = \begin{cases} 
  x - 1 & \text{if } x \geq 1; \\
  0 & \text{if } x = 0.
\end{cases}
\]
Now $0 - 1$ is not a natural number, so clearly another option is to leave $\text{Pred}(x)$ undefined for $x = 0$, rather than force it to be 0. We write
\[
\text{Pred}(x) = \begin{cases} 
  x - 1 & \text{if } x \geq 1; \\
  \uparrow & \text{if } x = 0
\end{cases}
\]
where $\uparrow$ indicates $\text{Pred}(0)$ does not have any value.

It is tempting to interpret the computation of $\text{Pred}(0)$ as an error. In order to define
\[
\text{Pred}(0) = \text{error}
\]
we have to expand $\mathbb{N}$ to
\[
\mathbb{N}^{\text{error}} = \mathbb{N} \cup \{\text{error}\}
\]
and define

$$Pred_{\text{error}} : \mathbb{N}_{\text{error}} \rightarrow \mathbb{N}^{\text{error}}$$

for $x \in \mathbb{N}_{\text{error}}$ by

$$Pred_{\text{error}}(x) = \begin{cases} 
  x - 1 & \text{if } x > 0; \\
  \text{error} & \text{if } x = 0; \\
  \text{error} & \text{if } x = \text{error}.
\end{cases}$$

In the case of predecessor, we can at least test if $x = 0$. This is not an option for all partial functions defined by programs.

We may allow functions to be partial and serve as operations in algebras. Such algebras are

**partial algebras.**

### 3.4 Algebras of Integers and Rationals

The extension of the natural numbers by negative numbers, to make the set $\mathbb{Z}$ of integers, facilitates calculation and leads to more interesting and useful algebras. The same remark is true of the extension of the integers to the set $\mathbb{Q}$ of rationals to accommodate division; the extension of the rationals to the set $\mathbb{R}$ of reals to accommodate measurements of irrational line segments; and the extension of the reals to the set $\mathbb{C}$ of complex numbers to accommodate the solution of polynomial equations. First let us look at the integers.

#### 3.4.1 Algebras of Integers

The subtraction $x - y$ of natural numbers $x$ and $y$ is unclear when $x < y$; the answer 0 is *ad hoc* and loses information:

$$1 - 10 = 0 \quad \text{and} \quad 1 - 1000 = 0$$

are different calculations that ought to have different consequences.

To calculate efficiently with subtraction we extend the set of natural numbers $\mathbb{N}$ to the set

$$\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$$

of *integers*.

Many functions on $\mathbb{N}$ extend to functions on $\mathbb{Z}$ in simple ways.

An important algebra of integers is

$$(\mathbb{Z}; \text{Zero, One}; \text{Add, Minus, Times})$$

which is displayed:
In the more familiar infix notation, we write the algebra

$$(\mathbb{Z}; 0, 1; +, -, .)$$

Here $-x$ is called the additive inverse operation and subtraction is derived by defining

$$x - y = x + (-y).$$

This algebra is called the ring of integers because it satisfies a certain set of algebraic laws; we will meet these laws in the next chapter.

Notice all integers can be created from the constants 0 and 1, by applying the operations of $+$ and $-$. These operations are constructors.

### 3.4.2 Algebras of Rationals

The division $x/y$ of integers $x$ and $y$ is not always defined. Take $22/7$: clearly, 7 divides 22, 3 times with remainder 1 and therefore $22/7$ is not an integer.

To calculate efficiently with division we extend the set $\mathbb{Z}$ of integers to the set

$$\mathbb{Q} = \left\{ \frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q \neq 0 \right\}$$

of rational numbers. Strictly speaking, we have defined a set of representations in which infinitely many different elements denote the same rational number. For example,

$$\frac{2}{1}, \frac{10}{5}, \frac{20}{10}, \frac{100}{50}, \ldots$$

or

$$\frac{4}{2}, \frac{6}{3}, \frac{8}{4}, \ldots$$

are the same. We define equality by

$$\frac{p_1}{q_1} = \frac{p_2}{q_2} \quad \text{if, and only if,} \quad p_1q_2 = p_2q_1.$$
The basic functions on the integers extend, for example:
\[
\begin{align*}
p_1 + p_2 & = \frac{p_1}{q_1} + \frac{p_2}{q_2} = \frac{p_1 \cdot q_2 + p_2 \cdot q_1}{q_1 \cdot q_2} \\
\frac{-p}{q} & = \frac{-p}{q} \\
p_1 - p_2 & = \frac{p_1}{q_1} - \frac{p_2}{q_2} = \frac{p_1 \cdot q_2 - p_2 \cdot q_1}{q_1 \cdot q_2}
\end{align*}
\]

Thus, the rationals are implemented in terms of the integers.

An important algebra of rationals is

\[(\mathbb{Q}; \text{Zero}, \text{One}; \text{Add}, \text{Minus}, \text{Times}, \text{Inverse})\]

which we display:

\begin{tabular}{|l|l|}
\hline
\textbf{algebra} & \textbf{Rationals} \\
\hline
\textbf{carriers} & \mathbb{Q} \\
\hline
\textbf{constants} & \text{Zero} : \rightarrow \mathbb{Q} \\
 & \text{One} : \rightarrow \mathbb{Q} \\
\hline
\textbf{operations} & \text{Add} : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q} \\
 & \text{Minus} : \mathbb{Q} \rightarrow \mathbb{Q} \\
 & \text{Times} : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q} \\
 & \text{Inverse} : \mathbb{Q} \rightarrow \mathbb{Q} \\
\hline
\end{tabular}

In the more familiar infix notation, we write the algebra

\[(\mathbb{Q}; 0, 1; +, -, \cdot, x^{-1})\]

This algebra is called the \textit{field} of rational numbers because it satisfies a certain set of algebraic laws. The function \(x^{-1}\) is called the multiplicative inverse operation. Note that \(x^{-1}\) is not defined for \(x = 0\), thus this operation is a partial function. Full division is derived by

\[x \cdot y^{-1}.\]

Notice all rational numbers can be created from the constants 0 and 1 by applying +, −, \(\cdot\), \(^{-1}\). Thus, these are constructor operations.

### 3.5 Algebras of Real Numbers

The data type of real numbers is the foundation upon which geometry, and the measurement and modelling of physical processes, is built. We will study these in depth later.
3.5.1 Measurements and real numbers

In making a measurement there is a ruler, scale or gauge that is based on a chosen unit and a fixed subdivision of the unit, for example: feet and inches, grams and milligram, hours and minutes, etc. Measurements are then approximated up to the nearest sub-unit. The numbers that record such measurements are the rational numbers. For example, 3 minutes 59 seconds is 239/60 seconds.

In ancient Greek mathematics, it was known that certain basic measurements could not be represented exactly by rational numbers. By Pythagoras’ Theorem, the hypotenuse of a right-angled triangle whose other two sides each measure 1 unit has a length of \( \sqrt{2} \) units. (See Figure 3.1.) But \( \sqrt{2} \) is not a rational number and so this hypotenuse cannot be measured exactly. An argument demonstrating that \( \sqrt{2} \) is not a rational number appears in Aristotle (Prior Analytics, Book 1 §23). Here is a detailed version:

**Theorem** \( \sqrt{2} \) is not a rational number.

**Proof** We use the method of *reductio ad absurdum*, or, as it is also known, proof by contradiction.

Suppose that \( \sqrt{2} \) was a rational number. This means that there exists some \( p, q \in \mathbb{Z} \), such that \( q \neq 0 \) and

\[
\left( \frac{p}{q} \right)^2 = 2. 
\]

(*)

By dividing out all the common factors of \( p \) and \( q \) we can assume, without any loss of generality, that \( \frac{p}{q} \) is a rational number in its lowest form, i.e., there is no integer, other than 1 or \(-1\), that divides both \( p \) and \( q \).

Now simplifying Equation (*)

\[
p^2 = 2q^2. 
\]

(**)

Thus, we know that \( p^2 \) is an even number, and this implies that \( p \) is an even number. If \( p \) is even, then there exists some \( r > 0 \) such that

\[
p = 2r.
\]

Substituting in Equation (**), we get

\[
(2r)^2 = 2q^2,
\]

\[
4r^2 = 2q^2.
\]

\[
2r^2 = q^2.
\]
Thus, $q^2$ is also an even number, and this implies $q$ is an even number.

We have deduced that both $p$ and $q$ are even and divisible by 2. This contradicts the fact that $p$ and $q$ have no common divisor, and the assumption that $p$ and $q$ exist. \hfill \Box

The real numbers are designed to allow a numerical quantity to be assigned to every point on an infinite line or continuum. Thus, a real number is used to measure and calculate exactly the sizes of any continuous line segments or quantities. There are a number of standard ways of defining the reals, all of which are based on the idea that

*real numbers can be approximated to any degree of accuracy by rational numbers.*

To define a real number we think of an infinite process of approximation that allows us to find a rational number as close to the exact quantity as desired. As we will see, in Chapter 8, these constructions or implementation methods for the real numbers (such as Cauchy sequences, Dedekind cuts or infinite decimals) can be proved to be equivalent.

The real numbers, like the natural numbers, are one of the truly fundamental data types. But unlike a natural number, a real number is an infinite datum and may not be representable exactly in computations. The approximations to real numbers used in computers must have finite representations or codings. In practice, there are gaps and separations between adjacent pairs of the real numbers that are represented. In fixed-point representations, the separation may be the same between all numbers whereas in floating-point representations the separation may vary and depend on the size of the adjacent values. Calculations with real numbers on a computer must take account of these approximations and unusual properties that they exhibit.

We will discuss the nature of real numbers in greater depth in Chapter 8. For the moment we are interested in making algebras of real numbers.

### 3.5.2 Algebras of Real Numbers

There are many interesting and useful algebraic operations on the set $\mathbb{R}$ of real numbers. Consider some of the functions that are associated with the set $\mathbb{R}$ of real numbers.

\[
\begin{align*}
+ : & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
- : & \mathbb{R} \to \mathbb{R} \\
.: & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
^{-1} : & \mathbb{R} \to \mathbb{R} \\
\sqrt{ } : & \mathbb{R} \to \mathbb{R} \\
\lVert \rVert : & \mathbb{R} \to \mathbb{R} \\
\exp : & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
\log : & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
\sin : & \mathbb{R} \to \mathbb{R} \\
\cos : & \mathbb{R} \to \mathbb{R} \\
\tan : & \mathbb{R} \to \mathbb{R}
\end{align*}
\]

Some simple algebras of real numbers can be obtained by selecting various subsets of functions and combining them with $\mathbb{R}$. For example:

\[
\begin{align*}
(\mathbb{R}; 0, 1; x + y, x, y, -x) \\
(\mathbb{R}; 0, 1; x + y, x, y, -x, x^{-1}) \\
(\mathbb{R}; 0, 1; x + y, x, y, -x, x^{-1}, \sqrt{x}, |x|)
\end{align*}
\]
3.5. **ALGEBRAS OF REAL NUMBERS**

We may add the Booleans and some basic tests to these algebras, for example

\[
= : \mathbb{R} \times \mathbb{R} \to \mathbb{B} \\
< : \mathbb{R} \times \mathbb{R} \to \mathbb{B}
\]

Collecting all these functions, and some famous constants, we may display an algebra thus:

<table>
<thead>
<tr>
<th>algebra</th>
<th>\textit{Reals}</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>(\mathbb{R}, \mathbb{B})</td>
</tr>
</tbody>
</table>
| constants| 0, 1, \(\pi, e\) : \(\to \mathbb{R}\) \[
t t, f f : \to \mathbb{B}
\]
| operations| \(+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}\) \\
| | \(- : \mathbb{R} \to \mathbb{R}\) \\
| | \(\times : \mathbb{R} \times \mathbb{R} \to \mathbb{R}\) \\
| | \(\neg : \mathbb{R} \to \mathbb{R}\) \\
| | \(\exp : \mathbb{R} \times \mathbb{R} \to \mathbb{R}\) \\
| | \(\log : \mathbb{R} \times \mathbb{R} \to \mathbb{R}\) \\
| | \(\sqrt{\cdot} : \mathbb{R} \to \mathbb{R}\) \\
| | \(\mid \mid : \mathbb{R} \to \mathbb{R}\) \\
| | \(\sin : \mathbb{R} \to \mathbb{R}\) \\
| | \(\cos : \mathbb{R} \to \mathbb{R}\) \\
| | \(\tan : \mathbb{R} \to \mathbb{R}\) \\
| | \(= : \mathbb{R} \times \mathbb{R} \to \mathbb{B}\) \\
| | \(< : \mathbb{R} \times \mathbb{R} \to \mathbb{B}\) |

Many more functions could be added, of course, but there is much to say about the operations included in the algebra above. Several are operations which do not return a value on all real number arguments

\[-^1, \log, \sqrt{\cdot}, \tan;\]

they are \textit{partial} functions rather than \textit{total} functions. So this example is a partial algebra.

Division \(-^1 : \mathbb{R} \to \mathbb{R}\) is not defined on the argument \(x = 0\). It can be defined as a total function by defining division on the set

\[
\mathbb{R} - \{0\}
\]

of non-zero real numbers:

\[-^1 : (\mathbb{R} - 0) \to \mathbb{R}.\]

Thus, the design of an algebra can be altered by restricting the domain of division to ensure it is total and adding its domain as a carrier \(\mathbb{R} - 0\) to the algebra. This is more tricky for some of the other partial functions.
Alternatively, we can require that an error flag is raised, i.e., we define a set
\[ R_{\text{error}} = R \cup \{ \text{error} \} \]
of real numbers with a distinguished element \( \text{error} \), and we set
\[ 0^{-1} = \text{error}. \]
Here, we must add the carrier \( R_{\text{error}} \), the error constant and redefine the operation
\[ ^{-1} : R \rightarrow R_{\text{error}} \]
along with its values under all the other operations. Similar problems and options arise with the other partial functions.

Many calculators and programming languages provide error information for partial operations. For any partial function \( f : R^n \rightarrow R \) with domain of definition
\[ \text{dom}(f) = \{ x \in R^n \mid f(x) \downarrow \} \]
we may add error messages by defining the new operation
\[ f_{\text{error}} : (R_{\text{error}})^n \rightarrow R_{\text{error}} \]
defined by:
\[ f_{\text{error}}(x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{if } x \in \text{dom}(f); \\ \text{error} & \text{if } x \notin \text{dom}(f); \\ \text{error} & \text{if } x_i = \text{error} \text{ for some } 1 \leq i \leq n. \end{cases} \]

However the study of algebras of these numbers has been dominated by the study of the algebras made from the basic operations of addition, subtraction, multiplication and, except in the case of the integers, division. These particular algebras have many properties in common and are best studied algebraically as examples of two general types of algebras called
\[ \textit{rings} \text{ and } \textit{fields}. \]

Other allied algebras made from matrices, polynomials and power series are also examples of rings and/or fields.

### 3.6 Data and the Analytical Engine

Recall Babbage's letter of 1835 about the Analytical Engine in Section 2.2. The conception of the engine was to perform the following operations, or any combination of them, to numbers consisting of 25 figures:

- add \( v_i \) to \( v_k \);
- subtract \( v_i \) from \( v_k \);
- multiply \( v_i \) by \( v_k \);
- divide \( v_i \) by \( v_k \);
- extract the square root of \( v_k \);
- reduce \( v_k \) to zero.
3.7. **Algebras of Strings**

The example is given of evaluating the formula \( p = \sqrt{(a + b)/cd} \).

The five operations of addition, subtraction, multiplication, division, and extraction of roots mentioned in the Letter to Quetelet are repeated in the introduction of Section III of his 1837 description of the Analytical Engine. There he notes that (i) further operations could be added to the mill and that (ii) the engine “possesses the power of treating the signs of the quantities on which it operates according to the rules of algebra and thus its use is greatly extended.”

Let us make an idealised model of the data type. Let \( R_k \) be the set of all positive and negative real numbers that can be represented by \( k \) figures or digits using the decimal notation. Clearly, there are bounds \( \pm M_k \) on the range of numbers allowed and

\[
R_k \subseteq [-M_k, M_k];
\]

for example, \( M_k = 10^k - 1 \). The details of the set \( R_k \) of positive and negative approximations to reals depend upon the arrangements for number representations and are of great importance for the work on the machine.

We have an algebra:

<table>
<thead>
<tr>
<th>algebra</th>
<th>Analytical Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>( R_k )</td>
</tr>
<tr>
<td>constants</td>
<td>( 0, 1, +M_k, -M_k ) : ( \rightarrow R_k )</td>
</tr>
<tr>
<td>operations</td>
<td>( + : R_k \times R_k \rightarrow R_k )</td>
</tr>
<tr>
<td></td>
<td>( - : R_k \times R_k \rightarrow R_k )</td>
</tr>
<tr>
<td></td>
<td>( \times : R_k \times R_k \rightarrow R_k )</td>
</tr>
<tr>
<td></td>
<td>( / : R_k \times R_k \rightarrow R_k )</td>
</tr>
<tr>
<td></td>
<td>( \sqrt{\cdot} : R_k \rightarrow R_k )</td>
</tr>
</tbody>
</table>

The discussion of data in works on the Analytical Engine is not limited to numbers, operations on numbers, and the signs and sizes of numbers. Other aspects of data are present, such as the coding of data, functions, addresses and symbols on punched cards. Further ideas arise concerning algebraic notions such as computing with imaginary numbers and manipulating symbolic expressions. Some of the discussions appear in Ada Lovelace’s notes, and in Babbage’s notebooks, but are short and speculative.

### 3.7  Algebras of Strings

Syntax is constructed by joining together symbols from some alphabet. A string is a sequence of symbols. Here, we consider how we can build algebras of strings. These are essential building blocks that we shall use later on to construct more complex models of syntax for programming languages.
3.7.1 Constructing Strings

Let $T$ be some non-empty *alphabet* or set of symbols. The nature of this set $T$ is not relevant when we consider how we can form and manipulate strings over $T$. So $T$ could equally well be the digits 0 to 9,

$$T = \{0, 1, 2, \ldots, 9\}$$

or the letters of the English language,

$$T = \{a, b, c, \ldots, z\}$$

or digits and letters

$$T = \{a, b, c, \ldots, z, 0, 1, 2, \ldots, 9\}$$

or digits, letters and punctuation symbols

$$T = \text{Set of ASCII characters.}$$

We focus here on the set $T^*$ of strings over $T$.

The simplest string we can have is the empty string

$$\epsilon : \rightarrow T^*.$$ 

Alternatively, we can build strings from symbols by repeatedly using an operation

$$\text{Prefix} : T \times T^* \rightarrow T^*$$

to add a symbol to the start of a string.

For example, for $T = \{0, 1, 2, \ldots, 9\}$:

$$\text{Prefix}(2, \text{Prefix}(0, \text{Prefix}(0, \text{Prefix}(1, \epsilon)))) = 2001$$

These two operations are sufficient to be able to build up the set $T^*$ of all possible strings over $T$. The operations $\epsilon$ and $\text{Prefix}$ are constructors. We model this with an algebra

<table>
<thead>
<tr>
<th>algebra</th>
<th>Basic Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$T, T^*$</td>
</tr>
<tr>
<td>constants</td>
<td>$\epsilon \rightarrow T^*$</td>
</tr>
<tr>
<td>operations</td>
<td>$\text{Prefix} : T \times T^* \rightarrow T^*$</td>
</tr>
</tbody>
</table>

3.7.2 Manipulating Strings

Once we have a set $T^*$ of strings, there are many useful operations we can define on it.

We can compose, or *concatenate*, two strings together with an operation

$$\text{Concat} : T^* \times T^* \rightarrow T^*.$$
3.7. ALGEBRAS OF STRINGS

For example,

\[ \text{Concat}(ab, ba) = abba. \]

Instead of always placing a symbol at the start of a string, we can insert a symbol into a specified position with a function

\[ \text{Insert} : T \times \mathbb{N} \times T^* \rightarrow T^*. \]

To ensure that this is a total function, we can place the symbol at the end of the string if the string is not sufficiently long enough for the specified position. For example:

\[ \text{Insert}(r, 3, \text{sting}) = \text{string} \quad \text{Insert}(r, 7, \text{sting}) = \text{stingr} \]

The inverse of \( \text{Insert} \) is to project out a symbol from a string:

\[ \text{Proj} : \mathbb{N} \times T^* \rightarrow (T \cup \{\text{error}\}). \]

Again, this is a potentially hazardous operation; if we choose a non-existent position, we return a value \( \text{error} \) to ensure that we have a total function, whilst flagging the problem. We cannot hide this problem by returning a value that is in the set \( T \), nor can we return the empty string as \( \varepsilon \notin T \).

For example,

\[ \text{Proj}(4, 2001) = 1 \quad \text{Proj}(5, 2001) = \text{error}. \]

We can define an operation

\[ | : T^* \rightarrow \mathbb{N} \]

to calculate the length of a string. For example,

\[ |\varepsilon| = 0 \quad |yngys| = 4. \]

We can also define tests on strings, for example, we can test whether two strings are the same with an operation

\[ \text{Eq} : T^* \times T^* \rightarrow \mathbb{B}. \]

Note that to define this operation, we would need to suppose that the alphabet \( T \) also has a test for equality on its symbols. Given such a test, we can also define a test

\[ \text{In} : T \times T^* \rightarrow \mathbb{B} \]

to check whether a symbol is present in a string.

We can design different algebras of strings by choosing some of these operations on strings. For example,

\[ (T, T^*; \varepsilon; \text{Prefix}) \]
\[ (T^*; \varepsilon; \text{Concat}) \]
\[ (T, T^*; \varepsilon; \text{Prefix}, \text{Concat}) \]
\[ (T, T^*, \mathbb{N}; \varepsilon; \text{Prefix}, | |) \]
\[ (T^*, \mathbb{B}; \text{Eq}) \]
\[ (T, T^*, \mathbb{B}, \mathbb{N}; \varepsilon, tt, ff, 0; \text{Prefix}, \text{In}, | |, \text{And}, \text{Succ}) \]

We display the fourth of these algebras:
algebra \textit{Strings with Length}

 carriers \( T, T^*, \mathbb{N} \)

 constants \( \epsilon : \to T^* \)
                                \( \text{Zero} : \to \mathbb{N} \)

 operations \( \text{Prefix} : T \times T^* \to T^* \)
                              \( \| : T^* \to \mathbb{N} \)
Exercises for Chapter 3

1. List all the 16 binary truth connectives

\[ f : \{0, 1\} \to \{0, 1\}. \]

Show that every connective can be expressed in terms of \textit{And} and \textit{Not}. Can every connective be expressed in terms of:

a. \textit{Or} and \textit{Not}?
b. \textit{Implies} and \textit{Not}?

2. Let \( X \) and \( Y \) be non-empty sets with \(|X| = n\) and \(|Y| = m\). Prove that the number of functions \( X \to Y \) is \( m^n \) and the number of subsets of \( X \) is \( 2^n \).

Using the provisional definition of an algebra from Section 3.1, derive a formula for the number of algebras containing only the set \( X \) with \( n \) elements and \( k \)-ary operations.

How many two-sorted algebras contain both \( X \) and \( Y \) and have binary operations?

3. Give truth tables for the connectives \textit{Or} and \textit{Implies} on \( \{tt, ff, u\} \) with the lazy interpretation. What identities hold between the connectives? Is it the case that

\[ \text{Implies}(x, y) = \text{Not}(\text{And}(x, \text{Not}(y)))? \]

4. It is possible to represent pairs of numbers by single numbers using \textit{pairing functions} which are a bijection

\[ \text{Pair} : N \times N \to N, \]

together with its inverse

\[ \text{Unpair} : N \to (N \times N) \]

with coordinate functions \( \text{Unpair}_1 : N \to N \) and \( \text{Unpair}_2 : N \to N \), such that

\[ \text{Unpair}(x) = (\text{Unpair}_1(x), \text{Unpair}_2(x)) \]

for \( x \in N \). Prove that,

\[ \text{Pair}(x, y) = 2^x(2^y + 1) - 1 \]

is a bijection. What is its inverse?

Using pairing functions show how to define functions

\[ n\text{-pair} : N^n \to N \]

that represent \( n \)-tuples of numbers as single numbers for all \( n = 2, 3, \ldots \).

5. Extend the definitions of the functions \textit{Quot} and \textit{Mod}, and \textit{Max} and \textit{Min}, on \( N \) to \( Z \). Can \textit{Fact}, \textit{Exp} and \textit{Log} be extended?
6. Some functions on \( \mathbb{N} \) are defined from functions on the set \( \mathbb{Q} \) of rationals or the set \( \mathbb{R} \) of real numbers. For example, consider the function:

\[
\text{NaturalsSqrt} : \mathbb{N} \rightarrow \mathbb{N} \\
\text{NaturalsSqrt}(x) = \lfloor \sqrt{x} \rfloor.
\]

where

\[
\lfloor \cdot \rfloor : \mathbb{R}^{+} \rightarrow \mathbb{N} \\
\lfloor x \rfloor = (\text{largest } k \in \mathbb{N})(k \leq x)
\]

is the floor function which rounds down a real number to the nearest integer. Write a while program on \( \mathbb{N} \) to compute NaturalsSqrt.

7. Define four algebras over the set \( \mathbb{Q} \) of rational numbers.

8. Define the set of integers as the product set

\[
\mathbb{Z} = \{+, -\} \times \mathbb{N}.
\]

Thus, an integer is represented by a pair \((+, n)\) or \((-n)\), for a natural number \(n \in \mathbb{N}\). Write out the definition of addition, additive inverse, and multiplication for this representation of the integers as pairs.

Define a function

\[
\phi : \mathbb{N} \rightarrow \mathbb{Z}
\]

by

\[
\phi(n) = (+, n).
\]

Show that for any \(m, n \in \mathbb{N}\):

\[
\phi(m + n) = \phi(m) + \phi(n) \\
\phi(-n) = -\phi(n) \\
\phi(m.n) = \phi(m) \cdot \phi(n)
\]

9. Represent the set \( \mathbb{Q} \) of rational numbers as a subset of \( \mathbb{Z} \times \mathbb{Z} \), namely

\[
\mathbb{Q} = \{(p, q) \mid p, q \in \mathbb{Z} \text{ and } q \neq 0\}
\]

and write out the definitions of equality, addition, additive inverse, multiplication and multiplicative inverse in terms of this representation of the rational numbers.

10. Show that the operations of addition, additive inverse, and multiplication on \( \mathbb{Q} \) “simulate” the corresponding operations of

\[
\mathbb{Z} = \{(p, q) \in \mathbb{Q} \mid q = 1\}.
\]

in the following sense: Define a function \(\phi : \mathbb{Z} \rightarrow \mathbb{Q}\) by \(\phi(x) = (x, 1)\) for all \(x \in \mathbb{Z}\). Show that for all \(x, y \in \mathbb{Z}\),

\[
\phi(x + y) = \phi(x) + \phi(y) \\
\phi(-x) = -\phi(x) \\
\phi(x \cdot y) = \phi(x) \cdot \phi(y)
\]
11. Show that $\sqrt{3}$ is not a rational number.

12. Show that $\sqrt{2} + \sqrt{3}$ is not a rational number.

13. Which of the following functions on the set $\mathbb{R}$ of real numbers are partial (i.e., they are not always defined) and what are their domains of definition?

   a. $x$;
   b. $\frac{1}{x}$;
   c. $x^2 - 2x + 1$;
   d. $\frac{1}{x^2 - 2x + 1}$;
   e. $x^2 + 1$;
   f. $\frac{x^2 - 1}{x^2 + 1}$;
   g. $\sin(x)$;
   h. $\frac{1}{\sin(x)}$;
   i. $\cos(x)$;
   j. $\frac{1}{\cos(x)}$;
   k. $\tan(x)$;
   l. $\frac{1}{\tan(x)}$;
   m. $e^x$;
   n. $e^{-x}$;
   o. $\sqrt{x}$; and
   p. $\frac{x}{\sqrt{x}}$.

14. In each of these algebras of real numbers, what set of numbers is constructed by applying the operations to the constants:

   a. $(\mathbb{R}; 0, 1; +)$;
   b. $(\mathbb{R}; 0, 1; +, -)$;
   c. $(\mathbb{R}; 0, 1; +, -, -^{-1})$; and
   d. $(\mathbb{R}; 0, 1; +, -, -^{-1})$?

   Does there exist an algebra of real numbers, with a finite set of constants and operations that can construct all real numbers?

15. Define the following functions on strings:

   a. the concatenation function

   $$\text{Concat} : T^* \times T^* \rightarrow T^*$$

   by using the function $\text{Prefix}$;
b. the length function

\[ |\cdot| : T^* \rightarrow \mathbb{N} \]

by using the function \( \text{Prefix} \);

c. the equality test

\[ E_q : T^* \times T^* \rightarrow \mathbb{B} \]

on strings by using the functions \( \text{Prefix} \) and the equality test \( E_{qr} : T \times T \rightarrow \mathbb{B} \) on symbols; and

d. the test

\[ I_n : T \times T^* \rightarrow \mathbb{B} \]

for membership of a string by using the functions \( \text{Prefix} \) and the equality test \( E_{qr} : T \times T \rightarrow \mathbb{B} \) on symbols.

16. Define functions

\[ \text{Left} : T^* \times \mathbb{N} \rightarrow T^* \]

\[ \text{Right} : T^* \times \mathbb{N} \rightarrow T^* \]

on strings such that

\[ \text{Left}(w, i) \]

gives the string consisting of the first \( i \) characters of \( w \) and

\[ \text{Right}(w, i) \]

gives the string consisting of the characters of \( w \) from position \( i + 1 \) onwards.

Show that for all strings \( w \in T^* \), and splitting points \( i \in \mathbb{N} \),

\[ \text{Concat}(\text{Left}(i, w), \text{Right}(i, w)) = w. \]

17. Define a test

\[ \text{IsRepeated} : T \times T^* \rightarrow \mathbb{B} \]

to check if a given symbol appears more than once in a string.

Using the function \( \text{IsRepeated} \), define a test

\[ \text{IsDistinct} : T^* \rightarrow \mathbb{B} \]

that returns true if none of the symbols in a given string appear more than once.

18. A complex number \( z \) is an expression of the form \( x + iy \) where \( x \) and \( y \) are real numbers and \( i \) denotes \( \sqrt{-1} \). Let \( \mathbb{C} \) be the set of complex numbers. Consider the representation of the set \( \mathbb{C} \) by all ordered pairs

\[ (a, b) \in \mathbb{R}^2 \]

of real numbers. Define the operations of addition, subtraction, multiplication and division on this representation \( \mathbb{R}^2 \) of the complex numbers.
### 3.7. **ALGEBRAS OF STRINGS**

19. Define an algebra over the complex numbers. For example, consider the following operations on the complex numbers

\[
\begin{align*}
\text{mod}(x + iy) &= \sqrt{x^2 + y^2} \\
\text{re}(x + iy) &= x \\
\text{im}(x + iy) &= y \\
\text{conj}(x + iy) &= x - iy.
\end{align*}
\]

20. Design a two-sorted algebra containing the sets

\[
\text{Bit} = \{0, 1\} \quad \text{and} \quad n\text{-Word} = \text{Bit}^n
\]

...together with constants and operations appropriate for constructing and processing computer words.
Assignment for Chapter 3
Chapter 4

Interfaces, Implementations, and Algebras

We are developing the idea of a data type in several stages. In the first stage, in Chapter 3, we observed that a data type does not consist of data only, but of

\[ \text{data together with operations and tests.} \]

We made a mathematical model of this programming concept using the idea of an algebra, which we defined, provisionally, as a family of sets combined together with families of elements and functions. We saw plenty of examples of data types and algebras in this sense. In this chapter we reach the second stage. We will add two vital components to the concept of data type and algebra, namely, the programming idea of

\[ \text{interface} \]

and the corresponding mathematical idea of

\[ \text{signature} \]

to model it.

Now, a data type is a collection of data, operations and tests in a specific representation. Representations of data are many and varied. There can be minor or major differences in equivalent data representations. For example, it seems hardly to matter whether the Booleans are represented by \{tt, ff\} or \{T, F\}, or some other two-element set; however, representing the natural numbers in base \( b = 10 \) or \( b = 2 \), or some other base, has significant consequences. If the representations of the data types are not equivalent then we can expect the differences to be important at some point in their use.

In the second stage of developing our idea of a data type, we want to separate the form of the data type from the details of the representations used in its implementation. To do this, we revise our idea of a data type by introducing these two aspects,

\[ \text{Data Type} = \text{Interface} + \text{Implementation}. \]

The interface is defined as a declaration containing names for the data, constants, operations and tests: it specifies the form of the data type, including the types of data, operations and tests. The implementation is defined in terms of specific representations for the data, operations
and tests in the data type. The names of the data and the operations can be fixed by declaring an interface for a data type, but there will always be considerable variation in the details of how the data and operations are implemented.

Now this distinction between form and representation also makes perfect sense for the mathematical idea of an algebra. We can also revise the provisional definition by adding a declaration of names for the sets, constants, and operations. This collection of names is called a

signature.

It acts simply an interface to the sets and functions of the algebra. The sets and functions that interpret these names constitute an algebra in the earlier, provisional sense of Chapter 3. A data type is still modelled by a many sorted algebra, but now we have:

\[
\text{Algebra} = \text{Signature} + \text{Interpretation.}
\]

It is the distinction between the signature and the interpretation that allows us to model explicitly, and begin to analyse mathematically, the endless variation of major and minor details associated with data representations within software. In this analysis problems arise that lead to further stages in the development of the notion of data type.

We will also introduce some simple ideas about building new data types and algebras. We will describe in general terms

*sharing, adding, and removing data sets and operations*

for signatures and algebras. These ideas correspond with the technical concepts of

*subalgebra, expansion and reduct,*

respectively.

The idea of an expansion is one formalisation of the idea that a

*data type imports or inherits another data type.*

The idea of a reduct is one formalisation of the idea that a

*data type encapsulates or hides part of another data type.*

Thus, early in our theory, we see data types and algebras acquire architecture.

In Section 4.1, we define the notion of signature. Section 4.2 is filled with examples of signatures. In Section 4.3, we give the new definition of algebra based upon signatures. Sections 4.3.2 and ?? are filled with examples of algebras. In Section ??, we introduce the ideas of constructions for generating data inside an algebra. In Section 4.6.2, we discuss subalgebras, expansions and reducts. In Section 4.7 we extend signatures and algebras to allow imports. In Section 4.4, we use the new definition in looking back at some of the special algebras we met in the last chapter.
4.1  Formal Definition of a Signature

The provisional definition of an algebra we gave in Section 3.1 must be refined to meet the needs of both the theoretical analysis and practical application of data types. What is missing from the provisional definition is an explicit, independent and complete notation for naming and uniquely identifying each set, constant and operation in an algebra. A collection of names for the sets of data, constants and operations of an algebra is called a

signature.

Thus, the proper definition of an algebra consists of both

**Syntax**  the names for the kinds of data, specific data elements and operations on data listed in the signature, and

**Semantics**  the sets, elements and functions that are assigned to interpret the names.

The importance of the concept of signature in the development and application of the theory of data cannot be underestimated:

*A signature is the interface between an algebra of data and its users.*

Therefore, in programming terms,

*A signature is a model of the interface between a data type and its users.*

We begin by considering the idea using an example.

4.1.1  Examples of Signatures for Algebras

Reconsider the algebra

\[ A = (\mathbb{N}; \text{Zero}; \text{Succ}, \text{Add}, \text{Mult}) \]

of natural numbers that we introduced in Section 3.3.1; it was called the *Standard Model of Peano Arithmetic*.

To make a signature we simply choose *names* for all the components of this algebra and declare them. To keep close to the notation we used for \( A \), we choose these names:

```plaintext
signature  Peano Arithmetic
sorts      nat
constants  zero : → nat
operations succ : nat → nat
            add : nat × nat → nat
            mult : nat × nat → nat
endsig
```
The name of the signature is the identifier *Peano Arithmetic*. We will think of the algebra \( A \) as the result of assigning to these names some specific sets, elements and functions, a process we call

> **interpreting the signature.**

Thus,

- sort *nat* is assigned the set \( \mathbb{N} \),
- constant *zero* is assigned the element \( \text{Zero} \in \mathbb{N} \),
- and operation symbols *succ*, *add* and *mult* are assigned the functions *Succ*, *Add* and *Mult* on \( \mathbb{N} \).

On assigning the sets and functions to the names in \( \Sigma_{\text{Peano Arithmetic}} \), we get the algebra

\[
A = (\mathbb{N}; \text{Zero}; \text{Succ}, \text{Add}, \text{Mult})
\]

What is the point? For familiar algebras with standard notations, the signature may seem pedantic or quite unnecessary. However, a crucial point is that several *other* algebras *could* be used to interpret these names. For example,

1. an equivalent algebra based on binary rather than decimal notation, or
2. an inequivalent algebra based on a finite set of numbers, or even
3. an algebra having nothing to do with natural numbers.

To a user of the data type, different implementations of \( \Sigma_{\text{Peano Arithmetic}} \) are likely to matter. This possible diversity is made explicit by using the signature as a model of an interface to data.

### 4.1.2 General Definition

We now give a formal definition of a signature. The definition is designed to capture a general concept in a mathematically satisfying way.

**Definition (Sorts and Types)** Let \( S \) be a non-empty set whose elements will be used to name the sets of data in an algebra. These elements

\[
\ldots, s, \ldots
\]

of \( S \) we call *sorts*.

If \( S \) is a non-empty set of sorts naming sets, then an expression

\[
s_1 \times \cdots \times s_n \quad \text{or} \quad s_1 \cdots s_n
\]

for \( n \geq 0 \) can be used to name a Cartesian product of the sets named by \( s_1, \ldots, s_n \). This includes the empty expression \( \lambda \) (when \( n = 0 \)). These expressions we call *product types*, and the number \( n \) of sorts is called the *arity* of the product type. Let \( S^* \) be the set of all such expressions, and \( S^+ = S^* - \{\lambda\} \).
4.1. FORMAL DEFINITION OF A SIGNATURE

We can also name the type of a function on the sets by pairs

\[(s_1 \times \cdots \times s_n, s)\] or \[(s_1 \cdots s_n, s)\]

which we will write

\[s_1 \times \cdots \times s_n \rightarrow s\]

These expressions we call operation or function types, and the number \(n\) of arguments is called the arity of the function type.

**Example** Consider these notations for computing with natural numbers. First, there is the set of sorts. There is just one type of data, so let

\[S = \{\text{nat}\} .\]

The sets of tuples of natural numbers are named by the product types

\[\lambda, \ \text{nat}, \ \text{nat} \times \text{nat}, \ \text{nat} \times \text{nat} \times \text{nat}, \ \ldots, \ \text{nat} \times \cdots \times \text{nat}, \ \ldots\]

\[\text{\(n\) times}\]

or, simply,

\[\lambda, \ \text{nat}, \ \text{nat}^2, \ \text{nat}^3, \ \ldots, \ \text{nat}^n, \ \ldots\]

The operation types are

\[\lambda \rightarrow \text{nat}, \ \text{nat} \rightarrow \text{nat}, \ \text{nat}^2 \rightarrow \text{nat}, \ \ldots, \ \text{nat}^n \rightarrow \text{nat}, \ \ldots\]

and these describe the type of elements of nat, unary operations, binary operations, \ldots, and \(n\)-ary operations, respectively.

Note that there is precisely one product and operation type of arity \(n\).

With this notation for sorts and types, we can define a signature.

**Definition (Many-Sorted Signature)** A signature consists of:

(i) a name \(\text{Name}\) for the signature which we call its identifier;

(ii) a non-empty set \(S\), the elements of which we call sorts; and

(iii) an \(S^* \times S\)-indexed family

\[<\Sigma_{w,s} \mid w \in S^*, s \in S>\]

of sets, the elements of which we call constant and operation symbols:

**Constant Symbols** For the empty word \(\lambda \in S^*\) and any sort \(s \in S\), each element

\[c \in \Sigma_{\lambda,s}\]

is called a constant symbol of sort \(s\). We also write

\[c : \rightarrow s.\]

**Operation Symbols** For each non-empty word \(w = s_1 \cdots s_n \in S^+\) and any sort \(s \in S\), each element

\[f \in \Sigma_{w,s}\]

is called an operation symbol with domain type \(w\), range type \(s\) and arity \(n\). We also write

\[f : s_1 \times \cdots \times s_n \rightarrow s.\]
Thus, the signature is the triple
\[
\Sigma_{Name} = (Name, S, \langle \Sigma_w, s \mid w \in S^*, s \in S \rangle).
\]

**Example** Consider the signature for Peano Arithmetic used in Section 4.1.1 As a tuple of sorts, constants and operations, a signature has the form
\[
\Sigma_{Name} = (Name; S; \langle \Sigma_w, s \mid w \in S^*, s \in S \rangle).
\]
In this example, the signature is
\[
\Sigma_{\text{PeanoArithmetic}}
\]
since the name is *PeanoArithmetic* and the sort set is
\[
S = \{\text{nat}\},
\]
the set of constant names is
\[
\Sigma_{\lambda, \text{nat}} = \{\text{zero}\},
\]
the sets of operation names are
\[
\begin{align*}
\Sigma_{\text{nat}, \text{nat}} &= \{\text{suc}\} \\
\Sigma_{\text{nat}, \text{nat}, \text{nat}} &= \{\text{add, mult}\} \\
\Sigma_{\text{nat}^n, \text{nat}} &= \emptyset
\end{align*}
\]
for \(n > 2\).

Recall our observations on notation in the Guide to the Reader. The definition of a signature as a tuple is an example of a definition designed for mathematical analysis because it
\[(i)\] captures the concept and its components both abstractly and precisely, and
\[(ii)\] is compact and efficient when reasoning about the concept in general.

The definition is essentially an expression or encoding of the idea in the language of set theory. However, as a notation for working with examples, the definition can be too concise and formal. So we use another notation to display the definition.

We also display a signature \(\Sigma_{Name}\) in the following manner:

<table>
<thead>
<tr>
<th><strong>signature</strong></th>
<th><strong>Name</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sorts</strong></td>
<td>(\ldots, s, \ldots)</td>
</tr>
<tr>
<td><strong>constants</strong></td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(c: \to s)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td><strong>operations</strong></td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(f: s_1 \times s_2 \times \cdots \times s_n \to s)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td><strong>endsig</strong></td>
<td></td>
</tr>
</tbody>
</table>
Our signatures will declare \textit{finitely} many sorts, constants and operations, so we make a restriction.

\textbf{Definition (Finite Signatures)} A signature is \textit{finite} if:

(i) there are finitely many sorts, i.e., $S$ is finite; and

(ii) there are finitely many constants and operations, i.e., $\Sigma_{w,s} \neq \emptyset$ for at most finitely many $w, s$.

Henceforth we make the assumption:

\textbf{Assumption} We will assume that each signature $\Sigma$ is finite.

4.2 Examples of Signatures

4.2.1 Examples of Signatures for Basic Data Types

We will illustrate the general idea of signature by giving signatures for some of the algebras of Booleans and numbers we met in Chapter 3. For particular algebras, a number of choices of notation for the sorts, constants and operations in a signature are always available. Our examples will be displayed rather than presented as tuples.

\textbf{Booleans} From among the many algebras of Booleans, here is a signature for a standard set of Boolean operations:

\begin{verbatim}
signature  Boolean
sorts      Bool
constants  true, false :\rightarrow Bool
operations and : Bool \times Bool \rightarrow Bool
              not : Bool \rightarrow Bool
endsig
\end{verbatim}

The operations are prefix notations. Notice that we have shortened the declaration of constants by making a list rather than the two declarations, i.e., $true : \rightarrow Bool$ and $false : \rightarrow Bool$.

\textbf{Naturals} We met a signature $\Sigma_{\text{Peano Arithmetic}}$ for the Peano Arithmetic in Section 4.1.1; it used prefix notation and was based on the algebra in Section 3.3. If we wanted to use the standard infix notation for successor, addition and multiplication then we can choose the following
signature *Naturals*:

```plaintext
signature  *Naturals*
sorts      *nat*
constants  0 :→ *nat*
operations _+ 1:  *nat* → *nat*
           _+ _:  *nat* × *nat* → *nat*
           _-_:  *nat* × *nat* → *nat*
endsig
```

**Integers**  In Section 3.4.1, we introduced an algebra of integers using a prefix notation for addition, subtraction and multiplication. Here is a signature *Integers*:

```plaintext
signature  *Integers*
sorts      *int*
constants  zero, one :→ *int*
operations add:  *int* × *int* → *int*
               minus:  *int* → *int*
               times:  *int* × *int* → *int*
endsig
```

Next, we give a signature *IntegersInfix* for the integers that uses the standard infix notation for addition, subtraction and multiplication:

```plaintext
signature  *IntegersInfix*
sorts      *int*
constants  0, 1 :→ *int*
operations _+_:  *int* × *int* → *int*
            _-_:  *int* → *int*
            _-_:  *int* × *int* → *int*
endsig
```
4.2. EXAMPLES OF SIGNATURES

Reals  In Section 3.5.2, we introduced an algebra of real numbers with lots of basic operations. Here is a signature Reals containing some infix and some postfix notations.

<table>
<thead>
<tr>
<th>signature</th>
<th>Reals</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>Bool</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1, $\pi \rightarrow$ real</td>
</tr>
<tr>
<td>operations</td>
<td>$_ + _ : real \times real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$_ - _ : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$_ _ : real \times real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$_ ^{-1} : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$\sqrt{_} : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$\sin : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$\cos : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$\tan : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$\exp : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$\log : real \rightarrow real$</td>
</tr>
<tr>
<td></td>
<td>$_ = : real \times real \rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>$_ &lt; : real \times real \rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>$_ _ : Bool \times Bool \rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>$_ _ _ : Bool \rightarrow Bool$</td>
</tr>
</tbody>
</table>

endsig

Notice that some of these names are for operations that are partial if applied to all real numbers. This is not visible in the notation here. Should it be? Perhaps we should refine the signature and introduce special names for sorts of non-zero real numbers, e.g.,

$\_ ^{-1} : real_{\neq 0} \rightarrow real$

and positive real numbers, e.g.,

$\sqrt{\_} : real_{\geq 0} \rightarrow real$.

Of course, we will need to add operations that link these new sorts to the sort real.

4.2.2 Subsets

Suppose we want to calculate with subsets of a given set. We shall have constants

$empty : \rightarrow subset$
to represent the set with no elements, and

\[ \text{universe} :\rightarrow \text{subset} \]

to represent the set which has all the elements of that which we are calculating over.

We shall combine subsets with the usual set manipulation operations:

\[ \text{union} : \text{subset} \times \text{subset} \rightarrow \text{subset} \]
\[ \text{intersection} : \text{subset} \times \text{subset} \rightarrow \text{subset} \]
\[ \text{complement} : \text{subset} \rightarrow \text{subset} \]

```
signature Subsets
sorts subset
constants empty : \rightarrow \text{subset}
         universe : \rightarrow \text{subset}
operations union : \text{subset} \times \text{subset} \rightarrow \text{subset}
               intersection : \text{subset} \times \text{subset} \rightarrow \text{subset}
               complement : \text{subset} \rightarrow \text{subset}
endsig
```

### 4.2.3 Strings

Suppose we have a signature \textit{Basic Strings with Length} for calculating with strings. We shall want to be able to create a string from some alphabet. If we allow strings of zero length, then we shall want a constant

\[ \text{empty} : \rightarrow \text{string} \]

for the empty string. We can build a string by prefixing letters from the alphabet using an operation

\[ \text{prefix} : \text{alphabet} \times \text{string} \rightarrow \text{string}. \]

To measure the length of a string, we shall use the natural numbers, and an operation

\[ \text{length} : \text{string} \rightarrow \text{nat} \]
4.2. EXAMPLES OF SIGNATURES

signature  Basic Strings with Length
sorts     alphabet, string, nat
constants empty : → string
           zero : → nat
operations prefix : alphabet × string → string
               length : string → nat
               succ : nat → nat
endsig

4.2.4 Storage Media

Thinking abstractly, a storage medium is simply something that stores data. We imagine that
it must have a store, and means of putting data into store and of taking data from store. These
"means" are operations on the store. Storage media are specified by different stores equipped
with different input-output operations.

What would be a simple interface for abstract storage media?

The following signature Storage models such an interface:

signature  Storage
sorts      store
           address
           data
constants
operations in : data × address × store → store
              out : address × store → data
endsig

For example, data structures store and access data in different ways: there are the record,
the array, the stack, the list, the queue, and many more. Some of these, such as the stack, do
not have addresses.

4.2.5 Machines

Thinking abstractly, a machine is simply a device that processes input and output. It has a
memory and a program; indeed it may be programmable. For simplicity, we imagine that it
must have

(i) a state that combines memory and commands, and

(ii) a means of reading input data into state and of writing the output data from the state.

The input of data changes the state of the machine. These ideas can be expressed by two
operations on the state. What would be a simple interface for abstract machines? An interface
to a machine is modelled by this signature \textit{Machine}:

<table>
<thead>
<tr>
<th>signature</th>
<th>\textit{Machine}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>\textit{state}</td>
</tr>
<tr>
<td></td>
<td>\textit{input}</td>
</tr>
<tr>
<td></td>
<td>\textit{output}</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>\textit{next} : \textit{state} \times \textit{input} \rightarrow \textit{state}</td>
</tr>
<tr>
<td></td>
<td>\textit{write} : \textit{state} \times \textit{input} \rightarrow \textit{output}</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3 Formal Definition of an Algebra

With the idea of a signature, we can define the idea of an algebra properly by assigning sets,
elements and functions to each sort, constant and operation named in a signature.

\textbf{Definition (Many-Sorted Algebra)} Let

\[
\Sigma_{Name} = (\text{Name}, S, < \Sigma_{w,s} \mid w \in S^*, s \in S > )
\]

be a signature. An algebra \textit{A with signature} \Sigma_{Name} or, more briefly, a

\textit{\Sigma_{Name}-algebra} \textit{A}

consists of:

(i) An \textit{S-indexed family}

\[
\langle A_s \mid s \in S \rangle
\]

of non-empty sets enumerated by sorts, where for each sort \( s \in S \) the set \( A_s \) interprets
the name \( s \) and is called the \textit{carrier} of sort \( s \).

(ii) An \textit{\( S^* \times S \)-indexed family}

\[
\langle \Sigma^A_{w,s} \mid w \in S^*, s \in S \rangle
\]

of sets of elements and sets of functions, enumerated by the constant and operation names.
4.3. FORMAL DEFINITION OF AN ALGEBRA

Constants For each sort \( s \in S \) and empty string \( \lambda \in S^* \),
\[
\Sigma_{\lambda,s}^A = \{ c^A \mid c \in \Sigma_{\lambda,s} \},
\]
where the element
\( c^A \in A_s \)
is called a constant of sort \( s \in S \) which interprets the constant symbol \( c \in \Sigma_{\lambda,s} \) in the algebra.

Operations For each non-empty word \( w = s(1) \cdots s(n) \in S^+ \) and each sort \( s \in S \)
\[
\Sigma_{w,s}^A = \{ f^A \mid f \in \Sigma_{w,s} \},
\]
where
\[
f^A : A_{s(1)} \times \cdots \times A_{s(n)} \to A_s
\]
or, more concisely,
\[
f^A : A^w \to A_s
\]
is called an operation or function with domain \( A^w = A_{s(1)} \times \cdots \times A_{s(n)} \), range \( A_s \) and
arity \( n \) which interprets the function symbol \( f \in \Sigma_{w,s} \) in the algebra.

This long and complicated definition (which includes within it the definition of a signature) seems a long way from the intuition that an algebra is just some sets together with some elements and functions. However, it is essentially the result of pinning down what the names for the sets, elements and functions are.

As we have already seen in the provisional definition, and in our illustrative examples, we display algebras in a similar way to signatures:

| algebra  | \( A \) |
| carriers | \ldots, A_s, \ldots |
| constants | \( \vdots \) |
| \( c^A : \to A_s \) | \( \vdots \) |
| operations | \( \vdots \) |
| \( f^A : A_{s_1} \times \cdots \times A_{s_n} \to A_s \) | \( \vdots \) |

4.3.1 General Notation and Examples

In our definition of a \( \Sigma \)-algebra \( A \), we have introduced the following new, general but simple, notation for the sets, elements and operations making up an algebra based on the names in the
signature \( \Sigma \):

<table>
<thead>
<tr>
<th>Name in Signature</th>
<th>Interpretation in Algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort name ( s )</td>
<td>non-empty set ( A_s )</td>
</tr>
<tr>
<td>product type ( w = s_1 \times \cdots \times s_n )</td>
<td>Cartesian product ( A_{s_1} \times \cdots \times A_{s_n} )</td>
</tr>
<tr>
<td>constant name ( c : \to s )</td>
<td>element ( c^A \in A_s )</td>
</tr>
<tr>
<td>operation name ( f : s_1 \times \cdots \times s_n \to s )</td>
<td>function ( f^A : A_{s_1} \times \cdots \times A_{s_n} \to A_s )</td>
</tr>
</tbody>
</table>

The point is that this is a general notation for algebras, designed to define and analyse data in general. This notation is fine for general cases but may seem cumbersome in examples, especially examples with familiar, or standard notation. However, even in familiar cases, where a user knows what to expect of a name, there is plenty of room for error. The signature demands answers to the question, given sort name \( s \), what exactly is the implementation of the data in \( A_s \). Let us try out this general notation.

**Example** Let \( \Sigma_{\text{Peano Arithmetic}} \) be the signature Peano Arithmetic in Section 4.1.1. Any \( \Sigma_{\text{Peano Arithmetic}} \)-algebra \( A \) that interprets the signature \( \Sigma_{\text{Peano Arithmetic}} \) can be written as follows:

\[
\begin{align*}
\text{algebra} & \quad A \\
\text{carriers} & \quad A_{\text{nat}} \\
\text{constants} & \quad \text{zero}^A : \to A_{\text{nat}} \\
\text{operations} & \quad \text{succ}^A : A_{\text{nat}} \to A_{\text{nat}} \\
& \quad \text{add}^A : A_{\text{nat}} \times A_{\text{nat}} \to A_{\text{nat}} \\
& \quad \text{mult}^A : A_{\text{nat}} \times A_{\text{nat}} \to A_{\text{nat}}
\end{align*}
\]

Now, we choose the signature Peano Arithmetic in Section 4.1.1 with a standard interpretation in mind, namely the Standard Model of Peano Arithmetic in Section 3.3.1, in which

(i) the carrier \( A_{\text{nat}} = \mathbb{N} = \{0, 1, 2, \ldots \} \), i.e., the natural numbers in decimal notation;

(ii) the constant \( \text{zero}^A = 0 \); and

(iii) the operations \( \text{succ}^A = \text{Succ} \), \( \text{add}^A = \text{Add} \) and \( \text{mult}^A = \text{Mult} \).

Of course, for such a familiar algebra there will be occasions when we will use the standard notation

\( \text{zero}^A = 0 \), \( \text{succ}^A = +1 \), \( \text{add}^A = + \) and \( \text{mult}^A = \).

This standard \( \Sigma_{\text{Peano Arithmetic}} \)-algebra is written as follows.
4.3. FORMAL DEFINITION OF AN ALGEBRA

| algebra   | $A_{\text{Standard}}$ |
| carriers  | $\mathbb{N}$          |
| constants | $0 : \rightarrow \mathbb{N}$ |
| operations| $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N}$  
|          | $\text{Add} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  
|          | $\text{Mult} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ |

Clearly, there are other choices for $A_{\text{nat}}$. The signature requires us to pause and be very precise about our interpretation.

We will now play around with these notations for algebras.

4.3.2 Examples of Algebras

4.3.3 Examples of Algebras for Basic Data Types

Let us illustrate the new definition and notations for algebras by interpreting some of the signatures for the Booleans and numbers given in Section 4.2. Our examples will be displayed rather than presented as tuples.

**Booleans** Given the signature $\Sigma_{\text{Boolean}}$ for a standard set of Boolean operations, first we apply the general notation for interpreting $\Sigma_{\text{Boolean}}$. An arbitrary $\Sigma_{\text{Boolean}}$-algebra has the form:

| algebra   | $A$ |
| carriers  | $A_{\text{Bool}}$ |
| constants | $true^A, false^A : \rightarrow A_{\text{Bool}}$ |
| operations| and$^A : A_{\text{Bool}} \rightarrow A_{\text{Bool}}$  
|          | not$^A : A_{\text{Bool}} \rightarrow A_{\text{Bool}}$ |

The standard interpretation of the Booleans is based on the set

$$B = \{tt, ff\}$$

and we might write, using common notation:
algebra \( A_{\text{Standard}} \)
carriers \( B \)
constants \( tt, ff : \to B \)
operations \( \wedge : B \times B \to B \)
\( \neg : B \to B \)

Since \( tt, ff \in B \), and \( \wedge \) and \( \neg \) are functions on \( B \), we do not have need of superscripts \( B \) etc.

**Integers** We introduced a signature \( \text{IntegersInfix} \) for an algebra of integers using a prefix notation for addition, subtraction and multiplication. Here is the general form of a \( \text{IntegersInfix} \)-algebra using the general notation:

| algebra   | \( A \) |
carriers   | \( A_{\text{int}} \) |
constants  | \( \text{zero}^A, \text{one}^A : \to A_{\text{int}} \) |
operations | \( \text{add}^A : A_{\text{int}} \times A_{\text{int}} \to A_{\text{int}} \) |
\( \text{minus}^A : A_{\text{int}} \to A_{\text{int}} \) |
\( \text{times}^A : A_{\text{int}} \times A_{\text{int}} \to A_{\text{int}} \) |

We also introduced a signature \( \text{Integers} \) for the integers that uses the standard infix notation for addition, subtraction and multiplication. The standard form of a \( \text{Integers} \)-algebra is based on the set

\[
Z = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}
\]

of integers in decimal notation. In the standard case, we need not drop the reference to \( Z \) on the operators of the signature, say by writing \( + \) for \( +^Z \).

| algebra   | \( A \) |
carriers   | \( Z \) |
constants  | \( 0, 1 : \to Z \) |
operations | \( _+^Z : Z \times Z \to Z \) |
\( _-^Z : Z \to Z \) |
\( _-^Z : Z \times Z \to Z \) |
4.3. FORMAL DEFINITION OF AN ALGEBRA

**Reals**  Here is a $\Sigma_{\text{Reals}}$-algebra of real numbers containing some infix and some postfix notations. We choose some standard representation $\mathbf{R}$ of the real numbers.

\[
\begin{array}{ll}
\text{algebra} & \text{Reals} \\
\text{carriers} & \mathbf{R}, \mathbf{B} \\
\text{constants} & 0^\mathbf{R}, 1^\mathbf{R}, \pi^\mathbf{R} : \to \mathbf{R} \\
\text{operations} & +^\mathbf{R} : \mathbf{R} \times \mathbf{R} \to \mathbf{R} \\
 & -^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \times^\mathbf{R} : \mathbf{R} \times \mathbf{R} \to \mathbf{R} \\
 & \div^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \sqrt{}^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & |{}^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \sin^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \cos^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \tan^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \exp^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & \log^\mathbf{R} : \mathbf{R} \to \mathbf{R} \\
 & =^\mathbf{R} : \mathbf{R} \times \mathbf{R} \to \mathbf{B} \\
 & <^\mathbf{R} : \mathbf{R} \times \mathbf{R} \to \mathbf{B} \\
 & \text{and}^\mathbf{R} : \mathbf{B} \times \mathbf{B} \to \mathbf{B} \\
 & \text{not}^\mathbf{R} : \mathbf{B} \to \mathbf{B} \\
\end{array}
\]

In many situations we might drop the superscript and subscript reference to $\mathbf{R}$ providing there could be no confusion as to which functions on reals were intended.

### 4.3.4 Storage Media

Thinking abstractly, an implementation of a storage medium is simply an algebra of signature $\Sigma_{\text{Storage}}$ which, using the general notation for interpreting signatures, has the form:

\[
\begin{array}{ll}
\text{algebra} & A \\
\text{carriers} & A_{\text{store}}, A_{\text{address}}, A_{\text{data}} \\
\text{constants} & \\
\text{operations} & \text{in}^A : A_{\text{data}} \times A_{\text{address}} \times A_{\text{store}} \to A_{\text{store}} \\
 & \text{out}^A : A_{\text{address}} \times A_{\text{store}} \to A_{\text{data}} \\
\end{array}
\]

However, we might choose a more suggestive notation for such an algebra:
4.3.5 Machines

Thinking abstractly, an implementation of a machine is simply an algebra of signature $\Sigma_{Machine}$. Using a suggestive notation for the carriers, a typical machine is an algebra of the form:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$State, Input, Output$</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
</tbody>
</table>
| operations | $next^A : State \times Input \rightarrow State$  
            | $write^A : State \times Input \rightarrow Output$ |

If the set $State$ is a finite set then $A$ is called a finite state machine.

4.3.6 Sets

Using the signature $\Sigma_{Subsets}$-algebra of Section 4.2.2, any $\Sigma_{Subsets}$-algebra $A$ using the general notation can be displayed in the following form:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$A_{\text{subset}}$</td>
</tr>
</tbody>
</table>
| constants | $\text{empty}^A : \rightarrow A_{\text{subset}}$  
            | $\text{universe}^A : \rightarrow A_{\text{subset}}$ |
| operations | $\text{union}^A : A_{\text{subset}} \times A_{\text{subset}} \rightarrow A_{\text{subset}}$  
              | $\text{intersection}^A : A_{\text{subset}} \times A_{\text{subset}} \rightarrow A_{\text{subset}}$  
              | $\text{complement}^A : A_{\text{subset}} \rightarrow A_{\text{subset}}$. |
4.3. FORMAL DEFINITION OF AN ALGEBRA

The intended interpretation is an algebra $A$ in which

(i) the carrier set $A_{\text{subset}}$ is the power set $\mathcal{P}(X)$ of a non-empty set $X$, i.e., the set of all subsets of the set $X$;

(ii) the constants are

\begin{align*}
\text{empty}^A &= \emptyset, \quad \text{the empty set;} \\
\text{universe}^A &= X, \quad \text{the given set} \ X;
\end{align*}

(iii) the operations are defined for any $V, W \in \mathcal{P}(X)$ by

\begin{align*}
\text{union}^A(V, W) &= V \cup W; \\
\text{intersection}^A(V, W) &= V \cap W; \\
\text{complement}^A(V) &= X - V
\end{align*}

This gives us the algebra:

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{algebra} & $A$ \\
\hline
\textbf{carriers} & $A_{\text{subset}} = \mathcal{P}(X)$ \\
\hline
\textbf{constants} & $\emptyset : \to \mathcal{P}(X)$ \\
& $X : \to \mathcal{P}(X)$ \\
\hline
\textbf{operations} & $\cup : \mathcal{P}(X) \times \mathcal{P}(X) \to \mathcal{P}(X)$ \\
& $\cap : \mathcal{P}(X) \times \mathcal{P}(X) \to \mathcal{P}(X)$ \\
& $\setminus : \mathcal{P}(X) \to \mathcal{P}(X)$ \\
\hline
\end{tabular}
\end{center}

4.3.7 Strings

Using the signature $\Sigma_{\text{Strings}}$ from Section ??, and applying our general mathematical notation, any $\Sigma_{\text{Basic Strings with Length}}$-algebra $A$ will have the following mathematical form:

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{algebra} & $A$ \\
\hline
\textbf{carriers} & $A_{\text{alphabet}}, A_{\text{string}}, A_{\text{nat}}$ \\
\hline
\textbf{constants} & $\text{empty}^A : \to A_{\text{string}}$ \\
& $\text{zero}^A : \to A_{\text{nat}}$ \\
\hline
\textbf{operations} & $\text{prefix}^A : A_{\text{alphabet}} \times A_{\text{string}} \to A_{\text{string}}$ \\
& $\text{length}^A : A_{\text{string}} \to A_{\text{nat}}$ \\
\hline
\end{tabular}
\end{center}

However, when we interpret $\Sigma_{\text{Basic Strings with Length}}$, we will actually have an algebra of strings of the form defined in Section 3.7.2, where
(i) the carrier sets
\[ A_{\text{alphabet}} = T, \quad A_{\text{string}} = T^*, \quad \text{and} \quad A_{\text{nat}} = \mathbb{N}; \]

(ii) the constants
\[ \text{empty}^A = \epsilon, \quad \text{zero}^A = 0; \]

and

(iii) the operations
\[ \text{empty}^A = \epsilon, \quad \text{prefix}^A = \text{Prefix} \quad \text{length}^A = |\cdot|, \quad \text{and} \quad \text{succ}^A = \_ + 1, \]

This gives us the algebra:

<table>
<thead>
<tr>
<th>algebra</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>T, T*, N</td>
</tr>
<tr>
<td>constants</td>
<td>( \epsilon : \to T^* )</td>
</tr>
<tr>
<td></td>
<td>( 0 : \to \mathbb{N} )</td>
</tr>
<tr>
<td>operations</td>
<td>( \text{Prefix} : T \times T^* \to T^* )</td>
</tr>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>( _ + 1 : \mathbb{N} \to \mathbb{N} )</td>
</tr>
</tbody>
</table>

### 4.4 Algebras with Booleans and Flags

Looking back on the many examples of algebras in Sections 3.2–3.7, some common features are noticeable. One prominent feature is the special role of the Booleans in defining tests on data in an algebra. Another is the use of special data to flag errors, unknowns, and other exceptional, and usually undesirable, circumstances. We will examine these features in general, and exercise further our new official definition of a \( \Sigma \)-algebra.

#### 4.4.1 Algebras with Booleans

Tests on data are needed to govern the flow of control in computations. Thus, most algebras that model data types will contain the Booleans, as tests are Boolean-valued functions. To define these algebras, we need to

(i) choose a notation for Booleans, and

(ii) require that the notation has a standard interpretation.

**Definition (Algebras with Booleans)**
4.4. ALGEBRAS WITH BOOLEANS AND FLAGS

(i) A signature $\Sigma$ is a signature with Booleans if

\[
\text{Bool}
\]

is a sort name in $\Sigma$;

\[\text{true, false} : \rightarrow \text{Bool}\]

are constant symbols in $\Sigma$; and

\[\text{not} : \text{Bool} \rightarrow \text{Bool}\]
\[\text{and} : \text{Bool} \times \text{Bool} \rightarrow \text{Bool}\]

are operation symbols in $\Sigma$.

(ii) A $\Sigma$-algebra $A$ is an algebra with Booleans if

\[
\Sigma
\]

is a signature with Booleans;

\[A_{\text{Bool}} = B = \{tt, ff\}\]

is a carrier of $A$;

\[\text{true}_{\text{Bool}} = tt\]
\[\text{false}_{\text{Bool}} = ff\]

are constants in $A$; and

\[\text{not}_{\text{Bool}}(b) = \neg b\]
\[\text{and}_{\text{Bool}}(b_1, b_2) = b_1 \land b_2\]

are the standard operations in $B$. That is, the sort $\text{Bool}$ and its associated constants and operations have their standard interpretation in $A$.

Once one has the Booleans, two kinds of test may be added to any algebra, namely,

\[\text{equality} \quad \text{and} \quad \text{conditionals}.\]

Total and Partial Equality

For each sort $s$, we can add the operation name

\[eq_s : s \times s \rightarrow \text{Bool}\]

in $\Sigma$. We can interpret this with the operation

\[eq^A_s : A_s \times A_s \rightarrow B\]

which we define to have the standard interpretation

\[eq^A_s(x, y) = \begin{cases} 
    tt & \text{if } x = y; \\
    ff & \text{if } x \neq y.
\end{cases} \]
Now, it may not be desirable to add equality with the standard interpretation. Two variations are:

\[ \text{eq}_s^A(x, y) = \begin{cases} 
  t & \text{if } x = y; \\
  \uparrow & \text{if } x \neq y.
\end{cases} \]

\[ \text{eq}_s^A(x, y) = \begin{cases} 
  \uparrow & \text{if } x = y; \\
  \downarrow & \text{if } x \neq y.
\end{cases} \]

The last interpretation is to be expected when testing infinite data such as real numbers or infinite sequences. Given two infinite sequences, it is possible to search for a difference between them, but it may not be possible to test they are equal.

**Conditional**

For each sort \( s \), we can add the operation name

\[ if_s : \text{Bool} \times s \times s \rightarrow s \]

in \( \Sigma \). We can interpret this with the operation

\[ if_s^A : B \times A_s \times A_s \rightarrow A_s \]

which we define by

\[ if_s^A(b, x, y) = \begin{cases} 
  x & \text{if } b = tt; \\
  y & \text{if } b = ff.
\end{cases} \]

### 4.4.2 Algebras with an Unspecified Element

There are several reasons why we might add a new element to an existing set of data. For example, we have added:

(i) \( u \) to denote unknown or unspecified in the Booleans (Section 3.2.4); and

(ii) \( \text{error} \) to denote an exception or error in data types (Sections 3.3.4 and 3.5.1).

And other situations require the addition of:

(iii) \( \text{overflow} \) to denote overflow in finite number systems;

(iv) \( +\infty, -\infty \) to denote points in infinite number systems;

(v) \( \uparrow \) or \( \perp \) to denote the undefined value of a function.

Let us define one of these processes in general.

Let \( A \) be any \( \Sigma \)-algebra with the Booleans. We make a new signature \( \Sigma^u \) by adding a new constant symbol \( \text{unspec}_s \) of each sort \( s \ldots, s, \ldots \) of \( \Sigma \), and transforming the operations of \( \Sigma \) to accommodate the unspecified elements:
4.4. ALGEBRAS WITH BOOLEANS AND FLAGS

\[
\begin{array}{|l|}
\hline
\text{signature} & \Sigma^u \\
\hline
\text{sorts} & \ldots, s^u, \ldots \\
\hline
\text{constants} & \ldots, c^u : \rightarrow s^u, \ldots \\
& \ldots, \text{unspecified}_s : \rightarrow s^u, \ldots \\
\hline
\text{operations} & \ldots, f^u : \quad s_1^u \times \cdots \times s_n^u \rightarrow s^u, \ldots \\
& \ldots, \text{is\_unspecified}_s : \quad s^u \rightarrow \text{Bool}, \ldots \\
\hline
\end{array}
\]

Now we show how to make a $\Sigma^u$-algebra $A^u$ from the $\Sigma$-algebra $A$. Let $A$ be a $\Sigma$-algebra and consider the effect of augmenting $A$ with special objects

\[ u_s \notin A_s \]

to represent an \textit{undefined} or \textit{unspecified datum} of sort $s$. We will make a new algebra $A^u$ with carriers

\[ A^u_s = A_s \cup \{u_s\}. \]

The constants of $A^u_s$ are those of $A_s$ together with

\[ u_s \]

interpreting \textit{unspecified}_s.

The operations of $A^u$ are derived from those of $A$ as follows: let

\[ F : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s \]

be an operation of $A$ then define the new operation

\[ F^u : A_{s_1}^u \times \cdots \times A_{s_n}^u \rightarrow A_s^u \]

by

\[ F^u(x_1, \ldots, x_n) = \begin{cases} 
F(x_1, \ldots, x_n) & \text{if } x_1 \in A_{s_1}, \ldots, x_n \in A_{s_n}; \\
u_s & \text{otherwise.}
\end{cases} \]

The restriction of operations on $A^u$ that makes them return an unspecified value if any of the input is unspecified is sometimes called a \textit{strictness assumption}.

We can extract $A$ from $A^u$ by means of the boolean valued function

\[ \text{IsUnspecified} : A^u_s \rightarrow \{tt, ff\} \]

that interprets \textit{is\_unspecified}_s, which we define for any $a \in A^u_s$ by

\[ \text{IsUnspecified}_s(a) = \begin{cases} 
tt & \text{if } a = u_s; \\
ff & \text{otherwise.}
\end{cases} \]
4.5 Generators and Constructors

The constants and operations of an algebra can be chosen for any number of reasons. They might be essential or convenient for handling data in a particular situation or application. Different situations will suggest different operations. Now, one job operations do is to compute, construct, or generate new data from given data. A simple question, relevant to any \( \Sigma \)-algebra \( A \), is this:

\[
\text{Given a subset } G \text{ of elements of } A, \text{ if we apply repeatedly all the operations of } A \\
\text{to the elements of } G, \text{ what set of elements do we generate? Can we find a } G \text{ from} \\
\text{which we can generate all the elements of } A? \\
\]

In terms of modelling data types, these questions concern what data can be accessed by

a programmer by a program applying the constants and operations declared in the interface. Perhaps there are data in the implementation that cannot be obtained via the interface?

The complete answers to these questions involve a considerable number of mathematical ideas, including subalgebras and terms. We will take our time over their explanation (ending in Chapter 12). However, the basic intuitions are easy to grasp. Let us study some examples, and formulate an important definition.

Example (Constructing Natural Numbers) Let \( \Sigma \) be the signature
and consider the standard \( \Sigma \)-algebra

\[
A = (\mathbb{N}; 0; \text{Succ})
\]

where \( \text{Succ}(n) = n + 1 \). Suppose we apply all the operations of \( A \) repeatedly to all the constants of \( A \). For such a simple algebra, it is easy to see that we generate:

\[
0, \text{Succ}(0), \text{Succ}(\text{Succ}(0)), \text{Succ}(\text{Succ}(\text{Succ}(0))), \ldots
\]

which in the standard algebra \( A \) is

\[
0, 1, 2, 3, \ldots
\]

In this example, we conclude that every element of \( A \) can be constructed from 0 by applying \( \text{Succ} \).

Now let us change the signature and the algebra slightly. Let \( \Sigma \) be:

\[
\begin{aligned}
\text{signature} \\
\text{sorts} & \quad \text{nat} \\
\text{constants} & \quad 0 : \rightarrow \text{nat} \\
\text{operations} & \quad \text{pred} : \text{nat} \rightarrow \text{nat} \\
\text{endsig}
\end{aligned}
\]

and consider the standard \( \Sigma \)-algebra

\[
A = (\mathbb{N}; 0, \text{Pred})
\]

where \( \text{Pred}(0) = 0 \) and \( \text{Pred}(n) = n - 1 \) if \( n > 0 \).

Suppose we apply all the operations of \( A \) repeatedly to all the constants of \( A \). This time, we get

\[
0, \text{Pred}(0), \text{Pred}(\text{Pred}(0)), \text{Pred}(\text{Pred}(\text{Pred}(0))), \ldots
\]

which in the standard algebra \( A \) is

\[
0, 0, 0, 0, \ldots
\]

In this example, we can construct very little.
Suppose we apply the operations of $A$ repeatedly to some subset of $G \subseteq \mathbb{N}$. Suppose $G = \{10\}$. Then we get:

\[
\begin{align*}
\text{Constant:} & \quad 0, \text{Pred}(0), \text{Pred(\text{Pred}(0))), \text{Pred(\text{Pred(Pred(0))))}, \ldots \\
G & \quad 10, \text{Pred}(10), \text{Pred(\text{Pred}(10))), \text{Pred(\text{Pred(Pred(10))))}, \ldots
\end{align*}
\]

which in $A$ is

\[
0, 0, 0, 0, \ldots \\
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0, 0, \ldots
\]

That is, we generate the set

\[
\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.
\]

More generally, if we apply the operations repeatedly to

\[
G = \{n\}, \text{we get } \{0, 1, 2, \ldots, n\},
\]

and

\[
G = \{m, n\}, \text{we get } \{0, 1, 2, \ldots, \max(m, n)\}.
\]

**Example (Constructing Rational Numbers)** Let $\Sigma$ be the signature

```
signature  Rationals
sorts      rat
constants  0, 1 :\rightarrow nat
operations +, - : rat \times rat \rightarrow rat
         - : rat \rightarrow rat
         \times : rat \times rat \rightarrow rat
         ^{-1} : rat \rightarrow rat
endsig
```

and consider the standard $\Sigma_{\text{Rationals}}$-algebra

\[
A = (\mathbb{Q}; 0, 1; +, -, \times^{\text{-1}})
\]

where the operations are the standard arithmetic operations.

Suppose we apply all the operations of $A$ repeatedly to all the constants of $A$. We will list the elements created in stages.

**Addition**

\[
0, 0 + 0, 0 + 0 + 0, \ldots, \\
1, 1 + 1, 1 + 1 + 1, \ldots
\]

(We will omit terms such as $0 + 1, 0 + 0 + 1 + 1, \ldots$) We add to these negative numbers:
4.6. SUBALGEBRAS

Subtraction

\[-1, -(1 + 1), -(1 + 1 + 1), \ldots\]

In particular, in \( A \), we get all the integers from addition and subtraction

\[\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\]

Applying multiplication at this stage does not produce new data as the product of two integers is an integer. However, we introduce some fractions:

Division

\[
\begin{align*}
&\frac{1}{1'}, \frac{1}{1+1'}, \frac{1}{1+1+1'}, \ldots \\
&\frac{1}{1'}, \frac{1}{(1+1)'}, \frac{1}{(1+1+1)'}, \ldots
\end{align*}
\]

In particular, in \( A \), we have added

\[1, \frac{1}{2}, \frac{1}{3}, \ldots \quad \text{and} \quad -1, -\frac{1}{2}, -\frac{1}{3}, \ldots\]

Now, applying multiplication to integers and the fractions we have, we introduce more numbers such as:

\[
\frac{1+1}{1+1+1'}, \frac{1+1+1}{1+1'}, \frac{-(1+1)}{1+1+1'}, \frac{-(1+1+1)}{1+1},
\]

or, in general, any fraction

\[
\frac{1+1+\cdots+1}{1+1+\cdots+1} \quad p \text{ times} \quad \text{or} \quad \frac{-(1+1+\cdots+1)}{1+1+\cdots+1} \quad q \text{ times}
\]

In particular, in \( A \), we have added any fraction

\[
\frac{p}{q}
\]

for \( q \neq 0 \).

We conclude that every element of \( Q \) can be constructed from 0 and 1 by applying the operations of \( A \).

These examples illustrate the following technical idea.

Definition (Constructors) Let \( A \) be a \( \Sigma \)-algebra. The constants and operations of \( \Sigma \) are constructors for \( A \) if every element of \( A \) can be computed by the repeated application of the operations of \( A \) to the constants of \( A \).

4.6 Subalgebras

To conclude this chapter, we consider some simple ideas about comparing and changing algebras.
The natural numbers are contained in the integers. The integers are contained in the rational numbers. The rational numbers are contained in the real numbers. We are used to expressing these ideas using sets and the subset relation, thus:

\[ \mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}. \]

Furthermore, we are used to the idea that the operations on these numbers are equivalent. Since \( \mathbb{R} \) extends \( \mathbb{Q} \), \( \mathbb{Q} \) extends \( \mathbb{Z} \), \( \mathbb{Z} \) extends \( \mathbb{N} \), we expect for \( m, n \in \mathbb{N} \),

\[ n +^N m, n +^Z m, n +^Q m, n +^R m \]

to be the same number. In fact, this is not correct: there are tricky details concerning representations of these numbers to take care of before we can make these statements precise and correct. For example, depending on the precise definitions of these sets, we have to convert between number representations, say, with transformations

\[
\begin{align*}
n \in \mathbb{N} & \iff +n \in \mathbb{Z} \\ z \in \mathbb{Z} & \iff +z \equiv 0 \in \mathbb{Q} \\
\frac{p}{q} \in \mathbb{Q} & \iff a_m \cdots a_0.b_0b_1 \cdots b_n \cdots
\end{align*}
\]

This process introduces conceptual and technical complications that we will sort out much later (when we write about homomorphisms in Chapter 7). However, intuitively, the idea seems clear.

When calculating with these numbers we use slightly different operations and, therefore, we use different signatures and algebras. For example, we might use

\[
\begin{align*}
(N; 0; n + 1, n + m, n.m) \\
(Z; 0, 1; x + y, -x, x.y) \\
(Q; 0, 1; x + y, -x, x.y, x^{-1}) \\
(R; 0, 1; x + y, -x, x.y, x^{-1}).
\end{align*}
\]

In this section, to express the idea that two algebras share data and operations, such as \( \mathbb{Q} \) and \( \mathbb{R} \), we will introduce the idea of a

\textit{subalgebra}.

In the next section, to express the idea that one \( \Sigma \)-algebra has more data sets and operations than another, we will introduce the concept of an

\textit{expansion}

and, conversely, that an algebra has fewer data sets and operations than another, we will introduce the concept of a

\textit{reduct}.
4.6. SUBALGEBRAS

4.6.1 Examples of Subalgebras

Let us look at a simple example of a subalgebra to introduce the idea. Consider an algebra of integers made from addition and subtraction.

Let $\Sigma$ be the signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>int</td>
</tr>
<tr>
<td>constants</td>
<td>$0 :\rightarrow$ int</td>
</tr>
</tbody>
</table>
| operations $+: int \times int \rightarrow int$  
- $-_>: int \rightarrow int$ |
| endsig    |          |

Let $A$ be the standard $\Sigma$-algebra of the integers based on the set

$$\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$$

of numbers in decimal notation:

$$A = (\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z})$$

Consider the subset

$$Even = \{ \ldots, -4, -2, 0, 2, 4, \ldots \}$$

of all even integers, i.e., integers divisible by 2. If we apply the constant and operations

$$0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z}$$

of $A$ to even integers, we get even integers.

First note that 0 is even. Let $2z_1$ and $2z_2$ be any even integers, where $z_1, z_2 \in \mathbb{Z}$. Then,

$$2z_1 +^\mathbb{Z} 2z_2 = 2(z_1 +^\mathbb{Z} z_2)$$

is also even. Let $2z$ be any even integer where $z \in \mathbb{Z}$. Then

$$-^\mathbb{Z}2z = 2(-^\mathbb{Z}z)$$

is also even.

We say that the even integers $Even$ are

closed under the operations

of $A$. Now because the set $Even$ is closed, we can make a $\Sigma$-algebra

$$B = (Even; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z})$$
that is contained in the $\Sigma$-algebra

$$A = (\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z}).$$

This $B$ we call a $\Sigma$-subalgebra of $A$.

Notice that the subset

$$Odd = \{\ldots, -3, -1, 1, 3, \ldots\}$$

of all odd numbers is not closed under the operations of $A$. For example,

$$1 + _\mathbb{Z} 3 = 4.$$

### 4.6.2 Subalgebras

Let us formulate the idea of a subalgebra in general.

**Definition (Subalgebra)** Let $A$ be an $S$-sorted $\Sigma$-algebra. An $S$-indexed family of subsets

$$B = \langle B_s \subseteq A_s \mid s \in S \rangle.$$ forms a $\Sigma$-subalgebra of $A$ if

(i) the subsets $B_s$ contain the constants of $A$ named in $\Sigma$; and

(ii) the subsets $B_s$ are closed under the operations of $A$ named in $\Sigma$. I.e., for each $f \in \Sigma$, applying each operation $f^A$ of $A$ to elements of $B$ produces elements of $B$: for

$$b_1 \in B_{s(1)}, \ldots, b_n \in B_{s(n)} \Rightarrow f^A(b_1, \ldots, b_n) \in B_s.$$

Given these conditions, we can make a $\Sigma$-algebra using $B$ and the operations of $A$, as illustrated in Figure 4.1.

![Figure 4.1: A is a subalgebra of B.](image)

If $B$ is a $\Sigma$-subalgebra of $A$ then we may omit reference to $\Sigma$ and simply say that $B$ is a subalgebra of $A$, writing

$$B \leq A.$$

If $B$ is a subalgebra of $A$ but $B \neq A$ then we say that $B$ is a proper subalgebra of $A$ (or $A$ is a proper extension of $B$), and write

$$B < A.$$
4.6. SUBALGEBRAS

Example (Integers) We can easily generalise the example of the even integers. Let $\Sigma$ and $A$ be as in Section 4.1.1. Let $n \in \mathbb{N}$ and define for $n > 0$,

$$n\mathbb{Z} = \{ z \in \mathbb{Z} \mid z \text{ is divisible by } n \}.$$  

A typical element of $n\mathbb{Z}$ has the form

$$nz$$

for some $z \in \mathbb{Z}$. We show that $n\mathbb{Z}$ is the carrier of a subalgebra using the definition in Section 4.1.2.

Claim For any $n > 0$, $B = (n\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z})$ is a subalgebra of $A = (\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z})$.

Proof Clearly $n\mathbb{Z} \subseteq \mathbb{Z}$. We must check the two closure conditions for constants and operations.

(i) There is only one constant symbol $0 \in \Sigma$. In $B$ it is interpreted as the standard integer zero of $A$ which is divisible by $n$ and

$$0^\mathbb{Z} \in n\mathbb{Z}$$

so $n\mathbb{Z}$ is closed under the constants of $A$.

(ii) There are two operations $+,- \in \Sigma$. Addition $+$ is interpreted in $B$ as the standard integer addition of $A$ and

$$nz_1 +^\mathbb{Z} nz_2 = n(z_1 +^\mathbb{Z} z_2) \in n\mathbb{Z}$$

So $n\mathbb{Z}$ is closed under addition of $A$. Subtraction $-$ is interpreted in $B$ as the standard integer subtraction of $A$ and

$$-^\mathbb{Z}nz = n(-^\mathbb{Z} z) \in n\mathbb{Z}$$

So $n\mathbb{Z}$ is closed under subtraction of $A$. 

\[ \square \]

4.6.3 Expansions and Reducts

When computing with an algebra $A$ it may be necessary, or convenient, to add new sets of data and appropriate operations. For instance, we may need to add

(i) Booleans,

(ii) natural numbers, or

(iii) finite and infinite sequences of elements from $A$.

Adding sets and operations leads to the construction of some new algebra $B$ that is an expansion of $A$.

There is an extensive range of constructions for extending signatures and algebras. We will formulate one simple and fundamental definition of an expansion or augmentation of a signature and an algebra.
Definition (Signature Expansion and Reduct) Let \( \Sigma \) be an \( S \)-sorted signature and \( \Sigma' \) an \( S' \)-sorted signature. We say that \( \Sigma' \) is an expansion of \( \Sigma \) or that \( \Sigma \) is a reduct or subsignature of \( \Sigma' \) if, and only if, \( \Sigma' \) has all the sorts, constant names and operation names of \( \Sigma \). More precisely,

1. Any sort in \( S \) is also in \( S' \); i.e.,
   \[ S \subseteq S'. \]

2. Any constant symbol in \( \Sigma \) is also in \( \Sigma' \); i.e., for any sort \( s \in S \), and any constant \( c : \rightarrow s \),
   \[ c \in \Sigma \quad \text{implies} \quad c \in \Sigma'. \]

3. Any function symbol in \( \Sigma \) is also in \( \Sigma' \); i.e., for any sorts \( s, s(1), \ldots, s(n) \in S \), and for any function symbol \( f : s(1) \times \cdots \times s(n) \rightarrow s \),
   \[ f \in \Sigma \quad \text{implies} \quad f \in \Sigma'. \]

Definition (Algebra Expansion) Let \( \Sigma' \) be an expansion of an \( S \)-sorted signature \( \Sigma \). Let \( A \) be a \( \Sigma \)-algebra and \( B \) a \( \Sigma' \)-algebra, then \( B \) is said to be a \( \Sigma' \)-expansion of \( A \) or \( A \) is a \( \Sigma \)-reduct of \( B \) if, and only if, \( B \) contains all the carriers, constants and operations of \( A \). More precisely,

1. for each sort \( s \in S \)
   \[ A_s = B_s; \]

2. for each sort \( s \in S \), each constant symbol \( c : \rightarrow s \in \Sigma \)
   \[ c^A = c^B; \]

and

3. for any sorts \( s, s(1), \ldots, s(n) \in S \), any function symbol \( f : s(1) \times \cdots \times s(n) \rightarrow s \in \Sigma \),
   \[ f^A = f^B. \]

We write

\[ B|_{\Sigma} \]

to denote the \( \Sigma \)-algebra \( A \) obtained from the \( \Sigma' \)-algebra \( B \) by removing the \( \Sigma' \) operations not named in \( \Sigma \), i.e., the \( \Sigma \)-reduct of the \( \Sigma' \)-algebra \( B \).

Example Suppose we have a signature \( \Sigma \) for real numbers:

<table>
<thead>
<tr>
<th>signature</th>
<th>Reals</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>real</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1 : real</td>
</tr>
<tr>
<td>operations</td>
<td>real \times real \rightarrow real</td>
</tr>
<tr>
<td></td>
<td>real \rightarrow real</td>
</tr>
<tr>
<td></td>
<td>real \times real \rightarrow real</td>
</tr>
<tr>
<td></td>
<td>real \rightarrow real</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>
and consider the standard algebra of real numbers

\[ A = (\mathbb{R}; 0, 1; +, -, \cdot, ^{-1}) \].

Let \( \Sigma_0 \) be \( \Sigma \) with the division operator removed. Then, the \( \Sigma_0 \)-reduct is the \( \Sigma_0 \)-algebra

\[ A|_{\Sigma_0} \]

which is \( A \) with the division operator removed.

**Example** We have seen the example of a signature and algebra with the Booleans in Section 4.3.1.

Suppose we expand \( \Sigma \) to \( \Sigma' \) by adding sorts constants and operations for Booleans. Then \( \Sigma' \) is

<table>
<thead>
<tr>
<th>signature</th>
<th>Reals with Booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>real, bool</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1 : ( \rightarrow ) real</td>
</tr>
<tr>
<td></td>
<td>( tt, ff : \rightarrow ) Bool</td>
</tr>
<tr>
<td>operations</td>
<td>( +, \cdot : ) real ( \times ) real ( \rightarrow ) real</td>
</tr>
<tr>
<td></td>
<td>( -, \cdot : ) real ( \rightarrow ) real</td>
</tr>
<tr>
<td></td>
<td>( \cdot^{-1} : ) real ( \rightarrow ) real</td>
</tr>
<tr>
<td></td>
<td>( \text{not} : ) Bool ( \rightarrow ) Bool</td>
</tr>
<tr>
<td></td>
<td>( \text{and} : ) Bool ( \times ) Bool ( \rightarrow ) Bool</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>

and consider the standard algebra of real numbers and Booleans

\[ B = (\mathbb{R}, \mathbb{B}; 0, 1, tt, ff; +, -, \cdot, ^{-1}, \neg, \land) \].

Then the \( \Sigma \)-reduct of the \( \Sigma' \)-algebra \( B \) is

\[ B|_{\Sigma} \]

which is, of course, \( A \).

## 4.7 Importing Algebras

Suppose we have a data type with

\[ \text{interface } \Sigma_{\text{Old}} \text{ and implementation } A_{\text{Old}} \]

and we wish to use it in creating a new data type with
interface $\Sigma_{New}$ and implementation $A_{New}$.

Suppose, too, that the old data type will not be changed in any way; simply, it will be used in the new data type. For example, if we want to add arrays to some existing data type of real numbers, then we must make a new data type containing both the real numbers and arrays of real numbers. Thus, in the type of construction we have in mind, the contents of $\Sigma_{Old}$ are included in $\Sigma_{New}$, and the contents of $A_{Old}$ are included in $A_{New}$. Using the definition of signature and algebra reduct, we express this formally by the condition

$$ \Sigma_{New} \text{ is an expansion of } \Sigma_{Old} \text{ and the reduct } A_{New}|_{\Sigma_{Old}} = A_{Old}. $$

The construction is called

**importing**

and is depicted in Figure 4.2.

![Diagram](image)

**Figure 4.2:** An impression of the idea of importing.

We start by reflecting on the construction of some of the algebras we met in the previous chapter, in order to formulate some general techniques for adding and removing data sets and operations from algebras. We extend the notation for displaying signatures with a new component

**import**

which allows us to describe concisely the addition of new data and operations to an existing signature. We will use **import** as a handy notation for specific tasks. However, it is deceptively simple, and the general idea of importing is quite complicated as we will see later.
4.7. IMPORTING ALGEBRAS

4.7.1 Importing the Booleans

Most of our data types contain tests that are needed in computations. Therefore, most of our many-sorted algebras contain the Booleans, their basic operations (e.g., Not, And), and possibly other Boolean-valued operations (e.g., equality, conditional). They are algebras with Booleans, as defined in Section 4.4.1. Two examples are the two-sorted algebras of Peano Arithmetic with Booleans (in Section 3.3) and real numbers with Booleans (in Section 3.5).

Now, we may think of constructing such a data type by adding new sets and operations to an existing data type of Booleans. This leads to the idea of

\textit{importing the Booleans}

into the new algebra, or of the new algebra

\textit{inherting the Booleans}.

Recalling the two stages of any algebraic construction, we describe this process as follows.

\begin{tabular}{|l|}
\hline
**Old Signature/Interface** First, let $\Sigma_{\text{Booleans}}$ be a signature for the Booleans \\
\hline

\begin{tabular}{|l|}
\hline
**signature** & $\text{Booleans}$ \\
**sorts** & $\text{bool}$ \\
**constants** & $true, false :\to \text{bool}$ \\
**operations** & $\text{not} : \text{bool} \to \text{bool}$ \\
& $\text{and} : \text{bool} \times \text{bool} \to \text{bool}$ \\
\hline
\end{tabular}

\end{tabular}

\begin{tabular}{|l|}
\hline
**New Signature/Interface** Suppose we are constructing a new signature $\Sigma_{\text{New}}$ from the signature $\Sigma_{\text{Booleans}}$ so that we may define some new operations and tests using the Booleans, such as conditionals and equality. The new signature $\Sigma_{\text{New}}$ may be defined concisely by \\
\hline

\begin{tabular}{|l|}
\hline
**signature** & $\text{New}$ \\
**import** & $\text{Booleans}$ \\
**sorts** & $\ldots, s_{\text{new}}, \ldots$ \\
**constants** & $\ldots, c :\to s, \ldots$ \\
**operations** & $\ldots, eq_{s} : s \times s \to \text{bool}, \ldots$ \\
& $\ldots, if_{s} : \text{bool} \times s \times s \to s, \ldots$ \\
& $\ldots, f : s(1) \times \cdots \times s(n) \to s_{\text{new}}, \ldots$ \\
\hline
\end{tabular}

\end{tabular}

where the sorts

\begin{tabular}{|l|}
\hline
\end{tabular}
\bull \ldots, s_{\text{new}}, \ldots \text{ used are new, i.e., } \textit{bool} \text{ is not in the list, and}

\bull \text{ the sorts } \ldots, s, \ldots, s(1), \ldots, s(n), \ldots \text{ used in the declarations of the new constants and functions may be from the new sorts declared, or the sort } \textit{bool} \text{ of the Booleans.}

This \texttt{import} notation is interpreted by:

(i) \text{ substituting all the components of the signature named in } \texttt{import} \text{ into the } \texttt{sorts, constants} \text{ and } \texttt{operations} \text{ declared after } \texttt{import} \text{ in } \Sigma_{\text{New}}; \text{ and}

(ii) \text{ allowing any } \texttt{sort} \text{ to be included in the type of any new operation.}

The new signature defined by the notation above is simply:

<table>
<thead>
<tr>
<th>signature</th>
<th>New$_0$</th>
</tr>
</thead>
</table>
| sorts | \texttt{bool},
\ldots, s_{\text{new}}, \ldots |
| constants | true, false :\rightarrow \texttt{bool}
\ldots, c :\rightarrow s, \ldots |
| operations | \textit{not} : \texttt{bool} \rightarrow \texttt{bool}
\textit{and} : \texttt{bool} \times \texttt{bool} \rightarrow \texttt{bool}
\ldots, \textit{eq$_s$} : s \times s \rightarrow \texttt{bool}, \ldots
\ldots, \textit{if$_s$} : \texttt{bool} \times s \times s \rightarrow s, \ldots
\ldots, \textit{f} : s(1) \times \cdots \times s(n) \rightarrow s, \ldots |

where we have used New$_0$ to indicate the removal of \texttt{import}. Clearly, \Sigma_{\text{New$_0$}} \text{ is an expansion of } \Sigma_{\text{Booleans}} \text{ in the precise sense defined in Section 4.6.3.}

Old Algebra/Implementation \texttt{Let } B \texttt{ be the algebra based on the set } B = \{tt, ff\} \texttt{ of Booleans in 3.2:}

<table>
<thead>
<tr>
<th>algebra</th>
<th>Booleans</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>B</td>
</tr>
<tr>
<td>constants</td>
<td>tt, ff :\rightarrow B</td>
</tr>
</tbody>
</table>
| operations | \textit{Not} : B \rightarrow B
\textit{And} : B \times B \rightarrow B |
New Algebra/Implementation The new $\Sigma_{New}$-algebra $A_{New}$ has the form:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$New$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$B$</td>
</tr>
<tr>
<td>carriers</td>
<td>$..., A_s, ...$</td>
</tr>
<tr>
<td>constants</td>
<td>$..., C : \rightarrow A_s, ...$</td>
</tr>
</tbody>
</table>
| operations| $..., Eq_s : s \times s \rightarrow B, ...$  
$..., If_s : B \times s \times s \rightarrow s, ...$  
$..., F : A_{s[1]} \times \cdots \times A_{s[n]} \rightarrow A_s, ...$ |

This notation is interpreted by substituting the components of the standard model $B$ of $\Sigma_{Boolean_s}$ into the relevant carriers, constants and operations. So, the new $\Sigma_{New}$ algebra is simply:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$New_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$B$</td>
</tr>
</tbody>
</table>
| constants | $tt, ff : \rightarrow B$  
$..., C : \rightarrow A_s, ...$ |
| operations| $Not : B \rightarrow B$  
$And : B \times B \rightarrow B$  
$..., Eq_s : A_s \times A_s \rightarrow B, ...$  
$..., If_s : B \times A_s \times A_s \rightarrow A_s, ...$  
$..., F : A_{s[1]} \times \cdots \times A_{s[n]} \rightarrow A_s, ...$ |

Clearly, the algebra $A_{Boolean_s}$ is a reduct of $A_{New_0}$ i.e.,

$$A_{New_0}|_{\Sigma_{Boolean_s}} = A_{Boolean_s}.$$  

4.7.2 Importing a Data Type in General

It is not difficult to see how the construction of importing the Booleans can be adapted to import other basic data types, such as the natural numbers or the real numbers, and indeed how it can also be generalised to import any data type. To import any data type, the construction is in two stages: on signatures and on algebras.
Old Signature/Interface  Suppose we wish to construct the signature \( \Sigma_{New} \) by importing
the signature \( \Sigma_{Old} \):

\[
\begin{align*}
\text{signature} & \quad Old \\
\text{sorts} & \quad \ldots, s_{old}^{old}, \ldots \\
\text{constants} & \quad \ldots, c_{old} : \rightarrow s_{old}^{old}, \ldots \\
\text{operations} & \quad \ldots, f_{old} : s_{1}^{old} \times \cdots \times s_{n}^{old} \rightarrow s_{old}^{old}, \ldots 
\end{align*}
\]

New Signature/Interface  The notation for signatures with imports we use has the general form:

\[
\begin{align*}
\text{signature} & \quad New \\
\text{import} & \quad Old \\
\text{sorts} & \quad \ldots, s_{new}^{new}, \ldots \\
\text{constants} & \quad \ldots, c_{new} : \rightarrow s, \ldots \\
\text{operations} & \quad \ldots, f_{new} : s(1) \times \cdots \times s(n) \rightarrow s, \ldots 
\end{align*}
\]

Now the idea is that \( \Sigma_{New} \) contains all the sorts, constants and operations of \( \Sigma_{Old} \),
together with some new sorts \( \ldots, s_{new}^{new}, \ldots \), and constants and operations involving, possibly,
both old and new sorts from \( \ldots, s_{old}^{old}, \ldots \) and \( \ldots, s_{new}^{new}, \ldots \). It is very
convenient to make the following assumption:

Assumption  The sort names are actually new, i.e.,

\[
\{\ldots, s_{old}^{old}, \ldots\} \cap \{\ldots, s_{new}^{new}, \ldots\} = \emptyset.
\]

The declarations of the new constants and functions can use either old sorts from \( \Sigma_{Old} \) or
new sorts from \( \Sigma_{New} \):

\[
s, s(1), \ldots, s(n), \ldots \in \{\ldots, s_{old}^{old}, \ldots\} \cup \{\ldots, s_{new}^{new}, \ldots\}.
\]

Flattening

What exactly is this new signature with the import construct? The line

\[
\text{import Old}
\]
means that the signature above is an abbreviation; it abbreviates the signature formed by substituting the sorts, constants and operations of $\Sigma_{Old}$ as follows:

\[
\begin{array}{|l|}
\hline
\text{signature} & New_0 \\
\text{sorts} & \ldots, s^{old}_1, \ldots \\
& \ldots, s^{new}, \ldots \\
\text{constants} & \ldots, c_{old} : \rightarrow s^{old}, \ldots \\
& \ldots, c_{new} : \rightarrow s, \ldots \\
\text{operations} & \ldots, f^{old}_1 : s^{old}_1 \times \ldots \times s^{old}_n \rightarrow s^{old}, \ldots \\
& \ldots, f_{new} : s(1) \times \ldots \times s(n) \rightarrow s, \ldots \\
\hline
\end{array}
\]

Clearly, $\Sigma_{New_0}$ is an expansion of $\Sigma_{Old}$.

The removal of \textbf{import} by means of substitution is called \textit{flattening}.

\textbf{Old Algebra/Implementation} To complete the construction, we must define an algebra $A_{New}$ of signature $\Sigma_{New}$ from an algebra $A_{Old}$ of signature $\Sigma_{Old}$.

Let us suppose that we interpret the old signature $\Sigma_{Old}$ with an algebra $A_{Old}$:

\[
\begin{array}{|l|}
\hline
\text{algebra} & Old \\
\text{carriers} & \ldots, A^{old}, \ldots \\
\text{constants} & \ldots, C_{old} : \rightarrow s^{old}, \ldots \\
\text{operations} & \ldots, F_{old} : s^{old}_1 \times \ldots \times s^{old}_n \rightarrow s^{old}, \ldots \\
\hline
\end{array}
\]

\textbf{New Algebra/Implementation} Let $A_{New}$ be constructed from $A_{Old}$ by:

\[
\begin{array}{|l|}
\hline
\text{algebra} & New \\
\text{import} & A_{Old} \\
\text{carriers} & \ldots, A^{new}, \ldots \\
\text{constants} & \ldots, C_{new} : \rightarrow A_s, \ldots \\
\text{operations} & \ldots, F_{new} : A_{s(1)} \times \ldots \times A_{s(n)} \rightarrow A_s, \ldots \\
\hline
\end{array}
\]
Flattening

Again, this notation means that the contents of $A_{\text{Old}}$ is to be substituted. This then gives us the algebra:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$New_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$\ldots, A_{s\text{old}}, \ldots$ $\ldots, A_{s\text{new}}, \ldots$</td>
</tr>
<tr>
<td>constants</td>
<td>$\ldots, C_{\text{old}} : \rightarrow A_{s\text{old}}, \ldots$ $\ldots, C_{\text{new}} : \rightarrow A_{s}, \ldots$</td>
</tr>
<tr>
<td>operations</td>
<td>$\ldots, F_{\text{old}} : A_{s\text{old}} \times \cdots \times A_{s\text{old}} \rightarrow A_{s\text{old}}, \ldots$ $\ldots, F_{\text{new}} : A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow A_{s}, \ldots$</td>
</tr>
</tbody>
</table>

Clearly, on substituting for the import construct, we get the reduct

$$A_{New_0} \mid_{\Sigma_{\text{Old}}} = A_{\text{Old}}.$$  

Thus $\Sigma_{\text{Old}}$ will be imported into $\Sigma_{\text{New}}$, or $\Sigma_{\text{New}}$ will inherit $\Sigma_{\text{Old}}$; and, similarly, $A_{\text{Old}}$ will be imported into $A_{\text{New}}$, or $A_{\text{New}}$ will inherit $A_{\text{Old}}$.

### 4.7.3 Example

Consider the following signature for computing with real numbers:

<table>
<thead>
<tr>
<th>signature</th>
<th>Reals with Integer Rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>Booleans, Integers, Reals</td>
</tr>
<tr>
<td>sorts</td>
<td></td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>$\text{less} : \text{real} \times \text{real} \rightarrow \text{bool}$ $\text{include} : \text{int} \rightarrow \text{real}$ $\text{round_up} : \text{real} \rightarrow \text{int}$</td>
</tr>
</tbody>
</table>

Given the specific signatures

$$\Sigma_{\text{Booleans}}, \Sigma_{\text{Integers}} \text{ and } \Sigma_{\text{Reals}}$$  

this defines a new signature

$$\Sigma_{\text{Reals with Integer Rounding}}$$
by combining all the sorts, constants and operations, and adding the new ones declared.

Flattening the notation for $\Sigma_{\text{Reals with Integer Rounding}}$, we get the following expansion

$$
\Sigma_{\text{Reals with Integer Rounding}}_0
$$

of $\Sigma_{\text{Booleans}}, \Sigma_{\text{Integers}}$ and $\Sigma_{\text{Reals}}$:

<table>
<thead>
<tr>
<th>signature</th>
<th>$\text{Reals with Integer Rounding}_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>bool</td>
</tr>
<tr>
<td>constants</td>
<td>true, false : $\rightarrow$ bool</td>
</tr>
<tr>
<td></td>
<td>zero, one : $\rightarrow$ int</td>
</tr>
<tr>
<td></td>
<td>zero$<em>{\text{real}}$, one$</em>{\text{real}}$, pi, e : $\rightarrow$ real</td>
</tr>
<tr>
<td>operations</td>
<td>not : $\text{bool} \rightarrow$ $\text{bool}$</td>
</tr>
<tr>
<td></td>
<td>and : $\text{bool} \times$ $\text{bool} \rightarrow$ $\text{bool}$</td>
</tr>
<tr>
<td></td>
<td>add : $\text{int} \times$ $\text{int} \rightarrow$ $\text{int}$</td>
</tr>
<tr>
<td></td>
<td>minus : $\text{int} \rightarrow$ $\text{int}$</td>
</tr>
<tr>
<td></td>
<td>times : $\text{int} \times$ $\text{int} \rightarrow$ $\text{int}$</td>
</tr>
<tr>
<td></td>
<td>add$_{\text{real}}$ : $\text{real} \times$ $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>minus$_{\text{real}}$ : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>times$_{\text{real}}$ : $\text{real} \times$ $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>invert : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>exp : $\text{real} \times$ $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>log : $\text{real} \times$ $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>sqrt : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>abs : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>sin : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>cos : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>tan : $\text{real} \rightarrow$ $\text{real}$</td>
</tr>
<tr>
<td></td>
<td>round-up : $\text{real} \rightarrow$ $\text{int}$</td>
</tr>
</tbody>
</table>

Similarly, given algebras $A_{\text{Booleans}}, A_{\text{Integers}}$ and $A_{\text{Reals}}$

interpreting the three imported signatures, the new $\Sigma_{\text{Reals with Integer Rounding}}$-algebra

$$
A_{\text{Reals with Integer Rounding}}_0
$$

is defined by combining all the carriers, constants and operations, and adding the order relation, the sort conversion of integer to real, and the ceiling function.

This construction has a simple architecture as shown in Figure 4.3:
Figure 4.3: Architecture of the algebra $A_{\text{Reals with Integer Rounding}}$
4.7. IMPORTING ALGEBRAS

Exercises for Chapter 4

1. List all the product types and operation types over the two sorts in $S = \{\text{nat, Bool}\}$.
   Why do we assume
   \[
   \text{nat} \times \text{Bool} \quad \text{and} \quad \text{Bool} \times \text{nat}
   \]
   are different product types? Devise formulae that count the number of two-sorted product
   and operation types of arity $n$.

2. Devise formulae that count the number of $m$-sorted product types and operation types
   of arity $n$.

3. Using the provisional definition of an algebra from Section 3.1, are the algebras
   \[
   (\text{N;} 0; n + 1) \quad \text{and} \quad (\text{N;} 0; n + 1, n + 1)
   \]
   the same algebras or not? Recast the definitions of these algebras using the formal
   definition of an algebra in Section 4.1. How do signatures affect the Counting Lemma in
   Section 3.2.1?

4. Let $\Sigma$ be a single-sorted signature with sort $s$ and sets $\Sigma_{s^k,s}$ of $k$-ary operation symbols,
   where $k = 0, 1, 2, \ldots, K$ and $K$ is the maximum arity. Show that the number of $\Sigma$-algebras
   definable on a carrier set $X$ with cardinality $|X| = n$ is
   \[
   \prod_{k=0}^{K} n^{k \cdot |\Sigma_{s^k,s}|}.
   \]
   The signature $\Sigma_{\text{Ring}}$ of a ring contains two constants 0 and 1, one unary operation $-$,
   and two binary operations $+$ and $\cdot$. How many $\Sigma_{\text{Ring}}$-algebras are definable on a carrier
   set $X$ with $n$-elements? Estimate how many $\Sigma_{\text{Ring}}$-algebras satisfy the axioms for a
   commutative ring.

5. Let $\Sigma$ be a two-sorted signature with sorts $s_1$ and $s_2$. Construct a formula for the number
   of $\Sigma$-algebras definable on sets $X_1$ and $X_2$ of cardinality $|X_1| = n_1$ and $|X_2| = n_2$. Hence,
   give a formula for the special case $s_1 = s$, $s_2 = \text{Bool}$, $|X_1| = n$ and $X_2 = \{tt, ff\}$.

6. Write down signatures to model interfaces for the following algebras of bits:
   
   a. bits;
   
   b. bytes; and
   
   c. n-bit words.

7. Write down signatures to model interfaces for the following algebras of numbers:
   
   a. rational numbers; and
   
   b. complex numbers.

8. Write down a signature to model the interface for the data of Babbage’s 1835 design for
   Analytical Engine.
9. Write down signatures to model interfaces for the following data structures:
   a. the array;
   b. the list; and
   c. the stack.

10. A data type interface is modelled by a signature. Model an idea of ”linking” two different interfaces by formulating conditions on mappings between signatures.

11. Use algebras to model the design of some of the data types in:
   (i) a pocket calculator; and
   (ii) a programming system of your choice.

12. Let $A$, $B$ and $C$ be any $S$-sorted $\Sigma$-algebras. Prove that if $B \leq A$, $C \leq A$ and $B \subset C$ then $B \leq C$.

13. We expand the algebra of Section 4.6.2 for integer addition with multiplication. Let $\Sigma$ be a signature for the algebra
    \[ A = (\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z}, \cdot^\mathbb{Z}). \]
    Show that for any $n \geq 1$,
    \[ B = (n\mathbb{Z}; 0^{n\mathbb{Z}}, +^{n\mathbb{Z}}, -^{n\mathbb{Z}}, \cdot^{n\mathbb{Z}}) \]
    is a $\Sigma$-subalgebra of $A$.

14. Show that the operations of $A = (\mathbb{Z}; 0^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z})$ are not constructors. What must be added to $A$ to equip it with a set of constructors?

15. Show that the algebra
    \[ (\mathbb{N}; 0; n + 1) \]
    has no proper subalgebras. What are the subalgebras of
    \[ (\mathbb{N}; 0; n \div 1)? \]

16. Let $A = (\mathbb{R}; 0; +, -)$ be the algebra of real number addition and subtraction. What are the carriers of the following subalgebras:
   a. $\langle 0 \rangle$;
   b. $\langle 1 \rangle$;
   c. $\langle 1, 2 \rangle$;
   d. $\langle \sqrt{2} \rangle$; and
   e. $\langle \pi \rangle$?

17. Let $A = (\mathbb{R}; 0, 1; +, -, \cdot, ^{-1})$ be the algebra of real number addition, subtraction, multiplication and division. What are the carriers of the following subalgebras:
   a. $\langle 0, 1 \rangle$;
4.7. Importing Algebras

b. \( \langle 2 \rangle \);
c. \( \langle \frac{1}{2} \rangle \);
d. \( \langle \sqrt{2} \rangle \); and
e. \( \langle \pi \rangle \)?

18. Let \( A = \{a, b, aa, ab, ba, bb, \ldots \} ; c, \circ \} \) be the algebra of all strings over \( a, b \) with concatenation. Which of the following sets form subalgebras of \( A \):

a. \( \{a^n \mid n \geq 0\} \);
b. \( \{b^n \mid n \geq 0\} \);
c. \( \{(ab)^n \mid n \geq 0\} \);
d. \( \{a^{2n} \mid n \geq 0\} \);
e. \( \{b^{3n} \mid n \geq 0\} \);

19. Let \( GL(2, \mathbb{R}) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid ad - bc \neq 0 \right\} \)
be the set of non-singular \( 2 \times 2 \) matrices with real number coefficients. This set is closed under matrix multiplication and matrix inversion, so forms an algebra

\[ A = (GL(2, \mathbb{R}); 1,.,^{-1}) \]

where

\[ 1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \]
is the identity matrix. Which of the following sets form subalgebras:

a. \( \left\{ \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \mid b \in \mathbb{Z} \right\} \);
b. \( \left\{ \begin{pmatrix} a & b \\ 0 & d \end{pmatrix} \mid ad \neq 0 \right\} \)?

20. When working with a particular class \( K \) of algebras, it is often important that

if \( A \in K \) and \( B \) is a subalgebra of \( A \), then \( B \in K \).

**Definition** A class \( K \) of \( \Sigma \)-algebras is said to be closed under the formation of subalgebras if, and only if, whenever \( A \in K \) and \( B \subseteq A \) then \( B \in K \).

a. Are the classes of semigroups, groups, rings and fields closed under the formation of subalgebras?
b. Is any class of algebras defined by equations closed under the formation of subalgebras?
c. Is the class of all finite structures of any signature \( \Sigma \) closed under the formation of subalgebras?
21. Using the import notation, redefine the following signatures and algebras:
   
   a. the standard model of Peano arithmetic with the Booleans (see Section 3.3.1);
   b. the real numbers with the Booleans (see Section 3.5); and
   c. strings with length (see Section 3.7).

22. In the general account of import how restrictive is the assumption that the sorts, constants and operation symbols must be new? Give an interpretation of import without the assumption, illustrating your answer with examples.
Chapter 5
Specifications and Axioms

We are developing the idea of a data type in several stages. In this chapter we reach the third stage. We will add a new component to the concept of both data type and algebra, namely, the programming idea of a specification

and the corresponding mathematical idea of an axiomatic theory to model it. In the context of data types, the term axiomatic theory is renamed axiomatic specification

In the second stage of developing our idea of a data type, we revised our idea of a data type by introducing these two aspects,

\[ \text{Data Type} = \text{Interface} + \text{Implementation} \]

Whilst the names of the data and the operations can be fixed by declaring an interface for a data type, there will always be considerable variation in the details of how the data and operations are implemented.

In the third stage, we reflect on this variation of implementations. We need precise criteria for data type implementations to be either acceptable or unacceptable. We answer the following question:

Specification Problem

How can we specify the properties of a data type for its users?

The user communicates with the data type via its interface which consists of operations. One solution is to list some of the algebraic properties of the operations in the interface that any acceptable implementation of the data type must possess. The algebraic criteria for acceptable data type implementations form a specification and we propose that for a data type:

\[ \text{Specification} = \text{Interface} + \text{Properties of Operations}. \]
For example, given names for operations on integers, we can require that any implementation must satisfy the basic laws of arithmetic, like associativity and commutativity, and perhaps some conditions on overflow.

In the mathematical model, the signature fixes a notation, and the interpretation models a choice for the data representation and the input-output behaviour of the algorithms implementing the operations of the data type. To analyse mathematically the diversity of representations and implementations we will postulate a list of algebraic properties of the constants and operations in the signature. We model data type specifications using the mathematical idea of an axiomatic theory which has the form,

\[ \text{Axiomatic Theory} = \text{Signature} + \text{Axioms}. \]

An algebra satisfies a theory if the axioms are true of its operations named in the signature. The specification of data types is a deep subject with huge scope. In this book we are merely pointing out its existence; for some information and guidance, see the Further Reading.

In Section 5.1, we take up the idea that interfaces have many useful implementations that form classes of algebras of common signatures defined by axiomatic specifications. We reflect on how an axiomatic specification provides a foundation for reasoning about classes of implementations.

We examine the fact that some properties can be proved true of all implementations of a specification, whilst others cannot, and must be added to a specification, if desired. In Section 5.2, we begin with a simple example of a class of implementations of the integers.

For the rest of the chapter, in Sections 5.3, 5.4, 5.5 and 5.6, we meet some axiomatic specifications of data types and examine some of their uses. To begin with, we are interested in specifying and reasoning with the data types of the integer, rational and real numbers. These systems have been studied in great depth by mathematicians over the centuries. Indeed, it is through the study of number systems, ideas and methods have emerged that can be applied to any data. Specifically, we look at the mathematical ideas of the commutative ring, field and group from the point of view of data type specifications, and at equation-solving in these kinds of algebras.

### 5.1 Classes of Algebras

It is usual in using or designing a data type that we end up with not one algebra, but a whole class of algebras that satisfy a range of design criteria. The class of algebras has a common interface with computations, namely its signature. It ought to have some standard properties that all algebras possess. The signature plays a fundamental rôle in

1. making precise the concept of an algebra as a model of an implementation;
2. defining classes of algebras with common properties;
3. proving or disproving further properties of operations of algebras; and
4. comparing algebras.
5.1. CLASSES OF ALGEBRAS

5.1.1 One Signature, Many Algebras

The usual situation when modelling data is that a signature $\Sigma$ is proposed and several $\Sigma$-algebras $A, B, C, \ldots$ are constructed as depicted in Figure 5.1. The signature models an interface and the different algebras model some different implementations for the interface. Commonly, given a signature $\Sigma$, there are infinitely many $\Sigma$-algebras of interest.

\begin{center}
\begin{tikzpicture}
  \node (Sigma) at (0,0) {$\Sigma$};
  \node (A) at (-1,-1) {$A$};
  \node (B) at (0,-1) {$B$};
  \node (C) at (1,-1) {$C$};
  \draw (Sigma) -- (A);
  \draw (Sigma) -- (B);
  \draw (Sigma) -- (C);
  \node at (1.5,0) {one signature-interface};
  \node at (-1.5,0) {many different algebras/implementations};
\end{tikzpicture}
\end{center}

Figure 5.1: One interface, many implementations. One signature, many algebras.

For example, the signature $\Sigma_{\text{subsets}}$ serves as an interface to an algebra of subsets of any chosen set; and the signature $\Sigma_{\text{Basic Strings with Length}}$ serves as an interface to an algebra of strings over any chosen alphabet.

**Definition (All $\Sigma$-algebras)** Let $\Sigma$ be any signature. Let 

$$\text{Alg}(\Sigma)$$

be the class of all algebras with signature $\Sigma$.

Very rarely are we interested in all $\Sigma$-algebras. A signature $\Sigma$ is designed for a purpose and $\text{Alg}(\Sigma)$ will contain many algebras that are irrelevant to that purpose. Usually, given a signature $\Sigma$, we are interested in finding a relatively small subclass

$$K \subseteq \text{Alg}(\Sigma)$$

of representations or implementations of $\Sigma$. To isolate and explore a subclass $K$ we must find and postulate relevant properties of $\Sigma$-algebras that we require to be true of all the algebras in $K$.

**Definition (Axiomatic Specification)** Let $\Sigma$ be any signature. Let $T$ be any set of properties of $\Sigma$-algebras. The pair $(\Sigma, T)$ is called an *axiomatic theory* or *axiomatic specification*. Let

$$\text{Alg}(\Sigma, T)$$

be the class of all $\Sigma$-algebras satisfying all the properties in $T$.

In the case that a class $K$ of $\Sigma$-algebras satisfies the properties of $T$, we have

$$K \subseteq \text{Alg}(\Sigma, T)$$

and this is depicted in Figure 5.2.

The axiomatic theory or specification $(\Sigma, T)$ performs three tasks when modelling data.
Restriction The properties in $T$ limit and narrow the range of implementation models, since algebras failing to satisfy any property in $T$ are discarded.

Standardisation The theory $T$ establishes some basic properties demanded of all implementations, and it equips $\Sigma$ with properties that are known to be independent of all implementations.

Analysis and Verification The theory $T$ is a basis from which to prove or disprove further properties of the operations in $\Sigma$ and the implementations.

5.1.2 Reasoning and Verification

Let us consider the last task of verification.

Let $(\Sigma, T)$ be an axiomatic specification. Consider some property $P$ based on the operations in $\Sigma$. For example, $P$ might be an equation

$$t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$$

where $t(x_1, \ldots, x_n)$ and $t'(x_1, \ldots, x_n)$ are terms formed by composing the operations in $\Sigma$ and applying them to variables $x_1, \ldots, x_n$. Or $P$ might be the existence of a solution to an equation

$$(\exists x_1, \ldots, x_n)[t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)].$$

Or $P$ might be the correctness

$$\{p\} S\{q\}$$

of a program $S$ with respect to an input condition $p$ and an output condition $q$.

Suppose the property $P$ can be proved using

(i) the axioms in $T$; and

(ii) general principles of logical reasoning.

Then we expect that the property $P$ will be true of every $\Sigma$-algebra that satisfies all the axioms in $T$. This expectation we will express as an informal principal of reasoning:
### 5.2. Classes of Algebras Modelling Implementations of the Integers

Consider the data type of the integers with a simple set of operations. An interface for the integer data type is modelled by the following signature $\Sigma_{\text{Integers}}$:

<table>
<thead>
<tr>
<th>signature</th>
<th>Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>int</td>
</tr>
<tr>
<td>constants</td>
<td>zero : $\rightarrow$ int</td>
</tr>
<tr>
<td>operations</td>
<td>add : int $\times$ int $\rightarrow$ int</td>
</tr>
<tr>
<td></td>
<td>minus : int $\rightarrow$ int</td>
</tr>
<tr>
<td></td>
<td>times : int $\times$ int $\rightarrow$ int</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>
The data type of integers allows many implementations. If each implementation is an algebra with signature $\Sigma_{\text{Integers}}$ then the signature can be interpreted by many algebras modelling these implementations. To explore the data type of integers we must explore a class of algebras, each algebra having the above signature $\Sigma_{\text{Integers}}$, i.e., a class

$$K \subseteq \text{Alg}(\Sigma_{\text{Integers}}).$$

In Section 5.3, we will give a set of axioms that defines such a class $K$.

**Example 1: Standard Model of the Integers**

An obvious $\Sigma_{\text{Integers}}$-algebra is the usual algebra

$$(\mathbb{Z}; 0; x + y, -x, x, y)$$

built from the infinite set

$$\mathbb{Z} = \{ \ldots, -2, -1, 0, 1, 2, \ldots \}$$

of integers and equipped with the standard operations of $+$, $-$, and $\cdot$ on the integers. Indeed, let us note that this algebra of integers is the arithmetic of all users. There are others. The set of integers is infinite and must be approximated in machine computations by a finite subset.

Consider the finite sets

$$\{0, 1, 2, \ldots, n - 1\}$$

and

$$\{-M, \ldots, -1, 0, 1, \ldots, +M\}$$

consisting of initial segments of the integers. What operations for arithmetic can be defined on them? We want to build useful operations that can interpret those of the signature $\Sigma_{\text{Integers}}$. There are some interesting complications in even the high-level designs for algebras modelling implementations of the integers; see Figure 5.3. We will explore these examples here and in the

<table>
<thead>
<tr>
<th>Interface Model</th>
<th>signature $\Sigma_{\text{Integers}}$ for integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf</td>
<td>finite integers</td>
</tr>
<tr>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$\forall$</td>
<td>$\forall$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Integer Model</th>
<th>integer base $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf</td>
<td>inf</td>
</tr>
<tr>
<td>$\forall$</td>
<td>$\forall$</td>
</tr>
<tr>
<td>Max-Min</td>
<td>Overflow</td>
</tr>
</tbody>
</table>

Figure 5.3: Classes of integer implementations.

exercises.

Again, the precise details of the representations of the numbers in these algebras have not been explained. The data may be written in binary, decimal or any other number base $b$ and the functions defined accordingly. This observation contributes infinitely many more algebras to the class of possible semantics for the integers (one for each base $b$).
5.2. CLASSES OF ALGEBRAS MODELLING IMPLEMENTATIONS OF THE INTEGERS

Example 2: Cyclic Arithmetic

In cyclic arithmetic, we take the initial segment

\[ Z_n = \{0, 1, 2, \ldots, n - 1\} \]

and arrange that the successor of the maximum element \( n - 1 \) is the minimum 0. Thus, counting is circular as shown in Figure 5.4.

\[ \begin{array}{c}
\text{Figure 5.4: Cyclic arithmetic}
\end{array} \]

Some functions on the integers \( Z \) are adapted to \( Z_n \) by applying the modulus function

\[ Mod_n : Z \to Z_n \]

defined by

\[ Mod_n(x) = x \mod n \]

= remainder on dividing \( x \) by \( n \)

Consider the \( \Sigma_{\text{Integer}} \) algebra of integers whose operations are modulo \( n \) arithmetic,

\[ Z_n = (\{0, 1, 2, \ldots, n - 1\}; 0, +_n, -_n, \cdot_n) \]

The operations \( +_n, -_n \) and \( \cdot_n \) are derived from +, − and · on \( Z \) as follows: for \( x, y \in \{0, 1, \ldots, n - 1\}, \)

\[ \begin{array}{c}
+_n x +_n y = x + y \mod n \\
-_n x = n - (x \mod n) \\
\cdot_n x \cdot y = x \cdot y \mod n
\end{array} \]

This choice leads to an infinite family of algebras, one for each choice of \( n \).

For example, take \( n = 5 \). The operations have the following tables:

\[ \begin{array}{c|cccccc}
  +_5 & 0 & 1 & 2 & 3 & 4 \\
  \hline
  0 & 0 & 1 & 2 & 3 & 4 \\
  1 & 1 & 2 & 3 & 4 & 0 \\
  2 & 2 & 3 & 4 & 0 & 1 \\
  3 & 3 & 4 & 0 & 1 & 2 \\
  4 & 4 & 0 & 1 & 2 & 3
\end{array} \]

\[ \begin{array}{c|cccccc}
  -_5 & 0 & 0 & 0 & 0 & 0 \\
  \hline
  0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 1 & 2 & 3 & 4 \\
  2 & 1 & 0 & 2 & 4 & 1 \\
  3 & 2 & 2 & 0 & 3 & 1 \\
  4 & 3 & 1 & 4 & 2 & 0
\end{array} \]

\[ \begin{array}{c|cccccc}
  \cdot_5 & 0 & 0 & 0 & 0 & 0 \\
  \hline
  0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 1 & 2 & 3 & 4 & 0 \\
  2 & 2 & 3 & 4 & 0 & 1 \\
  3 & 3 & 4 & 0 & 1 & 2 \\
  4 & 4 & 0 & 1 & 2 & 3
\end{array} \]

The standard infinite integers \( Z \) and the modular arithmetic \( Z_n \) both have mathematical elegance.
Example 3: Algebras that are not the integers

There are algebras $A \in Alg(\Sigma_{\text{Integers}})$ that do not model the integers.

Suppose we interpret the signature $\Sigma_{\text{Integer}}$ in a trivial way as follows. Let $A$ be any non-empty set, and choose some $a \in A$. We define the constants and operations in $\Sigma_{\text{Integer}}$ as follows: for $x, y \in A$,

\[
\begin{align*}
\text{zero}^A &= a \\
\text{add}^A(x, y) &= a \\
\text{minus}^A(x, y) &= a \\
\text{times}^A(x, y) &= a
\end{align*}
\]

The resulting algebra $A \in Alg(\Sigma_{\text{Integers}})$ does not qualify as a useful implementation of the integers, of course.

Other obvious examples are algebras of rational numbers

\[Q = (\mathbb{Q}; 0; +, -) \in Alg(\Sigma_{\text{Integers}})\]

and real numbers

\[R = (\mathbb{R}; 0; +, -) \in Alg(\Sigma_{\text{Integers}})\]

Although these algebras are not models of the integers, they have very similar properties.

Next we will consider specifications to remove unwanted models and implementations from $Alg(\Sigma_{\text{Integers}})$.

5.3 Axiomatic Specification of Commutative Rings

The standard algebra $\mathbb{Z}$ of integers has many convenient and useful algebraic properties, shared by the rational and real numbers. In particular, these number systems satisfy some simple equations which together form the set of laws or axiomatic specification for the class of algebras called the

\[\text{commutative rings with identity.}\]

5.3.1 The Specification

Here is a new, abstract signature to capture the basic operations of interest.

\[
\begin{align*}
\text{signature} & \quad CRing \\
\text{sorts} & \quad \text{ring} \\
\text{constants} & \quad 0, 1 : \text{ring} \\
\text{operations} & \quad + : \text{ring} \times \text{ring} \to \text{ring} \\
& \quad - : \text{ring} \to \text{ring} \\
& \quad \cdot : \text{ring} \times \text{ring} \to \text{ring}
\end{align*}
\]
Let us postulate that these operations satisfy the following laws or axioms.

<table>
<thead>
<tr>
<th>Axioms</th>
<th>$\mathcal{CRing}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associativity of addition</td>
<td>$(\forall x)(\forall y)(\forall z)[(x + y) + z = x + (y + z)]$ (1)</td>
</tr>
<tr>
<td>Commutativity for addition</td>
<td>$(\forall x)(\forall y)[x + y = y + x]$ (2)</td>
</tr>
<tr>
<td>Identity for addition</td>
<td>$(\forall x)[x + 0 = x]$ (3)</td>
</tr>
<tr>
<td>Inverse for addition</td>
<td>$(\forall x)[x + (-x) = 0]$ (4)</td>
</tr>
<tr>
<td>Associativity for multiplication</td>
<td>$(\forall x)(\forall y)(\forall z)[(x.y).z = x.(y.z)]$ (5)</td>
</tr>
<tr>
<td>Commutativity for multiplication</td>
<td>$(\forall x)(\forall y)[x.y = y.x]$ (6)</td>
</tr>
<tr>
<td>Identity for multiplication</td>
<td>$(\forall x)[x.1 = x]$ (7)</td>
</tr>
<tr>
<td>Distribution</td>
<td>$(\forall x)(\forall y)(\forall z)[x.(y + z) = x.y + x.z]$ (8)</td>
</tr>
</tbody>
</table>

Let $T_{\mathcal{CRing}}$ denote the set of eight axioms for commutative rings. The pair $(\Sigma_{\mathcal{CRing}}, T_{\mathcal{CRing}})$ is an axiomatic specification of the integers.

Let $\text{Alg}(\Sigma_{\mathcal{CRing}}, T_{\mathcal{CRing}})$ denote the class of all $\Sigma_{\mathcal{CRing}}$-algebras that satisfy all the axioms in $T_{\mathcal{CRing}}$.

**Definition** A $\Sigma_{\mathcal{CRing}}$-algebra $A$ is defined to be a *commutative ring* if, and only if, it satisfies the axioms above, i.e., $A \in \text{Alg}(\Sigma_{\mathcal{CRing}}, T_{\mathcal{CRing}})$.

Clearly the axioms are abstracted from the familiar properties of integer, rational and real arithmetic. Hence, the standard $\Sigma_{\mathcal{CRing}}$ algebras

$\mathbb{Z}, \mathbb{Q}$ and $\mathbb{R} \in \text{Alg}(\Sigma_{\mathcal{CRing}}, T_{\mathcal{CRing}})$.

More surprisingly, and rather usefully:

**Theorem** For any $n \geq 2$, the cyclic arithmetic

$\mathbb{Z}_n \in \text{Alg}(\Sigma_{\mathcal{CRing}}, T_{\mathcal{CRing}})$.

### 5.3.2 Deducing Further Laws

The axioms for a commutative ring with unity are a foundation on which to build general methods for calculating and reasoning with number systems independently of their representations.
To give a flavour of abstract calculation and reasoning, we will derive some further laws and techniques for equation solving.

We will prove some simple laws and identities from the axioms to demonstrate how algebraic laws can be deduced.

**Lemma** Let $A$ be a commutative ring with unit. For any $x, y \in A$, the following conditions hold:

(i) $(x + y).z = x.z + y.z$;

(ii) $0.x = x.0 = 0$;

(iii) $x.(-y) = (-x).y = -(x.y)$; and

(iv) $(-x).(-y) = x.y$.

**Proof** The deductions are as follows.

(i) 
\[
(x + y).z = z.(x + y) \quad \text{by Axiom 6;}
\]
\[
= z.x + z.y \quad \text{by Axiom 8;}
\]
\[
= x.z + y.z \quad \text{by Axiom 6.}
\]

(ii)
\[
0 + 0 = 0 \quad \text{by Axiom 3;}
\]
\[
(0 + 0).x = 0.x \quad \text{by part (i) of this Lemma;}
\]
\[
0.x + 0.x = 0.x \quad \text{by part (i) of this Lemma;}
\]
\[
(0.x + 0.x) + (-0.x) = 0.x + (-0.x) \quad \text{by Axioms 1 and 4;}
\]
\[
0.x + (0.x + (-0.x)) = 0 \quad \text{by Axiom 4;}
\]
\[
0.x + 0 = 0 \quad \text{by Axiom 3.}
\]

We leave cases (iii) and (iv) to the exercises. □

To begin with such calculations and derivations are long and slow. To proceed, we need to simplify notation so that arguments can be shorter and seem more familiar.

**Parentheses**

Axioms 1 and 2 allow us to drop brackets in summing a list of elements. For example, we can write simply
\[
x + y + z
\]
for any of
\[
(x + y) + z, \quad x + (y + z), \quad (x + z) + y, \quad x + (z + y),
\]
\[
(y + x) + z, \quad y + (x + z), \quad (y + z) + x, \quad y + (z + x),
\]
\[
(z + y) + x, \quad z + (y + x), \quad (z + x) + y, \quad z + (x + y),
\]
etc.

Another convention that is common is to drop the use of the multiplication symbol $\cdot$ in expressions. For example, we can write
\[
xy + yz + zx
\]
for
\[ x, y + y, z + z, x \]
since the 0 is easily inferred from the expression.

**Formal Integers**
If 1\(_A\) is the unit element in a commutative ring \(A\), then we can denote
\[ 1_A, 1_A + 1_A, 1_A + 1_A + 1_A, 1_A + 1_A + 1_A + 1_A, \ldots \]
by the familiar notation,
\[ 1_A, 2_A, 3_A, 4_A, \ldots \]
or, more simply,
\[ 1, 2, 3, 4, \ldots \]
For example, the notation
\[ 5 = 5_A = 1_A + 1_A + 1_A + 1_A + 1_A \]
when interpreted in \(\mathbb{Z}\) is 5, but in \(\mathbb{Z}_3\) it is 2, \(\mathbb{Z}_4\) it is 1, \(\mathbb{Z}_5\) it is 0 and in \(\mathbb{Z}_n\) for \(n > 5\) it is 5. It is vital to remember that if one sees \(5.x\) then this denotes
\[ (1_A + 1_A + 1_A + 1_A + 1_A).x, \]
i.e., the element of the ring formed by adding the unit to itself 5 times and multiplying by \(x\).

**Definition (Formal Integers)** The elements of the form
\[ 1_A + 1_A + \cdots + 1_A \]
are called *formal integers*.

We will make heavy use of them in working on the data type of real numbers in Chapter 8.

**Polynomials and Factorisation** With these conventions, expressions made from the operations of \(\Sigma_{CRing}\) are made simple and familiar. However, their meaning in a particular \(\Sigma_{CRing}\)-algebra may be complex and surprising. The axioms allow us to work abstractly with operations over a whole class \(Alg(\Sigma_{CRing}, T_{CRing})\) of interpretations.

Familiar algebraic expressions, associated with the integers and reals, like the polynomials
\[
\begin{align*}
x + 1 & \quad x + 2 & \quad x + 3 \\
x^2 + 3x + 2 & \quad x^2 + 5x + 6 & \quad x^3 + 6x^2 + 11x + 6
\end{align*}
\]
make sense in any commutative ring with unit.

**Definition (Polynomials)** Let \(\Sigma_{CRing}\) be a signature for commutative rings. Let \(x\) be any variable. A *polynomial* in variable \(x\) of degree \(n\) over \(A\) is an expression of the form
\[ a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]
where \(n \geq 0\), and the coefficients \(a_n, a_{n-1}, \ldots, a_1, a_0\) are formal integers, and \(a_n \neq 0\).
Using the axioms and lemmas, we can calculate with these formal polynomials. For example, we can expand algebraic expressions to reduce them to standard forms that are valid in all commutative rings:

\[(x + 1)(x + 2) = x^2 + 3x + 2.\]

The calculation is thus:

\[
(x + 1)(x + 2) = (x + 1)x + (x + 1)2 \quad \text{by Axiom 8;}
\]

\[
= x^2 + 1x + 2x + 2 \quad \text{by Lemma (i), Axiom 6 and conventions;}
\]

\[
= x^2 + (1 + 2)x + 2 \quad \text{by Lemma (i);}
\]

\[
= x^2 + 3x + 2 \quad \text{by convention}
\]

Similarly, we can deduce that

\[(x + 1)(x + 2)(x + 3) = x^3 + 6x^2 + 11x + 6\]

is valid in any commutative ring.

More generally, these calculations are summarised as follows.

**Lemma (Factorisation)** The following identities are valid in any commutative ring with unit:

(i) \( (x + p).(x + q) = x^2 + (p + q).x + pq, \)

(ii) \( (x + p).(x + q).(x + r) = x^3 + (p + q + r).x^2 + (pq + pr + qr).x + pqr \)

where \( p, q \) and \( r \) are formal integers.

**Proof** We calculate using the axioms

(i) \( (x + p).(x + q) = (x + p).x + (x + p).q \quad \text{by Distribution Law (8)}
\]

\[
= x.x + p.x + x.q + p.q \quad \text{by Lemma 5.3.2(i)}
\]

\[
= x^2 + p.x + q.x + p.q \quad \text{by Commutativity (6)}
\]

\[
= x^2 + (p + q).x + pq \quad \text{by Lemma 5.3.2(i)}
\]

(ii) We leave this case as an exercise.

\( \square \)

### 5.3.3 Solving Quadratic Equations in a Commutative Ring

The solution of polynomial equations of the form

\[a_nx^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0 = 0\]

is a central problem of mathematics with a huge and fascinating history. The problem depends on

(i) where the coefficients come from, and

(ii) where the solutions are to be found.
5.3. AXIOMATIC SPECIFICATION OF COMMUTATIVE RINGS

The usual places are the integers $\mathbb{Z}$, reals $\mathbb{R}$ and complex numbers $\mathbb{C}$, of course. We are also interested in $\mathbb{Z}_n$.

Only in the late eighteenth and early nineteenth centuries were there clear theoretical explanations of the problem and techniques of equation-solving. Two theorems stand out. C F Gauss proved the so-called Fundamental Theorem of Algebra:

*Any polynomial equation of degree $n$ with complex number coefficients has $n$ complex number solutions.*

Although there are simple algebraic formulae for finding solutions to polynomial equations of degree $n = 1, 2, 3$ and 4. with complex number coefficients, N H Abel proved the so-called *Unsolvability of the Quintic*:

*No simple algebraic formulae, based on polynomials augmented by $\sqrt{5}$, exist for finding solutions to polynomial equations of degree $n \geq 5$ with complex number coefficients.*

The subject of solving polynomials has had a profound effect on the development of algebra, number theory and geometry and on their many applications.

Here we will use equation-solving to give an impression of working with abstract specifications. We consider the problem *when the coefficients and solutions come from a commutative ring*.

Consider a simple quadratic equation. Given a commutative ring with unit $A$, find all $x \in A$ such that

$$x^2 + 3x + 2 = 0.$$

From the Factorisation Lemma, we know that for every commutative ring with unit,

$$x^2 + 3x + 2 = 0 \iff (x + 1)(x + 2) = 0.$$

Now if $A$ were $\mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$, our next step would be

$$(x + 1)(x + 2) = 0 \iff x + 1 = 0 \text{ or } x + 2 = 0 \iff x = -1 \text{ or } x = -2.$$

However, is this step valid for *all* commutative rings or just *some*?

**Definition** Let $A$ be a commutative ring with unit. If $x, y \in A$ and

$$x \neq 0, \quad y \neq 0 \quad \text{and} \quad x.y = 0$$

then $x$ and $y$ are called *divisors of zero*.

Now the ring $A$ has *no* divisors of zero if, and only if, for any $x, y \in A$

$$x.y = 0 \quad \text{implies} \quad x = 0 \text{ or } y = 0.$$

This is the property of $\mathbb{Z}, \mathbb{Q}$ and $\mathbb{R}$ that we used in the last stage in solving the equation. Is it true of all commutative rings with unit?
**Theorem** The property

\[ (\forall x)(\forall y)[x,y = 0 \Rightarrow x = 0 \lor y = 0] \]

of having no zero divisors is not true of all commutative rings with unit. Hence, the property cannot be proved from the axioms.

**Proof** The property is not true of the ring \( \mathbb{Z}_4 \) of integers modulo 4. The multiplication table of \( \mathbb{Z}_4 \) is:

\[
\begin{array}{c|cccc}
  & 0 & 1 & 2 & 3 \\
\hline
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 2 & 3 \\
2 & 0 & 2 & 0 & 2 \\
3 & 0 & 3 & 2 & 1 \\
\end{array}
\]

Clearly, in \( \mathbb{Z}_4 \), \( 2 \cdot 2 = 0 \) but \( 2 \neq 0 \).

Suppose the property were provable from the axioms. Then it would be true of all algebras satisfying the axioms. In particular, it would be true of \( \mathbb{Z}_4 \). Since it is not true of \( \mathbb{Z}_4 \) we have a contradiction and so the property is not provable. \( \square \)

Here is an illustration of the Soundness Principle and its corollary in action. We find that to proceed with our abstract analysis of quadratic equation-solving, we need to refine the specification by adding an extra necessary property.

**Definition (Integral Domain)** A commutative ring with unit having no zero divisors is called an integral domain.

Let \( T_{\text{IntD}} \) be the set of axioms formed by adding the no zero divisors property to \( T_{\text{CRing}} \). Then we have a new specification \( (\Sigma_{\text{CRing}}, T_{\text{IntD}}) \) and a new class of algebras \( \text{Alg}(\Sigma_{\text{CRing}}, T_{\text{IntD}}) \) as shown in Figure 5.5.

![Diagram](image)

Figure 5.5: Integral domain specification.

**Lemma** Let \( A \in \text{Alg}(\Sigma_{\text{CRing}}, T_{\text{IntD}}) \) and consider the quadratic equation

\[ x^2 + ax + b = 0 \]

in \( A \). Then, if the quadratic polynomial factorises

\[ x^2 + ax + b = (x + p).(x + q) \]

for some \( p, q \in A \), then

\[ x = -p \quad \text{and} \quad x = -q \]

are solutions.
5.4 Axiomatic Specification of Fields

The standard $\Sigma_{CRing}$-algebras of the integer, rational, real and complex numbers are all integral domains. The last three have an important additional operation:

\textit{division}

Abstractly, division is a unary operation $^{-1}$ and the inverse operation for multiplication, in the sense that

$$(\forall x)[x \neq 0 \Rightarrow x \cdot x^{-1} = x^{-1} \cdot x = 1].$$

If this operation $^{-1}$ is added to those in $\Sigma_{CRing}$, and the inverse axiom is added to $T_{CRing}$, then we can create a new specification for a new data type called $fields$.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{signature} & $Field$ \\
\textbf{sorts} & $field$ \\
\textbf{constants} & $0, 1 : \rightarrow field$ \\
\textbf{operations} & $- : field \rightarrow field$ \\
& $+ : field \times field \rightarrow field$ \\
& $\cdot : field \times field \rightarrow field$ \\
& $^{-1} : field \rightarrow field$ \\
\hline
\end{tabular}
\end{center}

Let us postulate that these operations satisfy the following laws or axioms.
<table>
<thead>
<tr>
<th>Axioms</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associativity of addition</td>
<td>$(\forall x)(\forall y)(\forall z)[(x + y) + z = x + (y + z)]$ (1)</td>
</tr>
<tr>
<td>Commutativity for addition</td>
<td>$(\forall x)(\forall y)[x + y = y + x]$ (2)</td>
</tr>
<tr>
<td>Identity for addition</td>
<td>$(\forall x)[x + 0 = x]$ (3)</td>
</tr>
<tr>
<td>Inverse for addition</td>
<td>$(\forall x)[x + (-x) = 0]$ (4)</td>
</tr>
<tr>
<td>Associativity for multiplication</td>
<td>$(\forall x)(\forall y)(\forall z)[(x.y).z = x.(y.z)]$ (5)</td>
</tr>
<tr>
<td>Commutativity for multiplication</td>
<td>$(\forall x)[x.y = y.x]$ (6)</td>
</tr>
<tr>
<td>Identity for multiplication</td>
<td>$(\forall x)[x.1 = x]$ (7)</td>
</tr>
<tr>
<td>Inverse for multiplication</td>
<td>$(\forall x)[x \neq 0 \Rightarrow x.x^{-1} = x^{-1}.x = 1]$ (8)</td>
</tr>
<tr>
<td>Distribution</td>
<td>$(\forall x)(\forall y)(\forall z)[x.(y + z) = x.y + x.z]$ (9)</td>
</tr>
<tr>
<td>Distinctness</td>
<td>$0 \neq 1$ (10)</td>
</tr>
</tbody>
</table>

We will study the axioms for a field, and refine them by adding further axioms about orderings, when we consider the data type of real numbers in Chapter 8.

Division allows us to advance with our equation-solving.

**Lemma (Fields are Integral Domains)** A field has no zero-divisors and hence is an integral domain.

**Proof.** Let $A$ be a field. Suppose $x, y \in A$ are zero-divisors, so

$$x \neq 0 \text{ and } y \neq 0 \text{ but } x.y = 0.$$ 

Now the inverses $x^{-1}$ and $y^{-1}$ exist because $A$ is a field. Multiplying them, and substituting $x.y = 0$, we have

$$(x.y)(x^{-1}.y^{-1}) = 0.(x^{-1}.y^{-1}),$$

and so, by commutativity of $\cdot$ and by Lemma 5.3.2(ii),

$$ (x.x^{-1}).(y.y^{-1}) = 0 $$

$$ 1.1 = 0 $$

$$ 1 = 0 $$

which is a contradiction. \hfill \square

Adding division allows us to solve all linear equations

$$ax + b = 0$$

in a field.
5.4. AXIOMATIC SPECIFICATION OF FIELDS

Lemma (Linear Equation Field Solutions) Let \( A \) be a field. Then the equation

\[
ax + b = 0
\]

for \( a \neq 0 \) has a solution

\[
x = -(a^{-1}b)
\]

in \( A \).

**Proof** Suppose \( ax + b = 0 \). Adding \( -b \) to both sides gives us

\[
(ax + b) + (-b) = 0 + (-b)
\]

which reduces to

\[
a.x + b + (b - b) = -b
\]

\[
a.x + 0 = -b
\]

\[
a.x = -b.
\]

Multiplying both sides by \( a^{-1} \) gives

\[
a^{-1}.(a.x) = a^{-1} \cdot -b
\]

\[
(a^{-1}a).x = -(a^{-1}b)
\]

\[
1.x = -(a^{-1}b)
\]

\[
x = -(a^{-1}b)
\]

\[
\square
\]

In fact the solution is unique.
We still cannot solve all equations because of the square root function.

**Example** The equation \( x^2 - 2 = 0 \) does not have a solution in the ring of real numbers.

**Lemma** Let \( A \) be a field. Consider

\[
a.x^2 + b.x + c = 0
\]

where \( a, b, c \in A \) and \( a \neq 0 \). The following are equivalent:

(i) the equation has two solutions

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

in \( A \); and

(ii) the element

\[
\sqrt{b^2 - 4ac} \in A.
\]
5.5 Axiomatic Specification of Groups and Abelian Groups

The axioms for a commutative ring and a field abstract the basic properties of

\[ +, \quad -, \quad \cdot \quad \text{and} \quad ^{-1} \]

from the corresponding operations on integer, rational and real numbers. In the case of a field, the axioms state:

(i) \( + \) is associative and commutative, and has an identity 0 and inverse \(-1\);

(ii) \( \cdot \) is associative and commutative, and has an identity 1 and inverse \(-1\); and

(iii) how \( + \) and \( \cdot \) interact according to the distributive laws.

Thus, we see that, as far as a field is concerned, \( + \) and \( \cdot \) have the same basic properties, whilst being different operators. These properties of binary operations make up the concept of group and Abelian group.

5.5.1 The Specification

We specify a group with the signature \( \Sigma_{\text{Group}}:\)

\[
\begin{align*}
\text{signature} & \quad \text{Group} \\
\text{sorts} & \quad \text{group} \\
\text{constants} & \quad e : \rightarrow \text{group} \\
\text{operations} & \quad \circ : \quad \text{group} \times \text{group} \rightarrow \text{group} \\
& \quad \circ^{-1} : \quad \text{group} \rightarrow \text{group}
\end{align*}
\]

and the laws \( T_{\text{Group}}:\)

\[
\begin{align*}
\text{axioms} & \quad \text{Group} \\
\text{Associativity of} \circ & \quad (\forall x)(\forall y)(\forall z)[(x \circ y) \circ z = x \circ (y \circ z)] \quad (1) \\
\text{Identity for} \circ & \quad (\forall x)[x \circ e = x] \quad (2) \\
\text{Inverse for} \circ & \quad (\forall x)[x \circ x^{-1} = e] \quad (3)
\end{align*}
\]

end

Adding the commutativity of \( \circ \) gives us an Abelian group:
5.5. AXIOMATIC SPECIFICATION OF GROUPS AND ABELIAN GROUPS

\[ \text{signature} \quad \text{Group} \]
\[ \text{sorts} \quad \text{group} \]
\[ \text{constants} \quad e : \rightarrow \text{group} \]
\[ \text{operations} \quad - \circ - : \text{group} \times \text{group} \rightarrow \text{group} \]
\[ -^{-1} : \text{group} \rightarrow \text{group} \]

with the laws \( T_{\text{AGroup}} \):

\[ \text{axioms} \quad \text{AGroup} \]
\[ \text{Associativity of } \circ \quad (\forall x)(\forall y)(\forall z)[(x \circ y) \circ z = x \circ (y \circ z)] \quad (1) \]
\[ \text{Commutativity of } \circ \quad (\forall x)(\forall y)[x \circ y = y \circ x] \quad (2) \]
\[ \text{Identity for } \circ \quad (\forall x)[x \circ e = x] \quad (3) \]
\[ \text{Inverse for } \circ \quad (\forall x)[x \circ x^{-1} = e] \quad (4) \]
end

The pairs
\[ (\Sigma_{\text{Group}}, T_{\text{Group}}) \quad \text{and} \quad (\Sigma_{\text{Group}}, T_{\text{AGroup}}) \]
are axiomatic specifications.

Let
\[ Alg(\Sigma_{\text{Group}}, T_{\text{Group}}) \quad \text{and} \quad Alg(\Sigma_{\text{Group}}, T_{\text{AGroup}}) \]
denote the classes of all \( \Sigma_{\text{Group}} \) algebras that satisfy the axioms in \( T_{\text{Group}} \) and \( T_{\text{AGroup}} \), respectively.

**Definition (Group)** A \( \Sigma_{\text{Group}} \)-algebra \( A \) is defined to be a group if it satisfies the group axioms, i.e., \( A \in Alg(\Sigma_{\text{Group}}, T_{\text{Group}}) \).

**Definition (Abelian Group)** A \( \Sigma_{\text{Group}} \)-algebra \( A \) is defined to be an Abelian group if it satisfies the Abelian group axioms, i.e., \( A \in Alg(\Sigma_{\text{Group}}, T_{\text{AGroup}}) \).

**Example** We have seen the following examples of Abelian groups:

\( i \) \( (\mathbb{Z}; 0; +, -) \);
\( ii \) \( (\mathbb{Q}; 0; +, -) \);
\( iii \) \( (\mathbb{Q}\setminus\{0\}; 1; \cdot, ^{-1}) \);
\( iv \) \( (\mathbb{R}; 0; +, -) \);
\( v \) \( (\mathbb{R}\setminus\{0\}; 1; \cdot, ^{-1}) \);
(vi) \((\mathbb{Z}_n; 0; +, -)\); and
(vii) \((\mathbb{Z}_p \setminus \{0\}; 1; \cdot, ^{-1})\) for \(p\) a prime number.

There are countless more examples. The most important examples are made from composing
transformations of data, objects and spaces.

### 5.5.2 Groups of Transformations

A **transformation** of a non-empty set \(X\) is simply a function

\[
f : X \rightarrow X.
\]

Let \(T(X)\) be the set of all transformations of \(X\).

The **composition** of functions is an operation

\[
\circ : T(X) \times T(X) \rightarrow T(X)
\]
defined for \(f, g \in T(X)\) by

\[
(f \circ g)x = f(g(x))
\]

for all \(x \in X\).

The operation of composition has an identity element, namely

\[
i : X \rightarrow X
\]
defined for all \(x \in X\) by

\[
i(x) = x,
\]
i.e., the identity function.

Not every function in \(T(X)\) has an inverse. If a transformation in \(T(X)\) has an inverse,
then it is said to be **invertible**. A transformation is invertible if, and only if, it is surjective and
injective, i.e., it is bijective.

Let \(\text{Sym}(X)\) be the set of all invertible transformations.

Taking the inverse of a transformation is an operation

\[
^{-1} : \text{Sym}(X) \rightarrow \text{Sym}(X).
\]

Since composition preserves invertible transformations, i.e.,

\[
f, g \in \text{Sym}(X) \implies f \circ g \in \text{Sym}(X)
\]

and the identity transformation \(i \in \text{Sym}(X)\), we can gather the operations together to form a
group

\[
A = (\text{Sym}(X); i, \circ, ^{-1})
\]
called the **group of permutations** on \(X\), or **symmetric group** on \(X\).

**Lemma** For any set \(X\) with cardinality \(|X| > 1\), the subalgebra \(A = (\text{Sym}(X); i, \circ, ^{-1})\) is a
group, but it is not an Abelian group.

**Proof** Exercise. \(\square\)
5.5. Axiomatic Specification of Groups and Abelian Groups

5.5.3 Matrix Transformations

Matrices are used to represent geometric transformations and form groups. The transformations are linear. This example requires an acquaintance with matrices.

A $2 \times 2$-matrix of real numbers is an array of the form

$$a = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

where $a_{11}, a_{12}, a_{21}, a_{22} \in \mathbb{R}$. Let $M(2, \mathbb{R})$ be the set of all $2 \times 2$-matrices. The $2 \times 2$-matrices represent linear transformations of the plane $\mathbb{R}^2$.

Matrix multiplication is an operation

$$\cdot : M(2, \mathbb{R}) \times M(2, \mathbb{R}) \to M(2, \mathbb{R})$$
defined by

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

The operation has an identity element, namely

$$i = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$ 

Not every matrix has an inverse. If a matrix has an inverse, it is said to be non-singular. A matrix is non-singular if, and only if, its determinant, defined by the operation

$$\text{det}(a) = a_{11}a_{22} - a_{12}a_{21},$$

is non zero. Let $GL(2, \mathbb{R})$ be the set of all $2 \times 2$-non-singular matrices.

Matrix inversion is an operation

$$^{-1} : GL(2, \mathbb{R}) \to GL(2, \mathbb{R})$$
defined by

$$a^{-1} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^{-1} = \begin{pmatrix} a_{22}/\text{det}(a) & -a_{12}/\text{det}(a) \\ -a_{21}/\text{det}(a) & a_{11}/\text{det}(a) \end{pmatrix}$$

Now matrix multiplication preserves non-singular matrices, i.e.,

$$a, b \in GL(2, \mathbb{R}) \text{ implies } a, b \in GL(2, \mathbb{R}).$$

And the identity $i \in GL(2, \mathbb{R})$. Thus, gathering the operations together forms a group

$$A = (GL(2, \mathbb{R}); i, \cdot, ^{-1})$$
called the group of $2 \times 2$-non-singular matrices, or general linear group.

**Lemma** $GL(2, \mathbb{R})$ is a group but it is not an Abelian group.

**Proof.** The checking of the three group axioms we leave to the exercises. To see that the commutative law of Abelian groups is not true of matrix multiplication, note that:

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$$

$\square$
5.5.4 Reasoning with the Group Axioms

The group axioms are very simple. However, they capture the fundamental properties of symmetry in an abstract and general way. They enable us to develop and prove techniques and properties with many applications. We will look at equation-solving in groups. First, we note the following.

Theorem The commutative law
\[ x \circ y = y \circ x \]
cannot be proved from the three axioms for a group.

Proof. Suppose, for a contradiction, that the commutative law was provable from the group axioms. Then, by the Soundness Principle, it would be true of all groups, i.e., all groups would be automatically Abelian. However, we know that the non-singular matrices \( GL(2, \mathbb{R}) \) is a group that does not satisfy the commutative law. This contradicts the assumption. \( \square \)

Groups have all the properties for solving simple equations

Lemma Let \( A \) be any group. For any \( a, b \in A \), the equations
\[ x \circ a = b \quad \text{and} \quad a \circ x = b \]
have unique solutions
\[ x = b \circ a^{-1} \quad \text{and} \quad x = a^{-1} \circ b, \]
respectively.

Proof. We must use the three group axioms to check the solutions given are correct, and then to show they are the only solutions possible.

Substituting \( x = b \circ a^{-1} \) in LHS of the equation,
\[ (b \circ a^{-1}) \circ a = b \circ (a^{-1} \circ a) \]
\[ = b \circ e \]
\[ = b \]
by associativity axiom;
by inverse axiom;
by identity axiom.

So the solution given is indeed a solution.

For uniqueness, suppose \( x \circ a = b \). Then
\[ x = x \circ e \]
\[ = x \circ (a \circ a^{-1}) \]
\[ = (x \circ a) \circ a^{-1} \]
\[ = b \circ a^{-1} \]
by identity axiom;
by inverse axiom;
by associativity axiom;
by the initial equation.

Similar arguments work for \( a \circ x = b \) and its solution \( x = a^{-1} \circ b^{-1} \). \( \square \)

Corollary For all \( a, b, c, d \in A \),
\[ c \circ a = d \circ a \quad \text{implies} \quad c = d \]
\[ a \circ c = a \circ d \quad \text{implies} \quad c = d \]

A group has only one identity element, and each element has only one inverse element.

The beauty and utility of the general theories of rings and fields, especially those parts that focus on these number algebras, is amazing and we will pay tribute to them by not attempting to trivialise them in this text. The reader is recommended to study the elements of this theory independently. Elementary introductions are Birkhoff and MacLane [1965], Herstein [1964] and Fraleigh [1967]; advanced works are van der Waerden [1949] and Cohn [1982].
5.6 Current Position

The mathematical concepts and the programming ideas about data they model, are summarised below.

<table>
<thead>
<tr>
<th>Mathematical Concept</th>
<th>Notation</th>
<th>Model for Programming Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature</td>
<td>$\Sigma$</td>
<td>interface for data type</td>
</tr>
<tr>
<td>$\Sigma$-algebra</td>
<td>$A$</td>
<td>concrete representation or implementation of a data type with interface $\Sigma$</td>
</tr>
<tr>
<td>class of all $\Sigma$-algebras</td>
<td>$\text{Alg}(\Sigma)$</td>
<td>class of all conceivable implementations or representations of a data type with interface $\Sigma$</td>
</tr>
<tr>
<td>axiomatic theory</td>
<td>$(\Sigma, T)$</td>
<td>specification of properties that representations or implementations of a data type must satisfy</td>
</tr>
<tr>
<td>axiomatic class</td>
<td>$\text{Alg}(\Sigma, T)$</td>
<td>class of all representations or implementations of a data type with signature $\Sigma$ satisfying the properties in $T$</td>
</tr>
</tbody>
</table>

The discussion in this chapter suggests two points of general interest when designing and specifying a data type:

**Interface**  
*We must select names for data and operations.*

**Specification**  
*We must consider what properties of the operations of the data type are needed or desired.*

Looking ahead to Chapter 7, and recalling some of the examples in Chapter 3, a third point of general interest is:

**Equivalence**  
*We must have ways of telling when are two implementations of the data type equivalent.*

A notion of the equivalence of implementations is needed in the comparison of say decimal and binary data representations of the integers. In the algebraic theory of data, this is done by mappings between algebras with the same signature $\Sigma$, called $\Sigma$-homomorphisms. A
\(\Sigma\)-homomorphism establishes a correspondence between the data in the algebras and the operations on that data named in \(\Sigma\). In particular, two specific implementations, modelled by algebras \(A\) and \(B\) with the same signature \(\Sigma\), are equivalent if there is a bijective \(\Sigma\)-homomorphism between them called an \(\Sigma\)-isomorphism. These ideas are the basis for Chapter 7.
5.6. CURRENT POSITION

Exercises for Chapter 5

1. Prove that \( \mathbb{Z}_n \) is a commutative ring with unit.

2. Write out the tables for all the cyclic arithmetic operations of \( \mathbb{Z}_6 \) and \( \mathbb{Z}_7 \).

3. Which of the following satisfy the no zero divisors property, i.e., are integral domains?
   a. \( \mathbb{Z}_5 \);
   b. \( \mathbb{Z}_6 \);
   c. \( \mathbb{Z}_7 \); and
   d. \( \mathbb{Z}_8 \).

   List all the zero divisors in each case, if any exist.

4. Prove that \( \mathbb{Z}_n \) has no divisors of zero, if, and only if, \( n \) is prime.

5. Let \( A \) be a commutative ring with unit. Consider the following cancellation law for all \( x, y, z \in A \),

   \[ x.y = x.z \text{ and } x \neq 0 \implies y = z. \]

   Prove that the following are equivalent:
   a. \( A \) satisfies the cancellation law; and
   b. \( A \) has no zero divisors.

6. Which of the following equations can be solved in the commutative ring of integers:
   a. \( x - 3 = 0 \);
   b. \( x + 3 = 0 \);
   c. \( 2x + 3 = 0 \);
   d. \( 3x + 3 = 0 \); and
   e. \( 9x + 3 = 0 \)?

7. Find all the solutions to the equation \( 3x = 1 \) in the commutative rings with unit
   a. \( \mathbb{Z}_5 \);
   b. \( \mathbb{Z}_6 \);
   c. \( \mathbb{Z}_7 \); and
   d. \( \mathbb{Z}_8 \).

8. Find all the solutions to the equation \( x^2 + 3x + 2 = 0 \) in the commutative rings with unit
   a. \( \mathbb{Z}_5 \);
   b. \( \mathbb{Z}_6 \);
   c. \( \mathbb{Z}_7 \); and
CHAPTER 5. SPECIFICATIONS AND AXIOMS

9. Consider maximum and minimum integers. First add constants to the signature $\Sigma_{\text{Integers}}$ of Section 5.2 of the integers to make the signature

```

<table>
<thead>
<tr>
<th>signature</th>
<th>Integers with $\text{max}$ and $\text{min}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>int</td>
</tr>
<tr>
<td>constants</td>
<td>$\text{zero}: \to \text{int}$</td>
</tr>
<tr>
<td></td>
<td>$\text{max}: \to \text{int}$</td>
</tr>
<tr>
<td></td>
<td>$\text{min}: \to \text{int}$</td>
</tr>
<tr>
<td>operations</td>
<td>$\text{add}: \text{int} \times \text{int} \to \text{int}$</td>
</tr>
<tr>
<td></td>
<td>$\text{minus}: \text{int} \to \text{int}$</td>
</tr>
<tr>
<td></td>
<td>$\text{times}: \text{int} \times \text{int} \to \text{int}$</td>
</tr>
</tbody>
</table>

endsig
```

Any $\Sigma_{\text{Integers}}$-algebra $A$ can become a $\Sigma_{\text{Integers with max and min}}$-algebra on choosing elements of $A$ and interpreting $\text{Max}$ and $\text{Min}$.

a. Consider the alternative overflow equations for the maximum and minimum elements

$$\text{Max} + 1 = \text{Min} \quad \text{and} \quad \text{Min} - 1 = \text{Max}.$$  

Show that modulo $n$ arithmetic $\mathbb{Z}_n$ forms a $\Sigma_{\text{Integers with max and min}}$-algebra that satisfies these properties.

b. Consider the overflow equations

$$\text{Max} + 1 = \text{Max} \quad \text{and} \quad \text{Min} - 1 = \text{Min}.$$  

Show, by adding the symbols $+\infty$ and $-\infty$ to $\mathbb{Z}$, how to extend the standard algebra of integers to form the $\Sigma_{\text{Integers with max and min}}$-algebra that satisfies the overflow equations. Does the algebra satisfy the properties of a commutative ring?

c. Test how $\text{Max} + 1$ works

i. in your favourite programming language;

ii. using your favourite spreadsheet; and

iii. using a pocket calculator.

10. Prove that $\mathbb{Z}_n$ is a commutative ring with unit.

11. Write out the tables for all the cyclic arithmetic operations of $\mathbb{Z}_6$ and $\mathbb{Z}_7$.

12. Which of the following satisfy the no zero divisors property, i.e., are integral domains?

a. $\mathbb{Z}_5$;

b. $\mathbb{Z}_6$;
5.6. CURRENT POSITION

c. \( \mathbb{Z}_r \); and
d. \( \mathbb{Z}_8 \).

List all the zero divisors in each case, if any exist.

13. Prove that \( \mathbb{Z}_n \) has no divisors of zero, if, and only if, \( n \) is prime.

14. Let \( A \) be a commutative ring with unit. Consider the following cancellation law: for all \( x, y, z \in A \),

\[ x.y = x.z \quad \text{and} \quad x \neq 0 \implies y = z. \]

Prove that the following are equivalent:

a. \( A \) satisfies the cancellation law; and
b. \( A \) has no zero divisors.

15. Which of the following equations can be solved in the commutative ring of integers:

a. \( x - 3 = 0 \);
b. \( x + 3 = 0 \);
c. \( 2x + 3 = 0 \);
d. \( 3x + 3 = 0 \); and
e. \( 9x + 3 = 0 \)?

16. Find all the solutions to the equation \( 3x = 1 \) in the commutative rings with unit

a. \( \mathbb{Z}_5 \);
b. \( \mathbb{Z}_6 \);
c. \( \mathbb{Z}_7 \); and
d. \( \mathbb{Z}_8 \).

17. Find all the solutions to the equation \( x^2 + 3x + 2 = 0 \) in the commutative rings with unit

a. \( \mathbb{Z}_5 \);
b. \( \mathbb{Z}_6 \);
c. \( \mathbb{Z}_7 \); and
d. \( \mathbb{Z}_8 \).

18. Consider maximum and minimum integers. First add constants to the signature \( \Sigma_{\text{Integers}} \) of Section 5.2 of the integers to make the signature
**signature**  \(\text{Integers with max and min}\)

**sorts**  \(\text{int}\)

**constants**  \(\text{zero} : \rightarrow \text{int}\)
\(\text{max} : \rightarrow \text{int}\)
\(\text{min} : \rightarrow \text{int}\)

**operations**  \(\text{add} : \text{int} \times \text{int} \rightarrow \text{int}\)
\(\text{minus} : \text{int} \rightarrow \text{int}\)
\(\text{times} : \text{int} \times \text{int} \rightarrow \text{int}\)

**endsig**

Any \(\Sigma_{\text{Integers}}\)-algebra \(A\) can become a \(\Sigma_{\text{Integers with max and min}}\)-algebra on choosing elements of \(A\) and interpreting \(\text{Max}\) and \(\text{Min}\).

a. Consider the alternative overflow equations for the maximum and minimum elements

\[\text{Max} + 1 = \text{Min}\quad \text{and}\quad \text{Min} - 1 = \text{Max}.\]

Show that modulo \(n\) arithmetic \(\mathbb{Z}_n\) forms a \(\Sigma_{\text{Integers with max and min}}\)-algebra that satisfies these properties.

b. Consider the overflow equations

\[\text{Max} + 1 = \text{Max}\quad \text{and}\quad \text{Min} - 1 = \text{Min}.\]

Show, by adding the symbols \(+\infty\) and \(-\infty\) to \(\mathbb{Z}\), how to extend the standard algebra of integers to form the \(\Sigma_{\text{Integers with max and min}}\)-algebra that satisfies the overflow equations. Does the algebra satisfy the properties of a commutative ring?

c. Test how \(\text{Max} + 1\) works

i. in your favourite programming language;

ii. using your favourite spreadsheet; and

iii. using a pocket calculator.
Assignment: Boolean Algebra

The data types of Booleans \{tt, ff\}, bits \{0, 1\} and subsets \(\mathcal{P}(X)\) of any set \(X\), have many algebraic properties in common. Much of what they have in common can be captured in a set of axioms often called the laws of Boolean algebra. The axioms, or laws, are equations expressing basic properties of some simple operations. The set of operations and their axioms constitute an axiomatic specification

\[(\Sigma_{BA}, T_{BA})\]

whose class

\[\text{Alg}(\Sigma_{BA}, T_{BA})\]

contains precisely the Boolean algebras. There are a number of equivalent sets of axioms that characterise Boolean algebras. We will choose a slight adaptation of the axiomatisation first found by E V Huntington in 1904. Boolean algebra is beautiful and a deep field; see the Further Reading.

<table>
<thead>
<tr>
<th>signature</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>(s)</td>
</tr>
<tr>
<td>constants</td>
<td>(0 : \rightarrow s)</td>
</tr>
<tr>
<td></td>
<td>(1 : \rightarrow s)</td>
</tr>
<tr>
<td>operations</td>
<td>(\cup : s \times s \rightarrow s)</td>
</tr>
<tr>
<td></td>
<td>(\cap : s \times s \rightarrow s)</td>
</tr>
<tr>
<td></td>
<td>(' : s \rightarrow s)</td>
</tr>
</tbody>
</table>

endsig
1. When working with a particular class $K$ of algebras, it is often important that

if $A \in K$ and $B$ is a subalgebra of $A$, then $B \in K$.

**Definition** A class $K$ of $\Sigma$-algebras is said to be *closed under the formation of subalgebras* if, and only if, whenever $A \in K$ and $B \leq A$ then $B \in K$.

a. Are the classes of semigroups, groups, rings and fields closed under the formation of subalgebras?

b. Is any class of algebras defined by equations closed under the formation of subalgebras?

c. Is the class of all finite structures of any signature $\Sigma$ closed under the formation of subalgebras?

2. Let $A$ be the following $\Sigma_{BA}$-algebra of the power set $\mathcal{P}(X)$ of an arbitrary set $X$.

| algebra | $A$ |
| carriers | $\mathcal{P}(X)$ |
| constants | $\emptyset : \rightarrow \mathcal{P}(X)$ |
| operations | $\cup : \mathcal{P}(X) \times \mathcal{P}(X) \rightarrow \mathcal{P}(X)$, $\cap : \mathcal{P}(X) \times \mathcal{P}(X) \rightarrow \mathcal{P}(X)$, $' : \mathcal{P}(X) \rightarrow \mathcal{P}(X)$ |
5.6. CURRENT POSITION

Prove that $A$ satisfies all the axioms in $T_{BA}$, i.e.,

$$A \in \text{Alg}(\Sigma_{BA}, T_{BA}).$$

3. Use the axioms in $T_{BA}$ to deduce that the following equations hold for every Boolean algebra:

a. $0' = 1$
   
   $1' = 0$

b. $x \cup x = x$
   
   $x \cap x = x$

   Idempotent Laws

c. $(x')' = x$

   Double Negation Law

d. $(x \cup y)' = x' \cap y'$
   
   $(x \cap y)' = x' \cup y'$

   De Morgan’s Laws

4. Show the following equations are not valid in $A$:

a. $0 \cap x = x$;

b. $1 \cup x = x$; and

c. $0 = 1$.

Correct the equations and prove the corrected properties are valid in all Boolean algebras.

The associativity axioms in the specification can be proved from the other 8 axioms and so they are, strictly speaking, redundant. If we remove them, we are left with the original axioms in Huntington [1904].
Chapter 6

Examples: Data Structures, Files, Streams and Spatial Objects

We began our study of data in Chapter 3 by introducing six kinds of data, namely

*Booleans, natural numbers, integers, rational numbers, real numbers and strings.*

We used them to explore some important ideas about data types. In particular, in Chapter 4, we showed how to model data types in general, using signatures and algebras. In Chapter 5, we showed how to specify data types using axiomatic theories. Most data types are constructed from these six data types. Indeed, in practice, one can make do with Booleans, integers, reals and strings.

Now we will consider modelling some new examples of data types, drawn from different subjects. Each example uses some general methods that construct new data types from old. The constructions involve adding new data sets and operations to data types, and importing one data type into another (recall Sections 4.6.2–4.7).

A data structure has operations for reading and updating and is a data type to be modelled by an algebra. Data structures are *general* constructions which can be made to store *any* data. They are sometimes called *generic* data types because they are designed to be applied to a wide class of data types. We begin with algebraic constructions of the popular data structures:

*records* and *arrays*.

We can apply these constructions to make many data types, such as,

*records of strings and integers,* and *arrays of real numbers.*

Hopefully, such data types are (or seem!) familiar. The constructions are not difficult. They provide us with practice in working with general ideas and in modelling data; and they result in some data types we will use later. Of course, there are dozens more data structures to consider, such as stacks, lists, queues, trees, graphs, etc. See textbooks on data structures (e.g., Dale and Walker [1996]) for these and many more to think about.

Next, we model algebraically the ubiquitous data type of

*files.*

Again, this is a general construction and can be applied to make many data types, such as

141
Files are a means of ensuring the persistency of data. Thus, a file is principally used for the storage of data for subsequent retrieval. The means of storing and accessing the data may vary, and leads to different models. To form a complete model of files, the management of files also needs to be considered. A file management system needs operations that take account of file identity and other factors such as ownership, access rights and read-write permissions. A file management system is a data type and is modelled as an algebra.

Now we turn to two data types that are natural and useful in modelling all sorts of systems — both natural and artificial — to be found in the world. We will model how data is distributed in time and space.

We model algebraically the important data type of

\[ \text{infinite streams.} \]

Again this is a general construction and can be applied to make many data types such as

\[ \text{infinite streams of bits, real numbers or strings.} \]

Now a \textit{stream} is a sequence

\[ \ldots, a_t, \ldots \]

of data \( a_t \) indexed by time \( t \). Time may be discrete or continuous. Streams are used to model \n
\[ \text{interactive systems.} \]

An interactive system is a system that interacts with an environment over time. The interactive system \textit{continually} receives inputs from its environment, computes for a period, and returns a result to its environment. Examples of interactive systems abound in digital hardware, operating systems and networks; and, more visibly, in software that control machines or provide interactive services for users. Typically, the system computes forever, measured by a discrete clock, and its behaviour can be modelled by a

\[ \text{transformation of streams} \]

as shown in Figure 6.1.

![Figure 6.1: A typical interactive system computing by transforming streams.](image)

We define a simple property of stream transformations called

\[ \text{finite determinacy} \]
which is a necessary property of stream transformers modelling computing systems. A stream transformation is finitely determined if its output at any time depends only on its input over a finite time interval.

As a case study, we examine the process of doing arithmetic with real numbers using their infinite decimal expansions. An infinite decimal is, essentially, a stream of digits, and multiplication transforms streams of digits to streams of digits. We prove that multiplication cannot be defined by a finitely determined stream transformer, and, hence, cannot be defined by an algorithm!

Lastly, we model algebraically a data type of

\begin{equation*}
\textit{spatial objects},
\end{equation*}

This general construction can be applied to model

\begin{equation*}
\textit{graphical objects and scenes}
\end{equation*}

and

\begin{equation*}
\textit{states of physical systems}.
\end{equation*}

A spatial object is an association

\[ a_x, \ldots \]

of data to all points \( x \) in space. Space can be continuous or discrete. The data is arbitrary. There are lots of operations on spatial objects modelled by an algebra. We apply these general ideas in creating a data type of use in computer graphics.

In Volume Graphics, objects are represented in three dimensions: every point in space has data of interest. Plenty of operations are needed to construct, transform and visualise scenes. We make a data type of spatial objects in which data that represents

\begin{equation*}
\textit{visibility and colour}
\end{equation*}

is assigned to every point of three-dimensional space. The data are real numbers specifying values for

\begin{equation*}
\textit{opacity, and red, green and blue},
\end{equation*}

respectively. We define a collection of four simple operations

\begin{equation*}
\textit{union, intersection, difference and blending}
\end{equation*}

on these three-dimensional spatial objects to make an algebra capable of generating some beautiful visualisations in easy ways.

These data types involve constructions of new data types from existing data. Given our algebraic model of data types, we will model these data type constructions in two stages:

\textbf{Signature/Interface}  Given a signature \( \Sigma_{\text{Old}} \), we construct a new signature \( \Sigma_{\text{New}} \). This is done by adding new sorts and operations to \( \Sigma_{\text{Old}} \).
CHAPTER 6. EXAMPLES: DATA STRUCTURES, FILES, STREAMS AND SPATIAL OBJECTS

**Algebra/Implementation** Given any $\Sigma_{\text{Old}}$-algebra $A_{\text{Old}}$, we construct a new $\Sigma_{\text{New}}$-algebra $A_{\text{New}}$. This is done by adding new carriers and functions to $A_{\text{Old}}$ that interpret the new sorts and operations added to make $\Sigma_{\text{New}}$.

For example, given a signature $\Sigma$ and $\Sigma$-algebra $A$, we show how to create a signature $\Sigma_{\text{Array}}$ of arrays over $\Sigma$, and corresponding algebra $A_{\text{Array}}$ of arrays over $A$.

### 6.1 Records

Records are an invaluable programming construct for representing all sorts of user-defined data. They are based on the idea of the Cartesian product of sets. A record data structure has fixed length $n$, and is able to store $n$ fields of data. These fields of data may be of different types. We will specify operations on records, and so make an algebraic model of the data type of records over any collection of data fields.

Suppose we have collected all the relevant fields of data in some $\Sigma$-algebra $A$. Suppose the sorts of $\Sigma$ are the names:

$$\ldots, s, \ldots$$

chosen by the programmer with the application in mind for these fields. Suppose the fields of data are the carrier sets of $A$:

$$\ldots, A_s, \ldots$$

Ultimately, the field algebra $A$ is typically made from a collection of basic types like Booleans, integers, strings and real numbers.

We fix the type of the record as follows. Let

$$w = s(1) \times \cdots \times s(n)$$

be a product type over (some of) these sorts. Note that the individual fields are not necessarily distinct from each other (i.e., we allow the case where $s(i) = s(j)$ but $i \neq j$). Then we can construct an algebra

$$\text{Record}_w(A)$$

to model records with data fields $A_{s(1)}, \ldots, A_{s(n)}$ and length $n$ from $A$ by choosing some basic operations.

#### 6.1.1 Signature/Interface of Records

We start by constructing a signature to name the data sets of the model that we will construct.

**Old Signature/Interface** Suppose we have some signature $\Sigma$ for the fields with name $\text{Name}$ and sorts

$$\ldots, s, \ldots$$
6.1. RECORDS

New Signature/Interface  To the sorts \(s_1, \ldots, s_n\) from the signature \(\Sigma\), we add a new sort

\[\text{record}_w\]

to name the records of type \(w\) where

\[w = s(1) \times \cdots \times s(n)\]

is a product type of not necessarily distinct field types.

Now let us consider what operations we want on records. As for any data structure, we need to create records and change them.

We create a record from constituent fields with a constructor function

\[\text{create} : s(1) \times \cdots \times s(n) \rightarrow \text{record}_w\]

This operation allows us to store data in a record.

Now we also want to be able to retrieve this data. For each of the \(n\) fields, we define a function

\[\text{get} \_\text{field}_i : \text{record}_w \rightarrow s_i\]

to return the \(i^{th}\) field of a record; in programming languages these operations are typically written

\[s_i.\]

So far, we have created a static data structure — we cannot change any aspect of a record once we have created it. To rectify this situation, we can define functions

\[\text{change} \_\text{field}_i : s_i \times \text{record}_w \rightarrow \text{record}_w\]

that will change the data stored in just the \(i^{th}\) field of a record.

Using the \texttt{import} notation, we combine these elements to give a signature \(\Sigma_{\text{record}_w}\) of records:

<table>
<thead>
<tr>
<th>signature</th>
<th>(\text{Record}_w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>\texttt{Name}</td>
</tr>
<tr>
<td>sorts</td>
<td>(\text{record}_w)</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>\texttt{create} : (s(1) \times \cdots \times s(n) \rightarrow \text{record}_w)</td>
</tr>
<tr>
<td>operations</td>
<td>\texttt{get} _\texttt{field}_i : (\text{record}_w \rightarrow s(1))</td>
</tr>
<tr>
<td>operations</td>
<td>\texttt{get} _\texttt{field}_i : (\text{record}_w \rightarrow s(n))</td>
</tr>
<tr>
<td>operations</td>
<td>\texttt{change} _\texttt{field}_i : (s(1) \times \text{record}_w \rightarrow \text{record}_w)</td>
</tr>
<tr>
<td>operations</td>
<td>\texttt{change} _\texttt{field}_i : (s(n) \times \text{record}_w \rightarrow \text{record}_w)</td>
</tr>
</tbody>
</table>
These operations may be expected to satisfy the following axioms:

\[
\begin{align*}
\text{axioms} \\
\text{get\_field}_i(\text{create}(x_1, \ldots, x_n)) &= x_i \\
\text{get\_field}_i(\text{change\_field}_j(R, x)) &= \begin{cases} x & \text{if } i = j; \\ \text{get\_field}_i(R) & \text{if } i \neq j. \end{cases} \\
\text{create}(\text{get\_field}_1(R), \ldots, \text{get\_field}_n(R)) &= R
\end{align*}
\]

endaxioms

### 6.1.2 Algebra/Implementation of Records

Now we model an implementation of the signature of records.

**Old Algebra/Implementation** First, we need to import the \(\Sigma\)-algebra \(A\) with carrier sets

\[ \ldots, A_s, \ldots \]

of data.

**New Algebra/Implementation** To the carrier sets \(\ldots, A_s, \ldots\) from the algebra \(A\), we add the new carrier set

\[ \text{Record}_w(A) = A_{s(1)} \times \cdots \times A_{s(n)} \]

to model the records of type \(w = s(1) \times \cdots \times s(n)\). Thus, the \(i^{th}\) field will be of type \(A_{s(i)}\).

Now let us implement the operations on records declared in the new signature \(\Sigma_{\text{Record}_w}\).

We create a record with a function

\[ \text{Create} : A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow \text{Record}_w(A) \]

defined by

\[ \text{Create}(a_1, \ldots, a_n) = (a_1, \ldots, a_n) \]

for fields \(a_1 \in A_{s(1)}, \ldots, a_n \in A_{s(n)}\).

To retrieve this data stored within a record we define a function

\[ \text{get\_field}_i : \text{Record}_w(A) \rightarrow A_{s(i)} \]

for each of the \(n\) fields, so that

\[ \text{get\_field}_i((a_1, \ldots, a_n)) = a_i \]

returns the field \(a_i \in A_{s(i)}\) of the record.

To modify the data stored in a record, we define a function

\[ \text{change\_field}_i : A_{s(i)} \times \text{Record}_w(A) \rightarrow \text{Record}_w(A) \]
for each of the \( n \) fields by

\[
\text{Change}_i(b, (a_1, \ldots, a_n)) = (a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n)
\]

replaces the field \( i \) of the record with the value \( b \in A_{s(i)} \), whilst leaving all the other components unaltered.

Combining the data sets involved, and the functions to construct, access and manipulate records, we get the algebra, in summary:

<table>
<thead>
<tr>
<th>Algebra</th>
<th>( \text{Record}_w(A) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>( A )</td>
</tr>
<tr>
<td>Carriers</td>
<td>( \text{Record}<em>w(A) = A</em>{s(1)} \times \cdots \times A_{s(n)} )</td>
</tr>
<tr>
<td>Constants</td>
<td></td>
</tr>
</tbody>
</table>
| Operations    | \begin{align*}
    \text{Create} & : A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow \text{Record}_w(A) \\
    \text{Get}_i & : \text{Record}_w(A) \rightarrow A_{s(i)} \\
    \vdots & \\
    \text{Get}_n & : \text{Record}_w(A) \rightarrow A_{s(n)} \\
    \text{Change}_i & : A_{s(1)} \times \text{Record}_w(A) \rightarrow \text{Record}_w(A) \\
    \vdots & \\
    \text{Change}_n & : A_{s(n)} \times \text{Record}_w(A) \rightarrow \text{Record}_w(A)
\end{align*} |
| Definitions   | \begin{align*}
    \text{Create}(a_1, \ldots, a_n) & = (a_1, \ldots, a_n) \\
    \text{Get}_i((a_1, \ldots, a_n)) & = a_i \\
    \vdots & \\
    \text{Get}_n((a_1, \ldots, a_n)) & = a_n \\
    \text{Change}_i(b, (a_1, \ldots, a_n)) & = (b_1, a_2, \ldots, a_n) \\
    \vdots & \\
    \text{Change}_n(b, (a_1, \ldots, a_n)) & = (a_1, \ldots, a_{n-1}, b_n)
\end{align*} |

It is possible to define more operations on records using operations on \( A \) and these simple operations on \( \text{Record}_w(A) \).

### 6.2 Dynamic Arrays

Arrays store data of the same type. The data is stored in locations or cells that have addresses. Arrays have operations that allow any location to be read or updated. In particular, arrays can be

(i) \textit{static} in the sense that they are of fixed length \( n \), like the record structure considered in Section 6.1; or
(ii) *dynamic* in the sense that an array can grow in size as required — say, if an element is inserted into a position that is beyond the current length of the array.

The addresses are usually based on a simple indexing of locations in space. Space is usually 1, 2 or 3 dimensions.

We construct an algebraic model of dynamic arrays. We will do this in a very general way by augmenting an algebra \( A \) with arrays of arbitrary length to store the elements of \( A \). Thus, for each set \( A_s \) of data in \( A \), we will have an array for that data of sort \( s \), as shown in Figure 6.2.

\[
\begin{array}{ccccccc}
\vdots \\
\text{s-array} & a_1 & a_2 & a_3 & \cdots & a_{l-1} & a_l & ? & ? & ? & \cdots \\
\rightarrow & \text{length } l & \rightarrow & \text{empty locations} & \rightarrow \\
\vdots 
\end{array}
\]

Figure 6.2: A dynamic 1-dimensional array to store data from \( A_s \) for each sort \( s \in S \).

As usual, we construct a signature then an algebra.

### 6.2.1 Signature/Interface of Dynamic Arrays

We first produce a signature for arrays that simply lists the sorts, constants and operations of our model.

**Old Signature/Interface** Suppose we have a signature \( \Sigma \) with sorts \( \ldots, s, \ldots \) to provide us with information about the elements that we shall store.

In each array of sort \( s \), some locations have data and others are empty or uninitialised. Thus, we shall also need to model error cases that can arise from trying to access empty or uninitialised addresses. Thus, we construct the new signature

\[ \Sigma^u \]

that will add to the signature \( \Sigma \) new sort names

\[ \ldots, s_u, \ldots \]

and constants

\[ u_s \text{ for each sort } s \text{ in } \Sigma \]

for distinguished elements which we can use to flag error conditions (as described in Section 4.4.2).

There are several choices for the addresses of arrays. For simplicity, we will model a one-dimensional array and use natural numbers as addresses. Thus, we shall also use the signature \( \Sigma_{\text{natural}} \) for addresses.
New Signature/Interface  Let us picture, informally, the idea of an array. We want to store and retrieve elements in given addresses within an array. So, we shall need sorts for data, addresses and arrays of data. We have sorts for data from the imported signature $\Sigma$ and we shall also import the signature $\Sigma_{\text{Natural}}$ to provide us with a scheme for addresses.

For each sort $s$ of the signature $\Sigma$, we are going to have an array that will store the elements of sort $s$. For each data sort $s$, let

$$\text{array}_s$$

be the sort of arrays of sort $s$.

To store elements in an array of sort $s$, we define the operation

$$\text{insert}_s : s \times \text{nat} \times \text{array}_s \to \text{array}_s$$

that will allow us to insert an element of sort $s$ at a given address into an array of sort $s$; and we define a constant

$$\text{null}_s :\to \text{array}_s$$

that represents an array with no stored values. These constants and operations will allow us to create arrays.

For the array to be dynamic though, we shall also need an operation

$$\text{length}_s : \text{array}_s \to \text{nat}$$

to determine the current length of an array of sort $s$.

Finally, to be able to access the information stored in an array of sort $s$, we need the operation

$$\text{read}_s : \text{nat} \times \text{array}_s \to s_u$$

that returns the data stored in a given address from an array; if the given array position has not been initialised (because no data has yet been inserted) then we can flag this with an unspecified element.

Using the import notation to combine the sets of sort names, constant symbols and operation symbols, we get the signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>$\text{Array}(\Sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$\Sigma^u, \Sigma_{\text{Natural}}$</td>
</tr>
<tr>
<td>sorts</td>
<td>$\ldots, \text{array}_s, \ldots$</td>
</tr>
<tr>
<td>constants</td>
<td>$\ldots, \text{null}_s :\to \text{array}_s, \ldots$</td>
</tr>
<tr>
<td>operations</td>
<td>$\ldots, \text{insert}_s : s \times \text{nat} \times \text{array}_s \to \text{array}_s, \ldots$</td>
</tr>
<tr>
<td></td>
<td>$\ldots, \text{length}_s : \text{array}_s \to \text{nat}, \ldots$</td>
</tr>
<tr>
<td></td>
<td>$\ldots, \text{read}_s : \text{nat} \times \text{array}_s \to s_u, \ldots$</td>
</tr>
</tbody>
</table>

These operations may be expected to satisfy the following axioms:
axioms
\[
\begin{align*}
\text{read}_s(i, \text{null}_s) &= u_s \\
\text{read}_s(i, \text{insert}_s(x, j, a)) &= \begin{cases} x & \text{if } i = j; \\ \text{read}_s(i, a) & \text{otherwise}. \end{cases} \\
\text{length}_s(\text{null}_s) &= 0 \\
\text{length}_s(\text{insert}(x, j, a)) &= \max(j, \text{length}_s(a))
\end{align*}
\]
endaxioms

6.2.2 Algebra/Implementation of Dynamic Arrays
Now we will make a precise model of these operations by constructing a \(\Sigma_{\text{Array}}\)-algebra \(A_{\text{Array}}\) from a \(\Sigma\)-algebra \(A\). There are different ways of modelling arrays. We will give a slightly elaborate model that emphasises the role of empty or uninitialised cells in an array.

**Old Algebra/Implementation** We implement the signature \(\Sigma\) with an algebra \(A\) to define the data that we want to store in the array. Then, we construct the algebra \(A^u\) from \(A\) to add extra elements for flagging errors as described in Section 4.4.2.

**New Algebra/Implementation** Let \(A_s\) be a non-empty set of data and let \(u_s \notin A_s\) be an object we may use to mark unspecified data. We will also use the algebra \(\mathbb{N}\) of natural numbers to implement the addressing mechanism. We can picture an example of an array of length \(l\) as shown in Figure 6.3.

<table>
<thead>
<tr>
<th>Data</th>
<th>(u)</th>
<th>(b)</th>
<th>(\cdots)</th>
<th>(b)</th>
<th>(u)</th>
<th>(a)</th>
<th>(u)</th>
<th>(u)</th>
<th>(\cdots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses</td>
<td>1</td>
<td>2</td>
<td>(l-2)</td>
<td>(l-1)</td>
<td>(l)</td>
<td>(l+1)</td>
<td>(l+2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3: Model of a finite array of length \(l\). After position \(l\), every cell is uninitialised. Some cells before position \(l\) may also be uninitialised.

We model a finite array by means of a function and a number, i.e., a pair

\[ a^* = (a, l), \]

where: the function

\[ a : \mathbb{N} \to A^u_s \]

gives the data stored in each address i.e.,

\[ a(i) = \text{datum from } A_s \text{ stored at address } i \]

and

\[ a(i) = u_s \text{ means address } i \text{ has not been initialised}; \]
and

\[ l \in \mathbb{N} \]

gives the length of the array. Furthermore, since the array is finite and of length \( l \), we will assume that

\[ a(i) = u_s \text{ for all } i > l. \]

We define the set

\[ \text{Arrays}_s(A) = \{(a, l) \in [\mathbb{N} \to A^u_s] \times \mathbb{N} \mid a(i) = u_s \text{ for } i > l\}. \]

of all finite arrays over \( A_s \). Thus, a finite array is modelled as an infinite sequence \( a \) of addresses, together with a bound \( l \) on the location of addresses that have been assigned elements, and an assignment of data to some of the addresses up to \( l \).

To interpret the operation \( \text{insert}_s \) we take the function

\[ \text{Insert}_s : A^u_s \times \mathbb{N} \times \text{Arrays}_s(A) \to \text{Arrays}_s(A) \]

defined by

\[ \text{Insert}_s(x, i, (a, l)) = \begin{cases} (b, l) & \text{if } i \leq l; \\ (b, i) & \text{if } i > l; \end{cases} \]

where the element \( x \) is inserted into the \( i^{th} \) position of the array \( (a, l) \in \text{Arrays}_s(A) \) by

\[ b(j) = \begin{cases} a(j) & \text{if } j \neq i; \\ x & \text{if } j = i. \end{cases} \]

Note that our operation \( \text{Insert}_s \) automatically extends the length of the array if we try to insert an element in a position that would otherwise be past the end of the array, i.e., if \( i > l \). In this case, the operation only adds one element to the array, as all the intermediate values in the positions between the end of the old array (at \( l \)) and the end of the new array (at \( i \)) retain their value of \( u \).

In addition to the constants of the imported algebras \( A^u \) for data and \( \mathbb{N} \) for addresses, we add that of the newly created array \( \text{Null}_s^* \in \text{Arrays}_s(A) \) defined as

\[ \text{Null}_s^* = (\text{Null}_s, 0) \]

for any \( i \in \mathbb{N} \) by

\[ \text{Null}_s(i) = u_s. \]

To interpret the operation \( \text{length}_s \) we take the function

\[ \text{Length}_s : \text{Arrays}_s(A) \to \mathbb{N} \]

defined by

\[ \text{Length}_s((a, l)) = l. \]

To interpret the operation \( \text{read}_s \) we take the function

\[ \text{Read}_s : \mathbb{N} \times \text{Arrays}_s(A) \to A^u_s \]
defined by

\[ \text{Read}_a(i, (a, l)) = a(i) \]

which reads the \( i \)th element of the array \((a, l) \in \text{Array}_a(A)\).

In summary, we have constructed the algebra:

<table>
<thead>
<tr>
<th>algebra</th>
<th>Array(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>( A^u, N )</td>
</tr>
<tr>
<td>carriers</td>
<td>( \ldots, \text{Array}_a(A)), \ldots</td>
</tr>
<tr>
<td>constants</td>
<td>( \ldots, \text{Null}_a^* : \rightarrow \text{Array}_a(A), \ldots )</td>
</tr>
<tr>
<td>operations</td>
<td>( \ldots, \text{Length}_a : \text{Array}_a(A) \rightarrow N, \ldots )</td>
</tr>
<tr>
<td></td>
<td>( \ldots, \text{Read}_a : N \times \text{Array}_a(A) \rightarrow A^u, \ldots )</td>
</tr>
<tr>
<td></td>
<td>( \ldots, \text{Insert}_a : A^u \times N \times \text{Array}_a(A) \rightarrow \text{Array}_a(A), \ldots )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ldots, \text{Null}_a^* = (\text{Null}_a, 0), \ldots )</td>
</tr>
<tr>
<td>( \ldots, \text{Null}_a(i) = u, \ldots )</td>
</tr>
<tr>
<td>( \ldots, \text{Read}_a(i, (a, l)) = a(i), \ldots )</td>
</tr>
<tr>
<td>( \ldots, \text{Insert}_a(x, i, (a, l)) = (b, \text{Max}(i, l)), \ldots )</td>
</tr>
</tbody>
</table>

where

\[ b(j) = \begin{cases} 
  a(j) & \text{if } j \leq \text{Max}(i, l) \text{ and } j \neq i; \\
  x & \text{if } j \leq \text{Max}(i, l) \text{ and } j = i; \\
  u & \text{otherwise.}
\end{cases} \]

\[ \ldots, \text{Length}_a((a, l)) = l, \ldots \]

### 6.3 Algebras of Files

The concept of a file of characters has proved to be truly fundamental. For example, in the UNIX operating system, most of the basic objects are files. The many types of file are classified by the types of operations that can be applied to them. A file system is a data type of files and operations on files. What are the essential properties of files and file systems?

Are files fundamentally characters, or are files more general? We can easily imagine files of any data. Thus, given a signature \( \Sigma \) and a \( \Sigma \)-algebra \( A \) modelling some data type, we want to construct a signature and an algebra

\[ \Sigma_{\text{File}} \text{ and } A_{\text{File}}, \]

to model a data type of files containing data from some algebra \( A \). File systems are complicated and continually evolving.

We shall consider two simple models of files that store data, and we shall produce an algebra for each:
6.3. ALGEBRAS OF FILES

(i) We start in Section 6.3.1 with a model called *SimpleFiles*, with signature and algebra

\[ \Sigma_{\text{SimpleFiles}} \text{ and } A_{\text{SimpleFiles}}. \]

Here, we compute with files which have a data content and a position element.

(ii) Then in Section 6.3.2, we consider a more interesting and complex model called

*Files with Names and Access Permissions*,

with signature and algebra

\[ \Sigma_{\text{NamedWritePermissionFiles}} \text{ and } A_{\text{NamedWritePermissionFiles}}. \]

Here, we compute with files which have a name, data content, position element and read/write open access permissions.

Finally, we also consider how to model a file system in Section 6.3.3.

6.3.1 A Simple Model of Files

We consider first a simple model of files in which we store

(i) data, and

(ii) the current position within that data.

**Signature/Interface**

We construct a signature \( \Sigma_{\text{SimpleFiles}} \) of useful operations on files.

**Old Signature/Interface** We want to store and retrieve strings of elements within a file. Thus, we shall construct the signature \( \Sigma_{\text{String}} \) for strings of elements from a signature \( \Sigma \) (recall Section 3.7). This contains the basic operation *Concat* of concatenation of strings. We shall also use the natural numbers to mark the current position within a file.

**New Signature/Interface** We introduce a sort *file* for our data type of files, and a sort *read_result*, to take care of the side-effects that the *read* function produces.

We can construct a file with the constant

\[ \text{empty} : \to \text{file} \]

for a newly created file with no contents, and the operation

\[ \text{write} : \text{file} \times \text{string} \to \text{file} \]

which inserts a string at the current position within the file and moves the current position forward accordingly.

We can read a portion of a file with the operation

\[ \text{read} : \text{file} \times \text{nat} \to \text{read_result}. \]
154 CHAPTER 6. EXAMPLES: DATA STRUCTURES, FILES, STREAMS AND SPATIAL OBJECTS

This returns the next specified number of characters from the file starting at the current position, and as a side-effect, moves the current position of the file forward. To make this side-effect explicit, we take auxiliary functions,

\[
\begin{align*}
\text{string}\_\text{content} & : \text{read}\_\text{result} \to \text{string} \\
\text{file}\_\text{content} & : \text{read}\_\text{result} \to \text{file}
\end{align*}
\]

which we can use to separate the file and string results from the application of the \textit{read} operation.

We can also arbitrarily move the current position in the file with an operation

\[
\text{move}\_\text{pos} : \text{file} \times \text{nat} \to \text{file}
\]

This results in a signature:

\[
\begin{array}{l}
\text{signature} \quad \text{SimpleFiles} \\
\text{import} \quad \text{String, Naturals} \\
\text{sorts} \quad \text{file, read_result} \\
\text{constants} \quad \text{empty} : \to \text{file} \\
\text{operations} \quad \text{write} : \quad \text{file} \times \text{string} \to \text{file} \\
\text{} \quad \text{read} : \quad \text{file} \times \text{nat} \to \text{read_result} \\
\text{} \quad \text{move}\_\text{pos} : \quad \text{file} \times \text{nat} \to \text{file} \\
\text{} \quad \text{string}\_\text{content} : \quad \text{read}\_\text{result} \to \text{string} \\
\text{} \quad \text{file}\_\text{content} : \quad \text{read}\_\text{result} \to \text{file}
\end{array}
\]

**Algebra/Implementation**

**Old Algebra/Implementation** We suppose that we implement the signature \(\Sigma\) with an algebra \(A\). Then, we form strings over \(A\) to give us the algebra \(A_{\text{String}}\) to implement the signature \(\Sigma_{\text{String}}\).

We shall also suppose that we implement the signature \(\Sigma_{\text{Naturals}}\) with an algebra \(A_{\text{Naturals}}\) in which we operate over the set \(\mathbb{N}\) of natural numbers. Note that the algebra \(A_{\text{Naturals}}\) will be provided from the algebra \(A_{\text{String}}\).

**New Algebra/Implementation** We can implement the signature \(\Sigma_{\text{File}}\) with an algebra \(A_{\text{File}}\) in the following manner.

We have a carrier set

\[
\text{File} = \text{String} \times \mathbb{N}
\]

of files that implements the sort \textit{file}, so that a file

\[(s, p) \in \text{File}\]
stores the data content \( s \) of the file and the current position \( p \) of the file.

We also have a carrier set

\[
\text{ReadResult} = \text{String} \times \text{File}
\]

that implements the sort \( \text{read\_result} \), so that

\[
(w, f)
\]

will store a string \( w \) and a file \( f \).

We implement the constant symbol \( \text{empty} \) with the constant

\[
\text{Empty} = (\epsilon, 0) \in \text{File};
\]

that represents a file with no data content, and current position set at 0.

We implement the function symbol \( \text{write} \) with the function

\[
\text{Write} : \text{File} \times \text{String} \rightarrow \text{File}
\]

so that

\[
\text{Write}(f, w)
\]

inserts the string \( w \) into the data portion of the file \( f \) at the current position of \( f \). It then also sets the current position of the file to be at the last character of the newly inserted string \( w \).

We implement the function symbol \( \text{read} \) with the function

\[
\text{Read} : \text{File} \times \mathbb{N} \rightarrow (\text{String} \times \text{File})
\]

so that

\[
\text{Read}(f, i)
\]

returns the next \( i \) characters of the data portion of the file \( f \) from the current position of \( f \). It then sets the current position of the file to be \( i \) characters further forwards. If this would take us past the end of the file, we just take those characters that lie before the end of the file, and set the current position to the end of the file.

We implement the auxiliary functions \( \text{string\_content} \) and \( \text{file\_content} \) with projection functions

\[
\begin{align*}
\text{StringContent} : \text{ReadResult} &\rightarrow \text{String} \\
\text{FileContent} : \text{ReadResult} &\rightarrow \text{File}
\end{align*}
\]

that are defined in the obvious manner:

\[
\begin{align*}
\text{StringContent}((w, f)) &= w \\
\text{FileContent}((w, f)) &= f.
\end{align*}
\]

Hence,

\[
\text{StringContent} (\text{Read}(f, i))
\]

returns the string that results from reading \( i \) characters from the file \( f \), and

\[
\text{FileContent} (\text{Read}(f, i))
\]
returns the updated file, whereby the current position has been advanced by \(i\) characters.

We implement the function symbol \(move\_pos\) with the function

\[
MovePos : File \times N \rightarrow File
\]

so that

\[
MovePos(f, p)
\]

returns a file with data content the same as \(f\), but with a new current position of \(p\). If \(p\) is greater than the number of characters in the data content of \(f\), we set the new current position to be at the last character of the file.

This gives us the algebra:

<table>
<thead>
<tr>
<th>algebra</th>
<th>SimpleFiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>String, N</td>
</tr>
<tr>
<td>carriers</td>
<td>(File = String \times N)</td>
</tr>
<tr>
<td></td>
<td>(Read_Result = String \times File)</td>
</tr>
<tr>
<td>constants</td>
<td>Empty : (\rightarrow File)</td>
</tr>
<tr>
<td>operations</td>
<td>(Write : File \times String \rightarrow File)</td>
</tr>
<tr>
<td></td>
<td>(Read : File \times N \rightarrow Read_Result)</td>
</tr>
<tr>
<td></td>
<td>(Move_Pos : File \times N \rightarrow File)</td>
</tr>
<tr>
<td></td>
<td>(String_Content : Read_Result \rightarrow String)</td>
</tr>
<tr>
<td></td>
<td>(File_Content : Read_Result \rightarrow File)</td>
</tr>
<tr>
<td>definitions</td>
<td>(Empty = (\epsilon, 0))</td>
</tr>
<tr>
<td></td>
<td>(Write((f_1 \cdots f_i, p), w) = (Concat(Concat(f_1 \cdots f_p, w), f_{p+1} \cdots f_i), p +</td>
</tr>
<tr>
<td></td>
<td>(Read((f_1 \cdots f_i, p), j) = \begin{cases} \epsilon, (f_1 \cdots f_i, p) \quad &amp; \text{if } j = 0; \ (f_{p+1} \cdots f_{\min(p+j,l)}, (f_1 \cdots f_i, \min(p+j,l))) &amp; \text{otherwise.} \end{cases})</td>
</tr>
<tr>
<td></td>
<td>(Move_Pos((f, p), j) = (f, \min(j,</td>
</tr>
<tr>
<td></td>
<td>(String_Content((w, f)) = w)</td>
</tr>
<tr>
<td></td>
<td>(File_Content((w, f)) = f)</td>
</tr>
</tbody>
</table>

### 6.3.2 Files with Identities and Simple Permissions

We now suppose that files have

\(i\) a name,

\(ii\) a data content,

\(iii\) a current position and


(iii) a simple permission flag indicating open or closed, and, if open, read-only or write.

**Signature/Interface**

**Old Signature/Interface** As for our model of simple files, we shall construct a signature \( \Sigma_{\text{String}} \) of strings of data elements from a signature \( \Sigma \) for data.

We shall also need a signature \( \Sigma_{\text{Names}} \) for names. The precise nature of names is irrelevant for our model, we simply suppose that there exists some set of names, and that we can test when two names are the same:

<table>
<thead>
<tr>
<th>signature</th>
<th>Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>bool</td>
</tr>
<tr>
<td>sorts</td>
<td>name</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>( eq_{\text{name}} : \text{name} \times \text{name} \to \text{bool} )</td>
</tr>
</tbody>
</table>

**New Signature/Implementation** As for simple files, we have the operations of \( \text{empty} \), \( \text{read} \), \( \text{write} \), \( \text{move} \_\text{pos} \), \( \text{string} \_\text{content} \) and \( \text{file} \_\text{content} \). However, the existence of names involves a change. For example, \( \text{empty} \) is no longer a constant, but an operation

\[
\text{empty} : \text{name} \to \text{file}
\]

that creates a file with no content, but name. In addition, we introduce some new operations.

We can ask whether a file is writable or not with an operation

\[
\text{writeable} : \text{file} \to \text{bool}.
\]

We can copy a portion of a file into another with an operation

\[
\text{copy} : \text{file} \times \text{file} \times \text{nat} \to \text{copy\_result}
\]

which also as a side effect moves forward the current position of the file that is being copied from by the amount of symbols that have been copied.

Hence, we have a sort \( \text{file} \) for our data type of files, a sort \( \text{read\_result} \), to take care of the side-effects that the \( \text{read} \) function produces, and a sort \( \text{copy\_result} \), to take care of the side-effects that the \( \text{copy} \) function produces.

This results in a signature:
signature  SimpleFiles
import   Names, String
sorts file, read_result, copy_result
constants
operations empty : name → file
read : file × nat → read_result
write : file × string → file
move_pos : file × nat → file
string_content : read_result → string
file_content : read_result → file
writable : file → bool
copy : file × file × nat → copy_result

Algebra/Implementation
We adapt the algebra $A_{SimpleFiles}$ of Section 6.3.1, so that the operations on files now operate over the new carrier set

$$File = Name \times String \times N \times Open$$

of files, where

$$(n, s, p, o) \in File$$

has name $n$, data content $s$, current position set at $p$, and a flag $o$ to denote its “degree” of openness.

Old Algebra/Implementation We import the algebra

$$A_{Names}$$

whereby we interpret the operation

$$Eq_{Name} : Name \times Name \rightarrow B$$

within the names algebra as being that of equality:

$$Eq_{Name}(m, n) = \begin{cases} \text{tt} & \text{if } m = n; \\ \text{ff} & \text{otherwise.} \end{cases}$$

We also import the algebra

$$A_{String}$$

of strings over the algebra $A$. 
6.3. ALGEBRAS OF FILES

New Algebra/Implementation  We use the set

\[ \text{Open} = \{ \text{write, read, closed} \} \]

to denote whether a file is open for reading and writing, reading alone, or is closed. Thus,

- if a file is closed, the operations other than opening it have no effect;

- if the file is open with read- but not write-permission, then the write operation has no effect; and

- if the file is open for reading and writing, all the operations of Section 6.3.1 are effective.

We also implement two additional functions in this model: we can test whether a file is open and is writable to, with the function

\[ \text{Writable} : \text{File} \rightarrow \mathbb{B}; \]

and we can copy a string from one file to another with the function

\[ \text{Copy} : \text{File} \times \text{File} \times \mathbb{N} \rightarrow (\text{File} \times \text{File}). \]

At the same time as the Copy copies data from a file, it moves the current position of the file forwards. Hence, copy not only changes the file that has been copied to, but it also changes the file that has been copied from.

Thus, we have:
algebra \( \text{NamedWritePermissionFiles} \)

import \( \text{Name}, \text{String} \)

carriers \( \text{File} = \text{Name} \times \text{String} \times \mathbb{N} \times \text{Open} \)
\( \text{Open} = \{ \text{write, read, closed} \} \)
\( \text{String} \times \text{File} \)
\( \text{File} \times \text{File} \)

constants

operations
\( \text{Empty} : \text{Name} \rightarrow \text{File} \)
\( \text{Read} : \text{File} \times \mathbb{N} \rightarrow (\text{String} \times \text{File}) \)
\( \text{Write} : \text{File} \times \text{String} \rightarrow \text{File} \)
\( \text{MovePos} : \text{File} \times \mathbb{N} \rightarrow \text{File} \)
\( \text{StringContent} : (\text{String} \times \text{File}) \rightarrow \text{String} \)
\( \text{FileContent} : (\text{String} \times \text{File}) \rightarrow \text{File} \)
\( \text{Writable} : \text{File} \rightarrow \text{B} \)
\( \text{Copy} : \text{File} \times \text{File} \times \mathbb{N} \rightarrow (\text{File} \times \text{File}) \)

definitions

\[
\text{Empty}(n) = (n, \epsilon, 0, \text{write})
\]
\[
\text{Read}(n, s_1 \cdots s_l, p, o, j) =
\begin{cases} 
  (\epsilon, (n, s_1 \cdots s_l, p, o)) & \text{if } j = 0 \text{ or } o = \text{closed}; \\
  (s_{p+1} \cdots s_{\min(p+j,l)}, (n, s_1 \cdots s_l, \text{Min}(p+j,l), o)) & \text{otherwise}.
\end{cases}
\]
\[
\text{Write}(n, s_1 \cdots s_l, p, o) =
\begin{cases} 
  (n, \text{Concat} \left( \text{Concat} (s_1 \cdots s_p, w), s_{p+1} \cdots s_l \right), p + |w|, o) & \text{if } o = \text{write}; \\
  (n, s_1 \cdots s_l, p, o) & \text{otherwise}.
\end{cases}
\]
\[
\text{MovePos}(n, s, p, o, j) = (n, s, \text{Min}(j, |s|), o)
\]
\[
\text{StringContent}(w, s) = s
\]
\[
\text{FileContent}(w, s) = s
\]
\[
\text{Writable}(n, s, p, o) =
\begin{cases} 
  \text{tt} & \text{if } o = \text{write}; \\
  \text{ff} & \text{otherwise}.
\end{cases}
\]
\[
\text{Copy}(f, f', p) =
\begin{cases} 
  (\text{Write}(f, \text{StringContent}(\text{Read}(f', p))), \\
  \text{FileContent}(\text{Read}(f', p)))
\end{cases}
\]

6.3.3 File System

We collect together named files into a file system. A file system consists of a repository of files, where all aspects of the files can be altered. Thus files can be created and destroyed. They can be renamed, and their contents viewed and changed. A file system will also typically provide a mechanism for controlling read- and write-access to the files.
6.3. ALGEBRAS OF FILES

Signature/Interface

Old Signature/Interface  We shall build on the signature \( \Sigma_{\text{NamedWritePermissionFiles}} \) of Section 6.3.2.

New Signature

A number of operations we introduce into our model have side-effects on the file system when we update a file in some manner. Accordingly, we introduce the sort set

\[ \text{file\_result} \]

and projection functions

\[
\begin{align*}
\text{get\_file} & : \text{file\_result} \to \text{file} \\
\text{get\_file\_system} & : \text{file\_result} \to \text{file\_system}
\end{align*}
\]

We open a file with an operation

\[ \text{open} : \text{name} \times \text{file\_system} \to \text{file\_result} \]

which not only opens the specified file in a manner determined by the permissions, but may also update the file system at the same time. We can close a file with an operation

\[ \text{close} : \text{file} \times \text{file\_system} \to \text{file\_system}, \]

Depending on the permissions that the file has, this operation may also update the file system with a new version of the file.

We can alter the read/write permissions of files with the functions

\[
\begin{align*}
\text{read\_open} & : \text{file} \times \text{file\_system} \to \text{file\_result} \\
\text{write\_open} & : \text{file} \times \text{file\_system} \to \text{file\_result}.
\end{align*}
\]

Again, these operations not only alter the given file, but may also require the file system to be updated also.

We can change the name of a file with an operation

\[ \text{rename} : \text{file} \times \text{name} \times \text{file\_system} \to \text{file\_result}. \]

If the new name is not in current use, then the operation is effective, and the file system will need to be updated accordingly.

To define these operations on files, we also introduce some auxiliary functions

\[
\begin{align*}
\text{in\_system} & : \text{name} \times \text{file\_system} \to \text{Bool} \\
\text{find} & : \text{name} \times \text{file\_system} \to \text{file} \\
\text{update} & : \text{file} \times \text{file\_system} \to \text{file\_system}
\end{align*}
\]
Algebra/Implementation

Old Algebra/Implementation  We implement this model as a list of files: we import the algebra

\[ A_{\text{FileSystem}} \]

to give us the means of constructing lists of files with the operations of

\[ \text{EmptyFileSystem} : \rightarrow \text{FileSystem} \]
\[ \text{AddFile} : \text{File} \times \text{FileSystem} \rightarrow \text{FileSystem}. \]

New Algebra

We introduce the carrier set

\( (\text{File} \times \text{FileSystem}) \)

of data, and appropriate projection functions

\[ \text{GetFile} : (\text{File} \times \text{FileSystem}) \rightarrow \text{File} \]
\[ \text{GetFileSystem} : (\text{File} \times \text{FileSystem}) \rightarrow \text{FileSystem} \]

to accommodate the side-effects of some of the operations in our model.

We open a file with an operation

\[ \text{Open} : \text{Name} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \]

such that a file can be opened with read- and write-permissions if it is not already open; otherwise, it is opened in read-only mode. Thus, any given file may be opened more than once, but there is only one writable version of the file open at any one time. Accordingly, names serve to uniquely identify the writable version of a file, but not any read-only versions. Similarly, the file system monitors which files are currently open for writing, but not any read-only versions.

We can set a file to be writable with a function

\[ \text{WriteOpen} : \text{File} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \]

if a writable version does not already exist. This sets the write permission of the file, and updates the file system accordingly to reflect this change. We can also set the file to be open for reading only, with the function

\[ \text{ReadOpen} : \text{File} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}). \]

This sets a file to be read-only open, and if the file was previously writable, it updates the file system to record the change.

When a file is closed with an operation

\[ \text{Close} : \text{File} \times \text{FileSystem} \rightarrow \text{FileSystem}, \]

it is written back to the file system if the file has write permission set, and otherwise the file system is not altered.

We can change the name of a file with an operation

\[ \text{Rename} : \text{File} \times \text{Name} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \]
which is effective if the new name has not already been used.

We implement these functions using auxiliary operations:

\[ \text{InSystem} : \text{Name} \times \text{FileSystem} \to \mathbb{B} \]

that checks to see whether there is currently a file with the given name in the system;

\[ \text{Find} : \text{Name} \times \text{FileSystem} \to (\text{File} \cup \{\text{NotThere}\}) \]

that returns a file with a given name from the file system; and

\[ \text{Update} : \text{File} \times \text{FileSystem} \to \text{FileSystem} \]

that replaces a previous version of a file in the system with a new one, if the new version is writable.
algebra \( \text{FileSystem} \)

import \( \text{FileSystem} = \text{List}(\text{NameWritePermissionFiles}) \)

 carriers \( \text{File} \times \text{FileSystem} \)

constants

operations \( \text{Open} : \quad \text{Name} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \)
\( \text{WriteOpen} : \quad \text{File} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \)
\( \text{ReadOpen} : \quad \text{File} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \)
\( \text{Close} : \quad \text{File} \times \text{FileSystem} \rightarrow \text{FileSystem} \)
\( \text{Rename} : \quad \text{File} \times \text{Name} \times \text{FileSystem} \rightarrow (\text{File} \times \text{FileSystem}) \)
\( \text{InSystem} : \quad \text{Name} \times \text{FileSystem} \rightarrow \text{B} \)
\( \text{Find} : \quad \text{Name} \times \text{FileSystem} \rightarrow (\text{File} \cup \{\text{NotThere}\}) \)
\( \text{Update} : \quad \text{File} \times \text{FileSystem} \rightarrow \text{FileSystem} \)
\( \text{GetFile} : \quad (\text{File} \times \text{FileSystem}) \rightarrow \text{File} \)
\( \text{GetFileSystem} : \quad (\text{File} \times \text{FileSystem}) \rightarrow \text{FileSystem} \)

definitions

\[
\text{Open}(n, fs) = \begin{cases} 
(\text{Empty}(n), \text{AddFile}(\text{Empty}(n), fs)) & \text{if InSystem}(n, fs) = \text{ff}; \\
(\text{WriteOpen}(\text{Find}(n, fs))) & \text{if InSystem}(n, fs) = \text{tt} \wedge \text{Writable}(\text{Find}(n, fs)) = \text{ff}; \\
(\text{ReadOpen}(\text{Find}(n, fs)), fs) & \text{if InSystem}(n, fs) = \text{tt} \wedge \text{Writable}(\text{Find}(n, fs)) = \text{tt}; \\
\end{cases}
\]

\[
\text{WriteOpen}(n, s, p, o, fs) = \begin{cases} 
((n, s, p, o, \text{write}), \text{Update}((n, s, p, \text{write}), fs)) & \text{if } o \neq \text{write} \wedge \text{Writable}(\text{Find}(n, fs)) = \text{ff}; \\
((n, s, p, o), fs) & \text{otherwise.} \\
\end{cases}
\]

\[
\text{ReadOpen}(n, s, p, o, fs) = \begin{cases} 
((n, s, p, \text{read}), \text{Update}((n, s, p, \text{read}), fs)) & \text{if } o = \text{read}; \\
(n, s, p, o, fs) & \text{otherwise.} \\
\end{cases}
\]

\[
\text{Close}(f, fs) = \begin{cases} 
\text{Update}(f, fs) & \text{if } \text{Writable}(f) = \text{tt}; \\
fs & \text{otherwise.} \\
\end{cases}
\]

\[
\text{Rename}(n, s, p, o, m, fs) = \begin{cases} 
(n, s, p, o, \text{write}), \text{AddFile}(m, s, p, \text{write}) & \text{if } \text{InSystem}(m, s) = \text{ff}; \\
(n, s, p, o) & \text{otherwise.} \\
\end{cases}
\]

\[
\text{Update}(f, \text{EmptyFileSystem}) = \text{AddFile}(f, \text{EmptyFileSystem})
\]

\[
\text{Update}((n, s, p, o), \text{AddFile}((n', s', p', d'), fs)) = \begin{cases} 
\text{AddFile}((n, s, p, o), fs) & \text{if } n = n'; \\
\text{AddFile}((n', s', o'), \text{Update}((n, s, p, o), fs)) & \text{otherwise}. \\
\end{cases}
\]

\[
\text{InSystem}(n, \text{EmptyFileSystem}) = \text{ff}
\]

\[
\text{InSystem}(n, \text{AddFile}((n', s', p', d'), fs)) = \begin{cases} 
\text{tt} & \text{if } n = n'; \\
\text{ff} & \text{otherwise}. \\
\end{cases}
\]

\[
\text{Find}(n, \text{EmptyFileSystem}) = \text{NotThere}
\]

\[
\text{Find}(n, \text{AddFile}((n', s', p', d'), fs)) = \begin{cases} 
(n', s', p', d') & \text{if } n = n'; \\
\text{Find}(n, fs) & \text{otherwise}. \\
\end{cases}
\]

\[
\text{GetFile}(f, fs) = f
\]

\[
\text{GetFileSystem}(f, fs) = fs
\]
6.4 Time and Data: A Data Type of Infinite Streams

Making models of systems is a commonplace activity. In computing, models are used to help understand a user’s requirements for a system. Models are also used in making software, to clarify the implementation options and their consequences. An aim of this book is to show how models reveal and make precise the concepts that shape the languages and tools used for specifications and programming. In science and engineering, similarly, models are essential in discovering and understanding how a physical or biological system works. Systems are analysed mathematically, computationally and experimentally. Models are formulated in mathematical theories and in software to investigate the data the system processes, calculates or measures.

Data types play a prominent role in modelling all systems because to describe a system we need to define carefully

- the data that characterises the system, and
- the operations on data that define the behaviour of the system.

Modelling usually involves the operation or evolutions of the system in time. Ultimately, the data is made from basic data types we have met.

Therefore, in modelling systems in the world, whether natural systems, like a particle in motion or a heart, or artificial systems, like a compiler or a bank, we need an inexhaustible supply of data and data types. Moreover, to model how these systems operate, evolve or react in time we need to model

\[ \text{data distributed in time.} \]

This we will do using the idea of a stream and making an algebra that models a data type of streams. In the next section we will consider the fact that systems are extended in space and models involve data distributed in space.

6.4.1 What is a stream?

A stream is a sequence

\[ \ldots, a_t, \ldots \]

of data \(a_t\) indexed by time points \(t\). Time may be discrete, in which case typically it is modelled by the set \(\mathbb{N}\) of natural numbers, or possibly the set \(\mathbb{Z}\) of integers. Time may be continuous, in which case typically it is modelled by the set \(\mathbb{R}_+\) of positive real numbers, or possibly the set \(\mathbb{R}\) of all the real numbers. The sequences can be finite or infinite.

Processing streams of data is a truly fundamental activity in Computer Science and its applications.

Most computing systems operate in discrete time. The algorithms underlying computers, operating systems and networks process infinite streams of bits, bytes and words. The programs that are embedded in instruments to monitor and control machines that do the world’s work, are programmed to process infinite streams of Booleans, integers, reals and strings.
Many programs must be designed to compute forever, accepting an infinite stream of inputs and returning an infinite stream of outputs.

In contrast to the vast scope of programming with streams, our task here is simply to think about streams as a data type. For any data, we want to program with streams of that data. Thus, given any \( \Sigma \)-algebra \( A \), we show how to construct a signature \( \Sigma_{Stream} \) and a \( \Sigma_{Stream} \)-stream algebra

\[
A_{Stream}
\]

that models streams over \( A \). To do this, we first have to look at time.

### 6.4.2 Time

Time is a deeply complicated idea with a fascinating history. To glimpse its philosophical richness, consult Whitrow [1980]; to glimpse its amazing history, consult Borst [1993]. Time is completely fundamental to all forms of mechanisation, and certainly to computing.

Discrete time is thought of as either

- an ordered sequence of isolated time units
- a consecutive sequence of time intervals or cycles

as suggested in Figure 6.4, or

as suggested in Figure 6.5.

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & \ldots & t - 1 & t & t + 1 & \ldots \\
\bullet & \bullet & \bullet & \bullet & \ldots & \bullet & \bullet & \bullet & \ldots \\
\end{array}
\]

*Figure 6.4: Examples of models of discrete and continuous time*

*Figure 6.5: Time modelled as cycles.*

The idea is evident in our experience of a digital clock showing hours and minutes: the clock displays 08:59 for an interval (of one minute) and then displays 09:00 for an interval (of one minute). On first reading the clock, we know we are somewhere in the interval 08:59. In either case, the time instants or time cycles are counted using the natural numbers. Thus, discrete time is modelled by a counting process: there is an initial or \( 0^{th} \) time instant or cycle, and a function that returns the next \( t + 1^{th} \) instant or cycle from the \( t^{th} \) instant or cycle. A device that counts time is called a clock. This conception is easy to model as an algebra.
6.4. TIME AND DATA: A DATA TYPE OF INFINITE STREAMS

Signature/Interface The signature $\Sigma_{\text{Clock}}$ for our clock names time as a sort, an initial time value start as a constant, and a function tick to count the passing of time.

\[
\begin{align*}
\text{signature} & \quad \text{Clock} \\
\text{sorts} & \quad \text{time} \\
\text{constants} & \quad \text{start} : \rightarrow \text{time} \\
\text{operations} & \quad \text{tick} : \text{time} \rightarrow \text{time}
\end{align*}
\]

This is a basic set of operations on time; some other useful operations on time concern the ordering of time.

Algebra/Implementation We implement the signature $\Sigma_{\text{Clock}}$ with a standard clock modelled by the algebra $A_{\text{Clock}}$:

\[
\begin{align*}
\text{algebra} & \quad \text{Clock} \\
\text{carriers} & \quad \mathbb{N} \\
\text{constants} & \quad \text{Start} : \rightarrow \mathbb{N} \\
\text{operations} & \quad \text{Tick} : \mathbb{N} \rightarrow \mathbb{N} \\
\text{definitions} & \quad \\
\text{Start} & = 0 \\
\text{Tick}(t) & = t + 1
\end{align*}
\]

Other clocks are also interesting and provide different interpretations of $\Sigma_{\text{Clock}}$. For example, a finite counting system like cyclic arithmetic $\mathbb{Z}_n$ is of use. A timer for cooking might display only minutes and be based on a $\Sigma_{\text{Clock}}$-algebra with carrier $\mathbb{Z}_{60}$.

Internally, it could also measure seconds and be based on the data $\mathbb{Z}_{60} \times \mathbb{Z}_{60}$.

The standard digital clock measures hours, minutes and seconds, and is implemented as an algebra with carrier $\mathbb{Z}_{24} \times \mathbb{Z}_{60} \times \mathbb{Z}_{60}$.

Notice these clocks require extra operations to be added to $\Sigma_{\text{Clock}}$. The operation of $\Sigma_{\text{Clock}}$ counts in only one unit of time.
6.4.3 Streams of Elements

Let us first clarify some mathematical ideas about streams and sequences.

**Definition (Streams)** Let \( A \) be any non-empty set. An *infinite sequence* or *stream* over \( A \) is a function

\[
a : \mathbb{N} \to A.
\]

For \( t \in \mathbb{N} \), the \( t^{th} \) element of the sequence is \( a(t) \) and sequences are often written

\[
a = a(0), a(1), \ldots \quad \text{or} \quad a = a_0, a_1, \ldots
\]

Let the set of all infinite sequences over \( A \) be denoted by

\[
[\mathbb{N} \to A].
\]

The term stream is appropriate when the set \( \mathbb{N} \) models a set of time cycles.

**Old Signatures/Interface** We shall need two signatures to construct streams. We shall need a signature \( \Sigma \) for the data that we shall store in the streams, and we shall need a signature \( \Sigma_{\text{Clock}} \) to count the passage of time.

**New Signature/Interface** We make a new signature \( \Sigma_{\text{Stream}} \) from the signatures \( \Sigma \) for the data elements and \( \Sigma_{\text{Clock}} \) for the clock.

```
signature Stream
import Name, Clock
sorts \ldots , stream(s), \ldots
constants
operations \ldots , read_s : \ stream(s) \times time \to s, \ldots
\ldots , write_s : s \times stream(s) \times time \to stream(s), \ldots
```

Depending upon the application, other names for the operations \( \text{read}_s \) and \( \text{write}_s \) may be more appropriate, such as

\( \text{send}_s \) and \( \text{receive}_s \)

or

\( \text{evaluate}_s \) and \( \text{update}_s \).

6.4.4 Algebra/Implementation

**Old Algebra/Implementation** We make a new algebra \( A_{\text{Stream}} \) from the algebras \( A \) that implements the signature \( \Sigma \) for the data elements, and \( A_{\text{Clock}} \) for the clock.
6.4. TIME AND DATA: A DATA TYPE OF INFINITE STREAMS

New Algebra/Implementation We make a $\Sigma_{\text{Stream}}$-algebra $A_{\text{Stream}}$ from any $\Sigma$-algebra $A$ and the clock $A_{\text{Clock}}$:

| algebra | $Stream$ |
| import | $Name, Clock$ |
| carriers | $\ldots,[N \rightarrow A_s], \ldots$ |
| constants | $\ldots, \text{Read}_s : N \times [N \rightarrow A_s] \rightarrow A_s, \ldots$ |
| operations | $\ldots, \text{Write}_s : A_s \times N \times [N \rightarrow A_s] \rightarrow [N \rightarrow A_s], \ldots$ |
| definitions | $\ldots, \text{Read}_s(a,t) = a(t), \ldots$ |
| | $\ldots, \text{Write}_s(x,t,a)(t') = \begin{cases} x & \text{if } t = t'; \\ a(t') & \text{if } t \neq t'. \end{cases}$ |

This algebra $A_{\text{Stream}}$ simply involves infinite sequences as data, the elements of which may be read by means of the new operations. There are several other operations that may be added to $A_{\text{Stream}}$ to model other uses of infinite sequences, for example, operations that insert data without losing data, translate data in time, or merge two streams (see Exercises).

6.4.5 Finitely Determined Stream Transformations and Stream Predicates

Imagine a computing system receiving data from, and returning data to, some environment. Suppose the system operates in discrete time measured by a discrete clock with time cycles

$$ T = \{0, 1, 2, \ldots\}. $$

Let the input data come from a set $A$ and the output data from a set $B$. We investigate the input-output behaviour of the system over time when it takes one of the following two forms:

1. Stream Transformer The system operates continuously in time and inputs a stream

$$ a = a(0), a(1), a(2), \ldots, a(t), \ldots \in [T \rightarrow A] $$

of data from $A$ and outputs a stream

$$ b = b(0), b(1), b(2), \ldots, b(t), \ldots \in [T \rightarrow B] $$

of data from $B$. Thus, the input-output behaviour of the system can be modelled by a stream transformation

$$ F : [T \rightarrow A] \rightarrow [T \rightarrow B]. $$
2. **Stream Predicate** The system operates continuously in time and inputs a stream
\[ a = a(0), a(1), a(2), \ldots, a(t), \ldots \in [T \to A] \]
of data from \( A \) and outputs a single truth value
\[ b \in B \]
from \( B = \{tt, ff\} \). Thus, the input-output behaviour of the system can be modelled by a stream predicate
\[ F : [T \to A] \to B. \]

Now computing systems are built from

*algorithms that process data in finitely many steps.*

This suggests a property of stream processing:

The output of a stream transformer
\[ F(a)(t) \]
at time \( t \) given input stream \( a \), and the output of a stream predicate
\[ F(a) \]
is computed in finitely many steps by algorithms that can access at most finitely many elements
\[ a(0), a(1), a(2), \ldots, a(l) \]
of the input stream \( a \). In the case of stream transformers, the number \( l \) depends on \( a \) and \( t \); in the case of stream predicates, \( l \) depends on \( a \).

This property is called *finite determinacy* and is shown in Figure 6.6. It has the following formal definition (which drops references to algorithms).

\[
\begin{array}{c}
\ldots, a(l+2), \ldots, a(l+1) \\
\text{system} \\
\{a(l), \ldots, a(1), a(0)\} \\
\text{output} \\
b(t), \ldots, b(1), b(0)
\end{array}
\]

Figure 6.6: System with finite determinacy principle.

**Definition (Finite Determinacy)**

1. A stream transformer \( F : [T \to A] \to [T \to B] \) is *finitely determined* if, given any input stream \( a \in [T \to A] \) and any time \( t \in T \), there exists an \( l \in T \) such that for every stream \( b \in [T \to A] \), if \( a \) and \( b \) agree up to time \( l \), i.e.,
\[ a(0) = b(0), a(1) = b(1), a(2) = b(2), \ldots, a(l) = b(l) \]
then
\[ F(a)(t) = F(b)(t). \]
2. A stream predicate \( F : [T \to A] \to B \) is *finitely determined* if, given any input stream \( a \in [T \to A] \) there exists an \( l \in T \) such that for every stream \( b \in [T \to A] \), if \( a \) and \( b \) agree up to time \( l \), i.e.,

\[
a(0) = b(0), a(1) = b(1), a(2) = b(2), \ldots, a(l) = b(l)
\]

then

\[
F(a) = F(b).
\]

An argument about the behaviour of computing systems and the rôle of algorithms was used to motivate the definition of this property of stream transformations and predicates. The argument actually suggests a hypothesis or thesis:

**Thesis** If a stream transformation or predicate is computable by algorithms then it is finitely determined. In particular, if a stream transformation or predicate is not finitely determined then it is not computable by algorithms.

**Examples**

Consider processing streams of integers.

1. Let \( F : [T \to \mathbb{Z}] \to [T \to \mathbb{Z}] \) be defined by

\[
F(a)(t) = a(0) + a(1) + a(2) + \cdots + a(2t).
\]

This is finitely determined because for any time \( t \) and streams \( a, b \), we have

\[
a(0) = b(0), a(1) = b(1), a(2) = b(2), \ldots, a(2t) = b(2t) \Rightarrow F(a)(t) = F(b)(t).
\]

2. Let \( Z : [T \to \mathbb{Z}] \to \mathbb{B} \) be the zero test defined by

\[
Z(a) = \begin{cases} 
    tt & \text{if } a = 0, 0, 0, \ldots, 0, \ldots; \\
    ff & \text{if } a \neq 0, 0, 0, \ldots, 0, \ldots.
\end{cases}
\]

Now, making these conditions explicit, we have

\[
Z(a) = \begin{cases} 
    tt & \text{if } (\forall t)[a(t) = 0]; \\
    ff & \text{if } (\forall t)[a(t) \neq 0].
\end{cases}
\]

The predicate \( Z \) is not finitely determined. Here is the proof.

Suppose for a contradiction it was finitely determined. Then, given a stream \( a \), there exists a bound \( l \) such that for any stream \( b \),

\[
a(0) = b(0), a(1) = b(1), a(2) = b(2), \ldots, a(l) = b(l), \Rightarrow Z(a) = Z(b).
\]

Choose \( a \) to be 0 everywhere and \( b \) that is not 0 only at time \( l + 1 \). Clearly,

\[
a(0) = b(0), a(1) = b(1), a(2) = b(2), \ldots, a(l) = b(l) \text{ but } Z(a) = tt \text{ and } Z(b) = ff.
\]

This contradicts the finite determinacy property.
6.4.6 Decimal Representation of Real Numbers

The real numbers are a number system that we use to measure quantities exactly. To measure
the length of a line or the circumference of a circle, we encounter numbers like
\[ \sqrt{2} \quad \text{and} \quad \pi \]
that are not rational numbers. The rational numbers are a number system that we use to measure quantities

*exactly, using a ruler or gauge divided into units.*

The real numbers are created by modelling the process of measurement using rational numbers. We use them to measure quantities

*accurately or exactly to any degree of precision.*

A real number is an abstraction from these measuring processes. Recall the discussion of real numbers in Chapter 3.

There are many ways of modelling this process and, hence, representing real numbers. Later, we will devote a whole chapter to the subject (Chapter 8) and these processes are examples of streams. Here, we will investigate the standard notation for real numbers, namely:

*infinite decimals.*

We will explain the method and show how to represent infinite decimals as streams. Then we will prove that they are unsuited to exact computation to arbitrary accuracy.

First, here are some real numbers and their infinite decimal representations:

\[
\begin{align*}
1 & \quad 1.000 \ldots 0 \ldots \\
1 & \quad 0.999 \ldots 9 \ldots \\
\frac{1}{3} & \quad 0.333 \ldots 3 \ldots \\
\sqrt{2} & \quad 1.41421356 \ldots \\
\pi & \quad 3.14159265 \ldots \\
\frac{22}{7} & \quad 3.14285714 \ldots \\
\end{align*}
\]

Most real numbers have one, and only one, infinite decimal representation. For example,

\[0.333 \ldots 3 \ldots \quad \text{and} \quad 0.666 \ldots 6 \ldots\]

are the unique decimal representations of \(\frac{1}{3}\) and \(\frac{2}{3}\), respectively. However,

\[1.000 \ldots 0 \ldots \quad \text{and} \quad 0.999 \ldots 9 \ldots\]

are two different decimal representations of 1.

In calculation, we use some finite part of the decimal representation as an approximation. For example, 1.4142 approximates \(\sqrt{2}\). The finite decimal notation

\[1.4142\]

stands for the sum

\[
1 + \frac{4}{10} + \frac{1}{100} + \frac{4}{1000} + \frac{2}{10000}
\]
6.4. TIME AND DATA: A DATA TYPE OF INFINITE STREAMS

of rational numbers, and is a rational number that measures using the base 10 units

\[
\frac{1}{10^0}, \frac{1}{10^1}, \frac{1}{10^2}, \frac{1}{10^3}, \frac{1}{10^4}.
\]

However, the infinite decimal notation

\[1.41421356\ldots\]

stands for the process of summing

\[
\frac{1}{10^0} + \frac{4}{10^1} + \frac{1}{10^2} + \frac{4}{10^3} + \frac{2}{10^4} + \frac{1}{10^5} + \frac{3}{10^6} + \frac{5}{10^7} + \frac{6}{10^8} + \cdots
\]

the rational numbers that measure using all possible base 10 units.

The term decimal can be confusing because it is used to denote two aspects of number notations:

(i) the use of base 10, and

(ii) the use of the so-called decimal point.

In this section, it is the use of the decimal point that attracts our attention.

**Definition (Decimal Representation of Reals)** A decimal representation of a real number has the form

\[b_mb_{m-1}\cdots b_1 b_0 . a_1 a_2 a_3 \cdots a_n \cdots\]

where

(i) \(m\) is a natural number, \(m \in \mathbb{N}\), and

(ii) \(b_m, b_{m-1}, \ldots, b_1, b_0, a_1, a_2, a_3, \ldots, a_n, \ldots \in \{0, 1, \ldots, 9\}\) are digits.

The notation stands for

\[b_m 10^m + b_{m-1} 10^{m-1} + \cdots + b_1 10^1 + b_0 10^0 + a_1 10^{-1} + a_2 10^{-2} + \cdots + a_n 10^{-n} + \cdots\]

We will show that computing with decimal representations of real numbers is problematic. For simplicity in notation, we will examine the problem of computing with decimal representations of real numbers in the closed interval

\[[0, 10] = \{r \in \mathbb{R} \mid 0 \leq r \leq 10\}\]

The numbers in \([0, 10]\) have a simple decimal representation that can easily be represented by streams as follows.

**Definition (Stream Representation of Decimals)** A real number \(x \in [0, 10]\) has a decimal representation of the simpler form

\[b_0 . a_1 a_2 a_3 \cdots a_n \cdots\]

where \(b_0, a_1, a_2, a_3, \ldots, a_n, \ldots \in \{0, \ldots, 9\}\). In particular, there is just one digit before the decimal point.
There is a trivial way to view or represent such decimals as streams, namely:

\[ a = b_0, a_1, a_2, a_3, \ldots a_n, \ldots \]

Thus, the set of decimals, and hence the reals in \([0, 10]\), can be represented using the set

\[ [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \]

of all streams of digits.

**Definition (Stream Representations of Functions)** A function

\[ f : [0, 10] \rightarrow [0, 10] \]

on real numbers is represented by a stream transformer

\[ F : [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \rightarrow [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \]

of streams of digits if, for any real number \( x \in [0, 10] \) and stream \( a \in [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \), we have that if the stream

\( a \) represents the decimal expansion of \( x \)

then the stream

\( F(a) \) represents the decimal expansion of \( f(x) \).

We restrict the domain of our function \( f(x) = 3x \) to the subset

\[ [0, 1] = \{ r \in \mathbb{R} \mid 0 \leq r \leq 1 \} \]

so that the range of \( f \) remains in \([0, 10]\). (Note that if \( 0 \leq x \leq 1 \) then \( 0 \leq f(x) \leq 3 \).)

**Theorem** The function \( f : [0, 1] \rightarrow [0, 10] \) defined by

\[ f(x) = 3x \]

for \( x \in [0, 1] \) cannot be defined by a finitely determined stream transformation

\[ F : [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \rightarrow [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \]

of the decimal representation of real numbers.

**Proof** Let the stream transformer

\[ F : [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \rightarrow [\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}] \]

calculate the function \( f(x) = 3x \) using decimals, i.e.,

stream \( a \) represents real \( x \in [0, 1] \) implies stream \( F(a) \) represents real \( 3x \).
6.4. TIME AND DATA: A DATA TYPE OF INFINITE STREAMS

Suppose for a contradiction that \( F \) is finitely determined. This means that for any stream \( a \) and output position \( k \) there is an input position \( l \) such that for all streams \( b \)

\[
a(i) = b(i) \text{ for } 0 \leq i \leq l \Rightarrow F(a)(k) = F(b)(k).
\]

We derive the contradiction by investigating the way \( F \) calculates \( 3x \) for \( x = \frac{1}{3} \). The input \( \frac{1}{3} \) has the unique decimal representation

\[
0.333\ldots
\]

and as an input stream is

\[
a = 0, 3, 3, 3, \ldots.
\]

However, the output \( f\left(\frac{1}{3}\right) = 3 \cdot \frac{1}{3} = 1 \) has two decimal representations

\[
0.999\ldots \quad \text{or} \quad 1.000\ldots,
\]

and so the output stream is either

\[
F(a) = 0, 9, 9, 9, \ldots \quad \text{or} \quad F(a) = 1, 0, 0, 0, \ldots.
\]

The argument divides into two cases.

Case 1 \( F(a) = 0, 9, 9, 9, \ldots \)

By the assumption that \( F \) is finitely determined, for \( k = 0 \) there is \( l \) such that for any stream \( b \)

\[
a(0) = b(0) = 0 \text{ and } a(i) = b(i) = 3 \text{ for } 1 \leq i \leq l \Rightarrow F(a)(0) = F(b)(0) = 0
\]

Choose a stream \( b \) with 9 at position \( l + 1 \) and thereafter, i.e.,

\[
b = 0, 3, 3, 3, \ldots, 3, 9, 9, 9, \ldots.
\]

By choice, \( a(i) = b(i) \) for \( 0 \leq i \leq l \). However, since

\[
3 \cdot 0.333\ldots 3999\ldots = 1.000\ldots 02\ldots
\]

we see that \( F(b)(0) = 1 \); in particular, although \( a \) and \( b \) agree up to \( l \),

\[
0 = F(a)(0) \neq F(b)(0) = 1.
\]

This contradicts the assumption that \( F \) is finitely determined.

Case 2 \( F(a) = 1, 0, 0, 0, \ldots \)

A similar argument leads to a contradiction. (Exercise.) \( \square \)

Corollary The function \( f : \mathbb{R} \rightarrow \mathbb{R} \) defined by \( f(x) = 3x \) cannot be computed by an algorithm using the decimal representation of the real numbers.

Proof Suppose there was an algorithm that computed \( f(x) = 3x \) using decimal representations. Then the input-output behaviour of the algorithm restricted to \([0, 1]\) can be defined by a stream transformation \( F \) on streams \([\mathbb{N} \rightarrow \{0, 1, \ldots, 9\}]\). Because the algorithm determines the \( n^{th} \) decimal place in finitely many steps \( l \), the stream transformer \( F \) is finitely determined. However, by the Theorem, no such finitely determined stream transformer exists. \( \square \)

Thus, given the Thesis, very little can be computed using infinite decimals.
6.5 Space and Data: A Data Type of Spatial Objects

In Section 6.4 we met streams. Streams are

\[ \text{data distributed in time,} \]

and typical examples of streams are the input and output of interactive systems. We constructed a simple algebra to model a data type of streams.

Now, in this section, we will model

\[ \text{data distributed in space.} \]

Here are some examples to shape our thinking:

- In computer graphics, data in three dimensions for transformation and visualisation (e.g., the data from medical scanners such as CT, NMR).
- In scientific modelling, states of a physical system, including both computed data from mathematical models and output from measuring instruments.
- In data storage media, states of both hardware (e.g., memories and disks) and software (e.g., arrays and other data structures).

Space and time are combined in animations of graphical data, in simulations of physical systems, and in the operation of storage media.

To model data distributed in space we will define the idea of a

\[ \text{spatial object or spatial data field} \]

and define operations to make data types of spatial objects. We will apply this general concept by making an algebra of use in Volume Graphics.

6.5.1 What is space?

Ideas of space are as complicated as notions of time. The ideas have great philosophical, physical and mathematical depth. In simple terms, we think of a space as a set of points. We often equip the space with some form of coordinate or addressing system to locate points. Indeed, it is common to identify points with their representation by coordinates in a coordinate system for the space. For example, commonly, we think of a point in the 2 dimensional plane as a pair \((x, y)\) of numbers in an orthogonal coordinate system. In particular, we think of a point in terms of some representation of the plane. Of course, there are infinitely many coordinate systems to choose from — we can simply vary the choice of origin, or the direction of the x-axis. With each choice, the pair \((x, y)\) of numbers will represent a different point. More radically, we can drop the condition that the axes are at right-angles, or are even straight lines. For many tasks we use polar coordinates. There are some more examples of systems in Figure 6.7.
<table>
<thead>
<tr>
<th>Space</th>
<th>Coordinate/Addressing system</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 dimensional continuous physical space</td>
<td>Real number coordinate system based on</td>
</tr>
<tr>
<td></td>
<td>● Cartesian coordinates, or</td>
</tr>
<tr>
<td></td>
<td>● polar coordinates</td>
</tr>
<tr>
<td></td>
<td>based on real numbers.</td>
</tr>
<tr>
<td>3 dimensional continuous physical space</td>
<td>Real number coordinate system based on</td>
</tr>
<tr>
<td></td>
<td>● Cartesian coordinates,</td>
</tr>
<tr>
<td></td>
<td>● polar coordinates, or</td>
</tr>
<tr>
<td></td>
<td>● cylindrical coordinates</td>
</tr>
<tr>
<td></td>
<td>based on real numbers.</td>
</tr>
<tr>
<td>2 dimensional discrete physical space</td>
<td>Integer number coordinate system based on</td>
</tr>
<tr>
<td></td>
<td>● Cartesian grid coordinates</td>
</tr>
<tr>
<td></td>
<td>based on integers.</td>
</tr>
<tr>
<td>3 dimensional discrete physical space</td>
<td>Integer number coordinate system based on</td>
</tr>
<tr>
<td></td>
<td>● Cartesian grid coordinates</td>
</tr>
<tr>
<td></td>
<td>based on integers.</td>
</tr>
<tr>
<td>Data storage media</td>
<td>Indexing addresses based on integers and strings</td>
</tr>
<tr>
<td>Internet space</td>
<td>Addresses such as URLs and IP addresses based on integers and strings</td>
</tr>
<tr>
<td>Static and mobile telephony space</td>
<td>Addresses such as phone and device numbers and physical coordinates based on integers and strings</td>
</tr>
</tbody>
</table>

Figure 6.7: Spaces and their coordinates.
The idea of space has been analysed in great depth in mathematics. It started with the ancient Greek view of geometry expounded in the 13 books of Euclid (c.300 BC). Here the points and lines, the circles and other objects they make up are created by algorithmic operations with ruler and compass. In the 17th century, a new view of space emerged that was essentially algebraic: in coordinate geometry, axes were used and objects like conic sections defined by equations, and calculations performed using the differential calculus. Again and again, new views of space are discovered and new theories are developed.

For our simple purposes we can define the notion of space rather simply:

**Definition (Space)** A *space* is a non-empty set \( X \), the elements of which are called *points*.

Depending on the example or application we may call points *locations* or *sites*. We will also identify spaces with coordinate systems.

**Example Continuous Space** Usually, we represent the 2 dimensional continuous plane by taking \( X = \mathbb{R}^2 \) where

\[
\mathbb{R}^2 = \{(x, y) \mid x, y \in \mathbb{R}\},
\]

and 3-dimensional space by taking \( X = \mathbb{R}^3 \) where

\[
\mathbb{R}^3 = \{(x, y, z) \mid x, y, z \in \mathbb{R}\}.
\]

We represent the \( n \)-dimensional continuous space by taking \( X = \mathbb{R}^n \) where

\[
\mathbb{R}^n = \{(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in \mathbb{R}\}.
\]

**Discrete Space** We represent a discrete plane by the 2-dimensional integer grid, taking \( X = \mathbb{Z}^2 \) where

\[
\mathbb{Z}^2 = \{(i, j) \mid i, j \in \mathbb{Z}\}.
\]

We represent a discrete 3 dimensional space by the 3-dimensional integer grid taking \( X = \mathbb{Z}^3 \) where

\[
\mathbb{Z}^3 = \{(i, j, k) \mid i, j, k \in \mathbb{Z}\}.
\]

We represent the \( n \)-dimensional discrete space by taking \( X = \mathbb{Z}^n \) where

\[
\mathbb{Z}^n = \{(x_1, \ldots, x_n) \mid x_1, \ldots, x_n \in \mathbb{Z}\}.
\]

**Address Space** We can represent the space \( X \) of addresses of a 1-dimensional array by taking

\[
X = \mathbb{N} \text{ or } X = \mathbb{Z}.
\]

Similarly, the address space of a 2 dimensional array by taking

\[
X = \mathbb{N}^2 \text{ or } X = \mathbb{Z}^2,
\]

the address space of a 3 dimensional array by taking

\[
X = \mathbb{N}^3 \text{ or } X = \mathbb{Z}^3,
\]

and the address space of an \( n \) dimensional array by taking

\[
X = \mathbb{N}^n \text{ or } X = \mathbb{Z}^n.
\]
6.5. **SPACE AND DATA: A DATA TYPE OF SPATIAL OBJECTS**

**Identifiers** In high level programming, variables and identifiers can be chosen to suit the program at hand. Typically, they are strings of symbols chosen from the English alphabet and digits. The variables specify locations where data is stored. In particular, the set \( X \) can be seen as a *space of names* that address a memory.

Since the late 19th century, ideas about space start from the idea of a set. Hence the ideas are abstract and general. Thinking about sets of points abstractly, without using a coordinate system representation, lead to concepts such as

*metric spaces,*

which are based on axiomatically specifying the process of measuring the distance between points, or the more abstract

*topological spaces,*

which are based on axiomatising the idea of subsets of neighbouring points. With such abstract views of space we can more thoroughly analyse ideas of space in ways that are independent of their representation in coordinate systems.

### 6.5.2 Spatial objects

Physical processes and abstract objects are modelled or measured by assigning some data to each point of a space. The basic idea of a spatial object is that any kind of data can be distributed all over any chosen space \( X \).

**Definition (Spatial object)** Let \( X \) be a non-empty set representing a space. Let \( A_1, \ldots, A_m \) be sets of data. This data we call the *attributes* of the points in the space. Let

\[
A = A_1 \times \cdots \times A_m.
\]

Then a *spatial data object* or *spatial data field* is represented by a map

\[
o : X \rightarrow A,
\]

which assigns attributes to points, i.e.,

\[
o(x) = \text{the } m \text{ data attributes characterising the object at point } x \in X.
\]

Let \( O(X, A) \) be the set of all spatial objects.

Since \( A = A_1 \times \cdots \times A_m \) there are \( m \) attributes of \( o(x) \), each of which depends on the point \( x \). Specifically, there are \( m \) functions

\[
a_1 : X \rightarrow A_1, \ldots, a_m : X \rightarrow A_m,
\]

to compute the \( m \) attributes independently, and we can write

\[
o(x) = (a_1(x), \ldots, a_m(x)).
\]

These functions \( a_1, \ldots, a_m \) we call the attribute fields or components of the spatial object \( o \).

This idea of a spatial object is *very general.*
Example An RGB Colour Model Consider data for the visualisation of 3-dimensional continuous space. Let the space \( X \) be 3-dimensional space \( \mathbb{R}^3 \). Suppose for each point \( p \in \mathbb{R}^3 \).

There are 4 attributes: one representing visibility by

\[ \text{op} \] the opacity of the point \( p \) in space;

and three representing colour by

\[ r \] the red value of the point \( p \) in space;

\[ g \] the green value of the point \( p \) in space;

\[ b \] the blue value of the point \( p \) in space.

The opacity \( \text{op} \) is measured on a real number scale from 0 to 1, i.e., the interval

\[ [0, 1] = \{ r \in \mathbb{R} \mid 0 \leq r \leq 1 \}. \]

Here the value 0 means that the point is transparent and the value 1 that it is opaque; values in between mean the point is translucent. The three colour values can also measured using real numbers in the interval \([0, 1]\).

So specifying the appearance of point \( p \in \mathbb{R}^3 \), we have a 4-tuple

\[(\text{op}, r, g, b)\]

of real numbers, each from the interval \([0, 1]\). Thus, in this case, with \( X = \mathbb{R}^3 \), as the space, and \( A = [0, 1]^4 \), as the set of attributes, a spatial data object

\[ o : X \to A \]

is a map

\[ o : \mathbb{R}^3 \to [0, 1]^4 \]

defined at each point \( p \in \mathbb{R}^3 \) by its attribute fields

\[ o(p) = (\text{op}(p), r(p), g(p), b(p)). \]

**n-dimensional attributes** Let the space \( X \) be 3-dimensional Euclidean space \( \mathbb{R}^3 \). Suppose there are \( n > 0 \) attributes for each point \( p \) in space, and each attribute is also a real number; then let \( A \) be \( n \)-dimensional space \( \mathbb{R}^n \). Here we have an \( n \)-tuple of real numbers as data measuring some properties at point \( p \). Thus, in this case, each spatial data object

\[ o : X \to A \]

is a map

\[ o : \mathbb{R}^3 \to \mathbb{R}^n \]

defined by \( n \) attribute fields

\[ a_1, \ldots, a_n : \mathbb{R}^3 \to \mathbb{R} \]

at each point \( p \in \mathbb{R}^3 \) by

\[ o(p) = (a_1(p), \ldots, a_n(p)). \]
6.5. **SPACE AND DATA: A DATA TYPE OF SPATIAL OBJECTS**

**States of systems** Consider any system or process extended in space and changing in time. The behaviour of the system is commonly described using states. States provide snapshots of the system at time instants by specifying relevant properties of each point in space. A sequence of states is used to describe the behaviour of the system as it operates or evolves in time. If \( X \) is a space and \( A \) is the data of interest in the model of the system then a state is a map of the form \( \sigma : X \rightarrow A \). Thus, states can be seen as an example of a spatial object.

**Identifiers** Let \( Var \) be a set of variables and let \( A \) be a set of data then the assignment of data to variables has the form

\[
a : Var \rightarrow A.
\]

The assignment of data \( A \) in memory to a space \( Var \) of names can also be seen as a spatial object.

### 6.5.3 Operations on Spatial Objects

Next we look at some operations that transform spatial objects. The combination of some set of spatial objects and some operations is a data type and is modelled by an algebra. These data types are designed to help process spatial objects in particular applications.

The simplest kind of operation on objects is one derived from an operation on attributes, or on space.

Suppose we have some binary operation

\[
f : A^2 \rightarrow A
\]

on the data attributes. Then we can define a new binary operation

\[
F : O(X, A)^2 \rightarrow O(X, A)
\]

on spatial data objects by

“applying the operation \( f \) to data at all points the space \( X' \)’.”

This is called a **pointwise extension** or **lifting** of the operation \( f \) and is defined as follows.

**Definition (Pointwise Attribute Operations)** Suppose

\[
o_1 : X \rightarrow A \text{ and } o_2 : X \rightarrow A
\]

are any objects. Then we define the new object

\[
F(o_1, o_2) : X \rightarrow A
\]

by applying \( f \) to the data at each point \( x \in X \) as follows:

\[
F(o_1, o_2)(x) = f(o_1(x), o_2(x))
\]
Example Let $X$ be any space. Let the attributes $A = [0, 1]$. Let the function $f$ on the attributes be

$$\max : [0, 1]^2 \to [0, 1]$$

defined by

$$\max(s_1, s_2) = \begin{cases} s_1 & \text{if } s_1 \geq s_2; \\ s_2 & \text{if } s_1 < s_2. \end{cases}$$

Then the pointwise extension

$$\Max : O(X, [0, 1])^2 \to O(X, [0, 1])$$

of $\max$ is defined for all spatial objects $o_1, o_2 \in O(X, [0, 1])$ at point $x \in X$ by

$$\Max(o_1, o_2)(x) = \max(o_1(x), o_2(x)).$$

The function $\Max$ simply makes a new spatial object $\Max(o_1, o_2)$ which has the highest value of the two spatial objects $o_1$ and $o_2$ at each point in $X$. We will use this operation on opacity shortly.

In the definition above we have used the notation $A$ for the attributes. In most cases, this set $A$ is built from $m$ attributes and

$$A = A_1 \times \cdots \times A_m.$$ 

Although the definition of the lifting is straightforward, as in the above definition, this fact complicates the form of the operations $f$. Clearly, substituting for $A$, the operation

$$f : A^2 \to A$$

is actually of the form

$$f : (A_1 \times \cdots \times A_m) \times (A_1 \times \cdots \times A_m) \to (A_1 \times \cdots \times A_m)$$

This means that to define an operation $f$ we define its $m$ component operations

$$f_1 : (A_1 \times \cdots \times A_m) \times (A_1 \times \cdots \times A_m) \to A_1,$$

$$\vdots$$

$$f_m : (A_1 \times \cdots \times A_m) \times (A_1 \times \cdots \times A_m) \to A_m.$$ 

which calculate the $m$ values of each attribute $f$. Thus, for $a, b \in A$,

$$f(a, b) = (f_1(a, b), \ldots, f_m(a, b)).$$

We will see some examples of this shortly.
6.5.4 Volume Graphics and Constructive Volume Geometry

In the world, the objects and scenes we see are 3 dimensional. Behind the surfaces of clothes and skin are flesh and bone. A cloud, or a fire, fills space with water, or hot gasses. In computer graphics, historically and currently, standard techniques represent 3 dimensional objects using their 2 dimensional surfaces.

The subject of volume graphics assumes that all objects are represented fully in 3 dimensions. Objects and scenes are built and rendered using data at points or small volumes of 3 dimensional space, called voxels, a term intended to contrast with the 2 dimensional notion of pixel.

Volume graphics originates in techniques for the visualisation of 3 dimensional arrays of data. In medical imaging, physical objects are measured by scanning instruments that produce a data file of measurements that approximate a 3 dimensional spatial object. In clinical applications, the aim is to visualise these digitalisations of bone and tissue for diagnosis and treatment. In physiology, for example, carefully produced digitised hearts allow us to use algorithms to perform digital dissections to explore anatomical structure; they are also needed to explore function by computational simulations and experimental measurements.

Clearly, all kinds of operations are needed to process such data sets. A fundamental general task is to combine a number of data sets to make more complex scenes. In physiology, for example, an aim is to integrate data from dissection, simulation and experiment to model the whole heart beating. In this cardiac application alone there is plenty of scope for discovering new operations.

Here we introduce a little volume graphics to illustrate the ideas about data distributed in space. In particular, the 3 dimensional data sets are examples of spatial objects. We define some operations by pointwise extensions, and give an algebra of spatial objects for modelling in volume graphics and visualisation. Our aim is show a simple data type at work in a very complex application.

However, in the case of volume graphics, there are many applications and so there are many possible attributes to use, operations to define, and algebras to make. Indeed, the creation of a wide variety of data types of spatial objects constitutes a very general, high-level, algebraic approach to Volume Graphics and is called Constructive Volume Geometry (CVG).

Spatial Objects

Why are spatial objects needed in volume graphics? Raw volumetric field data, such as a digitised heart, is discrete and bounded in 3 dimensional space. These properties turn out to be an obstacle to a consistent specification of operations on the data sets. In order to create data types with such operations, we introduce the idea of spatial objects, which, because they are defined at every point in 3 dimensional space, makes them more inter-operable.

We have already introduced some spatial objects for volume graphics in the example of Section 6.5.2. Recall that they are three dimensional objects with the four attributes of opacity, red, and green, and blue, and have the form

\[ o : \mathbb{R}^3 \rightarrow [0, 1]^4. \]

Each spatial object \( o \) is made from a 4-tuple of attribute fields

\[ op, r, g, b : \mathbb{R}^3 \rightarrow [0, 1], \]
and is defined at each point \( p \in \mathbb{R}^3 \) by
\[
o(p) = (\text{op}(p), r(p), g(p), b(p)).
\]

Let \( O(\mathbb{R}^3, [0, 1]^4) \) be the set of all such objects.

The opacity field “implicitly” defines the “visibility” of the object. Any point \( p \) that is not entirely transparent, i.e., \( \text{op}(p) \neq 0 \), is potentially visible to a rendering algorithm.

**Operations**

The aim is to define an CVG algebra of spatial objects based on the opacity and RGB model. Here is its signature:

\[
\begin{array}{|c|c|}
\hline
\text{signature} & \text{RGBalgebra} \\
\text{sorts} & sp\_obj \\
\text{constants} & \\
\hline
\text{operations} & \cup : \text{sp\_obj} \times \text{sp\_obj} \rightarrow \text{sp\_obj} \\
& \cap : \text{sp\_obj} \times \text{sp\_obj} \rightarrow \text{sp\_obj} \\
& - : \text{sp\_obj} \times \text{sp\_obj} \rightarrow \text{sp\_obj} \\
& + : \text{sp\_obj} \times \text{sp\_obj} \rightarrow \text{sp\_obj} \\
\hline
\end{array}
\]

Any algebra with this signature has a single sort \( sp\_obj \) and four binary operations. The algebra we are building interprets the sort \( sp\_obj \) with the set
\[
O(\mathbb{R}^3, [0, 1]^4).
\]

The four basic CVG operators are called

- union \( \cup (a_1, a_2) \)
- intersection \( \cap (a_1, a_2) \)
- difference \( -(a_1, a_2) \)
- blending \( +(a_1, a_2) \)

and examples of their use are illustrated in Figure 6.8.
Figure 6.8: The seven images illustrate the effect of the four operations on spatial objects to the interior of the objects. They are a cylinder $c$, a sphere $s$, and the effects of the operations $\cup(c, s)$, $\cap(c, s)$, $+(c, s)$, $-(c, s)$ and $-(s, c)$.
Mathematically, each one of these four operators is defined on all spatial objects by constructing its opacity, red, green and blue attribute fields. These attribute fields are in turn defined by pointwise extensions of simple arithmetic operations on the interval $[0, 1]$.

First, here are some six operations on the interval $[0, 1]$ that we will need.

\[
\begin{align*}
\max(s_1, s_2) &= \begin{cases} 
s_1 & \text{if } s_1 \geq s_2; \\
s_2 & \text{if } s_1 < s_2.
\end{cases} \\
\sub(s_1, s_2) &= \max(0, s_1 - s_2) \\
\min(s_1, s_2) &= \begin{cases} 
s_1 & \text{if } s_1 \leq s_2; \\
s_2 & \text{if } s_1 > s_2.
\end{cases} \\
\add(s_1, s_2) &= \min(1, s_1 + s_2) \\
\select(s_1, t_1, s_2, t_2) &= \begin{cases} 
t_1 & \text{if } s_1 \geq s_2; \\
t_2 & \text{if } s_1 < s_2.
\end{cases} \\
\mix(s_1, t_1, s_2, t_2) &= \begin{cases} 
t_1s_1 + t_2s_2 & \frac{s_1 + s_2}{s_1 + s_2} \text{ if } s_1 + s_2 \neq 0; \\
t_1 + t_2 & \frac{s_1 + s_2}{2} \text{ if } s_1 + s_2 = 0.
\end{cases}
\end{align*}
\]

Each of these operations on $[0, 1]$ can be applied all over the space $\mathbb{R}^3$ to create a pointwise extension which we will use as operations on the attribute fields. The first extension of $\max$ to $\text{Max}$ we gave as an example in 6.5.3. The six pointwise extensions are defined as follows:

Let $a_1, a_2, b_1, b_2 : \mathbb{R}^3 \to [0, 1]$. Let $p \in \mathbb{R}^3$

\[
\begin{align*}
\text{Max}(a_1, a_2)(p) &= \max(a_1(p), a_2(p)) \\
\text{Min}(a_1, a_2)(p) &= \min(a_1(p), a_2(p)) \\
\text{Sub}(a_1, a_2)(p) &= \sub(a_1(p), a_2(p)) \\
\text{Add}(a_1, a_2)(p) &= \add(a_1(p), a_2(p)) \\
\text{Select}(a_1, b_1, a_2, b_2)(p) &= \select(a_1(p), b_1(p), a_2(p), b_2(p)) \\
\text{Mix}(a_1, b_1, a_2, b_2)(p) &= \mix(a_1(p), b_1(p), a_2(p), b_2(p))
\end{align*}
\]

The binary operations on the set $O(\mathbb{R}^3, [0, 1]^4)$ of spatial objects are created from these operations on attribute fields as follows:

Let $a_1 = (a_{p_1}, r_1, g_1, b_1)$ and $a_2 = (a_{p_2}, r_2, g_2, b_2)$ be two spatial objects.
Since the spatial objects fill space, this allows the idea to accommodate objects defined mathematically or by digitalisation, whether they have a geometry or are amorphous. The operations transform the data everywhere but opacity plays an important role in deciding how objects are combined.

In Figure 6.8 we have an example of creating a scene by applying the operations.

In Figure 6.9 we have an example of visualising the heart by applying the operations.
Figure 6.9: Operations on heart data visualising fibres.
Figure 6.10: Tree Structure of a CVG Term and the Scene Defined by Term.
6.6 Historical Notes and Further Reading

Volume graphics is an alternate paradigm for computer graphics in which objects are represented by volumes instead of surface representations. An early manifesto for the generality of the volume approach is Kaufman et al. [1993]; an updated vision appears in Chen et al. [2000]. Constructive volume geometry (CVG) is a high level approach to volume graphics based on creating algebras of spatial objects: see Chen and Tucker [2000]. In CVG there are a number of data types of spatial objects and operations that can be used to put together images to form complex scenes. See http://www.swan.ac.uk/compsci/research/graphics/vg/cvg/index.html for more information and downloadable papers. CVG is a generalisation of constructive solid geometry (CSG) of surface graphics. In CSG, solids are described by characteristic functions $s: \mathbb{R}^3 \rightarrow B$ and algebras are created to build solid complex objects from simpler components. That technique is well established in CAD applications.
Exercises for Chapter 6

1. Use the record construction to create algebras that model the following data types:
   a. \( n \)-dimensional real space \( \mathbb{R}^n \); and
   b. \( n \)-dimensional integer grid \( \mathbb{Z}^n \).

2. Let \( \Sigma \) be a signature with \( n \) sorts, \( p \) constants and \( q \) operations. How many sorts, constants and operations does \( \Sigma_{\text{Array}} \) have?

3. Design a signature and an algebra to model the stack data structure. Check that the operations satisfy the equation
   \[
   \text{Pop}_s(\text{Push}_s(x, s)) = s.
   \]
   Is the equation
   \[
   \text{Push}_s(\text{Pop}_s(s), s) = s
   \]
   true?

4. Design a signature \( \Sigma_{2-\text{Array}} \) and a \( \Sigma_{2-\text{Array}} \)-algebra \( A_{2-\text{Array}} \) that model an interface and implementation of 2-dimensional arrays.

5. Design a signature \( \Sigma_{\text{Matrix}[n]} \) and a \( \Sigma_{\text{Matrix}[n]} \) algebra \( A_{\text{Matrix}[n]} \) that model an interface and implementation of \( n \times n \) matrices over the real numbers.

6. Design a signature and an algebra to model each of the following data structures:
   a. lists of data;
   b. queues of data; and
   c. binary trees of data.

7. For each of the following equations over \( \Sigma_{\text{SimpleFiles}} \), state whether they are always true, always false, or sometimes true, in the algebra \( A_{\text{SimpleFiles}} \). For those equations which are not always true, give an example when it is true and when it is false.
   a. \( \text{Read}(i, \text{Empty}) = (e, \text{Empty}) \);
   b. \( \text{Write}(\text{StringContent}(\text{Read}((F, p), n)), \text{FileContent}(\text{Read}((F, p), n))) = (F, p + n) \)
   c. \( \text{StringContent}(\text{Read}((F, p), \text{Succ}(n))) = \text{Prefix}(\text{StringContent}(\text{Read}((F, p)), \text{Succ}(\text{Zero})), \text{StringContent}(\text{Read}((F, \text{Succ}(p)), n))) \).

8. Let \( X \) and \( Y \) be non-empty sets. Create an algebra that models the data type of all total functions
   \[
   f : X \to Y.
   \]
   Compare your algebra with the algebras of arrays and streams.
9. Design a signature $\Sigma$ and an algebra $A$ that models a digital stopwatch that measures time intervals of $\frac{1}{10}$ second, from 0 to 15 minutes. Suppose the stopwatch has stop, continue and reset buttons.

10. Design a signature $\Sigma$ and an algebra $A$ that models a digital alarm clock. Suppose the alarm clock displays hours, minutes and seconds, and rings at any time given by hours and minutes only.

11. Let $\Sigma_{\text{Clock}}$ be the signature of the clock in Section 6.4.2. Define a $\Sigma_{\text{Clock}}$-algebra

$$A_{\text{Continuous Clock}}$$

that models continuous time using the real numbers, rather than discrete time using the natural numbers. To calculate with continuous time, what new functions are useful? What changes are needed in the construction of the $\Sigma_{\text{Stream}}$-algebra $A_{\text{Stream}}$ from the $\Sigma$-algebra $A$ to create an algebra of continuous streams over $A$?

12. Let $A$ be a non-empty set and let $T = \{0, 1, 2, \ldots\}$ be a set of time cycles. Let

$$a = a(0), a(1), a(2), \ldots, a(t), \ldots$$

and

$$b = b(0), b(1), b(2), \ldots, b(t), \ldots$$

be any streams in $[T \rightarrow A]$. Show that the following stream transformations are finitely determined; for each $t$, calculate $l$.

a. Merge : $[T \rightarrow A] \times [T \rightarrow A] \rightarrow [T \rightarrow A]$

$$\text{Merge}(a, b) = a(0), b(0), a(1), b(1), \ldots, a(t), b(t), \ldots$$

b. Insert : $A \times T \times [T \rightarrow A] \rightarrow [T \rightarrow A]$

$$\text{Insert}(x, t, a) = a(0), a(1), a(2), \ldots, a(t - 1), x, a(t), a(t + 1), \ldots$$

c. Forward : $T \times [T \rightarrow A] \rightarrow [T \rightarrow A]$

$$\text{Forward}(t, a) = a(t), a(t + 1), \ldots$$

d. Backward : $T \times A \times [T \rightarrow A] \rightarrow [T \rightarrow A]$

$$\text{Backward}(t, x, a) = x, x, x, \ldots, x, a(0), a(1), \ldots$$

where there are $t$ copies of $x$ before the elements $a(0), a(1), \ldots$ from $a$.

13. Write out the decimal expansions of

a. $\frac{1}{20}$,

b. $\frac{1}{60}$, and

c. $\frac{1}{19}$. 
6.6. HISTORICAL NOTES AND FURTHER READING

14. Prove that the decimal expansion of a rational number $\frac{p}{q}$ where $q \neq 0$ must either be finite and end in zeros,

\[ b_n b_{n-1} \cdots b_1 b_0, a_1 a_2 \cdots a_n 000 \cdots \]
or repeat, and end in a cycle,

\[ b_n b_{n-1} \cdots b_1 b_0, a_1 a_2 \cdots a_n a_{n+1} \cdots a_{n+k} a_{n+1} \cdots a_{n+k} \cdots \]

In the latter case, the cycle $a_{n+1} \cdots a_{n+k}$ has length $k$ and repeats forever.

15. Show that the equality of real numbers in $[0, 1]$ is not definable by a finitely determined stream predicate and cannot be computed.

16. Let $X$ be a space. Let $C$ be any set of coordinates or addresses for $X$. The association of coordinates to points is a mapping

\[ \alpha : C \to X \]

where

\[ \alpha(c) = x \] means $c$ is a coordinate or address for $x$.

Thus, a coordinate system is pair $(C, \alpha)$. What conditions on $\alpha$ are needed to ensure that

a. Every point $x$ has at least one coordinate $c$?

b. Every point $x$ has one and only one coordinate $c$?

c. Given any point its coordinates can be calculated?

17. Let $X$ be a space. Suppose that $C_1$ and $C_2$ are two different coordinate systems for $X$ with functions

\[ \alpha_1 : C_1 \to X \text{ and } \alpha_2 : C_2 \to X \]

associating coordinates to points. We define the systems $(C_1, \alpha_1)$ and $(C_2, \alpha_2)$ to be equivalent if there are two mappings:

\[ f : C_1 \to C_2 \]

that transforms each $C_1$ coordinate of a point to some $C_2$ coordinate and, conversely, a function

\[ g : C_2 \to C_1 \]

that transforms each $C_2$ coordinate of a point to some $C_1$ coordinate. These two transformation conditions are expressed by the equations

for all $c \in C_1 \quad \alpha_1(c) = \alpha_2(f(c))$

for all $c \in C_1 \quad \alpha_2(d) = \alpha_1(g(d))$

which are depicted in the commutative diagrams shown in Figure 6.11:

In what circumstances does

\[ g(f(c)) = c \text{ and } f(g(d)) = d \]

for all $c$ and $d$, i.e., are the transformations $f$ and $g$ inverses to one another?

In the case that $X$ is the 2 dimensional plane, by constructing $f$ and $g$, show that
19.4 CHAPTER 6. EXAMPLES: DATA STRUCTURES, FILES, STREAMS AND SPATIAL OBJECTS

Figure 6.11: Requirement for coordinate systems $C_1$ and $C_2$ to be equivalent.

a. Rectangular coordinate systems based at origin $p$ and $q$ are equivalent.
b. Rectangular coordinate systems based at the same origin $p$ but at different directions are equivalent.
c. Rectangular and polar coordinates at the same origin are equivalent.

18. Consider data for the visualisation of 3-dimensional discrete space. Let the space $X$ be the 3-dimensional grid $Z^3$. Suppose there are 4 attributes of each point $p$ in space representing visibility and colour as follows:

- $o_p$ the opacity of the point $p$ in space;
- $r$ the red value of the point $p$ in space;
- $g$ the green value of the point $p$ in space;
- $b$ the blue value of the point $p$ in space.

The three colour values are reals. The opacity is measured on a scale from 0 transparent to 1 opaque. Then let $A$ be the set $[0, 1] \times R^3$. Here we have an 4-tuple of real numbers as data measuring the appearance at point $x \in X$.

Thus, in this case, a spatial data object

$$o : X \rightarrow A$$

is a map

$$o : Z^3 \rightarrow ([0, 1] \times R^3)$$

defined at each point $(x, y, z) \in Z^3$ in the form:

$$o(i, j, k) = (o_p, r, g, b).$$

Describe a method of extending the discrete space object $o$ to a continuous space object

$$o' : R^3 \rightarrow ([0, 1] \times R^3)$$

coincident with $o$.

19. Suppose we have some transformation

$$t : X \rightarrow X$$
of the space. We can define a new operation

\[ T : O(X, A) \to O(X, A) \]

on spatial data objects by applying the transformation on the underlying space as follows. For any spatial object

\[ o : X \to A \]

the new object

\[ T(o) : X \to A \]

is defined by

\[ T(o)(x) = o(t(x)). \]

Taking \( X \) to be the 2 dimensional plane apply the following transformations to the spatial object: < To be completed. >

20. Show that union \( \cup(o_1, o_2) \) and intersection \( \cap(o_1, o_2) \) operations on spatial objects are commutative:

\[
\begin{align*}
\cup(o_1, o_2) & = \cup(o_2, o_1) \\
\cap(o_1, o_2) & = \cap(o_2, o_1)
\end{align*}
\]

and associative:

\[
\begin{align*}
\cup(o_1, (o_2, o_3)) & = \cup(((o_1, o_2), o_3) \\
\cap(o_1, (o_2, o_3)) & = \cap(((o_1, o_2), o_3).
\end{align*}
\]
Assignment for Chapter 6

Extend the model of the file system given in Section 6.3.3 to incorporate a more complex set of access permissions based on file ownership. Thus, there will be read- and write-access flags for:

- the owner of the file,
- a category of privileged user, and
- other users.

A file is automatically owned by the person who created it; they may grant access permissions to other users. Ownership of a file may only be re-assigned by the privileged user, but any one file must be owned by exactly one user at any point in time. The privileged user may also change permissions of any file.
Chapter 7

Abstract Data Types and Homomorphisms

The mathematical theory of many-sorted algebras serves as a basis for a mathematical theory of data. In Chapters 3 and 4, the notion of a many-sorted algebra was introduced and defined formally, and a number of fundamental examples were presented. These algebras, and the algebras modelling data type constructions in Chapter 6, will be used to define data types in computations in Part III.

In this chapter, we will study the concept of an

abstract data type.

Every data type has an interface and a family of implementations, representations or realisations. The idea of an abstract data type focuses on those properties of a data type that are common to, or are independent of, its implementations. Abstractions are made by fixing an interface and cataloguing properties of the operations. The idea of abstracting from implementation details is natural since data exists independently of its representation in programs. The idea is also practically necessary since representations of specific data types are rarely the same amongst programmers.

We will motivate and explore the general concept largely by investigating the

abstract data type of natural numbers.

The data type of natural numbers is the most simple, beautiful and important in Computer Science. Many ideas about data and computation can be usefully tested on the naturals, as indeed this chapter will show. Indeed, theoretically, we will argue all data types that can be represented on a computer must be representable with natural numbers. In Chapter 8, we will investigate the abstract data type of real numbers.

We can think of a signature $\Sigma$ as an interface and a many-sorted $\Sigma$-algebra $A$ as a mathematical model of a specific and concrete representation, or implementation, of a data type. Thus, we can think of a data type as a class of many-sorted algebras. To explore the idea of an abstract data type, we can investigate properties that are common to the algebras in a class modelling a family of data type implementations. In particular, we think of an abstract data type as a class of many-sorted $\Sigma$-algebras for which the details of the algebras are hidden.

We will consider the question:
When are two representations, or implementations, of a data type with a common interface equivalent?

This becomes the mathematical question:

When are two many-sorted $\Sigma$-algebras equivalent?

We will introduce the mathematical notion of

$\Sigma$-homomorphism

to compare $\Sigma$-algebras. Simply put, a

homomorphism is a data transformation that preserves the operations on data.

From the notion of $\Sigma$-homomorphism, we can derive the notion of

$\Sigma$-isomorphism

which, simply put, is a

data transformation that can be reversed.

These notions are the key to the problem of how to abstract from implementations and to define precisely the notion of an abstract data type.

A second question is

How do we characterise, or specify uniquely, a family of representations or implementations of a data type?

This becomes the mathematical question:

How do we characterise, or specify uniquely, a family of $\Sigma$-algebras?

We will use the axiomatic method, introduced in Chapter 4, to postulate properties of the operations of a data type that its implementations must possess. Specifically, we will answer these questions for the data type of the natural numbers.

We will focus on the simplest operations on natural numbers. We choose a signature containing names for the constant 0 and the successor operation $n + 1$, and present three axioms to govern their application. The most important axiom is

the principle of induction

which is based on the generation of all natural numbers by zero and successor. We prove Dedekind’s Theorem which says, informally, that

All $\Sigma$-algebras satisfying the three axioms are $\Sigma$-isomorphic to one another and, in particular, to the standard $\Sigma$-algebras of natural numbers in base 10 and 2.

The techniques and results are those of Richard Dedekind (1831–1916) who published them in Was Sind und was sollen die Zahlen? in 1888. The little book Dedekind [1888] is inspiring to study. Dedekind’s results were also obtained independently by Giuseppe Peano (1858-1932) to whom they are often attributed.

To explore further the power of the concept of a $\Sigma$-homomorphism, we use it to undertake another basic theoretical investigation. We ask:
What does it mean to implement a data type?

This becomes the mathematical question:

What does it mean to implement a $\Sigma$-algebra?

We will use homomorphisms to answer this question and, using computability theory on the natural numbers, we answer the question:

What data types can be implemented on a computer?

Homomorphisms are vitally important in our theory of data. It is not hard to see why this is so. Whenever we encounter data, we find we have

one interface and many implementations

which means that when modelling data, we have

one signature $\Sigma$ and many $\Sigma$-algebras.

To compare implementations, we must compare $\Sigma$-algebras. This is done using $\Sigma$-homomorphisms.

However, it is not easy to learn and master these mathematical ideas. A rigorous understanding of the data types of natural and real numbers emerged in the late nineteenth century, some 2300 years after Euclid’s great work surveyed the Greeks’ excellent knowledge of geometry and arithmetic.

The mathematical theory of algebras and homomorphisms is abstract and precise. Things that are abstract are hard to learn. Things that are precise are hard to learn. But, having learnt them, they are wonderful tools for analysis and understanding.

In contrast, things that are concrete or imprecise are easier to learn, but poorer tools for analysis and comprehension.

We conclude this chapter with a taste of the mathematical theory of algebras and homomorphisms. Our objective is to understand in much greater detail, the implications of having a $\Sigma$-homomorphism

$$\phi : A \rightarrow B$$

between two $\Sigma$-algebras $A$ and $B$.

We will examine the set $\text{im}(\phi)$ of data in $B$ that is computed by $\phi$. It forms a $\Sigma$-subalgebra of $B$, and is called the image of $\phi$. The “size” of $\text{im}(\phi)$ measures how close to a surjection $\phi$ is.

We will examine the extent to which $\phi$ identifies elements of $A$ using a equivalence relation $\equiv_\phi$ on $A$ called the kernel of $\phi$. Using the kernel, we build a new $\Sigma$-algebra

$$A/ \equiv_\phi$$

which is called the factor or quotient algebra of $A$ by $\equiv_\phi$. The size of $A/ \equiv_\phi$ measures how close to an injection $\phi$ is.

These technical ideas are combined in the main theorem:

**Homomorphism Theorem** $A/ \equiv_\phi$ is isomorphic to $\text{im}(\phi)$.

This fact will be applied here and in later chapters.
7.1 Comparing Decimal and Binary Algebras of Natural Numbers

Let us begin by examining a problem that is easy to state and obviously important. Each of the following algebras may be thought of as distinct implementations of the natural numbers:

- the algebra of natural numbers in base 10 or decimal form;
- the algebra of natural numbers in base 2 or binary form;
- the algebra of natural numbers in base \( b > 0 \);
- the algebra of natural numbers in 1s complement;
- the algebra of natural numbers in 2s complement;
- the algebra of natural numbers in modulo \( n \) arithmetic in a specific representation.

Given that there are many ways of representing the natural numbers, the question arises naturally as to:

When are two different representations of the natural numbers equivalent?

We can formulate this as the question:

When are two algebras of natural numbers equivalent?

Obviously, we will expect most of these algebras of natural numbers with common operations to be equivalent, but in what precise sense are they equivalent?

We also expect that modulo arithmetic over different factors will not be the same, i.e., using the sets

\[
\mathbb{Z}_n = \{0, 1, 2, \ldots, n\} \quad \text{and} \quad \mathbb{Z}_m = \{0, 1, 2, \ldots, m\}
\]

with \( m \neq n \), as a basis for the natural numbers, will not produce the same algebras. So our immediate problem is to provide criteria for deciding how two many-sorted algebras of natural numbers can be compared.

Consider two simple algebras of natural numbers with standard operations given by the signature:

\[
\text{signature} \quad \text{Ordered Naturals}
\]

\[
\text{sorts} \quad \text{nat, bool}
\]

\[
\text{constants} \quad 0 : \to \text{nat}
\]

\[
\text{true, false} : \to \text{bool}
\]

\[
\text{operations} \quad \text{succ} : \text{nat} \to \text{nat}
\]

\[
\text{add} : \text{nat} \times \text{nat} \to \text{nat}
\]

\[
\text{mult} : \text{nat} \times \text{nat} \to \text{nat}
\]

\[
\text{eq} : \text{nat} \times \text{nat} \to \text{bool}
\]

\[
\text{gt} : \text{nat} \times \text{nat} \to \text{bool}
\]
7.1. COMPARING DECIMAL AND BINARY ALGEBRAS OF NATURAL NUMBERS

but with different number representations, namely the numbers in decimal notation

\[ A_{\text{dec}} = (N_{\text{dec}}, B; 0_{\text{dec}}, tt, ff, \text{succ}_{\text{dec}}, \text{add}_{\text{dec}}, \text{mult}_{\text{dec}}, eq_{\text{dec}}, gt_{\text{dec}}), \]

where \( N_{\text{dec}} = \{0, 1, 2, 3, \ldots\} \), and the numbers in binary notation

\[ A_{\text{bin}} = (N_{\text{bin}}, B; 0_{\text{bin}}, tt, ff, \text{succ}_{\text{bin}}, \text{add}_{\text{bin}}, \text{mult}_{\text{bin}}, eq_{\text{bin}}, gt_{\text{bin}}), \]

where \( N_{\text{bin}} = \{0, 1, 10, 11, \ldots\} \).

7.1.1 Data Translation

To compare these two algebras, and indeed to show they are equivalent, we use translation functions:

\[ \alpha : N_{\text{dec}} \to N_{\text{bin}} \]

so that for each \( d \in N_{\text{dec}} \),

\[ \alpha(d) = \text{the binary number corresponding with decimal number } d \]

and, conversely,

\[ \beta : N_{\text{bin}} \to N_{\text{dec}} \]

so that for each \( b \in N_{\text{bin}} \),

\[ \beta(b) = \text{the decimal number corresponding with binary number } b. \]

Since we are aiming to demonstrate equivalence, we expect that these translations are reversible: for all \( d \in N_{\text{dec}} \)

\[ \beta(\alpha(d)) = d \]

and for all \( b \in N_{\text{bin}} \)

\[ \alpha(\beta(b)) = b. \]

As we shall see shortly, these properties of the functions \( \alpha \) and \( \beta \) concerning translation amount to the fact that \( \alpha \) and \( \beta \) are bijections between the sets \( N_{\text{dec}} \) and \( N_{\text{bin}} \).

7.1.2 Operation Correspondence

However, this translation of data, exchanging decimal and binary notations, is not sufficient. Since data is characterized by their operations as well as their representation, we desire that the operators are related appropriately. For example, addition and multiplication of decimal numbers must correspond with the addition and multiplication of binary numbers.

Consider first the translation function \( \alpha \) from decimal to binary. For example, take the successor operator. We expect that for any \( d \in N_{\text{dec}} \),

Successor Equation \[ \alpha(\text{succ}_{\text{dec}}(d)) = \text{succ}_{\text{bin}}(\alpha(d)). \]

This equation can be depicted in the commutative diagram shown in Figure 7.2.

Similarly, for the additive and multiplicative operators, we expect to get for all \( d_1, d_2 \in N_{\text{dec}} \),

\[ \alpha(\text{add}_{\text{dec}}(d_1, d_2)) = \text{add}_{\text{bin}}(\alpha(d_1), \alpha(d_2)) \]

Addition Equation \[ \alpha(\text{add}_{\text{dec}}(d_1, d_2)) = \text{add}_{\text{bin}}(\alpha(d_1), \alpha(d_2)) \]

Multiplication Equation \[ \alpha(\text{mult}_{\text{dec}}(d_1, d_2)) = \text{mult}_{\text{bin}}(\alpha(d_1), \alpha(d_2)). \]
For the relations, we expect that for all $d_1, d_2 \in \textbf{N}_{\text{dec}}$,

- **Equality Equation** \( \text{eq}_{\text{dec}}(d_1, d_2) = \text{eq}_{\text{bin}}(\alpha(d_1), \alpha(d_2)) \)
- **Order Equation** \( \text{gl}_{\text{dec}}(d_1, d_2) = \text{gl}_{\text{bin}}(\alpha(d_1), \alpha(d_2)) \)

The same properties, expressed by the same type of equations, should also hold for the translation function $\beta$ from $\textbf{N}_{\text{bin}}$ to $\textbf{N}_{\text{dec}}$. Thus, we expect to get for all $b_1, b_2 \in \textbf{N}_{\text{bin}},$

- **Successor Equation** \( \beta(\text{succ}_{\text{bin}}(b)) = \text{succ}_{\text{dec}}(\beta(b)) \)
- **Addition Equation** \( \beta(\text{add}_{\text{bin}}(b_1, b_2)) = \text{add}_{\text{dec}}(\beta(b_1), \beta(b_2)) \)
- **Multiplication Equation** \( \beta(\text{mul}_{\text{bin}}(b_1, b_2)) = \text{mul}_{\text{dec}}(\beta(b_1), \beta(b_2)) \)
- **Equality Equation** \( \text{eq}_{\text{bin}}(b_1, b_2) = \text{eq}_{\text{dec}}(\beta(b_1), \beta(b_2)) \)
- **Order Equation** \( \text{gl}_{\text{bin}}(b_1, b_2) = \text{gl}_{\text{dec}}(\beta(b_1), \beta(b_2)) \)

This is illustrated in Figure ??.

These properties of $\alpha$ and $\beta$ concerning operations make $\alpha$ and $\beta$ into so-called homomorphisms. Homomorphisms are translations between algebras that preserve the operations.

Since $\alpha$ and $\beta$ are inverses to one another, they are termed isomorphisms.

These types of criteria are appropriate for comparing any pair of algebras of numbers. We now investigate these mathematical properties in general, for any algebra, only later returning to the problem of comparing $\textbf{N}_{\text{dec}}$ and $\textbf{N}_{\text{bin}}$. 
7.2 Translations between Algebras of Data and Homomorphisms

In the light of the comparison of algebras of natural numbers in Section 7.1, we will now present a series of definitions that allow us to compare any algebras. Specifically, for any signature \( \Sigma \) and any \( \Sigma \)-algebras \( A \) and \( B \), we will define translations from the data of \( A \) to the data of \( B \) that preserve the operations named in \( \Sigma \). These translations will be called \( \Sigma \)-homomorphisms.

Of course, we will sometimes use \( \Sigma \)-homomorphisms with inverses which are also \( \Sigma \)-homomorphisms. These mappings will be called \( \Sigma \)-isomorphisms

and will be treated later, in Section 7.3.

7.2.1 Basic Concept

Here is the general concept of a homomorphism.
Definition (Homomorphism) Let $A$ and $B$ be many-sorted algebras with common signature $\Sigma$. A $\Sigma$-homomorphism $\phi : A \to B$ from $A$ to $B$ is a family

$$\phi = <\phi_s : s \in S>$$

of mappings such that for each sort $s \in S$:

Data Translation

(i) $\phi_s : A_s \to B_s$;

Operation Correspondence

(ii) the map $\phi$ preserves constants, i.e., for each constant $c :\to s$ in $\Sigma$

Constant Equation for $c$

$$\phi_s(c_A) = c_B;$$

(iii) the map $\phi$ preserves operations, i.e., for each operation

$$f : s(1) \times \cdots \times s(n) \to s$$

of $\Sigma$, and any $a_1 \in A_{s(1)}$, $a_2 \in A_{s(2)}$, $\cdots$, $a_n \in A_{s(n)}$ then

Operation Equation for $f$

$$\phi_s(f^A(a_1, a_2, \ldots, a_n)) = f^B(\phi_{s(1)}(a_1), \phi_{s(2)}(a_2), \ldots, \phi_{s(n)}(a_n));$$

This Operation Equation is depicted in the commutative diagram of Figure 7.3.

![Diagram illustrating the Operation Equation](image)

Figure 7.3: Commutative diagram illustrating the Operation Equation.

Commonly, our algebras have tests so we have equipped our algebras with a standard copy of the Booleans $B$. Here, $B$ occurs in both algebras $A$ and $B$, and need not be transformed.

Definition (Homomorphism preserving Booleans) Let $A$ and $B$ be many-sorted algebras with the Booleans and common signature $\Sigma$. A $\Sigma$-homomorphism $\phi : A \to B$ from $A$ to $B$ is a family

$$\phi = <\phi_s : s \in S>$$

of mappings such that for each sort $s \in S$:

Data Translation
7.2. **TRANSLATIONS BETWEEN ALGEBRAS OF DATA AND HOMOMORPHISMS**

(i) \( \phi_s : A_s \rightarrow B_s \);

(ii) for the sort \( \text{bool} \in S \), \( A_{\text{bool}} = B_{\text{bool}} = \{tt, ff\} \) and \( \phi_{\text{bool}} : A_{\text{bool}} \rightarrow B_{\text{bool}} \) is the identity mapping;

**Operation Correspondence**

(iii) the map \( \phi \) preserves constants, i.e., for each constant \( c : \rightarrow s \) in \( \Sigma \)

\[
\text{Constant Equation for } c \\
\phi_s(c_A) = c_B;
\]

(iv) the map \( \phi \) preserves operations, i.e., for each operation \( f : s(1) \times \cdots \times s(n) \rightarrow s \) of \( \Sigma \), and any \( a_1 \in A_s(1), a_2 \in A_s(2), \ldots, a_n \in A_s(n) \) then

\[
\text{Operation Equation for } f \\
\phi_s(f^A(a_1, a_2, \ldots, a_n)) = f^B(\phi_s(a_1), \phi_s(a_2), \ldots, \phi_s(a_n));
\]

and

(v) the map \( \phi \) preserves relations, i.e., for each relation \( r : s(1) \times \cdots \times s(n) \rightarrow \text{bool} \) in \( \Sigma \), and any \( a_1 \in A_s(1), a_2 \in A_s(2), \ldots, a_n \in A_s(n) \) then, recalling condition (ii),

\[
\text{Relation Equation for } r \\
r^A(a_1, a_2, \ldots, a_n) = r^B(\phi_s(a_1), \phi_s(a_2), \ldots, \phi_s(a_n)).
\]

To simplify notation in conditions (iv) and (v), let us define the product type

\[
w = s(1) \times s(2) \times \cdots \times s(n)
\]

and a function

\[
\phi^w : A^w \rightarrow B^w
\]

by

\[
\phi^w(a_1, a_2, \ldots, a_n) = (\phi_s(a_1), \phi_s(a_2), \ldots, \phi_s(a_n))
\]

that simply combines the corresponding component mappings \( \ldots, \phi_s, \ldots \) of \( \phi \). We can rewrite the Operation Equation

\[
\phi_s \circ f^A(a) = f^B \circ \phi^w(a)
\]

for all \( a = (a_1, \ldots, a_n) \in A^w \). This is an equation between functions

\[
\phi \circ f^A = f^B \circ \phi^w;
\]

and is displayed by the simple commutative diagram in Figure 7.4.

Condition (v) is the special case of (iv), taking \( s = \text{Bool} \) and applying property (ii). For relations, the equation is

\[
r^A = r^B \circ \phi^w
\]

and the diagram in Figure 7.5 must commute.
Figure 7.4: Commutative diagram illustrating the preservation of operations under a homomorphism.

\[
\begin{array}{ccc}
A^w & \\ \downarrow \phi^w & \searrow \phi & \downarrow \phi \\
B^w & \rightarrow & A_s \\
\downarrow f_B & \downarrow f_A & \downarrow \\
B_s & \rightarrow & A_s
\end{array}
\]

Figure 7.5: Commutative diagram illustrating the preservation of relations under a Boolean-preserving homomorphism.

\[
\begin{array}{ccc}
A^w & \\ \downarrow \phi^w & \searrow \phi^A & \downarrow \phi^B \\
B^w & \rightarrow & B
\end{array}
\]

7.2.2 Homomorphisms and Binary Operations

To understand the very general conditions (i)-(v) in the definition of a homomorphism, we need lots of illustrations. We will begin with some simple examples involving only a single constant, binary operation and binary relation. Binary operations and relations are operations and relations with two arguments. They occur everywhere and we have seen many in the concepts of group, ring and field.

Let \( \Sigma_{\text{BinSys}} \) be the signature

<table>
<thead>
<tr>
<th>signature</th>
<th>BinarySystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>s, bool</td>
</tr>
<tr>
<td>constants</td>
<td>e \rightarrow s</td>
</tr>
<tr>
<td></td>
<td>tt, ff \rightarrow bool</td>
</tr>
<tr>
<td>operations</td>
<td>_ \circ _ : s \times s \rightarrow s</td>
</tr>
<tr>
<td></td>
<td>_ = _ : s \times s \rightarrow bool</td>
</tr>
<tr>
<td></td>
<td>and : bool \times bool \rightarrow bool</td>
</tr>
<tr>
<td></td>
<td>not : bool \rightarrow bool</td>
</tr>
</tbody>
</table>

Here are some examples of \( \Sigma_{\text{BinSys}} \)-algebras:

- the natural numbers, integers, rational and real numbers equipped with either \( e = 0 \) and
7.2. TRANSLATIONS BETWEEN ALGEBRAS OF DATA AND HOMOMORPHISMS 207

○ = addition, or e = 1 and ○ = multiplication;
• strings over any alphabet equipped with the e = empty string and ○ = concatenation;
• subsets of a set with the e = empty set, and ○ = union, intersection, or difference;

Together with equality and the standard Boolean operations — some 11 types of algebra in all.

Suppose A and B are \( \Sigma_{Bin,Sys} \)-algebras. Applying the definition of a homomorphism that preserves the Booleans from Section 7.2.1, clause by clause, we obtain the following:

A \( \Sigma_{Bin,Sys} \)-homomorphism \( \phi : A \rightarrow B \) is a family

\[
\phi = \langle \phi_A, \phi_{bool} \rangle
\]

of two mappings, such that:

(i) \( \phi_A : A_A \rightarrow B_B \) and \( \phi_{bool} : A_{bool} \rightarrow B_{bool} \);

(ii) \( \phi_{bool} : \{tt, ff\} \rightarrow \{tt, ff\} \) is defined by

\[
\phi_{bool}(tt) = tt \quad \text{and} \quad \phi_{bool}(ff) = ff;
\]

(iii) Constant Equation \( \phi_A(e_A) = e_B \);

(iv) For all \( x, y \in A_A \),

\[
\text{Binary Operation Equation} \quad \phi_A(x \circ_A y) = \phi_A(x) \circ_B \phi_A(y);
\]

(v) For all \( x, y \in A_A \),

\[
\text{Binary Boolean-valued Operation Equation} \quad \phi_A(x =_A y) = (\phi_A(x) =_B \phi_A(y));
\]

or, more familiarly,

\[
x =_A y \quad \text{if, and only if,} \quad \phi_A(x) =_B \phi_A(y).
\]

Note that in the presence of equality as a basic operation, the homomorphism is automatically an injection.

7.2.3 Homomorphisms and Number Systems

We begin with some simple examples of homomorphisms between number systems. The number systems are simply algebras of real numbers or integers equipped with addition or multiplication. The algebras are all examples of Abelian groups (recall Section 5.5).

Let \( \Sigma_{Group} \) be the signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>group</td>
</tr>
<tr>
<td>constants</td>
<td>e : → group</td>
</tr>
<tr>
<td>operations</td>
<td>( \circ, -1 : ) group ( \times ) group ( \rightarrow ) group</td>
</tr>
</tbody>
</table>

\( -1 : \) group \( \rightarrow \) group
The laws for the operations of an Abelian group are as follows:

<table>
<thead>
<tr>
<th>Axioms</th>
<th>Abelian Group</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Associativity</strong></td>
<td>$(\forall x)(\forall y)(\forall z)[x \circ (y \circ z) = (x \circ y) \circ z]$</td>
</tr>
<tr>
<td><strong>Commutativity</strong></td>
<td>$(\forall x)(\forall y)[x \circ y = y \circ x]$</td>
</tr>
<tr>
<td><strong>Identity</strong></td>
<td>$(\forall x)[x \circ e = x]$</td>
</tr>
<tr>
<td><strong>Inverse</strong></td>
<td>$(\forall x)[x \circ x^{-1} = e]$</td>
</tr>
</tbody>
</table>

**Example 1: Real Numbers with Addition**

The addition and subtraction of real numbers is modelled by the $\Sigma_{\text{Group}}$-algebra

$G = (\mathbb{R}; 0; +, -)$

which satisfies the axioms of an Abelian group. We give some examples of $\Sigma_{\text{Group}}$-homomorphisms $\phi : G \to G$.

Let $\lambda \in \mathbb{R}$ be any real number and define a mapping

$\phi_\lambda : \mathbb{R} \to \mathbb{R}$

by

$\phi_\lambda(x) = \lambda x$

For instance, if $\lambda = 2$, then we have defined a doubling operation

$\phi_2(x) = 2x$.

**Lemma** For any $\lambda \in \mathbb{R}$,

$\phi_\lambda : G \to G$

is a $\Sigma_{\text{Group}}$-homomorphism.

**Proof** We must show that $\phi_\lambda$ preserves the constants and operations named in $\Sigma_{\text{Group}}$. We must check that three equations hold.

First, we consider the identity. Now

$\phi_\lambda(0) = \lambda 0$

$= 0$.

Let $x, y \in \mathbb{R}$. Then,

$\phi_\lambda(x + y) = \lambda(x + y)$; \hspace{1cm} (by definition of $\phi_\lambda$)

$= \lambda x + \lambda y$; \hspace{1cm} (by distribution law)

$= \phi_\lambda(x) + \phi_\lambda(y)$. \hspace{1cm} (by definition of $\phi_\lambda$)

Thus, we have shown the validity of
7.2. TRANSLATIONS BETWEEN ALGEBRAS OF DATA AND HOMOMORPHISMS

Group Operation Equation

\[ \phi_\lambda(x + y) = \phi_\lambda(x) + \phi_\lambda(y). \]

Finally, we consider subtraction. For \( x \in \mathbb{R} \),

Inverse Equation

\[
\begin{align*}
\phi_\lambda(-x) &= \lambda \cdot -x \\
&= -(\lambda \cdot x) \\
&= -\phi_\lambda(x).
\end{align*}
\]

These homomorphisms \( \phi_\lambda \) are important, both practically and theoretically. Practically, they represent a linear change of scale and can be generalised to linear transformations of space. Theoretically, they are the only homomorphisms that are also continuous functions on the real numbers.

Recall that, roughly speaking, a function \( f : \mathbb{R} \to \mathbb{R} \) is continuous if its graph can be drawn “without taking the pencil off the paper”; see Figure 7.6 and 7.7. A technical definition is this:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{continuity.png}
\caption{Function \( f \) continuous on \( \mathbb{R} \).}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{discontinuity.png}
\caption{Function \( f \) discontinuous at \( x = \ldots, -1, 0, 1, 2, \ldots \) }
\end{figure}

**Definition (Continuous function)** A function \( f : \mathbb{R} \to \mathbb{R} \) is continuous if, for any margin of error \( \epsilon > 0 \), \( \epsilon \in \mathbb{R} \), on the output of \( f \), there exists a margin of error \( \delta > 0 \), \( \delta \in \mathbb{R} \), on the input, such that for all \( x, y \in \mathbb{R} \),

\[ |x - y| < \delta \quad \text{implies} \quad |f(x) - f(y)| < \epsilon. \]

**Lemma** Let \( \psi : (\mathbb{R}; 0; +, -) \to (\mathbb{R}; 0; +, -) \) be any group homomorphism. If \( \psi \) is continuous, then there exists some \( \lambda \in \mathbb{R} \) such that

\[ \psi(x) = \phi_\lambda(x) = \lambda x \]

for all \( x \in \mathbb{R} \).
The proof is not difficult, but requires some properties of continuous functions on \( \mathbb{R} \) that would be better explained elsewhere. There are uncountably many group homomorphisms from \((\mathbb{R}; 0; +, -)\) to itself, and some have remarkable properties, for instance:

There is a group homomorphism \( \phi : (\mathbb{R}; 0; +, -) \rightarrow (\mathbb{R}; 0; +, -) \) such that, for any open interval

\[
(a, b) = \{ x \in \mathbb{R} : a < x < b \}
\]

and any real number \( y \in \mathbb{R} \), then there exists an \( x \in (a, b) \), such that

\[
\phi(x) = y.
\]

Such a homomorphism is severely discontinuous.

**Example 2: Real Numbers — Comparing Addition and Multiplication**

We compare the operations of addition and subtraction with those of multiplication and division on the real numbers. Let

\[
G_1 = (\mathbb{R}; 0; +, -)
\]

be the group of all real numbers based on addition. Let

\[
G_2 = (\mathbb{R}_+; 1; , -1)
\]

be the group of all strictly positive real numbers, i.e.,

\[
\mathbb{R}_+ = \{ x \in \mathbb{R} \mid x > 0 \},
\]

based on multiplication.

For any real number \( a \), the exponential to base \( a \) is a map

\[
\exp_a : \mathbb{R} \rightarrow \mathbb{R}_+
\]

defined for \( x \in \mathbb{R} \) by

\[
\exp_a(x) = a^x.
\]

Now the familiar index laws for exponentials are:

\[
\begin{align*}
a^0 &= 1 \\
a^{x+y} &= a^x \cdot a^y \\
a^{-x} &= \frac{1}{a^x}
\end{align*}
\]

and they translate into the following equations

- **Identity Equation**
  \[ \exp_a(0) = 1 \]
- **Group Operation Equation**
  \[ \exp_a(x + y) = \exp_a(x) \cdot \exp_a(y) \]
- **Inverse Equation**
  \[ \exp_a(-x) = \exp_a(x)^{-1} \]

These are precisely the equations that state that

\[ \exp_a : G_1 \rightarrow G_2 \] is a \( \Sigma_{\text{Group}} \)-homomorphism.
7.2. TRANSLATIONS BETWEEN ALGEBRAS OF DATA AND HOMOMORPHISMS 211

The logarithm to any base $a$ is a map

$$\log_a : \mathbb{R} \rightarrow \mathbb{R}_+$$

which is also a $\Sigma_{\text{Group}}$-homomorphism $\log_a : G_2 \rightarrow G_1$, because for all $x \in \mathbb{R}_+$,

- **Identity Equation**
  $$\log_a(1) = 0$$

- **Group Operation Equation**
  $$\log_a(x \cdot y) = \log_a(x) + \log_a(y)$$

- **Inverse Equation**
  $$\log_a(x^{-1}) = -\log_a(x)$$

Thus, the ideas that “exponentials turn addition into multiplication” and “logarithms turn multiplication into addition” are made precise by the statements $\exp_a$ and $\log_a$ are $\Sigma_{\text{Group}}$-homomorphisms. Of course, the fact that multiplication can be simulated by addition is an important discovery in the development of practical calculation; we will discuss this shortly, when we return to these examples in Section 7.3.

**Example 3: Integers**

Recall the discussion of the integers $\mathbb{Z}$ and the cyclic integers $\mathbb{Z}_n$ for $n \geq 2$ in Section 5.2, where they were presented as a family of algebras sharing the basic algebraic properties of rings. The point was that $\mathbb{Z}_n$ are finite algebras that look like $\mathbb{Z}$. We will make this point in another way using homomorphisms to model the idea that $\mathbb{Z}$ simulates $\mathbb{Z}_n$ for all $n \geq 2$.

Let

$$G_1 = (\mathbb{Z}; 0; +, -)$$

be the additive group of integers, and let

$$G_2 = (\mathbb{Z}_n; 0; +, -)$$

be the additive group of integers modulo $n \geq 2$.

**Lemma** The map $\phi : \mathbb{Z} \rightarrow \mathbb{Z}_n$ defined for all $x \in \mathbb{Z}$ by

$$\phi(x) = x \mod n$$

is a $\Sigma_{\text{Group}}$-homomorphism.

**Proof** We must show that $\phi$ preserves the constants and operations named in $\Sigma_{\text{Group}}$. The first property is trivial since

- **Identity Equation**
  $$\phi(0) = 0.$$  

We must show that for all $x, y \in \mathbb{Z}$:

- **Addition Equation**
  $$\phi(x + y) = \phi(x) + \phi(y)$$

- **Inverse Equation**
  $$\phi(-x) = -\phi(x)$$

Consider addition. Let

$$x = q_1n + r_1 \quad \text{and} \quad y = q_2n + r_2$$
for $0 \leq r_i < n$. Thus,

$$\phi(x) = r_1 \quad \text{and} \quad \phi(y) = r_2.$$ 

Consider the right-hand-side of the addition equation and add these values in $\mathbb{Z}_n$:

$$\phi(x) + \phi(y) = r_1 + r_2 \mod n.$$ 

Now suppose in $\mathbb{Z}$ that

$$\phi(x) + \phi(y) = r_1 + r_2 = q_3 n + r_3$$

for $0 \leq r_3 < n$. Then

$$\phi(x) + \phi(y) = r_3$$

in $\mathbb{Z}_n$. Now consider the left-hand-side of the addition equation. On adding

$$x + y = (q_1 + q_2)n + r_1 + r_2$$

$$= (q_1 + q_2 + q_3)n + r_3$$

where $0 \leq r_3 < n$. Hence

$$\phi(x + y) = r_3$$

in $\mathbb{Z}_n$. Thus $\phi(x + y) = \phi(x) + \phi(y)$.

Consider inverse. Let $x = qn + r_1$ for $0 \leq r_1 < n$. Then for the right-hand-side of the inverse equation,

$$\phi(x) = r_1 \quad \text{and} \quad -\phi(x) = n - r_1.$$ 

For the right-hand-side of the inverse equation,

$$-x = -qn - r_1$$

$$= -(q + 1)n + n - r_1$$

where $0 \leq n - r_1 < n$. Thus,

$$\phi(-x) = n - r_1$$

in $\mathbb{Z}_n$ and $\phi(-x) = -\phi(x)$. \[ \square \]

### 7.2.4 Homomorphisms and Machines

To suggest, even at this early stage, the breadth of the applications of algebraic homomorphisms, we look at an algebraic formulation of machines and the idea that one machine simulates another.

**Definition** A deterministic state machine $M$ consists of

(i) a non-empty set $I$ of input data;

(ii) a non-empty set $O$ of output data;

(iii) a non-empty set $S$ of states;

(iv) a state transition function $\text{Next} : S \times I \rightarrow S$;
7.3. EQUIVALENCE OF ALGEBRAS OF DATA: ISOMORPHISMS AND ABSTRACT DATA TYPES

(v) an output function

\[ Out : S \times I \rightarrow O. \]

We write

\[ M = (I, O, S, \text{Next, Out}). \]

If each component set \( I, O \) and \( S \) are finite then \( M \) is called a finite state machine or Moore machine.

It is easy to see that, as defined, \( M \) is a three-sorted algebra with two operations!

Let \( \Sigma_{\text{Machine}} \) be a signature for machines such as:

<table>
<thead>
<tr>
<th>signature</th>
<th>( \text{Machine} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>( \text{input, output, state} )</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>( \text{next : state} \times \text{input} \rightarrow \text{state} )</td>
</tr>
<tr>
<td></td>
<td>( \text{out : state} \times \text{input} \rightarrow \text{output} )</td>
</tr>
</tbody>
</table>

Formally, a machine is just a \( \Sigma_{\text{Machine}} \)-algebra. A finite deterministic state machine is simply a finite \( \Sigma_{\text{Machine}} \)-algebra.

Consider two machines

\[ M_1 = (I_1, O_1, S_1; \text{Next}_1, \text{Out}_1) \quad \text{and} \quad M_2 = (I_2, O_2, S_2; \text{Next}_2, \text{Out}_2). \]

**Definition (Machine Simulation)** Machine \( M_1 \) simulates machine \( M_2 \) if there exists a \( \Sigma_{\text{Machine}} \)-homomorphism \( \phi : M_1 \rightarrow M_2 \).

Let us unpack the algebraic definition of homomorphism to see what is involved.

Since the machine algebras are three-sorted, a \( \Sigma_{\text{Machine}} \) homomorphism \( \phi \) consists of three maps:

\[ \phi_{\text{input}} : I_1 \rightarrow I_2 \quad \phi_{\text{output}} : O_1 \rightarrow O_2 \quad \phi_{\text{state}} : S_1 \rightarrow S_2 \]

that preserve each operation of \( \Sigma_{\text{Machine}} \). That is, for any state \( s \in S_1 \) and input \( x \in I_1 \),

- **State Transition Equation** \( \text{Next}_2(\phi_{\text{state}}(s), \phi_{\text{input}}(x)) = \phi_{\text{state}}(\text{Next}_1(s, x)) \)
- **Output Equation** \( \text{Out}_2(\phi_{\text{state}}(s), \phi_{\text{input}}(x)) = \phi_{\text{output}}(\text{Out}_1(s, x)) \).

These equations when expressed as commutative diagrams, are as shown in Figure 7.8.

7.3 Equivalence of Algebras of Data: Isomorphisms and Abstract Data Types

Our immediate objective is to define precisely the equivalence of two algebras of common signature \( \Sigma \). We now generalise our examination of the equivalence of different algebras of natural numbers in Section 7.1, and formalise the idea that
Two \( \Sigma \)-algebras \( A \) and \( B \) are equivalent when there are two data transformations

\[
\alpha : A \rightarrow B \quad \text{and} \quad \beta : B \rightarrow A
\]

that are inverse to one another and preserve the operations on data named in \( \Sigma \).

This results in the concept of \( \Sigma \)-isomorphism which allows us to abstract from data representations and formalise the idea of an abstract data type.

### 7.3.1 Inverses, Surjections, Injections and Bijections

First, let us recall some basic properties of functions and their inverses that, clearly, we will need shortly.

**Definition (Inverses of Maps)** Let \( X \) and \( Y \) be non-empty sets. Let \( f : X \rightarrow Y \) be a map.

1. A map \( g : Y \rightarrow X \) is a **right inverse** function for \( f : X \rightarrow Y \) if for all \( y \in Y \),

\[
(f \circ g)(y) = f(g(y)) = y
\]

or, simply,

\[
f \circ g = id_Y
\]

where \( id_Y : Y \rightarrow Y \) is the identity map. A right inverse \( g \) is also called a **section** of \( f \).

2. A map \( g : Y \rightarrow X \) is a **left inverse** function for \( f : X \rightarrow Y \) if for all \( x \in X \),

\[
(g \circ f)(x) = g(f(x)) = x
\]

or, simply,

\[
g \circ f = id_X
\]

where \( id_X : X \rightarrow X \) is the identity map. A left inverse \( g \) is also called a **retraction** of \( f \).

3. A map \( g : Y \rightarrow X \) is an **inverse** function for \( f : X \rightarrow Y \) if it is both a left and right inverse, i.e.,

\[
f \circ g = id_Y \quad \text{and} \quad g \circ f = id_X
\]

Now these inverse properties, especially (1) and (2), may seem rather abstract, but they correspond nicely with some other, more familiar, basic properties of functions.
7.3. EQUIVALENCE OF ALGEBRAS OF DATA: ISOMORPHISMS AND ABSTRACT DATA TYPES

**Definition (Properties of Functions)** Let $X$ and $Y$ be non-empty sets. We identify three basic properties of a function $f : X \to Y$.

(i) The function $f$ is said to be **surjective**, or **onto**, if for any element $y \in Y$ there exists an element $x \in X$, such that

$$f(x) = y.$$  

Note that many elements of $X$ may be mapped to the same element in $Y$.

(ii) The function $f$ is said to be **injective**, or **one-to-one**, if for any elements $x_1, x_2 \in X$

$$x_1 \neq x_2 \text{ implies } f(x_1) \neq f(x_2)$$

or, equivalently,

$$f(x_1) = f(x_2) \text{ implies } x_1 = x_2.$$  

Note that there may be elements of $Y$ to which no element of $X$ is mapped.

(iii) The function $f$ is said to be **bijective**, or a **one-to-one correspondence**, if $f$ is both injective and surjective.

Here is one half of the connection.

If $f : X \to Y$ is surjective then there exists a map $g : Y \to X$ such that for $y \in Y$,

$$(f \circ g)(y) = y.$$  

The idea is that for $y \in Y$, the value

$$g(y) = \text{any choice } x \in X \text{ such that } f(x) = y.$$  

There are many choices for solutions and, therefore, many maps.

Each $g$ that satisfies the equation above is injective and is called a **right-inverse** or a **section** of $f$.

If $f : X \to Y$ is injective then there exists a map $g : Y \to X$ such that for $x \in X$,

$$(g \circ f)(x) = x.$$  

The idea is that for $y \in Y$,

$$g(y) = \begin{cases} \text{the unique } x \in X \text{ such that } y = f(x) & \text{if } y \in \text{im}(f), \\ \text{any element of } X & \text{if } y \notin \text{im}(f). \end{cases}$$  

There are many solutions and, therefore, many maps. Each $g$ that satisfies the equation above is surjective and is called a **left-inverse** or a **retraction** of $f$.

If $f : X \to Y$ is bijective it is both surjective and injective, so there exists a map $g : Y \to X$ which is both a right- and left-inverse for $f$, i.e., for all $x \in X$, $y \in Y$

$$(f \circ g)(y) = y \text{ and } (g \circ f)(x) = x.$$  

In this case there is only one map with this property; it is called the **inverse** of $f$ and it is written $f^{-1}$.

We can picture the map $f$ and its inverse $f^{-1}$ as a pair of functions with $f : X \to Y$ and $f^{-1} : Y \to X$ as shown in Figure 7.9.

In summary, we have:
**Lemma (Inverses)** Let $X$ and $Y$ be non-empty sets. Let $f : X \to Y$ be a map. Then,

(i) $f$ has a right-inverse if, and only if, $f$ is surjective;

(ii) $f$ has a left-inverse if, and only if, $f$ is injective; and

(iii) $f$ has an inverse if, and only if, $f$ is bijective.

To complete the proof, one has to show the existence of inverses imply surjectivity and injectivity.

### 7.3.2 Isomorphisms

Now we can define the key notion of isomorphism as a “reversible” homomorphism.

**Definition (Isomorphism)** Let $A$ and $B$ be $\Sigma$-algebras. A $\Sigma$-homomorphism $\phi : A \to B$ is a $\Sigma$-isomorphism if there is a $\Sigma$-homomorphism $\psi : B \to A$ that is both a right- and left-inverse for $\phi$. That is, if

$$\phi = \langle \phi_s : A_s \to B_s \mid s \in S \rangle \quad \text{and} \quad \psi = \langle \psi_s : A_s \to B_s \mid s \in S \rangle$$

then, for all $a \in A_s$

$$(\phi_s \circ \psi_s)(a) = a$$

and

$$(\psi_s \circ \phi_s)(b) = b.$$ 

The map $\psi$ is unique and is written $\phi^{-1}$.

**Definition** The algebras $A$ and $B$ are $\Sigma$-isomorphic if there exists a $\Sigma$-isomorphism $\phi : A \to B$, and we write

$$A \cong B.$$ 

Let us spell out the use of isomorphisms:

*Two algebraic structures of common signature $\Sigma$ are considered identical if, and only if, there is a $\Sigma$-isomorphism between them.*

Suppose that $A$ and $B$ are isomorphic by some $\phi$, then:
7.3. EQUIVALENCE OF ALGEBRAS OF DATA: ISOMORPHISMS AND ABSTRACT DATA TYPES

(i) the elements of \( A \) and \( B \) correspond uniquely to one another under \( \phi \) and there is an inverse function \( \phi^{-1} : B \rightarrow A \) such that

\[
\phi^{-1}(\phi(a)) = a \quad \text{and} \quad \phi(\phi^{-1}(b)) = b;
\]

and

(ii) these correspondences \( \phi : A \rightarrow B \) and \( \phi^{-1} : B \rightarrow A \) preserve the operations of both \( A \) and \( B \), since for each operation \( f : w \rightarrow s \) of \( \Sigma \), the diagrams shown in Figure 7.10 commute.

We identify two classes of \( \Sigma \)-homomorphism \( \phi : A \rightarrow B \):

**Definition (Properties of homomorphisms)** Let \( A \) and \( B \) be \( \Sigma \)-algebras.

1. A \( \Sigma \)-homomorphism \( \phi = \langle \phi_s \mid s \in S \rangle \) is a \( \Sigma \)-epimorphism if for each sort \( s \in S \), the function \( \phi_s : A_s \rightarrow B_s \) is surjective or onto; i.e., for every element \( b \in B_s \) there exists an element \( a \in A_s \), such that

\[
\phi_s(a) = b.
\]

2. A \( \Sigma \)-homomorphism \( \phi = \langle \phi_s \mid s \in S \rangle \) is a \( \Sigma \)-monomorphism if for each sort \( s \in S \), the function \( \phi_s : A_s \rightarrow B_s \) is injective or one-to-one; i.e., for all elements \( a_1, a_2 \in A_s \)

\[
a_1 \neq a_2 \quad \text{implies} \quad \phi_s(a_1) \neq \phi_s(a_2)
\]

or, equivalently,

\[
\phi_s(a_1) = \phi_s(a_2) \quad \text{implies} \quad a_1 = a_2.
\]

The following simple fact allows us a short cut to the notion of isomorphism.

**Lemma** Let \( \phi : A \rightarrow B \) be a \( \Sigma \)-homomorphism. If \( \phi \) is a bijection then its unique inverse \( \phi^{-1} : B \rightarrow A \) is also a \( \Sigma \)-homomorphism.

Thanks to this lemma, it is common to see an isomorphism defined to be a bijective homomorphism. Thus:

**Definition (Isomorphism)** A \( \Sigma \)-homomorphism \( \phi = \langle \phi_s \mid s \in S \rangle \) is a \( \Sigma \)-isomorphism if for each sort \( s \in S \), the function \( \phi_s : A_s \rightarrow B_s \) is bijective or a one-to-one correspondence; i.e., \( \phi_s \) is both injective and surjective.
Example: An Equivalence between Addition and Multiplication of Real Numbers

Recall the examples of $\exp_a$ and $\log_a$ from Section 7.2.3. These group homomorphisms are the inverses for one another.

**Lemma** The groups $G_1 = (\mathbb{R}; 0; +, -)$ and $G_2 = (\mathbb{R}_+, 1, \cdot, ^{-1})$ are isomorphic.

This fact, and the concept of homomorphism, is a basis for the traditional use of logarithms and antilogarithms in calculations. To speed multiplication and division of real numbers by transforming to problems using addition and subtraction, tables of values of $\log_{10}$ and $\exp_{10}$ were used to calculate with the formulae:

\[
x \cdot y = \exp_{10}(\log_{10}(x) + \log_{10}(y))
\]
\[
x \div y = \exp_{10}(\log_{10}(x) - \log_{10}(y))
\]

The early discovery of logarithms involved more complex isomorphisms than $\exp_{10}$ and $\log_{10}$, however.

John Napier (1550-1617) published the idea of tables of logarithms in his *Mirifici logarithmorum canonis descriprio* in 1614; the Latin title means *A description of the marvellous rule of logarithms*. The problem Napier solved was to simulate, or implement, multiplication and division by addition and subtraction. In the preface of his later work of 1616 (Napier [1616]) we find:

"Seeing there is nothing (right well beloved students in the Mathematics) that is so troublesome to Mathematicall practise, nor that doth more molest and hinder Calculators, than the Multiplications, Divisions, square and cubical Extractions of great numbers, which beside the tedious expense of time, are for the most part subject to many slippery errors. I began therefore to consider in my minde, by what certaine and ready Art I might remove those hindrances. And having thought upon many things to this purpose, I found at length some excellent briefe rules to be treated of (perhaps) hereafter. But amongst all, none more profitable than this, which together with the hard and tedious Multiplications, Divisions and Extractions of rootes, doth also cast away from the worke itselfe, even the very numbers themselves that are to be multiplied, divided and resolved into rootes, and putteth other numbers in their place, which performe as much as they can do, onely by Addition and Subtraction, Division by two or Division by three ..."

The first table based on the decimal system — where $\log 1 = 0$ and $\log 10 = 1$ — was published by Henry Briggs (1561-1630) in his *Arithmetica Logarithmica* of 1624. See Struik [1969] for extracts of Napier’s works and Fauvel and Gray [1987] for a selection of relevant extracts.

### 7.3.3 Abstract Data Types

Imagine some data type that can be implemented by a number of methods. The data type has in interface $\Sigma$. Each method leads to a $\Sigma$-algebra.

**Definition (Data Type)** A data type consists of:

(i) a signature $\Sigma$; and
(ii) a class $K$ of algebras $\ldots, A, \ldots$ of signature $\Sigma$.

Each algebra $A \in K$ is a model of representation of the data type. We want to think about data and its properties abstractly, that is, independently of representation. We formalise the idea that the two implementations, modelled by $\Sigma$-algebras $A$ and $B$, are equivalent by using the notion of $\Sigma$-isomorphism, of course.

**Definition (Abstraction Principle for Data Types)** Let $P$ be a property of a data type implementation. We say that $P$ is an *abstract property* if $P$ is an invariant under isomorphism, i.e., if $B$ is an algebra, and $A$ and $B$ are isomorphic algebras, then $P$ will also hold of $B$. Thus:

If $P$ is true of $A$ and $A \cong B$ then $P$ is true of $B$.

Simple examples of such properties include

(i) finiteness;

(ii) having constructors;

(iii) any property definable by a set of equations; and

(iv) any property definable by a set of first-order formulae.

In the light of the above discussion of abstract properties of implementations, we can define an abstract data type to be a class of implementations satisfying certain properties.

**Definition (Abstract Data Type)** An *abstract data type* consists of

(i) a signature $\Sigma$; and

(ii) a class $K$ of algebras of signature $\Sigma$ that is closed under isomorphism, i.e.,

$$A \in K \text{ and } B \cong A \text{ implies } B \in K.$$ 

For a few classical abstract data types, like the natural numbers and real numbers, we can create classes with the special property

All algebras in $K$ are isomorphic, i.e.,

$$A, B \in K \text{ implies } A \cong B$$

We will use the natural numbers as our first important example of an abstract data type.

### 7.4 Induction on the Natural Numbers

We begin our examination of the natural numbers by recalling the Principle of Induction. We assume the reader is familiar with the Principle and some of its applications, so our remarks will be brief. The Principle is based on an algebraic property of numbers.

Consider the signature
signature  Naturals
sorts    nat
constants  zero : → nat
operations  succ : nat → nat

which we can implement with the algebra

algebra  Naturals
 carriers  N
 constants  Zero : → N
 operations  Succ : N → N
 definitions

\[ \begin{align*}
  \text{Zero} &= 0 \\
  \text{Succ}(n) &= n + 1
\end{align*} \]

As we noticed earlier (in Chapter 3), all the natural numbers are generated from the constant zero by the repeated application of the operation succ. Thus,

zero, succ(zero), succ(succ(zero)), succ(succ(succ(zero))), ...

are terms denoting every natural number, whose values are

Zero, Succ(Zero), Succ(Succ(Zero)), Succ(Succ(Succ(Zero))), ...

or

\[ 0, 1, 1+1, 1+1+1, \ldots \]

The fact that all data can be built up from the constants by applying the operations is an important property of an algebra; such algebras are said to be minimal. Notice another special property of the way Succ generates the numbers: each number has one, and only one, construction using Succ.

A consequence of these properties of this algebra of natural numbers is a method of reasoning called the Principle of Induction.

Let \( P \) be a property of natural numbers. Sometimes this means \( P \) is a set of numbers,

\[ P \subseteq \mathbb{N} \]

and we write \( P(n) \) to mean \( n \in P \).

Alternatively, sometimes this means \( P \) is a predicate or Boolean-valued function,

\[ P : \mathbb{N} \rightarrow \mathbb{B} \]
and we say $P$ holds at $n$ if

$$P(n) = \text{tt}$$

or, simply, $P(n)$ is true. The Boolean-valued function $P$ determines the set

$$S = \{ n \in \mathbb{N} : P(n) = \text{tt} \}$$

of elements for which $P : \mathbb{N} \to \mathbb{B}$ holds.

### 7.4.1 Induction for Sets and Predicates

Suppose we want to prove that

*all natural numbers are in a set $S$*

or, equivalently,

*all natural numbers have a property $P$.*

The Principle of Induction is based upon two statements. Let $S \subseteq \mathbb{N}$. Suppose that:

**Base Case**  \quad  $0 \in S$.

**Induction Step** If $n \in S$ then $\text{succ}(n) \in S$.

Then we deduce that:

- $0 \in S$ because of the Base Case
- $1 \in S$ because $0 \in S$ and the Induction Step
- $2 \in S$ because $1 \in S$ and the Induction Step
- $\vdots$  \quad $\vdots$  \quad $\vdots$
- $n + 1 \in S$ because $n \in S$ and the Induction Step
- $\vdots$  \quad $\vdots$  \quad $\vdots$

We want to conclude that all the natural numbers are in the set, i.e.,

$$0, 1, 2, \ldots \in S \quad \text{or} \quad S = \mathbb{N},$$

because the natural numbers are precisely those made from 0 by applying the operator $\text{succ}$.

To conclude from the argument we simply *assume* the following general principle or axiom is true.

**Principle of Induction on Natural Numbers: Sets**

*Let $S$ be any set of natural numbers. If the following two statements hold:

**Base Case**  \quad  $0 \in S$.

**Induction Step** If $n \in S$ then $n + 1 \in S$.

Then $n \in S$ for all $n \in \mathbb{N}$, i.e. $S = \mathbb{N}$.*
The Principle of Induction for the natural numbers is commonly stated in terms of the properties, rather than sets, as follows:

**Principle of Induction on Natural Numbers: Properties**

Let $P$ be any property of the natural numbers. If the following two statements hold:

**Base Case**  
$P(0)$ is true.

**Induction Step**  
If $P(n)$ is true then $P(n+1)$ is true.

Then $P(n)$ is true for all $n$.

Alternatively, if $P$ is the name of the property and $n$ a variable, the Principle is concisely stated in logical formulae,

$$[P(0) \land \forall n(P(n) \rightarrow P(n + 1))] \rightarrow \forall nP(n).$$

The Principle can be weakened by placing restrictions on the classes of sets or predicates that can appear in the statement.

### 7.4.2 Course of Values Induction and Other Principles

A variation is this course-of-values induction.

**Principle of Course of Values Induction on Natural Numbers**

Let $P$ be any property of the natural numbers. If the following two statements hold:

**Base Case**  
$P(0)$ is true.

**Induction Step**  
If $P(i)$ is true for all $i < n$ then $P(n+1)$ is true.

Then $P(n)$ is true for all $n$.

Alternatively, if $P$ is the name of the property and $n$ a variable, the Course of Values Induction Principle is concisely stated in logical formulae,

$$[P(0) \land \forall n \forall i < n(P(i) \rightarrow P(n))] \rightarrow \forall nP(n).$$

### 7.4.3 Defining Functions by Primitive Recursion

The inductive generation of the natural numbers provides an inductive method of defining functions. A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is defined on all numbers $n \in \mathbb{N}$ by defining $f(0)$ and giving a method that computes the value of $f(n+1)$ from the value of $f(n)$. Here is a definition of one mechanism.
7.4. **INDUCTION ON THE NATURAL NUMBERS**

**Definition (Primitive Recursion)** Let \( g : \mathbb{N}^k \to \mathbb{N} \) and \( h : \mathbb{N} \times \mathbb{N}^k \times \mathbb{N} \to \mathbb{N} \) be total functions. Then the function

\[
f : \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}
\]

is defined by *primitive recursion* from the functions \( g \) and \( h \) if it satisfies the equations

\[
f(0, x) = g(x)
\]

\[
f(n + 1, x) = h(n, x, f(n, x))
\]

for all \( n \in \mathbb{N} \) and \( x \in \mathbb{N}^k \).

The equations allow us to calculate the values

\[
f(0, x), f(1, x), f(2, x), \ldots
\]

of the function \( f \) on any \( x \) by substitution as follows:

\[
f(0, x) = g(x)
\]

\[
f(1, x) = h(0, x, f(0, x)) = h(0, x, g(x))
\]

\[
f(2, x) = h(1, x, f(1, x)) = h(1, x, h(0, x, g(x)))
\]

\[
f(3, x) = h(2, x, f(2, x)) = h(2, x, h(1, x, h(0, x, g(x))))
\]

\[
\vdots
\]

It seems clear that there is only one function \( f \) that satisfies the equations and that we have an operational method for computing it on any \( n \in \mathbb{N} \).

**Lemma (Uniqueness)** If \( g \) and \( h \) are total functions then there is a unique total function \( f \) that is defined by primitive recursion from \( g \) and \( h \).

**Proof** We can prove uniqueness. Suppose that \( f_1 \) and \( f_2 \) are total functions that satisfy the primitive recursion equations, i.e.,

\[
f_1(0, x) = g(x)
\]

\[
f_1(n + 1, x) = h(n, x, f_1(n, x))
\]

and

\[
f_2(0, x) = g(x)
\]

\[
f_2(n + 1, x) = h(n, x, f_2(n, x))
\]

We will prove that for all \( n \) and \( x \) that \( f_1(n, x) = f_2(n, x) \), using the Principle of Induction on \( \mathbb{N} \).
CHAPTER 7. ABSTRACT DATA TYPES AND HOMOMORPHISMS

**Base Case** Consider the case $n = 0$:

\[
\begin{align*}
    f_1(0, x) & = g(x) \\
    & = f_2(0, x)
\end{align*}
\]

Therefore, the functions are equal at 0.

**Induction Step** As induction hypothesis, suppose that

\[
f_1(n, x) = f_2(n, x) \quad \text{(IH)}
\]

and consider the values of $f_1(n + 1, x)$ and $f_2(n + 1, x)$. Now

\[
\begin{align*}
    f_1(n + 1, x) & = h(n, x, f_1(n, x)) \\
    & = h(n, x, f_2(n, x)) \quad \text{(by the induction hypothesis (IH))} \\
    & = f_2(n + 1, x) \quad \text{(by the definition of $f_2$)}
\end{align*}
\]

Therefore, if the functions are equal at $n$, we have shown that they are also equal at $n + 1$.

By the Principle of Induction, we conclude that

\[
f_1(n, x) = f_2(n, x)
\]

for all $n$ and $x$. \qed

**Examples**

Consider the basic functions of arithmetic.

Addition is primitive recursive over successor. It is easy to see that $add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ satisfies

\[
\begin{align*}
    add(0, m) & = m \\
    add(n + 1, m) & = \text{succ}(add(n, m)).
\end{align*}
\]

Multiplication is primitive recursive over addition. The function $mult : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ satisfies

\[
\begin{align*}
    mult(0, m) & = m \\
    mult(n + 1, m) & = add(m, mult(n, m)).
\end{align*}
\]

Other functions appear in the exercises.

### 7.5 Naturals as an Abstract Data Type

Let us reconsider the algebras of natural numbers in Section 7.1. First, we focus on the simple operation of counting named in the following signature $\Sigma_{\text{Naturals}}$. 

signature  Naturals
sorts       nat
constants  zero : → nat
operations succ : nat → nat

As we noted earlier, there is a wide range of algebras that have this signature and that
perform a type of counting. The algebras can be infinite or finite, and possess unexpected
properties. In fact, the class Alg(Σ_{Naturals}) is very diverse.

We will classify those Σ_{Naturals}-algebras that are isomorphic to the standard algebra for
counting
\[ N_{dec} = \{0, 1, 2, \ldots\}; 0, n + 1 \].

Hence we will show that all the infinite algebras of natural numbers, represented using different
bases, are isomorphic. To classify the standard algebras we need just three axioms concerning
the properties of the constant zero and operation succ in Σ_{Naturals}:

axioms  Dedekind

\[(\forall x)[\text{succ}(x) \neq \text{zero}]\]
\[(\forall x)(\forall y)[\text{succ}(x) = \text{succ}(y) \Rightarrow x = y]\]

Induction  \((\forall X)[\text{zero} \in X \text{ and } (\forall x)[x \in X \Rightarrow \text{succ}(x) \in X] \Rightarrow (\forall x)[x \in X]]\]

Theorem (Dedekind)

1. The Σ_{Naturals}-algebras N_{dec} and N_{bin} satisfy Dedekind’s Axioms.

2. Let A and B be any Σ_{Naturals}-algebras satisfying Dedekind’s Axioms. Then
\[ A \cong B. \]

Corollary  All algebras satisfying Dedekind’s Axioms are isomorphic with N_{dec}.

First we explore the axioms and then we prove the Theorem. The signature Σ_{Naturals} and
Induction Axioms 1 and 2 express, in a very abstract way, the essence of a counting system:
there is a first element zero and an operation that is injective, returning a next element succ(x)
that is unique to x (Axiom ....) and does not return the first element (Axiom .......).
Here are some \( \Sigma \)-algebras of integers (in decimal notation) that satisfy the axioms.

\[
\begin{align*}
(\{0, 1, 2, \ldots\}; 0, n + 1) \\
(\{1, 2, 3, \ldots\}; 1, n + 1) \\
(\{11, 12, 13, \ldots\}; 11, n + 1) \\
(\{-1, 0, 1, 2, \ldots\}; -1, n + 1) \\
(\{-19, -18, \ldots, 0, 1, 2, \ldots\}; -19, n + 1)
\end{align*}
\]

In particular, let \( k \) be any integer, positive or negative,

\[
(\{k, k + 1, k + 2, \ldots\}; k, n + 1)
\]

satisfies the axioms.

**Lemma (Minimality of Induction Axioms)** Let \( A = (N; a; \text{succ}_A) \) be any \( \Sigma_{\text{Natural}} \)-algebra satisfying Dedekind’s Axioms. Then the carrier of \( A \) is the set

\[
N = \{a, \text{succ}_A(a), \text{succ}_A^2(a), \ldots, \text{succ}_A^n(a), \ldots\}.
\]

In particular, the carrier can be enumerated, without repetitions, by applying the successor function to the first element, and is an infinite set.

**Proof** Let \( E = \{a, \text{succ}_A(a), \text{succ}_A^2(a), \ldots, \text{succ}_A^n(a), \ldots\} \) be the set of elements of \( N \) enumerated by \( \text{succ}_A \). Clearly,

\[
E \subseteq N.
\]

That \( N \subseteq E \) follows from the Induction Axiom. To see that every element of \( N \) is in the enumeration we note that:

- \( a \in N \); and
- if \( \text{succ}_A^i(a) \in N \) then \( \text{succ}_A^{i+1}(a) \in N \).

By Induction Axiom 3, \( A = N \).

To see that there are no repetitions, suppose \( i \geq j \) and the \( i^{th} \) and \( j^{th} \) elements are the same:

\[
\text{succ}_A^i(a) = \text{succ}_A^j(a).
\]

Then, by applying Axiom 2, \( i - j \) times to \( \text{succ}_A \), we get

\[
\text{succ}_A^{i-j}(a) = a.
\]

By Axiom 1, this holds only when \( i - j = 0 \) and \( i = j \). Thus, all elements in the enumeration \( N \) are distinct. In particular, \( A \) is infinite. \( \square \)

We will now use this lemma to prove Dedekind’s Theorem.

The first statements that \( \mathbb{N}_{\text{dec}} \) and \( \mathbb{N}_{\text{bin}} \) satisfy the axioms, we will not prove. It is tempting to say that the statement is obvious, since the axioms are taken from the standard arithmetical structures. However, that these algebras satisfy the axioms is actually a formal property that can be verified using a precise representation of the numbers.
7.5. NATURALS AS AN ABSTRACT DATA TYPE

Proof of Dedekind’s Theorem

Let

\[ A = (N; a; \text{succ}_A) \quad \text{and} \quad B = (M; b; \text{succ}_B) \]

be any two \( \Sigma_{\text{Natural}} \)-algebra satisfying Dedekind’s Axioms.

By the Minimality of Induction Axiom Lemma, we know the carriers of \( A \) and \( B \) can be enumerated thus:

\[ N = \{ a, \text{succ}_A(a), \text{succ}_A^2(a), \ldots, \text{succ}_A^n(a), \ldots \} \]

and

\[ M = \{ b, \text{succ}_B(b), \text{succ}_B^2(b), \ldots, \text{succ}_B^n(b), \ldots \}. \]

We define a map

\[ \phi : N \rightarrow M \]

by

\[ \phi(a) = b \]

and for \( n > 0, \)

\[ \phi(\text{succ}_A^n(a)) = \text{succ}_B^n(b). \]

We will show that the map is a \( \Sigma_{\text{Natural}} \)-isomorphism \( \phi : A \rightarrow B. \)

Now it is easy to check that \( \phi \) is a bijection. The map \( \phi \) is clearly surjective because each element \( \text{succ}_B^n(b) \) of \( M \) is the image of an element \( \text{succ}_A^n(a) \) of \( N \). To see that it is injective, we suppose

\[ \phi(\text{succ}_A^i(a)) = \phi(\text{succ}_A^j(a)). \]

Thus, by definition of \( \phi, \)

\[ \text{succ}_A^i(b) = \text{succ}_A^j(b). \]

This is the case if, and only if, \( i = j. \)

Let us show it preserves operations and is a \( \Sigma \)-homomorphism: clearly it preserves constants, since by definition,

\[ \phi(a) = b. \]

Next, we show that it also preserves successor, i.e., for any \( x \in N, \)

\[ \phi(\text{succ}_A(x)) = \text{succ}_B(\phi(x)). \]

Now for any \( x \in N \) there is one, and only one, \( n \) such that \( x = \text{succ}_A^n(a) \). Hence

\[ \phi(\text{succ}_A(x)) = \phi(\text{succ}_A(\text{succ}_A^n(a))) = \phi(\text{succ}_A^{n+1}(a)) = \text{succ}_B^{n+1}(b) = \text{succ}_B(\text{succ}_B^n(b)) = \text{succ}_B(\phi(\text{succ}_A^n(a))) = \text{succ}_B(\phi(x)). \]
Definition (Abstract Data Type of Natural Numbers) We may define the abstract data type of the natural numbers to be the class

$$\text{Alg}(\Sigma_{\text{Natural}_{\text{Natural}}}, E_{\text{Dedekind}})$$

of all $\Sigma_{\text{Natural}_{\text{Natural}}}$-algebras satisfying Dedekind’s Axioms (see Figure 7.11).

![Diagram](image)

Figure 7.11: The class of all $\Sigma_{\text{Natural}_{\text{Natural}}}$-algebras.

### 7.6 Computable Data Types

In practice, abstract ideas about data are essential in the analysis, development and documentation of systems. However, ultimately, they are reduced to low-level executable representations of data made from machine words. Thus, the data types that are implementable are modelled by algebras of data that also possess machine word representations.

We will use our algebraic theory of data, and especially homomorphisms, to answer the following question

**What abstract data types can be implemented on a digital computer?**

Now, to determine which abstract data types can be implemented on a computer, we must abstract away from the seemingly infinite variety of machine architectures. We shall see more of this idea later in Chapter 16, but here we will focus on the data types that machines process.

Today, we think of digital machines as processing (electronically) data types consisting of

- bits and $n$-bit words,

for different $n$, such as the convenient byte ($n = 8$) and common $n = 32$ bit word. The operations on bits and $n$-bit words are many and varied, depending as they do on the many and varied designs for machine architectures. In practice, however, these data types implement mainly arithmetic processes that represent data and govern the operation of machines. In history, digital computation, by hand and by machine, has not been based on bits and $n$-bit words, but on many different forms of arithmetic notations. Indeed, at a slightly higher level of abstraction, a more convenient data type to describe digital computation is the natural numbers.

For a theoretical investigation, we will ignore bounds on the size of memory, words, etc. Instead, we might devote some thought to the question
What is a digital computation?

First, we will make the slight data abstraction:

We suppose that the basic data type of a digital machine is the natural numbers. Data is coded using natural numbers.

This simplification allows us to avoid worrying about the parameter of word size, for example. We will investigate the following mathematically precise version of the implementation question for data types:

What algebras can be represented by sets of natural numbers and functions on the natural numbers?

7.6.1 Representing an Algebra using Natural Numbers

Let Σ be any signature. Suppose, for simplicity, that Σ is single-sorted and has the form

\((s; c_1, \ldots, c_r; f_1, \ldots, f_q)\).

Let the data type implementation be modelled by a Σ-algebra \(A\) which has the form:

\[ A = (A; c^A_1, \ldots, c^A_r; f^A_1, \ldots, f^A_q) \]

where the \(c^A_i\)'s are constants and \(f^A_j\)'s are operations. This algebra models some representation of a data type with interface Σ. If the data is digital, then we assume it can be represented using natural numbers.

To represent or code the data in this algebra \(A\) using natural numbers, we need the following:

1. A map \(\alpha : \mathbb{N} \to A\) to code all the data in \(A\); if \(a \in A\) and \(n \in \mathbb{N}\) and \(\alpha(n) = a\), we say that \(n\) is an \(\alpha\)-code for \(a\). Every \(a \in A\) must have at least one code \(n \in \mathbb{N}\).

2. The relation \(\equiv_\alpha\) which is used to detect duplication of codes for data is defined for \(m, n \in \mathbb{N}\) by

\[ m \equiv_\alpha n \text{ if, and only if, } \alpha(m) = \alpha(n). \]

This relation is an equivalence relation.

**Definition (Numbering)** A numbering of \(A\) is a surjective mapping

\[ \alpha : \mathbb{N} \to A \]

The equivalence relation \(\equiv_\alpha\) is called the kernel of the numbering.

In theory, a numbering \(\alpha\) is a digital representation of the data in \(A\).

Having coded the data by numbers, we must be able to simulate the operations on data by operations on their codes.

To represent the constants and operations of the algebra \(A\) on numbers, we need the following:
3. For each constant $c_i$ in $\Sigma$ naming $c_i^A \in A$, we need to choose a number $c_i^N \in N$ such that

$$\alpha(c_i^N) = c_i^A$$

for $1 \leq i \leq r$.

4. For each basic operation $f_i$ in $\Sigma$ naming a function $f_i^A : A^{n_i} \to A$, we need a function

$$f_i^N : N^{n_i} \to N$$

called a tracking function, that simulates the operations of $A$ in the number codes.

This idea of simulation we make precise in the equation

$$f_i^A(\alpha(x_1), \ldots, \alpha(x_{n_i})) = \alpha(f_i^N(x_1, \ldots, x_{n_i}))$$

for $1 \leq i \leq q$, such that the diagram shown in Figure 7.12 commutes.

![Diagram](image)

Figure 7.12: Tracking functions simulating the operations of $A$.

This analysis leads to the following ideas.

**Definition (Numbered Algebra)** An algebra is said to be numbered if

(i) there exists a surjective coding map $\alpha$

(ii) for each constant $c_i \in \Sigma$, a number $c_i^N \in N$, and

(iii) for each function $f_i$ in $\Sigma$, tracking functions $f_i^N$.

**Hypothesis** If a data type represented by an algebra $A$ is implementable on a digital computer, then $A$ is numbered.

### 7.6.2 Algebraic Definition

Let us look more closely at the details of this idea of digital representation. The components of a numbered algebra are:

(i) a coding map; and

(ii) some tracking machinery.

These have a natural and familiar structure.

In our definition of a numbered algebra above, we have the following components:
7.6. COMPUTABLE DATA TYPES

- The codes for constants and tracking functions can be combined together to form the algebra

\[ \Omega = (N; c_1^N, \ldots, c_r^N, f_1^N, \ldots, f_q^N) . \]

This is an algebra with the same signature \( \Sigma \) as \( A \). In particular, this is an algebra of numbers with number-theoretic operations.

- The conditions on \( \alpha \) as a coding map for data are equivalent to the statements that:

  (i) \( \alpha \) is surjective;
  (ii) the function

  \[ \alpha : \Omega \to A \]

  is a \( \Sigma \)-homomorphism; and
  \( (iii) \equiv_\alpha \) is the kernel of \( \alpha \).

Thus, the notion of homomorphism is exactly the idea we need to clarify and make precise the idea of digital representation!

7.6.3 Computable Algebras

So far, the analysis of the implementation question has resulted in the observation:

If a data type can be implemented on a digital computer then it is modelled by a \( \Sigma \)-algebra \( A \) that possesses a surjective \( \Sigma \)-homomorphism

\[ \alpha : \Omega \to A \]

from a \( \Sigma \)-algebra \( \Omega \) of natural numbers.

Having represented the \( \Sigma \)-algebra \( A \) of data by a \( \Sigma \)-algebra \( \Omega \) of numbers, the question arises:

What \( \Sigma \)-algebras of numbers can be implemented on a computer?

More specifically, for a representation \( \alpha : \Omega \to A \), we need to know that the code set, kernel, constants and tracking functions are actually computable on natural numbers.

For functions, relations and sets to be computable, there must exist algorithms to compute them. The code set is \( N \), and the constants are just numbers, so they are computable. Thus, to perform digital computation, we need to postulate algorithms to

(i) compute the tracking functions \( f^N \), and

(ii) decide the kernel \( \equiv_\alpha \).

Computable functions and sets on \( N \) are the subject matter of Computability Theory. This was started in the 1930’s by A Church, A M Turing, E Post and S C Kleene. One of its many early achievements was to capture the concept of a

function definable by an algorithm on \( N \).
Many different ways to design algorithms and write specifications and programs have been invented and analysed. These mathematical models of computation have defined the same class of functions, often called the

*partial recursive functions on* \( \mathbb{N} \).

Such philosophical investigations and mathematical results are the basis of the

*Church-Turing Thesis* (1936) The set of functions on \( \mathbb{N} \) definable by algorithms

is the set of partial recursive functions on \( \mathbb{N} \).

**Definition** A \( \Sigma \)-algebra \( A \) is *computable* if, and only if, there is a \( \Sigma \)-algebra of numbers \( \Omega \) with partial recursive operations on \( \mathbb{N} \) and a \( \Sigma \)-epimorphism \( \alpha : \Omega \rightarrow A \) such that the kernel \( \Xi_\alpha \) is recursively decidable.

Assuming this thesis, one simple answer to the implementation question is:

**Hypothesis** *The abstract data types implementable on a digital computer are those possessing an implementation modelled by a computable \( \Sigma \)-algebra.*

### 7.7 Properties of Homomorphisms

We have spent quite some time on the idea of a homomorphism, motivating it, illustrating it, defining it. We have used it in Dedekind’s Theorem, and in an analysis of data type representation and digital computation. It ought to be clear that homomorphisms are an extremely important technical notion, with interesting and useful applications. We conclude this chapter with a taste of the general theory that rounds off our picture of homomorphisms, and has some applications later on. Our goal is to prove the Homomorphism Theorem in Section 7.9.

If \( A \) and \( B \) are \( \Sigma \)-algebras and \( \phi : A \rightarrow B \) is a \( \Sigma \)-homomorphism, then we know that the data in \( A \) corresponds with at least some of the data in \( B \). The set of data in \( B \) is called the image of \( \phi \), as shown in Figure 7.13.

![Diagram](image)

**Figure 7.13:** The image \( im(\phi) \) of a homomorphism \( \phi \).

**Definition** If \( A \) and \( B \) are \( S \)-sorted \( \Sigma \)-algebras and \( \phi = < \phi_s : A_s \rightarrow B_s \mid s \in S > \) is a \( \Sigma \)-homomorphism from \( A \) to \( B \), we define the *image* of \( A \) under \( \phi \) by

\[
im(\phi) = < \im_s(\phi) \mid s \in S >
\]
where
\[ \text{im}_s(\phi) = \{ b \in B_s \mid b = \phi_s(a) \text{ for some } a \in A_s \}. \]

We may also use the notation
\[ \phi(A) = <\phi_s(A_s) \mid s \in S > \]
for the image of \( A \) under \( \phi \).

**Lemma** Let \( A \) and \( B \) be \( S \)-sorted \( \Sigma \)-algebras and \( \phi : A \to B \) a \( \Sigma \)-homomorphism. Then \( \text{im}(\phi) \) is a \( \Sigma \)-subalgebra of \( B \).

**Proof** Clearly, \( \text{im}(\phi) \) contains the constants of \( B \), since for each constant symbol \( c : \to s \in \Sigma \),
\[ c^B = \phi_s(c^A) \]
since \( \phi \) is a homomorphism.

We must show that \( \text{im}(\phi) \) is closed under the operations of \( B \). Let \( f : s(1) \times \cdots \times s(n) \to s \in \Sigma \) be any function symbol. For any elements \( b_1 \in \text{im}(\phi_{s(1)}), \ldots, b_n \in \text{im}(\phi_{s(n)}) \) of the image of \( A \) under \( \phi \), there exist elements \( a_1 \in A_{s(1)}, \ldots, a_n \in A_{s(n)} \), such that
\[ \phi_{s(1)}(a_1) = b_1, \ldots, \phi_{s(n)}(a_n) = b_n. \]
Thus, on substituting,
\[ f^B(b_1, \ldots, b_n) = f^B(\phi_{s(1)}(a_1), \ldots, \phi_{s(n)}(a_n)) \]
\[ = \phi_s(f^A(a_1, \ldots, a_n)) \]
since \( \phi \) is a \( \Sigma \)-homomorphism. Thus, \( f^B(b_1, \ldots, b_n) \in \text{im}(\phi_s) \) because \( f^A(a_1, \ldots, a_n) \in A \). \( \square \)

Another simple property of homomorphisms is that we can compose them.

**Lemma (Composing Homomorphisms)** If \( A, B \) and \( C \) are \( S \)-sorted \( \Sigma \)-algebras and \( \phi : A \to B \) and \( \psi : B \to C \) are \( \Sigma \)-homomorphisms, then the composition \( \psi \circ \phi : A \to C \) is a \( \Sigma \)-homomorphism.

**Proof** For each constant symbol \( c : \to s \in \Sigma \),
\[ c^C = \psi(c^B) \quad \text{and} \quad c^B = \psi(c^A) \]
as \( \phi \) and \( \psi \) are homomorphisms; substituting for \( c^B \), we have
\[ \psi(\phi(c^A)) = \psi \circ \phi(c^A). \]

For each function symbol \( f : s(1) \times \cdots \times s(n) \to s \in \Sigma \), and any elements \( a_1 \in A_{s(1)}, \ldots, a_n \in A_{s(n)} \),
\[ \psi \circ \phi(f^A(a_1, \ldots, a_n)) = \psi(\phi(f^A(a_1, \ldots, a_n))) \quad \text{(by definition)} \]
\[ = \psi(\phi_{s(1)}(a_1), \ldots, \phi_{s(n)}(a_n)) \quad \text{(since \( \phi \) is a \( \Sigma \)-homomorphism)} \]
\[ = f^C(\psi_{s(1)}(\phi_{s(1)}(a_1)), \ldots, \psi_{s(n)}(\phi_{s(n)}(a_n))) \quad \text{(since \( \psi \) is a \( \Sigma \)-homomorphism)} \]
\[ = f^C(\psi_{s(1)} \circ \phi_{s(1)}(a_1), \ldots, \psi_{s(n)} \circ \phi_{s(n)}(a_n)) \quad \text{(by definition)} \]
Therefore, \( \psi \circ \phi \) is a homomorphism. \( \square \)
7.8 Congruences and Quotient Algebras

There are plenty of equivalence relations in modelling computing systems. They arise, for example, where different notations represent the same data, or different programs implement the same specification. In this section, we show how an equivalence relation on an algebra may allow us to construct a new algebra, called the quotient or factor algebra, which turns out to be an invaluable tool in modelling.

7.8.1 Equivalence Relations and Congruences

Recall the definition of an equivalence relation on a set $S$.

**Definition (Equivalence Relations)** Let $S$ be any non-empty set. An *equivalence relation* $\equiv$ on $S$ is a binary, reflexive, symmetric and transitive relation on $S$, This means that $\equiv \subseteq S \times S$ which, writing $\equiv$ in infix notation, satisfies the following properties: the relation is

- **Reflexive**
  \[(\forall x \in S)[x \equiv x]\]
- **Symmetric**
  \[(\forall x \in S)(\forall y \in S)[x \equiv y \text{ implies } y \equiv x]\]
- **Transitive**
  \[(\forall x \in S)(\forall y \in S)(\forall z \in S)[x \equiv y \text{ and } y \equiv z \text{ implies } x \equiv z]\]

Given an equivalence relation $\equiv$ on a set $S$, we can define for each element $x \in S$ the *equivalence class*

\[ [x] \subseteq S \]

of $x$ with respect to $\equiv$ to be the set

\[ [x] = \{ y \mid y \equiv x \}. \]

of all elements in $S$ equivalent with $x$.

It is easily shown that the equivalence classes of $S$ with respect to $\equiv$ form a *partition* of the set $S$, i.e., a collection of disjoint subsets whose union is the set $S$. We let $S/\equiv$ denote the set of all equivalence classes of members of $S$ with respect to $\equiv$,

\[ S/\equiv = \{ [x] \mid x \in S \}. \]

We call $S/\equiv$ the *quotient* or factor set of $S$ with respect to the equivalence relation $\equiv$.

Given a quotient set $S/\equiv$, we can choose an element from each equivalence class in $S/\equiv$ as a *representative* of that class.

**Definition** A subset $T \subseteq S$ of elements, such that every equivalence class $[x]$ has some representative $r \in T$ and no equivalence class has more than one representative $r \in T$, is known as a *transversal* or, in certain circumstances, a *set of canonical representatives* or *normal forms* for $S/\equiv$.

The special kind of equivalence relation on the carrier set of an algebra $A$ which is “compatible” with the operations of $A$ is known as a *congruence*. 

7.8. CONGRUENCES AND QUOTIENT ALGEBRAS

Definition (Congruence) Let $A$ be an $S$-sorted $\Sigma$-algebra and

$$\equiv = \{s \subseteq A_s \mid s \in S\}$$

an $S$-indexed family of binary relations on $A$, such that for each $s \in S$ the relation $\equiv_s$ is an equivalence relation on the carrier set $A_s$. Then $\equiv$ is a $\Sigma$-congruence on $A$ if, and only if, the relations $\equiv_s$ satisfy the following substitution condition: for each function symbol $f : s(1) \times \cdots \times s(n) \to s$ and any $a_1, a'_1 \in A_{s(1)}, \ldots, a_n, a'_n \in A_{s(n)}$,

$$a_1 \equiv a'_1 \text{ and } \cdots \text{ and } a_n \equiv a'_n \text{ implies } f^A(a_1, \ldots, a_n) \equiv_s f^A(a'_1, \ldots, a'_n).$$

We let $\text{Con}(A)$ denote the set of all $\Sigma$-congruences on $A$. If $\equiv$ is a $\Sigma$-congruence on $A$, then for any $a \in A_s$, $[a]$ denotes the equivalence class of $a$ with respect to $\equiv_s$,

$$[a] = \{a' \in A_s \mid a' \equiv_s a\}.$$ 

Sometimes we index congruences $\equiv^\theta$, $\equiv^\psi$, etc., in which case we write $[a]_{\equiv^\theta}$ for the equivalence class of $a$ with respect to $\equiv^\theta$.

7.8.2 Quotient Algebras

The substitution condition of a congruence is precisely what is required in order to perform the following construction.

Definition (Quotient Algebra) Let $\equiv$ be a $\Sigma$-congruence on an $S$-sorted $\Sigma$-algebra $A$. The quotient algebra $A/\equiv$ of $A$ by the congruence $\equiv$ is the $\Sigma$-algebra with $S$-indexed family of carrier sets

$$(A/\equiv) = \langle (A/\equiv)_s \mid s \in S \rangle,$$

where for each sort $s \in S$,

$$(A/\equiv)_s = (A_s/\equiv_s),$$

and the constants and operations of the quotient algebra are defined as follows.

For each constant symbol $c : \to s \in \Sigma$, we interpret

$$c^{A/\equiv} = [c^A].$$

For each function symbol $f : s(1) \times \cdots \times s(n) \to s$, we interpret

$$f^{A/\equiv}([a_1], \ldots, [a_n]) = [f^A(a_1, \ldots, a_n)].$$

We must check that what we have called the quotient algebra is indeed a $\Sigma$-algebra. The point is that the definition of $f^{A/\equiv}$ we have given depends on the choice of representations.

Lemma Let $A$ be an $S$-sorted $\Sigma$-algebra and $\equiv$ a $\Sigma$-congruence on $A$. Then, the quotient algebra $A/\equiv$ is an $S$-sorted $\Sigma$-algebra.

Proof Since the carriers of $A$ are non-empty and we have defined the constants, we need only check that for each function symbol $f : s(1) \times \cdots \times s(n) \to s$, the corresponding operation $f^{A/\equiv}$ is well-defined as a function

$$f^{A/\equiv} : (A/\equiv)_{s(1)} \times \cdots \times (A/\equiv)_{s(n)} \to (A/\equiv)_s.$$
Consider any \( a_i, a'_i \in A_{s(i)} \) for \( 1 \leq i \leq n \) and suppose that \( a_i \equiv_{s(i)} a'_i \) for each \( 1 \leq i \leq n \). We must show

\[
f^{A/\equiv}([a_1], \ldots, [a_n]) = f^{A/\equiv}([a'_1], \ldots, [a'_n]),
\]

i.e., \( f^{A/\equiv}([a_1], \ldots, [a_n]) \) does not depend upon the choice of representatives for the equivalence classes \( [a_1], \ldots, [a_n] \). By assumption, \( \equiv \) is a \( \Sigma \)-congruence. So by definition, if \( a_i \equiv_{s(i)} a'_i \) for each \( 1 \leq i \leq n \), then

\[
f^{A}(a_1, \ldots, a_n) \equiv_s f^{A}(a'_1, \ldots, a'_n),
\]

i.e.,

\[
[f^{A}(a_1, \ldots, a_n)] = [f^{A}(a'_1, \ldots, a'_n)].
\]

Then, by definition of \( f^{A/\equiv} \),

\[
f^{A/\equiv}([a_1], \ldots, [a_n]) = f^{A/\equiv}([a'_1], \ldots, [a'_n]).
\]

\[ \square \]

**Examples**

1. For any \( S \)-sorted \( \Sigma \)-algebra, the family \( \leq_s | s \in S \) \), where \( \leq_s \) is the equality relation on \( A_s \), is a \( \Sigma \)-congruence on \( A \), known as the *equality, null or zero congruence* \( \equiv \).

   What is the relationship between \( A \) and \( A/\equiv \)?

2. The \( S \)-indexed family \( A^2 = <A^2_s | s \in S> \), where \( A^2 = A_s \times A_s \), is also a \( \Sigma \)-congruence on \( A \), known as the *unit congruence*.

   As an exercise, we leave it to the reader to check that \( A/A^2 \) is a unit algebra.

3. Consider the commutative ring of integers. For any \( n \in \mathbb{N} \), we define a relation for \( x, y \in \mathbb{Z} \) by

\[
x \equiv^n y \text{ if, and only if, } x \mod n = y \mod n.
\]

Equivalently, \( x \equiv^n y \) means that

- if \( x \geq y \), then \( x - y = kn \) for some \( k \in \mathbb{N} \), and
- if \( y \geq x \), then \( y - x = kn \) for some \( k \in \mathbb{N} \).

It is easy to check that \( \equiv^n \) is an equivalence relation and, indeed a congruence on the algebra.

### 7.9 Homomorphism Theorem

We will now gather together the technical ideas of Sections 7.7 and 7.8.

Given any \( \Sigma \)-homomorphism \( \phi : A \to B \), we can construct a \( \Sigma \)-congruence in a canonical way.
Definition (Kernel) Let $A$ and $B$ be $\Sigma$-algebras and $\phi : A \rightarrow B$ be a $\Sigma$-homomorphism. The kernel of $\phi$ is the binary relation

$$\equiv_{\phi} = \{ s : \in S \}$$

on $A$ defined by

$$a \equiv_{\phi} a' \text{ if, and only if, } \phi_s(a) = \phi_s(a')$$

for all $a, a' \in A_s$.

Lemma Let $\phi : A \rightarrow B$ be a $\Sigma$-homomorphism. The kernel $\equiv_{\phi}$ of $\phi$ is a $\Sigma$-congruence on $A$.

Proof Since equality on $B$ is an equivalence relation on $B$, it is easy to see that $\equiv_{\phi}$ is an equivalence relation on $A$. To check that the substitutivity condition holds, consider any function symbol $f : s(1) \times \cdots \times s(n) \rightarrow s \in \Sigma$ and any $a_1, a'_1 \in A_{s(1)}, \ldots, a_n, a'_n \in A_{s(n)}$. Suppose that we have

$$a_1 \equiv_{\phi_{s(1)}} a'_1, \ldots, a_n \equiv_{\phi_{s(n)}} a'_n.$$

Since $\phi$ is a $\Sigma$-homomorphism, then

$$\phi_s(f^A(a_1, \ldots, a_n)) = f^B(\phi_{s(1)}(a_1), \ldots, \phi_{s(n)}(a_n))$$

$$= f^B(\phi_{s(1)}(a'_1), \ldots, \phi_{s(n)}(a'_n)) \text{ by hypothesis}$$

$$= \phi_s(f^A(a'_1, \ldots, a'_n)).$$

So, by definition,

$$f^A(a_1, \ldots, a_n) \equiv_{\phi} f^A(a'_1, \ldots, a'_n).$$

Therefore, $\equiv_{\phi}$ satisfies the substitutivity condition. So $\equiv_{\phi}$ is a $\Sigma$-congruence on $A$. □

Conversely, given any $\Sigma$-congruence $\equiv$ on a $\Sigma$-algebra $A$, we can construct a $\Sigma$-homomorphism $\phi : A \rightarrow (A/\equiv)$ in a canonical way.

Definition (Natural map) Let $A$ be a $\Sigma$-algebra and $\equiv$ a $\Sigma$-congruence on $A$. The natural map or quotient map

$$\text{nat} : A \rightarrow (A/\equiv)$$

of the congruence is a family $\langle \text{nat}_s \mid s \in S \rangle$ of maps, defined for any $a \in A_s$ by

$$\text{nat}_s(a) = [a]_{\phi_s}.$$

For a kernel congruence $\equiv_{\phi}$, we let $\text{nat}_{\phi}$ denote the corresponding natural mapping.

Lemma Let $A$ be a $\Sigma$-algebra and $\equiv$ any $\Sigma$-congruence on $A$. The natural map

$$\text{nat} : A \rightarrow (A/\equiv)$$

of the congruence $\equiv$ is a $\Sigma$-epimorphism.

Proof To check that $\text{nat}$ is a $\Sigma$-homomorphism, consider any constant symbol $c : \rightarrow s \in \Sigma$. Then

$$\text{nat}_s(c^A) = [c^A] = c^{A/\equiv}.$$
by definition of $A/\equiv$.

For any function symbol $f : s(1) \times \cdots \times s(n) \to s \in \Sigma$, and any $a_1 \in A_{s(1)}, \ldots, a_n \in A_{s(n)}$, we have

$$\text{nat}(f^A(a_1, \ldots, a_n)) = [f^A(a_1, \ldots, a_n)] = f^A/\equiv([a_1], \ldots, [a_n])$$

by definition of $A/\equiv$,

$$= f^A/\equiv(\text{nat}(a_1), \ldots, \text{nat}(a_n)).$$

So the natural mapping $\text{nat}$ is a $\Sigma$-homomorphism and clearly $\text{nat}$ is surjective, i.e., $\text{nat}$ is a $\Sigma$-epimorphism. □

The First Homomorphism Theorem asserts that for any $\Sigma$-epimorphism $\phi : A \to B$, the homomorphic image $\phi(A)$ and the quotient algebra $A/\equiv_{\phi}$ are isomorphic, and hence, for the purposes of algebra, identical.

**Theorem (First Homomorphism Theorem)** If $\phi : A \to B$ is a $\Sigma$-epimorphism, then there exists a $\Sigma$-isomorphism

$$\psi : (A/\equiv_{\phi}) \to B$$

such that for all $a \in A$,

$$\phi(a) = \psi(\text{nat}_{\phi}(a))$$

i.e., the diagram shown in Figure 7.14 commutes, where $\text{nat}_{\phi} : A \to (A/\equiv_{\phi})$ is the natural

$$\begin{array}{ccc}
A & \xrightarrow{\phi} & B \\
\downarrow{\text{nat}_{\phi}} & & \downarrow{\psi} \\
A/\equiv_{\phi} & & \\
\end{array}$$

Figure 7.14: First Homomorphism Theorem.

mapping associated with the kernel $\equiv_{\phi}$ of $\phi$.

**Proof** Define $\psi$ by

$$\psi_s([a]_{\phi_s}) = \phi_s(a).$$

If $[a]_{\phi_s} = [b]_{\phi_s}$ then $\phi_s(a) = \phi_s(b)$. So $\psi_s([a]_{\phi_s}) = \psi_s([b]_{\phi_s})$ and therefore, $\psi_s([a]_{\phi_s})$ is uniquely defined.

To check that $\psi$ is a $\Sigma$-homomorphism, consider any constant symbol $c : \to s \in \Sigma$. Then

$$c^B = \phi(c^A) = \psi_s([c^A]_{\phi_s}) = \psi_s(c^{A/\equiv_{\phi}}).$$

Consider any function symbol $f : s(1) \times \cdots \times s(n) \to s \in \Sigma$, and any $a_1 \in A_{s(1)}, \ldots, a_n \in A_{s(n)}$. Then

$$f^B(\psi_{s(1)}([a_1]_{\phi_{s(1)}}), \ldots, \psi_{s(n)}([a_n]_{\phi_{s(n)}})) = f^B(\phi_{s(1)}(a_1), \ldots, \phi_{s(n)}(a_n)).$$
7.9. HOMOMORPHISM THEOREM

by definition of $\psi_s$;

$$= \phi_s(f^A(a_1, \ldots, a_n))$$

since $\phi$ is a $\Sigma$-homomorphism;

$$= \psi_s([f^A(a_1, \ldots, a_n)])$$

by definition of $\psi_s$;

$$= \psi_s(f^A/_{\equiv \phi}[a_1]_{\phi(1)}, \ldots, [a_n]_{\phi(n)})$$

by definition of $A/_{\equiv \phi}$.

Therefore, $\psi$ is a $\Sigma$-homomorphism.

Since $\psi$ is surjective, $\phi$ is surjective. For any $a, a' \in A$, if $[a]_\phi \neq [a']_\phi$ then $\phi(a) \neq \phi(a')$, and so

$$\psi([a]_\phi) \neq \psi([a']_\phi).$$

Therefore, $\psi$ is also injective, and hence bijective. So $\psi$ is a $\Sigma$-isomorphism. \qed

Example

In Section 7.6, we modelled the implementation of a data type on a digital computer using a numbering. We showed that:

Suppose a data type is modelled by a $\Sigma$-algebra $A$. If the data type is implementable on a digital computer, then $A$ can be coded using a $\Sigma$-algebra $\Omega$ of natural numbers and a $\Sigma$-epimorphism

$$\alpha : \Omega \to A.$$ 

Now, applying the First Homomorphism Theorem to this situation, we conclude immediately that

Suppose a data type is modelled by a $\Sigma$-algebra $A$. If the data type is implementable on a digital computer, then $A$ is $\Sigma$-isomorphic to a quotient algebra $\Omega/_{\equiv_\alpha}$ of natural numbers, i.e.,

$$A \cong \Omega/_{\equiv_\alpha}.$$
Exercises for Chapter 7

1. The translations of natural number representations from decimal to another radix $b > 0$, and back again, are part of the input and output procedures of many computations. Find algorithms that accomplish these transformations.

2. Let $G_1 = (\mathbb{R}; 0, +, -)$ and $G_2 = (\mathbb{R}_+, 1, ., ^{-1})$. Which of the following are $\Sigma_{\text{Group}}$-homomorphisms from $G_1$ to $G_2$:
   a. $f(x) = \log_2(x^k)$ for any fixed $k$;
   b. $f(x) = 2^{x^{-k}}$ for any fixed $k$;
   c. $f(x) = \sin(x)$; and
   d. $f(x) = x^2$?

3. Show that there is an isomorphism between the algebra
   $$(\{tt, ff\}; tt, ff; Or, Not)$$
   and the algebra
   $$(\{0, 1\}; 0, 1; \text{Vor}, \text{Vnot})$$
   with $Vnot(0) = 1$ and $Vnot(1) = 0$, and
   $$\text{Vor}(0, 0) = 0 \quad \text{Vor}(1, 1) = 1$$
   $$\text{Vor}(1, 0) = 1 \quad \text{Vor}(0, 1) = 1$$

4. Show that the relation $\cong$ of isomorphism is an equivalence relation, i.e., isomorphism is reflexive, symmetric and transitive on the class $\text{Alg}(\Sigma)$ of all $\Sigma$-algebras.

5. Let $\Sigma_{\text{Ring}}$ be the signature of rings. Let
   $$R_1 = (\mathbb{Z}; 0, 1, +, -, )$$
   be the ring of integers and
   $$R_2 = (\mathbb{Z}_n; 0, 1, +, -, )$$
   be the ring of cyclic integers for $n \geq 2$. Show that
   $$\phi : \mathbb{Z} \rightarrow \mathbb{Z}_n$$
   defined for $x \in \mathbb{Z}$ by $\phi(x) = x \mod n$ is a $\Sigma_{\text{Ring}}$-homomorphism.

6. Let $\Sigma_{\text{Group}}$ be the signature of groups. Let $G_1$ and $G_2$ be any groups. Let $\phi : G_1 \rightarrow G_2$ be any map. Show that if for all $x \in G_1$,
   $$\phi(x \cdot y) = \phi(x) \cdot \phi(y)$$
   then
   $$\phi(x^{-1}) = \phi(x)^{-1}.$$ 
   Hence, $\phi$ is a $\Sigma_{\text{Group}}$-homomorphism if, and only if, $\phi$ preserves the multiplication operation.
7. Let $G_1 = (G_1, o_1, e_1)$ and $G_2 = (G_2, o_2, e_2)$ be groups. Let $\phi : G_1 \to G_2$ be a group homomorphism. Define the kernel of $\phi$ by

$$ ker(\phi) = \{ g \in G_1 \mid \phi(g) = e_2 \}. $$

Show that

$\phi$ is an injection $\iff ker(\phi) = \{ e_1 \}$. 

8. Prove the lemma in Section 7.3.2.

9. Prove the following results using the Principle of Induction:

a. For any natural number $n > 0$,

$$ 1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n + 1)(n + 2). $$

b. For any natural number $n > 3$,

$$ 2^n < n! $$

c. For any natural number $n > 0$,

$$ n^3 - n $$

is divisible by 6.

10. Which of the following equations are true of the natural numbers?

a. For any natural number $n > 0$,

$$ 1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2. $$

b. For any natural number $n \geq 0$,

$$ n^2 + n + 41 $$

is a prime number.

In each case give a proof using the Principle of Induction or give a counter-example.

11. Prove the following using the Principle of Induction. For any $r \geq 1$ and all $n \in \mathbb{N}$,

$$ \sum_{i=1}^{n} i(i + 1)(i + 2) \cdots (i + r - 1) = \frac{1}{r+1}n(n + 1)(n + 2) \cdots (n + r) $$

12. Prove the Principle of Course of Values Induction from the Principle of Induction on the natural numbers.

13. Using the primitive recursive equations, calculate by substitution the values of $\text{add}(4, 7)$ and $\text{mult}(4, 7)$.

14. Show that $\text{exp} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by $\text{exp}(n, m) = m^n$ is primitive recursive over multiplication.
15. Let \( g : \mathbb{N}^k \to \mathbb{N}^k \) be a total function. A function \( f : \mathbb{N} \times \mathbb{N}^k \to \mathbb{N}^k \) is defined by *iteration* over \( g \) if

\[
\begin{align*}
  f(0, x) &= x \\
  f(n + 1, x) &= g(f(n, x)).
\end{align*}
\]

Show that

\[
f(n, x) = g \circ \cdots \circ g(x) \quad n \text{ times}
\]

\[
= g^n(x).
\]

Check that if \( f \) is definable by iteration over \( g \) then it is definable by primitive recursion over the identity function \( i : \mathbb{N}^k \to \mathbb{N}^k \) and \( g \).

16. Let \( \Sigma \) be the signature

<table>
<thead>
<tr>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
</tr>
<tr>
<td>constants</td>
</tr>
<tr>
<td>operations</td>
</tr>
</tbody>
</table>

and let

\[
A_1 = (\mathbb{N}; 0, n + 1) \quad \text{and} \quad A_2 = (\mathbb{N}; k, g)
\]

where \( g : \mathbb{N} \to \mathbb{N} \) is any function. Show that any \( \Sigma \)-homomorphism \( \phi : A_1 \to A_2 \) is primitive recursive in \( g \).

17. Give a definition that generalises the concept of primitive recursion on \( \mathbb{N} \) to functions of the form

\[
f : \mathbb{N} \times A \to B
\]

where \( A \) and \( B \) are non-empty sets. Show that \( f \) is uniquely defined by adapting the proof of Lemma 7.4.3.

18. Give a definition that generalises the concept of iteration on \( \mathbb{N} \) (in Question 15) to functions of the form

\[
f : \mathbb{N} \times A \to B
\]

where \( A \) and \( B \) are non-empty sets. Show that iterations are primitive recursions. Under what circumstances can primitive recursions be defined using iterations?
Chapter 8

Abstract Data Type of Real Numbers

What are numbers? Numbers are very abstract data. Number systems are data types that are used to measure quantities and to calculate new quantities from old ones. The natural numbers, integers, rational numbers, real numbers and complex numbers are a sequence

\[ \mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C} \]

of number systems, one extending the next, that either measure more, or calculate better, or both. This is a rather utilitarian view, and seems too simple to explain the depth of our interest and knowledge of numbers in mathematics. However, it is a good starting point for thinking about the origins of number systems, and, therefore, about the origins and foundations of data.

When we gave examples of algebras of real numbers in Section 3.5, we discussed briefly the real numbers and described it as a data type that models the line allowing us to measure any line segment. Recall how the rational number system leaves many gaps in measurements.

The rational numbers are a number system designed to model the process of measuring quantities. Operationally, in order to measure, some unit is chosen and subdivided to subunits, corresponding with the whole and fractional parts of rational numbers. It is easy to construct geometrical figures that cannot be measured exactly. For example, even in measuring line segments a problem arises: the hypotenuse of a right-angled triangle with unit sides is \( \sqrt{2} \) units. Or the circumference of a circle of diameter 1 is known to be \( \pi \). Neither \( \sqrt{2} \) or \( \pi \) are rational numbers. Thus, these lengths indicate fundamental gaps in the rational number model of measurements. Specifically, we think of the real numbers as a number system designed to assign a number to measure every length exactly.

In this chapter we will investigate the data type of real numbers in the same manner that we investigated the data type of natural numbers. We will begin by considering the problem:

*To design and build a data type to measure the line.*

As with the natural numbers, the key ideas about building representations of the real numbers were discovered only in the nineteenth century. We will introduce the different methods of Richard Dedekind and Georg Cantor, both of which construct approximations of real numbers from rational numbers.

Dedekind’s and Cantor’s are just two among many representations of the real numbers. The question arises: Are they equivalent? More generally, we ask:

*When are two constructions or representations of the real numbers equivalent?*
In our mathematical theory of data, this becomes the mathematical question:

*When are two algebras of real numbers isomorphic?*

We answer this question in stages. First, we prove a beautiful theorem that shows there is a set of axioms that is sufficient to characterise the real numbers up to isomorphism. The axioms define algebras called

*complete ordered fields*

and the theorem says that

*all complete ordered fields are isomorphic.*

The axioms define the properties of the basic algebraic operations + and . and the order relation $<$ on the real numbers.

Next, we will give the full construction of the data type of real numbers using the method of Cantor, who represented a real number by an infinite process of approximate measurements by rational numbers. This is an involved construction involving infinite sequences of rational numbers. We prove that the algebra of Cantor real numbers satisfies the axioms of a complete ordered field. Hence, we can conclude that

*all algebras satisfying the axioms of a complete ordered field are isomorphic to the algebra of Cantor real numbers.*

In Sections 8.1 and 8.2, we explain these ideas in some detail, before devoting ourselves to the proofs of the theorems in Sections 8.3, 8.4 and 8.5. Finally, in Section 8.6, we look at computing with the real numbers using so called “exact” representations and the familiar, but flawed, fixed and floating point representations.

In this chapter we will barely scratch the surface of the theory of the real numbers. There are many excellent books available; for example, Bertrand Russell [1967] gives a full and excellent mathematical account of the naturals, rationals and reals, which we have largely followed in Sections 8.3, and 8.4 of this chapter. For details and related results see Further Reading.

### 8.1 Representations of the Real Numbers

The development of a number system that meets the requirements of measuring exactly the line or continuum has proved to be a long and complex process. Our understanding of the real numbers has progressed slowly, influenced by the needs of mathematics and scientific calculations. Indeed, we can follow its development in some detail over 2500 years of history. Establishing theoretical foundations for mathematical developments such as the calculus have involved sorting out a number of subtle, conflicting and inconsistent ideas about the reals. For example, the development of the calculus introduced infinitesimals and raised the fundamental question:

*To what extent are reals finite quantities, or can be represented by finite objects?*
8.1. REPRESENTATIONS OF THE REAL NUMBERS

The development of mathematics and its applications in physics has taken place with good techniques and intentions, but poor foundations and understanding. Only in the middle of the nineteenth century did mathematics achieve the level of precision, rigour and abstractness that is necessary to be able to settle the basic questions about the reals and other numerical data types. This conceptual maturity is exactly what is needed to understand data in computing. The concept of the continuum has been fundamental for millennia and is immensely rich. Research on the foundations of the continuum continues: for example, on models for computation with the reals, and on the uses of infinitesimals (see Further Reading).

We will discuss some of the methods of constructing representations of the reals from the rationals. As we will see in the next section, these constructions, representations or implementation methods for the real numbers can be proved to be equivalent. Let us review the problem in detail.

8.1.1 The Problem

The line or continuum is not adequately represented by the data type of rational numbers: there are gaps such as $\sqrt{2}$ and $\sqrt{3}$; indeed, there are infinitely many gaps and there are several ways of formulating and proving properties that show that

\textit{most points on the line cannot be measured exactly by rational numbers.}

The problem is:

\textit{To create new numbers which will allow us to represent faithfully the line, and which constitute a number system that measures and calculates exactly with all points and distances.}

The required number system $\mathbb{R}$ is called the real number system or the data type of real numbers.

The rational numbers $\mathbb{Q}$ form the data type we use to make measurements. To create the real numbers $\mathbb{R}$ we will use rational numbers that measure the line \textit{approximately}. The real numbers will be constructed from the rationals, just as the rationals were constructed from the integers, and the integers from the natural numbers. The reals must have certain algebraic properties. In fact the number system $\mathbb{R}$ must support a great deal more, namely many centuries worth of concepts, methods and theorems in geometry and analysis! Let us summarise the requirements for the data type as follows:
Informal Algebraic Requirements

1. $\mathbb{R}$ will be an extension of the rational number system $\mathbb{Q}$.

2. $\mathbb{R}$ will have algebraic operations, including addition, subtraction, multiplication and division for calculation.

Informal Geometric Requirements

3. $\mathbb{R}$ will have an ordering relation $<$ that corresponds with the ordering of the line.

4. Every element of $\mathbb{R}$ can be approximated by an element of $\mathbb{Q}$.

5. The number system $\mathbb{R}$ must represent the line completely.

Analytical Requirements

6. The number system $\mathbb{R}$ must be able to represent geometric objects and methods in a faithful way, through the theory of coordinate geometry based on algebraic equations of curves and surfaces.

7. The number system $\mathbb{R}$ must be able to represent analytic objects and methods in a faithful way, through the theory of the differential and integral calculus.

Obviously, it is an important task to build this data type $\mathbb{R}$ and a huge enterprise to check on all these requirements, some of which are not precisely defined such as 6 and 7. We will merely sketch two methods of data representation, due to Richard Dedekind and Georg Cantor, and comment on why they meet the requirements 1–5. At first sight Dedekind’s method seems more abstract, but it is based directly on a geometric intuition. It was discovered in 1858 and published rather later, in 1872.

8.1.2 Method of Richard Dedekind (1858)

The idea that a line is full of points and that there are no gaps caused by missing points, is an axiom of geometry. The analysis of Dedekind focuses on a precise geometric characterisation of this idea which is called the continuity or completeness of the line. The axiom by which he attributed to the line its continuity is this:

**Axiom (Dedekind’s Continuity Axiom)** If all points of the straight line fall into two classes such that every point of the first class lies to the left of every point of the second class, then there exists one and only one point which produces this division of all points into two classes, this severing of the straight line into two portions.

The property is illustrated in Figure 8.1.

The idea is adapted to build a representation of a point using two sets of rational numbers.
8.1. REPRESENTATIONS OF THE REAL NUMBERS

\[ A_1 \quad \bullet \quad A_2 \]

\[ A_1 \quad \bullet \quad A_2 \]

Figure 8.1: Dedekind’s continuity axiom; \( A_1 \) is the first class and \( A_2 \) the second class.

**Definition** A cut is a pair \((A_1, A_2)\) of sets of rational numbers such that

(i) \( A_1 \cup A_2 = \mathbb{Q} \); and

(ii) for any \( a_1 \in A_1 \) and \( a_2 \in A_2 \), \( a_1 < a_2 \).

Let \( DCut \) be the set of all cuts.

A cut consists of two sets of rational numbers, and so models the division of the line into two classes of measurable points. Dedekind’s axiom leads us to propose that:

*Each cut represents a point on the line.*

In particular,

*a point on the line can be modelled by the sets of all measurable points to its left and right.*

Let \( R_{Dedekind} \) be the set of all points represented by \( DCut \).

First, let us look at some simple examples of cuts.

**Example (Rational Cuts)** Notice that any rational number \( \frac{p}{q} \) can define a cut in two simple ways: a *right cut*

\[ A_1 = \{ a \in \mathbb{Q} | a < \frac{p}{q} \} \quad \text{and} \quad A_2 = \{ a \in \mathbb{Q} | a \geq \frac{p}{q} \} \]

where the rational \( \frac{p}{q} \) is the least among the second class \( A_2 \), or a *left cut*

\[ A_1 = \{ a \in \mathbb{Q} | a \leq \frac{p}{q} \} \quad \text{and} \quad A_2 = \{ a \in \mathbb{Q} | a > \frac{p}{q} \} \]

where the rational \( \frac{p}{q} \) is the greatest among the first class \( A_1 \). For example, consider the cuts for a commonly used rational approximation to \( \pi \):

\[ A_1 = \{ a \in \mathbb{Q} | a < \frac{22}{7} \} \quad \text{and} \quad A_2 = \{ a \in \mathbb{Q} | a \geq \frac{22}{7} \}; \]

here \( \frac{22}{7} \) is the least number in \( A_2 \). Or,

\[ A_1 = \{ a \in \mathbb{Q} | a \leq \frac{22}{7} \} \quad \text{and} \quad A_2 = \{ a \in \mathbb{Q} | a > \frac{22}{7} \}; \]

here \( \frac{22}{7} \) is the greatest number in \( A_1 \). Clearly, \( \frac{22}{7} \) has two different cuts representing it.
Example (Irrational Cuts) Of course, the purpose of cuts is to define numbers that are not rational. Here are cuts that are not determined by rational points:

\[ A_1 = \{a \in \mathbb{Q} | a^2 < 2\} \quad \text{and} \quad A_2 = \{a \in \mathbb{Q} | a^2 \geq 2\}, \]

or

\[ A_1 = \{a \in \mathbb{Q} | a^2 \leq 2\} \quad \text{and} \quad A_2 = \{a \in \mathbb{Q} | a^2 > 2\}. \]

There does not exist a unique rational number which is either a least upper bound or a greatest lower bound for these sets. Those bounds are \(\sqrt{2}\) which is not an element of either \(A_1\) or \(A_2\). The idea is that these cuts represent \(\sqrt{2}\).

However, the cut itself is an object that represents either the greatest among the lower bounds for \(A_1\) or least among the upper bounds for \(A_2\).

With reference to the requirements, we have seen that the cuts are made from rationals and that the rational numbers \(\mathbb{Q}\) can be embedded into \(D\text{Cut}\). To proceed we must define:

(i) when two cuts are equivalent representations of the same point;

(ii) how to add, subtract, multiply and divide cuts; and

(iii) how to order two cuts.

We will omit this discussion and turn instead to the second method of representing real numbers; for further material on cuts see the exercises and Further Reading.

8.1.3 Method of Georg Cantor (1872)

Imagine a process that can measure approximately a point with increasing and unlimited accuracy. The process generates an infinite sequence of points that can be measured by the rational numbers

\[ a_1, a_2, \ldots, a_n, \ldots, a_m, \ldots \in \mathbb{Q}. \]

At each stage \(n\), it is possible to improve on \(a_n \in \mathbb{Q}\) at a later stage \(m > n\) with a closer measurement \(a_m \in \mathbb{Q}\).

The idea of Cantor’s method is to use an infinite sequence of rationals, to approximate a point to an arbitrary degree of accuracy. A point will be represented by an infinite sequence

\[ a_1, a_2, \ldots, a_n, \ldots \]

of rational numbers that get closer and closer as the sequence grows. This means that the difference

\[ a_n - a_m \]

becomes increasingly small as \(n\) and \(m\) become increasingly large. The type of sequences Cantor used are now known as Cauchy sequences. The real numbers will be represented using these sequences.

Definition (Cauchy Sequence) A sequence of rationals \(a_1, a_2, \ldots, a_n, \ldots\) is a Cauchy sequence of rationals if for any rational number \(\epsilon > 0\), there is a natural number \(n_0\), such that for all \(n, m > n_0\), we have

\[ |a_n - a_m| < \epsilon. \]

Let \(CSeq\) be the set of all Cauchy sequences of rationals.
8.1. REPRESENTATIONS OF THE REAL NUMBERS

Each Cauchy sequence is a measuring process that represents a point on the line. Let $R_{\text{Cantor}}$ be the set of points represented by $CSeq$. With each Cauchy sequence $a_1, a_2, \ldots$ the approximation process is this: given any measure of accuracy $\epsilon > 0$, however small, it is possible to choose a stage $N$ in the sequence such that for all later stages $n, m > N$, the accuracy is within $\epsilon$, i.e.,

$$|a_n - a_m| < \epsilon.$$

8.1.4 Examples

(1) Now each rational number $\frac{p}{q}$ measures itself. Define a sequence

$$a_i = \frac{p}{q}$$

for $i = 1, 2, \ldots$. Clearly, for any accuracy $\epsilon > 0$ and $N > 1$, we have for all $n, m > N$,

$$|a_n - a_m| = \left| \frac{p}{q} - \frac{p}{q} \right| = \left| \frac{p}{q} \right| = 0 < \epsilon.$$

(2) The idea of a Cauchy sequence is implicit in the standard decimal notation for real numbers.

**Lemma** The decimal representation of a number defines a Cauchy sequence of real numbers.

**Proof** Let

$$d.d_1d_2 \ldots d_nd_{n+1} \ldots$$

be any decimal representation, where

$$d \in \mathbb{Z} \quad \text{and} \quad d_n \in \{0, 1, \ldots, 9\} \quad \text{for} \quad n = 1, 2, \ldots.$$

Note that $d$ is the “whole number part” before the decimal point, and the $d_n$ form the “decimal part” after the decimal point.

Define a new sequence $a = a_1, a_2, \ldots, a_n, \ldots$ of rational numbers by

$$a_1 = d.d_1$$
$$a_2 = d.d_1d_2$$
$$\vdots$$
$$a_n = d.d_1d_2 \ldots d_n$$
$$\vdots$$

These finite decimals are rational numbers, e.g.,

$$a_n = d + \frac{d_1}{10} + \frac{d_2}{10^2} + \cdots + \frac{d_n}{10^n}.$$
We show that \( a \) is a Cauchy sequence.

Given any \( \epsilon > 0 \), choose \( N \) such that
\[
10^{-N} = 0.00\ldots01 \quad N \text{ decimal places}
\]
\[
< \epsilon.
\]
Now, for any \( m, n > N \), if (say) \( n > m \), then
\[
|a_n - a_m| = 0.00\ldots0d_{m+1}\ldots d_n
\]
\[
< 0.00\ldots01 \quad N \text{ decimal places}
\]
\[
< 10^{-N}
\]
\[
< \epsilon.
\]

Thus, \( a \) is a Cauchy sequence. \( \square \)

Unlike the case of Dedekind cuts, the set \( CSeq \) contains infinitely many representations for the same point or real. To proceed we must define:

(i) when two Cauchy sequences are equivalent representations of the same point;

(ii) how to add, subtract, multiply and divide Cauchy sequences; and

(iii) how to order two Cauchy sequences.

These are not straightforward. We will explain each relation and operation in turn in Section 8.4.

### 8.2 The real numbers as an abstract data type

A real number can be defined or represented in several ways. Standard methods, such as those of Cantor, Dedekind, or of infinite decimals, nested intervals and continuous fractions, are based on approximating real numbers to any degree of accuracy using rational numbers. The rational numbers and integers also have a number of representation methods. Thus, we are led to the question:

When are two representations of the real numbers equivalent?

In this section we will apply the line of thought concerning the naturals as an abstract data type, expressed in Section 7.5, to the real numbers. We expect to answer the question using

(i) algebras to model concrete representations of the reals, such as
\[
A_{\text{Cantor}} \quad \text{and} \quad A_{\text{Dedekind}}
\]

and

(ii) isomorphisms between algebras to model the equivalence of concrete representations of the reals, such as
\[
\phi : A_{\text{Cantor}} \to A_{\text{Dedekind}}.
\]
8.2. **THE REAL NUMBERS AS AN ABSTRACT DATA TYPE**

Furthermore, we will explain how the real numbers can be characterised axiomatically in terms of properties of their operations and ordering.

The reals are definable uniquely (up to isomorphism) as any algebra satisfying the axioms of a

*complete ordered field*,

a concept we will explain in this section. This means the following:

*All the commonly used properties of the reals can be proved from the axioms of a complete ordered field; and so these common properties will hold in any algebra satisfying the axioms.*

*All complete ordered fields are isomorphic.*

*The standard constructions of the reals by Cantor’s method, and so on, form complete ordered fields.*

We will give a detailed account, with full proofs, of these facts.

### 8.2.1 Real numbers as a field

Let us begin our explanation of these concepts and theorems by considering the basic operations on reals of addition, subtraction, multiplication, and division. We will specify these operations using axioms, as we did in Chapter 5. First, we may name these operations in the signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>field</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1 : → field</td>
</tr>
<tr>
<td>operations</td>
<td><em>+</em>, <em>-</em>, <em>·</em>, <em>⁻¹</em> : field × field → field</td>
</tr>
</tbody>
</table>

Among the wide range of algebras with this signature, we wish to classify those \( \Sigma_{Field} \) algebras that have precisely the algebraic properties of the real numbers. The axioms of a field specify some very simple and familiar properties of the operations on reals and are given below.
<table>
<thead>
<tr>
<th>Axioms</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associativity of addition</td>
<td>$(\forall x)(\forall y)(\forall z)[(x + y) + z = x + (y + z)]$</td>
</tr>
<tr>
<td>Identity for addition</td>
<td>$(\forall x)[x + 0 = x]$</td>
</tr>
<tr>
<td>Inverse for addition</td>
<td>$(\forall x)[x + (-x) = 0]$</td>
</tr>
<tr>
<td>Commutativity of addition</td>
<td>$(\forall x)(\forall y)[x + y = y + x]$</td>
</tr>
<tr>
<td>Associativity of multiplication</td>
<td>$(\forall x)(\forall y)(\forall z)[(x.y).z = x.(y.z)]$</td>
</tr>
<tr>
<td>Identity for multiplication</td>
<td>$(\forall x)[x.1 = x]$</td>
</tr>
<tr>
<td>Inverse for multiplication</td>
<td>$(\forall x)[x \neq 0 \Rightarrow x.(x^{-1}) = 1]$</td>
</tr>
<tr>
<td>Commutativity of multiplication</td>
<td>$(\forall x)(\forall y)[x.y = y.x]$</td>
</tr>
<tr>
<td>Distribution</td>
<td>$(\forall x)(\forall y)(\forall z)[x.(y + z) = x.y + x.z]$</td>
</tr>
<tr>
<td>Distinctness</td>
<td>$0 \neq 1$</td>
</tr>
</tbody>
</table>

**Definition (Field)** Let $A$ be any $\Sigma_{Field}$ algebra satisfying the axioms of a field. Then $A$ is said to be a **field**.

Let $T_{Field}$ be the set of these 10 axioms and let

$$\text{Alg}(\Sigma_{Field}, T_{Field})$$

be the class of all fields.

Remember that there are many algebras that satisfy these axioms including infinite algebras such as rational numbers, real numbers and complex numbers, and finite algebras such as modulo $p$ arithmetic $\mathbb{Z}_p$ where $p$ is a prime number.

### 8.2.2 Real numbers as an ordered field

To the operations of a field we add the ordering relation. To do this we will create a new signature by combining the signatures of the field and the Booleans and adding a Boolean valued operation to test elements in the ordering.
8.2. THE REAL NUMBERS AS AN ABSTRACT DATA TYPE

\[
\begin{array}{|l|}
\hline
\textbf{signature} & \textit{Ordered Field} \\
\hline
\textbf{import} & \textit{Field, Booleans} \\
\hline
\textbf{sorts} & \\
\hline
\textbf{constants} & \\
\hline
\textbf{operations} & \geq : \textit{field} \times \textit{field} \rightarrow \textit{Bool} \\
\hline
\end{array}
\]

To the axioms of a field, we add the axioms that express the properties of the Booleans and, in particular, axioms that express the properties of the ordering. The operation \( x \geq y \) is specified by these six axioms.

\[
\begin{array}{|l|}
\hline
\textbf{axioms} & \textit{Ordered Field} \\
\hline
\textit{Reflexivity} & (\forall x)[x \geq x] \\
\hline
\textit{Antisymmetry} & (\forall x)(\forall y)[x \geq y \text{ and } y \geq x \Rightarrow x = y] \\
\hline
\textit{Transitivity} & (\forall x)(\forall y)(\forall z)[x \geq y \text{ and } y \geq z \Rightarrow x \geq z] \\
\hline
\textit{Total Order} & (\forall x)[x \geq 0 \text{ or } x = 0 \text{ or } -x \geq 0] \\
\hline
\textit{Addition} & (\forall x)(\forall y)(\forall x')[\forall y'][x \geq y \text{ and } x' \geq y' \Rightarrow x + x' \geq y + y'] \\
\hline
\textit{Multiplication} & (\forall x)(\forall y)(\forall z)[x \geq y \text{ and } z \geq 0 \Rightarrow z \cdot x \geq z \cdot y] \\
\hline
\end{array}
\]

\textbf{Definition} We derive from \( \leq \) some new relations as follows:

(i) the function \( \leq : \textit{field} \times \textit{field} \rightarrow \textit{Bool} \) which we define for all \( x, y \) by

\[
x \leq y \quad \text{if, and only if,} \quad y \geq x;
\]

(ii) the function \( > : \textit{field} \times \textit{field} \rightarrow \textit{Bool} \) which we define for all \( x, y \) by

\[
x > y \quad \text{if, and only if,} \quad x \geq y \text{ and } x \neq y;
\]

(iii) the function \( < : \textit{field} \times \textit{field} \rightarrow \textit{Bool} \) which we define for all \( x, y \) by

\[
x < y \quad \text{if, and only if,} \quad x > y.
\]

An element \( x \) of an ordered field is \textit{non-negative} if \( x \geq 0 \). An element \( x \) of an ordered field is \textit{positive} if \( x > 0 \).

The connection between the positive elements and the ordering \( \geq \) is simply:
Lemma (1) For all \( x, y \)

(i) \( x \geq y \Leftrightarrow x - y \geq 0 \);

(ii) \( x > y \Leftrightarrow x - y > 0 \).

Proof It is easy to deduce these properties from the axioms of an ordered field. Consider the implication \( \Rightarrow \) of (i). Suppose \( x \geq y \). Then if we add \(-y\) to both sides, we know from the addition axiom for ordered fields that

\[
x + (-y) \geq y + (-y).
\]

By the additive inverse axiom for fields, we have

\[
x - y \geq 0.
\]

The reverse implication \( \Leftarrow \) of (i) is a similar argument (in which \( y \) is added to \( x - y \geq 0 \)).

Statement (ii) follows from (i) thus:

\[
x > y \iff x \geq y \text{ and } x \neq y \quad \text{by definition of } >; \\
    \iff x - y \geq 0 \text{ and } x \neq y \quad \text{by statement (i)}; \\
    \iff x - y > 0 \quad \text{by definition of } >.
\]

\( \square \)

Definition (Ordered Field with Booleans) Let \( A \) be any \( \Sigma_{\text{Ordered Field}} \) algebra satisfying the axioms of a field and an ordered field. Then \( A \) is said to be an ordered field with Booleans.

Let \( T_{\text{Ordered Field}} \) be the set of these 16 field and ordering axioms and

\[
\text{Alg}(\Sigma_{\text{Ordered Field}}, T_{\text{Ordered Field}})
\]

the class of all ordered fields.

Again, there are many algebras that satisfy these axioms including the rational numbers and real numbers. However, the complex numbers do not form an ordered field! To prove this fact, we must get better acquainted with the ordering axioms by deducing some properties from its axioms.

Lemma (3) Let \( A \) be an ordered field with Booleans. In \( F_A \) we have

(i) \( 1 > 0 \);

(ii) \( n1 = 1 + 1 + \cdots + 1 \text{ (n times)} > 0 \);

(iii) for all \( x \in F_A \), if \( x \neq 0 \) then \( x^2 > 0 \).

Proof (i) Suppose for a contradiction that \( 1 < 0 \). Then, by the total order axiom, and since \( 1 \neq 0 \), we have

\[
-1 > 0.
\]

By the multiplication axiom,

\[
(-1)(-1) > 0.
\]
8.2. **THE REAL NUMBERS AS AN ABSTRACT DATA TYPE**

But, by a lemma in Section 5.3.2

\((-1).(-1) = 1.1.\)

So

\[\begin{align*}
1 & = 1.1 \\
& > 0,
\end{align*}\]

which contradicts our assumption. Hence, in fact, \(1 > 0.\)

\((\ii)\) Applying the addition axiom to \(1 > 0,\) we deduce immediately that

\[1 + 1 > 0, 1 + 1 + 1 > 0, \ldots, 1 + 1 + \cdots + 1 > 0\]

and hence for any \(n \in \mathbb{Z},\)

\[n1 > 0.\]

\((\iii)\) Let \(x \in F_1\) and \(x \neq 0.\) If \(x > 0\) then \(x^2 > 0\) by the multiplication axiom. If \(x < 0\) then \(-x > 0\) by the total ordering axiom and

\[(-x).(-x) > 0\]

by the multiplication axiom. However,

\[x^2 = (-x).(-x)\]

by the lemma in Section 5.3.2, and so if \(x < 0\) we also have

\[x^2 > 0.\]

\(\square\)

This lemma enables us to prove the following:

**Theorem** There does not exist an ordering on the complex numbers \(\mathbb{C}\) that allows it to be an ordered field.

**Proof** The complex numbers form a field containing the number \(i\) such that

\[i^2 = -1.\]

Suppose, for a contradiction, there exists an ordering \(\leq_{\mathbb{C}}\) on \(\mathbb{C}\) that satisfies the ordered field axioms.

By the Soundness of Deduction Principle (in Section 5.1.2), the properties of ordered fields proved in the above lemma are true of \(\leq_{\mathbb{C}}.\) In particular, we will examine the facts that

\((i)\) \(1^\mathbb{C} >_{\mathbb{C}} 0^\mathbb{C};\)

\((\ii)\) for all \(z \in \mathbb{C}, \ z \neq 0^\mathbb{C},\)

\[z >_{\mathbb{C}} 0^\mathbb{C}\]

implies \(z^2 >_{\mathbb{C}} 0^\mathbb{C}.\)
Let us ask: Is \( i >_C 0^C \) or \( 0_C >_C i \)?

If \( i >_C 0^C \) then by property (iii),
\[
i^2 >_C 0^C.
\]

But \( i^2 = -1 \), and so
\[
-1 >_C 0^C \quad \text{and} \quad 0^C <_C 1^C,
\]
which contradicts property (i).

Thus, we have shown \( i \) is neither positive nor negative and no such ordering \( <_C \) can exist.

\[\square\]

### 8.2.3 Completeness of the ordering

The ordering plays a crucial role in thinking about the continuum. Dedekind’s continuity axiom in Section 8.1.2 is based on ordering.

**Definition (Lower Bound and Greatest Lower Bound)** Let \( F \) be an ordered field and let \( S \) be a subset of \( F \).

(i) The element \( b \in F \) is a **lower bound** for \( S \) if, for all \( s \in S \), \( b \leq s \).

(ii) The element \( b \in F \) is a **greatest lower bound** for \( S \) if,

(a) \( b \) is a lower bound; and

(b) there is no other lower bound \( c \in F \) for \( S \), such that \( b < c \).

The condition (b) is equivalent to the following:

(b') for all \( c \in F \), if for all \( s \in S \), \( c < s \), then \( c \leq b \).

We write \( \text{glb}(S) \) for the greatest lower bound of \( S \), if it exists. Another commonly used term for the greatest lower bound is *infimum*, which we denote \( \inf(S) \).

**Example** Here is an example of a set
\[
S = \{1.5, 1.42, 1.415, 1.4143, 1.41422, \ldots \}
\]
of rational numbers that has lower bounds in the rationals \( Q \) but no greatest lower bound. However, if \( S \) is a subset of the real numbers, then we have
\[
\sqrt{2} = \text{glb}\{1.5, 1.42, 1.415, 1.4143, 1.41422, \ldots \}.
\]

**Definition (Upper Bound and Least Upper Bound)** Let \( F \) be an ordered field and let \( S \) be a subset of \( F \).

(i) The element \( b \in F \) is an **upper bound** for \( S \) if, for all \( s \in S \), \( b \geq s \).

(ii) The element \( b \in F \) is a **least upper bound** for \( S \) if,

(a) \( b \) is an upper bound; and

(b) there is no other upper bound \( c \in F \) for \( S \), such that \( c < b \).
The condition (b) is equivalent to the following:

(b) for all \( c \in F \), if for all \( s \in S \), \( c > s \), then \( c \geq b \).

We write \( \text{lub}(S) \) for the least upper bound of \( S \), if it exists. Another commonly used term for the least upper bound is \( \text{supremum} \), which we denote \( \text{sup}(S) \).

**Example** The set

\[ S = \{ 1.4, 1.41, 1.414, 1.4142, 1.41421, \ldots \} \]

of rational numbers has upper bounds in the rationals \( \mathbb{Q} \) but no least upper bound. However, if \( S \) is a subset of the real numbers, then we have

\[ \sqrt{2} = \text{lub}\{ 1.4, 1.41, 1.414, 1.4142, 1.41421, \ldots \} \].

**Definition (Completeness)** An ordered field \( F \) is said to be complete if every non-empty subset \( S \) that has a lower bound also possesses a greatest lower bound.

Clearly, from the examples of \( \sqrt{2} \), neither the lower bound, nor the upper bound, need be in \( S \).

This condition is a little complicated to write out as an axiom in a specification. First, we make some formulae for the ideas of lower bound and greatest lower bound.

That \( x \) is a lower bound for a set \( S \) is defined by the formula

\[ \text{lower bound}(x, S) \equiv (\forall s)[s \in S \Rightarrow x \leq s] \]

That \( x \) is a greatest lower bound for a set \( S \) is defined by the formula

\[ \text{greatest lower bound}(x, S) \equiv \text{lower bound}(x, S) \text{ and } (\forall c)[\text{lower bound}(c, S) \Rightarrow c \leq x] \]

Here is the specification:

<table>
<thead>
<tr>
<th>axioms</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completeness</td>
<td>((\forall S)[\emptyset \neq S \subseteq F \text{ and } (\exists c)[\text{lower bound}(c, S)]] \Rightarrow (\exists b)[\text{greatest lower bound}(b, S)])</td>
</tr>
</tbody>
</table>

**Definition (Complete Ordered Field with Booleans)** Let \( A \) be any \( \Sigma_{\text{Ordered Field}} \) algebra satisfying the axioms of a field, an ordered field and a complete ordered field. Then \( A \) is said to be a complete ordered field with Booleans.

Let

\[ T_{\text{Complete Ordered Field}} = T_{\text{Field}} \cup T_{\text{Ordered Field}} \cup T_{\text{Completeness}} \]

be the set of these 17 axioms and

\[ \text{Alg}(\Sigma_{\text{Ordered Field}}; E_{\text{Complete Ordered Field}}) \]

be the class of all complete ordered fields.
8.3 Uniqueness of the Real Numbers

The axioms of a complete ordered field constitute an axiomatic specification of real numbers. Indeed, the properties of being a complete ordered field characterise the real numbers in the following way:

**Theorem (Uniqueness)** Let $A$ and $B$ be any algebras of signature $\Sigma_{\text{Ordered Field}}$ satisfying the axioms of a complete ordered field. Then $A \cong B$.

Later, we will use this Uniqueness Theorem to prove the following:

**Theorem** All $\Sigma_{\text{Ordered Field}}$ algebras satisfying the axioms of a complete ordered field are isomorphic with $\mathbb{R}_{\text{Cantor}}$.

Any algebra that is a complete ordered field is a representation of the real numbers.

8.3.1 Preparations and Overview of Proof of the Uniqueness of the Reals

Consider, in detail, the concept of a $\Sigma_{\text{Ordered Field}}$-isomorphism. Applying the general definition of a homomorphism, given for arbitrary algebras in the previous chapter, to $\Sigma_{\text{Ordered Field}}$ algebras results in the following concept.

**Definition (Ordered Field Homomorphism)** Let $A$ and $B$ be any $\Sigma_{\text{Ordered Field}}$-algebras. Let the carriers of sort $\text{field}$ be $F_A$ and $F_B$, respectively.

A $\Sigma_{\text{Ordered Field}}$-homomorphism is a pair

$$\phi = (\phi_{\text{field}}, \phi_{\text{Boolean}})$$

of mappings

$$\phi_{\text{field}} : F_A \rightarrow F_B$$

and the identity map

$$\phi_{\text{Boolean}} = id_{\text{Boolean}} : B \rightarrow B$$

that preserves the constants and operations of $\Sigma_{\text{Ordered Field}}$ as follows: for all $a, b \in F$,

(i) Constants

$$\phi_{\text{field}}(0_A) = 0_B \quad \text{and} \quad \phi_{\text{field}}(1_A) = 1_B$$

(ii) Addition

$$\phi_{\text{field}}(a + b) = \phi_{\text{field}}(a) + \phi_{\text{field}}(b)$$

(iii) Additive Inverse

$$\phi_{\text{field}}(-a) = -\phi_{\text{field}}(a)$$

(iv) Multiplication

$$\phi_{\text{field}}(a.b) = \phi_{\text{field}}(a).\phi_{\text{field}}(b)$$

(v) Multiplicative Inverse

$$\phi_{\text{field}}(a^{-1}) = (\phi_{\text{field}}(a))^{-1}$$
8.3. UNIQUENESS OF THE REAL NUMBERS

(vi) Order
\[ a < b \text{ implies } \phi_{\text{field}}(a) < \phi_{\text{field}}(b) \]

**Lemma (Injectivity of Order Preserving Maps)** Let \( \phi : A \to B \) be order-preserving, i.e.,
\[ a < b \text{ implies } \phi(a) < \phi(b). \]

Then \( \phi \) is injective.

**Proof** If \( a \neq b \) then either \( a < b \) or \( b < a \), since \( < \) is a total order. Suppose \( a < b \). Then \( \phi(a) < \phi(b) \), since \( \phi \) is order-preserving and \( \phi(a) \neq \phi(b) \). Similarly, if \( b < a \) we can deduce that \( \phi(a) \neq \phi(b) \).

Hence, if \( a \neq b \) then \( \phi(a) \neq \phi(b) \). Thus, \( \phi \) is injective. \( \square \)

The standard models of the real numbers are built from the rational numbers in different ways. In the proof, we will show that any algebra that satisfies the axioms has a similar structural property, and that this enables us to construct an isomorphism between any two algebras that satisfies the axioms. The proof is in three stages, as follows.

Let \( A \) and \( B \) be any \( \Sigma_{\text{Ordered Field}} \)-algebras satisfying the axioms.

**Stage 1: Recovering the rationals in \( A \) and \( B \)**

Using the field \( \mathbb{Q} \) of rational numbers as our model, we construct subalgebras
\[ A_{\text{Rat}} \text{ and } B_{\text{Rat}} \]
of \( A \) and \( B \) using the \( \Sigma_{\text{Ordered Field}} \) operations applied to identity elements \( 1_A \) and \( 1_B \), of \( A \) and \( B \) respectively. Then we define a map
\[ \phi : A_{\text{Rat}} \to B_{\text{Rat}} \]
and prove that \( \phi \) is an ordered field isomorphism between these subalgebras, which are subfields of \( A \) and \( B \).

**Stage 2: Approximating \( A \) and \( B \) by \( A_{\text{Rat}} \) and \( B_{\text{Rat}} \)**

Using the Completeness Property, we show that the elements of \( A \) can be approximated by the elements of \( A_{\text{Rat}} \); and similarly for \( B \) and \( B_{\text{Rat}} \).

**Stage 3: Constructing the isomorphism between \( A \) and \( B \)**

Using this approximation property of Stage 2, and the isomorphism \( \phi : A_{\text{Rat}} \to B_{\text{Rat}} \), we define a map
\[ \Phi : A \to B \]
and prove it is an ordered-field isomorphism.

In particular, we say that \( \Phi \) is a lifting of \( \phi \) from \( A_{\text{Rat}} \) to \( A \) in the sense that, for \( a \in A_{\text{Rat}} \),
\[ \Phi(a) = \phi(a). \]
8.3.2 Stage 1: Constructing The Rational Ordered Subfields

Let $A$ be any $\Sigma_{\text{Ordered Field}}$-algebra satisfying the axioms of an ordered field. We will build a special $\Sigma_{\text{Ordered Field}}$-subalgebra $A_{\text{Rat}}$ of $A$ that also satisfies the axioms and is called the rational subfield of the ordered field $A$.

Let the carriers of $A$ be $F_A$ and $B$ of sorts field and $\text{Bool}$, respectively. The two carriers of $A_{\text{Rat}}$ will be denoted $R_A \subseteq F_A$ and $B$ of sorts field and $\text{Bool}$, respectively.

Now, $A_{\text{Rat}}$ is the subalgebra of $A$ built from the identity elements $0_A$ and $1_A$ by repeatedly applying all the operations to them. For example, starting with $0_A$ and $1_A$ we can build the following elements of $R_A$:

$$\begin{align*}
0_A, 1_A, & 1_A + 1_A, 1_A + 1_A + 1_A, \\
(1_A)^{-1}, & (1_A + 1_A)^{-1}, (1_A + 1_A + 1_A)^{-1}, \\
(1_A)(1_A + 1_A)^{-1}, & (1_A + 1_A)(1_A + 1_A + 1_A)^{-1}, \\
((1_A)(1_A + 1_A + 1_A)^{-1})(1_A + 1_A)(1_A + 1_A + 1_A)^{-1}, & \ldots
\end{align*}$$

To better understand the subset $R_A$ of $F_A$, we introduce some notation that simplifies these expressions.

**Definition (Formal sums)** Let $a \in F_A$ and $n \in \mathbb{Z}$.

If $n > 0$ define

$$na = a + a + \cdots + a \quad (n \text{ times}).$$

If $n = 0$ define

$$0a = 0_A.$$

If $n < 0$ define

$$na = -((-n)a).$$

For example,

$$3a = a + a + a$$

and

$$-3a = -((-(-3)a) = -(3a) = -(a + a + a).$$

Here is a useful set of properties, when we need to calculate using this notation.

**Lemma (1)** For any $a \in F_A$ and any integers $n, m \in \mathbb{Z}$, we have:

(i) $na + ma = (n + m)a$;

(ii) $n(a \cdot b) = (na) \cdot b$;
8.3. **UNIQUENESS OF THE REAL NUMBERS**

(iii) \( n(ma) = (nm)a \); and

(iv) if \( n > 0 \) and \( a > 0_A \) then \( na > 0_A \).

**Proof**  We leave the proof as an exercise. The equations are proved by induction in the case \( n \geq 0 \), and the field and ordering axioms are used to complete the argument in the case \( n < 0 \). \( \square \)

In building \( R_A \), we use this sum notation \( n_A \) in the case \( a = 1_A \).

**Definition (Formal Integers)** Let \( 1_A \in F_A \) be the identity and \( n \in \mathbb{Z} \). Then the element \( n1_A \in F_A \) is called a **formal integer**.

**Lemma (2)** For any \( a \in F_A \) and any integers \( n, m \in \mathbb{Z} \), we have:

(i) \( (n1_A)(m1_A) = (nm)1_A \); and

(ii) \( n1_A = (m1_A) \) if, and only if, \( n = m \).

**Proof**  Equation (i) is proved using (ii) and (iii) of Lemma 1 as follows:

\[
(n1_A)(m1_A) = n(1_A(m1_A)) \quad \text{by Equation (ii)};
\]

\[
= n(m1_A) \quad \text{by identity axiom};
\]

\[
= (nm)1_A \quad \text{by Equation (iii)}.
\]

Property (ii) uses Lemma 1(iv) and the axioms. \( \square \)

Next, we extend the notation to include division.

**Definition (Formal Rationals)** Let \( 1_A \in F_A \) be the identity and \( m, n \in \mathbb{Z} \). If \( n > 0 \) then define

\[
\frac{m1_A}{n1_A} = (m1_A)(n1_A)^{-1}
\]

which exists because \( n1_A \neq 0_A \) in an ordered field.

So

\[
\frac{m1_A}{n1_A} = \frac{1_A + 1_A + \cdots + 1_A \ (m \ times)}{1_A + 1_A + \cdots + 1_A \ (n \ times)}
\]

These elements we call **formal rationals**.

The set \( R_A \) of elements of \( F_A \) can now be defined by

\[
R_A = \left\{ \frac{m1_A}{n1_A} \mid m, n \in \mathbb{Z} \text{ and } n > 0 \right\}.
\]

The requirement that \( n > 0 \) is to ensure that the formal rationals have a standard representation.

We first consider order and equality for formal rationals.

**Lemma (3)** For any \( m, n, p, q \in \mathbb{Z} \) with \( n > 0 \) and \( q > 0 \), we have

(i) \( \frac{m1_A}{n1_A} < \frac{p1_A}{q1_A} \) in \( F_A \) if, and only if, \( mq < np \) in \( \mathbb{Z} \).

(ii) \( \frac{m1_A}{n1_A} = \frac{p1_A}{q1_A} \) in \( F_A \) if, and only if, \( mq = np \) in \( \mathbb{Z} \).
**Proof** We prove condition (i). By definition,

\[
\frac{m1_A}{n1_A} < \frac{p1_A}{q1_A} \iff (m1_A)(n1_A)^{-1} < (p1_A)(q1_A)^{-1}.
\]

Since \(n1_A > 0_A\) and \(q1_A > 0_A\), we can multiply both sides by both elements, to get

\[
\iff (m1_A)(q1_A) < (p1_A)(n1_A).
\]

By Lemma 2(i),

\[
\iff (mq)1_A < (np)1_A.
\]

Rearranging, by Lemma 2 in Section 8.3.2

\[
\iff 0_A < (np)1_A - (mq)1_A.
\]

By Lemma 1(i),

\[
\iff 0_A < (np - mq)1_A
\]

\[
\iff 0 < np - mq
\]

\[
\iff mq < np.
\]

\[\square\]

It is easy to deduce from Lemma 3 that

**Lemma (4)** For any \(m, n \in \mathbb{Z}\) with \(n > 0\) we have

(i) \[\frac{m1_A}{n1_A} = 0_A \text{ if, and only if, } m = 0.\]

(ii) \[\frac{m1_A}{n1_A} = 1_A \text{ if, and only if, } m = n.\]

**Proof** Consider (i). Take \(p = 0, q = 1\) in Lemma 3(ii), we have

\[
\frac{m1_A}{n1_A} = 0_A \iff m.1 = n.0
\]

\[
\iff m = 0.
\]

Consider (ii). Take \(p = 1, q = 1\) in Lemma 3(ii), we have

\[
\frac{m1_A}{n1_A} = 1_A \iff m.1 = n.1
\]

\[
\iff m = n.
\]

\[\square\]

Next, we consider the operations on formal rationals.

**Lemma (5)** For any \(m, n, p, q \in \mathbb{Z}\) with \(n > 0\) and \(q > 0\), we have

(i) \[\frac{m1_A}{n1_A} + \frac{p1_A}{q1_A} = \frac{(mq + np)1_A}{nq1_A};\]
8.3. UNIQUENESS OF THE REAL NUMBERS

(ii) \[ \frac{m_1 A}{n_1 A} \cdot \frac{p_1 A}{q_1 A} = \frac{mp_1 A}{nq_1 A}; \]

(iii) \[ \frac{m_1 A}{n_1 A} = \frac{(-m)_1 A}{n_1 A}; \]

(iv) If \( m > 0 \) then

\[
\left( \frac{m_1 A}{n_1 A} \right)^{-1} = \frac{n_1 A}{m_1 A};
\]

If \( m < 0 \) then

\[
\left( \frac{m_1 A}{n_1 A} \right)^{-1} = \frac{(-n)_1 A}{(-m)_1 A}.
\]

**Proof** We use the various lemmas and axioms to calculate these identities.

(i) Addition

By definition,

\[ \frac{m_1 A}{n_1 A} + \frac{p_1 A}{q_1 A} = \frac{(m_1 A).n_1 A}{(n_1 A)} + \frac{(p_1 A).q_1 A}{(q_1 A)}^{-1}. \]

Now multiplying the first and second terms by

\[ 1_A = (q_1 A).q_1 A^{-1} \quad \text{and} \quad 1_A = (n_1 A).n_1 A^{-1} \]

respectively, and rearranging using the field axioms, we get

\[
= \left[ (m_1 A).(q_1 A) \right] \cdot \left[ (n_1 A).(q_1 A)^{-1} \right] + \left[ (n_1 A).(p_1 A) \right] \cdot \left[ (n_1 A).(q_1 A) \right]^{-1}.
\]

By Lemma 2(i),

\[
= \left[ (mq)_1 A \right] \cdot \left[ (nq)_1 A^{-1} \right] + \left[ (np)_1 A \right] \cdot \left[ (nq)_1 A \right]^{-1}.
\]

By distribution law,

\[
= \left[ (mq)_1 A + (np)_1 A \right] \cdot \left[ (nq)_1 A^{-1} \right].
\]

By Lemma 1(i),

\[
= \left[ (mq + np)_1 A \right] \cdot \left[ (nq)_1 A^{-1} \right].
\]

Therefore, by definition,

\[
= \frac{(mq + np)_1 A}{(nq)_1 A}.
\]

(ii) Multiplication

By definition,

\[
\left( \frac{m_1 A}{n_1 A} \right) \left( \frac{p_1 A}{q_1 A} \right) = \left( (m_1 A).(n_1 A)^{-1} \right) \cdot \left( (p_1 A).(q_1 A)^{-1} \right).
\]

By commutativity and inverse properties,

\[
= \left( (m_1 A).(p_1 A) \right) \cdot \left( (n_1 A).(q_1 A) \right)^{-1}.
\]

By Lemma 2(i),

\[
= \left( (mp)_1 A \right) \cdot \left( (nq)_1 A \right)^{-1}.
\]

Therefore, by definition,

\[
= \frac{(mp)_1 A}{(nq)_1 A}.
\]
Additive inverse.

By definition,

$$\left( \frac{m1_A}{n1_A} \right) + \left( -\frac{m1_A}{n1_A} \right) = \frac{m1_A}{n1_A} + \frac{(-m)1_A}{n1_A}$$

By case (i) of this Lemma,

$$= \frac{(mn - nn)1_A}{nn1_A}$$

$$= \frac{01_A}{nn1_A}$$

By Lemma 4(i),

$$= 0_A.$$

The two cases for multiplicative inverse we leave as exercises.

\(\Box\)

**Theorem (1)** Let \(A\) be any \(\Sigma_{\text{Ordered Field}}\)-algebra that is an ordered field with Booleans. Then \(A_{\text{Rat}}\) with carriers \(R_A\) and \(B\) constitutes a \(\Sigma_{\text{Ordered Field}}\)-subalgebra of \(A\) that is also an ordered field with Booleans.

**Proof** First, we show that \(R_A\) is closed under the operations of \(\Sigma_{\text{Ordered Field}}\). Clearly, \(B\) is closed under the operations and and not.

The constants 0\(_A\) and 1\(_A\) are in \(R_A\) by Lemma 4. \(R_A\) is closed under the operations of +, -, \(-1\) by Lemma 5(i)-(iv), respectively.

Obviously, \(A_{\text{Rat}}\) and \(B\) are closed under \(<\) since both truth values are in \(B\).

Now all the axioms of an ordered field are true of all elements of \(F_A\). Thus, they are true for all elements of the subfield \(R_A\) of \(F_A\). For example, since

$$x + y = y + x$$

holds for all of \(F_A\), it must hold for all of the subset of \(R_A\).  \(\Box\)

**Theorem (2)** Let \(A\) and \(B\) be \(\Sigma_{\text{Ordered Field}}\)-algebras that are ordered fields with Booleans. Let \(A_{\text{Rat}}\) and \(B_{\text{Rat}}\) be the \(\Sigma_{\text{Ordered Field}}\)-subalgebras of rational elements that are ordered subfields of \(A\) and \(B\), respectively. Then the map

$$\phi = (\phi_{\text{field}}, \phi_{\text{Bool}}) : A_{\text{Rat}} \rightarrow B_{\text{Rat}}$$

that is defined on field elements by \(\phi_{\text{field}} : R_A \rightarrow R_B\), where

$$\phi_{\text{field}}(\frac{m1_A}{n1_A}) = \frac{m1_B}{n1_B}$$

and by the identity function \(\phi_{\text{Bool}} = \text{id}_{\text{Bool}} : B \rightarrow B\), is a \(\Sigma_{\text{Ordered Field}}\)-isomorphism.

**Proof** We have to prove that \(\phi\) preserves all the operations in \(\Sigma_{\text{Ordered Field}}\) and is a bijection.
8.3. UNIQUENESS OF THE REAL NUMBERS

(i) Addition:

\[ \phi_{\text{field}} \left( \frac{m1_A}{n1_A}, \frac{p1_A}{q1_A} \right) = \phi_{\text{field}} \left( \frac{(mq + pn)}{nq1_B} \right) \]

by Lemma 5(i);

\[ = \frac{(mq + pn)1_B}{nq1_B} \]

by definition of \( \phi_{\text{field}} \);

\[ = \frac{m1_B}{n1_B} + \frac{p1_B}{q1_B} \]

by Lemma 5(ii);

\[ = \phi_{\text{field}} \left( \frac{m1_A}{n1_A} \right) + \phi_{\text{field}} \left( \frac{p1_A}{q1_A} \right) \]

by definition of \( \phi_{\text{field}} \).

(ii) Multiplication:

\[ \phi_{\text{field}} \left( \frac{m1_A}{n1_A}, \frac{p1_A}{q1_A} \right) = \phi_{\text{field}} \left( \frac{mp1_A}{nq1_A} \right) \]

by Lemma 5(ii);

\[ = \frac{mp1_B}{nq1_B} \]

by definition of \( \phi_{\text{field}} \);

\[ = \frac{m1_B}{n1_B} \cdot \frac{p1_B}{q1_B} \]

by Lemma 5(ii);

\[ = \phi_{\text{field}} \left( \frac{m1_A}{n1_A} \right) \cdot \phi_{\text{field}} \left( \frac{p1_A}{q1_A} \right) \]

by definition of \( \phi_{\text{field}} \).

The constants, and additive and multiplicative inverses, are easy exercises.

Consider the order relation. By Lemma 3,

\[ \frac{m1_A}{n1_A} < \frac{p1_A}{q1_A} \iff mq < np \]

applying Lemma 3 to \( A \);

\[ \iff \frac{m1_B}{n1_B} < \frac{p1_B}{q1_B} \]

applying Lemma 3 to \( B \);

\[ \iff \phi_{\text{field}} \left( \frac{m1_A}{n1_A} \right) < \phi_{\text{field}} \left( \frac{p1_A}{q1_A} \right) \]

by definition of \( \phi_{\text{field}} \).

This shows that \( \phi_{\text{field}} \) preserves the ordering.

Furthermore, we see that \( \phi_{\text{field}} \) is injective since it is order-preserving (Lemma of Section 8.3.1). Clearly, \( \phi_{\text{field}} \) is surjective by the definitions of \( R_B \) and \( \phi_{\text{field}} \). Since \( \phi_{\text{Bool}} \) is the identity, it is a bijection. Hence, \( \phi = (\phi_{\text{field}}, \phi_{\text{Bool}}) \) is a \( \Sigma_{\text{Ordered Field}} \)-isomorphism. \( \square \)

### 8.3.3 Stage 2: Approximation by the Rational Ordered Subfield

We will now show that all the elements of \( A \) can be approximated by the elements of \( A_{\text{Rat}} \), i.e., the elements of \( F_A \) can be approximated by the elements of \( R_A \). The approximation property we will prove is that

\[ R_A \text{ is dense in } F_A, \]

which means that
for any $a, b \in F_A$ with $a < b$, there exists some $r \in R_A$ such that $a < r < b$.

**Lemma (Archimedes’ Property)** Let $A$ satisfy the complete ordered field axioms. Let $a, b \in F_A$ and $a > 0_A$. Then there exists $m \in \mathbb{Z}$, $m > 0$ such that

$$ma > b.$$ 

**Proof** Suppose, for a contradiction, the property does not hold. Then, there is an $a > 0_A$ such that for all $m \in \mathbb{Z}$, $m > 0$,

$$ma < b.$$ 

Clearly,

$$S = \{ma \mid m \in \mathbb{Z} \text{ and } m > 0\}$$

is bounded above by $b \in F_A$. Since $A$ is complete, the set $S$ has a least upper bound $c$ and $c \leq b$. For all $m > 0$,

$$(m + 1)a \in S$$

and

$$(m + 1)a \leq c.$$ 

Thus, for all $m > 0$,

$$ma \leq c - a.$$ 

This implies that $c - a$ is also an upper bound of $S$ and, since $a > 0_A$, we have that

$$c - a < c.$$ 

This contradicts that $c$ is the least upper bound, so the property holds. □

**Corollary** For all $a \in F_A$ and $n \in \mathbb{Z}$ with $n > 0$, there exists $m \in \mathbb{Z}$ such that

$$a < \frac{m1_A}{n1_A}.$$ 

**Proof** Let $b = \frac{11_A}{n1_A}$. By Archimedes’ Property, since $b > 0_A$ there exists $m \in \mathbb{Z}$ such that

$$a < mb.$$ 

Now

$$mb = m[11_A] \cdot (n1_A)^{-1} \text{ by definition of } b;$$

$$= (m1_A) \cdot (n1_A)^{-1} \text{ by Lemma 1(ii);}$$

$$= \frac{(m1_A)}{(n1_A)} \text{ by Lemma 1(iii);}$$

$$= \frac{m1_A}{n1_A} \text{ by definition.}$$

□

**Theorem (Density)** Let $A$ be a complete ordered field with Booleans, and $A_{\text{Rat}}$ be its subfield of formal rationals with Booleans. For any $a, b \in F_A$ with $a < b$, there exists $r \in R_A$ such that

$$a < r < b.$$
8.3. **UNIQUENESS OF THE REAL NUMBERS**

**Proof** Now \( b - a > 0 \) so, by Archimedes’ Property, there exists an \( n > 0 \) such that

\[ n(b - a) > 1_A. \]

Hence, multiplying by \((n1_A)^{-1}\), we have that

\[ b - a > \frac{11_A}{n1_A}. \]

Let

\[ S = \{ m \mid b < \frac{m1_A}{n1_A} \}. \]

By the Corollary to Archimedes’ Property, \( S \neq \emptyset \). Now, again by the Corollary, \( S \) is bounded below because there is some \( k \) such that

\[ -b < \frac{k1_A}{n1_A} \]

and, multiplying by \(-1\), for all \( m \in S \),

\[ \frac{(-k)1_A}{n1_A} < b < \frac{m1_A}{n1_A}. \]

By Lemma 3(i) of Section 8.3.2,

\[ -k < m \]

for all \( m \in S \). Since \( S \) is a set of integers that is bounded below, it has a minimum \( m_0 + 1 \). Thus,

\[ \frac{m_01_A}{n1_A} < b < \frac{(m_0 + 1)1_A}{n1_A} \]

since \( m_0 < m_0 + 1 \) and \( m_0 \notin S \). Now

\[ a < \frac{m_01_A}{n1_A} \]

follows from \((*)\), for rearranging

\[ b - \frac{11_A}{n1_A} > a \]

so

\[ \frac{(m_0 + 1)1_A}{n1_A} - \frac{11_A}{n1_A} = \frac{m_01_A}{n1_A} > a. \]

Thus, the required element \( r \in R_A \) is \( \frac{m_01_A}{n1_A} \).

\[ \square \]

**Corollary** For any \( a \in F_A \), there exists \( r, r' \in R_A \) such that

\[ r < a < r'. \]

**Proof** Now \( a - 1_A < a < a + 1_A \), and applying the Density Theorem twice produces

\[ a - 1_A < r < a \quad \text{and} \quad a < r' < a + 1_A, \]

as required.

\[ \square \]
Definition (Upper and lower cuts) In a complete ordered field $A$, we can define the lower cut for $a \in F_A$,

$$L_a = \{ r \in R_A \mid r < a \}$$

and the upper cut for $a \in F_A$,

$$U_a = \{ r \in R_A \mid a \leq r \}.$$

Theorem (Cut Equivalence) For every $a \in F_A$,

$$a = \sup L_a = \inf U_a$$

Proof Now $L_a$ is not empty by the Corollary. Since $L_a$ is bounded above by $a$, $\sup L_a$ exists and

$$\sup L_a \leq a.$$ 

Suppose, for a contradiction, that

$$\sup L_a < a.$$ 

Then, by the Density Theorem, there exist $r \in R_A$ such that

$$\sup L_a < r < a.$$ 

This means that $r \in L_a$ but $r$ is bigger than the supremum, which is a contradiction. Thus,

$$\sup L_a = a.$$ 

The argument for $a = \inf U_a$ is similar. 

8.3.4 Stage 3: Constructing the Isomorphism

We know that if $A$ and $B$ are $\Sigma_{\text{Ordered Field}}$ algebras that are complete ordered fields with Booleans, then $A_{\text{Rat}}$ and $B_{\text{Rat}}$ are subalgebras of $A$ and $B$, respectively, and that they are:

(i) ordered fields with Booleans (by Theorem 1);

(ii) $\Sigma_{\text{Ordered Field}}$-isomorphic under $\phi : A_{\text{Rat}} \to B_{\text{Rat}}$ (by Theorem 2); and

(iii) dense in $A$ and $B$, respectively (by Density Theorem).

We will prove that $\phi$ can be lifted to a $\Sigma_{\text{Ordered Field}}$-isomorphism $\Phi : A \to B$, i.e., for $r \in A_{\text{Rat}}$, $\Phi(r) = \phi(r)$.

Theorem (4) There is a $\Sigma_{\text{Ordered Field}}$ isomorphism

$$\Phi = (\Phi, id) : A \to B.$$ 

Proof First, we lift $\phi$ to a mapping $\hat{\phi}$ between all subsets of $R_A$ and $R_B$. Let $\mathcal{P}(R_A)$ and $\mathcal{P}(R_B)$ be the power sets of $R_A$ and $R_B$, respectively. For any $S \in \mathcal{P}(R_A)$, we define

$$\hat{\phi}(S) = \{ \phi(r) \mid r \in S \}.$$ 

Clearly, $\hat{\phi}(S) \in \mathcal{P}(R_B)$. 

8.3. **UNIQUENESS OF THE REAL NUMBERS**

Lemma

\[ \hat{\phi} : \mathcal{P}(R_A) \to \mathcal{P}(R_B) \]

is a bijection.

**Proof** Exercise. \( \square \)

Now, for any \( a \in F_A \), let \( L_a \) and \( U_a \) be the lower and upper cuts of \( a \). So by the Cut Equivalence Theorem in Section 8.3.3,

\[ a = \sup L_a = \inf U_a. \]

Clearly, every element of \( \hat{\phi}(L_a) \) is less than or equal to every element of \( \hat{\phi}(U_a) \), since \( \phi \) is order-preserving, and so we can deduce that

\[ \sup \hat{\phi}(L_a) \leq \inf \hat{\phi}(U_a). \]

Lemma

\[ \sup \hat{\phi}(L_a) = \inf \hat{\phi}(U_a). \]

**Proof** For a contradiction, suppose that

\[ \sup \hat{\phi}(L_a) < \inf \hat{\phi}(U_a). \]

By applying the Density Theorem twice, we know that there exist \( \phi(r_1) \) and \( \phi(r_2) \) in \( R_B \) such that

\[ \sup \hat{\phi}(L_a) < \phi(r_1) < \phi(r_2) < \inf \hat{\phi}(U_a). \]

For any \( s \in L_a \), \( \phi(s) \in \hat{\phi}(L_a) \), so \( \phi(s) < \phi(r_1) \) and, hence, \( s < r_1 \). It follows that

\[ a = \sup L_a \leq r_1. \]

Similarly,

\[ r_2 \leq \inf U_a = a. \]

Combining, we get \( r_1 = r_2 \), which contradicts the fact that \( \phi(r_1) < \phi(r_2) \) (since \( \phi \) is order-preserving). Hence, the lemma holds. \( \square \)

Here is the definition of the extension \( \Phi \) of \( \phi \). For any \( a \in F_A \),

\[ \Phi(a) = \sup \hat{\phi}(L_a) = \inf \hat{\phi}(U_a). \]

We will now check that \( \Phi \) has the necessary ordering and algebraic properties.

Lemma \( \Phi \) is an order-preserving bijection.

**Proof** Suppose \( a, b \in F_A \) and \( a < b \). First, we show that \( \Phi(a) < \Phi(b) \).

Let \( r_1, r_2 \in R_A \) satisfy

\[ a < r_1 < r_2 < b. \]

Then \( r_1 \in U_a \) and \( \phi(r_1) \in \hat{\phi}(U_a) \) and, hence,

\[ \Phi(a) = \inf \hat{\phi}(U_a) \leq \phi(r_1). \]
Similarly, \( r_2 \in L_b \) and \( \Phi(r_2) \in \hat{\Phi}(L_b) \) and

\[
\Phi(b) = \sup \hat{\Phi}(L_b) \geq \phi(r_2).
\]

Together we have

\[
\Phi(a) \leq \phi(r_1) < \phi(r_2) \leq \Phi(b)
\]

since \( \phi \) is order-preserving.

Next, we prove that \( \Phi \) is a bijection. Since \( \Phi \) is order-preserving, it is injective. We show that the range of \( \Phi \) is \( F_B \).

Let \( c \in F_B \) and define

\[
T = \{ a \mid a \in F_A \text{ and } \phi(a) \geq c \}.
\]

By the Corollary to the Density Theorem, there are \( \phi(r_1) \) and \( \phi(r_2) \) in \( R_B \) with

\[
\phi(r_1) < c < \phi(r_2).
\]

Hence, \( r_2 \in T \) and since \( r_1 \) is a lower bound of \( c \), we know an infimum of \( T \) exists. Let this be \( b \).

Suppose \( \Phi(b) < c \). Then some \( \phi(r) \in R_B \) exists with

\[
\Phi(b) < \phi(r) < c.
\]

This implies \( r \) is a lower bound of \( T \) and that \( b < r \), which is not possible.

Now suppose \( c < \Phi(b) \) so that for some \( \phi(s) \in R_B \),

\[
c < \phi(s) < \Phi(b).
\]

Then \( s \in T \) and \( s < b \) which is not possible.

Therefore, \( \Phi(b) = c \) and the range of \( \Phi \) is \( F_B \). \( \square \)

**Lemma** For all \( a, b \in F_A \),

(i) \( \Phi(a + b) = \Phi(a) + \Phi(b) \);

(ii) \( \Phi(-a) = -\Phi(a) \); and

(iii) \( \Phi(ab) = \Phi(a) \Phi(b) \).

**Proof** (i) Addition

Given any \( a, b \in F_A \), for any \( r, s \in R_A \) with \( r < a \) and \( s < b \), we have \( r + s \leq a + b \) (by the ordered field axioms). Applying the order-preserving map \( \Phi \), we get

\[
\Phi(r + s) \leq \Phi(a + b)
\]

and

\[
\phi(r + s) \leq \Phi(a + b)
\]

since \( \Phi \) is \( \phi \) on \( R_A \). Now \( \phi : R_A \to R_B \) is a \( \Sigma_{\text{Ordered Field}} \)-homomorphism, so

\[
\phi(r + s) = \phi(r) + \phi(s).
\]
Substituting \( \phi(r) + \phi(s) \leq \Phi(a + b) \)

and rearranging,

\[ \phi(r) \leq \Phi(a + b) - \phi(s) \]

for all \( r \in L_a \). This implies that

\[ \Phi(a) = \sup \phi(L_a) \leq \Phi(a + b) - \phi(s). \]

Rearranging,

\[ \phi(s) \leq \Phi(a + b) - \Phi(a) \]

for all \( s < b \). Hence,

\[ \Phi(b) = \sup \phi(L_b) \leq \Phi(a + b) - \Phi(a). \]

Hence, rearranging

\[ \Phi(a) + \Phi(b) \leq \Phi(a + b). \]

By a similar argument, using \( \inf \phi(U_a) \) and \( \inf \phi(U_b) \), we can deduce that

\[ \Phi(a + b) \leq \Phi(a) + \Phi(b). \]

Hence, we deduce they are equal and (i) is true.

(ii) Additive Inverse

Clearly, because of (i),

\[ \Phi(a) + \Phi(-a) = \Phi(a - a) = \Phi(0_A) = 0_B. \]

So

\[ \Phi(-a) = -\Phi(a). \]

(iii) Multiplication

Let \( a, b \in F_A \) and suppose \( a, b > 0_A \). For any \( r, s \in R_A \) with

\[ 0_A < r \leq a \quad \text{and} \quad 0_A < s \leq b \]

we have that \( rs \leq ab \) and that

\[ \Phi(r.s) \leq \Phi(a.b) \]

since \( \Phi \) is order-preserving. Now \( \Phi = \phi \) on \( R_A \) and \( \phi \) is a homomorphism so

\[ \Phi(r.s) = \phi(r.s) = \phi(r).\phi(s) = \Phi(r).\Phi(s), \]

and we deduce that

\[ \Phi(a).\Phi(b) \leq \Phi(a.b). \]

8.3. UNIQUENESS OF THE REAL NUMBERS
Similarly, we can obtain
\[ \Phi(a, b) \leq \Phi(a) \cdot \Phi(b) \]
and hence
\[ \Phi(a) \cdot \Phi(b) = \Phi(a, b) \]
for \( a, b > 0 \).
Suppose \( a = 0 \) or \( b = 0 \). Then trivially,
\[
\Phi(a, b) = \Phi(0, 0) = 0 = \Phi(a) \cdot \Phi(b).
\]

If \( a < 0, b > 0 \), then we argue as follows:
\[
\Phi(a, b) = -\Phi(-a, b) = -(\Phi(-a) \cdot \Phi(b)) = \Phi(a) \cdot \Phi(b).
\]

The cases of \( a > 0, b < 0 \) and \( a < 0, b < 0 \) are treated similarly.

This completes the proof of the lemma and the proof of the Uniqueness Theorem.

\[ \square \]

8.4 Cantor’s Construction of the Real Numbers

We will build a representation of the data type
\[ (\mathbb{R}; 0, 1; +, -) \]
of real numbers using Cauchy sequences of rational numbers. Now the set \( \text{CSeq} \) of Cauchy sequences contains infinitely many representations of the same point, and this fact will require constant attention, for when we operate on different representations of the same real number, we need to ensure that the results do not represent different answers, i.e., operations on representations are “well-defined”, or extend to what they represent. We begin with equality.

8.4.1 Equivalence of Cauchy Sequences

**Definition (Equality of Representations of Reals)** Consider two Cauchy sequences
\[
a = a_1, a_2, \ldots, a_n, \ldots \quad \text{and} \quad b = b_1, b_2, \ldots, b_n, \ldots
\]
of rationals. We say that the sequences \( a \) and \( b \) are \textit{equivalent} if for any rational number \( \epsilon > 0 \), there is a natural number \( n_0 \), such that for all \( n > n_0 \), we have \( |a_n - b_n| < \epsilon \). Let \( a \equiv b \) denote the equivalence of Cauchy sequences \( a \) and \( b \).
8.4. CANTOR’S CONSTRUCTION OF THE REAL NUMBERS

Two Cauchy sequences are equivalent if, for any degree of accuracy \( \epsilon > 0 \), there is some point \( n_0 \), after which the elements of the sequence are within \( \epsilon \) of each other.

For example, the sequence

\[
1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, \frac{1}{n}, \ldots
\]

is equivalent with

\[
0, 0, 0, 0, \ldots,
\]

because for any number \( \epsilon = \frac{a}{\beta} > 0 \), there is a natural number \( n_0 = \lfloor 1 + \frac{\beta}{a} \rfloor \) such that \( |\frac{1}{n} - 0| < \epsilon \) for all \( n > n_0 \).

**Lemma** The relation \( \equiv \) is an equivalence relation on \( CSeq \).

**Proof** We will show that \( \equiv \) satisfies the three necessary properties.

Let

\[
a = a_1, a_2, \ldots, \quad b = b_1, b_2, \ldots \quad \text{and} \quad c = c_1, c_2, \ldots
\]

be any three Cauchy sequences.

(i) \( \equiv \) is reflexive.

Clearly, for any \( \epsilon > 0 \) and all \( n = 1, 2, \ldots \)

\[
|a_n - a_n| = |0| = 0 < \epsilon
\]

and so \( a \equiv a \).

(ii) \( \equiv \) is symmetric.

Suppose \( a \equiv b \). Then, by definition, for any \( \epsilon > 0 \), there exists a stage \( n_0 \), such that for all later stages \( n > n_0 \),

\[
|a_n - b_n| < \epsilon.
\]

Clearly, we have that for any \( \epsilon > 0 \), there exists \( n_0 \) such that for all \( n > n_0 \),

\[
|b_n - a_n| = |a_n - b_n| < \epsilon
\]

and, by definition, \( b \equiv a \).

(iii) \( \equiv \) is transitive.

Suppose \( a \equiv b \) and \( b \equiv c \). Then, by definition, for any \( \epsilon > 0 \), there exists a stage \( n_0 \), such that for all later stages \( n > n_0 \),

\[
|a_n - b_n| < \frac{\epsilon}{2} \quad \text{and} \quad |b_n - c_n| < \frac{\epsilon}{2}.
\]

For such \( n \), adding the inequalities, we deduce that

\[
|a_n - c_n| \leq |a_n - b_n| + |b_n - c_n| < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon
\]

and, so by definition, \( a \equiv c \).
Lemma  Let \( a = a_1, a_2, \ldots \) be a Cauchy sequence satisfying the following condition:

\[
\text{For any } \delta > 0 \text{ and any } n, \text{ there exists some } m > n \text{ such that } |a_m| < \delta. \tag{*}
\]

Then \( a \equiv 0 \), the zero Cauchy sequence.

Proof Since \( a \) is a Cauchy sequence, for any \( \delta > 0 \), there is an \( n_0 \), such that for all \( m, n > n_0 \),

\[
|a_n - a_m| < \frac{\delta}{2}.
\]

If \( a \) satisfies the condition \( * \), then given \( n_0 \), there exists some \( m > n_0 \), such that

\[
|a_m| < \frac{\delta}{2}.
\]

To prove the Cauchy sequence is equivalent to 0, we calculate for any \( n > m \),

\[
|a_n - 0| = |a_n| = |a_n - a_m + a_m| \\
\leq |a_n - a_m| + |a_m| \\
< \frac{\delta}{2} + \frac{\delta}{2} \\
< \delta
\]

Hence, \( a \equiv 0 \). \qed

Corollary  Let \( a = a_1, a_2, \ldots \) be a Cauchy sequence such that \( a \not\equiv 0 \). Then there exists some \( \delta > 0 \) and \( k \) such that for all \( n > k \),

\[
|a_n| \geq \delta.
\]

Each real number has a unique representation by means of an equivalence class of Cauchy sequences.

Definition  A Cantor real number is an equivalence class of Cauchy sequences under the equivalence relation \( \equiv \). If \( a = a_1, a_2, \ldots \) is a Cauchy sequence, then the equivalence class

\[
[a] = \{b \in CSeq \mid b \equiv a\}
\]

is the Cantor real number it represents. The set of Cantor real numbers is the set

\[
R_{\text{Cantor}} = CSeq/\equiv = \{[a] \mid a \in CSeq\}\]

We will build the data type of real numbers from the data type of Cauchy sequences. We will define operations on Cauchy sequences that “lift” to equivalence classes, i.e., to Cantor real numbers. Typically, to define, say, a binary operation, we define a function \( f \) on Cauchy sequences and check the following two conditions apply:

Operations on Representations  The operation \( f \) maps Cauchy sequences to Cauchy sequences, i.e.,

\[
a, b \in CSeq \implies f(a, b) \in CSeq.
\]
Lifting Operations and Congruences The operation \( f \) preserves equivalent Cauchy sequences, i.e.,
\[
a \equiv a' \quad \text{and} \quad b \equiv b' \quad \text{implies} \quad f(a, b) \equiv f(a', b').
\]
We say that \( \equiv \) is congruence with respect to operation \( f \).
We begin with the first task.

8.4.2 Algebra of Cauchy Sequences

We begin by equipping \( CSeq \) with operations to form the algebra
\[
(CSeq; 0, 1; +, -, \cdot, ^{-1}, <).
\]
The operations on Cauchy sequences are pointwise extensions of the corresponding operations on the rationals, although some care is needed in the case of the multiplicative inverse.

First, we give an important property of Cauchy sequences we often use.

Lemma (Boundedness) Each Cauchy sequence \( a = a_1, a_2, \ldots \) is bounded in the following sense: there is an \( M \in \mathbb{Q} \) such that
\[
|a_n| \leq M.
\]

Proof Given a Cauchy sequence \( a = a_1, a_2, \ldots \), we choose \( \epsilon = 1 \) and let \( n_0 \) be such that for all \( n, m > n_0 \),
\[
|a_n - a_m| < 1.
\]
For any \( n > n_0 + 1 \), we have
\[
|a_n| = |a_{n_0+1} + a_n - a_{n_0+1}|
\leq |a_{n_0+1}| + |a_n - a_{n_0+1}|
< |a_{n_0+1}| + 1.
\]
Take \( M \) to be the maximum of
\[
|a_1|, |a_2|, \ldots, |a_{n_0}|, |a_{n_0+1}| + 1
\]
then
\[
|a_n| < M
\]
for all \( n = 1, 2, \ldots \). \( \square \)

Definition (Addition) Let
\[
a = a_1, a_2, \ldots, a_n, \ldots \quad \text{and} \quad b = b_1, b_2, \ldots, b_n, \ldots
\]
be sequences of rationals. We add two sequences of rational numbers by applying the addition operation of rational numbers in a pointwise fashion: we define the sum \( a + b \) of these sequences to be the sequence
\[
a + b = a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n, \ldots
\]
of rationals.

Lemma (Addition) If \( a \) and \( b \) are Cauchy sequences then \( a + b \) is a Cauchy sequence.
\textbf{Proof} If \(a\) and \(b\) are Cauchy sequences then, by the Boundedness Lemma, we know they are bounded by rationals \(M_a > 1\) and \(M_b > 1\). Take
\[
M = \max(M_a, M_b).
\]
Then we have that for all \(n\),
\[
|a_n| < M \quad \text{and} \quad |b_n| < M.
\]
Since they are Cauchy sequences, for any \(\epsilon > 0\), there are \(n_0(a)\) and \(n_0(b)\), such that for all \(m, n > n_0 = \max(n_0(a), n_0(b))\), we have
\[
|a_n - a_m| < \frac{\epsilon}{2M} \quad \text{and} \quad |b_n - b_m| < \frac{\epsilon}{2M}.
\]
For such \(m, n\) it follows that
\[
\left| (a_n + b_n) - (a_m + b_m) \right| \leq |a_n - a_m| + |b_n - b_m| < \frac{\epsilon}{2M} + \frac{\epsilon}{2M} = \frac{\epsilon}{M} < \epsilon.
\]
\[
\square
\]
\textbf{Definition (Multiplication)} Let \[a = a_1, a_2, \ldots, a_n, \ldots \quad \text{and} \quad b = b_1, b_2, \ldots, b_n, \ldots \]
be sequences of rationals. We multiply two sequences of rational numbers by applying the multiplication operation of rational numbers in a pointwise fashion: we define the product \(a \cdot b\) of these sequences to be the sequence
\[a \cdot b = a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n, \ldots\]
of rationals.  

\textbf{Lemma (Multiplication)} If \(a\) and \(b\) are Cauchy sequences then \(a \cdot b\) is a Cauchy sequence.  

\textbf{Proof} If \(a\) and \(b\) are Cauchy sequences then, by the Boundedness Lemma, we can find a common bound \(M\) such that
\[
|a_n| \leq M \quad \text{and} \quad |b_n| \leq M
\]
for all \(n = 1, 2, \ldots\). And since they are Cauchy sequences, for any \(\epsilon > 0\), there is a common stage \(n_0\) such that for \(m, n > n_0\), we have
\[
|a_n - a_m| < \frac{\epsilon}{2M} \quad \text{and} \quad |b_n - b_m| < \frac{\epsilon}{2M}.
\]
Then, for such \(m, n\) it follows that
\[
|a_n b_n - a_m b_m| = |a_n (b_n - b_m) + b_m (a_n - a_m)| \leq |a_n||b_n - b_m| + |b_m||a_n - a_m| < M \frac{\epsilon}{2M} + M \frac{\epsilon}{2M} = \frac{\epsilon}{2} + \frac{\epsilon}{2} < \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon
\]
\[
\square
\]
Definition (Subtraction) Let
\[ a = a_1, a_2, \ldots, a_n, \ldots \text{ and } b = b_1, b_2, \ldots, b_n, \ldots \]
be sequences of rationals. We define the additive inverse of a sequence of rational numbers by
\[ -a = -a_1, -a_2, \ldots, -a_n, \ldots \]
We define the subtraction \(a - b\) of the two sequences to be the sequence
\[ a - b = a + (-b) = a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n, \ldots \]
of rationals.

Lemma (Subtraction) If \(a\) and \(b\) are Cauchy sequences then \(-a\) and \(a - b\) are Cauchy sequences.

Proof These properties follow easily from the lemmas above. Note that
\[ c = -1, -1, \ldots \]
is a Cauchy sequence. By the Multiplication Lemma,
\[ c.a = -1.a_1, -1.a_2, \ldots, -1.a_n, \ldots \]
\[ = -a_1, -a_2, \ldots, -a_n, \ldots \]
\[ = -a \]
is a Cauchy sequence.
Similarly, by the Addition Lemma,
\[ a + (c.b) = a_1 + (-b_1), a_2 + (-b_2), \ldots, a_n + (-b_n), \ldots \]
\[ = a_1 - b_1, a_2 - b_2, \ldots, a_n - b_n, \ldots \]
\[ = a - b \]
is a Cauchy sequence. \qed

Definition (Division) Let
\[ a = a_1, a_2, \ldots, a_n, \ldots \text{ and } b = b_1, b_2, \ldots, b_n, \ldots \]
be sequences of rationals. If
\[ a_n \neq 0 \]
for all \(n\), then we define the multiplicative inverse of \(a\) by
\[ (a)^{-1} = a_1^{-1}, a_2^{-1}, \ldots, a_n^{-1}, \ldots. \]
If
\[ b_n \neq 0 \]
for all \(n\), then we define the division \(\frac{a}{b}\) of the two sequences to be the sequence
\[ \frac{a}{b} = a.(b)^{-1} = a_1 b^{-1}, a_2 b_2^{-1}, \ldots, a_n b_n^{-1}, \ldots \]
of rationals.
For the definitions to work, the operation of inverse must preserve Cauchy sequences. In
general, it does not. However, we can add a further hypothesis:

Lemma (Division) Let \( a = a_1, a_2, \ldots, a_n, \ldots \) be a Cauchy sequence satisfying these two prop-
erties:

(i) \( a_n \neq 0 \) for all \( n \); and

(ii) there exists \( \delta > 0 \) and stage \( k \), such that for all \( n > k \),

\[
|a_n| \geq \delta
\]

i.e., \( a \neq 0 \).

Then the sequence \( a^{-1} \) is a Cauchy sequence.

Proof Condition (i) enables us to define the sequence \( a^{-1} \). Let \( \delta \) and \( k \) be as in Condition (ii).
For any \( \epsilon > 0 \), since \( a \) is a Cauchy sequence, there exists \( n_0 \) such that

\[
|a_n - a_m| < \delta^2 \epsilon
\]

for all \( m, n > n_0 \). If \( N = \max(k, n_0) \), then for all \( m, n > N \), we have

\[
|a_n^{-1} - a_m^{-1}| = \frac{|a_m - a_n|}{a_n a_m} < \frac{\delta^2 \epsilon}{\delta \delta} = \epsilon.
\]

Hence, \( a^{-1} \) is a Cauchy sequence.

The case of division \( \frac{a}{b} \) is left as an exercise.

Finally, we consider the ordering.

Definition (Non-negative Cauchy Sequence) A Cauchy sequence \( a \in CSeq \) is said to be
non-negative if for any \( \epsilon > 0 \) there is a stage \( N \) such that

\[-\epsilon < a_n\]

for all \( n > N \).

Let \( NNCSeq \) be the set of all non-negative Cauchy sequences.

From this we can define an ordering on Cauchy sequences.

Definition (Ordering) Let \( a, b \in CSeq \). Define

\[ a \leq b \quad \text{if, and only if,} \quad a - b \in NNCSeq. \]

We will establish a number of basic properties of non-negative Cauchy sequences that will be useful later.

Lemma If \( a \) is a Cauchy sequence, then either \( a \) or \( -a \) is a non-negative Cauchy sequence.
**Proof** Suppose, for a contradiction, neither $a$ nor $-a$ is non-negative. Then there is some $\epsilon > 0$, such that for each $n_0$, there are $m, n > n_0$ and for which,

$$a_n \leq -\epsilon \quad \text{and} \quad -a_m \leq -\epsilon.$$  

Adding these inequalities gives

$$a_n - a_m \leq -2\epsilon \quad \text{and} \quad |a_n - a_m| \geq 2\epsilon.$$  

This contradicts that $a$ is a Cauchy sequence.  

\[ \square \]

**Lemma** If $a$ and $-a$ are non-negative Cauchy sequences then $a \equiv 0$.

**Proof** Suppose $a$ and $-a$ are non-negative. Then, by definition, for any $\epsilon > 0$, there is an $n_0$ such that

$$-\epsilon < a_n \quad \text{and} \quad -\epsilon < -a_n$$

or, equivalently,

$$|a_n| < \epsilon$$

for $n > n_0$. This is the condition for $a \equiv 0$.  

\[ \square \]

**Lemma (Operations on Non-Negative Cauchy Sequences)** If $a$ and $b$ are non-negative Cauchy sequences, then

(i) $a + b$ is a non-negative Cauchy sequence, and

(ii) $a \cdot b$ is a non-negative Cauchy sequence.

**Proof** (i) By the Boundedness Lemma, there is a bound $M > 0$ such that

$$|a_n| \leq M \quad \text{and} \quad |b_n| \leq M$$

for all $n$. Without loss of generality, we may assume $M \geq 2$.

Since $a$ and $b$ are non-negative, for any $\epsilon > 0$ there is a stage $n_0$ such that

$$-\frac{\epsilon}{M} < a_n \quad \text{and} \quad -\frac{\epsilon}{M} < b_n$$

for all $n > n_0$. Hence, for such $n$,

$$a_n + b_n > \frac{2\epsilon}{M} > -\epsilon,$$

since $M \geq 2$.

This proves that the sum is non-negative.

(ii) We need to show that $a_n \cdot b_n > -\epsilon$ for all $n > n_0$ where $n_0$ is as in (i). This breaks down into four sub-cases.

(a) $a_n \geq 0$ and $b_n \geq 0$. Clearly $a_n \cdot b_n \geq 0 > -\epsilon$.

(b) $a_n < 0$ and $b_n < 0$. Clearly $a_n \cdot b_n \geq 0 > -\epsilon$.  

(c) $a_n < 0$ and $b_n > 0$. Then

$$a_n \cdot b_n > \left( -\frac{\epsilon}{M}\right) b_n$$

$$> -\epsilon.$$

(d) $a_n > 0$ and $b_n < 0$. Similarly.

In each case, we see that $a_n \cdot b_n > -\epsilon$, and this proves that the product $a_n \cdot b_n$ is non-negative.

\[\square\]

8.4.3 Algebra of Equivalence Classes of Cauchy Sequences

The Cantor reals are equivalence classes of Cauchy sequences. To define the algebraic operations on Cantor reals, we have to lift the operations on Cauchy sequences to equivalence classes of Cauchy sequences. We will consider the operations in turn, giving a particularly full explanation in the first case of addition.

Addition

Let $[a]$ and $[b]$ be any Cantor reals, given by representatives of the equivalence classes $a, b \in CSeq$. We wish to define the addition of the Cantor reals by

$$[a] + [b] = [a + b]$$

where $a + b$ is defined in Section 8.4.2. However, there is a problem with the idea of defining operations on equivalence classes using representatives. How do we know that different choices of representatives, say $a', b' \in CSeq$ for $[a]$ and $[b]$ yield the same result. The question is this:

Is the operation of $+$ on equivalences properly defined? That is, given $a, a', b, b' \in CSeq$, do we know that

$$[a] = [a'] \text{ and } [b] = [b'] \quad \text{implies} \quad [a] + [b] = [a'] + [b']?$$

Equivalently, by definition, this condition can be written as

$$[a] = [a'] \text{ and } [b] = [b'] \quad \text{implies} \quad [a + b] = [a' + b'],$$

or, simply, as the congruence condition

$$a \equiv a' \text{ and } b \equiv b' \quad \text{implies} \quad a + b \equiv a' + b'.$$

We will verify this last property.

Lemma (Congruence) For any $a, a', b, b' \in CSeq$,

$$a \equiv a' \text{ and } b \equiv b' \quad \text{implies} \quad a + b \equiv a' + b'.$$
8.4. CANTOR’S CONSTRUCTION OF THE REAL NUMBERS

Proof Suppose \( a \equiv a' \) and \( b \equiv b' \). For any \( \epsilon > 0 \), there exists a common stage \( N \) such that

\[
|a_n - a'_n| < \frac{\epsilon}{2} \quad \text{and} \quad |b_n - b'_n| < \frac{\epsilon}{2}
\]

for all \( n > N \). For such \( n \), we can deduce that

\[
|(a_n + b_n) - (a'_n + b'_n)| = |a_n - a'_n + b_n - b'_n| \\
\leq |a_n - a'_n| + |b_n - b'_n| \\
< \frac{\epsilon}{2} + \frac{\epsilon}{2} \\
< \epsilon.
\]

Hence, \( a + b \equiv a' + b' \). \( \square \)

Thanks to the Congruence Lemma for addition on \( CSeq \), we may now conclude that the following idea works:

Definition Given any Cantor reals \([a], [b]\) with any representations \( a, b \in CSeq \), then we define their addition by

\( [a] + [b] = [a + b] \).

Additive Inverse

We wish to define the additive inverse of a Cantor real \([a]\), represented by \( a \in CSeq \), by

\( -[a] = [-a] \).

To ensure this definition works, we must prove that for any \( a, a' \in CSeq \),

\( [a] = [a'] \) implies \( -[a] = [-a'] \).

Equivalently, we must prove a congruence lemma for \(- \) on \( CSeq \).

Lemma (Congruence for \(- \)) For any \( a, a' \in CSeq \),

\( a \equiv a' \) implies \( -a \equiv -a' \).

Proof If \( a \equiv a' \) then, by definition, for any \( \epsilon > 0 \) there is an \( N \) such that

\[
|a_n - a'_n| < \epsilon
\]

for all \( n > N \). Now, rearranging the expression, we have

\[
|(-a_n) - (-a'_n)| = |a'_n - a_n| \\
= |a_n - a'_n| \\
< \epsilon.
\]

Hence \( -a \equiv -a' \). \( \square \)

Definition Given any Cantor real \([a]\) with any representation \( a \in CSeq \), we define the additive inverse by

\( -[a] = [-a] \).
Multiplication

We wish to define the multiplication of two Cantor reals \([a]\) and \([b]\), represented by \(a, b \in CSeq\) respectively, by

\[
[a] \cdot [b] = [a \cdot b].
\]

To ensure that this definition works, we must prove a congruence lemma for \(\cdot\) on \(CSeq\).

**Lemma (Congruence for \(\cdot\)).** For any \(a, a', b, b' \in CSeq\),

\[
a \equiv a' \text{ and } b \equiv b' \implies a \cdot b \equiv a' \cdot b'.
\]

**Proof** Since Cauchy sequences are bounded (Boundedness Lemma), there exists a common bound \(M > 0\) for \(a\) and \(b'\), i.e., for all \(n\),

\[
|a_n| \leq M \quad \text{and} \quad |b'_n| < M.
\]

Since \(a \equiv a'\) and \(b \equiv b'\), for any \(\varepsilon > 0\) there is a common stage \(N\) such that for \(n > N\),

\[
|a_n - a'_n| < \frac{\varepsilon}{2M} \quad \text{and} \quad |b_n - b'_n| < \frac{\varepsilon}{2M}.
\]

Hence, for such \(n\),

\[
|a_n \cdot b_n - a'_n \cdot b'_n| = |a_n (b_n - b'_n) + b'_n (a_n - a'_n)| \\
\leq |a_n| |b_n - b'_n| + |b'_n| |a_n - a'_n| \\
< M \left( \frac{\varepsilon}{2M} \right) + M \left( \frac{\varepsilon}{2M} \right) \\
< \varepsilon.
\]

Hence, \(a \cdot b \equiv a' \cdot b'\). \(\square\)

**Definition** Given any Cantor reals \([a]\) and \([b]\), with representations \(a, b \in CSeq\), then we define their multiplication by

\[
[a] \cdot [b] = [a \cdot b].
\]

**Multiplicative Inverse**

We wish to define the multiplicative inverse of a Cantor real \([a] \neq [0]\), represented by \(a \in CSeq\). This is not as straightforward as the previous cases. Now if \([a] \neq [0]\), then

\(a \neq 0\).

By the Corollary in Section 8.4.1, there exists some \(\varepsilon > 0\) and an \(n_0\) such that for all \(n > n_0\),

\[
|a_n| \geq \varepsilon.
\]

We define a new sequence \(c\) by

\[
e_n = \begin{cases} 
\varepsilon & \text{if } n \leq n_0; \\
|a_n| & \text{if } n > n_0.
\end{cases}
\]

It is easy to check the following:
8.4. CANTOR’S CONSTRUCTION OF THE REAL NUMBERS 283

Lemma c is a Cauchy sequence and c ≡ a, and so

\[ [c] = [a]. \]

Proof Exercise. \( \square \)

Now for all \( n, c_n \neq 0 \) and so, by the Division Lemma in Section 8.4.2, we have \( c^{-1} \in CSeq \). We wish to define

\[ [a]^{-1} = [c^{-1}] \]

but must first prove a congruence lemma.

Lemma (Congruence for \( -1 \)) For any \( a, a' \in CSeq \) such that

\[ a \neq 0 \quad \text{and} \quad a' \neq 0 \]

and for all \( n \),

\[ a_n \neq 0 \quad \text{and} \quad a'_n \neq 0. \]

Then

\[ a \equiv a' \quad \text{implies} \quad a^{-1} \equiv a'^{-1}. \]

Proof Since \( a \neq 0 \) and \( a' \neq 0 \), by the Corollary in Section 8.4.1, there is a \( \delta > 0 \) and stage \( k \), such that

\[ |a_n| \geq \delta \quad \text{and} \quad |a'_n| \geq \delta \]

for all \( n > k \).

Since \( a \equiv a' \), for any \( \epsilon > 0 \), there is an \( n_0 \) such that

\[ |a_n - a'_n| < \delta^2 \epsilon \]

for all \( n > n_0 \). If \( N = \max(k, n_0) \) then for all \( n > N \),

\[ |a_n^{-1} - a'_n^{-1}| = \frac{|a'_n - a_n|}{|a_n a'_n|} \leq \frac{|a'_n - a_n|}{\delta^2 |a'_n|} \leq \frac{\delta^2 \epsilon}{\delta \cdot \delta} \leq \epsilon. \]

Hence \( a^{-1} \equiv a'^{-1} \). \( \square \)

Definition Given any Cantor real \( [a] \neq [0] \), represented by \( a \in CSeq \), we define the multiplicative inverse by

\[ [a]^{-1} = [c]^{-1} \]

where \( c \in CSeq \) is the transformation of \( a \) defined above.

We have defined +, −, , and \( ^{-1} \) on \( CSeq/\equiv \) and what remains is the ordering.

We wish to define the ordering of two Cantor reals \( [a] \) and \( [b] \), represented by \( a, b \in CSeq \), by

\[ [a] \leq [b] \quad \text{if, and only if,} \quad a \leq b \]
or, equivalently,
\[ [a] \leq [b] \] if, and only if, \( b - a \) is a non-negative Cauchy sequence.

We need the following fact.

**Lemma (Congruence for \( \leq \))** For any \( a, a', b, b' \in CSeq \),
\[ a \equiv a', b \equiv b' \text{ and } a \leq b \] implies \( a' \leq b' \).

**Proof** By the Congruence Lemma for subtraction,
\[ a - b \equiv a' - b'. \]

By the following lemma, we know that \( a' - b' \) is non-negative, and hence that \( a' \leq b' \). \[ \square \]

**Lemma** If \( a \) is a non-negative Cauchy sequence and \( a \equiv a' \) then \( a' \) is a non-negative Cauchy sequence.

**Proof** For any \( \epsilon > 0 \) there is a stage \( n_0 \) such that
\[ -\frac{\epsilon}{2} < a_n, \]
and
\[ |a_n - b_n| < \frac{\epsilon}{2} \]
for all \( n > n_0 \). Thus, for such \( n \),
\[ -\frac{\epsilon}{2} < b_n - a_n \]
and
\[ b_n = a_n + (b_n - a_n) \]
\[ > -\frac{\epsilon}{2} - \frac{\epsilon}{2} \]
\[ > -\epsilon. \] \[ \square \]

### 8.4.4 Cantor Reals are a Complete Ordered Field

A complete ordered field is an algebra with five operations named in the signature \( \Sigma_{\text{Ordered Field}} \), that satisfy the 14 axioms in the set \( T_{\text{Complete Ordered Field}} \). The Uniqueness Theorem in Section 8.3 says that
\[ A, B \in Alg(\Sigma_{\text{Ordered Field}}, T_{\text{Complete Ordered Field}}) \Rightarrow A \cong B \]

Now, in Section 8.4.2 we constructed a \( \Sigma_{\text{Ordered Field}} \)-algebra
\[ (CSeq; 0, 1; +, -, \ldots, -1, \leq) \]
of Cauchy sequences of rational numbers to model and implement the real numbers, guided by ideas about measuring the line. As we saw in Section 8.4.2, this algebra is not a complete ordered field, although it is a commutative ring with multiplicative identity.
Then in Section 8.4.3, we used the equivalence relation \( \equiv \) to construct the \( \Sigma_{\text{Ordered Field}} \)-algebra

\[
R_{\text{Cantor}} = (C\text{Seq}/ \equiv; [0], [1]; a + b - c - d - e - f, a \leq b)
\]

of equivalence classes of Cauchy sequences of rationals. This required us to prove \( \equiv \) was a \( \Sigma_{\text{Ordered Field}} \)-congruence.

What we have to prove is this:

**Theorem** The \( \Sigma_{\text{Ordered Field}} \)-algebra \( R_{\text{Cantor}} \) is a complete ordered field, i.e.,

\[
R_{\text{Cantor}} \in \text{Alg}(\Sigma_{\text{Ordered Field}}, T_{\text{Complete Ordered Field}}).
\]

An immediate corollary of the Uniqueness Theorem is this:

**Corollary** Any complete ordered field is isomorphic with \( R_{\text{Cantor}} \).

**Proof of the Theorem**

We will prove that the \( \Sigma_{\text{Ordered Field}} \)-algebra \( R_{\text{Cantor}} \) satisfies

(a) the field axioms;

(b) the ordering axioms; and

(c) the completeness axiom.

We begin with (a), which follows from the following:

**Lemma** For all \( a, b, c \in C\text{Seq} \),

- **Associativity of addition** \( ([a] + [b]) + [c] = [a] + ([b] + [c]) \)
- **Identity for addition** \( [a] + [0] = [a] \)
- **Inverse for addition** \( [a] + (-[a]) = [0] \)
- **Commutativity for addition** \( [a] + [b] = [b] + [a] \)
- **Associativity for multiplication** \( ([a].[b]).[c] = [a].([b].[c]) \)
- **Identity for multiplication** \( [a].[1] = [a] \)
- **Inverse for multiplication** \( [a] \neq [0] \Rightarrow [a].[([a]^{-1})] = [1] \)
- **Commutativity for multiplication** \( [a].[b] = [b].[a] \)
- **Distribution** \( [a].([b] + [c]) = [a].[b] + [a].[c] \)
- **Distinctness** \( [0] \neq [1] \)

**Proof** Each of these properties follows from the corresponding property of the rational numbers.
CHAPTER 8. ABSTRACT DATA TYPE OF REAL NUMBERS

Associativity of addition We calculate

\((a + b) + c = \) \(a + (b + c)\) \hspace{1em} \text{by definition;}  \\
\((a + b) + c = \) \(a + (b + c)\) \hspace{1em} \text{associativity of rationals addition;}

Identity for addition We calculate

\(a + 0 = a + 0\) \hspace{1em} \text{by definition;}
\(0 + a = 0 + a\) \hspace{1em} \text{commutativity of rationals addition;}
\(a = a\) \hspace{1em} \text{rationals identity.}

Inverse for addition We calculate

\(a + (-a) = a + (-a)\) \hspace{1em} \text{by definition;}
\(a - a = 0\) \hspace{1em} \text{rationals addition.}

The remaining axioms are verified similarly. We tackle the inverse for multiplication and leave the rest as exercises.

Inverse for Multiplication Given \(a \neq 0\) we choose a Cauchy sequence \(a'\) such that

\([a'] = [a]\) \hspace{1em} \text{and} \hspace{1em} a'_n \neq 0 \text{ for all } n.

Then we can calculate

\([a].[a']^{-1} = [a].[a']^{-1}\) replacing \(a\) by \(a'\);
\([a].[a']^{-1} = [a'.a']^{-1}\) \hspace{1em} \text{by definition of }^{-1};
\([a'.a']^{-1} = [1]\) \hspace{1em} \text{by inverse property of the rationals.}

\(\square\)

The next task (b) is to show that \(R_{\text{Cantor}}\) is an ordered field. This follows from the following lemma.

Lemma For all \(a, b, c \in CSeq,\)

Reflectivity \hspace{1em} \([a] \leq [a]\)

Antisymmetry \hspace{1em} \([a] \leq [b] \text{ and } [b] \leq [a] \Rightarrow [a] = [b]\)

Transitivity \hspace{1em} \([a] \leq [b] \text{ and } [b] \leq [c] \Rightarrow [a] \leq [c]\)

Total Order \hspace{1em} Either \([a] \leq [b]\) \text{ or } \([b] \leq [a]\)

Addition \hspace{1em} \([a] \leq [b] \Rightarrow [a] + [c] \leq [b] + [c]\)

Multiplication \hspace{1em} \([a] \leq [b] \text{ and } [0] \leq [c] \Rightarrow [a].[c] \leq [b].[c]\)
8.4. CANTOR’S CONSTRUCTION OF THE REAL NUMBERS

Proof

Reflexivity
Now \([a] - [a] = [a - a] = [0]\), which is non-negative. Thus, \([a] \leq [a]\) by definition of \(\leq\) on Cauchy sequences.

Antisymmetry
Suppose \([a] \leq [b]\) and \([b] \leq [a]\). Then \(b - a\) and \(a - b\) are both non-negative. By Lemma ?? in Section 8.4.2,

\[a - b \equiv 0 \quad \text{and} \quad b - a \equiv 0.\]

In particular, \([a] = [b]\). 

Transitivity
Suppose \([a] \leq [b]\) and \([b] \leq [c]\). Then \(b - a\) and \(c - b\) are non-negative and, by Lemma ?? in Section 8.4.2,

\[c - a = (b - a) + (c - b)\]

is non-negative. Thus, \([a] \leq [c]\).

Total Order
Given any \(a, b\), consider \(b - a\). By Lemma ??, either \(b - a\) is non-negative, or \(-(b - a) = a - b\) is non-negative. Hence, either

\([a] \leq [b] \quad \text{or} \quad [b] \leq [a].\)

Addition
If \([a] \leq [b]\) then \(b - a\) is non-negative. Now

\[(b + c) - (a + c) = b - a\]

is non-negative, and so

\[[a + c] \leq [b + c]\]

\[[a] + [c] \leq [b] + [c].\]

Multiplication
If \([a] \leq [b]\) and \([0] \leq [c]\) then \(b - a\) and \(c - 0 = c\) are non-negative. Multiplying,

\[(b - a).c = bc - ac\]

is non-negative, and so

\[[ac] \leq [bc]\]

\[[a].[c] \leq [b].[c].\]

\(\square\)

The final task (c) is to prove the completeness axiom holds for \(\mathbb{R}_{\text{Cantor}}\). To do this, some preparations are necessary.

The rational numbers play an important rôle throughout, of course, but here we are to prove that the approximation process is complete in a precise sense. Consider how the rational numbers are embedded in \(\mathbb{R}_{\text{Cantor}}\).
If \( \frac{p}{q} \in \mathbb{Q} \), we write

\[
\left( \frac{p}{q} \right) = \frac{p}{q}, \frac{p}{q}, \ldots, \frac{p}{q}, \ldots
\]

for the constant sequence where every element is \( \frac{p}{q} \). Clearly, \( \left( \frac{p}{q} \right) \in CSeq \). Further, we write

\[
\left[ \frac{p}{q} \right] = \{ a \in CSeq \mid a \equiv \left( \frac{p}{q} \right) \}
\]

for the equivalence class of all Cauchy sequences equivalent to \( \left( \frac{p}{q} \right) \). Strictly speaking, we have a mapping

\[
\phi : \mathbb{Q} \to R_{\text{Cantor}}
\]

defined for \( \frac{p}{q} \in \mathbb{Q} \) by

\[
\phi\left( \frac{p}{q} \right) = \left[ \frac{p}{q} \right]
\]

that embeds \( \mathbb{Q} \) inside \( R_{\text{Cantor}} \).

Recalling the properties of a \( \Sigma_{\text{Ordered Field}} \)-homomorphism from Section 8.3.1 we show that:

**Theorem** The mapping \( \phi : \mathbb{Q} \to R_{\text{Cantor}} \) is a \( \Sigma_{\text{Ordered Field}} \)-homomorphism that is injective.

We leave the proof of this theorem to Exercise 10. Of particular use is this order-preserving property

\[
\frac{p}{q} < \frac{p'}{q'} \quad \text{implies} \quad \phi\left( \frac{p}{q} \right) < \phi\left( \frac{p'}{q'} \right)
\]

or

\[
\frac{p}{q} < \frac{p'}{q'} \quad \text{implies} \quad \left[ \frac{p}{q} \right] < \left[ \frac{p'}{q'} \right].
\]

**Lemma** For any \( [a] \in R_{\text{Cantor}} \), there exists \( c, d \in \mathbb{Z} \) such that

\[
[(c)] < [a] < [(d)].
\]

**Proof** Since \( a \) is a Cauchy sequence, choosing \( \epsilon = 1 \), there exists \( N \) such that

\[
|a_n - a_m| < 1
\]

for all \( n, m > N \). Let \( k > N \) and \( c, d \in \mathbb{Z} \) such that

\[
c \leq a_k - 1 \quad \text{and} \quad d \geq a_k + 1.
\]

Then for all \( n > k \),

\[
c < a_n < d.
\]

It follows that

\[
(a_n - c) \quad \text{and} \quad (d - a_n)
\]

are non-negative, and hence,

\[
[(c)] < [a] \quad \text{and} \quad [a] < [(d)].
\]

\( \square \)
Lemma Let \( c, d \in CSeq \) and suppose \( c \equiv d \). For any \( a, b \in CSeq \), if
\[
[(c_n)] \leq [a] \leq [b] \leq [(d_n)]
\]
for all \( n \), then \( a \equiv b \) and
\[
[a] = [b].
\]

Proof Since \( c \equiv d \), for any \( \epsilon > 0 \), there is some \( n_0 \) such that
\[
|d_{n_0} - c_{n_0}| < \frac{\epsilon}{3}.
\]
By hypothesis, \( [(c_{n_0})] \leq [(d_{n_0})] \), so \( c_{n_0} \leq d_{n_0} \) and
\[
0 \leq d_{n_0} - c_{n_0} < \frac{\epsilon}{3}.
\]
By hypothesis, \( [(c_{n_0})] \leq [a] \) and \( [b] \leq [(d_{n_0})] \), and thus,
\[
(a_n - c_{n_0}) \quad \text{and} \quad (d_{n_0} - b_n)
\]
are non-negative sequences. Therefore, there exists some \( n_1 \) such that for all \( n > n_1 \),
\[
-\frac{\epsilon}{3} \leq a_n - c_{n_0} \quad \text{and} \quad -\frac{\epsilon}{3} \leq d_{n_0} - b_n.
\]
Rearranging,
\[
-a_n \leq \frac{\epsilon}{3} - c_{n_0} \quad \text{and} \quad b_n \leq \frac{\epsilon}{3} + d_{n_0}
\]
and adding,
\[
b_n - a_n \leq \frac{2\epsilon}{3} + (d_{n_0} - c_{n_0})
\]
\[
< \frac{2\epsilon}{3} + \frac{\epsilon}{3}
\]
\[
< \epsilon.
\]
Finally, we note from the hypothesis that
\[
[a] \leq [b]
\]
that the sequence \((b_n - a_n)\) is non-negative. This means there is some \( n_2 \) such that for all \( n > n_2 \),
\[
-\epsilon < b_n - a_n.
\]
Setting \( N = \max(n_1, n_2) \) we have that
\[
|a_n - b_n| < \epsilon
\]
for all \( n > N \). This means that \( a \equiv b \). \( \square \)

Lemma (Completeness) Any non-empty subset \( B \) of \( R_{Cantor} \) that is bounded below has a greatest lower bound.

Proof Given any lower bound \([a]\) of \( B \), we construct a Cauchy sequence
\[
c = c_0, c_1, c_2, \ldots, c_t, \ldots
\]
of rational numbers such that \([c]\) is a greatest lower bound of \( B \). We define \( c \) inductively on \( t \).
Basis $t = 0$.

The first element $c_0$ is defined as follows. Suppose $B$ is bounded below by $[a]$. Then, for all $[b] \in B$, $[a] \leq [b]$. By Lemma..., there exist integers $m$ and $n$ such that

$$[(m)] \leq [a] \leq [b] \leq [(n)].$$

In particular, $m$ determines a class that is a lower bound for $B$, and $n$ determines a class that is not a lower bound. Therefore, there must exist a largest integer $c_0$ such that $m \leq c_0 < n$ and

$$[(c_0)]$$

is a lower bound for $B$ but $[(c_0 + 1)]$ is not a lower bound for $B$.

This $c_0$ is the first element of $c$.

Induction Step From the element $c_t$, we define $c_{t+1}$ as follows. Suppose $c_t$ is a rational number such that

$$[(c_t)]$$

is a lower bound for $B$ but $[(c_t + \frac{n}{10^{t+1}})]$ is not a lower bound for $B$.

Consider the expression

$$c_t + \frac{n}{10^{t+1}}.$$

We know this determines a lower bound for $n = 0$ and that it is not a lower bound for $n = 10$. Therefore, there exists some $0 < n_{t-1} < 10$ such that $n_{t}$ is the largest integer such that

$$c_{t+1} = c_t + \frac{n_t}{10^{t+1}}$$

determines a lower bound. This defines the element $c_{t+1}$ from $c_t$.

Lemma

$$c = c_0, c_1, c_2, \ldots \in CSeq$$

Proof Given any $\epsilon > 0$, choose some $t$ such that

$$\frac{1}{10^t} < \epsilon.$$

Now, if $t < m < n$, then we may show by induction on the formula defining $c_t$,

$$c_t \leq c_m \leq c_n < c_t + \frac{1}{10^t}.$$

Hence, for all $m, n > t$, we have

$$|c_m - c_n| < \frac{1}{10^t} < \epsilon$$

and $c$ is a Cauchy sequence. □

Consider $[c]$. Now for each $t$,

$$c_t \leq c_{t+1}.$$

For any choice $k$, the constant sequence $(c_k)$ satisfies

$$[(c_k)] \leq [c]$$

(1)
because
\[
[c] - [(c_k)] = [c - (c_k)] \\
= [(c_t) - (c_k)] \\
= [(c_t - c_k)]
\]

and \( c_t - c_k \) is non-negative.

Now, for any \( t > k \) we have, by definition,
\[
c_t < c_k + \frac{1}{10^k}
\]

and, hence,
\[
[c] \leq [(c_k + \frac{1}{10^k})]. \tag{2}
\]

Combining inequalities (1) and (2) gives for all \( k \),
\[
[(c_k)] \leq [c] \leq [(c_k + \frac{1}{10^k})] \tag{3}
\]

**Claim** \([c] \) is a lower bound of \( B \), i.e., for all \([b] \in B\),
\[
[c] \leq [b].
\]

**Proof of Claim** Suppose, for a contradiction, that there is \([b] \in B\) such that \([b] < [c]\). Hence, using (3) we have for all \( k \),
\[
[(c_k)] \leq [b] < [c] \leq [(c_k + \frac{1}{10^k})].
\]

However, it is easy to see that
\[
(c_k) \equiv (c_k + \frac{1}{10^k}).
\]

By Lemma..., \([b] = [c]\), which is a contradiction.

**Claim** \([c] \) is the greatest lower bound of \( B \).

**Proof of Claim** Suppose, for a contradiction, that there exists a lower bound \([d] \) for \( B \) such that
\[
[c] < [d].
\]

Since for every \( k \), the constant sequence class
\[
[(c_k + \frac{1}{10^k})]
\]

is not a lower bound, we have from (3) above that for all \( k \),
\[
[(c_k)] \leq [c] < d \leq [(c_k + \frac{1}{10^k})].
\]

By Lemma ..., this again implies \([c] = [d]\) which is a contradiction. Hence, \([c] \) is the greatest lower bound. \(\square\)
8.5 Computable Real Numbers

In each method for constructing the real number representation of the line, a point is represented by an infinite object; for example, the number $\sqrt{2}$ is represented by two infinite sets of rationals (Dedekind), or by one of infinitely many infinite sequences of rationals (Cantor).

How can we compute with these infinite objects that represent real numbers?

To give the representation of a real number, such as $\sqrt{2}$, we need an algorithm. To add two real numbers $a$ and $b$ we need an algorithm that inputs algorithms for the representations of $a$ and $b$ and returns an algorithm for the representation of $a + b$.

A real number is a computable real number if there is an algorithm that allows us to compute a representation of the number, i.e. a rational number approximation to the real number to any given degree of accuracy. This idea must be made exact by applying it to a definition of a representation of real numbers that spells out the sense of approximation. Consider Dedekind and Cantor’s ways of making the idea of approximation exact.

**Definition (Computable numbers following Dedekind)** A real number is computable if there exists a computable Dedekind cut that represents it. This means that there exists a pair $(A_1, A_2)$ of sets of rational numbers and decision algorithm that determine for any $\frac{p}{q} \in \mathbb{Q}$ whether or not $\frac{p}{q} \in A_1$ or $\frac{p}{q} \in A_2$.

**Definition (Computable numbers following Cantor)** A real number is computable if there exists a computable Cauchy sequence that represents it. This means:

(i) there is an algorithm that computes a function $a : \mathbb{N} \rightarrow \mathbb{Q}$ that enumerates a sequence of rationals $a_1, a_2, \ldots, a_n, \ldots$, that is, for $n \in \mathbb{N}$, $a(n) = a_n$; and

(ii) an algorithm that computes a function $N : \mathbb{Q} \rightarrow \mathbb{N}$ such that for any rational number $\epsilon > 0$, the natural number $N(\epsilon)$ has the property that

for all $n, m > N(\epsilon)$, we have $|a_n - a_m| < \epsilon$.

A number of questions arise concerning the algebraic properties of these computable numbers, for example:

If $a$ and $b$ are computable real numbers, then are $a + b$ and $a \cdot b$ computable real numbers?

The answer is yes and in fact the familiar simple functions on reals return real numbers:

**Theorem** The set of computable real numbers forms an ordered subfield of the ordered field of real numbers.

The field of computable real numbers has many important properties but it is not a complete ordered field. Of primary interest are questions about computing with computable numbers such as:

Are there algorithms that transform the algorithms that enumerate approximations of $a$ and $b$ to algorithms that enumerate an approximation for $a + b$, $a \cdot b$, $-a$, and $a^{-1}$?
This leads to the questions about the computability of the set of reals. The field of reals is not a computable algebra in the sense of Section 7.6, though it has algorithms for its operations. The whole subject demands a careful study that will look at the computability of data representations and the calculus.

Most of the real numbers we know and use come from solving equations (e.g., the algebraic numbers) and evaluating equationally defined sequences (e.g., $e$ and $\pi$) and are computable.

However, most real numbers are non-computable. This is because the set of computable reals is infinite and countable, since the set of algorithms is countable. However, the set of reals is infinite and not countable by Cantor's diagonal method. The infiniteness of the set of reals is strictly bigger than the infiniteness of the set of computable reals. The fact that the set of reals is uncountable implies there does not exist a method of making finite representations or codings for all real numbers. Real numbers are inherently infinite objects.

## 8.6 Representations of Real Numbers and Practical Computation

Our requirements analysis for the data type of real numbers focussed on the problem of measuring the line. We have developed a lot of theory which has given us

1. Insight into the fundamental roles of measurement, approximations and the rational numbers.

2. An axiomatic specification of the data type of real numbers, i.e.,

$$(\Sigma_{\text{Ordered Field}}, T_{\text{Complete Ordered Field}}).$$

3. A proof that all representations or implementations of the axiomatic specification are equivalent, i.e., the uniqueness of algebras satisfying the axioms up to isomorphism.

4. A particular representation based on Cauchy sequences of rational numbers.

5. Questions about the scope and limits of computing with real numbers.

We will conclude this chapter by addressing rather briefly the following questions:

*Can we perform practical computations with Cauchy representations of real numbers?*

and:

*What is the relationship between Cauchy representations and floating point representations?*
8.6.1 Practical Computation

Now a Cantor real number is represented by two functions

\[(a, M)\]

in which

\[a : \mathbb{N} \to \mathbb{Q}\]

generates the Cauchy sequence

\[a(1), a(2), \ldots\]

of rational numbers and

\[M : \mathbb{Q} \to \mathbb{N}\]

defines the accuracy of the sequence by means of the Cauchy property

\[(\forall \epsilon)(\forall n)(\forall m)[n > M(\epsilon) \text{ and } m > M(\epsilon) \Rightarrow |a(n) - a(m)| < \epsilon].\]

The function \(M\) is called the \textit{modulus of convergence} for the sequence \(a\).

Given a Cantor real number with representation \((a, M)\), we can define an approximation function

\[\text{approx} : \mathbb{Q} \to \mathbb{Q}\]

such that for any error margin \(\epsilon \in \mathbb{Q}\),

\[\text{approx}(\epsilon) = \text{some rational number within } \epsilon \text{ of the real number represented by } (a, M)\]

by

\[\text{approx}(\epsilon) = a(M(\epsilon)).\]

More formally, in terms of equivalence classes of Cauchy sequences, if \([\text{approx}(\epsilon)]\) and \([\epsilon]\) are the \textit{classes} of constant sequences \(\text{approx}(\epsilon)\) and \(\epsilon\), then

\[[a] - [\epsilon] < [\text{approx}(\epsilon)] < [a] + [\epsilon]\]

\textbf{Definition (Fast Cauchy Sequences)} A Cauchy sequence \(a\) is said to be a \textit{fast converging sequence} if

\[(\forall n)(\forall m)[m > n \Rightarrow |a(n) - a(m)| < 2^{-n}]\]

This means that the rational \(a(n)\) approximates the real to within \(2^{-n}\). More formally, we have that the modulus function has the property

\[M(2^{-n}) = n\]

and

\[\text{approx}(2^{-n}) = a(n).\]

\textbf{Theorem} For every Cauchy sequence \(a\), there is a fast Cauchy sequence \(b\) that represents the same Cantor real number. \textit{i.e.,}

\[[a] = [b].\]
8.6. REPRESENTATIONS OF REAL NUMBERS AND PRACTICAL COMPUTATION

A computable Cantor real number is represented by two algorithms

$$ (\alpha_a, \alpha_M) $$

for computing these functions $a$ and $M$ where $\alpha_a$ computes the sequence $a$ and $\alpha_M$ computes the modulus of convergence $M$. By sequentially composing these algorithms $\alpha_a$ and $\alpha_M$ we get an algorithm

$$ \alpha_{approx} = \alpha_M; \alpha_a $$

for computing $approx$. Furthermore, there is an algorithm that transforms the algorithms $\alpha_a$, $\alpha_M$ and $\alpha_{approx}$ to algorithms for a fast Cauchy sequence that represents the same real number.

To represent computable Cantor real numbers and program their operations in practice, however, fast Cauchy sequences are not enough. Space and time resources demand several refinements. For example, the operations on rational numbers quickly lead to rationals $\frac{p}{q}$ with large numerators $p$ and denominators $q$. The space needed to store a rational number is

$$ \log_2(|p|) + \log_2(|q|) + c $$

where $c$ is a constant. Indeed, some iterative processes can yield an exponential growth in the lengths of $p$ and $q$. Squaring $\frac{p}{q}$ to get $\frac{p^2}{q^2}$ can double the space required to

$$ 2(\log_2(|p|) + \log_2(|q|)) + c. $$

This is a problem that can be ameliorated by using dyadic rationals.

**Definition (Dyadic rationals)** A rational number $r \in \mathbb{Q}$ is a dyadic rational if it is of the form

$$ r = \frac{b}{2^k} $$

where $b \in \mathbb{Z}$ and $k \in \mathbb{Z}$. Let $\mathbb{Q}_{Dyadic}$ be the set of all dyadic rationals.

A dyadic number is a rational number with a finite binary expansion. The space needed to store a dyadic rational $b.2^{-k}$ is

$$ \log_2(|b|) + \log_2(|k|) + c $$

where $c$ is a constant. In practice, $k$ will often be smaller than $b$.

A first refinement is to restrict the representations to Cauchy sequences of dyadic rationals. To accomplish this, we need the following facts.

**Theorem** The set $\mathbb{Q}_{Dyadic}$ of dyadic rationals is closed under

$$ +, -, \text{ and } . $$

and hence forms a sub-ring of the commutative ring $\mathbb{Q}$ of rational numbers. Furthermore, the set $\mathbb{Q}_{Dyadic}$ is a countably infinite set that is dense in $\mathbb{Q}$, i.e., for any $r_1, r_2 \in \mathbb{Q}$ there is $b.2^{-k} \in \mathbb{Q}_{Dyadic}$ such that

$$ r_1 < \frac{b}{2^k} < r_2. $$
It is essential that $b$ and $k$ are not bounded for these properties. Clearly, $Q_{Dyadic}$ is not closed under multiplicative inverse since

$$3 = \frac{3}{2^0} \in Q_{Dyadic}$$

but

$$\frac{1}{3} \not\in Q_{Dyadic}$$

Another refinement is to restrict further the set of Cauchy sequences to be used, e.g., by adding extra error information.

### 8.6.2 Floating Point Representations

Floating point representations of real numbers are designed to use standard memory structures of computers in a straightforward way. In practice, there are many variations and refinements, but one can think of them as a form of dyadic representation.

The connection with floating point representations is made by

(i) placing bounds on the $b$ and $k$; and  

(ii) replacing an infinite sequence of rationals by a single rational

$$\frac{b}{2^k}.$$  

The floating point representation based on $b$ and $k$ define a subset $Float(b, k)$ of the real numbers. The subset has two problems:

(i) the distribution of the real numbers leaves serious gaps; and  

(ii) the field axioms fail to hold for the basic operations.

The fact that $Float(b, k)$ is not a subfield of $\mathbb{R}$ is particularly disappointing: the user’s ideas and expectations of real number algebra collapse. Indeed, creating the algebra of $Float(b, k)$ seems to be a mathematical challenge that seems to offer little mathematical reward, but considerable technological reward.
8.6. REPRESENTATIONS OF REAL NUMBERS AND PRACTICAL COMPUTATION

Exercises for Chapter 8

1. Prove the following. Let \( m, n \in \mathbb{N} \) and suppose \( m \neq k^n \) for any \( k \in \mathbb{N} \). Then \( \sqrt[n]{m} \) is an irrational number.

2. Prove that for all rational numbers \( x, y \in \mathbb{Q} \),
   
a. \( |x + y| = |y + x| \);
   b. \( |x - y| = |y - x| \);
   c. \( |x.y| = |y.x| \);
   d. \( |x.y| = | -x.y| = |x - y| = | -x - y| \);
   e. \( |x^2| = |x|^2 = x^2 \);
   f. \( -|x| \leq x \leq |x| \);
   g. \( |x + y| \leq |x| + |y| \);
   h. \( |x - y| \leq |x| + |y| \);
   i. \( ||x| - |y|| \leq |x - y| \);

3. Prove that the set \( \mathbb{Q} \) of all rational numbers is countably infinite.

4. Show that for any rational numbers \( x, y \in \mathbb{Q} \),
   \[
   x < \frac{1}{2}(x + y) < y.
   \]
   Hence, deduce that for any \( x, y \in \mathbb{Q} \), if \( x \neq y \) then there are infinitely many rational numbers between \( x \) and \( y \). Deduce that
   \[
   \{ r \in \mathbb{Q} \mid x < r < y \}
   \]
   is countably infinite.

5. Give a geometric construction that uses an (unmarked) ruler and compass that divides any line segment into \( n \) equal parts, for any \( n \in \mathbb{N} \) and \( n \geq 1 \).

6. Which of the following statements are true in any field? If the statement is true, deduce it from the axioms. If it is false, give an example of a field in which it fails.
   a. \( 1.0 = 0 \);
   b. \( (\forall x)(\forall y)(\forall z)((x + y).z = x.z + y.z) \);
   c. \( (\forall x)(\forall y)(\forall z)((x + y)^2 = x^2 + (1 + 1).x.y + y^2) \);
   d. \( (\forall x)(\forall y)(\forall z)((x + y).(x - y) = x^2 - y^2) \);
   e. \( (\forall x)(\exists y)[x = y^2] \);
   f. \( (\forall x)(\forall y)[x.y = 0 \Leftrightarrow x = 0 \text{ or } y = 0] \);
g. all fields are infinite.

7. Show that the set $\text{Seq}$ of all sequences of rational numbers

$$\text{Seq} = \{a = (a_1, a_2, \ldots) \mid a_n \in \mathbb{Q}, n \geq 1\}$$

forms a commutative ring with multiplicative identity under the pointwise operations. Show that $\text{Seq}$ is not a field.

8. Show that the set $\text{CSeq}$ of all Cauchy sequences of rational numbers forms a commutative ring with multiplicative identity under the pointwise operations. Show that $\text{CSeq}$ is not a field.

9. Define a map $\phi : \text{CSeq} \to \text{CSeq}/\equiv$ by

$$\phi(a) = [a]$$

for $a \in \text{CSeq}$. Show $\phi$ is a ring homomorphism that is surjective.

10. Prove that the mapping

$$\phi : \mathbb{Q} \to \mathbb{R}_{\text{Cantor}}$$

is an injective field homomorphism.

11. A sequence $c = c_1, c_2, \ldots$ is a fast Cauchy sequence if, for all $m, n$ with $m < n$,

$$|c_n - c_m| < 2^{-n}.$$

Show that every fast Cauchy sequence is a Cauchy sequence.

Show that for every Cauchy sequence $a$, there exists a fast Cauchy sequence $c$, such that

$$c \equiv a.$$
Part II

Syntax
Introduction

There are countless types of written texts that are defined by special notations and rules that govern their composition. The symbols, notations, layout and other conventions, and rules that characterise a category of text together form the syntax of the category of text.

English, like most natural languages, has many examples of particular types of text: novels, essays, poems, prayers, letters, postal addresses, shopping lists, forms, laws, contracts, last wills and testaments, examination papers, school reports, etc. It is rarely easy to define such categories of text in natural languages. Often, it is easy to give a set of basic rules and refine them by contemplating examples. However, to give a list of rules that completely defines a category of natural language texts is difficult, if not impossible.

Texts are not confined to natural languages. In artificial languages, classes of texts that are of great interest are notation systems, such as mathematical formulae, chemical formulae and musical notation. These texts — along with poems and postal addresses — are two dimensional.

Languages for programming also have many categories. Examples are programs, procedures, data types, objects, interfaces, specifications, files, hyper-texts, identifiers, names and addresses. There are many tools for processing these texts such as compilers, interpreters, word processors, communication protocols, editors and browsers. It is possible to give a complete set of rules for the description of artificial languages, and often it is not terribly difficult.

In Part 1 we will study the elements of the theory of syntax with an emphasis on applications to programming languages. Among the topics we meet are the method of defining concrete syntax using a context free grammar, the limitations of purely context free definitions, kernel languages and their extensions, and the use of algebraic methods to specify abstract syntax.

We will begin with the problem of defining the precise form of texts. In Chapter 9, we introduce grammars, derivations and formal languages, and consider some simple examples of syntactic specifications such as addresses (both mail and internet). In Chapter 10, we look at the specification of interface definition languages and imperative programming languages. In Chapter 11, we develop some of the mathematical theory of context grammars and apply results, such as the Pumping Lemma, to answer questions about the syntax of programming languages. Finally in Chapter 12, we introduce a more abstract approach to syntax based on term algebras.
Chapter 9

Syntax and Grammars

Languages are systems in which we express ourselves and communicate with one another. The languages we speak are called natural languages and the majority have a written form. In computing we also communicate with machines. To instruct a machine we must express our intentions with great precision. Since the early days of computing, thousands of written languages have been developed to do just this. The programming languages in which we write programs to control machines are defined formally by strict rules.

A program is a text. It is made from different kinds of notations and linguistic notions — such as types, identifiers, variables, expressions, constructs, procedures and so on. The notations and notions obey rules that specify how to form texts that are valid programs in the language. In an analogy with natural languages, we say that the notations, notions and rules form the grammar of the language. The grammar and the texts it defines together form the syntax of the programming language.

There are other kinds of languages of interest in computer science, in addition to programming languages. For example, there are languages for describing architectures, writing formal specifications, and reasoning about components, in both software and hardware. Logical languages, like the language of first order logic, are used extensively in computing. In fact, logical languages were the first languages to be defined formally with a syntax and a semantics. They were invented to analyse the foundations of mathematical reasoning, and our theoretical understanding of programming languages originates with concepts developed in logic.

There are also languages consisting of special types of texts with restricted uses that must be precisely defined. For example, addresses are small pieces of text that are always governed by strict formation rules.

We will study syntax in order to understand how to specify programming languages, and other languages of interest. Our first theory of syntax views a program as a

\textit{string of symbols}

and a programming language as a set of such strings. It is based on the mathematical ideas of a

\textit{formal language},

which is a set of strings made from an arbitrary alphabet, and a

\textit{grammar},

303
which is a set of rules for forming strings. The theory has proven to be of immense practical value.

In this chapter and the next, we will develop just enough of the theory of grammars to understand the basic ideas involved in the practical definition of the syntax of programming languages. Specifically, in this chapter, in Section 9.1, we introduce alphabets and formal languages and show just how widely applicable they are. In Section 9.2, we define grammars and explain how to use their rules to make formal derivations of strings that generate formal languages. In Section 9.3, we introduce some practical techniques used to define the syntax of languages, including a modular method of writing grammars, and the BNF notation. Programming languages are neither small nor simple and the modular grammars allow us to introduce syntactic categories in stages, while the BNF provides a notation that is easy to read and remember. We illustrate the process of formally defining languages with these tools by explaining two case studies, both of which are examples of addresses. In Section 9.4, we define, in some detail, the following:

- postal addresses; and
- World Wide Web addresses.

The next chapter is devoted to examples of the formal definition of programming languages.

## 9.1 Alphabets, Strings and Languages

### 9.1.1 Formal Languages

**Definition (Alphabet)** An alphabet is a finite non-empty set $T$. We shall consider the elements of $T$ to be symbols.

Simple familiar examples are the alphabet

$$T_{\text{Digit}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

of decimal digits and the lower-case alphabet

$$T_{\text{English \ Alphabet}} = \{a, b, c, \ldots, x, y, z\}$$

of the English language.

**Definition (String)** Given any symbols $t_1, t_2, \ldots, t_n \in T$, we can form the string, or word,

$$w = t_1 t_2 \cdots t_n$$

over the alphabet $T$. The set of all strings or words over the alphabet $T$ is

$$T^* = \{t_1 t_2 \cdots t_n \mid n \geq 0, t_1, \ldots, t_n \in T\}$$

For example, the numbers

1984 and 2001
are words over the alphabet $T_{\text{Digit}}$ of decimal digits, and

$$\text{forwards \ and \ sdrawkcal}$$

are words over the lower-case English alphabet $T_{\text{English Alphabet}}$.

The length $|w|$ of a string $w$ is the number of symbols from the alphabet that it contains. Thus, if $w = t_1 \cdots t_n$ then $|w| = n$. For example,

$$|2000| = 4 \quad \text{and} \quad |\text{forwards}| = 8.$$ 

The empty string uses no symbols from $T$ and is represented by the symbol $\epsilon$. The length $|\epsilon| = 0$. We refer to the set of all non-empty strings over an alphabet $T$ as $T^+$. Thus,

$$T^+ = T^* \setminus \{\epsilon\}.$$ 

**Definition (Concatenation)** Given two strings

$$u = u_1 \cdots u_m \quad \text{and} \quad v = v_1 \cdots v_n$$

over $T$, the concatenation $uv$ of $u$ and $v$ is the string:

$$uv = u_1 \cdots u_m v_1 \cdots v_n$$

The length

$$|uv| = |u| + |v|.$$ 

We write $w^n$ for the string made by concatenating $n$ copies of $w$, i.e.,

$$w^n = w \cdots w \quad (n \text{ times}).$$

For example:

$$(\text{forwards})^3 = \text{forwards forwards forwards}$$

A formal language is some set of strings over an alphabet. In particular, it has the property that we can define exactly those strings that that are in the language. We can define a formal language $L$ by enumerating, or in effect defining a dictionary that lists all the strings of $L$. Alternatively, we can describe a formal language $L$ by describing the pattern of the strings that are in $L$. Mathematically, we can model both of these methods of describing a formal language with the idea of forming a *set* of strings.

**Definition (Formal Language)** A formal language $L$ over an alphabet $T$ is simply some subset

$$L \subseteq T^*$$

of the set $T^*$ of all possible strings over $T$.

For brevity, we say *language* for formal language.
9.1.2 Simple Examples

Let us look at some example languages.

Let us start with a very simple alphabet $T_{ab}$ consisting of just two letters:

$$T_{ab} = \{a, b\}.$$ 

Here is a partial enumeration

$$T^n_{ab} = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}.$$ 

of the set of all possible strings over $T_{ab}$. Notice that there are $2^{n+1} - 1$ words of length at most $n$.

At first sight, this may seem to be very limited. However, if we had chosen the symbols 0 and 1 instead of $a$ and $b$ it is obvious strings of two letters have unlimited use since they include all binary numbers.

Let us consider just a few examples of languages over $T_{ab}$.

$$L = \{\epsilon\} \quad L = \emptyset$$

$$L = \{a, b\} \quad L = \{abba\}$$

$$L = \{a^n | n \text{ is even}\} \quad L = \{a^p | p \text{ is prime}\}$$

$$L = \{a^n b^n | n \geq 1\} \quad L = \{a^n b^{n+1} | n \geq 1\}$$

$$L = \{(ab)^n | n \geq 0\} \quad L = \{(ba)^n | n \geq 0\}$$

There are many languages over $T_{ab}$ which have useful practical properties. For instance, if we considered the symbols $a$ and $b$ to represent some form of left and right parentheses, then this leads us to examples of bracketing methods:

<table>
<thead>
<tr>
<th>Alphabet</th>
<th>Bracketing Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{[]} = {(),}$</td>
<td>$L_{()} = {(n)^n</td>
</tr>
<tr>
<td>$T_{\text{begin end}} = {\text{begin, end}}$</td>
<td>$L_{\text{begin end}} = {\text{begin}^n \text{ end}^n</td>
</tr>
<tr>
<td>$T_{\text{please thank-you}} = {\text{please, thank-you}}$</td>
<td>$L_{\text{please thank-you}} = {\text{please}^n \text{thank-you}^n</td>
</tr>
<tr>
<td>$T_{\text{hello goodbye}} = {\text{hello, goodbye}}$</td>
<td>$L_{\text{hello goodbye}} = {\text{hello}^n \text{goodbye}^n</td>
</tr>
</tbody>
</table>

9.1.3 Natural Language Examples

The concept of a formal language was originally developed by Noam Chomsky (b. 1928) in 1956 (Chomsky [1956]) to model aspects of natural languages.

1. If we take the alphabet

$$T_{\text{English Alphabet}} = \{a, b, \ldots, z\}$$

then we can consider the language

$$L_{\text{English Words}} = \{w \in T_{\text{English Alphabet}} | w \text{ is an English word}\}$$

consisting of all English words over the alphabet $T_{\text{English Alphabet}}$. So,

$$\text{island} \in L_{\text{English Words}} \quad \text{but} \quad \text{ynys} \notin L_{\text{English Words}}.$$
2. If we remove the vowels from the alphabet $T_{English\ Alphabet}$, to give

$$T_{English\ Consonants} = \{a, b, \ldots, z\} \setminus \{a, e, i, o, u\}.$$

and consider the language

$$L_{English\ Consonant\ Words} = \{w \in T_{English\ Consonants} \mid w\ is\ an\ English\ word\}$$

then we have a much smaller language. It is non-empty though, as for example,

$$\text{rhythm} \in L_{English\ Consonant\ Words}.$$

3. The lower-case alphabet for Welsh is

$$T_{Welsh\ Alphabet} = \{a, b, c, ch, d, dd, e, f, ff, g, ng, h, i, l, ll, m, n, o, p, ph, r, rh, s, t, th, u, w, y\}.$$

Considering the language

$$L_{Welsh\ Words} = \{w \in T_{Welsh\ Alphabet}^* \mid w\ is\ a\ Welsh\ word\},$$

then, we find that the word

$$\text{ynys} \in L_{Welsh\ Words},$$

which means “island” in English. If we also add spaces to the alphabet of Welsh letters, we can form the language

$$L_{Welsh\ Phrases} = \{w \in (T_{Welsh\ Alphabet} \cup \{\})^* \mid w\ is\ a\ Welsh\ word\}$$

of Welsh phrases. Then the pair of strings

$$\text{gwyddor cyrifiadur} \in L_{Welsh\ Phrases}$$

which means “computer science” in English.

4. The lower-case alphabet for ancient Greek is

$$T_{Ancient\ Greek\ Alphabet} = \{\alpha, \beta, \gamma, \delta, \varepsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \omicron, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}.$$

(See Figure 9.1.)

If we define the language

$$L_{Ancient\ Greek\ Phrases} = \{w \in (T_{Ancient\ Greek\ Alphabet} \cup \{\})^* \mid w\ is\ a\ phrase\ of\ ancient\ Greek\}.$$

Then, the phrase

$$\gamma\nu\omega\theta\iota\sigma\varepsilon\alpha\upsilon\tau\omicron\nu \in L_{Ancient\ Greek\ Phrases}$$

which means “know thyself”. 
<table>
<thead>
<tr>
<th>Lower Case</th>
<th>Upper Case</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>A</td>
<td>Alpha</td>
</tr>
<tr>
<td>β</td>
<td>B</td>
<td>Beta</td>
</tr>
<tr>
<td>γ</td>
<td>Γ</td>
<td>Gamma</td>
</tr>
<tr>
<td>δ</td>
<td>Δ</td>
<td>Delta</td>
</tr>
<tr>
<td>ε</td>
<td>E</td>
<td>Epsilon</td>
</tr>
<tr>
<td>ζ</td>
<td>Z</td>
<td>Zeta</td>
</tr>
<tr>
<td>η</td>
<td>H</td>
<td>Eta</td>
</tr>
<tr>
<td>θ</td>
<td>Θ</td>
<td>Theta</td>
</tr>
<tr>
<td>ι</td>
<td>I</td>
<td>Iota</td>
</tr>
<tr>
<td>κ</td>
<td>K</td>
<td>Kappa</td>
</tr>
<tr>
<td>λ</td>
<td>λ</td>
<td>Lambda</td>
</tr>
<tr>
<td>μ</td>
<td>M</td>
<td>Mu</td>
</tr>
<tr>
<td>ν</td>
<td>N</td>
<td>Nu</td>
</tr>
<tr>
<td>ξ</td>
<td>Ξ</td>
<td>Xi</td>
</tr>
<tr>
<td>ο</td>
<td>O</td>
<td>Omicron</td>
</tr>
<tr>
<td>π</td>
<td>Π</td>
<td>Pi</td>
</tr>
<tr>
<td>ρ</td>
<td>P</td>
<td>Rho</td>
</tr>
<tr>
<td>σ</td>
<td>Σ</td>
<td>Sigma</td>
</tr>
<tr>
<td>τ</td>
<td>T</td>
<td>Tau</td>
</tr>
<tr>
<td>υ</td>
<td>Y</td>
<td>Upsilon</td>
</tr>
<tr>
<td>ϕ</td>
<td>Φ</td>
<td>Phi</td>
</tr>
<tr>
<td>χ</td>
<td>Χ</td>
<td>Khi/Chi</td>
</tr>
<tr>
<td>ψ</td>
<td>Ψ</td>
<td>Psi</td>
</tr>
<tr>
<td>ω</td>
<td>Ω</td>
<td>Omega</td>
</tr>
</tbody>
</table>

Figure 9.1: The alphabet of Attic Greek of the Classical Period (5th–4th century BC).

5. Restrictions of natural languages are also used. For example, consider the alphabet
\( T_{\text{Shipping}} = \{ \text{DOVER, FISHER, WHITE, PORTLAND, PLYMOUTH, …, NORTH, SOUTH, EAST, WEST, NORTHERLY, SOUTHERLY, EASTERLY, WESTERLY NORTHEAST, NORTHWEST, SOUTHEAST, SOUTHWEST, NORTHEASTERLY, NORTHWESTERLY, SOUTHEASTERLY, SOUTHWESTERLY, CALM, LIGHT, FRESH, STRONG, GALE, FORCE, VARIABLE, VEERING, BACKING, CYCLONIC, LOW, LOWS, HIGH, COMPLEX, RISING, FALLING, MOVING, CHANGE GOOD, POOR, MODERATE, HAZE, MIST, FOG, PATCHES, SMOKE, PRECIPITATION, RAIN, SHOWERS, SQUALLY, WINTRY, SLEET, SNOW, ICING, SLOW, SLOWLY, QUICKLY, STEADILY, SPREADING, LITTLE, 0, 1, 2, …, METRE, METRES, MILE, MILES, SIGHT, CENTRED TODAY, YESTERDAY, TOMORROW, NOW, THEN, HOURS, EXPECTED, SOON, BECOMING, OCCASIONALLY, MAINLY, POSSIBLY, OTHERWISE, TIMES, WEATHER, FORECAST, SHIPPING, ISSUED, REPORTS, WARNINGS, COASTAL, STATIONS, MET., OFFICE, GENERAL, SYNOPSIS, AREA, FORECASTS, MONDAY, TUESDAY, …, JANUARY, FEBRUARY, … WITH, WITHIN, IN, AND, OR, BUT, FOR, AT, TO, OF, THE, … \} \)

This is used to give the language

\( L_{\text{Shipping Weather Forecasts}} = \{ w \in T^* | w \text{ is a "legal" shipping weather forecast} \} \)

of shipping weather forecasts issued by the UK meteorological office.

### 9.1.4 Languages of Addresses

We examine some examples of languages that we can form by considering the concept of an address.

1. Let

\[ T = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z \} \cup \{ 0, 1, …, 9, -, `, "", ", !, \}. \]

Let

\( L_{\text{Postal Addresses}} = \{ w \in T^* | w \text{ is a postal address in the United Kingdom} \} \)

Then both
Department of Computer Science  
University of Wales Swansea  
Singleton Park  
SWANSEA  
SA2 8PP

and

Adran Gwyddor Cyfrifiadur  
Prifysgol Cymru Abertawe  
Parc Singleton  
ABERTAWE  
SA2 8PP

are strings in \( L_{Postal \ Addresses} \).

2. Continuing the idea of addresses, uniform resource locators (URLs) of the internet provide some interesting examples of languages.

Let

\[ T = \{a, b, c, \ldots, x, y, z, A, B, C, \ldots, X, Y, Z, 0, 1, \ldots, 9, /, \ldots \} \]

Let

\[ L_{HTTP} = \{w \in T^* \mid w \text{ is an hypertext transfer protocol address}\} \]

Simple addresses such as

http://www.w3.org \hspace{1cm} http://www.swansea.ac.uk

are in \( L_{HTTP} \), but the alphabet needs extending to capture all hypertext transfer protocol addresses. For example,

http://www.google.com/search?q=swansea+university+computer+science

http://www.google.com/search?q=as_q=&num=10&btnG=Google+Search&as_oq=&as_epq=Swansea+University&as_eq=&as_occt=title&lr=lang_en&as_dt=i&as_sitesearch=&safe=off

9.1.5 Programming Language Examples

Now we consider some examples concerning data and programming languages.

1. Let

\[ T_{Infix \ Arithmetic} = \{0, 1, +, -, (, )\} \]

The expressions of type natural number are strings such as:

\[
\begin{align*}
(0 + 1) + 1 & \quad 0 + (1 + 1) \\
0 + 1 & \quad 1 + 0 \\
0 + ((0 + 1) + 1) & \quad (0 + (0 + 1)).(0 + 1)
\end{align*}
\]

However,

\[ 1 + 2 \notin T_{Infix \ Arithmetic} \quad \text{and} \quad x.1 \notin T_{Infix \ Arithmetic} \]

since \( 2 \notin T_{Infix \ Arithmetic} \) and \( x \notin T_{Infix \ Arithmetic} \).
2. Let us consider some more forms of arithmetic expressions. Let

\[ T_{\text{Prefix \ Arithmetic}} = \{ \text{zero, succ, add, mult, , ,(,)} \}. \]

The expressions of type natural number are strings such as

\[ \text{succ(succ(zero))} \quad \text{add(zero, succ(succ(zero)))} \quad \text{mult(add(zero, succ(zero)), succ(zero))} \]
\[ \text{add(succ}(n)\text{zero), succ}(n)\text{zero)}. \]

Let

\[ L_{\text{Prefix \ Arithmetic \ Expressions}} = \{ w \in T_{\text{Prefix \ Arithmetic}} | w \text{ is an arithmetic expression} \}. \]

3. Consider the signatures that name the data and operations in algebras. Each signature is a string. Let

\[ T_{\text{Signature}} = \{ \text{signature, sorts, constants, operations, endsig, \text{\&, , , ;, \times, \rightarrow, \ a, b, \ldots, z, A, B, \ldots, Z}} \}. \]

Let

\[ L_{\text{Signature}} = \{ w \in T_{\text{Signature}} | w \text{ is a "legal" signature} \}. \]

Then the string

\[ \begin{align*}
\text{signature} & \quad \text{count} \\
\text{sorts} & \quad \text{nat} \\
\text{constants} & \quad \text{zero :\text{nat}} \\
\text{operations} & \quad \text{successor : nat \rightarrow nat} \\
\text{endsig} & \quad \end{align*} \]

is in \( L \).

4. Consider the while programs we want to define exactly. Each program is a string over an alphabet. Let

\[ T = \{ \text{skip, read, write, if, then, else, fi, while, do, od, begin, end, ;, :: :=} \]
\[ +, -, *, /, \text{mod, =, >, <, =, <, >, =, (, )}, \text{true, false, not, and, or, 0, 1, \ldots, 9, a, b, \ldots, z, A, B, \ldots, Z} \} \]

and

\[ L = \{ w \in T^* | w \text{ is a "legal" while program} \}. \]

### 9.2 Grammars and Derivations

The examples illustrate the idea that many kinds of syntax are formed by making strings of symbols and notations. But some of the formal languages we have seen in Section 9.1, such as the set of English words, are difficult to define exactly. The problem is to find methods of defining formal languages.

Grammars are a specification method for formal languages.
9.2.1 Grammars

A grammar is a mathematical idea designed to specify formal languages. Essentially, it is a collection of rules to generate the strings of a language. The rules of a grammar define how we can form a string by means of step by step substitutions.

**Description**

A grammar $G$ has four components:

(i) an alphabet $T$ of terminal symbols which we shall form strings over;

(ii) a different set $N$ of non-terminal symbols which we shall use to control the substitutions that generate the strings;

(iii) a particular symbol $S$ from $N$ that we shall use to initiate the substitution process; and

(iv) a set $P$ of substitution rules.

The purpose of a grammar $G = (T, N, S, P)$ is to define a formal language

$$L(G) \subseteq T^*$$

over the alphabet $T$ by applying the production rules of $P$ beginning with start symbol $S$.

A sequence of applications of the production rules is called a *derivation*. The non-terminals in the rules permit substitutions in strings. When all non-terminals have been eliminated, then the derivation ends and we are left with a string of symbols from the alphabet. Hence the alternate word “terminal” for elements of the alphabet: terminals remain when a derivation terminates and all the non-terminals have been removed. We will explain how this works in general shortly, but first we present the formal definition, then we will look at some examples.

**Definition (Grammar)** A grammar $G = (T, N, S, P)$ consists of:

(i) a finite set $T$ called the *alphabet of terminal symbols*;

(ii) a finite set $N$ of *non-terminal symbols*, or *variable symbols*, with $N \cap T = \emptyset$;

(iii) a special non-terminal symbol $S \in N$ called the *start symbol*; and

(iv) a finite set $P$ of *rules*, or *productions*, each of which has the form:

$$u \rightarrow v$$

where the left-hand string $u \in (T \cup N)^+$ is non-empty, and the right-hand string $v \in (T \cup N)^*$. Note that both the left- and right-hand strings can contain terminals and/or non-terminals.

We present a grammar as a 4-tuple

$$G = (T, N, S, P)$$

and also use a displayed version, particularly for examples:
### 9.2. Grammars and Derivations

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>$T$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N$</td>
</tr>
<tr>
<td>start</td>
<td>$S$</td>
</tr>
<tr>
<td>rules</td>
<td>$P$</td>
</tr>
</tbody>
</table>

#### 9.2.2 Examples

1. Consider a grammar:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{01}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$0, 1$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow 1$  $S \rightarrow 0S$ $S \rightarrow 1S$</td>
</tr>
</tbody>
</table>

We show how the rules generate strings such as:

1001 and 011

First, we number the production rules for ease of reference:

1. $S \rightarrow 1$
2. $S \rightarrow 0S$
3. $S \rightarrow 1S$

Then applying Rule 3 to the start symbol $S$, we get:

$S \rightarrow 1S$

Applying Rule 2 twice to $S$:

$\rightarrow 10S$
$\rightarrow 100S$
Finally, applying Rule 3 to $S$:

$$
\rightarrow 1001
$$

Another alternative is to apply Rule 2 to the start symbol $S$:

$$
S \rightarrow 0S
$$

Then to apply Rule 3 to $S$:

$$
S \rightarrow 01S
$$

And finally to apply Rule 1 to $S$:

$$
S \rightarrow 011
$$

So a derivation begins with $S$ and proceeds by applying the rules in any order, for any number of times; a derivation is a path in the tree illustrated in Figure 9.2.

![Figure 9.2: Possible derivations from the grammar $G^{01}$.](image)

What are the strings over $\{0,1\}$ that these production rules generate? The rules generate the set of all strings over 0 and 1 which end in a 1:

$$
L(G^{01}) = \{w1 \mid w \in \{0,1\}^*\}
$$

2. Let $G^{ab} = (\{a,b\}, \{S\}, S, \{S \rightarrow ab, S \rightarrow aSb\})$. This is displayed as:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{ab}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td>$S \rightarrow ab \quad S \rightarrow aSb$</td>
</tr>
</tbody>
</table>
9.2. GRAMMARS AND DERIVATIONS

\[
S \\
ab \quad aSb \\
aabb \quad aaSbb
\]

Figure 9.3: Possible derivations from the grammar \( G^{ab} \).

Here derivations proceed as a path in the tree in Figure 9.3.

What are the strings over \{a, b\} that these rules generate? The productions generate the set

\[ L(G^{ab}) = \{a^n b^n \mid n \geq 1 \} \]

of strings.

3. Let \( G^a = (\{a\}, \{S\}, S, \{S \to \epsilon, S \to aSa\}) \). This is displayed as:

<table>
<thead>
<tr>
<th>grammar ( G^a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals ( a )</td>
</tr>
<tr>
<td>nonterminals ( S )</td>
</tr>
<tr>
<td>start symbol ( S )</td>
</tr>
<tr>
<td>productions ( S \to \epsilon \ S \to aSa )</td>
</tr>
</tbody>
</table>

Here derivations begin with \( S \) and proceeds as a path in the tree in Figure 9.4. Notice

\[
S \\
\epsilon \quad aSa \\
aa \quad aaSaa
\]

Figure 9.4: Possible derivations from the grammar \( G^a \).

that we suppress any empty string \( \epsilon \) generated as part of a non-empty string; this is a convention which is observed for clarity as concatenating the empty string \( \epsilon \) has no effect. Thus, the productions generate the set

\[ L(G^a) = \{a^{2n} \mid n \geq 1\} \]

of strings over \{a\}.
9.2.3 Derivations

Now let us consider the substitution process that is captured by derivations. First, we examine a single derivation

\[
\text{sut} \Rightarrow \text{sut}
\]

where we substitute the string \(v\) for the string \(u\) using a production

\[
u \rightarrow v.
\]

Thus, \(\Rightarrow\) is a relation on strings. In particular, it tells us how we can rewrite a non-empty string to some string, provided that there is a production rule that we can apply.

So, more formally, we define a relation

\[
\Rightarrow_G \subseteq (T \cup N)^+ \times (T \cup N)^*
\]

with respect to a grammar \(G = (T, N, S, P)\) as follows.

**Definition (One-Step Derivation)** Given a grammar \(G = (T, N, S, P)\), we one-step derive a (possibly empty) word \(w' \in (T \cup N)^*\) from a non-empty word \(w \in (T \cup N)^+\):

\[
w \Rightarrow_G w'
\]

if, and only if, there is a production \(u \rightarrow v \in P\) and we can write

\[
w = s u t \text{ and } w' = s v t
\]

where \(s, t \in (T \cup N)^*\) are possibly empty strings.

We also use the phrases immediately generated or one-step generated in place of one-step derived. And we simply write

\[
w \Rightarrow w'
\]

in place of \(w \Rightarrow_G w'\), when there is no danger of ambiguity arising.

We want to consider the effect of performing a number of derivations. We define a relation

\[
w \Rightarrow^* w',
\]

to say that a word \(w'\) can be produced from a word \(w\) by a sequence of derivations:

**Definition (Derivation)** Given a grammar \(G = (T, N, S, P)\), we derive a (possibly empty) word \(w' \in (T \cup N)^*\) from a non-empty word \(w \in (T \cup N)^+\):

\[
w \Rightarrow^*_G w'
\]

if, either

(i) \(w = w'\), or

(ii) there is a sequence of non-empty words \(w_0, w_1, \ldots, w_{n-1} \in (T \cup N)^+\) and a final (possibly empty) word \(w_n \in (T \cup N)^*\), such that

(a) \(w_0 = w\);
(b) for \( 1 \leq i \leq n-1 \), \( w_i \Rightarrow_G w_{i+1} \); and
(c) \( w_n = w' \).

Again, we shall also use the terminology that the string \( w \) generates the string \( w' \) or alternatively that \( w' \) is derived from \( w \), and we simply write

\[ w \Rightarrow^* w' \]

in place of \( w \Rightarrow^*_G w' \), when there is no danger of ambiguity arising.

### 9.2.4 Language Generation

We now have all the components that we need to explain how we generate a language from a grammar. Recall the form of a typical grammar:

| grammar | \( G \) |
| terminals | \( T \) |
| nonterminals | \( N \) |
| start symbol | \( S \) |
| productions | \( P \) |

We generate a language \( L \) from a grammar \( G \) by considering all the possible strings \( w \in T^* \) of terminal symbols that we can generate from the start symbol \( S \). In such derivations we may make use of non-terminal symbols in the substitutions that arise from the use of production rules.

**Definition (Language)** Let \( G = (T, N, S, P) \) be a grammar. The language \( L(G) \subseteq T^* \) generated by the grammar \( G \) is defined by

\[ L(G) = \{ w \in T^* \mid S \Rightarrow^*_G w \} \]

Of course, there is not a single way of defining a language: if we can find one grammar to define a language, there will be infinitely many other grammars to define it.

**Definition (Grammar Equivalence)** We will say that two grammars \( G_1 \) and \( G_2 \) are equivalent if, and only if,

\[ L(G_1) = L(G_2) \]

Of these different methods for describing how we can construct a language, some will lead to clearer descriptions, whilst others may lead to more efficient implementations. We shall see more of this idea in the next chapter.
Example

Consider the grammar $G^{ab}$ of Examples 9.2.2(ii). A typical derivation is:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^n b^n.$$ 

This involved the use of $n$ one-step derivations: $n-1$ applications of the second production and one application of the first to eliminate the non-terminal symbol $S$. This demonstrates that

$$S \Rightarrow^* G a^n b^n$$

and hence,

$$\{a^n b^n \mid n \geq 1\} \subseteq L(G^{ab}).$$

9.3 Specifying Syntax using Grammars: Modularity and BNF Notation

There are two major ways of improving the readability of grammars. The first is simply a question of presentation, and leads just to an alternative notation. The second method though is the use of modularity, a powerful idea that we have already seen in action in specifying data and interfaces with algebras and signatures.

9.3.1 Designing Syntax using Grammars

The procedure for specifying syntax is roughly as follows.

Let $L$ be some language in need of a precise specification.

First, one chooses an alphabet $T$ such that

$$L \subseteq T^*.$$ 

Secondly, one creates a simple grammar $G$ such that

$$L \subseteq L(G).$$

And thirdly, one addresses the problem of removing the undesirable strings in the set

$$L(G) \setminus L$$

as shown in Figure 9.5.

The problem of unwanted strings is a common occurrence in language design in practice.

9.3.2 A Simple Example of Building a Language using Grammars

We will show how the mathematical concept of a grammar allows us to build definitions for a simple programming language. The main components of a **while** programming language for computing with natural numbers are:

(i) identifiers;
9.3. SPECIFYING SYNTAX USING GRAMMARS: MODULARITY AND BNF NOTATION

![Diagram of language specification using a grammar](image)

Figure 9.5: Specifying a language using a grammar.

(i) natural numbers;
(ii) arithmetic expressions;
(iii) Boolean expressions; and
(iv) programs.

We will give a series of grammars that defines each component in turn.

A Simple Language

We start with a grammar $G^{Identifiers}$ for generating identifiers:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{Identifiers}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>a, b, ..., z, A, B, ..., Z</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Letter, Id</td>
</tr>
<tr>
<td>start</td>
<td>Id</td>
</tr>
<tr>
<td>rules</td>
<td>Id $\rightarrow$ Letter</td>
</tr>
<tr>
<td></td>
<td>Id $\rightarrow$ Letter Id</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ a</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ b</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ z</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ A</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ B</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>Letter $\rightarrow$ Z</td>
</tr>
</tbody>
</table>

Now we give a grammar $G^{Numbers}$ to generate numbers (in decimal notation):
Next, we construct a grammar $G^{\text{Arithmetic Expressions}}$ for generating arithmetic expressions from the two previous grammars for identifiers and numbers:

The alphabets, start symbols and rules of the grammars $G^{\text{Identifiers}}$ and $G^{\text{Numbers}}$ in the grammar $G^{\text{Arithmetic Expressions}}$ for arithmetic expressions are included by means of the new construct import. We will explain the meaning of import shortly.

We define a grammar $G^{\text{Boolean Expressions}}$ for generating Boolean expressions by:
### 9.3. Specifying Syntax Using Grammars: Modularity and BNF Notation

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{\text{Boolean Expressions}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G^{\text{Arithmetic Expressions}}$</td>
</tr>
<tr>
<td>alphabet</td>
<td>true, false, not, and, or, =, &lt;</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$BExp, BOp1, BOp2, RelOp$</td>
</tr>
<tr>
<td>start</td>
<td>$BExp$</td>
</tr>
</tbody>
</table>
| rules | $BExp \rightarrow BOp1 BExp$  
$BExp \rightarrow BExp BOp2 BExp$  
$BExp \rightarrow AExp RelOp AExp$  
$BExp \rightarrow$ true  
$BExp \rightarrow$ false  
$BOp1 \rightarrow$ not  
$BOp2 \rightarrow$ and  
$BOp2 \rightarrow$ or  
$RelOp \rightarrow =$  
$RelOp \rightarrow <$ |

The alphabet, start symbol and production rules of the grammar $G^{\text{Arithmetic Expressions}}$ are needed to define Boolean expressions and again are included using `import`.

Finally, we define a grammar $G^{\text{while}}$ for generating `while` programs, using the grammars we defined for generating arithmetic and Boolean expressions, by:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{\text{while}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G^{\text{Arithmetic Expressions}}, G^{\text{Boolean Expressions}}$</td>
</tr>
<tr>
<td>alphabet</td>
<td>skip, if, then, else, fi, while, do, od, =, :=, ;</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Program</td>
</tr>
<tr>
<td>start</td>
<td>Program</td>
</tr>
</tbody>
</table>
| rules | $Program \rightarrow$ skip  
$Program \rightarrow Id := AExp$  
$Program \rightarrow Program ; Program$  
$Program \rightarrow$ if $BExp$ then $Program$ else $Program$ fi  
$Program \rightarrow$ while $BExp$ do $Program$ od |

Actually, importing the grammar of arithmetic expressions into the grammar for `while` programs is redundant since it is already a part of the grammar for Boolean expressions.
9.3.3 Modularity and the Import Construct

We built the grammar $G^{while}$ for programs from grammars for its subsidiary components arithmetic expressions and Boolean expressions. The dependencies are illustrated in Figure 9.6. The picture suggests a modular structure or architecture for the grammar $G^{while}$ and hence the

![Diagram showing dependencies between grammars](image)

Figure 9.6: Component grammars used to construct the grammar $G^{while}$.

language. The grammars are included via the import construct. Let us define what it means.

**Definition (Importing Grammars)** Let $H$ be a grammar

<table>
<thead>
<tr>
<th>grammar</th>
<th>$H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>$G$</td>
</tr>
<tr>
<td>alphabet</td>
<td>$B$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$M$</td>
</tr>
<tr>
<td>start</td>
<td>$R$</td>
</tr>
<tr>
<td>rules</td>
<td>$Q$</td>
</tr>
</tbody>
</table>

that imports the grammar $G$:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>$A$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N$</td>
</tr>
<tr>
<td>start</td>
<td>$S$</td>
</tr>
<tr>
<td>rules</td>
<td>$P$</td>
</tr>
</tbody>
</table>
Then the flattened grammar $H$ is defined to be:

<table>
<thead>
<tr>
<th>grammar</th>
<th>Flattened $H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabet</td>
<td>$A \cup B$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$N \cup M$</td>
</tr>
<tr>
<td>start</td>
<td>$R$</td>
</tr>
<tr>
<td>rules</td>
<td>$Q \cup P$</td>
</tr>
</tbody>
</table>

### 9.3.4 User Friendly Grammars and BNF Notation

Programming languages are rarely simple and the grammars used to define them are rarely small. The concise mathematical notation that is suitable for analysis needs to be exchanged for a notation that is suitable for writing and reading large grammars.

**Backus-Naur Form (BNF)**

A popular method of presenting a grammar for a programming language is that of Backus-Naur Form which is based on the following conventions.

1. The terminal symbols of the grammar are written in bold font.
2. The non-terminal symbols of the grammar are familiar terms enclosed in angle brackets, e.g., `<statement>`, `<expression>`, `<identifier>`.
3. The start symbol is the non-terminal that is presented first.
4. The symbol ::= replaces and extends $\rightarrow$ by listing the productions possible for a non-terminal; alternative possibilities for the right-hand sides of a particular production are separated with the symbol $|$.

For example, we abbreviate the five rules:

```
<BExp> ::= <BOp1> <BExp>
<BExp> ::= <BExp> <BOp2> <BExp>
<BExp> ::= <AExp> <RelOp> <AExp>
<BExp> ::= true
<BExp> ::= false
```

by the “composite” rule:

```
<BExp> ::= <BOp1> <BExp> | <BExp> <BOp2> <BExp> |
         <AExp> <RelOp> <AExp> |
         true | false
```
Elemental Grammar Examples

We shall find use for some basic grammars in many of the grammars that we consider. The first gives us the letters of the English alphabet:

\[
\text{bnf } \text{Letter}
\]

\[
\text{rules}
\]

\[
<\text{Letter}> ::= <\text{LowerCase}> \mid <\text{UpperCase}>
\]

\[
<\text{LowerCase}> ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z
\]

\[
<\text{UpperCase}> ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z
\]

And the second gives us numbers and sequences of digits; numbers differ from sequences of digits by starting at 1 and disallowing leading zeroes.

\[
\text{bnf } \text{Number}
\]

\[
\text{rules}
\]

\[
<\text{Number}> ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid <\text{Number}> <\text{Digits}>
\]

\[
<\text{Digit}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

\[
<\text{Digits}> ::= <\text{Digit}> \mid <\text{Digit}> <\text{Digits}>
\]

9.4 What is an Address?

Addresses are everywhere in computing and communication. Obvious high-level examples are:

(i) variable names for memory locations;

(ii) file names; and

(iii) universal resource locators.

Obvious low-level examples are

(i) IP addresses for computers;

(ii) numbers and other codes for mobile phones; and

(iii) bar-codes for goods.

Addresses are often defined to be inputs to protocols such as HTTP. The idea of an address has enormous scope and, hence, is complex. Addresses provide excellent examples of syntax in need of formal definition by grammars.
9.4. WHAT IS AN ADDRESS?

9.4.1 Postal Addresses

Let us examine how we can construct British postal addresses for mail that is posted within the United Kingdom.

**Strings versus Displays**

Consider the address:

Department of Computer Science, University of Wales Swansea, Singleton Park, Swansea, SA2 8PP

This string is a postal address in the United Kingdom, (as discussed in Example 2 of Section 9.1.4).

When addressing an envelope, we could *display* this string as:

Department of Computer Science,  
University of Wales Swansea,  
Singleton Park,  
Swansea,  
SA2 8PP

or:

Department of Computer Science,  
University of Wales, Swansea,  
Singleton Park,  
Swansea, SA2 8PP

Furthermore, in official practice the commas are removed, the postal town capitalised, all the lines are aligned with each other, and each syntactic component is on a separate line, as follows:

Department of Computer Science  
University of Wales Swansea  
Singleton Park  
SWANSEA  
SA2 8PP

In some ways, the display of the address is a part of the address.

**UK Postal Addresses**

A UK postal address can normally be categorised as

(i) a rural address;

(ii) a town address; or

(iii) a business or organisation address.
We will devise grammars to construct addresses of each type so that we can define:

```plaintext
bnf    PostalAddresses
import RuralAddress, TownAddress, CorporateAddress
rules  <Address> ::= <RuralAddress> | <TownAddress> | <CorporateAddress>
```

First we consider elements that are common to all three types of address, namely:

(i) postcodes;
(ii) phrases;
(iii) line-breaks.

A picture of the construction is shown in Figure 9.7.

![Figure 9.7: Structure of addresses.](image)

**Postcodes**

In the UK, the combination of a postcode and a house name or number, uniquely identifies any postal address. Currently, a postcode has one of three forms, as described in the rules below. Thus, a postcode may have seven or eight characters, with a separating space before the final three characters. An example of a postcode is:

SA2 8PP
9.4. WHAT IS AN ADDRESS?

```
bnf  Postcode

import  Letter, Number

rules
<Postcode> ::= <Letter> <Letter> <Digit> <Space> <Digit> <Letter> <Letter> |
               <Letter> <Digit> <Digit> <Space> <Digit> <Letter> <Letter> |
               <Letter> <Letter> <Digit> <Digit> <Digit> <Letter> <Letter> <Letter> |
               <Space> <Digit> <Letter> <Letter>

<Space> ::= ________________________________
```

Note that the non-terminal <Space> derives the single space character.

**Phrases**

In order to write an address, we shall want to form strings of characters. Here, we shall define a phrase as being formed either entirely from letters, or a mixture of letters and certain punctuation marks. For example, we want to be able to have phrases such as the following:

John O’Groats   Westward Ho!   Weston-super-mare   “Melfort”

We will also want to have a restricted form of such phrases whereby any letters used are in capitals.

```
bnf  Phrase

import  Letter

rules
<Phrase> ::= <Letter> | <Letter> <Phrase> | <Punctuation> <Letter> <Phrase> | <Phrase> <Punctuation>

<UpperCasePhrase> ::= <UpperCase> | <UpperCase> <UpperCasePhrase> | <Punctuation> <UpperCase> <UpperCasePhrase> | <UpperCasePhrase> <Punctuation>

<Punctuation> ::= - | ’ | “ | ” | ! | .
```

We shall have newline characters as a separate type of punctuation:
\textbf{bnf} \hspace{1em} \textit{Newline}

\textbf{rules}
\begin{align*}
\langle \text{Newline} \rangle & ::= \\
\end{align*}

\textbf{Rural Addresses}

Properties in the UK which are in very rural locations typically are identified with a house name, the nearest village and the postal town which sorts the mail for the surrounding district.

\textbf{bnf} \hspace{1em} \textit{RuralAddress}

\textbf{import} \hspace{1em} \textit{Postcode, Phrase, Newline}

\textbf{rules}
\begin{align*}
\langle \text{RuralAddress} \rangle & ::= \langle \text{BuildingName} \rangle \ \langle \text{Newline} \rangle \ \langle \text{Village} \rangle \ \langle \text{Newline} \rangle \\
& \hspace{1em} \langle \text{PostalTown} \rangle \ \langle \text{Newline} \rangle \ \langle \text{Postcode} \rangle \\
\langle \text{BuildingName} \rangle & ::= \langle \text{Phrase} \rangle \\
\langle \text{Village} \rangle & ::= \langle \text{Phrase} \rangle \\
\langle \text{PostalTown} \rangle & ::= \langle \text{UpperCasePhrase} \rangle
\end{align*}

\textbf{Town Addresses}

In less rural locations, single-residence property identifiers may be a house name, number, range of numbers, or combination of name and number(s). Multi-residence properties, such as flats are typically identified by a room number and the name of the property.
In addition, these less rural properties are typically located on some named road, within a district of a town; properties that are located in the centre of a town frequently omit the district.

Corporate Addresses

Corporate addresses, are typically addressed by department, organisation, building, road and postal town. However, the omission of department or building are frequent variations.
The image contains a segment of a document with a BNF (Backus-Naur Form) grammar for a Corporate Address. The BNF rules define how to construct a Corporate Address using phrases for Department, Organisation, Building, Road, Postal Town, and Newline. The flattened grammar then presents the same information in a more compact form. The document also includes a description of how to flatten all the components into a single entity.
### BNF - Flattened Postal Addresses

**rules**

\[
\text{<Address>} : \text{<RuralAddress>} | \text{<TownAddress>} | \text{<CorporateAddress>}
\]

\[
\text{<RuralAddress>} : \text{<BuildingName>} \text{<Newline>} \text{<Village>} \text{<Newline>}
\text{<PostalTown>} \text{<Newline>} \text{<Postcode>}
\]

\[
\text{<TownAddress>} : \text{<Building>} \text{<Road>} \text{<Newline>} \text{<District>} \text{<Newline>}
\text{<PostalTown>} \text{<Newline>} \text{<Postcode>} | \\
\text{<Building>} \text{<Road>} \text{<Newline>}
\text{<PostalTown>} \text{<Newline>} \text{<Postcode>}
\]

\[
\text{<CorporateAddress>} : \text{<Department>} \text{<Newline>} \text{<Organisation>} \text{<Newline>}
\text{<Building>} \text{<Road>} \text{<Newline>}
\text{<PostalTown>} \text{<Newline>} \text{<Postcode>} | \\
\text{<Department>} \text{<Newline>} \text{<Organisation>} \text{<Newline>}
\text{<Building>} \text{<Newline>} \text{<Newline>} \text{<Postcode>} | \\
\text{<Organisation>} \text{<Newline>} \text{<Building>} \text{<Road>} \text{<Newline>}
\text{<PostalTown>} \text{<Newline>} \text{<Postcode>}
\]

\[
\text{<Village>} : \text{<Phrase>}
\]

\[
\text{<Road>} : \text{<Phrase>}
\]

\[
\text{<District>} : \text{<Phrase>}
\]

\[
\text{<Department>} : \text{<Phrase>}
\]

\[
\text{<Organisation>} : \text{<Phrase>}
\]

\[
\text{<Building>} : \text{<Number>} | \\
\text{<Number>} - \text{<Number>} | \\
\text{<Number>} \text{<Letter>} | \\
\text{<Number>} \text{<BuildingName>} | \\
\text{<Number>} \text{<Letter>} \text{<BuildingName>} | \\
\text{<Number>} - \text{<Number>} \text{<BuildingName>} | \\
\text{<BuildingName>}
\]

\[
\text{<BuildingName>} : \text{<Phrase>}
\]

\[
\text{<PostalTown>} : \text{<UpperCasePhrase>}
\]

\[
\text{<Postcode>} : \text{<Letter>} \text{<Letter>} \text{<Digit>} \\
\text{<Space>} \text{<Digit>} \text{<Letter>} \text{<Letter>} | \\
\text{<Letter>} \text{<Digit>} \text{<Digit>} \\
\text{<Space>} \text{<Digit>} \text{<Letter>} \text{<Letter>} | \\
\text{<Letter>} \text{<Letter>} \text{<Digit>} \text{<Digit>} \\
\text{<Space>} \text{<Digit>} \text{<Letter>} \text{<Letter>} \\
\text{<Letter>} \text{<Letter>} \text{<Digit>} \text{<Digit>} \text{<Letter>} \text{<Letter>}
\]

\[
\text{<Phrase>} : \text{<Letter>} | \text{<Letter>} \text{<Phrase>} | \\
\text{<Punctuation>} \text{<Letter>} \text{<Phrase>} | \\
\text{<Phrase>} \text{<Punctuation>}
\]

\[
\text{<UpperCasePhrase>} : \text{<UpperCase>} | \text{<UpperCase>} \text{<UpperCasePhrase>} | \\
\text{<Punctuation>} \text{<UpperCase>} \text{<UpperCasePhrase>} | \\
\text{<UpperCasePhrase>} \text{<Punctuation>}
\]

\[
\text{<Punctuation>} : \text{<Symbol>} | \text{<Symbol>} \text{<Symbol>} | \\
\text{<Symbol>} \text{<Symbol>} \text{<Symbol>} | \text{<Symbol>}
\]

\[
\text{<Symbol>} : \text{<Case>} | \text{<UpperCase>}
\]

\[
\text{<Case>} : \text{<Letter>} \\
\text{<UpperCase>} : \text{<Letter>} \text{<UpperCase>}
\]
CHAPTER 9. SYNTAX AND GRAMMARS

Derivations

Using these rules we can derive the postcode SA2 8PP by the following sequence of one-step reductions:

\[
\text{<Postcode> } \Rightarrow \text{ <Letter> <Letter> <Digit> <Space> <Digit> <Letter> <Letter>}
\]
\[
\Rightarrow S \text{ <Letter> <Digit> <Space> <Digit> <Letter> <Letter>}
\]
\[
\Rightarrow \text{ SA <Digit> <Space> <Digit> <Letter> <Letter>}
\]
\[
\Rightarrow \text{ SA2 <Space> <Digit> <Letter> <Letter>}
\]
\[
\Rightarrow \text{ SA2 <Digit> <Letter> <Letter>}
\]
\[
\Rightarrow \text{ SA2 8 <Letter> <Letter>}
\]
\[
\Rightarrow \text{ SA2 8P <Letter>}
\]
\[
\Rightarrow \text{ SA2 8PP}
\]

We can use this derivation in deriving a whole address. For example,

\[
\text{<Address> } \Rightarrow \text{ <CorporateAddress>}
\]
\[
\Rightarrow \text{ <Department> <Newline> <Organisation> <Newline>}
\]
\[
\text{<Road> <Newline> <PostalTown> <Newline> <Postcode>}
\]
\[
\Rightarrow^* \text{ <Department>}
\]
\[
\text{<Organisation>}
\]
\[
\text{<Road>}
\]
\[
\text{<PostalTown>}
\]
\[
\text{<Postcode>}
\]
\[
\Rightarrow^* \text{ Department of Computer Science}
\]
\[
\text{University of Wales Swansea}
\]
\[
\text{Singleton Park}
\]
\[
\text{SWANSEA}
\]
\[
\text{SA2 8PP}
\]

Non-Standard Formats

As we have seen, these rules allow us to generate an address, but they do restrict the format to the official postal template, which may or may not be desirable.

The precise details of a postal address matter, unfortunately. Clearly, different countries have different formats. Indeed, layouts can be dictated by the manufacturers of automatic sorting machines. For example, the once elegant

Department of Computer Science,
University of Wales, Swansea,
Singleton Park,
Swansea, SA2 8PP,
Wales

with the use of punctuation marks as information separators is no longer in fashion.
9.4.2 World Wide Web Addresses

Internet World Wide Web (web) pages are identified by an address that are used by the Hyper-text Transfer Protocol (HTTP). One of the strengths of HTTP (as opposed to other internet protocols such as FTP) is that hyperlinks abstract addresses away from the user. Addresses, though, are the means of accessing web pages, either indirectly by following a hyperlink, or typed directly.

A typical home page of a business or organisation is constructed as shown in Figure 9.8a. This is the name of the host or server on which the file is stored. There are of course, normally many web pages located on a particular host. These files are again distinguished from each other by names. The storage (and hence retrieval) of these files may be organised by directories. Thus, this gives rise to another form of web address as shown in Figure 9.8b(i) and b(ii).

The third type of web address is one designed to allow data to be passed back and forth from the user to the server. This is used for example in searching for information or form filling, particularly when performed over a series of different web pages. This is shown in Figure 9.8c.

(a) \texttt{http://www.apublisher.co.uk}

\hspace{1cm} host

(b) \texttt{http://www.apublisher.co.uk/books}

\hspace{1cm} host \hspace{1cm} path

(i) \texttt{http://www.apublisher.co.uk/books/computing.html}

\hspace{1cm} host \hspace{1cm} path

(ii) \texttt{http://www.apublisher.co.uk/search?data+syntax+semantics}

\hspace{1cm} host \hspace{1cm} search

Figure 9.8: Internet web page address construction.

Thus, in defining a grammar for HTTP addresses, we need to combine descriptions of hosts, paths and query strings:

\begin{verbatim}
bnf  HTTP Addresses
import Host, Path, QueryString
rules
<HTTP Address> ::= http: // <Host> | http: // <Host> / <Path> | http: // <Host> / <Path> ? <QueryString>
\end{verbatim}
A picture of the construction is shown in Figure 9.9.

![Diagram](image)

Figure 9.9: Structure of http addresses.

**Hosts**

Hosts are usually identified by a name. There are other means though. When a host address is given as a name, it is resolved to a numerical address known as an Internet Protocol (IP) address, by the internet domain name service. Thus, an alternative means of identifying a host is to enter the IP address of the host directly.

The retrieval of web pages is just one service that may be provided by a host. Therefore, when requesting a web page, the host needs to be informed that this is the service required. This is achieved through the use of port numbers. Web services are associated with a particular default port number, so it does not need to be explicitly stated. However, in circumstances where a web service is using a non-standard port number, it needs to be included as part of the address; see, for example, Comer [1995] for further details.

```plaintext
bnf  Host
import  String

rules
<Host> ::= <HostName> | <IPAddress> | <HostName> : <Port> | <IPAddress> : <Port>
<HostName> ::= <Letter> <String> | <Letter> <String>. <HostName>
<IPAddress> ::= <Digits>. <Digits>. <Digits>. <Digits>
<Port> ::= <Number>
```

**Paths**

A path determines the location of a file on a host.
bnf  Path

import String

rules
<Path> ::= <Segment> |
    <Segment> / |
    <Segment> / <Path>
<Segment> ::= <String>

If the path given determines a directory on the host, rather than a specific file, then a
default file for that directory is returned. In fact corporate home page addresses are simply an
alternative for specifying the default file located on the host. For example, an address

http://www.apublisher.co.uk/books

actually locates the file:

http://www.apublisher.co.uk/books/index.html

or

http://www.apublisher.co.uk/books/default.htm

depending on the server used.

Data Transfer

A web page is simply a file that is located on a server. However, some interactivity between
a user and a site can be created by the host creating files in response to a user's actions. For
example, a host may create a one-off temporary file that is removed from the host, or may
assemble a response “on-the-fly” in reply to a user query.

For example, suppose we want to locate web pages containing the words “data”, “syntax”,
and “semantics”. Using a search engine located at http://www.apublisher.co.uk, the act of
submitting a query may be implemented by the browser sending the string formed by append-
ing “search?q=data+syntax+semantics” to “http://www.apublisher.co.uk”. When the server
receives this string, it executes the program search on the input parameters data, syntax and
semantics. The output of the program, in the form of a list of hypertext references, is then
returned directly to the user.
Strings

Finally, we need to define the text that can be written in an address. An address is composed of a string of symbols which can be letters, digits or certain punctuation marks. Some symbols, such as spaces and letters from non-English alphabets cannot appear directly in a string, but instead are encoded numerically in hexadecimal notation. For example, a file

Data Syntax and Semantics

located on a server http://www-compsci.swan.ac.uk is addressed as

http://www-compsci.swan.ac.uk/Data%20Syntax%20and%20Semantics

where % indicates that the next number is to be interpreted as the hexadecimal notation of a character, and 20 is the hexadecimal notation for the space character.

Flattened Grammar for HTTP addresses

If we take the individual component grammars that we constructed for HTTP addresses and flatten them into a single entity, we get:
9.4. WHAT IS AN ADDRESS?

bnf  Flattened HTTP Addresses

rules
<HTTP Address> ::= http: // <Host> |
                    http: // <Host> / <Path> |
                    http: // <Host> / <Path> ? <QueryString>
<Path> ::= <Segment> |
          <Segment> / |
          <Segment> / <Path>
<Segment> ::= <String>
<QueryString> ::= <String> | <String> + <QueryString>
<String> ::= <Letter> | <Digit> | <Punctuation> |
            <Letter> <String> | <Digit> <String> |
            <Punctuation> <String>
<Punctuation> ::= $ | - | _ | . | & | + | -
                ! | * | " | ' | ( ) | , | % <Hex> <Hex>
<Hex> ::= <Digit> | a | b | c | d | e | f | A | B | C | D | E | F
<Letter> ::= <LowerCase> | <UpperCase>
<LowerCase> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s |
               t | u | v | w | x | y | z
<UpperCase> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
               Q | R | S | T | U | V | W | X | Y | Z
<Number> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <Digit> <Digit>
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Digits> ::= <Digit> | <Digit> <Digits>
Exercises for Chapter 9

1. Give a formula for the number of words of length at most $n$ with an alphabet with $m$ elements.

2. Write down some examples of the binary numbers produced by the grammar of Examples 9.2.2(1). What kind of binary numbers are they?

3. Give a grammar to define the language

$$ \{ \text{please}^n \text{thank-you}^n \mid n \geq 1 \}.$$ 

4. Write out the grammar $G^{\text{Arithmetic Expressions}}$ of Section 9.3.2 using the display format but without importing any other grammars.

5. Give a grammar to define the language $L = \{a^n b^{n+1} \mid n \geq 1 \}$.

6. Using the grammar $G^{ab}$ of Examples 9.2.2(2), prove that $L(G^{ab}) = \{a^n b^n \mid n \geq 1 \}$.

7. How many non-terminals, or variables, and production rules are there in the grammar for WP?

8. Derive the following program:

```
begin
  a := 1
end
```

9. Derive the following program:

```
begin
  a := 1 ;
  while a <> 0 do
    a := a +1
  od
end
```

How many steps are involved in the derivation?

10. Let $G_1 = (T_1, N_1, S_1, P_1)$ and $G_2 = (T_2, N_2, S_2, P_2)$ be two grammars. Define $G_2$ to be a subgrammar of $G_1$ if

$$ T_2 \subseteq T_1, \quad N_2 \subseteq N_1, \quad \text{and} \quad P_2 \subseteq P_1. $$

Show that if $G_2$ is a subgrammar of $G_1$ then

$$ L(G_2) \subseteq L(G_1). $$
Assignment for Chapter 9

This assignment charts the rôle of addresses in some uses of the internet.
HTTP is but one form of internet protocol which uses addresses that are examples of Uniform Resource Locators (URLs). Other examples of protocols that use URLs are File Transfer Protocols (FTP) and the Gopher Protocol. The World Wide Web (W3) Consortium publish a formal definition of URLs, using an extended form of BNF, for various protocols at http://www.w3.org

Develop a modular grammar for URLs.

Develop a modular grammar for file names and paths for an operating system of your choice, such as UNIX, MacOS or Windows.

Write an account of the transformations of addresses that are involved in fetching a file using the HTTP protocol. Include an explanation of IP addresses, exceptional behaviour and its error messages.
Chapter 10

Languages for Interfaces, Specifications and Programs

In the previous chapter we introduced the basic ideas of formal language and grammar and applied them in the specification of the syntax of addressing systems. In this chapter we will consider applications of these basic ideas to programming languages. The methods are largely the same — modular grammars written in BNF — but the examples are much more complicated. One new method we introduce is the technique of defining an extended syntax by a syntactic transformation that reduces it to kernel syntax. We will specify two interface definition languages and three programming languages.

In Section 10.1, we begin by specifying an interface definition language for data types by defining a grammar for the language of

[signatures.]

The formal definition enables us to examine more closely some syntactic properties of signatures, which are typical of many declarations in programming languages. In Section 10.2, the interface declaration language is made modular by adding a construct import that allows a signature to refer to other signatures in its definition. We have seen examples of the use of this construct in defining examples of signatures earlier on; now we will analyse and specify the construct in general. We give a second interface definition language for data types by defining the grammar for the language [signatures with imports.]

In Section 10.3, we examine the import construct. In simple terms, it is a convenient notation for defining the standard signatures of algebras. Thus, we view the second signature language as an extension of the first signature language. To define the meaning of the import construct we will define a syntactic transformation that tries to reduce the extended syntax (signatures with imports) to the kernel syntax (signatures without imports). To do this we assume there is a library of signatures, called a repository, and an operation on the signatures in the library that substitutes actual signatures in place of references to signatures. This syntactic transformation is called flattening.

In Section 10.5, we begin the specification of the syntax of the while language with the first of three steps. We give a definition of the
**while** language for computing on the data type of natural numbers.

In Section 10.6, we generalise this grammar to define the

**while** language for computing on any chosen, fixed data type.

From this we can derive grammars for the definition of **while** programs on other choices of data types modelled by algebras, e.g., strings and real numbers. Each data type has its own grammar for **while** programs. In the third stage we create a single grammar that defines **while** programs for all data types. In Section 10.7, by adapting and combining this grammar with the grammar for signatures in Section 10.2 we define the

**while** language for computing on all data types.

Here the grammar generates the signature and then **while** programs. We answer questions of how to extend the **while** language with other features such as arrays, variable declarations, and so on.

In this part we are developing the essentials of the theory of syntax in stages. We have chosen to accumulate a large number of examples, before beginning the mathematical theory in the next chapter. The grammars used in Chapter 9 and here are of a special kind, called *context-free grammars*. In the next chapter, we introduce the mathematical theory of context-free grammars. This elegant subject enables us to understand the uses and limitations of these practically important grammars. Later, in Chapter 12, we give a second theory of syntax that views syntax more abstractly. It is based on using algebras to model data types of syntax. A programming language is specified by an algebra whose operations create new programs from old.

### 10.1 Interface Definition Languages

A signature is a syntactic interface to a data type. It is purely a piece of syntax, and hence we can form the language

\[
\text{Signature}
\]

of

*a simple interface definition language for data types.*

We use this language in Section 10.7 to consider

*a **while** language \( WP = \bigcup_{\Sigma \in \text{Signature}} WP(\Sigma) \) for computing on all data types.*

### 10.1.1 Mathematical Definition of Signatures

In Chapter 4, we gave a mathematical definition of a signature and illustrated the idea with some examples. First, recall the mathematical definition of a signature \( \Sigma \) from Section ??:
signature $\Sigma$
sorts $\ldots, s, \ldots$
constants $\ldots, c : \rightarrow s, \ldots$
operations $\ldots, f : s(1) \times \cdots \times s(n) \rightarrow s, \ldots$
endsig

where we require two properties: the Sort Declaration Property and the Unique Naming Property.

**Definition (Sort Declaration Property)** A signature has the Sort Declaration Property if:

(i) each sort $s$ in a constant declaration $c : \rightarrow s$ appears in the sort declaration; and

(ii) each sort $s(1), \ldots, s(n), s$ in a function declaration $f : s(1) \times \cdots \times s(n) \rightarrow s$ appears in the sort declaration.

**Definition (Unique Naming Property)** A signature has the Unique Naming Property if each constant or function name appears at most once in the signature.

For example, the signature for Peano Arithmetic in Section 4.1.2 satisfies each of these properties. Shortly, we will see examples of “signatures” — strings accepted by our grammar — where the properties fail to hold.

A more concrete type of definition is needed based on appropriate identifiers for sorts, elements and sets.

Thus, in defining a syntax for signatures a number of choices and decisions will be made.

### 10.1.2 A Simple Interface Definition Language for Data Types

Let us consider how we can construct a grammar to model a language for signatures.

We first model signatures without imports, then extend our model in a simple manner to include imports.

**Structure**

We define a signature without imports in terms of the syntactic categories of the name identifier, sort, constant and operation symbols.
#%  

```
bnf    Signature

import Names, Sorts, Constants, Operations

rules
<Signature> ::= signature <Name> <Newline> <Sorts> <Newline>
               <Constants> <Newline> <Operations> <Newline> endsig
```

**Names**

We choose, somewhat arbitrarily, to define a name as a string of letters. Note that we define both individual names and non-empty lists of names.

```
bnf Names

import Letter

rules
<Names> ::= <Name> | <Name>, <Names>
<Name> ::= <Letter> | <Letter> <Space> <Name> | <Letter> <Name>
```

**Sorts**

We define the set of sort symbols to be lists of names. Note that we ensure the sort set is non-empty.

```
bnf Sorts

import Names

rules
<Sorts> ::= sorts <SortList>
<SortList> ::= <Sort> | <Sort>, <SortList> | <Sort> <Newline> <SortList>
<Sort> ::= <Name>
```

**Constants**

We declare a set of constant symbols as a list of names and type declarations. Thus, each constant may have an individual type declaration, or we may group together constants that are
of the same sort.

\[
\text{bnf} \quad \text{Constants}
\]

\[
\text{import} \quad \text{Sorts, Names}
\]

\[
\text{rules}
\]

\[
<\text{Constants}> :::= \text{constants} <\text{ConstantList}> | \\
\quad \text{constants}
\]

\[
<\text{ConstantList}> :::= <\text{Constant}> | <\text{Constant}> \text{<Newline>} <\text{ConstantList}>
\]

\[
<\text{Constant}> :::= <\text{Names}> : <\text{Sort}>
\]

\section*{Operations}

Similarly with operation symbols, each operation is a name or names and a type declaration. Note that an operation must have at least one sort in its domain.

\[
\text{bnf} \quad \text{Operations}
\]

\[
\text{import} \quad \text{Sorts, Names}
\]

\[
\text{rules}
\]

\[
<\text{Operations}> :::= \text{operations} <\text{OperationList}> | \\
\quad \text{operations}
\]

\[
<\text{OperationList}> :::= <\text{Operation}> | <\text{Operation}> \text{<Newline>} <\text{OperationList}>
\]

\[
<\text{Operation}> :::= <\text{Names}> : <\text{DomainSort}> \to <\text{Sort}>
\]

\[
<\text{DomainSort}> :::= <\text{Sort}> | <\text{Sort}> \times <\text{DomainSort}>
\]

\section*{Flattened Version}

If we map out the dependencies between the grammars that we have defined to construct our language of signatures, we get the architecture shown in Figure 10.1.

If we unfold this architecture by substituting the appropriate grammars that we imported, we get the single flattened grammar:
buf \textit{Flattened Signature}

\textbf{rules}

\begin{itemize}
\item \texttt{<Signature>} ::= \texttt{signature <Name>} <Newline> <Sorts> <Newline> \texttt{<Operations>} <Newline> \texttt{endsig}
\item \texttt{<Names>} ::= \texttt{<Name>} | \texttt{<Name>}, \texttt{<Names>}
\item \texttt{<Name>} ::= \texttt{<Letter>} | \texttt{<Letter>} <Space> \texttt{<Name>} | \texttt{<Letter>} \texttt{<Name>}
\item \texttt{<Sorts>} ::= \texttt{sorts <SortList>} <Newline>
\item \texttt{<SortList>} ::= \texttt{<Sort>} | \texttt{<Sort>}, \texttt{<SortList>} | \texttt{<Sort>} <Newline> \texttt{<SortList>}
\item \texttt{<Sort>} ::= \texttt{<Name>}
\item \texttt{<Constants>} ::= \texttt{constants <ConstantList>} <Newline> | \texttt{constants <Newline>}
\item \texttt{<ConstantList>} ::= \texttt{<Constant>} | \texttt{<Constant> <Newline> <ConstantList>}
\item \texttt{<Constant>} ::= \texttt{<Name>} : \texttt{<Sort>}
\item \texttt{<Operations>} ::= \texttt{operations <OperationList>} <Newline> | \texttt{operations <Newline>}
\item \texttt{<OperationList>} ::= \texttt{<Operation>} | \texttt{<Operation> <Newline> <OperationList>}
\item \texttt{<Operation>} ::= \texttt{<Names>} : \texttt{<DomainSort>} \rightarrow \texttt{<Sort>}
\item \texttt{<DomainSort>} ::= \texttt{<Sort>} | \texttt{<Sort> x <DomainSort>}
\item \texttt{<Letter>} ::= \texttt{<LowerCase>} | \texttt{<UpperCase>}
\item \texttt{<LowerCase>} ::= \texttt{a} | \texttt{b} | \texttt{c} | \texttt{d} | \texttt{e} | \texttt{f} | \texttt{g} | \texttt{h} | \texttt{i} | \texttt{j} | \texttt{k} | \texttt{l} | \texttt{m} | \texttt{n} | \texttt{o} | \texttt{p} | \texttt{q} | \texttt{r} | \texttt{s} | \texttt{t} | \texttt{u} | \texttt{v} | \texttt{w} | \texttt{x} | \texttt{y} | \texttt{z}
\item \texttt{<UpperCase>} ::= \texttt{A} | \texttt{B} | \texttt{C} | \texttt{D} | \texttt{E} | \texttt{F} | \texttt{G} | \texttt{H} | \texttt{I} | \texttt{J} | \texttt{K} | \texttt{L} | \texttt{M} | \texttt{N} | \texttt{O} | \texttt{P} | \texttt{Q} | \texttt{R} | \texttt{S} | \texttt{T} | \texttt{U} | \texttt{V} | \texttt{W} | \texttt{X} | \texttt{Y} | \texttt{Z}
\end{itemize}

\section{Example}

A typical example of a signature in this language is:

\begin{verbatim}
buf signature Foo

Names: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

Constants: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

Operations: operation1(a), operation2(b), operation3(c)

End sig
\end{verbatim}
signature  $\text{natbool}$  
sorts  $\text{nat, bool}$  
constants  $\text{zero : } \rightarrow \text{nat}$  
$\text{true, false : } \rightarrow \text{bool}$  
operations  $\text{succ : } \text{nat} \rightarrow \text{nat}$  
$\text{add : } \text{nat} \times \text{nat} \rightarrow \text{nat}$  
$\text{not : } \text{bool} \rightarrow \text{bool}$  
$\text{and : } \text{bool} \times \text{bool} \rightarrow \text{bool}$  
$\text{eq : } \text{nat} \times \text{nat} \rightarrow \text{bool}$  
$\text{leq : } \text{nat} \times \text{nat} \rightarrow \text{bool}$  
$\text{eq : } \text{bool} \times \text{bool} \rightarrow \text{bool}$  
endsig

However, we can also derive signatures with features that are not desirable.

signature  $\text{nat}$  
sorts  $\text{nat}$  
constants  $\text{zero : } \rightarrow \text{nat}$  
operations  $\text{succ : } \text{nat} \rightarrow \text{nat}$  
$\text{add : } \text{nat} \times \text{nat} \rightarrow \text{nat}$  
$\text{eq : } \text{nat} \times \text{nat} \rightarrow \text{bool}$  
endsig

It is easy to derive signatures with the unwanted property that a sort (e.g., $\text{bool}$) appears in a function declaration but does not appear in the sort declaration. The question arises:

*Can we adapt the grammar to restore the Sort Declaration Property?*

Later we will meet a similar problem concerning variable declarations in $\text{while}$ programs. We will develop techniques to show that we have reached a limitation of these simple (context-free) grammars: the Sort Declaration Property cannot be defined by context-free rules.

### 10.2 A Modular Interface Definition Language for Data Types

In Section 10.1, we considered a language for the design of simple interfaces. Our language had no imports. Here, we extend this simple language to produce a language that we can use for
defining interfaces in a modular manner.
   The meaning of an interface defined with import cannot be explained without reference
to the interfaces which it imports. Thus, we shall also extend our language ideas to include
the notion of a library or repository of interfaces. The meaning of an interface with respect
to a given repository can then be explained by regarding the import mechanism as a form of
abbreviation if we expand out or flatten the imports of an interface with respect to a repository
of interfaces.

10.2.1 Signatures with Imports

It is a simple operation to add imports to our language for signatures.

Imports

First we define a syntactic category of imports which consist of a non-empty list of names.

```
bnf Imports
   import Names

rules
   <Imports> ::= imports <ImportList> <Newline> | imports <Newline>
   <ImportList> ::= <Name> | <Name> , <ImportList> | <Name> <Newline> <ImportList>
```

Signatures with Optional Imports

Then we define a signature with imports to be either a non-importing signature, or else a sig-
nature with identifier, non-empty import list, sorts, constants and operations.

```
bnf Signature with Imports
   import Imports, Signature

rules
   <Signature> ::= signature <Name> <Newline>
                  <Imports> <Newline> <Sorts> <Newline>
                  <Constants> <Newline> <Operations> <Newline> endsig
```
Flattened Version

Thus, the architecture of our language of signatures with imports can be represented by Figure 10.2.

![Diagram](attachment:image.png)

Figure 10.2: Architecture of signatures with imports.

Flattening out this architecture by substituting the grammars that are imported yields:
Flattened Signature with Imports

rules
<Signature> ::= signature <Name> <Newline>  
<Imports> <Newline> <Sorts> <Newline>  
<Constants> <Newline> <Operations> <Newline> endsig  |  
signature <Name> <Newline> <Sorts> <Newline>  
<Constants> <Newline> <Operations> <Newline> endsig  
<Names> ::= <Name> | <Name> , <Names>  
<Name> ::= <Letter> | <Letter> <Space> <Name> | <Letter> <Name>  
<Imports> ::= imports <ImportList> <Newline> |  
imports <Newline>  
<ImportList> ::= <Name> | <Name> , <ImportList> |  
<Name> <Newline> <ImportList>  
<Sorts> ::= sorts <SortList> <Newline>  
<SortList> ::= <Sort> | <Sort> , <SortList> | <Sort> <Newline> <SortList>  
<Sort> ::= <Name>  
<Constants> ::= constants <ConstantList> <Newline> |  
constants <Newline>  
<ConstantList> ::= <Constant> | <Constant> <Newline> <ConstantList>  
<Constant> ::= <Names> : <Sort>  
<Operations> ::= operations <OperationList> <Newline> |  
operations <Newline>  
<OperationList> ::= <Operation> | <Operation> <Newline> <OperationList>  
<Operation> ::= <Names> : <DomainSort> -> <Sort>  
<DomainSort> ::= <Sort> | <Sort> x <DomainSort>  
<Letter> ::= <LowerCase> | <UpperCase>  
<LowerCase> ::= a b c d e f g h i j k l m n o p q r s t |  
u v w x y z  
<UpperCase> ::= A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

10.3 Flattening

Now we have a language of signatures, we can define operations on the elements of our language, i.e., we can define operations on signatures. We consider here how we can define the flattening process on signatures.

We have two types of signatures, one with imports, which we generate using the rule

<Signature> ::= signature <Name> <Newline>  
<Imports> <Newline> <Sorts> <Newline>  
<Constants> <Newline> <Operations> <Newline> endsig
and one without, which we generate using the rule
\[
\text{<Signature>} ::= \text{signature } \langle\text{Name}\rangle \langle\text{Newline}\rangle \langle\text{Sorts}\rangle \langle\text{Newline}\rangle \langle\text{Constants}\rangle \langle\text{Newline}\rangle \langle\text{Operations}\rangle \langle\text{Newline}\rangle \text{endsig}
\]

When we flatten a signature, we try to reduce a signature with imports to a signature without imports.

### 10.3.1 Repositories

Suppose we have a signature \( \Sigma \) which imports the signatures:

\[
\ldots, \Sigma_{\text{import}}, \ldots
\]

Then, we need each of the signatures \( \ldots, \Sigma_{\text{import}}, \ldots \) to be able to flatten \( \Sigma \).

**Definition (Repository)** A collection of signatures is called a repository.

So, to attempt to flatten a signature, we need a repository of signatures. We extend our language of signatures to include repositories:

```plaintext
bnf Repository
import Signature with Imports
rules
<Repository> ::= repository <Name>
      <Contents>
      endrepository
<Contents> contains <RepositoryList> | contains
<RepositoryList> ::= <Signature> | <Signature>,<RepositoryList>
```

Given a repository, we shall need a function

\[
\text{Extract} : \text{Name} \times \text{Repository} \rightarrow \text{Signature with Imports}
\]

so that \( \text{Extract}(n,R) \) will pick out an interface with name \( n \) from a repository \( R \).

Of course, there is no guarantee that the repository will be suitable for flattening a particular signature. There could be many problems, the most obvious of which is that the repository does not contain all of the signatures that we need to flatten a signature. So flattening is a function

**Flatten : Signature with Imports \times Repository \rightarrow Signature with Imports**

If all goes well, and we can flatten a signature \( \Sigma \) then we will be left with a signature

\[
\text{Flatten}(\Sigma) \in \text{Signature} \subseteq \text{Signature with Imports};
\]
but if there are problems, then we will be left with a signature

\[ Flatten(\Sigma) \notin \text{Signature}. \]

### 10.3.2 Dependency Graphs

Given a repository of signatures, we can determine the import relationship (if any) between them.

**Definition (Dependency Graph)** A *dependency graph* of a repository is a graph where:

1. the nodes of the graph are the signatures of the repository;
2. there is an edge in the graph \( I \rightarrow J \), if, and only if, there is a signature \( I \) in the repository which imports a signature \( J \), and \( J \) is in the repository.

When we build a dependency graph for a repository, we may find that there may be other problems when we try to flatten a signature using the repository. For example, we may find that the repository does not contain a signature that is listed as an import in another. We may find that two interfaces are mutually dependent on one another, which will show up as a cycle in the graph.

### 10.3.3 Flattening Algorithm

Given a repository and a signature with imports, we can attempt trace its dependencies by traversing the dependency graph. In order to flatten a signature, we need to combine the signatures that it depends on. So how do we combine signatures? Let us suppose we have an operation

\[ \text{Expand} : \text{Signature with Imports} \times \text{Signature with Imports} \rightarrow \text{Signature with Imports} \]

so that

\[ \text{Expand}(\Sigma, \Sigma') \]

will create a signature with the same name as \( \Sigma \), and will join the imports, sorts, constants and operations of the signatures \( \Sigma \) and \( \Sigma' \).

If we represent signatures using records then an algorithm for performing flattening is:
Informal algorithm to perform flattening

(* Given a signature \( \Sigma \) as input, create its flattened version \( \text{Flattened} \). *)

(* Create a copy \( \text{Flattened} \) of \( \Sigma \), and rename it as \( \text{Flattened} \ \Sigma \). *)

\( \text{Flattened} := \Sigma; \)

\( \text{Flattened.Name} := \text{concat}(\text{"Flattened"}, \Sigma.\text{Name}); \)

(* Pick out the imports \( I \) of \( \Sigma \). *)

\( I := \Sigma.\text{Imports}; \)

(* Whilst there is an import in \( \text{Flattened} \). *)

\text{while} \( I \neq \emptyset \) \text{do}

(* Pick an import \( i \). *)

\text{choose} \( i \in I \)

(* Replace \( i \) with the imports that \( i \) depends on. *)

\( I := I - \{i\} \cup \text{Extract}(i,R).\text{Imports}; \)

(* Update \( \text{Flattened} \)'s import list. *)

\( \text{Flattened.Imports} := I; \)

(* Flatten \( \text{Flattened} \) with the signature named \( i \) in the repository \( R \). *)

\( \text{Flattened} := \text{Expand}(\text{Flattened}, \text{Extract}(i,R)) \)

\text{od}

10.3.4 Example

Suppose we have a repository \( R \) with the signatures

\[ \text{Integers, Bool, Integer Group, Integer Ring, Integer Group with Booleans, Integer Ring with Booleans.} \]

as shown in Figure 10.3.

If we flatten the signature \( \text{Integer Group with Booleans} \), we get the signature

\[ \]
Figure 10.3: Dependency graph.
10.4 Languages for Data Type Specifications

A data type can be specified using

(i) an interface; and

(ii) a list of properties of the operations declared in the interface.

In Chapter 5, and later chapters, we explained this method and considered many examples. We will extend our interface definition language into a language for writing axiomatic specifications for data types. As an exercise in the design and definition of syntax it is informative and not too difficult.

10.4.1 Specifications

An interface is modelled by a signature, and the list of properties by an axiomatic theory. Thus, seen as an independent component, a specification might take the form:

```
spec
  signature Name
  sorts ... s, ...
  constants ... c : s, ...
  operations ... f : s(1) \times ... \times s(n) \rightarrow s, ...
  endsig
  axioms Name
    : a
    : ...
  endaxioms
endspec
```

For example, our favourite example of commutative rings would become:

Example
spec
signature CRing
sorts ring
constants 0, 1 : → ring
operations + : ring × ring → ring
− : ring → ring
. : ring × ring → ring
endsig
axioms CRing
(∀x)(∀y)(∀z)[(x + y) + z = x + (y + z)]
(∀x)(∀y)[x + y = y + x]
(∀x)[x + 0 = x]
(∀x)[x + (−x) = 0]
(∀x)(∀y)(∀z)[(x.y).z = x.(y.z)]
(∀x)(∀y)[x.y = y.x]
(∀x)[x.1 = x]
(∀x)(∀y)(∀z)[x.(y + z) = x.y + x.z]

endaxioms
endspec

We wish to build a grammar from which we can derive specifications. We have already
developed a grammar for signatures in Section 10.1 (we have even extended it with import
in Section 10.2). Thus, the problem is to extend the signature grammar with rules to specify
axioms.

What are these axioms? They are properties of the constants and operations that appear
in the signature. Looking back over examples we see some variety.

In the specification of commutative rings, the axioms are all very similar; the commutativity axiom

(∀x)(∀y)[x + y = y + x]

is typical. Each axiom is an equation that is forced to be true of all elements. However, we
did meet, in Chapter 5, the integral domain property for certain commutative rings which was
a little more complicated:

(∀x)(∀y)[x.y = 0 ⇒ x = 0 ∨ y = 0].

Here, the axiom involves the logical connectives

⇒ and ∨

of implication and disjunction, respectively. Another important property is the division axiom
in the specification of fields:

(∀x)[x ≠ 0 ⇒ x.x⁻¹ = x⁻¹.x = 1].
This involves the logical connectives

\[ \neg \quad \text{and} \quad \Rightarrow \]

of negation and implication, respectively.

The induction axiom in Dedekind’s characterisation of the natural numbers is more complicated again:

\[ (\forall X)[0 \in X \land (\forall x)[x \in X \Rightarrow succ(x) \in X] \Rightarrow \forall x[x \in X]] \]

This axiom involves the logical connectives

\[ \land \quad \text{and} \quad \Rightarrow \]

of conjunction and implication, respectively, and two different types of quantification:

\( \forall x \) quantification over data; and

\( \forall X \) quantification over sets of data.

Note, too, that these logical operators are nested.

The completeness property for ordered fields is even more complex (see the exercises).

These examples suggest that the axioms in our specifications are expressed using

(i) variables;

(ii) constants and operations from a signature;

(iii) logical connectives; and

(iv) quantifiers.

What is their structure? Very roughly speaking, at the heart of each axiom are simple expressions that are made from the constants, operations and tests in a signature and are either true or false. These expressions will be called \textit{atomic expressions}. An axiom is made by putting together atomic expressions using the logical connectives and constraining atomic expressions using quantifiers.

We will now proceed to develop a formal language for axioms. For simplicity, we will not allow quantification over sets, only quantification over data. Because of this restriction of quantification to elements of the data type, the language we develop will contain what will be called

\textit{first-order axiomatic specifications}

If quantification over sets were allowed, then it would contain what would be called

\textit{second-order axiomatic specifications}.

10.4.2 Languages for First-Order Specifications

We will develop the grammar for our language of first-order specifications in a top-down and modular way. The structure of the grammar is depicted in Figure 10.4.

A specification consists of a signature and some axioms.
Figure 10.4: Modular grammar (and hence algebraic) structure for first order formulae specifications.

\[
\text{bnf Specification} \\
\text{import Signature} \\
\text{rules} \\
<\text{specification}> ::= \text{spec} <\text{signature}> <\text{newline}> <\text{axioms}> \text{endspec}
\]

Next, we define lists of axioms.

\[
\text{bnf Axioms} \\
\text{import FirstOrderFormula} \\
\text{rules} \\
<\text{axioms}> ::= \text{axioms} <\text{axiom list}> \text{endaxioms} | \\
\text{axioms endaxioms} \\
<\text{axiom list}> ::= <\text{axiom}> <\text{newline}> | \\
<\text{axiom}> <\text{newline}> <\text{axiom list}> \\
<\text{axiom}> ::= <\text{formula}>
\]

Each axiom is a first-order formula. A first-order formula is

(i) an atomic formula,
(ii) a quantified formula, or
(iii) a connective formula.
A connective formula is a formula to which the operations of

(i) negation,
(ii) disjunction,
(iii) conjunction, or
(iv) implication.

A quantified formula is a formula in which some variables are

(i) universally, or
(ii) existentially quantified.

```
bnf  FirstOrderFormula

import  AtomicFormula

rules
<formula> ::= <atomic_formula> | 
              <quantified_formula> | 
              <connective_formula>

<connective_formula> ::= ¬ <formula> | 
                       <formula> ∧ <formula> | 
                       <formula> ∨ <formula> | 
                       <formula> ⇒ <formula>

<quantified_formula> ::= ( <quantifier><variable>)[ <formula> ]

<quantifier> ::= ∀ | ∃
```

An atomic formula is

(i) a truth value of true or false; or
(ii) formed from the application of a Boolean-valued operation from the signature to a list of terms.
bnf \ \text{AtomicFormula}

import \ Term

rules
\begin{align*}
\text{<atomic\_formula>} & \ ::= \text{<truth\_values>} \mid \\
& \quad \text{<relation\_application>} \\
\text{<truth\_values>} & \ ::= \text{true} \mid \text{false} \\\n\text{<relation\_application>} & \ ::= \text{<Boolean\_operation> (<term\_list>)}
\end{align*}

A term is

(i) a constant;

(ii) a variable; or

(iii) formed from the application of a function to a list of terms.

bnf \ \text{Term}

import \ Signature

rules
\begin{align*}
\text{<term\_list>} & \ ::= \text{<term>} \mid \\
& \quad \text{<term>, <term\_list>} \\
\text{<term>} & \ ::= \text{<constant>} \mid \\
& \quad \text{<variable>} \mid \\
& \quad \text{<function\_application>} \\
\text{<function\_application>} & \ ::= \text{<function> (<term\_list>)}
\end{align*}

**Flattened Version** If we now assemble the components of the modular grammar we get:
10.4.3 Languages for Equational Specifications

The first order formulae can be classified according to their logical complexity. Perhaps the simplest kind of first order formula is an equation. For example,

\[(x + y) + z = x + (y + z)\]
\[x + y = y + x\]
are equations and, in fact, all the axioms of a commutative ring are equations.

We can classify specifications by classifying their formulae allowed as axioms. An equational specification is a specification with only equational formulae.

We define a grammar for equational specifications. The structure of the grammar is depicted in Figure ??.

Figure 10.5: Modular grammar (and hence algebraic) structure for equational logic specifications.

```
bnf EquationalSpecification
import Signature
rules <equal_specification> ::= spec <signature> <newline> <equal_axioms> endspec
```

An equational axiom is an equational formula.

```
bnf EquationalAxioms
import EquationalFormula
rules
<equal_axioms> ::= axioms <equal_axiom_list> endaxioms | axioms endaxioms
<equal_axiom_list> ::= <equal_axiom> <newline> | <equal_axiom> <newline> <equal_axiom_list>
<equal_axiom> ::= <equal_formula>
```

An equational formula is a universally quantified equation. An equation is formed by applying an equality relation to terms.
\textbf{bnf} \hspace{1em} \textit{EquationalFormula} \\
\textbf{import} \hspace{0.5em} \textit{Term} \\
\textbf{rules} \\
<\text{equal}\_\text{formula}> ::= ( <\text{quantifier}> <\text{variable}> ) [ <\text{equation}> ] \\
<\text{equation}> ::= <\text{term}> = <\text{term}> \\
<\text{quantifier}> ::= \forall \\

\textbf{Flattened Version} \hspace{1em} Assembling the components, we have:

\textbf{bnf} \hspace{1em} \textit{EquationalSpecification} \\
\textbf{import} \hspace{0.5em} \textit{Signature} \\
\textbf{rules} \\
<\text{equal}\_\text{specification}> ::= \textbf{spec} \ <\text{signature}> \ <\text{newline}> \ <\text{equal}\_\text{axioms}> \ \textbf{endspec} \\
<\text{equal}\_\text{axioms}> ::= \textbf{axioms} \ <\text{equal}\_\text{axiom\ list}> \ \textbf{endaxioms} \ | \\
\hspace{1em} \textbf{axioms} \ \textbf{endaxioms} \\
<\text{equal}\_\text{axiom\ list}> ::= <\text{equal}\_\text{axiom}> \ <\text{newline}> \ | \\
\hspace{1em} <\text{equal}\_\text{axiom}> \ <\text{newline}> <\text{equal}\_\text{axiom\ list}> \\
<\text{equal}\_\text{axiom}> ::= <\text{equal}\_\text{formula}> \\
<\text{equal}\_\text{formula}> ::= ( <\text{quantifier}> <\text{variable}> ) [ <\text{equation}> ] \\
<\text{equation}> ::= <\text{term}> = <\text{term}> \\
<\text{quantifier}> ::= \forall \\
<\text{term\ list}> ::= <\text{term}> \ | \\
\hspace{1em} <\text{term}>, <\text{term\ list}> \\
<\text{term}> ::= <\text{constant}> \ | \\
\hspace{1em} <\text{variable}> \ | \\
\hspace{1em} <\text{function\ application}> \\
<\text{function\ application}> ::= <\text{function}> (<\text{term\ list}>)}
10.5 A While Programming Language over the Natural Numbers

We will now develop a series of grammars that will show how we can pin down certain aspects of the syntax of programs and interfaces.

10.5.1 A Grammar for while Programs over Natural Numbers

We shall present a grammar for a while programming language (WP) for computation on natural numbers in this style. We will refine the definition of the language given earlier: we introduce extra features to the language, and enforce precedence between the operations on the natural numbers.

Underlying Data Type

The language we are designing in this section will only be for computing over the natural numbers. In particular, we fix the underlying data type to have a signature of the form:

<table>
<thead>
<tr>
<th>signature</th>
<th>Naturals with Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>nat, Bool</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1 : nat → nat</td>
</tr>
<tr>
<td></td>
<td>true, false : Bool</td>
</tr>
<tr>
<td>operations</td>
<td>+ : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>- : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>* : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>/ : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>mod : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>= : nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>&gt; : nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>&gt;= : nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>&lt;: nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>&lt;= : nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>&lt;&gt; : nat × nat → Bool</td>
</tr>
<tr>
<td></td>
<td>not : Bool → Bool</td>
</tr>
<tr>
<td></td>
<td>or : Bool × Bool → Bool</td>
</tr>
<tr>
<td></td>
<td>and : Bool × Bool → Bool</td>
</tr>
</tbody>
</table>

Programs

Now let us consider the structure of while programs in a top-down manner, as shown in Figure 11.12.
10.5. A WHILE PROGRAMMING LANGUAGE OVER THE NATURAL NUMBERS

Figure 10.6: Architecture of while programs over the natural numbers.

We define a program to consist of a list of individual commands separated by semi-colons. A command will either be concerned with input and output, or it will not.

```
bnf While Programs over Natural Numbers
import Statements, I/O

rules
<while program> ::= begin <command list> end
<command list> ::= <command> | <command list> ; <command>
<command> ::= <statement> | <i/o statement>
```

Input-Output Statements

We have simple input-output mechanisms that will allow us to read values into identifiers and to output the values of identifiers.

```
bnf I/O
import Identifiers

rules
<i/o statement> ::= <read statement> | <write statement>
<read statement> ::= read (<identifier list> )
<write statement> ::= write (<identifier list> )
<identifier list> ::= <identifier> | <identifier list> , <identifier>
```
Statements

Alternatively, statements may be

(i) an assignment

\[ x := e \]

from an identifier \( x \) to a value of an expression \( e \);

(ii) a conditional statement

\[
\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}
\]

which directs the flow of control of the program to the statement \( S_1 \) if the Boolean expression \( b \) evaluates to true, and to the statement \( S_2 \) otherwise;

(iii) an iterative statement

\[
\text{while } b \text{ do } S_0 \text{ od}
\]

which repeatedly executes the statement \( S_0 \) whilst the Boolean expression \( b \) evaluates to true; or

(iv) a null statement

\[
\text{skip}
\]

which has no effect.

<table>
<thead>
<tr>
<th>bnf</th>
<th>\textit{Statements}</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>\textit{Expressions, BooleanExpressions}</td>
</tr>
</tbody>
</table>
| rules | \[
<\text{statement}> ::= \text{<assignment statement>} | \text{<conditional statement>} | \text{<iterative statement>} | \text{<null statement>}
\]
| \[
<\text{assignment statement}> ::= \text{<identifier>} := \text{<expression>}
\]
| \[
<\text{conditional statement}> ::= \text{if } <\text{comparison}> \text{ then } <\text{command list}> \text{ else } <\text{command list}> \text{ fi}
\]
| \[
<\text{iterative statement}> ::= \text{while } <\text{comparison}> \text{ do } <\text{command list}> \text{ od}
\]
| \[
<\text{null statement}> ::= \text{skip}
\] |

Boolean Expressions

Statements are dependent on the tests that we can perform. A comparison is either

(i) true or false;
(iii) a comparison of two expressions; or

(iii) a Boolean combination of comparisons.

We can compare expressions using the relational operators $=, <, >, \geq, \leq$ or $\neq$. Note that these are the operations of the underlying data type *Naturals with Tests* that have a domain type involving *nat* and return type of *Bool*.

We can combine comparisons with the Boolean operations *not*, *and*, and *or*. Again, note that these are the operations of the underlying data type that have a domain type involving only *Bool* and return type of *Bool*.

The rules have been constructed so as to enforce precedence between the operators. In particular, the means by which we have constructed these rules means that *not* binds more tightly than *and*, which in turn means that *and* binds more tightly than *or*. Thus, a Boolean expression

$$b_1 \text{ and not } b_2 \text{ or } b_3$$

can only have the interpretation

$$b_1 \text{ and } (\text{not}(b_2 \text{ or } b_3)).$$

Any other interpretation must be bracketed suitably. This has the advantage that brackets can be omitted in some circumstances.

<table>
<thead>
<tr>
<th>buf</th>
<th>BooleanExpressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>Identifiers</td>
</tr>
<tr>
<td>rules</td>
<td>&lt;comparison&gt; ::= &lt;Boolean expression&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Boolean expression&gt; ::= &lt;Boolean term&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Boolean term&gt; ::= &lt;Boolean factor&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;Boolean factor&gt; ::= &lt;Boolean atom&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;relational operator&gt; ::= =</td>
</tr>
<tr>
<td></td>
<td>&lt;Boolean atom&gt; ::= true</td>
</tr>
</tbody>
</table>

**Expressions**

Both statements and conditional tests are directly dependent on expressions over the signature of the underlying data type *Naturals with Tests*. We can form an expression as

(i) an identifier,
(ii) a number, or

(iii) combining expressions with +, -, *, / or mod.

Again, the operations to combine expressions are those operations of *Naturals with Tests* which have return type *nat*.

The precedence of operators encoded by these rules is that *, / and mod bind more tightly than + and −.

```
bnf  Expressions

import  Identifiers

rules

<expression> ::= <term> | <expression> <adding operator> <term>
<term> ::= <factor> | <term> <multiplying operator> <factor>
<factor> ::= <atom> | (<expression> )
<adding operator> ::= + | -
<multiplying operator> ::= * | / | mod
<atom> ::= <identifier> | <number>
```

**Identifiers**

We construct variables, or identifiers, as strings of letters or digits that start with a letter.

```
bnf  Identifiers

import  Letter, Number

rules

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
```

**Flattened Version**

If we take all the component grammars we have constructed, we can produce the flattened grammar:
### 10.5. A WHILE PROGRAMMING LANGUAGE OVER THE NATURAL NUMBERS

#### bnf  Flattened While Programs over Natural Numbers

**rules**

<while program> ::= begin <command list> end

<command list> ::= <command> | <command list> ; <command>

<command> ::= <statement> | <read statement> | <write statement>

<statement> ::= <assignment statement> | <conditional statement> | <iterative statement> | <null statement>

<assignment statement> ::= <identifier> := <expression>

<conditional statement> ::= if <comparison> then <command list>

else <command list> fi

<iterative statement> ::= while <comparison> do <command list> od

<null statement> ::= skip

<read statement> ::= read (<identifier list> )

<write statement> ::= write (<identifier list> )

<identifier list> ::= <identifier> | <identifier list> , <identifier>

<comparison> ::= <Boolean expression> | <expression> <relational operator> <expression>

<Boolean expression> ::= <Boolean term> | <Boolean expression> or <Boolean term>

<Boolean term> ::= <Boolean factor> | <Boolean term> and <Boolean factor>

<Boolean factor> ::= <Boolean atom> | (<comparison> )| not <Boolean factor>

<expression> ::= <term> | <expression> <adding operator> <term>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<factor> ::= <atom> | ( <expression> )

<adding operator> ::= + | -

<multiplying operator> ::= * | / | mod

<relational operator> ::= = | < | > | >= | <= | <>

<Boolean atom> ::= true | false

<atom> ::= <identifier> | <number>

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

<number> ::= <digit> | <number> <digit>

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

A | B | C | D | E | F | G | H | I | J | K | L | M | N |

O | P | Q | R | S | T | U | V | W | X | Y | Z |

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
10.5.2 Example

We can derive the program

\[
\begin{align*}
& \text{begin} \\
& \quad x := x + 1 \\
& \text{end}
\end{align*}
\]

from the grammar \(G^{\text{while/naturals}}\) as follows:

<while program>

\[
\Rightarrow \text{begin } \text{<command list> } \text{end} \\
\Rightarrow \text{begin } \text{<command> } \text{end} \\
\Rightarrow \text{begin } \text{<statement> } \text{end} \\
\Rightarrow \text{begin } \text{<assignment statement> } \text{end} \\
\Rightarrow \text{begin } \text{<identifier> } := \text{<expression> } \text{end} \\
\Rightarrow \text{begin } \text{<letter> } := \text{<expression> } \text{end} \\
\Rightarrow \text{begin } \text{<letter> } := \text{<expression> } \text{<adding operator> } \text{<term> } \text{end} \\
\Rightarrow \text{begin } x := \text{<expression> } \text{<adding operator> } \text{<term> } \text{end} \\
\Rightarrow \text{begin } x := \text{<term> } \text{<adding operator> } \text{<term> } \text{end} \\
\Rightarrow \text{begin } x := \text{<term> } + \text{<term> } \text{end} \\
\Rightarrow \text{begin } x := \text{<factor> } + \text{<term> } \text{end} \\
\Rightarrow \text{begin } x := \text{<factor> } + \text{<factor> } \text{end} \\
\Rightarrow \text{begin } x := \text{<atom> } + \text{<factor> } \text{end} \\
\Rightarrow \text{begin } x := \text{<atom> } + \text{<atom> } \text{end} \\
\Rightarrow \text{begin } x := \text{<identifier> } + \text{<atom> } \text{end} \\
\Rightarrow \text{begin } x := \text{<identifier> } + \text{<number> } \text{end} \\
\Rightarrow \text{begin } x := \text{<letter> } + \text{<number> } \text{end} \\
\Rightarrow \text{begin } x := \text{<letter> } + \text{<digit> } \text{end} \\
\Rightarrow \text{begin } x := x + \text{<digit> } \text{end} \\
\Rightarrow \text{begin } x := x + 1 \text{ end}
\]

The grammar produces basic \texttt{while} programs over natural numbers. Several extensions come to mind, such as adding control constructs (e.g., \texttt{for}-loops, \texttt{repeat} statements, declarations) and further data types. Recall the discussion in Chapter 1, and see the exercises at the end of this chapter.

10.6 A While Programming Language over a Data Type

In the last sections we gave grammars that modelled the syntax of

\[
a \text{while language } WP(\Sigma_{\text{Naturals with tests}}) \text{ for computing on a data type of natural numbers with fixed signature } \Sigma_{\text{Naturals with tests}}.
\]

To compute on another data type, we need to adapt the grammars to model the syntax of

\[
a \text{while language } WP(\Sigma) \text{ for computing on an arbitrary data type of fixed signature } \Sigma.
\]
This adaptation is informative, for in abstracting from the natural numbers to any data type, we clarify how data and the flow of control are combined in imperative programming. The adaptation can be used to define all sorts of special languages, such as a while language for real number computation, or for computational geometry, or for syntax processing.

### 10.6.1 While Programs for an Arbitrary, Fixed Signature

We can construct a while program grammar which computes over the types and operations specified by an arbitrary fixed signature \( \Sigma \). First, we choose and fix any signature \( \Sigma_{\text{Fixed}} \) containing the Booleans as shown below:

```
signature Fixed
sorts ..., s, ...
        Bool
constants :
c : \rightarrow s
...             \text{true, false :} \rightarrow \text{Bool}
operations :
f : s(1) \times \cdots \times s(n) \rightarrow s
...             \vdots
r : t(1) \times \cdots \times t(m) \rightarrow \text{bool}
...             \text{not :} \text{Bool} \rightarrow \text{Bool}
\text{and :} \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
\text{or :} \text{Bool} \times \text{Bool} \rightarrow \text{Bool}
endsig
```

We use the notation \( f \) and \( r \) to highlight the operations on the data; we use the operations ..., r, ... as tests on the data.

Next, we choose the alphabet and variables. We simply choose names for constants, operations and tests, say:

\[ c \text{ for } c, \quad f \text{ for } f \text{ and } r \text{ for } r; \]

and we remove reference to the operations on natural numbers in Section 10.5.

Thus, we need to adjust the form of expressions and Boolean expressions that we can construct.

We illustrate the structure of the grammar that we produce in Figure 10.7.
Figure 10.7: Architecture of while programs over a fixed signature. Grammars that differ from those for while programs over the natural numbers are indicated by rectangles with non-rounded corners.

**Boolean Expressions**

As before, a Boolean expression can be

(i) true or false, or

(ii) a Boolean combination of comparisons.

But instead of forming comparisons using the relational operators of Naturals with Tests, we now form comparisons using the relational operators

$$\ldots, r, \ldots$$

of Fixed. In addition, we allow identifiers to represent truth values.

```
bnf Boolean Expressions
import Expressions

rules
<Boolean expression> ::= <identifier> | true | false |
not (<Boolean expression>) |
and (<Boolean expression>, <Boolean expression>) |
or (<Boolean expression>, <Boolean expression>) |
\ldots | r (<t(1)-expression>, \ldots, <t(m)-expression>) | \ldots
```

Note that any comparison we build up has to be done in prefix form. For example, we write

- `not(b)` instead of `not b`
- `and(b_1, b_2)` instead of `b_1 and b_2`
- `or(b_1, b_2)` instead of `b_1 or b_2`.

This might seem a retrograde step, but it fits in with the way in which we are forced to write comparisons involving expressions. The problem is that although we have a syntax for the relations

\[ \ldots, r, \ldots \]

on expressions, we do not have any information on the relative priorities of all of these relations.

Although the syntax is more clumsy to write, it does mean though that the structure of the grammar is more transparent.

**Expressions**

The way in which we form expressions over the signature $Fixed$ is analogous to the manner in which we formed expressions over the signature $Naturals$ with Tests.

As before, an expressions can be

(i) an identifier.

But, instead of numbers, we can have

(ii) constants

\[ \ldots, c, \ldots \]

from $Fixed$.

And instead of combining expressions with the operations of $Naturals$ with $Fixed$, we

(iii) combine expressions with the operations

\[ \ldots, f, \ldots \]

from $Fixed$

At this point, our model of expressions has an additional level of complexity not so evident with forming expressions over the two-sorted $Naturals$ with Tests. The reason why we had two categories of expressions — expressions and Boolean expressions — when dealing with $Naturals$ with Tests is because this signature had two sorts $nat$ (which yielded expressions) and $Bool$ (which yielded Boolean expressions).

The signature $Fixed$ though has one or more sorts. We require $Fixed$ to have a sort $Bool$, and to have optionally have additional sorts. Each of these different sorts

\[ \ldots, s, \ldots \]

corresponds to an expression

$s$-expression.

So for each constant

\[ c : \rightarrow s \]

of $Fixed$ of sort $s$, we can form an $s$-expression:

\[ c \]
And for each operation
\[ f : s(1) \times \cdots \times s(n) \to s \]
of \textit{Fixed} of range sort \( s \), we can form an \( s \)-expression
\[ f(e_1, \ldots, e_n) \]
\textit{provided} that we have an \( s(1) \)-expression \( e_1, \ldots \), and an \( s(n) \)-expression \( e_n \).

Thus, the sequence of rules for the nonterminals
\[ \ldots, < s \text{-expression }>, \ldots \]
try to ensure the expressions are correctly sorted in accepted strings, but the identifiers are not sorted.

In general, an expression can be an expression of any type; i.e., an expression is a Boolean expression or any of \( \ldots, s \text{-expression}, \ldots \).

\begin{verbatim}
bnf  Expressions

import Identifiers, Boolean Expressions

rules
<expression> ::= <Boolean expression> |
          \cdots | < s-expression > | \cdots
          \vdots

< s-expression > ::= <identifier> |
          \cdots | c | \cdots
          \cdots | f( < s(1) -expression >, \ldots, < s(n) -expression > ) | \cdots
          \vdots
\end{verbatim}

Again, we are forced into abandoning encoding any precedence rules between operators as we do not know what they should be. As with Boolean expressions though, the structure of expressions is more transparent as a result.

**Flattened Version**

We combine the amended grammars for expressions and Boolean expressions with the component grammars for programs, statements, i/o statements, identifiers, letters and numbers from Section 10.5, to give:
### 10.7 A While Programming Language over all Data Types

In Section 10.6.1, we introduced a technique for producing **while** programming languages which compute over a fixed but arbitrary signature. The expressions in the resulting language are constrained to those which compute using the constants and functions declared in the fixed

---

**Flattened While Programs with Fixed Interface**

<table>
<thead>
<tr>
<th>rule type</th>
<th>notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>While program</td>
<td><code>begin &lt;command list&gt; end</code></td>
</tr>
<tr>
<td>Command list</td>
<td>`&lt;command&gt;</td>
</tr>
<tr>
<td>Command</td>
<td>`&lt;statement&gt;</td>
</tr>
<tr>
<td>Read statement</td>
<td><code>read (&lt;identifier list&gt;)</code></td>
</tr>
<tr>
<td>Write statement</td>
<td><code>write (&lt;identifier list&gt;)</code></td>
</tr>
<tr>
<td>Identifier list</td>
<td>`&lt;identifier&gt;</td>
</tr>
<tr>
<td>Statement</td>
<td>`&lt;assignment statement&gt;</td>
</tr>
<tr>
<td>Conditional</td>
<td><code>if &lt;Boolean expression&gt; then &lt;command list&gt;</code>&lt;br&gt;<code>else &lt;command list&gt; fi</code></td>
</tr>
<tr>
<td>Iterative statement</td>
<td><code>while &lt;Boolean expression&gt; do &lt;command list&gt; od</code></td>
</tr>
<tr>
<td>Null statement</td>
<td><code>skip</code></td>
</tr>
<tr>
<td>Boolean expression</td>
<td>`&lt;identifier&gt;</td>
</tr>
<tr>
<td>Expression</td>
<td>`&lt;Boolean expression&gt;</td>
</tr>
<tr>
<td>S-expression</td>
<td>`&lt;identifier&gt;</td>
</tr>
<tr>
<td>Identifier</td>
<td>`&lt;letter&gt;</td>
</tr>
<tr>
<td>Number</td>
<td>`&lt;digit&gt;</td>
</tr>
<tr>
<td>Letter</td>
<td>`a</td>
</tr>
<tr>
<td>Digit</td>
<td>`0</td>
</tr>
</tbody>
</table>
signature which contains the built-in type \texttt{bool}. In this section we consider a single \texttt{while}
programming language which computes over all possible declared signatures.

The basic approach is to design a grammar which structures strings in its language in
two sections. The first section consists of an interface definition and the second section the
imperative program itself. The intention is that the program section is constrained to compute
over the signature declared in the first section of the string. For example, to compute over
a data type with signature \( \Sigma \), we can use a \texttt{while} program \( S \in WP(\Sigma) \). This program is
essentially a pair

\[(\Sigma, S)\].

The set of \texttt{while} programs is

\[ WP = \{ (\Sigma, S) \mid \Sigma \in Sig \text{ and } S \in WP(\Sigma) \} \].

The program \((\Sigma, S)\) is displayed as

\[
\text{program signature } \Sigma \text{ endsig; begin } S \text{ end}
\]

**Programs**

Programs are now a combination of interface definition and a sequence of commands.

\[
\begin{align*}
\textbf{bnf} & \quad \text{While Programs over Any Interface} \\
\textbf{import} & \quad Signature, Statements, I/O \\
\textbf{rules} & \quad \\
<\text{while program}> & ::= \text{program} <\text{Signature}> ; <\text{while body}> \\
<\text{while body}> & ::= \text{begin} <\text{command list}> \text{ end} \\
<\text{command list}> & ::= <\text{command}> \mid <\text{command list}> ; <\text{command}> \\
<\text{command}> & ::= <\text{statement}> \mid <i/o \text{ statement}> \\
\end{align*}
\]

We illustrate the structure of the grammar for \texttt{while} programs over any data type in
Figure 10.8.

**Interface Declaration**

The interface declaration is defined by the grammar \textit{Signature} given in Sections 10.1 and 10.2.
But for ease of reference we reproduce its flattened version here.
Figure 10.8: Architecture of *while* programs over any data type. Grammars that differ from those for *while* programs over a fixed signature are indicated by rectangles with non-rounded corners.

<table>
<thead>
<tr>
<th>bnf</th>
<th>Flattened Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rules</strong></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Signature&gt;</code> ::= <strong>signature</strong> <code>&lt;Name&gt;</code> <code>&lt;Newline&gt;</code> <code>&lt;Sorts&gt;</code> <code>&lt;Constants&gt;</code> <code>&lt;Operations&gt;</code></td>
<td><strong>endsig</strong></td>
</tr>
<tr>
<td><code>&lt;Names&gt;</code> ::= <code>&lt;Name&gt;</code></td>
<td><code>&lt;Name&gt;</code>, <code>&lt;Names&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Name&gt;</code> ::= <code>&lt;Letter&gt;</code></td>
<td><code>&lt;Letter&gt;</code> <code>&lt;Space&gt;</code> <code>&lt;Name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Sorts&gt;</code> ::= <code>&lt;sorts</code> <code>&lt;SortList&gt;</code> <code>&lt;Newline&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SortList&gt;</code> ::= <code>&lt;Sort&gt;</code></td>
<td><code>&lt;Sort&gt;</code>, <code>&lt;SortList&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Sort&gt;</code> ::= <code>&lt;Name&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Constants&gt;</code> ::= <code>&lt;constants</code> <code>&lt;ConstantList&gt;</code> <code>&lt;Newline&gt;</code></td>
<td><code>&lt;Name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;ConstantList&gt;</code> ::= <code>&lt;Constant&gt;</code></td>
<td><code>&lt;Constant&gt;</code> <code>&lt;Newline&gt;</code> <code>&lt;ConstantList&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Constant&gt;</code> ::= <code>&lt;Names&gt;</code> : <code>&lt;Sort&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Operations&gt;</code> ::= <code>&lt;operations</code> <code>&lt;OperationList&gt;</code> <code>&lt;Newline&gt;</code></td>
<td><code>&lt;Name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;OperationList&gt;</code> ::= <code>&lt;Operation&gt;</code></td>
<td><code>&lt;Operation&gt;</code> <code>&lt;Newline&gt;</code> <code>&lt;OperationList&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Operation&gt;</code> ::= <code>&lt;Names&gt;</code> : <code>&lt;DomainSort&gt;</code></td>
<td><code>&lt;Sort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;DomainSort&gt;</code> ::= <code>&lt;Sort&gt;</code></td>
<td><code>&lt;Sort&gt;</code> <code>&lt;DomainSort&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Letter&gt;</code> ::= <code>&lt;LowerCase&gt;</code></td>
<td><code>&lt;UpperCase&gt;</code></td>
</tr>
<tr>
<td><code>&lt;LowerCase&gt;</code> ::= <code>&lt;a</code> <code>&lt;b</code> <code>&lt;c</code> <code>&lt;d</code> <code>&lt;e</code> <code>&lt;f</code> <code>&lt;g</code> <code>&lt;h</code> <code>&lt;i</code> <code>&lt;j</code> <code>&lt;k</code> <code>&lt;l</code> <code>&lt;m</code> <code>&lt;n</code> <code>&lt;o</code> <code>&lt;p</code> <code>&lt;q</code> <code>&lt;r</code> <code>&lt;s</code> <code>&lt;t</code> <code>&lt;u</code> <code>&lt;v</code> <code>&lt;w</code> <code>&lt;x</code> <code>&lt;y</code> <code>&lt;z</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;UpperCase&gt;</code> ::= <code>&lt;A</code> <code>&lt;B</code> <code>&lt;C</code> <code>&lt;D</code> <code>&lt;E</code> <code>&lt;F</code> <code>&lt;G</code> <code>&lt;H</code> <code>&lt;I</code> <code>&lt;J</code> <code>&lt;K</code> <code>&lt;L</code> <code>&lt;M</code> <code>&lt;N</code> <code>&lt;O</code> <code>&lt;P</code> <code>&lt;Q</code> <code>&lt;R</code> <code>&lt;S</code> <code>&lt;T</code> <code>&lt;U</code> <code>&lt;V</code> <code>&lt;W</code> <code>&lt;X</code> <code>&lt;Y</code> <code>&lt;Z</code></td>
<td></td>
</tr>
</tbody>
</table>
### Input/Output Statements

The statements concerned with input and output processing remain unaltered.

```
bnf  I/O

import Identifiers

rules
  <i/o statement> ::= <read statement> | <write statement>
  <read statement> ::= read (<identifier list>)
  <write statement> ::= write (<identifier list>)
  <identifier list> ::= <identifier> | <identifier list>, <identifier>
```

### Statements

The other forms of statements are also unaffected.

```
bnf  Statements

import Expressions, BooleanExpressions

rules
  <statement> ::= <assignment statement> | <conditional statement> | 
                  <iterative statement> | <null statement>
  <assignment statement> ::= <identifier> ::= <expression>
  <conditional statement> ::= if <Boolean expression> then <command list>
                             else <command list> fi
  <iterative statement> ::= while <Boolean expression> do <command list> od
  <null statement> ::= skip
```

### Boolean Expressions

Boolean expressions are similar to those we formed over the signature \textit{Fixed}. The difference here is that the Boolean expressions we can form using the interface definition depend on what signature is declared. So we cannot say at this point

(i) what the names of any tests will be,

(ii) nor what their arguments will look like, either in terms of their number or type.

We can only give the amorphous description that a Boolean expression can be formed as the application of some relation to some list of expressions.
Expressions

Similar remarks apply to expressions. Expressions that we will be able to form depend on the constants and operations defined in the interface. We cannot know at this point what the names of any constants or operations will be. And we cannot know how many arguments an operation will take, nor what the types of any of those arguments will be.

Identifiers

The identifiers that we can form remain unchanged.
Flattened Version

By substituting in the appropriate grammars that are imported, we can form the flattened grammar:
Flattened While Programs over any Interface

rules
<while program> ::= program <Signature> ; <while body>
<Signature> ::= signature <Name> <Newline>
          ::= <Sorts> <Constants> <Operations>
<Names> ::= <Name> | <Name> , <Names>
>Name ::= <Letter> | <Letter> <Name> |<Space> <Name>
<Sorts> ::= sorts <SortList> <Newline>
<SortList> ::= <Sort> | <Sort> , <SortList>
          ::= <Sort> <Newline> <SortList>
<Sort> ::= <Name>
<Constants> ::= constants <ConstantList> <Newline>
         ::= constants <Newline>
<ConstantList> ::= <Constant> | <Constant> <Newline> <ConstantList>
<Constant> ::= <Names> : <Sort>
<Operations> ::= operations <OperationList> <Newline>
         ::= operations <Newline>
<OperationList> ::= <Operation> | <Operation> <Newline> <OperationList>
<Operation> ::= <Names> : <DomainSort> → <Sort>
<DomainSort> ::= <Sort> | <Sort> × <DomainSort>
<while body> ::= begin <command list> end
<command list> ::= <command> | <command list> ; <command>
<command> ::= <statement> | <read statement> | <write statement>
<read statement> ::= read ( <identifier list> )
<write statement> ::= write ( <identifier list> )
<identifier list> ::= <identifier> | <identifier list> , <identifier>
<statement> ::= <assignment statement> | <conditional statement> |
              <iterative statement> | <null statement>
<assignment statement> ::= <identifier> := <expression>
<conditional statement> ::= if <Boolean expression> then <command list>
                  else <command list> fi
<iterative statement> ::= while <Boolean expression> do <command list> od
<null statement> ::= skip
<Boolean expression> ::= <identifier> | true | false |
                      ::= not ( <Boolean expression> )|
                      ::= and ( <Boolean expression> , <Boolean expression> )|
                      ::= or ( <Boolean expression> , <Boolean expression> )|
                      ::= <opname> (<expression list> )
<expression> ::= <identifier> | <constname> |<expression list>)
          ::= <identifier> | <expression> <expression list>
<expression list> ::= <letter> | <identifier> <letter> |
               ::= <identifier> <digit>
<number> ::= <digit> | <number> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
          ::= r | s | t | u | v | w | x | y | z |
          ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N |
          ::= O | P | Q | R | S | T | U | V | W | X | Y | Z |
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
The grammar $G^{\text{while}}$ produces programs of the desired form, but also produces strings that are defective in a number of ways. For example,

(i) we can define interfaces in which sorts are used but not declared;

(ii) we can produce strings in which operations used in the body have not been declared in the signature;

(iii) we can form expressions in which an operation does not have the right number of arguments;

(iv) we can form expressions in which an operation its operations are not of the correct types.
10.7. A WHILE PROGRAMMING LANGUAGE OVER ALL DATA TYPES

Exercises for Chapter 10

1. Add rules to the grammar for first-order language specification in Section 10.4.2 to enable variables that appear in the axioms to be declared.

2. Design two metalanguages for writing grammars in BNF
   a. A kernel language for grammars without `import`, and
   b. An extended language for grammars with `import`,
   and define them formally using grammars. Recalling the intended meaning of the `import` mechanism from Section 4.7, give a formal definition using the two languages for grammars by constructing a syntactic transformation that reduces the extended language to the kernel language. (Hint: Since flattening signatures is analogous to flattening grammars, adapt the flattening algorithm for signatures to produce a flattening algorithm for the grammars in the extended grammar language.)

3. Derive a program for Euclid’s algorithm.

4. Add terminal symbols, non-terminal symbols and production rules to the grammar of WP in order to define
   a. `repeat` statements;
   b. `for` statements;
   c. concurrent assignments;
   d. variable declarations; and
   e. `case` statements.

5. Adapt the grammars of `while` programs over natural numbers in Section 9.3.2 and Section 10.5.1 to create a grammar for a language of `while` programs over a signature for real numbers.

6. Using the grammar of Section 10.6 for any fixed signature, derive a grammar for `while` computation over the signature for the natural numbers in Section 10.5. How many terminals, non-terminals and production rules does your grammar have? What is the difference between this grammar and that in Section 10.5?

7. Using the grammar of Section 10.6 for any fixed signature, derive grammars for `while` languages with the following data types:
   a. the integers;
   b. the real numbers;
   c. the integers and the real numbers; and
   d. characters and strings.
Assignment for Chapter 10

Consider the grammar for the **while** programming language for all signatures. How many terminal, non-terminals and production rules does it possess? Augment the grammar to add the following constructs to the language:

(a) **repeat** statements;

(b) concurrent assignment statements; and

(c) variable declarations in which identifiers are declared along with their sorts.

List the ways that strings derivable by the grammar may contain features that we would expect to call syntax errors in the final programming language.

Derive the following programs from the new grammar:

(i) a program over the real numbers whose signature has the sort declaration property, body has the variable declaration property, but whose body contains sorts not declared in the signature; and

(ii) a program over the real numbers whose signature has the sort declaration property but whose body fails to have the variable declaration body.
Chapter 11

Context-Free Grammars and Programming Languages

We have introduced the concepts of formal language and grammar and used them to define languages for imperative programs and interfaces. The grammars we used were large but fairly easy to create and apply. However, we encountered undesirable syntactic properties that seemed awkward to rule out.

The grammars we used were of a special kind. Recall, that in general, a grammar with terminals $T$ and nonterminals $N$, a production rule may be of the form

$$u \rightarrow v$$

where $u \in (N \cup T)^+$ and $v \in (N \cup T)^*$ are both strings of non-terminals, and/or terminals. In the grammars we actually used, the production rules all had the simpler form

$$A \rightarrow v$$

where $A \in N$. Such grammars are called context-free grammars.

Context-free grammars are an important class of grammars since they have an attractive and illuminating theory that is eminently useful in practice — as our earlier examples suggest.

In this Chapter we focus on context-free grammars and explore their uses and limitations. We will explain the basic theory of derivation trees, normal forms and recognition algorithms.

We show that features concerning

(i) variable declarations in programs, and

(ii) concurrent assignments

cannot be defined. At the heart of these proofs is an important result called the Pumping Lemma for context-free languages. The Assignment for the chapter is to show that features concerning the declaration of sorts in signatures cannot be defined by context-free grammars.

11.1 Chomsky Hierarchy and Context Free Grammars

The complexity of a grammar arises from the nature and, to a lesser extent, the number of its rules. The grammar for while programs in Section 10.5.1 contained over 100 production rules but cannot be considered complicated to use. One reason is that its rules have a simple form.
CHAPTER 11. CONTEXT-FREE GRAMMARS AND PROGRAMMING LANGUAGES

Let $G$ be a grammar with a set $T$ of terminals and a set $N$ of non-terminals. The following classification of $G$, by means of the complexity of its production rules $P$, is called the Chomsky Hierarchy.

**John: Problem** With the current definition, we cannot generate $L = \{\epsilon\}$ with a c.s. grammar, therefore it is only true for $L \neq \{\epsilon\}$ that

\[ \text{regular} \subseteq \text{cf} \subseteq \text{cs} \subseteq \text{grammar}. \]

(Which is the solution that Hopcroft and Ullman take, as their starting point is that c.s. grammars are a restriction of grammars whereby the RHS of productions are at least as long as the LHS.)

Another solution I've found (in Revesz) is to specifically add to the definition of cs grammars, the extra case that $S \rightarrow \epsilon$ providing that $S$ does not appear on the RHS of any production rule.

**Definition (Chomsky Hierarchy)** A grammar $G$ is said to be of Type 0 or unrestricted if, and only if, it is any grammar of the kind already defined, i.e., we place no restrictions on the production rules, which all have the form

\[ u \rightarrow v \]

where $u \in (T \cup N)^+$ is a non-empty string and $v \in (T \cup N)^*$ is any string.

A grammar $G$ is said to be of Type 1 or context-sensitive if, and only if, all its productions have the form

\[ uAv \rightarrow uv \]

where $A \in N$ is a non-terminal which rewrites to a non-empty string $w \in (T \cup N)^+$, but only where $A$ is in the context of the strings $u, v \in (T \cup N)^*$.

A grammar $G$ is said to be of Type 2 or context-free if, and only if, all its productions have the form

\[ A \rightarrow w \]

where $A \in N$ is a non-terminal that can always be used to rewrite to a string $w \in (T \cup N)^*$.

A grammar $G$ is said to be of Type 3 or regular if, and only if, all its productions have the form:

\[ A \rightarrow uB \text{ or } A \rightarrow u \]  \hspace{1cm} (1)

or else

\[ A \rightarrow Bu \text{ or } A \rightarrow u \]  \hspace{1cm} (2)

where $A, B \in N$ are non-terminals and $u \in T^*$ is a string. If all the productions are of the form (1) then we say $G$ is a right-linear grammar; and if all are of the form (2) then we say $G$ is a left-linear grammar.

A language $L \subseteq T^*$ is unrestricted, context-sensitive, context-free or regular if, and only if, there exists a grammar $G$ of the relevant type, such that $L(G) = L$. The four conditions on grammars are related as follows:

\[ \text{regular} \Rightarrow \text{context-free} \Rightarrow \text{context-sensitive} \Rightarrow \text{grammar} \]

and are illustrated in Figure 11.1.
11.1. Example

Consider the language

\[ L^{a^{2n}} = \{ a^i \mid i \text{ is even} \}. \]

We can generate this language using an unrestricted grammar:

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{unrestricted} a^{2n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>( a )</td>
</tr>
<tr>
<td>nonterminals</td>
<td>( S )</td>
</tr>
<tr>
<td>start symbol</td>
<td>( S )</td>
</tr>
<tr>
<td>productions</td>
<td>( S \rightarrow \epsilon )</td>
</tr>
<tr>
<td></td>
<td>( S \rightarrow aa )</td>
</tr>
<tr>
<td></td>
<td>( a \rightarrow aaa )</td>
</tr>
</tbody>
</table>

Or using a context-sensitive grammar:

**Problem**: Definition of c.s. grammar precludes from having epsilon production.

<table>
<thead>
<tr>
<th>grammar</th>
<th>( G^{cs} a^{2n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>( a )</td>
</tr>
<tr>
<td>nonterminals</td>
<td>( S )</td>
</tr>
<tr>
<td>start symbol</td>
<td>( S )</td>
</tr>
<tr>
<td>productions</td>
<td>( S \rightarrow \epsilon )</td>
</tr>
<tr>
<td></td>
<td>( S \rightarrow aaS )</td>
</tr>
<tr>
<td></td>
<td>( aS \rightarrow aSaa )</td>
</tr>
</tbody>
</table>

Or using the context-free grammar:
of Examples 9.2.2(3). Or using the regular grammar:

| grammar | \( G_{\text{reg}} \ a^{2n} \) |
| nonterminals | \( S \) |
| nonterminals | \( S \) |
| productions | \( S \rightarrow \epsilon \) |
| productions | \( S \rightarrow aaS \) |

Note that the grammar

(i) \( G^{\text{unrestricted}} \ a^{2n} \) is of type 0 and is not context-sensitive (and hence, is not context-free or regular);

(ii) \( G^{\text{cs}} \ a^{2n} \) is context-sensitive and is (hence, also of type 0), and is not context-free (and hence, not regular);

(iii) \( G^{\text{cf}} \ a^{2n} \) is context-free (and hence, also context-sensitive and type 0), and is not regular; and

(iv) \( G^{\text{reg}} \ a^{2n} \) is regular (and hence, also context-free, context-sensitive and type 0).

Each of these grammars generates the language \( L_{a^{2n}} \):

\[
L(G^{\text{unrestricted}} \ a^{2n}) = L(G^{\text{cs}} \ a^{2n}) = L(G^{\text{cf}} \ a^{2n}) = L(G^{\text{reg}} \ a^{2n}) = L_{a^{2n}}.
\]

The language \( L_{a^{2n}} \) is regular, as we can find a regular grammar \( G^{\text{reg}} \ a^{2n} \) such that \( L_{a^{2n}} = L(G^{a^{2n}}) \).

### 11.1.2 Examples

(i) All the grammars of Section 9.3.2 are context-free.
(ii) The grammars developed to generate the **while** language in Section 10.5.1 are context-free.

(iii) The grammar

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G^{abc}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b, c$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, A, B$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow abc$</td>
<td>$Ab \rightarrow bA$</td>
</tr>
<tr>
<td>$S \rightarrow aAbc$</td>
<td>$Ac \rightarrow Bbcc$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

is context-sensitive.

The form of the rules have a profound effect on the derivations. We will consider some basic properties of the derivations that are possible with context-free grammars.

### 11.2 Regular Languages

To appear.

#### 11.2.1 Regular Grammars

To appear.

#### 11.2.2 Languages Generated by Regular Grammars

To appear.

**Finite State Machines**

To appear.

**Regular Expressions**

To appear.

### 11.3 Limitations of Regular Grammars

To appear.
11.3.1 The Pumping Lemma for Regular Languages

To appear.

11.3.2 Constructs that are not Regular

To appear.

11.4 Derivation Trees for Context Free Grammars

As we have seen in our examples, it is often useful to picture a derivation from a context-free grammar as a tree. Each node of the tree is labelled with either a non-terminal symbol or with a terminal symbol of the grammar. An interior node has label $A$ and $n$ children labelled $X_1$ to $X_n$ (from the left) if, and only if, there exists a production rule $A \rightarrow X_1X_2 \cdots X_n \in P$, as shown in Figure 11.2.

$$
\begin{array}{c}
A \\
X_1 X_2 \cdots X_n
\end{array}
$$

Figure 11.2: Representation of the application of a production rule $A \rightarrow X_1X_2 \cdots X_n$.

Suppose we have a grammar $G$ with terminals $T$, non terminals $N$, start symbol $S$ and productions $P$. A string $w \in L(G)$ in the language defined by the grammar always has a derivation tree with root $S$ and strings of terminal symbols at the leaves only, such that the left-to-right concatenation of the terminal strings at the leaves gives the string $w$.

**Definition (Derivation Tree)** Consider a derivation

$$w \Rightarrow^* w'$$

of a string $w'$ from a string $w$ consisting of $n$ steps

$$w = w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_{i-1} \Rightarrow w_i \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w_n = w'.$$

At each step $i$ a production rule is applied to produce $w_i$ from $w_{i-1}$. There are two cases.

If a production rule only terminal symbols on its right-hand side, i.e., it is of the form $A \rightarrow a_1 \cdots a_n$, then a derivation $A \Rightarrow a_1 \cdots a_n$ is possible from the grammar. This gives a derivation subtree with root $A$ and a single leaf child $a_1 \cdots a_n$ as shown in Figure 11.3.

$$
\begin{array}{c}
A \\
a_1 a_2 \cdots a_n
\end{array}
$$

Figure 11.3: Derivation sub-tree for production rule $A \rightarrow a_1 \cdots a_n$.

A production rule with at least one non-terminal symbol on the right-hand side, say

$$A \rightarrow u_0A_1u_1 \cdots u_{n-1}A_n u_n$$
11.4. DERIVATION TREES FOR CONTEXT FREE GRAMMARS

for nonterminals $A, A_1, \ldots, A_n \in N$, (possibly empty) strings of terminals $u_0, \ldots, u_n \in T^*$ and $n \geq 1$, gives a possible derivation

$$A \Rightarrow u_0A_1u_1 \cdots u_{n-1}A_nu_n.$$  

This gives a derivation sub-tree with root $A$ and children whose concatenation gives the string $u_0A_1u_1 \cdots u_{n-1}A_nu_n$. Further derivations from the non-terminal symbols $A_1, \ldots, A_n$ will in later steps give sub-trees with roots $A_1, \ldots, A_n$, as shown in Figure 11.3.

![Figure 11.4: Derivation sub-tree for production rule $A \rightarrow u_0A_1u_1 \cdots u_{n-1}A_nu_n$.]

Given a derivation tree $D \in Tree(N, T)$ with terminals from $T$ at the leaves and non-terminals from $N$ at internal nodes, we can define a function

$$frontier : Tree(N, T) \rightarrow T^*$$

to recover the string $frontier(D)$ produced by $D$ by:

$$frontier(D) = \begin{cases} 
D & \text{if leaf}(D) = \text{tt}; \\
frontier(D_1) \cdot frontier(D_2) \cdots \cdot frontier(D_n) & \text{if leaf}(D) = \text{ff} \text{ and } D_1, \ldots, D_n \text{ are the subtrees of } D 
\end{cases}$$

where $\text{leaf} : Tree \rightarrow B$ is a predicate that determines whether a tree is a leaf or not.

11.4.1 Examples

Recall that

$$\Rightarrow$$

indicates a 1-step reduction in a derivation by applying 1 production rule, and that

$$\Rightarrow^n$$

indicates $n$ reductions in a derivation by applying $n$ production rules.

(i) Consider the grammar of Examples 9.2.2(1). Some sample derivations and their corresponding trees are shown in Figure 11.5.

(ii) Consider the grammar of Examples 9.2.2(2). Some sample derivations and their corresponding trees are shown in Figure 11.6.

(iii) Consider the grammar of Examples 9.2.2(3). Some sample derivations and their corresponding trees are shown in Figure 11.7.

(iv) Consider the grammar $\mathcal{G}^{Boolean\ Expressions}$ of Section 9.3.2(vi) for generating Boolean expressions. Some sample derivations and their corresponding trees are shown in Figure 11.8.
\( S \rightarrow 1 \quad S \rightarrow 0S \rightarrow 01 \quad S \rightarrow 1S \rightarrow 10S \rightarrow 101 \)

Figure 11.5: Derivation trees for Examples 9.2.2(1).

\( S \rightarrow \epsilon \quad S \rightarrow aSb \rightarrow ab \quad S \rightarrow a^nSb^n \rightarrow a^n b^n \)

Figure 11.6: Derivation trees for Examples 9.2.2(2).

(v) Consider the \textit{WP} Language, whose syntax was given by the BNF in Section 10.5.1. The derivation tree for the derivation of the program

\[
\begin{align*}
\text{begin} \\
x := & x + 1 \\
\text{end}
\end{align*}
\]

in Example 10.5.2 is given in Figure 11.9

Notice that one important feature of derivation trees is that they may factor out or hide the precise order in which different production rules are applied to a string. Consider, for example, the grammar

\( S \rightarrow \epsilon \quad S \rightarrow aS \rightarrow aa \quad S \rightarrow a^nSa^n \rightarrow a^{2n} \)

Figure 11.7: Derivation trees for Examples 9.2.2(3).
Then, there are two derivations for the string \( ab \):

\[
S \rightarrow AB \rightarrow aB \rightarrow ab
\]

and

\[
S \rightarrow AB \rightarrow Ab \rightarrow ab.
\]

The difference is in the order of applying the three production rules.

However, these derivations are considered equivalent when drawn as trees, as there is only one derivation tree for \( ab \), namely:

### 11.4.2 Ambiguity

In many grammars it is possible to derive the same string in more than one way by varying the order of some production rules. In some grammars it is possible to derive the same string using different choices of production rules, i.e., there is more than one derivation tree for any string that can be produced by the grammar. Such grammars are called ambiguous. An example of an ambiguous grammar is
\[ S \]

\[ A \quad B \]

\[ a \quad b \]

**grammar** \( G^\text{ambiguous} \) \( \text{ab} \)

**terminals** \( a, b \)

**nonterminals** \( S \)

**start symbol** \( S \)

**productions**
\[ S \rightarrow aS \]
\[ S \rightarrow b \]
\[ S \rightarrow ab \]

The grammar \( G \) is ambiguous as the string \( ab \) can be derived in with different choices of rules, giving two different derivation trees as shown in Figure 11.10.

We would like to ensure that any derivation of a string is unique.

*Can we always remove such ambiguities or not?*
Clearly, in the case of $G^{\text{ambiguous}}_{ab}$, we can; for example the grammar $G^{ab}$ generates the same language

$$L(G^{ab}) = L(G^{\text{ambiguous}}_{ab}) = \{a, b\}$$

as $L(G^{\text{ambiguous}}_{ab})$.

If we cannot always remove ambiguities, then can we find a context-free language $L$ for which every possible context-free grammar $G$ that generates $L = L(G)$ contains an ambiguity? Such a language is

$$L_{\text{inherently ambiguous}} = \{a^n b^n c^n d^n \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^n d^n \mid n \geq 1, m \geq 1\}$$

and is said to be inherently ambiguous. For a proof see Hopcroft and Ullman [1979] (note that this is hard). This demonstrates that whilst for some particular context-free languages we may be able to produce an unambiguous grammar, this will not be true in general.

Fortunately, the problem of inherent ambiguity does not seem to arise in programming languages. However:

**Theorem** The problem of deciding whether a given context-free grammar is ambiguous is undecidable.

For a proof of this, see Hopcroft and Ullman [1979].

### 11.5 Normal Forms for Context Free Grammars

We have seen that a language can be defined by different grammars. In practice, it is easy to find different grammars that define the same language. For each language there are infinitely many grammars available to define them. This raises the possibility of developing methods for simplifying grammars by transforming their rules.

Any context-free grammar can be transformed into an equivalent grammar that has all of its production rules in certain simple formats called normal forms. There are two important normal forms for context-free grammars: Chomsky Normal Form and Greibach Normal Form.

**Definition (Chomsky Normal Form)** A grammar is said to be in Chomsky Normal Form if all of its productions are either of the form

$$A \rightarrow BC \quad \text{or} \quad A \rightarrow u$$

where $u$ is a terminal symbol and $A, B, C$ are non-terminal symbols.

**11.5.1 Theorem (Chomsky Normal Form)**

*Any context-free grammar $G$ can be transformed to a grammar $G^{CNF}$, which is in Chomsky Normal Form, such that their languages are equivalent, i.e., $L(G) = L(G^{CNF})$.***
Proof. Let \( G = (T, N, S, P) \) be any context-free grammar. Then we can find an equivalent grammar \( G^{C_{NF}} \) which is in Chomsky Normal Form, in two stages. In the first stage we separate terminals and variables in the right-hand-sides of rules.

Consider a production \( p \in P \); we transform it as follows:

If it has a single terminal symbol on the right, then the production is in an acceptable form. If not, consider \( p \) to be of the form

\[
S \rightarrow X_1 X_2 \cdots X_m
\]

where \( m \geq 2 \). If \( X_i \) is a terminal, say \( u \), then introduce a new non-terminal \( C_u \) and a new production \( C_u \rightarrow u \) (which is in the allowed form). In this way, for all the \( X_i \)'s that are terminals, replace the occurrence of \( X_i \) by \( C_u \). Let the new set of non-terminal be \( N' \) and the new set of productions be \( P' \). Consider the grammar \( G' = (T, N', S, P') \). We prove that \( L(G') = L(G) \).

If \( a \Rightarrow_G b \) then \( a \Rightarrow^*_G b \), thus \( L(G) \subseteq L(G') \). Now we demonstrate (by induction) that if \( A \Rightarrow^*_G w \) then \( A \Rightarrow^*_G w_i \) for \( A \in N \) and \( w \in T^* \). The result is trivial for one step derivations. Let \( A \Rightarrow^*_G w \) be a \((k+1)\) step derivation. The first step must be of the form \( A \rightarrow B_1 B_2 \cdots B_m \) for \( m \geq 2 \), and we can write \( w = w_1 w_2 \cdots w_m \), where \( B_i \Rightarrow^*_G w_i \) for \( 1 \leq i \leq m \). If \( B_i \) is a \( C_u \) for some terminal \( u \) then \( w_i \) must be \( u \). By construction of \( P' \) there is a production \( A \rightarrow X_1 X_2 \cdots X_n \) of \( P \) where

\[
X_i = \begin{cases} 
B_i & \text{if } B_i \in N; \\
u_i & \text{otherwise.}
\end{cases}
\]

For those \( B_i \in N \) we know \( B_i \Rightarrow^*_G w_i \) takes less than \( k+1 \) steps, so by the induction assumption \( X_i \Rightarrow^*_G w_i \) and hence \( A \Rightarrow^*_G w \).

We have now proved that any context-free language can be generated by a grammar with rules of the form

\[
A \rightarrow u \quad \text{or} \quad A \rightarrow B_1 B_2 \cdots B_m
\]

for \( m \geq 2 \), where all the \( B_i \)'s are non-terminal symbols and \( u \) is a terminal symbol.

In our second stage, we now modify such a grammar \( G' = (T, N', S, P') \) by adding some additional symbols to \( N' \) and replacing some productions of \( P' \). For each production

\[
A \rightarrow B_1 B_2 \cdots B_m
\]

where \( m \geq 3 \), we create new non-terminals \( D_1, D_2, \ldots, D_{m-2} \) and replace \( A \rightarrow B_1 B_2 \cdots B_m \) by the set of productions:

\[
A \rightarrow B_1 D_1 \\
D_1 \rightarrow B_2 D_2 \\
D_2 \rightarrow B_3 D_3 \\
\vdots \\
D_{m-2} \rightarrow B_{m-1} B_m
\]

If \( N'^{C_{NF}} \) is the new set of non-terminals and \( P'^{C_{NF}} \) is the new set of productions, then \( G'^{C_{NF}} = (T, N'^{C_{NF}}, S, P'^{C_{NF}}) \) is in Chomsky Normal Form and it is clear that \( L(G') = L(G^{C_{NF}}) \), by essentially the same proof as above, and hence \( L(G) = L(G^{C_{NF}}) \).

Another important form of context-free grammar is an apparently modest generalisation of regular grammars.
11.6. PARSING ALGORITHMS FOR CONTEXT FREE GRAMMARS

**Definition (Greibach Normal Form)** A grammar is said to be in *Greibach Normal Form* if all of its productions are either of the form

\[ A \rightarrow u \quad \text{or} \quad A \rightarrow uX_1X_2 \cdots X_n \]

where \( u \in T \) is a terminal and \( X_1, X_2, \ldots, X_n \in N \) are non-terminals.

Thus, the right-hand sides of any production rule consists of a *terminal* symbol, followed by zero or more occurrences of non-terminal symbols.

11.5.2 Theorem (Greibach Normal Form)

Any \( e \)-free context-free grammar \( G \) can be transformed to a grammar \( G^{\text{GNF}} \), which is in Greibach Normal Form, such that their languages are equivalent, i.e., \( L(G) = L(G^{\text{GNF}}) \).

11.6 Parsing Algorithms for Context Free Grammars

The purpose of a grammar \( G \) is to define a language \( L(G) \) by means of rules for rewriting strings. A basic problem is this:

**Definition (Recognition Problem)** Let \( L(G) \subseteq T^* \) be a language defined by a grammar \( G \) with alphabet \( T \). How can we decide, for any given word \( w \in T^* \), whether or not \( w \in L(G) \)?

An algorithm \( A_G \) that decides membership for \( L(G) \) may be called a *recogniser*, or *parser*, for \( G \). The difficulty of deriving a recogniser will depend on the complexity of the production rules for the grammar.

*Does every grammar have a recogniser?*

*If a grammar has a recogniser, then what is its complexity?*

11.6.1 Grammars and Machines

The analysis of the recognition problem for the grammars of the Chomsky Hierarchy was an important early achievement in the theory of grammars. The problem was solved by characterising the grammars and their languages in terms of simple machine models. The prototype result was the following:

**Theorem** A language \( L \) is definable by an unrestricted grammar if, and only if, it is recognisable by a Turing Machine.

This means that for any language \( L \subseteq T^* \), the following are equivalent:

(i) there is a type 0 grammar \( G \) with alphabet \( T \) and start symbol \( S \) such that

\[ w \in L \iff S \Rightarrow_G^* w \]

and

(ii) there is a Turing Machine \( M \) with set \( T \) of symbols such that

\[ w \in L \iff M \text{ halts on input string } w, \text{ i.e., } M(w) \downarrow. \]
The sets of strings over an alphabet recognised by the halting of a Turing Machine are called *semidecidable* or *recursively enumerable* sets of strings. Such sets of strings are precisely the sets of strings that can be listed or enumerated by algorithms.

**Corollary** A formal language is definable by a grammar if, and only if, it can be enumerated by an algorithm.

The theorem came naturally from the equivalence between Emil Post’s production systems and Turing Machines discovered earlier in Computability Theory. A basic discovery of Computability Theory is that

there exist semidecidable (or recursively enumerable) sets that are not decidable (or recursive).

**Corollary** There exist grammars for which no recognition algorithm exists, i.e., the recognition problem is algorithmically undecidable.

The classification of the recognition problem for other grammars of the Chomsky Hierarchy led to some new machine models and algorithms.

We can measure, or estimate, the complexity of parsing with $A_G$ (and hence that of $G$) in terms of resources, such as space and time needed to recognise, or fail to recognise, a word. This can be taken as a function $f$ of word length

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

The algorithm $A_G$ is of order $f(n)$ if, to decide for any $w \in T^*$ with $|w| = n$, whether or not $w \in L(G)$, takes order $f(n)$ steps; as usual we will write: $A_G$ is $O(f(n))$. The number of steps in the computation $A_G(w)$ to decide if $w \in L(G)$ is bounded by $f(|w|)$.

The languages and grammars of the Chomsky Hierarchy can be characterised by the complexity of the recognition problem, as measured by machine models.

**11.6.2 Theorem (Language Recognition)**

A language $L_i$ is definable by a grammar $G_i$ of type $i$, for $i = 0, 1, 2$ and 3 if, and only if, $L_i$ is recognisable by a machine $M_i$ of the type given in the following table:

<table>
<thead>
<tr>
<th>Language Type $L_i$</th>
<th>Machine Model Equivalent $M_i$</th>
<th>Complexity of Recognition of word $w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: Unrestricted</td>
<td>Turing Machine</td>
<td>Undecidable</td>
</tr>
<tr>
<td>1: Context Sensitive</td>
<td>Linear Bounded Automaton</td>
<td>Decidable</td>
</tr>
<tr>
<td>2: Context Free</td>
<td>Push Down Stack Automaton</td>
<td>Cubic Time $O(</td>
</tr>
<tr>
<td>3: Regular</td>
<td>Finite State Automaton</td>
<td>Linear Time $O(</td>
</tr>
</tbody>
</table>

This type of classification motivated the development of new grammars with practical recognition algorithms.
11.6. PARSING ALGORITHMS FOR CONTEXT FREE GRAMMARS

11.6.3 A Recognition Algorithm for Context-Free Grammars

Let $G$ be a context-free grammar in Chomsky Normal Form. Let

$$w = t_1 t_2 \cdots t_i \cdots t_{i+j} \cdots t_n \in L(G)$$

with $|w| = n$. Let $w_{ij} = t_i \cdots t_{i+j}$ be the substring of $w$ starting at the $i$th character and having length $j$. So for example, $w_{1n} = w$. We consider the general problem of deciding for any $i, j \in \mathbb{N}$ and non-terminal $A \in \mathbb{N}$ whether or not

$$A \Rightarrow^* w_{ij}.$$ 

Clearly $w \in L(G)$ if, and only if, $S \Rightarrow^* w_{1n}$. We will then solve the more general problem by induction on the length $j$ of strings.

**Base Case**

These are unit segments. If $A \Rightarrow w_{i1}$ then there must be a production rule $A \rightarrow w_{i1}$, which can be trivially decided by inspection of the set $P$.

**Induction Step**

Here $A \Rightarrow^* w_{ij}$ if, and only if, there is some production $A \rightarrow BC$ and some breakpoint $k$ with $1 \leq k < j$ such that $B \Rightarrow^* w_{ik}$ and $C \Rightarrow^* w_{(i+k)(j-k)}$. Since the size of the two new segments $w_{ik}$ and $w_{(i+k)(j-k)}$ are less than $j$, being of length $k$ and $j-k$ respectively, we can exploit the induction step (this gives us an inductive algorithm) to decide whether such derivations exist.

To formalise the algorithm, we define for any word $w \in T^*$ and $i, j \in \mathbb{N}$ the set

$$N_{ij} = \{ A \mid A \text{ is a non-terminal such that } A \Rightarrow^* w_{ij} \}.$$ 

Notice that $1 \leq i \leq n - j + 1$ since starting a substring at $i$ of length $j$ means that $j \leq n - i + 1$. Thus, $w \in L(G)$ if, and only if, $S \in N_{1n}$.

begin (* bottom up construction of $N_{1n}$*)

for $i := 1$ to $n$ do

$N(i, 1) := \{ A \mid A \rightarrow a \in P \text{ and } w_i = a \}$ od;

for $j := 2$ to $n$ do

for $i := 1$ to $n - j + 1$ do

$N(i, j) = \emptyset$ ;

for $k := 1$ to $j - 1$ do

$N(i, j) := N(i, j) \cup \{ A \mid A \rightarrow BC \in P, B \in N(i, k) \text{ and } C \in N(i + k, j - k) \}$

od

od

end

end

There exist many parsing algorithms for special types of context-free grammars; we shall consider the Cocke-Younger-Kasami (CYK) algorithm. For a word $w$ of length $n$, the algorithm requires $O(n^3)$ steps in the worst case. There exist more efficient algorithms, particularly when more restrictions are placed upon the grammar $G$. 
11.7 The Pumping Lemma for Context Free Languages

The study of derivations for a class of grammars is an important task for theoretical understanding and practical applications.

The derivations of context-free grammars have several properties that enhance considerably the usefulness of context-free grammars. One such property is the Pumping Lemma. Originally discovered by Bar-Hillel et al. [1961], it is a fundamental theorem in the study of formal languages.

11.7.1 Pumping Lemma

Let $G$ be a context-free grammar with alphabet $T$. Then there exists a number $k = k(G) \in \mathbb{N}$ that will depend on the grammar $G$ such that, if a string $z \in L(G)$ and the length $|z| > k$ then we can write

$$z = uvwxy$$

as the concatenation of strings $u, v, w, x, y \in T^*$ and

(a) the length of the string $|vx| \geq 1$ (i.e., $v$ and $x$ are not both empty strings);

(b) the length of the mid-portion string $|vwx| \leq k$; and

(c) for all $i \geq 0$, the string

$$uv^iwx^i \in L(G).$$

Before proving the result let us explore what it means and consider an application.

The theorem says that for any context-free grammar $G$ there is a number $k$ that depends on the grammar $G$ such that for any string $z$ that is longer than length $k$, i.e.,

$$|z| > k,$$

it is possible to split up the string $z$ into five segments $u, v, w, x$ and $y$, i.e.,

$$z = uvwxy$$

with the property that either $v$ or $x$ is non-trivial, the middle segment $vwx$ is smaller than length $k$, and, in particular, removing $v$ and $x$, or copying $v$ and $x$, gives us infinitely many strings that the rules of the grammar will also accept, i.e.,

$$uvw \qquad z = uvwxy \qquad uvvvwxxy \qquad uvvwxxxy \qquad \vdots \qquad uv^iwx^i y \qquad \vdots$$

are all in $L(G)$. 


11.7. THE PUMPING LEMMA FOR CONTEXT FREE LANGUAGES

11.7.2 Applications of the Pumping Lemma

An application of the Pumping Lemma is to demonstrate that the language

\[ L = \{a^i b^i c^i \mid i \geq 0\} \]

is not context-free. Suppose \( L = L(G) \) for some context-free grammar \( G \). Let \( k \) be the constant from the Pumping Lemma. We can choose \( n > k/3 \) and consider \( z = a^n b^n c^n \), as this satisfies \(|a^n b^n c^n| > k\). By the Pumping Lemma we can rewrite \( z \) by \( z = uvwx \), with \( v \) and \( x \) not both empty and such that for \( i \geq 0 \) \( uv^iwx^i y \in L \). Consider \( v \) and \( x \) in \( z \). Suppose \( v \) contained a break, i.e., two symbols from \( \{a, b, c\} \). Then \( uv^2wx^2y \in L \) but in \( v^2 \) either \( b \) would precede \( a \), or \( c \) would precede \( b \), so \( v \) cannot contain a break; similarly \( x \) cannot contain a break. Thus \( v \) and \( x \) must contain only 1 symbol each (otherwise from the Pumping Lemma the symbol order will be disturbed). However in this case the Pumping Lemma destroys the requirement that all three symbols be present in equal numbers, since if \( v \) contains \( a \) and \( x \) contained \( c \) then pumping increases the number of \( a \)'s and \( c \)'s but not \( b \)'s. Hence, \( v \) and \( x \) must both be empty, contradicting the requirements, so there is no context-free grammar for \( L \).

The Pumping Lemma is not always used to give negative results regarding the non-existence of context-free grammars for languages. For example, we can prove that there exist algorithms to determine if a context-free grammar generates a language that is

- empty;
- finite; or
- infinite.

Further details are given in Hopcroft and Ullman [1979].

11.7.3 Proof of the Pumping Lemma

We show that if a word \( z \in L(G) \) is long enough then \( z \) has a derivation of the form:

\[ S \Rightarrow^* uAy \Rightarrow^* uvAx \Rightarrow^* uvwxy \]

in which the non-terminal \( A \) can be repeated in the following manner

\[ A \Rightarrow^* vAx \tag{1} \]

before it is eliminated

\[ A \Rightarrow^* w \tag{2} \]

Having found derivations of the appropriate form it is possible to “pump” using \( G \) as follows

\[ S \Rightarrow^* uAy \Rightarrow^* uvAx \text{ by (1)} \]
\[ \Rightarrow^* uv^2Ax^2y \text{ by (1)} \]
\[ \vdots \text{ i times} \]
\[ \Rightarrow^* uv^iAx^i y \text{ by (1)} \]
\[ \Rightarrow^* uv^iwx^i y \text{ by (2)} \]

An observation is that
long words need long derivations.

Let

\[ l = \max\{|\alpha| \mid A \to \alpha \in P\} + 1, \]

i.e., the longest right hand side of any production in \( G \). In any derivation of a word \( z \in L(G) \), the replacement of the non-terminal \( A \) by the string \( \alpha \), by applying \( A \to \alpha \), can introduce \( |\alpha| < l \) symbols.

**Lemma** If the height of the derivation tree for a word \( z \) is \( h \) then \( |z| \leq l^h \). Conversely if \( |z| > l^h \) then the height of its derivation tree is greater than \( h \).

Let \( m = |N| \) be the number of non-terminals in the grammar. We choose \( k = l^{m+1} \). By the above Lemma, the height of derivation tree \( Tr \) for the string \( z \) with \( |z| > k \) is greater than \( m+1 \). So at least one path \( p \) through \( Tr \) is longer than \( m+1 \) and some non-terminal in this path must be repeated. Travel path \( p \) upwards from the leaf searching for repeated non-terminals. Let \( A \) be the first such non-terminal encountered. We can picture \( Tr \) as shown in Figure 11.11:

![Derivation tree](image)

Figure 11.11: Derivation tree \( Tr \) for the string \( z = uvwxz \) where the height of \( Tr \) is such that there must be at least one repetition of a non-terminal \( A \). This is enforced through the constant \( k \) of the pumping lemma. The value of \( k = l^{m+1} \) is calculated from the length \( l \) of the longest right-hand side of the production rules and the number \( m = |N| \) of non-terminals in the grammar \( G \).

Take \( w \) to be the terminal string of subtree whose root is the lower \( A \), \( vwx \) to be the terminal string of subtree whose root is the upper \( A \), and \( u \) and \( y \) the remains of \( z \). Clearly we can also deduce \( S \to^* uv^iwx^i y \) for \( i \geq 0 \).

From our choice of \( A \) there can be no repeated non-terminals on the path \( p \) below the upper \( A \) so the height of the subtree of root upper \( A \) is less than \( m+1 \). From the above Lemma, \( |vwx| \leq l^{m+1} = k \). Finally we choose the shortest derivation sequence for \( z \) so we cannot have

\[ S \to^* uAy \to^+ uAy \to^* wwy = uvwxy. \]

Thus \( |wx| \geq 1 \). \( \square \)
11.8 Limitations of context-free grammars

In this and the following section, we will consider two programming features that cannot be defined by any context-free grammar. In Exercises for Chapter 9, we saw that we can introduce

(i) variable declarations, and

(ii) concurrent assignments

into our while language by context-free grammars but that in each case the language possessed some undesirable features. Suppose we wish to remove these features by adding the restrictions that:

(a) all identifiers in a program appear in the declaration; and

(b) variables are not duplicated in a concurrent assignment.

We will show that these restrictions cannot be imposed on a programming language, if we want the language to remain definable by a context-free grammar. The theorem for variable declarations is known as Floyd’s Theorem.

We first met the problem of undesirable features in a language defined by a context-free grammar when studying the interface definition language for signatures. The Sort Declaration Property also escapes context-free grammars. We see this in the assignment for this chapter.

Roughly speaking, these examples illustrate that context-free grammars cannot guarantee that:

a list of syntactic items is “complete”

or that

a list of syntactic items contains “no repetitions”.

Context-free grammars define a set of strings on which to base the definition of a programming language but cannot include even simple restrictions necessary to prepare for its intended semantics.

11.8.1 Variable Declarations and Floyd’s Theorem

Consider the simple language for while programs over the natural numbers. We can extend WP to include the declaration of identifiers by altering the BNF for programs of Section 10.5.1:

```
bnf     While Programs over Natural Numbers with Variable Declarations

import Statements, I/O, Declarations

rules
<while program> ::= begin <declaration> <command list> end |
             begin <command list> end
<command list> ::= <command> | <command list> ; <command>
<command>     ::= <statement> | <i/o statement>
```
And adding a BNF for variable declarations.

```
bnf Declarations
import Identifiers
rules
<declaration> ::= nat <identifier list> ;
<identifier list> ::= <identifier> | <identifier> , <identifier list>
```

Thus, the overall structure of `while` programs is shown in Figure 11.12.

![Diagram of While Programs over Natural Numbers with Variable Declarations](image)

Figure 11.12: Architecture of `while` programs over the natural numbers with declarations.

And the flattened grammar that results is:
11.8. LIMITATIONS OF CONTEXT-FREE GRAMMARS

**bnf** Flattened While Programs over Natural Numbers with Variable Declarations

**rules**

<while program> ::= begin <declaration> <command list> end |
begin <command list> end
<declaration> ::= nat <identifier list> ;
<identifier list> ::= <identifier> | <identifier>, <identifier list>
<command list> ::= <command> | <command list> ; <command>
<command> ::= <statement> | <read statement> | <write statement>
<statement> ::= <assignment statement> | <conditional statement> | 
<iterative statement> | <null statement>
<assignment statement> ::= <identifier> ::= <expression>
<conditional statement> ::= if <comparison> then <command list>
else <command list> fi
<iterative statement> ::= while <comparison> do <command list> od
<null statement> ::= skip
<read statement> ::= read (<identifier list>)
<write statement> ::= write (<identifier list>)
<identifier list> ::= <identifier> | <identifier list> , <identifier>
<comparison> ::= <Boolean expression> |
<Boolean expression> ::= <Boolean term> | 
<Boolean term> ::= <Boolean factor> |
<Boolean factor> ::= <Boolean atom> and <Boolean factor>
<expression> ::= <Boolean term> | (<<comparison> >)
<term> ::= <factor> | <term> <multiplying operator> <factor>
<factor> ::= <atom> | (<<expression> >)
<adding operator> ::= + | -
<multiplying operator> ::= * | / | mod
<relational operator> ::= = | < > | >= | <= | <>
<Boolean atom> ::= true | false
<atom> ::= <identifier> | <number>
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
<number> ::= <digit> | <number> <digit>
<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | 
r | s | t | u | v | w | x | y | z |
A | B | C | D | E | F | G | H | I | J | K | L | M | N |
O | P | Q | R | S | T | U | V | W | X | Y | Z | -
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

But now consider the condition on WP that:
Declaration Condition

In each program all the identifiers of the program are declared.

This property is not guaranteed by the given grammar. Can we define a context-free grammar that does meet this requirement? Floyd’s Theorem tells us that this is impossible.

11.8.2 Theorem (Floyd’s)

Let

\[ \text{Decl}_{n,m} = \text{begin nat } a^n b^m; a^n b^m := 0 \text{ end} \]

and let

\[ L \supseteq \{ \text{Decl}_{n,m} \mid n, m \geq 0, n + m > 0 \} \]

be a programming language containing the set of all such declarations.

Suppose that

(1) in any program of \( L \) all the identifiers of the program are declared; and

(2) identifiers may be of any length.

Then \( L \) cannot be defined by a context-free grammar.

Proof. Suppose, by way of contradiction, that there did exist a context-free grammar \( G \) that defined the language \( L \). Let \( k \) be the constant for \( G \) postulated by the Pumping Lemma. For \( n, m > k \), we have \( \text{Decl}_{n,m} \in L \) and \( |\text{Decl}_{n,m}| > k \). Thus, all the conditions for the Pumping Lemma are satisfied, so we can pump on \( \text{Decl}_{n,m} \).

Hence, we can write

\[ \text{Decl}_{n,m} = uvwx \]

with

(a) \( |vx| \geq 1 \);

(b) \( |vwx| < k \); and

(c) for any \( i \geq 0 \), \( P_i = uv^iwx^iy \in L(G) \), i.e., \( P_i \) is also a valid program.

Consider the structure of

\[ P_1 = \text{Decl}_{n,m} = \text{begin nat } a^n b^m; a^n b^m := 0 \text{ end}. \]

Since \( n, m > k \) and condition (b) must be satisfied, we deduce that one of three cases must hold.

First, suppose the symbol ; is not in \( vwx \). Then there are two cases:

(i) \( vwx \) is contained in the segment \( \text{begin nat } a^n b^m \), or

(ii) \( vwx \) is contained in the segment \( a^n b^m := 0 \text{ end} \).

Otherwise, suppose ; is in \( vwx \). Then
(iii) \( vwx \) is contained in the segment \( b^n; a^n \).

We consider the three cases in turn:

(i) Consider \( P_0 = uwy \). This is in \( L \) by the Pumping Lemma. If either \textbf{begin} or \textbf{nat} were in \( v \) or \( x \), then \( P_0 \) would invalidate the property that identifiers must be declared. Thus \( v \) and \( x \) are subscripts of \( a^n b^n \). However, since \( |vx| \geq 1 \) the program \( P_0 \) must contain an assignment to an identifier which is not the identifier that was declared. This contradicts our assumption so this case cannot hold.

(ii) Consider \( P_2 = uv^2wx^2y \). This is in \( L \) by the Pumping Lemma. If \( \text{:=}, 0 \) or \textbf{end} were in \( v \) or \( x \), then \( P_2 \) would not be syntactically correct for various reasons (exercise). However, if \( v \) and \( x \) are subscripts of \( a^n b^n \), then the program \( P_0 \) would involve an assignment in which the identifier had not been declared. Again, therefore this case cannot hold.

(iii) Consider \( P_0 = uwy \) and the location of \( ; \). This cannot lie in \( v \) or \( x \) since then \( P_0 \) would not be syntactically correct. Thus either

(a) \( vwx \) is a substring of \( b^n \) or \( a^n \); or

(b) \( v \) is a substring of \( b^n \), \( w \) is a substring of \( a^n b^n \) and \( x \) is a substring of \( a^n \).

Since \( |vx| \geq 1 \), \( P_0 \) would involve an undeclared identifier. Thus, this final case cannot hold.

\[ \square \]

### 11.8.3 The Concurrent Assignment Construct

Consider the \textbf{while} language, whose syntax was given by the BNF in Figure 10.5.1. We now augment this language with a deterministic parallel construct — the \textit{concurrent assignment} — to form the concurrent \textbf{while} language.

The concurrent assignment allows us to perform multiple assignments in parallel. Its syntax is given by

\[
\langle \text{concurrent assign} \rangle \ ::= \langle \text{assignment statement} \rangle | \\
\langle \text{identifier} \rangle , \langle \text{concurrent assign} \rangle , \langle \text{expression} \rangle .
\]

The intention is that a concurrent assignment of the form

\[
x_1 := e_1, x_2 := e_2, \ldots, x_n := e_n
\]

will assign the value of the expression \( e_i \) to \( x_1, e_2 \) to \( x_2, \ldots, e_n \) to \( x_n \) in \textit{parallel}.

With this motivation in mind, it would seem natural to apply the restriction:

**Definition (Distinctness Condition)** All the variables on the left-hand side of a concurrent assignment must be distinct.

Clearly, this is just a syntactic restriction we are placing on our language, so we should be able to specify this condition by using a grammar. Without the restriction, our language is context-free, by inspection of the production rules above. However, we show that adding this restriction takes the language outside the class of context-free languages.
11.8.4 Theorem

Let
\[\text{ConcAssign}_n = \text{begin } a, a^2, \ldots, a^n := 0, 0, \ldots, 0 \text{ end}\]

and let
\[L \supseteq \{\text{ConcAssign}_n \mid n \geq 1\}\]
be a programming language containing the set of all such concurrent assignment programs.

Suppose that

1. all the variables on the left-hand side of the assignment are distinct; and
2. there are the same number of expressions on the right-hand side of the assignment, as there are variables on the left-hand side;
3. and variables may be of any length.

Then \(L\) cannot be defined by a context-free grammar.

Proof. Assume for contradiction that \(L\) can be generated by a context-free grammar \(G\). Let \(k\) be the constant for \(G\) postulated by the Pumping Lemma. For \(n > k/2\), we have \(\text{ConcAssign}_n \in L\) and \(|\text{ConcAssign}_n| > k\). Thus, all the conditions for the Pumping Lemma are satisfied, so we can pump on \(\text{ConcAssign}_n\).

Hence, we can write \(\text{ConcAssign}_n = uvwx\) with

1. \(|ux| \geq 1|;
2. \(|vwx| < k|; and
3. for any \(i \geq 0, P_i = uv^iwx^iy \in L(G)\).

From the syntax of \(L\), we cannot have any of the symbols \texttt{begin}, \(\texttt{:=}\) or \texttt{end} being present in either of the strings \(v\) or \(x\), as they could be removed by pumping. As we also require that \(|ux| \geq 1\), one of the following conditions must hold, depending on whether \(\texttt{:=}\) is in \(w\) (case \(iii\)), or not (cases \(i\) and \(ii\)):

1. \(uvx\) is contained in the segment \(0, \ldots, 0\); or
2. \(uvx\) is contained in the segment \(a, a^2, \ldots, a^n\); or
3. \(uvx\) is contained in the segment \(a, a^2, \ldots, a^n := 0, \ldots, 0\).

We consider each of these cases in turn.

(i) \(uvx\) is contained in the segment \(0, \ldots, 0\). If a \(,\) appears in either \(v\) or \(x\), then the program \(P_0 = uvwy\) would be syntactically incorrect, having more variables on the left-hand side, than values on the right-hand side. If \(v\) or \(x\) were just a \(0\) though, then the program \(P_0 = uvwy\) would also be syntactically incorrect, having two adjacent commas.

(ii) \(uvx\) is contained in the segment \(a, a^2, \ldots, a^n\). Again, if a \(,\) appears in either \(v\) or \(x\), then the program \(P_0 = uvwy\) would be syntactically incorrect, this time having fewer variables on the left-hand side, than values on the right-hand side. So \(v\) and \(x\) can contain no commas.

Suppose \(v\) contains no commas. Then \(v\) is part, or all of an identifier, say \(a^0\).
\[ v = a^q \] If \( v \) is all of an identifier \( a^q \) then the program \( P_0 = uvwy \) would be syntactically incorrect, having two adjacent commas.

\[ v = a^p \] Suppose that \( v \) is part of an identifier \( a^p \) of \( a^q \), where there exists some \( r > 0 \) such that \( p + r = q \). Then consider the program \( P_0 = uvwy \). It will contain an identifier that has already appeared in the list, as it will be shorter than \( a^q \). (The particular identifier that is repeated in \( P_0 \) need not be \( a^q \), as the string \( x \) could also be part of this same identifier.)

So \( v \) must be the empty string \( \epsilon \).

Similarly, the string \( x \) must also be the empty string \( \epsilon \). But this violates condition (a) of the Pumping Lemma which requires that \( v \) and \( x \) cannot both be empty.

\[(iii)\] \( uv^2x \) is contained in the segment \( a,a^2,\ldots,a^n := 0,\ldots,0 \). We first note that the \( := \) must be contained in \( w \), otherwise it could be removed in program \( P_0 \). Hence, \( v \) must be contained in the segment \( a,a^2,\ldots,a^n \), and \( x \) in the segment \( 0,\ldots,0 \).

From the reasoning of \((ii)\), we can exclude the possibility that \( v \) contains no comma. In addition \( v \) cannot consists only of a comma, and cannot start and end with a comma or the program \( P_2 = uv^2wx^2y \) would be syntactically incorrect, having two adjacent commas. Thus, \( v \) must contain at least one comma and part, or all of at least one identifier.

If \( v \) has more than one comma then \( v \) contains at least all of one identifier, say \( a^p \). But then the program \( P_2 \) would repeat the variable \( a^p \) on the left-hand side of the assignment.

Therefore \( v \) contains only one comma, and part(s) of either 1 or 2 identifiers. Thus, there are three possibilities for \( v \):

\[ v = a^p, \quad v = \cdot a^p \quad \text{or} \quad v = a^p \cdot a^q \]

depending on whether \( v \) contains part of one or two variables.

\[ v = a^p, \quad \text{or} \quad v = \cdot a^p \] Consider that \( v \) contains one comma and part of a variable \( a^p \) of \( a^q \), where there exists some \( r \geq 0 \) such that \( p + r = q \). Then the program \( P_2 \) would contain a repeated variable on the left-hand side of the assignment, namely \( a^p \), as \( p \leq q \).

\[ v = a^p, a^q \] Consider that \( v \) contains parts of two identifiers which are separated by a comma. Let \( a^p \) be part of an identifier \( a^q \) and \( a^q \) part of an identifier \( a^t \) where there exists some \( m, r \geq 0 \) such that \( p + r = q \) and \( s + m = t \).

Now consider the program \( P_3 = uv^2wx^3y \). It will have \( v \) repeated three times, so the string \( v^3 \) will have the form:

\[ a^p, a^q, a^p, a^q, a^p, a^q \]

Thus, \( P_3 \) has the variable \( a^{q+p} \) repeated on the left-hand side of the assignment.

Therefore \( v \) cannot contain part of an identifier. Thus, \( v = \epsilon \).

As \( v = \epsilon \), pumping can only affect the right-hand side of the assignment. But we know from \((i)\) that we cannot pump only on some segment of \( 0,\ldots,0 \).

Hence, \( L \) is not context-free.
11.9 Historical Notes

As explained in the historical notes of the Introduction, the origins of formal language theory lie outside computer science in theoretical linguistics and logic. In summary, the following describes the history of our notions concerning formal languages. The simple mathematical definitions are to be found in two publications by Noam Chomsky:

Three models for the description of language, IRE Transactions on Language Theory IT-2 (1956) 113-124; and

On certain formal properties of grammars, Information and Control 2 (1959) 137-167.

<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>J Backus</td>
<td>BNF notations for the formal definition of Algol 60 with machine independence and compilation in mind.</td>
</tr>
<tr>
<td>1961</td>
<td>Y Bar-Hillel, A Perles, E Shamir</td>
<td>Study of context-free grammars and regular languages, including the Pumping Lemma.</td>
</tr>
<tr>
<td>1962</td>
<td>R W Floyd</td>
<td>Ambiguity. Floyd’s Theorem.</td>
</tr>
<tr>
<td>1962</td>
<td>D G Cantor</td>
<td>Ambiguity is undecidable</td>
</tr>
<tr>
<td>1965</td>
<td>D Knuth</td>
<td>LR(k) grammars, unambiguity and linear time parsing.</td>
</tr>
<tr>
<td>1965-66</td>
<td>Cocke, Kasami, Younger</td>
<td>(O(n^3)) parsing for context-free grammars</td>
</tr>
<tr>
<td>1975</td>
<td>L Valiant</td>
<td>(O(n^{2.81})) parsing for context-free grammars</td>
</tr>
</tbody>
</table>
11.9. HISTORICAL NOTES

Exercises for Chapter 11

1. Using the grammars $G$ and $G'$ of Examples 11.1.2(ii), show that the languages $L(G)$ and
$L(G')$ are equivalent.

2. Let $G$ be the grammar of Examples 11.1.2(iii). What is the language $L(G)$ that is
generated by $G$?

3. Give the derivation tree for the derivation of the sentence $aaabbb$ from the grammar of
Example 9.2.2(2).

4. Prove that for any grammar $G$, the terminal symbols occur only at the leaves and nowhere
else in the derivation tree.

5. Prove that for any grammar $G$, any sentence only has terminal symbols at the leaves.

6. Give another derivation that will correspond to the derivation tree given in 11.4.1(v).

7. Consider the following sets of characters:
   a. $B = \{0, 1, &, \vee, \supset\}$;
   b. $AE = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (, ), +, -, /, \ast\}$.

   a. Give four words that can be generated from each set.
   b. Give grammars:
      i. $G1$ to produce Boolean expressions using $B$ as the terminal set; and
      ii. $G2$ to produce arithmetic expressions using $AE$ as the terminal set.
   c. Give four examples of each of the following:
      i. strings that are in $B^*$ but not in $L(G1)$; and
      ii. strings that are in $AE^*$ but not in $L(G2)$.
   d. Give the productions of your grammar $G1$ in Greibach Normal Form, and the pro-
ductions of $G2$ in Chomsky Normal Form.
   e. Does the Pumping Lemma apply to grammars $G1$ and $G2$, and if so why?

8. Give derivation trees in $G1$ for the following:
   a. 11 & 01;
   b. 1 ⊃ 0 & 10; and
   c. 010 ∨ 1 & 111.

9. Give derivation trees in $G2$ for the following:
   a. $21 * 5$;
   b. $(3 + 4) * 10$; and
   c. $(5 - 3)/(10 + 2)$. 

10. Write out the transformation process for grammars in the Chomsky Normal Form Theorem as an algorithm in imperative pseudocode. What is the complexity of the algorithm for transforming a context-free grammar into Chomsky Normal Form?

11. Complete the details of the proof of the Chomsky Normal Form Theorem.

12. Give the grammar of Examples 9.2.2(2) in Chomsky Normal Form.

13. Give the grammar of Examples 9.2.2(2) in Greibach Normal Form.

14. The parity of a binary number is defined to be the sum modulo 2 of all its digits, for example:

\[
\begin{align*}
\text{parity}(11) &= 0 \\
\text{parity}(10) &= 1 \\
\text{parity}(111) &= 1.
\end{align*}
\]

Consider the following language where the \( b_i \) range over the terminal set \( \{0, 1\} \):

\[ L = \{ pb_1 b_2 \cdots b_n \mid p = \text{parity}(b_1 b_2 \cdots b_n) \}. \]

a. Give a grammar that produces \( L \).

b. Is the grammar context-free?

c. Use the Pumping Lemma to discover if there is a context-free grammar for \( L \).

15. Show that the CYK algorithm is dominated by the inner loop and that its computational cost is \( O(n^3) \).

16. What assumptions about programming language constructs for manipulating sets are implicit in the CYK algorithm, and the analysis of its complexity?

17. What is the complexity of the CYK algorithm in terms of the input size of the grammar?

18. Prove the following Pumping Lemma for regular languages. If \( L \) is a regular language, and \( w \in L \) is a sufficiently long word, then \( w \) can be written as \( xyz \), with:

a. \( |y| > 0 \);

b. and for all \( i > 0 \), \( xy^iz \in L \).

19. Use the above result to decide if the following languages can be generated by a regular grammar:

a. \( \{ ab^i \mid i > 0 \} \);

b. \( \{ ab^i \mid i, j > 0 \} \).

20. Provide a proof for the lemma used in the proof of the Pumping Lemma.

21. Prove that the concurrent assignment construction generates all strings of the form

\[ x_1, \ldots, x_n := e_1, \ldots, e_n \]

for variables \( x_1, \ldots, x_n \) and expressions \( e_1, \ldots, e_n \).
22. Extend the WP grammar of Section 10.5.1 to define the alternative construct:

\[ x_1, \ldots, x_m := e_1, \ldots, e_n \]

where \( m \) need not be equal to \( n \).

23. Consider the Proof of Theorem 11.8.4. There are many places where we can offer alternative proofs of the individual sub-cases. For example, in case (i), when we consider \( v = \_ \), we said that this case could not hold, as the program \( P_2 \) would then have two adjacent commas. We could equally well have used the explanation that \( P_0 \) would have two adjacent zeros, which would make it syntactically incorrect. Find some more places where we could use different arguments to prove that a sub-case could not hold. Give full explanations as to why your alternatives are equally valid.

24. In the Proof of Theorem 11.8.4, we set \( n > k/2 \). By the Pumping Lemma we also require that \( |vw| < k \), and hence \( |vw| < 2n \). But the longest identifier \( a^n \) has length \( n \). Thus, \( |a^{n-1}a^n| = 2n > k \). What consequences result from this observation in the Proof of Theorem 11.8.4?
Assignment for Chapter 11: A Language for Signatures

In Sections 10.1 and import idl, we presented an interface definition language for data types, namely

\[ \textit{a language for signatures} \]

This assignment shows that the desirable Sort Declaration Property is not definable by context-free grammars.

Prove the following:

**Theorem** The set of signatures with the Sort Declaration Property is not context-free.

Consider the proof of Floyd's Theorem for the declaration of variables and adapt the argument to show the theorem. The basic stages are as follows:

1. Assume, for a contradiction, that there is a context-free grammar \( G \) that defines the language
   \[ L = \{ \Sigma \in \text{Sig} \mid \Sigma \text{ has the Sort Declaration Property} \} \]

2. Consider the simple single-sorted signature
   \[ \Sigma_{n,m} = \textit{signature sorts } a^n b^m; \textit{operations } c : \rightarrow a^n b^m \textit{ endsig } \in L \]
   where \( n, m \geq 0 \) and \( n + m > 0 \) which contains only one constant \( c \) of sort \( a^n b^m \). Use the Pumping Lemma to derive the contradiction that \( L(G) = L \).
Chapter 12

Abstract Syntax and Algebras of Terms

The diverse set of examples of syntax we have met in earlier chapters have all been modelled as

*strings of symbols.*

The theory of grammars, which is based on the simple concept of a string, provides a theory of syntax that combines theoretical depth with ease of understanding, and practical breadth with ease of application.

However, from the examples we have met, it is clear that syntax has more structure than that of a string, i.e., linear sequence of written symbols. In a simple program text, there are plenty of components playing different roles: there are alpha-numeric symbols, operations and tests, bracketing symbols, declarations, assignments and control constructs, with validity conditions on reserved words, precedence, priority and completeness. Indeed, some syntax can be justly called two-dimensional. The postal addresses are commonly *displayed* in two dimensions (recall that in Chapter 9, we used the *newline* token to code that feature). More dramatically, musical notation is two dimensional.

Syntax has structure based on

(i) how it is built up;

(ii) how one checks its validity;

and, of course, on

(iii) what it is *intended* to mean.

The structure of syntax is weakly represented in the simple linear notion of a string. Commonly, we associate parse trees to strings to better organise and reveal structural aspects of syntax.

A particular strength of grammars and strings is that they provide tools for specifying in complete detail the *actual* syntax that is to used. Invariably, to a user a syntax is a string of particular symbols from a particular alphabet. The form of syntax that we must specify for users is commonly called

*concrete syntax.*
CHAPTER 12. ABSTRACT SYNTAX AND ALGEBRAS OF TERMS

There is a more general approach to syntax, which aims to analyse syntactic structure based on (i)–(iii) above. It recognizes the forms defined by the syntactic rules, and how syntax is built up or broken down. It is especially concerned with how to process syntax when giving it a meaning or semantics. This form of syntax is commonly called

abstract syntax.

Abstract syntax is independent of the particular symbols and notations used. While concrete syntax is easy to recognize and process, abstract syntax is more subtle.

In this chapter we will focus on modelling syntax with a more a structured representation. We will complement the theory of syntax based on strings with a theory based on

terms.

A term is a composition of operation symbols applied to constants and variables. It allows large syntactic components to be combined in many ways — each way is given by an operation. In contrast, the string based approach allows syntactic components to be combined in essentially one way, namely some form of string concatenation.

We begin in Section 12.1 by studying the idea of a term and, in particular, the way terms are constructed. To reason about terms, and to define functions on terms, we formulate principles of structural induction or recursion on terms in Section 12.2. We show how these induction principles are used in transforming terms, in mapping terms to trees (in Section 12.3), and to evaluating terms in data types (in Section 12.5).

Terms, like strings and numbers, are an important form of data. In Section 12.4 we use some simple ideas from Part I about modelling data types by algebras, to model the
data type of terms.

By equipping the set of terms with operations we create algebras of terms, called
term algebras.

With this algebraic approach to terms, we are able to show that the ideas of

- a function defined by induction or recursion on terms;
- a term evaluation map; and
- a homomorphism on a term algebra

are equivalent concepts (in Section 12.6) and we prove that:

Every algebra is isomorphic to some algebra of terms divided by a congruence relation on terms.

After the general algebraic preparations, we apply this abstract view of syntax to programming languages. In Section 12.7, as an instructive and representative example, we develop an algebraic model of the abstract syntax of the while language. The approach is quite general and, in Section 12.8, we explain the general method. At the heart of the method is the use of term algebras to capture the abstract syntax of the language to be specified. This corresponds with the use of the context-free grammar to lay down the right syntactic components.
In fact, we show that *context-free grammars are equivalent with term algebras* by studying the Rus-Goguen-Thatcher algorithm that transforms a context-free grammar $G$ into a signature $\Sigma^G$ and, hence, a term algebra $T(\Sigma^G)$ that can be used to define the same language. This term algebra method is applied in case studies of programming language syntax definitions. In Section 12.9 we discuss the algebraic approach to languages that are not context-free.

### 12.1 Terms

#### 12.1.1 Single-Sorted Terms

To make the notion of a term clearer, we shall consider the ideas when we only have one sort in our signature.

**Definition (Single-sorted terms)** Let $\Sigma_{SS}$ be a single-sorted signature

<table>
<thead>
<tr>
<th>signature</th>
<th>$SS$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>$s$</td>
</tr>
<tr>
<td>constants</td>
<td>$\ldots, c : s, \ldots$</td>
</tr>
<tr>
<td>operations</td>
<td>$\ldots, f : s^n \rightarrow s, \ldots$</td>
</tr>
</tbody>
</table>

Let $X$ be a set of variables. We inductively define the set $T(\Sigma_{SS}, X)$ of terms by:

(i) each constant symbol $c : s$ in $\Sigma_{SS}$ is a term

$$c \in T(\Sigma_{SS}, X);$$

(ii) each variable symbol $x \in X$ is a term

$$x \in T(\Sigma_{SS}, X);$$

(iii) for each $n$-ary function symbol $f : s^n \rightarrow s$ in $\Sigma_{SS}$ and any terms $t_1, \ldots, t_n$,

$$f(t_1, \ldots, t_n) \in T(\Sigma_{SS}, X)$$

is a term; and

(iv) nothing else is a term.

**Example: Boolean terms**

Consider the signature $\Sigma_{Boolean}$ which we define as follows:
**signature**  
Boolean

**sorts**  
Bool

**constants**  
true, false :→ Bool

**operations**  
¬ : Bool → Bool  
∧ : Bool × Bool → Bool  
∨ : Bool × Bool → Bool  
⇒: Bool × Bool → Bool

endsig

and the set \( X = \{ x_1, x_2, x_3, \ldots \} \) of Bool-sorted variables.

We can form terms in \( T(\Sigma_{\text{Boolean}}, X) \):

\[
\begin{align*}
\text{true}, & \quad \text{false}, \\
\land (x_1, \land (x_2, x_3)), & \quad \land (x_1, \lor (x_2, x_3)) \\
\Rightarrow (x_1, x_2), & \quad \Rightarrow (x_1, \Rightarrow (x_2, x_3)), \\
& \Rightarrow (x_1, \lor (\neg (x_1), \text{true})), \\
& \Rightarrow (x_1, \Rightarrow (\Rightarrow (x_2, x_3), x_4))
\end{align*}
\]

In their more familiar infix form:

\[
\begin{align*}
\text{true}, & \quad \text{false}, \\
(x_1 \land x_2) \land x_3, & \quad x_1 \land (x_2 \lor x_3), \\
x_1 \Rightarrow x_2, & \quad x_1 \Rightarrow (x_2 \Rightarrow x_3), \\
x_1 \lor \neg x_1 & \Rightarrow \text{true}, \\
x_1 \Rightarrow ((x_2 \Rightarrow x_3) \Rightarrow x_4)
\end{align*}
\]

These syntactic expressions are also known as *propositional expressions* or *propositional formulae*.

**Example: Natural number terms**

Consider the signature \( \Sigma_{\text{Naturals}} \) which we define as follows:

**signature**  
Naturals

**sorts**  
nat

**constants**  
zero : → nat

**operations**  
succ : nat → nat  
add : nat × nat → nat  
mult : nat × nat → nat

endsig
12.1. TERMS

Let X be a nat-sorted set of variable symbols. Let \( x, y, z \in X \).
Then the set \( T(\Sigma_{\text{Naturals}}, X) \) of terms includes:

\[
\begin{align*}
    x & \quad \text{succ}(x) & \cdots & \text{succ}^n(x) & \cdots \\
    \text{add}(\text{zero}, x) & \quad \text{add}(x, y) & \quad \text{add}(\text{succ}(x), \text{succ}^2(y)) \\
    \text{mult}(x, \text{succ}(y)) & \quad \text{mult}(\text{zero}, \text{add}(x, \text{zero})) & \quad \text{add}(x, \text{mult}(y, \text{succ}(\text{zero}))).
\end{align*}
\]

Example: Real number terms

The algebra we extracted from Babbage’s letter on the Analytical Engine in Section 3.6 can be given the following single-sorted signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>Analytical Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>real</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1, Max, Min : \to real</td>
</tr>
<tr>
<td>operations</td>
<td>( + : \text{real} \times \text{real} \to \text{real} )</td>
</tr>
<tr>
<td></td>
<td>( - : \text{real} \times \text{real} \to \text{real} )</td>
</tr>
<tr>
<td></td>
<td>( \cdot : \text{real} \times \text{real} \to \text{real} )</td>
</tr>
<tr>
<td></td>
<td>( \div : \text{real} \times \text{real} \to \text{real} )</td>
</tr>
<tr>
<td></td>
<td>( \sqrt : \text{real} \to \text{real} )</td>
</tr>
</tbody>
</table>

Composing these functions in various ways leads to a range of familiar useful formulae or expressions, which are all examples of terms over \( \Sigma_{\text{Analytical Engine}} \). Algebraic expressions concerning real numbers use standard conventions in infix notation and brackets that simplify the appearance of terms. For example, here are some terms over \( \Sigma_{\text{Analytical Engine}} \):

\[
\begin{align*}
    x + (y + z) & \quad (x + y) + z \\
    x (y, z) & \quad (x, y) z
\end{align*}
\]

In working with real numbers these terms are to evaluate to the same values and so we write

\[
\begin{align*}
    x + y + z \\
    x y z
\end{align*}
\]

respectively.

A quadratic expression

\[
ax^2 + bx + c
\]

is a simplification of some term such as

\[
(a.(x x) + (b.x)) + c
\]
or

\[((a.x).x) + ((b.x) + c)\].

There is a great deal of algebraic analysis behind these familiar and seemingly obvious simplifications. Here are some more terms in their usual form:

<table>
<thead>
<tr>
<th>Term</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a + b)/cd)</td>
<td>(\sqrt{s(s - a)(s - b)(s - c)})</td>
</tr>
<tr>
<td>Quadratic formula</td>
<td>(-b + \sqrt{b^2 - 4ac}/2a)</td>
</tr>
<tr>
<td>(k)-th root</td>
<td>(\sqrt[\cdots]x) ((k) times)</td>
</tr>
<tr>
<td>Polynomial</td>
<td>(a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0)</td>
</tr>
<tr>
<td>Conic Section Formula</td>
<td>(ax^2 + by^2 + cxy + dx + ey + f)</td>
</tr>
<tr>
<td>Pendulum formula</td>
<td>(2\pi \sqrt{T/g})</td>
</tr>
<tr>
<td>Inverse square formula</td>
<td>(\frac{k a_0^2}{r^2})</td>
</tr>
</tbody>
</table>

Most terms have variables, though not all. We will define precisely for any term \(t \in T(\Sigma_{SS}, X)\), the set

\(\text{var}(t)\)

of all variables from \(X\) appearing in \(t\). For example,

\[\text{var}(x_1 + (x_2 + x_3)) = \{x_1, x_2, x_3\}\]
\[\text{var}(\text{suc}^2(\text{zero})) = \emptyset\]

**Definition (Term Variables)** Let \(t \in T(\Sigma_{SS}, X)\). The set \(\text{var}(t)\) of all variables in \(t\) is defined by induction on the structure of \(t\).

**Constant** \(t \equiv c\) then:

\(\text{var}(t) = \emptyset\)

**Variable** \(t \equiv x_i\) then:

\(\text{var}(t) = x_i\)

**Operation** \(t \equiv f(t_1, \ldots, t_n)\) then:

\(\text{var}(t) = \text{var}(t_1) \cup \cdots \cup \text{var}(t_n)\)
12.1. TERMS

Example The set $T(\Sigma_{\text{Natural}})$ of variable-free terms includes:

\[
\begin{align*}
\text{zero} & \quad \text{succ}(\text{zero}) \quad \text{succ}^2(\text{zero}) \quad \cdots \quad \text{succ}^n(\text{zero}) \quad \cdots \\
\text{add}(\text{zero}, \text{zero}) & \quad \text{add}(\text{zero}, \text{succ}(\text{zero})) \quad \text{add}(\text{succ}^n(\text{zero}), \text{succ}^n(\text{zero})) \\
\text{mult}(\text{zero}, \text{zero}) & \quad \text{mult}(\text{zero}, \text{succ}(\text{zero})) \quad \text{mult}(\text{succ}(\text{zero}), \text{zero}) \\
\text{add}(\text{zero}, \text{mult}(\text{succ}^n(\text{zero}), \text{succ}^n(\text{zero}))) & \quad \text{mult}(\text{succ}^n(\text{zero}), \text{succ}^n(\text{zero})) \\
\text{mult}(\text{add}(\text{succ}^n(\text{zero}), \text{succ}^n(\text{zero})), \text{succ}^p(\text{zero})).
\end{align*}
\]

A term is closed if it does not contain any variables, i.e.,

\[\text{var}(t) = \emptyset\]

otherwise it is called open, when

\[\text{var}(t) \neq \emptyset.\]

The set $T(\Sigma_{SS}, \emptyset)$ of closed terms over a signature $\Sigma_{SS}$ and the empty set of variables is denoted by

\[T(\Sigma_{SS}).\]

12.1.2 Many-Sorted Terms

Signatures are naturally many-sorted. Operations take arguments of different sorts. This complicates the notion of terms somewhat: all constants, variables and terms have various sorts.

Definition (Many-sorted terms) Let $\Sigma_{MS}$ be the signature

<table>
<thead>
<tr>
<th>signature</th>
<th>$MS$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>$\ldots, s, \ldots$</td>
</tr>
<tr>
<td>constants</td>
<td>$\ldots, c : \rightarrow s, \ldots$</td>
</tr>
<tr>
<td>operations</td>
<td>$\ldots, f : s(1) \times \cdots \times s(n) \rightarrow s, \ldots$</td>
</tr>
</tbody>
</table>

Let

\[X = \langle X_s | s \in S \rangle\]

be a family of non-empty sets of variables indexed by the sort set $S$ of $\Sigma_{MS}$. For each sort $\ldots, s, \ldots$, the set

\[X_s\]

is a non-empty set of variable symbols of sort $s$.

The set $T(\Sigma_{MS}, X)_s$ of all terms of sort $s$ is inductively defined by:

\[(i) \text{ each constant symbol } c : \rightarrow s \text{ is a term of sort } s, \text{ i.e., } c \in T(\Sigma_{MS}, X)_s;\]
(ii) each variable symbol $x \in X_s$ is a term of sort $s$, i.e.,

$$x \in T(\Sigma_{MS}, X)_s;$$

(iii) for each function symbol $f : s(1) \times \cdots \times s(n) \to s$

and any terms $t_1 \in T(\Sigma_{MS}, X)_{s(1)}, \ldots, t_n \in T(\Sigma_{MS}, X)_{s(n)}$,

the expression $f(t_1, \ldots, t_n)$ is a term of sort $s$, i.e.,

$$f(t_1, \ldots, t_n) \in T(\Sigma_{MS}, X)_s;$$

and

(iv) nothing else is a term.

**Example: Terms over natural numbers and Booleans**

Let us now consider two-sorted terms over the following signature $\Sigma_{\text{Naturals with Tests}}$:

<table>
<thead>
<tr>
<th>signature</th>
<th>Naturals with Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>nat, Bool</td>
</tr>
<tr>
<td>constants</td>
<td></td>
</tr>
<tr>
<td>zero :</td>
<td>$\to$ nat</td>
</tr>
<tr>
<td>true :</td>
<td>$\to$ Bool</td>
</tr>
<tr>
<td>false :</td>
<td>$\to$ Bool</td>
</tr>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>succ :</td>
<td>nat $\to$ nat</td>
</tr>
<tr>
<td>pred :</td>
<td>nat $\to$ nat</td>
</tr>
<tr>
<td>add :</td>
<td>nat $\times$ nat $\to$ nat</td>
</tr>
<tr>
<td>mult :</td>
<td>nat $\times$ nat $\to$ nat</td>
</tr>
<tr>
<td>equals :</td>
<td>nat $\times$ nat $\to$ Bool</td>
</tr>
<tr>
<td>less than :</td>
<td>nat $\times$ nat $\to$ Bool</td>
</tr>
<tr>
<td>not :</td>
<td>Bool $\to$ Bool</td>
</tr>
<tr>
<td>and :</td>
<td>Bool $\times$ Bool $\to$ Bool</td>
</tr>
<tr>
<td>or :</td>
<td>Bool $\times$ Bool $\to$ Bool</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>

Let $X_{\text{nat}} = \{x_1, x_2, \ldots\}$ be a nat-sorted set of variable symbols and $X_{\text{Bool}} = \{b_1, b_2, \ldots\}$ be a Bool-sorted set of variable symbols. We define $X = < X_{\text{nat}}, X_{\text{Bool}} >$. 
12.2. **INDUCTION ON TERMS**

We can form the terms of sort \( \text{nat} \):

\[
\begin{align*}
x_2 \\
\text{succ}(x_2) \\
\text{succ}((\text{succ}(x_2))) \\
\text{succ}^n(x_2) \\
\text{add}(x_1, x_2) \\
\text{mult}(\text{pred}(x_1), \text{succ}(x_2)) \\
\text{succ}(\text{add}(\text{zero}, x_1), \text{mult}(x_2, x_3)) \\
\ldots
\end{align*}
\]

Now consider the terms of sort \( \text{Bool} \):

\[
\begin{align*}
b_1 \\
\text{true} \\
\text{false} \\
\text{and}(\text{true}, \text{true}) \\
\text{not}(b_1) \\
\text{or}(b_1, \text{and}(\text{false}, b_2)) \\
\text{equals}(x_1, \text{zero}) \\
\text{and}(\text{equals}(x_2, \text{succ}(\text{zero})), \text{not}(\text{less}\text{than}(x_2, \text{succ}(\text{zero})))) \\
\ldots
\end{align*}
\]

Here, terms of sort \( \text{nat} \) are subterms of some terms of sort \( \text{Bool} \). Notice that no constants or variables of sort \( \text{Bool} \) appear as subterms of an operator with range sort \( \text{nat} \) as these operators only have domain sort \( \text{nat} \). Thus, all subterms of sort \( \text{nat} \) are also terms of sort \( \text{nat} \).

### 12.2 Induction on Terms

Terms are syntactic objects which are built from constant symbols and variables by the application of function symbols. Starting from constants \( \ldots, c, \ldots \) from signature \( \Sigma \) and variables \( \ldots, x, \ldots \) from \( X \), the operation symbols are applied again and again: given an operation symbol \( f \) and terms \( t_1, \ldots, t_n \), construct the new term

\[ f(t_1, \ldots, t_n). \]

All the terms are made in this way. Each term is the result of a unique process of choosing constants and variables and applying function symbols. Different choices or orders of application lead to different terms.

The structure of terms is similar to that of natural numbers, in which each number is built from the constant 0 by applications of the function \( n + 1 \) in a unique way.

In analogy with the natural numbers, we may use this way of generating terms to formulate methods of reasoning and defining transformations by induction or recursion on terms.

First, we explain these ideas in the case of single-sorted terms. Then, we repeat the explanation in the case of many-sorted terms.
12.2.1 Principle of Induction for Single-Sorted Terms

Now given any \( \Sigma \)-term \( t \), either \( t \) is some constant symbol \( c \), or \( t \) is some variable, or \( t \) is a term built up by function application and has the form \( f(t_1, \ldots, t_n) \). Induction exploits this simple fact.

### Principle of Induction for Single-Sorted Terms

Let \( \Sigma \) be a single-sorted signature and \( T(\Sigma, X) \) the set of all terms over \( \Sigma \). Let \( P \) be a property of \( \Sigma \)-terms, i.e., \( P \subseteq T(\Sigma, X) \) is a set of terms having that property; we write \( P(t) \) for \( t \in P \). If the following two statements hold:

- **Base Case**
  - \( P(c) \) is true for each constant symbol \( c \in \Sigma \);
  - \( P(x) \) is true for each variable \( x \in X \).

- **Induction Step**
  Let \( t_1, \ldots, t_n \in T(\Sigma, X) \) be any terms of sorts \( s \). Let \( f : s^n \to s \in \Sigma \) be any function symbol. If \( P(t_1), \ldots, P(t_n) \) are all true then

\[
P(f(t_1, \ldots, t_n))
\]

is also true.

Then

\[
P(t) \text{ is true for all terms } t \in T(\Sigma, X).
\]

12.2.2 Principle of Induction for Many-Sorted Terms

Given any \( s \)-sorted \( \Sigma \)-term \( t \), either \( t \) is some \( s \)-sorted constant symbol \( c : \to s \), or \( t \) is some \( s \)-sorted variable, or \( t \) is a term built up by the application of some function \( f : s(1) \times \cdots s(n) \to s \) and has the form \( f(t_1, \ldots, t_n) \).
12.2. INDUCTION ON TERMS

Principle of Induction for Many-Sorted Terms

Let \( \Sigma \) be an \( S \)-sorted signature and \( T(\Sigma, X) \) the set of all terms over \( \Sigma \). Let \( P \) be a property of all \( \Sigma \)-terms, i.e., \( P \subseteq T(\Sigma, X) \) is a set of terms having that property; we write \( P(t) \) for \( t \in P \). If the following two statements hold:

**Base Case**

\( P(c) \) is true for each constant symbol \( c : s \in \Sigma \) and for each sort \( s \in S \);
\( P(x) \) is true for each variable \( x \in X_s \) and for each sort \( s \in S \).

**Induction Step**

Let \( t_1 \in T(\Sigma, X)_{s_1} \ldots t_n \in T(\Sigma, X)_{s_n} \) be any terms of sorts \( s_1, \ldots, s_n \). Let \( f : s(1) \times \cdots \times s(n) \to s \in \Sigma \) be any function symbol. If \( P(t_1), \ldots, P(t_n) \) are all true then

\[ P(f(t_1, \ldots, t_n)) \]

is also true.

Then

\( P(t) \) is true for all terms \( t \in T(\Sigma, X) \).

12.2.3 Functions on Terms

**Defining Functions by Structural Induction or Recursion for Single-Sorted Terms**

The construction properties of 0 and \( n + 1 \) are used to define functions by primitive recursion on all natural numbers (see Section 7.4.3). Similarly, the construction properties of constants, variables and function symbols are used to define functions on all terms.

**Definition (Structural Induction)**

Let \( \Sigma \) be a single-sorted signature and \( T(\Sigma, X) \) the set of all \( \Sigma \)-terms with variables from \( X \). Let \( A \) be any non-empty set.

A function

\[ \phi : T(\Sigma, X) \to A \]

is defined by *structural induction* or *structural recursion* if for each:

1. Constant \( c \in \Sigma \), there exists some element \( c^A \in A \),
2. Variable \( x \in X \), there exists some element \( x^A \in A \),
3. Function symbol \( f \in \Sigma \), there exists some function \( f^A : A^n \to A \),

such that the following equations are satisfied for all constants \( c \in \Sigma \), variables \( x \in X \) and terms \( t_1, \ldots, t_n \in T(\Sigma, X) \):

\[
\begin{align*}
\phi(c) &= c^A \\
\phi(x) &= x^A \\
\phi(f(t_1, \ldots, t_n)) &= f^A(\phi(t_1), \ldots, \phi(t_n))
\end{align*}
\]
Before exploring this general method for defining functions on terms, we record an important fact. It seems clear that the equations describe how to evaluate \( \phi \) on every term, and that only one function \( \phi \) could satisfy these equations.

**Lemma (Uniqueness)** The function \( \phi : T(\Sigma, X) \to A \) satisfying the equations is uniquely defined, i.e., if \( \phi_1 \) and \( \phi_2 \) are two functions satisfying the equations then \( \phi_1 = \phi_2 \).

**Proof** See Exercise 6. \( \Box \)

**Example** Consider ways of defining the size of a term. First, one can measure a term by the height or depth of application or nesting of its operations, or, simply, by the number of operations in the term.

**Height** Let

\[
\text{Height} : T(\Sigma, X) \to \mathbb{N}
\]

be the function that measures the height \( \text{Height}(t) \) of a term \( t \in T(\Sigma, X) \). This function is defined by structural induction as follows:

\[
\begin{align*}
\text{Height}(c) &= 1 \\
\text{Height}(x) &= 1 \\
\text{Height}(f(t_1, \ldots, t_n)) &= \max(\text{Height}(t_1), \ldots, \text{Height}(t_n)) + 1
\end{align*}
\]

Comparing these equations with the general definition by structural induction, we see that \( A = \mathbb{N} \) and:

(i) for each constant \( c \in \Sigma \), the element \( c^A \) is

\[
1 \in \mathbb{N};
\]

(ii) for each variable \( x \in X \), the element \( x^A \) is

\[
1 \in \mathbb{N};
\]

(iii) for each \( n \)-ary operation symbol \( f \in \Sigma \), the function \( f^A : \mathbb{N}^n \to \mathbb{N} \) is such that

\[
\text{max}(m_1, \ldots, m_n)
\]

**Size** Let

\[
\text{Size} : T(\Sigma, X) \to \mathbb{N}
\]

be the function that measures the size \( \text{Size}(t) \) of a term \( t \in T(\Sigma, X) \). This function is defined by structural induction as follows:

\[
\begin{align*}
\text{Size}(c) &= 1 \\
\text{Size}(x) &= 1 \\
\text{Size}(f(t_1, \ldots, t_n)) &= \text{Size}(t_1) + \cdots + \text{Size}(t_n) + 1
\end{align*}
\]

Comparing these equations to the general definition by structural induction, we see that \( A = \mathbb{N} \) and
12.2. **INDUCTION ON TERMS**

(i) for each constant $c \in \Sigma$, the element $c^A$ is

$$1 \in \mathbb{N};$$

(ii) for each variable $x \in X$, the element $x^A$ is

$$1 \in \mathbb{N};$$

(iii) for each $n$-ary operation symbol $f \in \Sigma$, the function $f^A : \mathbb{N}^n \to \mathbb{N}$ is such that

$$f^A(m_1, \ldots, m_n) = (m_1 + \cdots + m_n) + 1.$$

12.2.4 **Comparing Induction on Natural Numbers and Terms**

**Theorem** The following are equivalent:

1. The Principle of Induction on $\mathbb{N}$.

2. The Principle of Induction on $T(\Sigma, X)$ for any non-empty signature $\Sigma$.

**Proof.** First, we prove that (1) implies (2). Suppose (1) holds. We have to prove that the statement of structural induction holds for terms, using statement (1).

Let $\text{Height} : T(\Sigma, X) \to \mathbb{N}$ be the height function defined above. Let $P \subseteq T(\Sigma, X)$ be a property of terms.

Suppose

(i) $P(c)$ and $P(x)$ are true; and

(ii) if $P(t_1), \ldots, P(t_n)$ are true, then $P(f(t_1, \ldots, t_n))$ is true.

We show that $P(t)$ is true for all $t \in T(\Sigma, X)$ by induction on the height of terms.

**Basis**

All terms $t$ are of height 1: $\text{Height}(t) = 1$.

**Induction Step**

Let $k > 1$. Suppose $P(t)$ is true for all $t$ such that $\text{Height}(t) < k$. Now for any term $t$ with $\text{Height}(t) = k$, it has the form

$$t \equiv f(t_1, \ldots, t_n)$$

where $\text{Height}(t_1), \ldots, \text{Height}(t_n) < k$. Now, by the Induction Hypothesis, we know

$$P(t_1), \ldots, P(t_n)$$

are true.

So, by the Principle of Induction on $\mathbb{N}$,

$$P(t)$$

is true.

For a proof that (2) implies (1), see the exercises at the end of this chapter.
12.3 Terms and Trees

Terms can be displayed as trees. We demonstrate the process in the simple single-sorted case, before generalising to the many-sorted case.

12.3.1 Single-Sorted Trees

**Definition** Consider the single-sorted signature \( \Sigma_{SS} \). Let \( X \) be an \( \{s\} \)-sorted set of variable symbols. Let \( \text{Tree}(\Sigma_{SS}, X) \) be a set of trees whose nodes are labelled by constants, variables or function symbols, i.e., by elements of \( \Sigma_{SS} \cup X \). We map the terms of the term algebra into trees by the map

\[
\text{Tr} : T(\Sigma_{SS}, X) \to \text{Tree}(\Sigma_{SS}, X)
\]

which is defined as follows:

(i) for each constant symbol \( c : s \) in \( \Sigma \), the tree

\[
\text{Tr}(c) = c
\]

i.e., a single node labelled by \( c \);

(ii) for each variable symbol \( x \in X \), the tree

\[
\text{Tr}(x) = x
\]

i.e., a single node labelled by \( x \); and

(iii) for each function symbol \( f : s^n \to s \) in \( \Sigma \) and any terms \( t_1, \ldots, t_n \in T(\Sigma, X) \), the tree \( \text{Tr}(f(t_1, \ldots, t_n)) \) consists of a node labelled by \( f \), and \( n \) edges, to which the sub-trees \( \text{Tr}(t_1), \ldots, \text{Tr}(t_n) \) are attached by their roots. This is shown in Figure 12.1.

![Figure 12.1: The tree \( \text{Tr}(f(t_1, \ldots, t_n)) \).](image)

**Example**

1. Recall the single-sorted term algebra \( T(\Sigma_{\text{Natural}s}, \emptyset) \) of Example 12.1.1. Figure 12.2 shows some terms and their tree representation.

2. Recall the term algebra \( T(\Sigma_{\text{Natural}s}, X) \) of Example 12.1.1. Figure 12.3 shows some terms and their tree representation.
12.3. TERMS AND TREES

\begin{figure}[h]
\centering
\begin{align*}
\text{succ} & \quad & \text{add} \\
\mid & & \mid \\
\text{succ} & \quad & \text{zero} \\
\mid & & \mid \\
\text{zero} & & \text{succ}
\end{align*}
\begin{align*}
\text{Tr}(\text{succ}(\text{succ}(\text{zero}))) & & \text{Tr}(\text{add}(\text{zero}, \text{succ}(\text{zero})))
\end{align*}
\caption{Tree representation of some terms of the algebra $T(\Sigma_{\text{Naturals}}, \emptyset)$.}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{succ} & \quad & \text{add} \\
\mid & & \mid \\
\mid & & \mid \\
\text{succ} & \quad & \text{succ} \\
\mid & & \mid \\
\text{succ} & \quad & \text{zero}
\end{align*}
\begin{align*}
\text{Tr}(\text{succ}^n(x)) & & \text{Tr}(\text{add}(x, \text{mult}(y, \text{succ}(\text{zero}))))
\end{align*}
\caption{Tree representation of some terms of the algebra $T(\Sigma^{\text{Naturals}}, X)$.}
\end{figure}

12.3.2 Many-Sorted Trees

We now consider the many-sorted case.

**Definition** Let $\Sigma$ be an $S$-sorted signature and $X$ an $S$-sorted set of variable symbols. Let $\text{Tree}(\Sigma, X)$ be a set of trees. We map the terms of the term algebra into trees by the maps

\[ \ldots, \text{Tr}_s : T(\Sigma, X)_s \to \text{Tree}(\Sigma, X)_s, \ldots \]

which is defined as follows:

(i) for each constant symbol $c : \to s$ in $\Sigma$, the tree

\[ \text{Tr}_s(c) = \cdot c \]

i.e., a single node labelled by $c$;

(ii) for each variable symbol $x \in X_s$, the tree

\[ \text{Tr}_s(x) = \cdot x \]

i.e., a single node labelled by $x$; and

(iii) for each function symbol $f : s(1) \times \cdots \times s(n) \to s$, and any terms $t_1 \in T(\Sigma, X)_{s(1)}$, $\ldots$, $t_n \in T(\Sigma, X)_{s(n)}$, the tree $\text{Tr}_s(f(t_1, \ldots, t_n))$ consists of a node labelled by $f$, and $n$ edges, to which the sub-trees $\text{Tr}_{s(1)}(t_1), \ldots, \text{Tr}_{s(n)}(t_n)$ are attached by their roots. This is shown in Figure 12.4.

**Example**

Recall the \{nat, Bool\}-sorted term algebra $T(\Sigma_{\text{Naturals}} \text{ with } \text{Tests}, X)$ of Example 12.1.2. Figure 12.5 shows some terms and their tree representation.
12.4 Term Evaluation

A term over a signature is a composition of constants, variables and operations. It denotes a composite operation (if it has variables) or an element (if it does not have variables). We will now show how terms are evaluated in a set of data. We again need to use induction.

The semantics of terms is given by a set $A$ and a map

$$\bar{v} : T(\Sigma, X) \rightarrow A,$$

where $\bar{v}(t)$ is the semantics or value of term $t$. To calculate $\bar{v}$ we must

(i) interpret the constant symbols by elements of $A$;

(ii) interpret the operation symbols by maps on $A$; and

(iii) assign elements of $A$ to variables.

First we describe the simplest case where $\Sigma$ is a single-sorted signature.

12.4.1 Single-Sorted Term Evaluation

Let $c$ be a constant symbol. We interpret $c$ by an element $c^A \in A$. Let $f$ be an $n$-ary operation symbol. We interpret $f$ by an operation $f^A : A^n \rightarrow A$. Clearly we have an algebra:
### 12.4. TERM EVALUATION

| algebra   | $A$ |
| carriers | $A$ |
| constants | $\ldots, c^A : \rightarrow A, \ldots$ |
| operations | $\ldots, f^A : A^n \rightarrow A, \ldots$ |

With respect to these fixed interpretations, term evaluation (in a call by value style) is given by the following.

Given an assignment

$$v : X \rightarrow A$$

of an element $v(x)$ to each variable $x \in X$, we define the evaluation map

$$\overline{v} : T(\Sigma, X) \rightarrow A$$

by induction on the structure of terms:

$$\begin{align*}
\overline{v}(c) &= c_A \\
\overline{v}(x) &= v(x) \\
\overline{v}(f(t_1, \ldots, t_n)) &= f_A(\overline{v}(t_1), \ldots, \overline{v}(t_n)).
\end{align*}$$

#### 12.4.2 Many-Sorted Term Evaluation

We now extend this definition to a many-sorted signature $\Sigma$. Let $S$ be a non-empty set and let $\Sigma$ be an $S$-sorted signature.

Let $c : \rightarrow s$ be a constant symbol in $\Sigma$. We interpret $c$ by an element $c^A_s \in A_s$. Let $f : s(1) \times \cdots \times s(n) \rightarrow s$ be an operation symbol of $\Sigma$. We interpret $f$ by an operation $f^A : A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow A_s$. Clearly we have a $\Sigma$-algebra:

| algebra   | $A$ |
| carriers | $A_{s(1)}, \ldots, A_{s(n)}$ |
| constants | $\ldots, c^A_s : \rightarrow A_s, \ldots$ |
| operations | $\ldots, f^A_s : A^w \rightarrow A_s, \ldots$ |

Given assignments

$$\ldots, v_s : X_s \rightarrow A_s, \ldots$$

of elements $\ldots, v_s(x), \ldots$ to variables $\ldots, x_s \in X_s, \ldots$, we define the evaluation maps

$$\ldots, \overline{v}_s : T(\Sigma, X)_s \rightarrow A_s, \ldots$$
by induction on the structure of terms:

\[
\begin{align*}
\varphi_s(c) &= c_A \\
\varphi_s(x) &= v(x) \\
\varphi_s(f(t_1, \ldots, t_n)) &= f_A(\varphi_s(t_1), \ldots, \varphi_s(t_n)).
\end{align*}
\]

Later we will see the construction of \( \varphi \) from \( v \) as an important idea generalising the Principle of Induction to a general algebraic setting through the concepts of freeness and initiality. For the moment let us refer to it as an extension property.

### 12.4.3 Term Substitution

Another operation on \( T(\Sigma, X) \) worth noting is that of term substitution (which allows us to define call by name). Let \( t \) be a term, \( \overline{x} = x_1, \ldots, x_n \) be a sequence of variable symbols and \( \overline{t} = t_1, \ldots, t_n \) be a sequence of terms; we wish to define the term customarily denoted

\[
t(\overline{x}/\overline{t}) \text{ or } t(x_1/t_1, \ldots, x_n/t_n)
\]

obtained by substituting the term \( t_i \) for the variable symbol \( x_i \), for each \( i = 1, \ldots, n \), throughout \( t \). This is trivially done by the extension property. Given \( \overline{x} = x_1, \ldots, x_n \) and \( \overline{t} = t_1, \ldots, t_n \) we define an assignment

\[
v = v(\overline{x}, \overline{t}) : X \to T(\Sigma, X)
\]

by

\[
v(x) = \begin{cases} 
  x & \text{if } x \notin \{x_1, \ldots, x_n\}; \\
  t_i & \text{if } x = x_i.
\end{cases}
\]

Then by the method of extending \( v \) to \( \varphi \) we obtain

\[
\varphi : T(\Sigma, X) \to T(\Sigma, X)
\]

which carries out the required substitution of \( t_i \) for \( x_i \), for \( i = 1, \ldots, n \), for all terms.

This can be refined further by a new map

\[
\text{sub}^n : T(\Sigma, X) \times X^n \times T(\Sigma, X)^n \to T(\Sigma, X)
\]

which substitutes any \( n \)-tuple \( \overline{t} = t_1, \ldots, t_n \) of terms for any \( n \)-tuple \( \overline{x} = x_1, \ldots, x_n \) of variables into a term \( t \). This is defined as follows: given \( \overline{x} = x_1, \ldots, x_n \) and \( \overline{t} = t_1, \ldots, t_n \) we define the assignment \( v(\overline{x}, \overline{t}) : X \to T(\Sigma, X) \) and uniformising \( v \) we define

\[
\text{sub}^n(t, \overline{x}, \overline{t}) = \varphi(t).
\]

The extension property can also be used to define a change in variable names. Let \( X \) and \( Y \) be sets of variables of the same cardinality. Suppose \( X = \{x_i | i \in I\} \) and \( Y = \{y_i | i \in I\} \). We can define the effect on terms of the transformation of variables from \( X \) to \( Y \) as follows. Consider the term algebras \( T(\Sigma, X) \) and \( T(\Sigma, Y) \). On choosing a variable transformation as an assignment \( v : X \to Y \) which is a bijection say \( v(x_i) = y_i \) we obtain, by the extension property, a map \( \varphi : T(\Sigma, X) \to T(\Sigma, Y) \) that transforms all the terms. It is possible to prove that \( \varphi \) is a bijection.
12.5 Term Algebras

The process of building terms is algebraic. For example, for each operation symbol $f$, given terms $t_1, \ldots, t_n$ we construct the new term $f(t_1, \ldots, t_n)$. We are using a term constructor operation $F$ that creates the new term,

$$F(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

This observation leads us to create an algebra of terms by adding term constructor operations. We will go through the construction twice, for both single- and many-sorted terms. In each case, for any signature $\Sigma$, we turn the set $T(\Sigma, X)$ of terms into a $\Sigma$-algebra.

12.5.1 Single-Sorted Term Algebras

**Definition** The algebra $T(\Sigma_{SS}, X)$ is defined to have:

(i) the carrier set $T(\Sigma_{SS}, X)$;

(ii) for each constant symbol $c$ in the signature $\Sigma_{SS}$, there is a constant $c^{T(\Sigma_{SS}, X)} : T(\Sigma_{SS}, X)$ which is defined by

$$c^{T(\Sigma_{SS}, X)} = c;$$

(iii) and for each $n$-ary function symbol $f : s^n \to s$ in the signature $\Sigma_{SS}$ and any terms $t_1, \ldots, t_n \in T(\Sigma_{SS}, X)$, there is a function $f^{T(\Sigma_{SS})} : (T(\Sigma_{SS}, X))^n \to T(\Sigma_{SS}, X)$ defined by

$$f^{T(\Sigma_{SS})}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$

We display the algebra as follows:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$\Sigma_{SS}$-terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$T(\Sigma_{SS}, X)$</td>
</tr>
<tr>
<td>constants</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$c^{T(\Sigma_{SS}, X)} : T(\Sigma_{SS}, X)$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td>operations</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$f^{T(\Sigma_{SS})} : T(\Sigma_{SS}, X)^n \to T(\Sigma_{SS}, X)$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td></td>
</tr>
</tbody>
</table>
Example Consider again the signature $\Sigma_{\text{Natural}s}$ of Section 12.1.1 and its $\Sigma_{\text{Natural}s}$-algebra of terms:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$\Sigma_{\text{Natural}s}$-terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$T(\Sigma_{\text{Natural}s}, X)$</td>
</tr>
<tr>
<td>constants</td>
<td>$0^T(\Sigma_{\text{Natural}s}, X) : \rightarrow T(\Sigma_{\text{Natural}s}, X)$</td>
</tr>
<tr>
<td>operations</td>
<td>$\text{succ}^T(\Sigma_{\text{Natural}s}, X) : T(\Sigma_{\text{Natural}s}, X)^n \rightarrow T(\Sigma_{\text{Natural}s}, X)$</td>
</tr>
<tr>
<td></td>
<td>$\text{add}^T(\Sigma_{\text{Natural}s}, X) : T(\Sigma_{\text{Natural}s}, X)^n \rightarrow T(\Sigma_{\text{Natural}s}, X)$</td>
</tr>
<tr>
<td></td>
<td>$\text{mult}^T(\Sigma_{\text{Natural}s}, X) : T(\Sigma_{\text{Natural}s}, X)^n \rightarrow T(\Sigma_{\text{Natural}s}, X)$</td>
</tr>
</tbody>
</table>

Here, given any terms $t, t_1, t_2 \in T(\Sigma_{\text{Natural}s}, X)$, for example we can take:

$$0^T(\Sigma_{\text{Natural}s}, X) = 0$$

$$\text{succ}^T(\Sigma_{\text{Natural}s}, X)(t) = \text{succ}(t)$$

$$\text{add}^T(\Sigma_{\text{Natural}s}, X)(t_1, t_2) = \text{add}(t_1, t_2)$$

$$\text{mult}^T(\Sigma_{\text{Natural}s}, X)(t_1, t_2) = \text{mult}(t_1, t_2)$$

Thus,

$$\text{add}^T(\Sigma_{\text{Natural}s}, X)(\text{mult}(x, \text{succ}(y)), \text{succ}(\text{succ}(y))) = \text{add}(\text{mult}(x, \text{succ}(y)), \text{succ}(\text{succ}(y))).$$

12.5.2 Many-Sorted Term Algebras

Now we extend our single-sorted definition of Section 12.5.1, to many sorts.

Definition The algebra $T(\Sigma_{MS}, X)$ is defined to have:

(i) the carrier sets $\ldots, T(\Sigma_{MS}, X)_s, \ldots$;

(ii) for each constant symbol $c : \rightarrow s$ in the signature $\Sigma_{MS}$, there is a constant

$$c^T(\Sigma_{MS}, X) : \rightarrow T(\Sigma_{MS}, X)_s$$

which is defined by

$$c^T(\Sigma_{MS}, X) = c;$$

(iii) and for each function symbol $f : s(1) \times \cdots \times s(n) \rightarrow s$ in the signature $\Sigma_{MS}$ there is a function

$$f^T(\Sigma_{MS}, X) : (T(\Sigma_{MS}, X))_{s(1)} \times \cdots \times (T(\Sigma_{MS}, X))_{s(n)} \rightarrow T(\Sigma_{MS}, X)_s$$

which given any terms $t_1 \in T(\Sigma_{MS}, X)_{s(1)}, \ldots, t_n \in T(\Sigma_{MS}, X)_{s(n)}$, is defined by

$$f^T(\Sigma_{MS}, X)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n).$$
12.5. **TERM ALGEBRAS**

We display the algebra as follows:

```
<table>
<thead>
<tr>
<th>algebra</th>
<th>$\Sigma_{MS}$-terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$\ldots, T(\Sigma_{MS}, X)_{s}, \ldots$</td>
</tr>
<tr>
<td>constants</td>
<td>( c^T(\Sigma_{MS}, X) : T(\Sigma_{MS}, X)<em>s \rightarrow T(\Sigma</em>{MS}, X)_s )</td>
</tr>
<tr>
<td>operations</td>
<td>( f^T(\Sigma_{MS}, X) : T(\Sigma_{MS}, X)<em>{s(1)} \times \cdots \times T(\Sigma</em>{MS}, X)<em>{s(n)} \rightarrow T(\Sigma</em>{MS}, X)_s )</td>
</tr>
</tbody>
</table>
```

**Example** Consider again the signature $\Sigma_{\text{Naturals with Tests}}$ of Section 12.1.1 and its $\Sigma_{\text{Naturals}}$-algebra of terms:
algebra \( \Sigma_{\text{Naturals with Tests, } X} \) with Test-terms

carriers
\[ \begin{align*}
T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}}
\end{align*} \]

constants
\[ \begin{align*}
\text{zero}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
\text{true}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \\
\text{false}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}}
\end{align*} \]

operations
\[ \begin{align*}
\text{succ}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
\text{pred}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
\text{add}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad (T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}})^2 \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
\text{mult}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad (T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}})^2 \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}} \\
\text{equals}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad (T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}})^2 \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \\
\text{lessThan}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad (T(\Sigma_{\text{Naturals with Tests, } X})_{\text{nat}})^2 \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \\
\text{not}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \\
\text{and}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}} \\
\text{or}^{T(\Sigma_{\text{Naturals with Tests, } X})} : & \quad (T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}})^2 \rightarrow T(\Sigma_{\text{Naturals with Tests, } X})_{\text{Bool}}
\end{align*} \]

12.6 Homomorphisms

12.6.1 Structural Induction, Term Evaluation and Homomorphisms

The data type of \( \Sigma \)-terms is modelled by the term algebra \( T(\Sigma, X) \). We now reconsider structural induction on terms. Speaking roughly, the following are equivalent:
• structural induction on $T(\Sigma, X)$;
• term evaluation on $T(\Sigma, X)$; and 
• homomorphism on $T(\Sigma, X)$.

Let $T(\Sigma, X)$ be the algebra of $\Sigma$-terms. Let $A$ be a set. We say the function

$$v : T(\Sigma, X) \to A$$

is defined by structural induction or recursion from elements $\ldots, c^A, \ldots, f^A, \ldots$ from $A$ and assignment $\sigma : X \to A$ if:

$$v(c) = c^A$$
$$v(x) = \sigma(x)$$
$$v(f(t_1, \ldots, t_n)) = f^A(v(t_1), \ldots, v(t_n)).$$

This special type of recursive definition is a form of term evaluation (see Section 12.4).

Let us note that the important equations that are used in the definition of the extension

$$\overline{v} : T(\Sigma, X) \to A$$

of

$$v : X \to A$$

can be rewritten so as to involve formally the algebraic structure of the algebra of terms $T(\Sigma, X)$, as follows:

$$\overline{v}(c^{T(\Sigma,X)}) = \overline{v}(c) = c^A$$
$$\overline{v}(x) = \overline{v}(x) = v(x)$$
$$\overline{v}(f^{T(\Sigma,X)}(t_1, \ldots, t_n)) = \overline{v}(f(t_1, \ldots, t_n)) = f^A(\overline{v}(t_1), \ldots, \overline{v}(t_n))$$

We see that the map $\overline{v}$ preserves a relationship between the operations on the term algebra $T(\Sigma, X)$ and on the semantic algebra $A$. This structure preserving mapping provides another example of a homomorphism between algebras.

**Lemma (Term Evaluation is a Homomorphism)** Let $A$ be a $\Sigma$-algebra. The term evaluation function

$$\overline{v} : T(\Sigma, X) \to A$$

is a $\Sigma$-homomorphism.
There are, in fact, some more properties hidden in this simple observation — properties that turn out to have profound consequences in the theory of syntax and semantics. We reformulate the observation very carefully.

**Theorem (Initiality of Term Algebras)** Let $A$ be any $\Sigma$-algebra. Let $v : X \to A$ be any map assigning values in $A$ to the variables in $X$. Then there is a one, and only one, $\Sigma$-homomorphism

$$\overline{v} : T(\Sigma, X) \to A$$

which extends $v$ from $X$ to $T(\Sigma, X)$.

**Proof** The existence of a $\Sigma$-homomorphism is clear from the equations for term evaluation. We have to prove that there is only one $\Sigma$-homomorphism, i.e., that $\overline{v}$ is unique.

Suppose that $\phi, \psi : T(\Sigma, X) \to A$ are two $\Sigma$-homomorphisms that both extend the assignment map $v : X \to A$. This means that for every variable $x \in X$,

$$\phi(x) = \psi(x) (= v(x)).$$

We prove that for every $\Sigma$-term $t \in T(\Sigma, X)$,

$$\phi(t) = \psi(t) (= \overline{v}(t)).$$

We do this by structural induction on terms.

**Basis**

(i) Constants, $t \equiv c$. Here, since $\phi$ and $\psi$ are $\Sigma$-homomorphisms, they preserve constants:

$$\phi(c) = c^A \quad \text{and} \quad \psi(c) = c^A$$

and $\phi(t) = \psi(t)$.

(ii) Variables, $t \equiv x$. Here, since $\phi$ and $\psi$ extend $v$, we have:

$$\phi(x) = v(x) \quad \text{and} \quad \psi(x) = v(x)$$

and $\phi(t) = \psi(t)$.

**Induction Step** In the general case, we consider any $\Sigma$-term, say $t \equiv f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are $\Sigma$-subterms. The Induction Hypothesis is that:

$$\phi(t_1) = \psi(t_1), \ldots, \phi(t_n) = \psi(t_n).$$

We calculate:

$$\begin{align*}
\phi(t) &= \phi(f(t_1, \ldots, t_n)) \\
&= f^A(\phi(t_1), \ldots, \phi(t_n)) \quad \text{since $\phi$ is a $\Sigma$-homomorphism} \\
&= f^A(\psi(t_1), \ldots, \psi(t_n)) \quad \text{since $\phi = \psi$ on subterms by Induction Hypothesis} \\
&= \psi(f(t_1, \ldots, t_n)) \quad \text{since $\psi$ is a $\Sigma$-homomorphism} \\
&= \psi(t).
\end{align*}$$

By the Principle of Structural Induction, we have that $\phi(t) = \psi(t)$ for all $t \in T(\Sigma, X)$. □
12.6.2 Representing Algebras using Terms

Let us review more closely the rôle of the term evaluation homomorphism. Given an assignment

\[ v : X \to A \]

data from \( \Sigma \)-algebra \( A \) to variables from \( X \), each term \( t \in T(\Sigma, X) \) can be evaluated as \( \overline{v}(t) \in A \). The term \( t \) defines a sequence of operations from \( \Sigma \) that applied in the \( \Sigma \)-algebra \( A \), constructs the element \( \overline{v}(t) \in A \); \( t \) represents \( \overline{v}(t) \), given the assignment \( v \).

Two terms \( t_1, t_2 \in T(\Sigma, X) \) can represent different constructions of the same element if \( \overline{v}(t_1) = \overline{v}(t_2) \) in \( A \). Since \( \overline{v} \) is a \( \Sigma \)-homomorphism, we know that if

\[ t_1, \ldots, t_n \text{ represents } \overline{v}(t_1), \ldots, \overline{v}(t_n) \]

then

\[ f(t_1, \ldots, t_n) \text{ represents } \overline{v}(f(t_1, \ldots, t_n)) \]

because of the \( \Sigma \)-homomorphism equation for the operation symbol \( f \in \Sigma \):

\[ \overline{v}(f(t_1, \ldots, t_n)) = f(\overline{v}(t_1), \overline{v}(t_n)) \]

**Definition (Generator)** The assignment \( v : X \to A \) generates the \( \Sigma \)-algebra \( A \) if, for all \( a \in A \), there is some \( t \in T(\Sigma, X) \) such that,

\[ \overline{v}(t) = a. \]

Equivalently, \( v \) generates \( A \) if the term evaluation \( \Sigma \)-homomorphism is surjective.

**Theorem** Let \( A \) be any \( \Sigma \)-algebra. Let \( X \) be a set of variables and \( v : X \to A \) an assignment. Suppose that \( v \) generates \( A \). Then \( A \) is \( \Sigma \)-isomorphic to a factor algebra \( T(\Sigma, X)/\equiv_\sigma \) of the term algebra \( T(\Sigma, X) \). In particular, the congruence is the kernel \( \sim_\sigma \) of the term evaluation homomorphism, i.e.,

\[ A \cong T(\Sigma, X)/\equiv_\sigma \]

where for \( t_1, t_2 \in T(\Sigma, X) \),

\[ t_1 \equiv_\sigma t_2 \iff \overline{v}(t_1) = \overline{v}(t_2). \]

**Proof** This follows immediately from an application of the Homomorphism Theorem. \( \square \)

### 12.7 An Algebra of while Programs

We shall now consider a more abstract approach to programming language syntax that focuses on the key components in syntactical clauses that have a semantical meaning.

We reconsider the **while** programming language \( WP \) from a more abstract point of view and develop an algebraic model of its syntax. The language is based primarily on three types of syntactic object:

- commands;
- expressions; and
• Boolean expressions.

These are syntactic objects that have a meaning or semantics. In contrast, the grammar Flattened While Programs over Natural Numbers of Section 10.5.1 contains twenty-seven syntactic objects, not all of which have a meaning. Section 10.5.1 gives a concrete syntax for the while programming language, taking a low-level approach to syntax. We will redesign a simplified version of the language WP by developing an algebra $WP(\Sigma)$ for its abstract syntax.

### 12.7.1 Constructing an Algebra of while programs

We shall construct an algebra $WP(\Sigma)$ of while programs by collecting together sets of syntactic components and defining operations on them that build up the while programs.

Let $\Sigma$ be the signature of natural numbers with the Booleans:

<table>
<thead>
<tr>
<th>signature</th>
<th>$\Sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>$\text{nat, Bool}$</td>
</tr>
<tr>
<td>constants</td>
<td>$0 : \rightarrow \text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{true, false} : \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td>operations</td>
<td>$\text{plus : nat x nat} \rightarrow \text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{times : nat x nat} \rightarrow \text{nat}$</td>
</tr>
<tr>
<td></td>
<td>$\text{eq : nat x nat} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td></td>
<td>$\text{not : Bool} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td></td>
<td>$\text{and : Bool x Bool} \rightarrow \text{Bool}$</td>
</tr>
<tr>
<td></td>
<td>$\text{or : Bool x Bool} \rightarrow \text{Bool}$</td>
</tr>
</tbody>
</table>

endsig

Let $\text{Comm}(\Sigma)$ $\text{Exp}(\Sigma)$ $\text{BExp}(\Sigma)$ be sets of commands, expressions and Boolean expressions over the signature $\Sigma$. These sets, together with $\text{N}$ and $\text{Var} = \{x_0, x_1, x_2, \ldots\}$ will be the carriers of the algebra $WP(\Sigma)$ of the language of while programs over $\Sigma$. To build all the syntactic components of the language, we define suitable functions over these carriers, to define the set of commands, expressions and Boolean expressions.

First, we note that a while program consists of commands over expressions and Boolean expressions. A while program is formed from:

• the atomic programs which consist of
  • empty programs; and
  • assignments;
and are combined together with

- the program forming constructs of
  - sequencing;
  - conditional branching; and
  - iteration.

Atomic Programs

The atomic programs are the constant

\[
\text{skip} : \rightarrow \text{Comm}(\Sigma)
\]

corresponding to the “identity” or “empty” program, and the assignments constructed by the operation

\[
\text{Assign} : \text{Var} \times \text{Exp}(\Sigma) \rightarrow \text{Comm}(\Sigma)
\]

\[
\text{Assign}(v, e) = v := e
\]

from variables and expressions.

Program Operations

Thus, the set \(\text{Comm}(\Sigma)\) of commands is given by applying the three operations

\[
\text{Seq} : \text{Comm}(\Sigma) \times \text{Comm}(\Sigma) \rightarrow \text{Comm}(\Sigma)
\]

\[
\text{Seq}(S_1, S_2) = S_1; S_2
\]

\[
\text{Cond} : \text{BEExp}(\Sigma) \times \text{Comm}(\Sigma) \times \text{Comm}(\Sigma) \rightarrow \text{Comm}(\Sigma)
\]

\[
\text{Cond}(b, S_1, S_2) = \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}
\]

\[
\text{Iter} : \text{BEExp}(\Sigma) \times \text{Comm}(\Sigma) \rightarrow \text{Comm}(\Sigma)
\]

\[
\text{Iter}(b, S_0) = \text{while } b \text{ do } S_0 \text{ od}
\]

that model the program forming constructs.

Building Expressions

An atomic expression is a number or an identifier. More complex expressions can be built up from adding or multiplying two more basic expressions. Thus, the set \(\text{Exp}(\Sigma)\) of expressions is given by the operations

\[
\text{Id} : \text{Var} \rightarrow \text{Exp}(\Sigma)
\]

\[
\text{Id}(v) = v
\]

\[
\text{Num} : \mathbb{N} \rightarrow \text{Exp}(\Sigma)
\]

\[
\text{Num}(n) = n
\]
for atomic expressions, and

\[
Add : Exp(\Sigma) \times Exp(\Sigma) \rightarrow Exp(\Sigma)
\]
\[
Add(e_1, e_2) = e_1 + e_2
\]

\[
Times : Exp(\Sigma) \times Exp(\Sigma) \rightarrow Exp(\Sigma)
\]
\[
Times(e_1, e_2) = e_1 \times e_2
\]

for the expression forming constructs.

**Building Tests**

Finally, we describe the nature of the algebra \(BExp(\Sigma)\) of tests. A Boolean expression can be

- true or false;
- a comparison of two expressions; or
- some Boolean combination of a less complex Boolean expression.

We model this situation by the constants

\[
\text{true} \rightarrow BExp(\Sigma)
\]
\[
\text{false} \rightarrow BExp(\Sigma)
\]

and the operations:

\[
Eq : Exp(\Sigma) \times Exp(\Sigma) \rightarrow BExp(\Sigma)
\]
\[
Eq(e_1, e_2) = e_1 = e_2
\]

\[
\text{Not} : BExp(\Sigma) \rightarrow BExp(\Sigma)
\]
\[
\text{Not}(b) = \text{not } b
\]

\[
\text{And} : BExp(\Sigma) \times BExp(\Sigma) \rightarrow BExp(\Sigma)
\]
\[
\text{And}(b_1, b_2) = b_1 \text{ and } b_2
\]

\[
\text{Or} : BExp(\Sigma) \times BExp(\Sigma) \rightarrow BExp(\Sigma)
\]
\[
\text{Or}(b_1, b_2) = b_1 \text{ or } b_2
\]

**Variables**

We describe the set \(Var\) of variables as the constants

\[
\ldots x_0, x_1, x_2, \ldots \rightarrow Var.
\]
### 12.7. An Algebra of While Programs

<table>
<thead>
<tr>
<th>algebra</th>
<th>$WP(\Sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$Comm(\Sigma)$ $Exp(\Sigma)$ $BExp(\Sigma)$ $N$ $Var$</td>
</tr>
<tr>
<td>constants</td>
<td>skip $\rightarrow$ $Comm(\Sigma)$ true, false $\rightarrow$ $BExp(\Sigma)$ $x_0, x_1, x_2, \ldots$ $\rightarrow$ $Var$ $0, 1, 2, \ldots$ $\rightarrow$ $N$</td>
</tr>
<tr>
<td>operations</td>
<td>$Assign : Var \times Exp(\Sigma) \rightarrow Comm(\Sigma)$ $Seq : Comm(\Sigma)^2 \rightarrow Comm(\Sigma)$ $Cond : BExp(\Sigma) \times Comm(\Sigma)^2 \rightarrow Comm(\Sigma)$ $Iter : BExp(\Sigma) \times Exp(\Sigma) \rightarrow Exp(\Sigma)$ $Id : Var \rightarrow Exp(\Sigma)$ $Num : N \rightarrow Exp(\Sigma)$ $Add : Exp(\Sigma) \times Exp(\Sigma) \rightarrow Exp(\Sigma)$ $Times : Exp(\Sigma) \times Exp(\Sigma) \rightarrow Exp(\Sigma)$ $Not : BExp(\Sigma) \rightarrow BExp(\Sigma)$ $Or : BExp(\Sigma) \times BExp(\Sigma) \rightarrow BExp(\Sigma)$ $And : BExp(\Sigma) \times BExp(\Sigma) \rightarrow BExp(\Sigma)$ $Eq : Exp(\Sigma) \times Exp(\Sigma) \rightarrow BExp(\Sigma)$</td>
</tr>
<tr>
<td>definitions</td>
<td>$Assign(v, e) = v := e$ $Seq(S_1, S_2) = S_1 ; S_2$ $Cond(b, S_1, S_2) = if b then S_1 else S_2 fi$ $Iter(b, S_0) = while b do S_0 od$ $Id(v) = v$ $Num(n) = n$ $Add(e_1, e_2) = e_1 + e_2$ $Mult(e_1, e_2) = e_1 * e_2$ $Not(b) = \text{not } b$ $Or(b_1, b_2) = b_1 \text{ or } b_2$ $And(b_1, b_2) = b_1 \text{ and } b_2$ $Eq(e_1, e_2) = e_1 = e_2$</td>
</tr>
</tbody>
</table>

Figure 12.6: The algebra $WP(\Sigma)$ of while programs and summary of definitions of operations.
12.7.2 Representing while programs as terms

Let us equip this algebra with the signature $\Sigma_{WP}$.

\[
\begin{array}{ll}
\text{signature} & WP \\
\text{sorts} & \text{comm} \ exp \ bexp \ nat \ var \\
\text{constants} & \text{skip}: \rightarrow \text{comm} \\
& \text{true, false}: \rightarrow \text{bexp} \\
& x_1, x_2, \ldots: \rightarrow \text{var} \\
& 0, 1, 2, \ldots: \rightarrow \text{nat} \\
\text{operations} & \text{assign}: \text{var} \times \text{exp} \rightarrow \text{comm} \\
& \text{seq}: \text{comm}^2 \rightarrow \text{comm} \\
& \text{cond}: \text{bexp} \times \text{comm}^2 \rightarrow \text{comm} \\
& \text{iter}: \text{bexp} \times \text{comm} \rightarrow \text{comm} \\
& \text{id}: \text{var} \rightarrow \text{exp} \\
& \text{num}: \text{nat} \rightarrow \text{exp} \\
& \text{add}: \text{exp} \times \text{exp} \rightarrow \text{exp} \\
& \text{times}: \text{exp} \times \text{exp} \rightarrow \text{exp} \\
& \text{not}: \text{bexp} \rightarrow \text{bexp} \\
& \text{or}: \text{bexp} \times \text{bexp} \rightarrow \text{bexp} \\
& \text{and}: \text{bexp} \times \text{bexp} \rightarrow \text{bexp} \\
& \text{eq}: \text{exp} \times \text{exp} \rightarrow \text{bexp} \\
\end{array}
\]

Thus, we say $WP(\Sigma)$ is a $\Sigma_{WP}$ algebra. We can construct terms over $\Sigma_{WP}$ in the standard way. First, consider the algebra

\[T(\Sigma_{WP})\]

of closed terms. Typical examples are

\[\text{assign}(x_1, \text{add}(\text{id}(x_2), \text{id}(x_3)))\]
\[\text{iter}(\text{not}(\text{eq}(\text{id}(x_1), \text{num}(0))), \text{assign}(x_1, \text{add}(\text{id}(x_1), \text{num}(1))))\]

These terms correspond with the **while** programs:

\[x_1 := x_2 + x_3\]
\[\text{while not } (x_1 = 0) \text{ do } x_1 := x_1 + 1 \text{ od}\]

This correspondence can be further explored by examples and the representations seem to be equivalent. But to understand it properly we are led to define a map

\[\phi: T(\Sigma_{WP}) \rightarrow WP(\Sigma)\]

that interprets terms as programs, so that, for example,

\[\phi(\text{assign}(x_1, \text{add}(\text{id}(x_2), \text{id}(x_3)))) = x_1 := x_2 + x_3\]
12.7. AN ALGEBRA OF WHILE PROGRAMS

In fact, \( \phi \) is defined by structural induction on terms, as follows:

\[
\begin{align*}
\phi(\text{skip}) & = \text{skip} \\
\phi(\text{assign}(x,e)) & = x := e \\
\phi(\text{semi}(S_1,S_2)) & = S_1; S_2 \\
\phi(\text{cond}(b,S_1,S_2)) & = \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
\phi(\text{iter}(b,S_0)) & = \text{while } b \text{ do } S_0 \text{ od}
\end{align*}
\]

This map turns out to be an example of:

(i) a term evaluation map \( \mathfrak{v} \) defined from a trivial assignment map

\[ v : \emptyset \to WP(\Sigma); \]

and

(ii) a \( \Sigma_{WP} \)-isomorphism.

Notice that each program can be constructed from \( Exp(\Sigma) \) and \( BExp(\Sigma) \) by applying the basic operations of the algebra (the algebra of while programs is minimal). This is a result of the following:

**Theorem** The algebra \( WP(\Sigma) \) of while programs over the signature \( \Sigma \) of naturals and Booleans is isomorphic with the closed term algebra of \( T(\Sigma_{WP}) \).

**Proof.** Exercise. \( \square \)
Example As the algebra $T(\Sigma_{\text{WP}})$ is a term algebra, we can represent program fragments as terms, which in turn we can represent as trees. For example:

1. We can represent the program fragment

$$x_2 := 3; \text{skip}$$

by the term

$$\text{seq}(\text{assign}(x_2, \text{num}(3)), \text{skip})$$

whose tree is shown in Figure 12.7.

![Tree representation of the term $x_2 := 3; \text{skip}$.]

2. We can represent the program fragment

$$\text{while not } x_0 = 2 \text{ do } x_0 := x_0 + 1 \text{ od}$$

by the term

$$\text{iter}(\text{not}(\text{eq}(\text{id}(x_0), \text{num}(2))), \text{assign}(x_0, \text{add}(\text{id}(x_0), \text{num}(1))))$$

whose tree is shown in Figure 12.8.

![Tree representation of the term $\text{while not } x_0 = 2 \text{ do } x_0 := x_0 + 1 \text{ od}$.]
12.8 Context free grammars and terms

The abstract syntax of while programs captured in the algebra $WP(\Sigma)$ can also be defined by a grammar, which has the same structure as the algebra. This turns out to be an example of a general pattern. In this section we study an algorithm that transforms any context free grammar $G$ into a signature $\Sigma^G$, and hence into a term algebra $T(\Sigma^G)$ which can also be used to define the language $L(G)$.

12.8.1 Outline of Algorithm

Recall that in a context free grammar $G = (T, N, S, P)$, the production rules $P$ are constrained to have the form

$$A \rightarrow w$$

where $A \in N$ is some non-terminal, and $w \in (T \cup N)^+$ is some string formed from terminal and/or non-terminal symbols.

We divide the production rules of the context free grammar into two distinct sets:

(i) those production rules which have no non-terminal symbols on the right-hand-side, i.e. they are of the form

$$A \rightarrow t_1 \cdots t_n$$

for $A \in N$, $t_1, \ldots, t_n \in T$; and

(ii) those production rules which have at least one variable symbol on the right-hand-side, i.e. they are of the form

$$A \rightarrow u_0 A_1 u_1 \cdots u_{n-1} A_n u_n$$

for $A, A_1, \ldots, A_n \in N$, $u_0, u_1, \ldots, u_n \in T^*$.

In a derivation of a string, when we use a rule of type (1) to replace a non-terminal in a string, that is the final step for that non-terminal: it has now been replaced by terminal symbols which cannot be rewritten. The only further possible derivation(s) for that string result from any other non-terminal symbols present in the string.

A rule of type (2) on the other hand, will always be an intermediate step in a derivation, with further derivations needed to rewrite the non-terminals which are introduced by the application of that production rule.

The rules of type (1) will give the constants in the algebra, and those of type (2), the functions. The sorts of each are determined by the variable symbols present in the production rules.

Because we use a context free grammar, the terminal symbols generated at any stage only determine the representation of the string that is generated, they do not have any influence on how the string is generated. Thus, we only use the terminal symbols to describe how we interpret the constant and function symbols.

12.8.2 Construction of a signature from a context free grammar

We now give a formal description of how we form a signature $\Sigma^G$ from the grammar $G$, from which we generate the term algebra $T(\Sigma^G)$. First, we need the following definition.
**Definition** For any context free grammar $G = (T, N, S, P)$ we inductively define a function

\[
\text{nonterminals} : (T \cup N)^* \rightarrow N^*
\]
on strings that returns all the non-terminal symbols present in a string $w \in (T \cup N)^*$, (possibly with repetitions) in the same order in which they originally appeared. So, a string $w$ contains no variable symbols if, and only if, $\text{nonterminals}(w) = \lambda$. We define the function $\text{nonterminals}$ for any $u \in (T \cup N)$, and $w \in (T \cup N)^*$ by

\[
\text{nonterminals}(\lambda) = \lambda
\]

\[
\text{nonterminals}(u.x) = \begin{cases} 
\text{nonterminals}(x) & \text{if } u \in T; \\
\text{u}.\text{nonterminals}(x) & \text{otherwise.}
\end{cases}
\]

**Example** Let $T = \{a, e, i, o, u\}$ be the set of vowels and $N = \{a, b, \ldots, z\} \setminus T$, the set of consonants. The function $\text{nonterminals} : (T \cup N)^* \rightarrow N^*$ removes terminal symbols from strings, so in this example it will remove all the vowels from a word, whilst leaving the consonants untouched. So, applying $\text{nonterminals}$ to some sample words over $(T \cup N)^*$ we get:

\[
\begin{align*}
\text{nonterminals}(\text{alphabet}) & = \text{lpht} \\
\text{nonterminals}(\text{swansea}) & = \text{wns} \\
\text{nonterminals}(\text{ouï}) & = \lambda \\
\text{nonterminals}(\text{rhythm}) & = \text{rhythm}
\end{align*}
\]

**Signature Construction Algorithm** Let $G = (T, N, S, P)$ be a context free grammar. Let each of the productions $A \rightarrow w$ of $P$ be labelled with a unique identifier $i$:

\[
i : A \rightarrow w
\]

We construct the signature $\Sigma^G$ as follows.

(i) We have the set $N$ of sorts being the non-terminal symbols of $G$.

(ii) For each production rule

\[
i : A \rightarrow w
\]
in $P$ with

\[
\text{nonterminals}(w) = \lambda
\]

we define a constant symbol

\[
i : \rightarrow A.
\]

(iii) For each production rule

\[
 j : A \rightarrow w
\]
in $P$ with

\[
\text{nonterminals}(w) = A_1 A_2 \cdots A_n
\]

we define a function symbol

\[
 j : A_1 \times \cdots \times A_n \rightarrow A.
\]
12.8. CONTEXT FREE GRAMMARS AND TERMS

12.8.3 Algebra \( T(\Sigma^G) \) of language terms

Given the signature \( \Sigma^G \), we now form the closed term algebra \( T(\Sigma^G) \) in the manner described in Section 12.5. Thus we get:

(i) for each production rule of the form
\[
i : A \rightarrow t_1 \cdots t_n
\]
for any variable symbol \( A \in N \) and any terminal symbols \( t_1, \ldots, t_n :\rightarrow T \), there will be a constant
\[
i \rightarrow T(\Sigma^G)_A;
\]
and

(ii) for each production rule of the form
\[
j : A \rightarrow u_0 A_1 u_1 \cdots u_{n-1} A_n u_n
\]
for any non-terminal symbols \( A, A_1, \ldots, A_n \in N \) and any strings of terminal symbols \( u_0, \ldots, u_n \in T^* \) there will be a function
\[
j : T(\Sigma) A_1 \times \cdots \times T(\Sigma) A_n \rightarrow T(\Sigma) A,
\]

We use a \( \Sigma^G \)-homomorphism \( \phi \) to interpret the terms of the term algebra as follows.

(i) Given a production rule
\[
i : A \rightarrow t_1 \cdots t_n
\]
in the grammar \( G \) which has given rise to a constant symbol
\[
i :\rightarrow A
\]
in the signature \( \Sigma^G \), we interpret the term \( i \) by:
\[
\phi(i) = t_1 \cdots t_n
\]

(ii) Given a production rule
\[
j : A \rightarrow u_0 A_1 u_1 \cdots u_{n-1} A_n u_n
\]
which has given rise to a function symbol
\[
j : T(\Sigma) A_1 \times \cdots \times T(\Sigma) A_n \rightarrow T(\Sigma) A,
\]
in the signature \( \Sigma^G \), and given terms \( a_1 \in T(\Sigma) A_1, \ldots, a_n \in T(\Sigma) A_n \), we interpret the term
\[
j(a_1, \ldots, a_n)
\]
by
\[
\phi(j(a_1, \ldots, a_n)) = u_0 a_1 u_1 \cdots u_{n-1} a_n u_n.
\]
User-Friendly Algebraic Model

In practice, we can typically use the production rules to give suggestive names to the constant and function symbols and to hide some of this technical machinery. A grammar $G$

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$\ldots, t, t_1, \ldots, t_n, \ldots$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$\ldots, A, A_0, A_1, \ldots, A_n, \ldots$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$A_0$</td>
</tr>
<tr>
<td>productions</td>
<td>$A \rightarrow t_1 \cdots t_n$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow u_0 A_1 u_1 \cdots u_{n-1} A_n u_n$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

where $u_0, \ldots, u_n \in T^*$ are strings of terminals gives rise to an algebraic model:

<table>
<thead>
<tr>
<th>algebra</th>
<th>$T(\Sigma^G)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers</td>
<td>$\ldots, T(\Sigma^G)<em>{A_0}, T(\Sigma^G)</em>{A_1}, \ldots, T(\Sigma^G)_{A_n}$</td>
</tr>
<tr>
<td>constants</td>
<td>$t_1 \cdots t_n \mapsto A$</td>
</tr>
<tr>
<td>的操作</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>operations</td>
<td>$u_0 A_1 u_1 \cdots u_{n-1} A_n : A_1 \times \cdots \times A_n \rightarrow A$</td>
</tr>
<tr>
<td></td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

12.8.4 Observation

This construction has the property that each term in the algebra $T(\Sigma^G)$ corresponds to a parse tree for each string that can be derived from the grammar $G$. In particular, for each nonterminal symbol $A \in N$, each string of the grammar derived from $A$ will have an associated term in the carrier set $T(\Sigma^G)_A$. 
Example Consider the context free grammar

\[ G = (\{a, b\}, \{S\}, S, P) \]

with labelled production rules \( P \)

\[
\begin{align*}
\alpha : & \ S \rightarrow ab \\
\beta : & \ S \rightarrow aSb
\end{align*}
\]

which generates the language \( L(G) = \{a^n b^n \mid n \geq 1\} \). We can transform \( G \) into the signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>( \Sigma^G )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>( S )</td>
</tr>
<tr>
<td>constants</td>
<td>( \alpha : \rightarrow S )</td>
</tr>
<tr>
<td>operations</td>
<td>( \beta : S \rightarrow S )</td>
</tr>
<tr>
<td>endsig</td>
<td></td>
</tr>
</tbody>
</table>

Thus, the signature \( \Sigma^G \) has

(i) a sort set which consists of the start symbol \( S \) (as this is the only variable symbol in the grammar);

(ii) a constant symbol \( \alpha : \rightarrow S \) which corresponds to the production \( \alpha : S \rightarrow ab \) in \( G \); and

(iii) a function symbol \( \beta : S \rightarrow S \) which corresponds to the production \( \beta : S \rightarrow aSb \) in \( G \).

For example, the derivation in the grammar \( G \):

\[
S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb
\]

corresponds to the term \( \beta(\beta(\alpha)) \in T(\Sigma^G) \). Figure 12.9 shows some sample parse trees for derivations in \( G \) with their corresponding terms and trees in the algebra \( T(\Sigma^G) \).
Parse trees for derivation. Corresponding term in $T(\Sigma^G)$.

\[
\begin{align*}
S & \\
& \xrightarrow{\beta} \alpha \\
& \xrightarrow{\beta^2} \beta^2(\alpha)
\end{align*}
\]

Figure 12.9: Derivations of sample phrases with their corresponding algebraic trees and terms.
12.8.5 Algebra of while programs revisited

Consider the following BNF for a simplified while language:

\[
\begin{align*}
<\text{statement}> &::= \skip | \\
& \quad <\text{identifier}> ::= <\exp> | \\
& \quad <\text{statement}> ; <\text{statement}> | \\
& \quad \text{if} <\text{Bool exp}> \text{ then } <\text{statement}> \text{ else } <\text{statement}> \text{ fi} | \\
& \quad \text{while} <\text{Bool exp}> \text{ do } <\text{statement}> \text{ od} \\
<\exp> &::= <\text{identifier}> | \\
& \quad <\text{number}> | \\
& \quad <\exp> + <\exp> | \\
& \quad <\exp> * <\exp> \\
<\text{Bool exp}> &::= \text{true} | \\
& \quad \text{false} | \\
& \quad \text{not} <\text{Bool exp}> | \\
& \quad <\text{Bool exp}> \text{ or } <\text{Bool exp}> | \\
& \quad <\text{Bool exp}> \text{ and } <\text{Bool exp}> | \\
& \quad <\exp> = <\exp> \\
<\text{identifier}> &::= x_0 \mid x_1 \mid x_2 \mid \ldots \\
<\text{number}> &::= 0 \mid 1 \mid 2 \mid \ldots
\end{align*}
\]

This gives us a context free grammar \( G^{WP} \). We can transform \( G^{WP} \) into a signature \( \Sigma^{WP} \):
signature  \( WP \)

sorts  
\(<\text{statement}>\), \(<\text{exp}>\), \(<\text{Bool}\ \text{exp}>\), \(<\text{identifier}>\), \(<\text{number}>\)

constants  
\( \text{skip} : \rightarrow <\text{statement}> \)
\( \text{true}, \text{false} : \rightarrow <\text{Bool}\ \text{exp}> \)
\( x_0, x_1, x_2, \ldots : \rightarrow <\text{identifier}> \)
\( 0, 1, 2, \ldots : \rightarrow <\text{number}> \)

operations  
\( \text{assign} : <\text{variable}> \times <\text{exp}> \rightarrow <\text{statement}> \)
\( \text{seq} : <\text{statement}> \times <\text{statement}> \rightarrow <\text{statement}> \)
\( \text{cond} : <\text{Bool}\ \text{exp}> \times <\text{statement}> \times <\text{statement}> \rightarrow <\text{statement}> \)
\( \text{iter} : <\text{Bool}\ \text{exp}> \times <\text{statement}> \rightarrow <\text{statement}> \)
\( \text{id} : <\text{variable}> \rightarrow <\text{exp}> \)
\( \text{num} : <\text{number}> \rightarrow <\text{exp}> \)
\( \text{add} : <\text{exp}> \times <\text{exp}> \rightarrow <\text{exp}> \)
\( \text{times} : <\text{exp}> \times <\text{exp}> \rightarrow <\text{exp}> \)
\( \text{not} : <\text{Bool}\ \text{exp}> \rightarrow <\text{Bool}\ \text{exp}> \)
\( \text{and} : <\text{Bool}\ \text{exp}> \times <\text{Bool}\ \text{exp}> \rightarrow <\text{Bool}\ \text{exp}> \)
\( \text{or} : <\text{Bool}\ \text{exp}> \times <\text{Bool}\ \text{exp}> \rightarrow <\text{Bool}\ \text{exp}> \)
\( \text{eq} : <\text{exp}> \times <\text{exp}> \rightarrow <\text{Bool}\ \text{exp}> \)

endsig

We can now form the closed term algebra \( T(\Sigma^{WP}) \) from the signature \( \Sigma^{WP} \), giving us an algebra isomorphic to the algebra \( WP(\Sigma) \) of Figure 12.6.

### 12.9 Context Sensitive Languages

We can only use the algorithm in Section 12.8 to describe how we can construct an algebra from a context free grammar. In order to build an algebra to describe a context sensitive language, we have to define additional operations and equations that describe the context sensitive features of the language. The purpose of these new operations is to impose constraints upon the context free part of the language to remove strings that are generated by the context free grammar, but which we do not want in the language.

We used this approach, in an informal manner, to specify the context sensitive features explained in Sections 11.8.1 and 11.8.3. This is a common approach to specifying context sensitive programming languages, see Dijkstra [1962], for example.

The algebraic approach to the specification of non-context free languages is a mathematically precise formulation for defining the context sensitive aspects of languages. Whilst we can completely specify a context sensitive language in a formal way by using a context sensitive grammar, it can be quite a difficult task as it is at such a low level.

**Example** Consider the language \( L = \{a^n b^n c^m | n \geq 1 \} \). We know that this language is not context free from Section 11.7.2. We can be establish that this language is context sensitive by
12.9. CONTEXT SENSITIVE LANGUAGES

giving the following grammar $G_{a^n b^n c^n}$ to define $L = L(G_{a^n b^n c^n})$:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{a^n b^n c^n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b, c$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S, A, B, C$</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow abc$</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow aAbc$</td>
<td></td>
</tr>
<tr>
<td>$Ab \rightarrow bA$</td>
<td></td>
</tr>
<tr>
<td>$Ac \rightarrow Bbcc$</td>
<td></td>
</tr>
<tr>
<td>$bB \rightarrow Bb$</td>
<td></td>
</tr>
<tr>
<td>$aB \rightarrow aaA$</td>
<td></td>
</tr>
<tr>
<td>$aB \rightarrow aa$</td>
<td></td>
</tr>
</tbody>
</table>

Notice that this low-level context-sensitive grammar does not reflect the structure of the language. Compare this with an algebraic description of $L$.

Consider the context free superset

$$L^{a^i b^j c^k} = \{a^i b^j c^k|i, j, k \geq 1\}$$

of the language $L$. We can describe this language easily with the context free grammar:

<table>
<thead>
<tr>
<th>grammar</th>
<th>$G_{a^i b^j c^k}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
<td>$a, b, c$</td>
</tr>
<tr>
<td>nonterminals</td>
<td>$S, A, B, C$</td>
</tr>
<tr>
<td>start symbol</td>
<td>$S$</td>
</tr>
<tr>
<td>productions</td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow ABC$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow a$</td>
<td></td>
</tr>
<tr>
<td>$A \rightarrow aA$</td>
<td></td>
</tr>
<tr>
<td>$B \rightarrow b$</td>
<td></td>
</tr>
<tr>
<td>$B \rightarrow bB$</td>
<td></td>
</tr>
<tr>
<td>$C \rightarrow c$</td>
<td></td>
</tr>
<tr>
<td>$C \rightarrow cC$</td>
<td></td>
</tr>
</tbody>
</table>

Equivalently, we can describe $L^{a^i b^j c^k}$ with the algebra:
\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{algebra} & $L^{a|b|c}$  \\
\hline
\textbf{carriers} & $L^{a|b|c}_a = \{ t \mid t \text{ is the parse tree for } a^i b^j c^k, i, j, k \geq 1 \}$  \\
& $L^{a|b|c}_b = \{ t \mid t \text{ is the parse tree for } a^i, i \geq 1 \}$  \\
& $L^{a|b|c}_c = \{ t \mid t \text{ is the parse tree for } b^i, i \geq 1 \}$  \\
\hline
\textbf{constants} & $a : \rightarrow A$  \\
& $b : \rightarrow B$  \\
& $c : \rightarrow C$  \\
\hline
\textbf{operations} & $ABC : A \times B \times C \rightarrow S$  \\
& $aA : A \rightarrow A$  \\
& $bB : B \rightarrow B$  \\
& $cC : C \rightarrow C$  \\
\hline
\end{tabular}
\end{center}

Now we need to restrict the language $L^{a|b|c}$ to the language $L = L^{a^n b^n c^n}$ that we actually want. We do this by adding functions to our algebraic description. We describe how these functions operate by using equations. These functions ensure that the following constraint is satisfied:

\textbf{Parity Condition}

Every string of $L$ has equal numbers of $a$’s, $b$’s and $c$’s.

First we design a function
\[ \text{chk} : L^{a|b|c} \rightarrow (L^{a^n b^n c^n} \cup \{\text{error}\}) \]
so that:

(i) given the input $t = ABC(aA^n(a), bB^n(b), cC^n(c))$ (for $n \geq 0$) that represents a legal string $a^{i+1} b^{j+1} c^{k+1} \in L$, $\text{chk}(t)$ returns $t$; and

(ii) given the input $t = ABC(aA^i, bB^j(b), cC^k(c))$ where $i$, $j$ and $k$ are not all equal, that represents an illegal string $a^{i+1} b^{j+1} c^{k+1} \notin L$, $\text{chk}(t)$ returns $\text{error}$.

The idea behind the definitions we provide below, is that we recurse down in parallel on each of the terms representing strings of $a$’s, $b$’s and $c$’s, until we either:

(i) end up with more of one element than another, which results in $\text{error}$; or

(ii) end up with the term $ABC(a, b, c)$ which represents the string $abc \in L$, so we return the original term.
To perform this latter task, we use the function

\[
\text{add} : (L^{a^n b^n c^n} \cup \{\text{error}\}) \rightarrow (L^{a^n b^n c^n} \cup \{\text{error}\})
\]

so that \(\text{add}(t)\) extends the term \(t\) representing a string \(a^n b^n c^n\), to that representing \(a^{n+1} b^{n+1} c^{n+1}\). In the case that \(t = \text{error}\), we simply propagate this message along.

<table>
<thead>
<tr>
<th>algebra</th>
<th>(L^{a^n b^n c^n})</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>(L^{a^i b^j c^k})</td>
</tr>
<tr>
<td>carriers</td>
<td>((L^{a^n b^n c^n} \cup {\text{error}}) = {t \mid t \text{ is the parse tree for } a^n b^n c^n, n \geq 1 } \cup {\text{error}}))</td>
</tr>
<tr>
<td>constants</td>
<td>(\text{error} \rightarrow (L^{a^n b^n c^n} \cup {\text{error}}))</td>
</tr>
<tr>
<td>operations</td>
<td>(\text{chk} : L^{a^i b^j c^k} \rightarrow (L^{a^n b^n c^n} \cup {\text{error}}))</td>
</tr>
<tr>
<td></td>
<td>(\text{add} : (L^{a^n b^n c^n} \cup {\text{error}}) \rightarrow (L^{a^n b^n c^n} \cup {\text{error}}))</td>
</tr>
</tbody>
</table>
| equations     | \[
\begin{align*}
\text{chk}(ABC(a, b B(t_B), t_C)) & = \text{error} \\
\text{chk}(ABC(a, t_B, c C(t_C))) & = \text{error} \\
\text{chk}(ABC(a A(t_A), b, t_C)) & = \text{error} \\
\text{chk}(ABC(a A(t_A), t_B, c)) & = \text{error} \\
\text{chk}(ABC(t_A, b, c C(t_C))) & = \text{error} \\
\text{chk}(ABC(t_A, b B(t_B), c)) & = \text{error} \\
\text{chk}(ABC(a, b, c)) & = ABC(a, b, c) \\
\text{chk}(ABC(a A(t_A), b B(t_B), c C(t_C))) & = \text{add}(\text{chk}(ABC(t_A, t_B, t_C))) \\
\text{add(\text{error})} & = \text{error} \\
\text{add}(ABC(t_A, t_B, t_C)) & = ABC(a A(t_A), b B(t_B), c C(t_C))
\end{align*}
\] |

For \(i, j, k \geq 0\), each string \(a^{i+1} b^{j+1} c^{k+1}\) is represented by a term

\[
ABC(a A^i(a), b B^j(b), c C^k(c)),
\]

and each error-reported string can be represented by the term

\[
t = \text{chk}(ABC(a A^i(a), b B^j(b), c C^k(c))),
\]

which we can also represent as an algebraic tree:
The equations tell us that we can reduce every initial algebraic tree to one of the forms shown below:

Every term that does not reduce to `error` represents a parse tree of the form shown below:
12.9. CONTEXT SENSITIVE LANGUAGES

Exercises for Chapter 12

1. Define by structural induction on terms the size function

\[ \text{Size} : T(\Sigma, X) \rightarrow \mathbb{N} \]

for terms.

2. Write the signature \( \Sigma_{\text{Naturals}} \) of Examples 12.1.1 in set format.

3. Write the signature \( \Sigma_{\text{Naturals with Tests}} \) of Examples 12.1.2 in set format.

4. List, with reasons, which of the following are terms over the signature \( \Sigma_{\text{Naturals with Tests}} \) of Examples 12.1.2, and which are not:
   a. \( \text{succ} \)
   b. \( \text{and}(x, \text{not}(\text{false})) \)
   c. \( \text{equals}(0, \text{succ}(0)) \)
   d. \( \text{pred}(\text{succ}(0), 0) \)
   e. \( \text{mult}(\text{succ}(0), \text{pred}(x(\text{mult}(0, \text{succ}(0)))))) \)
   f. \( \text{add}(\text{mult}(x, x), \text{mult}(\text{succ}(\text{succ}(0)), y), x)) \)
   g. \( \text{less\_than}(\text{or}(\text{false}, \text{true}), \text{succ}(\text{add}(x, y))) \)
   h. \( \text{or}(\text{and}(\text{true}, b), \text{less\_than}(x, \text{mult}(x, 0))) \)

5. Give a tree representation for each of the terms of Exercise 4.

6. Using the Principle of Structural Induction on single-sorted terms, prove the Uniqueness Lemma of Section 12.2.3, that functions defined by structural induction are unique. (Compare with the Uniqueness Lemma of Section 7.4.3).

7. Let \( \Sigma \) be a single-sorted signature. Show that the map \( \overline{\nu} : T(\Sigma, X) \rightarrow A \) is a homomorphism.

8. Let \( \Sigma \) be an \( S \)-sorted signature. Show that the \( S \)-indexed family

\[ \overline{\nu} = \langle \overline{\nu}_s : T(\Sigma, X)_s \rightarrow A_s | s \in S \rangle \]

of maps is a homomorphism.

9. Consider the closed term algebra \( T(\Sigma) = T(\Sigma, \emptyset) \). What information do we need to know in this case to define the evaluation map \( \overline{\nu} \) from the term algebra to an algebra \( A \)?

10. Consider the signature \( \Sigma_{\text{Ring}} \):
Let \( X = \{x\} \). What is \( T(\Sigma, X) \)?

Let \( v : X \rightarrow \mathbb{Z} \) be defined by \( v(x) = 0 \). Evaluate \( v(t) \) where:

\[
\begin{align*}
\text{a. } & t = ((x + (x.x)).x) + (1 + 1); \text{ and} \\
\text{b. } & t = ((x.x + 1) + ((x.x).x) + 1).
\end{align*}
\]

11. Show that for any \( t \in T(\Sigma, \{x\}) \) there exists \( t' \in T(\Sigma, \{x\}) \) such that

\[
t' \equiv a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.
\]

where \( x^n \) is the multiplication of \( x \) by itself \( n \) times, and \( a_n \) is the addition of 1 to itself \( a_n \) times.

12. Generalise the term substitution function \( \text{sub}^n \) to the term evaluation function

\[
te^n : T(\Sigma, X) \times X^n \times A^n \rightarrow A
\]

which substitutes the values \( \bar{a} = (a_1, \ldots, a_n) \in A^n \) for the variables \( \bar{x} = (x_1, \ldots, x_n) \) in \( t \in T(\Sigma, X) \) and computes the value

\[
te^n(t, \bar{a}, \bar{x}) = t(\bar{a}/\bar{x}).
\]

(Warning: beware the case where \( t \) contains a variable not in the list \( \bar{x} \).)

13. Given the constant symbol \( \text{zero} \), variables \( \{x, y, z\} \) and operation symbols:

\[
\begin{align*}
succ : & \text{nat} \rightarrow \text{nat} \\
sum : & \text{nat} \times \text{nat} \rightarrow \text{nat} \\
times : & \text{nat} \times \text{nat} \rightarrow \text{nat} \\
\text{muldiv} : & \text{nat} \times \text{nat} \rightarrow \text{nat}
\end{align*}
\]

where \( \text{muldiv}(x, y, z) \) is the function \( x \times y/z \):

(a) give four terms in the term algebra using each operation;

(b) give four terms \( \text{not} \) in the term algebra.

14. Give a valuation map for the above term algebra into an algebra

\[
(N; 0, \text{add}, \text{one}, \text{add}, \text{mul}, \text{md})
\]

define the operations carefully.
12.9. CONTEXT SENSITIVE LANGUAGES

15. Use your valuation map to evaluate (carefully) the value of the following expressions, given that \( v(x) = 5 \), \( v(y) = 10 \) and \( v(z) = 7 \):

   (a) \( \text{succ}(\text{sum}(x, z)) \);
   (b) \( \text{muldiv}(\text{succ}(x), y, \text{add}(\text{Succ}(0), z)) \);
   (c) \( \text{times}(x, \text{add}(y, \text{succ}(\text{succ}(0)))) \);
   (d) \( \text{add}(\text{succ}(0), \text{times}(\text{succ}(0), z)) \).

16. What changes to the algebra \( T(\Sigma_{WP}) \) are needed to incorporate (i) the reals? (ii) characters?

17. By adding appropriate arithmetical operations to \( \Sigma_{WP} \), define a program for Euclid’s algorithm as a term.

18. Give an algebra and a BNF grammar for the \textbf{while} language generated from an arbitrary signature that names the elements and operations for an arbitrary set of data.
Part III
Semantics
Introduction

To appear
Chapter 13

Input-Output Semantics

We have modelled the syntax and semantics of many data types by signatures and algebras. We have modelled the concrete and abstract syntax of many languages, including our chosen imperative programming language, the **while** language, by grammars and algebras. Now we begin the study of the semantics of languages with that of the **while** programming language. Our problem is

**Problem of Semantics**

*To model mathematically what happens when any **while** program performs a computation over any data type.*

To do this, we shed much of the rich concrete syntax of the **while** language and reduce it to a simple abstract syntax. The important syntactic features of complex identifiers and declarations are not needed for semantics. What are needed are simple notations for the data type operations, variables, expressions and commands.

More technically, for any data type whose interface is modelled by a signature $\Sigma$, we will give a concise definition of the set

$$WP(\Sigma)$$

of all **while** programs over $\Sigma$. Then we will make a mathematical model of the behaviour of any **while** program $S \in WP(\Sigma)$ in execution on any implementation of the data type modelled by $\Sigma$-algebra $A$.

In this chapter we will build a mathematical model whose equations allow us to derive the output of a program from its input. This kind of model is called an

*input-output semantics*

for the programming language.

Later, in Chapter 15, we will build more detailed models which allow us to calculate each step in a computation. This kind of model is called an

*operational semantics*

for the programming language.
The **while** programming language $WP(\Sigma)$ defines computations on *any* underlying data type with interface the signature $\Sigma$ and implementation the $\Sigma$-algebra $A$. It constitutes an excellent kernel language with many extensions. Clearly, syntactic features that do not require semantics can be added to enrich the language. By choosing different signatures and algebras, a range of programs and, indeed, programming languages can be specified easily.

In general, we can extend to a new language $\mathcal{L}$ by adding new constructs to the **while** programming language by three methods. First, we can extend the data types. For example, to add arrays or streams to the **while** language, we need only add them to a data type signature $\Sigma$ to make expansions $\Sigma_{Array}$ and $\Sigma_{Stream}$ and then take

$$WP(\Sigma_{Array}) \quad \text{and} \quad WP(\Sigma_{Stream})$$

Now we can substitute signatures and apply our methods to give a semantics for a **while** language with arrays and a **while** language with streams, respectively.

Second, we can extend the language with constructs by using syntactic definitions that specify the new constructs in terms of the old. For example, we may add **repeat** or **case** statements by defining them in terms of the constructs of the **while** language. Here a process of flattening reduces the extended language $\mathcal{L}$ to the kernel language $WP(\Sigma)$ and our semantic methods can be applied immediately. An obvious question arises:

*Which imperative constructs can be reduced to those of the kernel **while** language?*

For example, can procedures and recursion be added in this way?

Third, we can extend the language with new constructs that have their own semantics. This is easy to do for constructs such as **repeat** and **case**, but harder for constructions such as recursion. Extensions with concurrent assignments and non-deterministic choices we do not expect to be able to reduce to a sequential deterministic kernel like the **while** language.

These methods are depicted in Figure 13.1.

---

**Figure 13.1**: Extending the kernel **while** language.
13.1. SOME SIMPLE SEMANTIC PROBLEMS

Let us look further at the raw material of programming, in order to think about the problems of defining the semantics a programming language; these are the problems that we can expect to analyse by our mathematical models.

13.1 Some Simple Semantic Problems

To think scientifically and reason mathematically about imperative programming, we must master the problem of defining exactly the semantics of an imperative programming language. To appreciate the problem, we consider how to define precisely the data used in a program and the effect or meaning of a simple programming statement such as the assignment. Then we will consider how to define the effect of a sequence of statements, such as a program for calculating the greatest common divisor of two numbers.

13.1.1 Semantics of a Simple Data Type

Consider a simple interface for programming with natural numbers. It is a list of names for data and operations:

<table>
<thead>
<tr>
<th>signature</th>
<th>Naturals for Euclidean Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>nat, bool</td>
</tr>
<tr>
<td>constants</td>
<td>0 : nat → nat</td>
</tr>
<tr>
<td></td>
<td>true, false : → bool</td>
</tr>
<tr>
<td>operations</td>
<td>mod : nat × nat → nat</td>
</tr>
<tr>
<td></td>
<td>≠ : nat × nat → bool</td>
</tr>
</tbody>
</table>

Statements in the program for the Euclidean Algorithm will use these notations (see Section 13.1.3). But this is notation; what does it denote, what are their semantics?

There are several choices for the set of natural numbers, such as

\[
N_{\text{Dec}} = \{0, 1, 2, \ldots\} \\
N_{\text{Bin}} = \{0, 1, 10, \ldots\}
\]

represented in decimal and binary, respectively. And there are finite choices of the form

\[
\{0, 1, \ldots, Max\}.
\]

Each choice has appropriate functions to interpret the operator mod and test eq. The important point is that some collection of sets of data and functions on data must be made in order to begin interpreting the statements.

13.1.2 Semantics of a Simple Construct

We will define the semantics of a series of program fragments based on different assignment statements.
Assignment 1: \begin{varialign*} & \textbf{begin} & \textbf{var} & x, y: \textbf{nat}; & x := y & \textbf{end} \\
\end{varialign*}

"Make the value of \( y \) the new value of \( x \). The value of \( y \) is not changed but the old value of \( x \) is lost."

This first fragment can be applied to any data type, not just the naturals. This second fragment depends on the properties of the data:

Assignment 2: \begin{varialign*} & \textbf{begin} & \textbf{var} & x, y, z: \textbf{nat}; & x := y + z & \textbf{end} \\
\end{varialign*}

"Evaluate the sum of the data that are the values of \( y \) and \( z \), and make this the new value of \( x \). The values of \( y \) and \( z \) are not changed but the old value of \( x \) is lost."

The operation of addition always returns a value: it is a total function. If the data type of naturals is finite then a maximum natural number will exist but the above description will still be valid. Clearly to define the assignment we must also write out the meaning of the operators it contains. Now the semantics of the second fragment can be also applied to numbers such as the integers, rational numbers, real numbers or complex numbers.

Assignment 3: \begin{varialign*} & \textbf{begin} & \textbf{var} & x, y, z: \textbf{nat}; & x := (y + z)/2 & \textbf{end} \\
\end{varialign*}

"Evaluate the sum of the data that are the values of \( y \) and \( z \). Divide by 2 and, if this number is a natural number, then make this the new value of \( x \). The values of \( y \) and \( z \) are not changed but the old value of \( x \) is lost. If however the division leads to a rational then \ldots"

Because division by 2 in the naturals does not always have a value (e.g., \( 3/2 \) is not a natural) the assignment contains an operator that is a partial function. The above meaning is left incomplete because there are several choices available. First, one can ignore the problem:

(i) \((y + z)/2\) can be rounded up or rounded down and the computation proceed;

(ii) \(x, y, z\) do not change their values and the computation proceeds;

or one can indicate there is a problem:

(iii) an undefined element \( u \) can be added to the data type, to indicate there is no value, which can be assigned to \( x \) and the computation proceed;

(iv) an error message can be displayed and the computation suspended.
13.1. SOME SIMPLE SEMANTIC PROBLEMS

This particular problem with division can be avoided by changing the data type to the rationals. However the basic semantical difficulties with partial operators remain. Another example, involving the real numbers, is this:

\[
\text{Assignment 4: } \begin{align*}
\text{begin } & \text{var } x, y, z : \text{real}; \\
& x := \sqrt{(y + z)/2} \text{ end}
\end{align*}
\]

"Evaluate the sum of the data that are the values of \(y\) and \(z\), and divide by 2. Take the square root of this datum, if it exists, and make this the new value of \(x\). The values of \(y\) and \(z\) are not changed but the old value of \(x\) is lost. If the square root does not exist then . . ."

Since there are no square roots for negative numbers we must complete the semantics by choosing an option. Again the particular problem can be avoided by changing the type to the complex numbers.

As one thinks about these meanings one is struck by the increasing length of the descriptions and the choice of statements as to what happens when, in particular, operators return no value. In fact there is already a decision in the first case of Assignment 1: here is a different non-standard meaning for the assignment:

\[
\text{Assignment 5: } \begin{align*}
\text{begin } & \text{var } x, y : \text{nat}; \\
& x := y \text{ end}
\end{align*}
\]

"Transfer the value of \(y\) to be the new value of \(x\). The value of \(y\) is set to a special value denoting empty or unspecified \(u\), and the old value of \(x\) is lost."

If this non-standard interpretation is selected then the other semantic options can be rewritten, the choices multiply and the descriptions become more elaborate.

Try this exercise of carefully describing semantics with some of the other constructs we have mentioned in the previous section.

Of course, what we need is a mathematical formulation of the semantics that is precise, concise and capable of a full logical analysis. Shortly, we will define the semantics of assignments in a simple formula:

\[
M(\text{begin var } x, y, z, \ldots : \text{data}; x := e(y, z, \ldots) \text{ end})(\sigma) = \sigma[x/V(e)(\sigma)]
\]

Roughly speaking, the left hand side means: this is the state after applying the assignment fragment. The right hand side means: take the state \(\sigma\) and replace the value of \(x\) by the value obtained by evaluating the expression \(e(y, z, \ldots)\) on \(\sigma\), and leaving all other variables \(y, z, \ldots\) unchanged.

This formula works for all of the above cases where the semantic decisions concern only the data types. Remarkably, the formula works for any data type consisting of data equipped with operators. The option (4) requires us to add the idea of an error state to the set of possible states.
13.1.3 Semantics of a Simple Program

Let us look at a program that is based on the imperative constructs mentioned.

We consider a famous and ancient algorithm. The largest natural number that divides two given natural numbers $m$ and $n$ is called the greatest common divisor of $m$ and $n$, and is written $\gcd(m, n)$; for example, $\gcd(45, 12) = 3$.

Euclid’s Algorithm is the name given to a method for computing $\gcd(m, n)$ found in Book VII of Euclid’s Elements, although the algorithm is older. Here it is expressed as a simple imperative program based on the data type of natural numbers:

```
program Euclid(input : x, y; output : y);
signature Naturals for Euclidean Algorithm
sorts nat, bool
constants 0 : → nat
        true, false : → bool
operations mod : nat × nat → nat
              ≠ : nat × nat → bool
endsig
body
var x, y, z : nat;
begin
  z := x mod y;
  while z ≠ 0 do
    x := y;
    y := z;
    z := x mod y
  od
end
```

How can we describe exactly the behaviour of the program? What is a computation by this program?

A state of a computation can be defined to be the three integer values $p$, $q$, $r$ of the variables $x$, $y$, $z$. Let

$$\mathbb{N} = \mathbb{N}_{\text{Dec}} = \{0, 1, 2, \ldots\}$$

denote the natural numbers. Hence a state is represented by a 3-tuple

$$(p, q, r) \in \mathbb{N}^3$$
13.2. OVERVIEW

A computation can be defined to be a sequence of these states starting with some initial state

\[(p_1, q_1, r_1), (p_2, q_2, r_2), \ldots, (p_k, q_k, r_k), \ldots\]

Here \((p_k, q_k, r_k)\) is the state of the computation at the \(k\)-th step. The step from \((p_k, q_k, r_k)\) to \((p_{k+1}, q_{k+1}, r_{k+1})\) is determined by processing a statement in the program. This sequence is a program trace which tracks the values of the variables as the computation proceeds.

Let us work out an example using our informal understanding of these programming constructs. The computation of \(gcd(45, 12)\) by Euclid’s algorithm consists of the following steps:

<table>
<thead>
<tr>
<th>Step Number</th>
<th>Value of (x)</th>
<th>Value of (y)</th>
<th>Value of (z)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>12</td>
<td>?</td>
<td>Initial state</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>12</td>
<td>9</td>
<td>First assignment</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>Entered loop</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>Re-enter loop</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>Exit loop</td>
</tr>
</tbody>
</table>

The ? indicates that the value of \(z\) can be anything at the start of the computation. Notice that the idea of state chosen avoids, or hides, the meaning of the read and write statements, and the evaluation of the Boolean tests.

Of particular importance are exceptional properties of the program:

What is \(gcd(0, n)\) and \(gcd(0, 0)\)? What does the program do for these values?

If the semantics of the integers is not the infinite set

\[N = \{0, 1, 2, \ldots\}\]

but some finite set

\[\{-M, \ldots, -2, -1, 0, 1, 2, \ldots, +M\}\]

that approximates \(N\), then does the program still compute greatest common divisors?

Clearly, the answers to these latter questions depend on the data type of integers and the meanings of greatest common divisor, and the operation of division. They also involve properties of the Booleans.

There are some other obvious questions we can ask: How do we know the program computes the greatest common divisor? What is its efficiency?

13.2 Overview

A while program produces a computation by specifying a sequence of assignments and tests on the values of its variables. At any stage, the values of the variables constitute a state of a computation. For each while program \(S\) we imagine that, given any starting or initial state \(\sigma\) for a computation, the program generates a finite sequence

\[\sigma_0, \sigma_1, \ldots, \sigma_n\]
or an infinite sequence
\[ \sigma_0, \sigma_1, \ldots, \sigma_k, \ldots \]
of states as the constructs in the program \( S \) are processed in the order prescribed; of course, the first state \( \sigma_0 \) in the sequence is the given initial state \( \sigma \).

These sequences are the basis of a trace of a program execution. The finite sequence is called a terminating or convergent computation sequence; the infinite sequence is called a non-terminating or divergent computation sequence. A terminating computation sequence has a last or final state \( \sigma_n \) from which we expect that an output can be abstracted. A non-terminating computation sequence has no final state and, since the computation proceeds forever, we do not expect an output.

In our first attempt at defining the semantics of a program we will focus on this concept of the input-output behaviour of a program.

Suppose the program \( S \) is over a data-type with signature \( \Sigma \), i.e., \( S \in WP(\Sigma) \). Suppose the data type is implemented by a \( \Sigma \)-algebra \( A \). We will define the concept of a state of a computation using data from the algebra \( A \) and hence the set
\[ \text{State}(A) \]
of all possible states of all possible computations using data from \( A \). The input-output behaviour of a program \( S \) is specified by a function
\[ M^w_A(S) : \text{State}(A) \rightarrow \text{State}(A) \]
such that for any state \( \sigma \in \text{State}(A) \)
\[ M^w_A(S)(\sigma) = \text{the final state of the computation generated by a program } S \]
\[ \text{from an initial state } \sigma, \text{ if such a final state exists.} \]

Since a while loop may execute forever, we do not expect the function \( M^w_A(S) \) to be defined on all states. That is, we expect \( M^w_A(S) \) to be a partial function. If there is a final state \( \tau \) of the computation of \( S \) on \( \sigma \) we say the computation is terminating and we write
\[ M^w_A(S)(\sigma) \downarrow \quad \text{or} \quad M^w_A(S)(\sigma) \downarrow \tau \]
otherwise it is non-terminating and we write
\[ M^w_A(S)(\sigma) \uparrow . \]

To model the behaviour of programs, we will solve the following problem:

**Problem of Input-Output Semantics** To give a precise mathematical definition of the input-output function \( M^w_A(S) \).

In this first attempt at modelling program behaviour, it is important to understand that we are seeking to define the input-output behaviour of a program and not every step of a computation by the program. It turns out that to define \( M^w_A(S) \) we do not need a full analysis of every step of a computation of \( S \). The analysis is directed toward the idea that the program computes a partial function \( f \) on \( A \). Hence, input-output semantics is an abstraction from the behaviour of
13.3. DATA

programs. There are many methods for defining the semantics of a program \( S \) that are based on intuitions of how the program operates, step-by-step; these methods are called \textit{operational semantics}.

The mathematical modelling of the operation of a program is a subtle activity. Even in the case of simple \texttt{while} programs, there is considerable scope for the analysis of the step

\[ \ldots, \sigma_i, \sigma_{i+1}, \ldots \]

from one state \( \sigma_i \) to the next \( \sigma_{i+1} \) in a computation. The level of detail involved, or considered, in a step determines the level of abstraction of the semantics and affects our analysis of time in computations and hence of the performance of the programs.

In this chapter we describe a simple method of defining the input-output semantics of \texttt{while} programs, guided by some operational ideas. We will specify the

- data
- states of a computation
- operations and tests on states
- control and sequencing of actions in commands

that allow \( S \) to generate computations. In a later chapter these methods will be reviewed and a more detailed operational semantics studied; this will generate a full trace of a program’s operation.

### 13.3 Data

The interfaces and implementations of data types are modelled by signatures and algebras, respectively. We met these elements of the theory of data in Chapter 3, and showed how to integrate the signatures into the syntax of the \texttt{while} language in Chapter 9. Our first task is to integrate the algebras into the semantics of the \texttt{while} language. We will do this for the language \( WP(\Sigma) \) of \texttt{while} programs over a fixed signature \( \Sigma \).

We will quickly recall and settle notations and examples of algebras which we will use to illustrate our semantical definitions and computations. We suppose a signature \( \Sigma \) has the form:


<table>
<thead>
<tr>
<th>signature Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts ..., s, ...</td>
</tr>
<tr>
<td>(\text{Bool})</td>
</tr>
<tr>
<td>constants ..., c : (\rightarrow s, ...)</td>
</tr>
<tr>
<td>(\text{true, false} : \rightarrow \text{Bool})</td>
</tr>
<tr>
<td>operations ..., f : (s(1) \times \cdots \times s(n) \rightarrow s, ...)</td>
</tr>
<tr>
<td>..., r : (t(1) \times \cdots \times t(m) \rightarrow \text{bool}, ...)</td>
</tr>
<tr>
<td>(\text{not} : \rightarrow \text{Bool})</td>
</tr>
<tr>
<td>(\text{and} : \rightarrow \text{Bool})</td>
</tr>
<tr>
<td>(\text{or} : \rightarrow \text{Bool})</td>
</tr>
<tr>
<td><strong>endsig</strong></td>
</tr>
</tbody>
</table>

and the \(\Sigma\)-algebra \(A\) has the form:

<table>
<thead>
<tr>
<th>algebra (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriers ..., (A, ...)</td>
</tr>
<tr>
<td>(B)</td>
</tr>
<tr>
<td>constants ..., (c^A : \rightarrow A, ...)</td>
</tr>
<tr>
<td>(\text{true}^A, \text{false}^A : \rightarrow B)</td>
</tr>
<tr>
<td>operations ..., (f^A : A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow A, ...)</td>
</tr>
<tr>
<td>..., (r^A : A_{t(1)} \times \cdots \times A_{t(m)} \rightarrow B, ...)</td>
</tr>
<tr>
<td>(\text{not}^A : \rightarrow B)</td>
</tr>
<tr>
<td>(\text{and}^A : B \times B \rightarrow B)</td>
</tr>
<tr>
<td>(\text{or}^A : B \times B \rightarrow B)</td>
</tr>
</tbody>
</table>

We assume that the signature \(\Sigma\) and algebra \(A\) are both standard with respect to the Booleans. That is, we assume that the Boolean sorts and operations have their standard interpretation in the algebra \(A\).

### 13.3.1 Algebras of Naturals

Consider a two-sorted signature \(\Sigma_{\text{can}}\) of natural numbers and Booleans defined by:
Let 
\[ A = (N, B; 0, \text{tt}, \text{ff}; n + 1, n + m, n \times m, \land, -, <) \]
be an algebra with signature \( \Sigma_{\text{Peano}} \).

### 13.3.2 Algebra of Reals

Consider a two-sorted signature \( \Sigma_{\text{or-reals}} \) of real numbers and Booleans defined by:

Let 
\[ A = (R, B; 0, \text{tt}, \text{ff}; +, -, \times, \land, -, <) \]
be an algebra with signature \( \Sigma_{\text{or}} \). If we add division we have a new signature \( \Sigma_{\text{of-reals}} \) defined by
**signature**  Ordered field of reals  
**sorts**  \( real, Bool \)

**constants**  
- zero, one : \( \rightarrow real \)  
- true, false : \( \rightarrow Bool \)

**operations**  
- add : \( real \times real \rightarrow real \)  
- minus : \( real \times real \rightarrow real \)  
- mult : \( real \times real \rightarrow real \)  
- divide : \( real \times real \rightarrow real \)  
- and : \( Bool \times Bool \rightarrow Bool \)  
- not : \( Bool \rightarrow Bool \)  
- less than : \( real \times real \rightarrow Bool \)

Let  
\[
A = (\mathbb{R}, \mathbb{B}; 0, 1, \mathit{tt}, \mathit{ff}; +, -, \times, ^{-1}, \land, \lor, <)
\]

be an algebra with signature \( \Sigma_{of-reals} \).

### 13.4 States

To define the semantics of \( WP(\Sigma) \), we must first define the state of a computation over algebra \( A \) and then how to apply expressions, Boolean expressions and commands to these states. The data of \( A \) belong to the family  
\[
\langle A_s \mid s \in S \rangle
\]

of carrier sets of \( A \). Each sort of data needs its own store.

We will consider **while** programs that operate over some \( S \)-sorted family  
\[
Var = \langle Var_s \mid s \in S \rangle
\]

of variables, where  
\[
Var_s
\]

is the set of all variables of sort \( s \).

For each sort \( s \in S \) an **\( s \)-state over \( A \)** is a map:  
\[
\sigma_s : Var_s \rightarrow A_s
\]

which represents a possible configuration of a store of data of sort \( s \) from \( A \). The idea is that  
\[
\sigma_s(x) = \text{value in } A_s \text{ of variable } x \in Var_s.
\]

Let \( \text{State}_s(A) \) be the set of all \( s \)-sorted states over \( A \).  
A **state over \( A \)** is a family  
\[
\sigma = \langle \sigma_s \mid s \in S \rangle
\]
of \( s \)-states over \( A \) and represents a possible configuration of a complete store of data from \( A \).

The set of all states over \( A \) represents all configurations of this abstract state over \( A \), and is given by

\[
State(A) = \langle State_s(A) \mid s \in S \rangle.
\]

\[
\begin{array}{|c|c|}
\hline
\text{sort } s & \text{variables } x_0^s, x_1^s, \ldots, x_n^s, \ldots \\
\hline
\text{state } \sigma_s & \text{values } \sigma_s(x_0^s), \sigma_s(x_1^s), \ldots, \sigma_s(x_n^s), \ldots \\
\hline
\end{array}
\]

\[
\vdots
\]

Figure 13.2: The store modelled by state \( \sigma \).

**Notational Convention**

In practice, we can often drop reference to sorts and allow them to be inferred from the context. We will sometimes write

\[
x \in \text{Var}, \sigma \in \text{State}(A) \text{ and } \sigma(x) \in A
\]

instead of

\[
x \in \text{Var}_s, \sigma \in \text{State}_s(A) \text{ and } \sigma(x) \in A_s.
\]

### 13.4.1 Example

Let \( \text{Var}_{\text{real}} = \{r_1, r_2, \ldots \} \) and \( \text{Var}_{\text{Bool}} = \{b_1, b_2, \ldots \} \) be a set of real- and Bool-sorted variables respectively. Let \( \sigma = \langle \sigma_{\text{real}}, \sigma_{\text{Bool}} \rangle \) be a state over \( A \), where

\[
\sigma_{\text{real}} : \quad \text{Var}_{\text{real}} \rightarrow \mathbb{R}
\]

\[
\sigma_{\text{Bool}} : \quad \text{Var}_{\text{Bool}} \rightarrow \mathbb{B}
\]

are real-states and Bool-states, respectively. A state can be visualised as shown in Table 13.1 below; for example \( \sigma_{\text{real}}(r_3) = \sqrt{2} \):

<table>
<thead>
<tr>
<th>\text{Var}_{\text{real}}</th>
<th>r_1</th>
<th>r_2</th>
<th>r_3</th>
<th>r_4</th>
<th>r_5</th>
<th>r_6</th>
<th>\ldots</th>
<th>r_4</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\sigma_{\text{real}}(r_i)</td>
<td>0</td>
<td>1.5</td>
<td>12</td>
<td>\pi</td>
<td>\sqrt{2}</td>
<td>1</td>
<td>\ldots</td>
<td>-4</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{Var}_{\text{Bool}}</th>
<th>b_1</th>
<th>b_2</th>
<th>b_3</th>
<th>b_4</th>
<th>b_5</th>
<th>b_6</th>
<th>\ldots</th>
<th>b_4</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\sigma_{\text{Bool}}(b_i)</td>
<td>tt</td>
<td>ff</td>
<td>tt</td>
<td>ff</td>
<td>ff</td>
<td>ff</td>
<td>\ldots</td>
<td>ff</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

Table 13.1: An example of real-sorted and Bool-sorted states.
13.4.2 Substitutions in States

Let \( \sigma = \langle \sigma_s | s \in S \rangle \) be a state over \( A \). Let \( s \in S \), \( x \in \text{Var}_s \) and \( a \in A_s \). To change the value of a variable \( x \) in the state \( \sigma \) to the new value \( a \) we require a substitution operation

\[
sub_s : A_s \times \text{Var}_s \times \text{State}_s \rightarrow \text{State}_s
\]

that transforms \( \sigma_s \) into a new state \( sub_s(x, a, \sigma_s) \); we invariably write the value of the operation

\[ \sigma_s[a/x] \]

rather than \( sub_s(x, a, \sigma_s) \). This substitution is defined by:

\[
sub_s(x, a, \sigma_s)(y) = \sigma_s[a/x](y) = \begin{cases} 
\sigma_s(y) & \text{if } y \neq x; \\
a & \text{otherwise.}
\end{cases}
\]

So \( \sigma_s \) is unchanged except that the new value of \( x \) is \( a \) and the old value is lost; in particular, for \( y \neq x \),

\[ \sigma_s[a/x](y) = \sigma_s(y) \]

**Example** Consider the state \( \sigma \) given in Table 13.1. Then the substitution

\[ \sigma_{\text{real}}[3.142/r_i] \]

will give the new \( \sigma_{\text{real}} \) state:

<table>
<thead>
<tr>
<th>Var_{\text{real}}</th>
<th>r_1</th>
<th>r_2</th>
<th>r_3</th>
<th>r_4</th>
<th>r_5</th>
<th>r_6</th>
<th>\ldots</th>
<th>r_i</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_{\text{real}}(r_i) )</td>
<td>0</td>
<td>1.5</td>
<td>12</td>
<td>3.142</td>
<td>( \sqrt{2} )</td>
<td>1</td>
<td>\ldots</td>
<td>-4</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

Table 13.2: New state.

13.5 Operations and tests on states

We define the value of general expressions and Boolean expressions on a state.

13.5.1 Expressions

We define the *value* of an expression \( e \) on a state \( \sigma \) over \( A \) by means of the function

\[ V_A : \text{Exp}(\Sigma) \rightarrow (\text{State}(A) \rightarrow A) \]

where for \( e \in \text{Exp}(\Sigma) \) the purpose of the function

\[ V_A(e) : \text{State}(A) \rightarrow A \]

is, for \( \sigma \in \text{State}(A) \)

\[ V_A(e)(\sigma) = \text{the value in } A \text{ of the expression } e \text{ on the state } \sigma. \]
In detail, $V_A$ is a family

$$\langle V_A^s \mid s \in S \rangle$$

deﬁned by induction on the structure of terms over $\Sigma$ simultaneously for each sort $s \in S$ by:

$$V_A^s(c)(\sigma) = c^A$$
$$V_A^s(x)(\sigma) = \sigma(x) = \sigma_s(x)$$
$$V_A^s(f(e_1, \ldots, e_n))(\sigma) = f^A(V_A^s[1](e_1)(\sigma), \ldots, V_A^s[n](e_n)(\sigma))$$

for $c \in \Sigma_k, s$ for $x \in \text{Var}_s$ for $f \in \Sigma_{w,s}, e_i \in \text{Exp}_{s[i]}(\Sigma)$.

### 13.5.2 Example

Recall the two-sorted signature $\Sigma_{of}$ and algebra $A$ of real numbers in Section 13.3.2. Let $t_1 = \text{add}(x, y)$ and $t_2 = \text{mult}(\text{plus}(x, \text{one}), \text{minus}(x, \text{one}))$ be terms over $\Sigma_{of}$ of sort real; so $x, y \in \text{Var}_{\text{real}}$. If the variables have the values

$$\sigma_{\text{real}}(x) = 1$$
$$\sigma_{\text{real}}(y) = 3.142$$

in a state $\sigma = (\sigma_{\text{real}}, \sigma_{\text{bool}})$, then we get the following evaluations of the terms $t_1$ and $t_2$ on the state $\sigma$:

$$V_A^\text{real}(t_1)(\sigma) = V_A^\text{real}(\text{add}(x, y))(\sigma)$$
$$= +(V_A^\text{real}(x)(\sigma), V_A^\text{real}(y)(\sigma))$$
$$= +(\sigma_{\text{real}}(x), \sigma_{\text{real}}(y))$$
$$= +(1, 3.142)$$
$$= 4.142$$

$$V_A^\text{real}(t_2)(\sigma) = V_A^\text{real}(\text{mult}(\text{plus}(x, \text{one}), \text{minus}(x, \text{one}))))(\sigma)$$
$$= \times(V_A^\text{real}(\text{plus}(x, \text{one}))(\sigma), V_A^\text{real}(\text{minus}(x, \text{one}))(\sigma))$$
$$= \times(+V_A^\text{real}(x)(\sigma), V_A^\text{real}(\text{one})(\sigma)), -(V_A^\text{real}(x)(\sigma), V_A^\text{real}(\text{one})(\sigma)))$$
$$= \times(+\sigma_{\text{real}}(x), 1), -(\sigma_{\text{real}}(x), 1))$$
$$= \times(+1, 1), -(1, 1))$$
$$= \times(2, 0)$$
$$= 0$$

### 13.5.3 Tests

The semantics of Boolean expressions is the special case $s = \text{Bool}$ of the semantics of expressions: we use $V_A^{\text{bool}}$. It is helpful to give tests special treatment and notations: We define the
value of a Boolean expression $b$ on a state $\sigma$ over $A$ by means of the function

$$W_A : BExp(\Sigma) \rightarrow (State(A) \rightarrow \mathbb{B})$$

where for $b \in BExp(\Sigma)$ the purpose of the function

$$W_A(b) : State(A) \rightarrow \mathbb{B}$$

is, for $\sigma \in State(A)$

$$W_A(b)(\sigma) = \text{the value in } \mathbb{B} \text{ of the Boolean expression } b \text{ on the state } \sigma.$$ 

We write out the inductive definition of the above function on the syntactic structure of $b$:

$$W_A(\text{true})(\sigma) = tt$$
$$W_A(\text{false})(\sigma) = ff$$
$$W_A(r(e_1, \ldots, e_n))(\sigma) = r^A(V_A(e_1)(\sigma), \ldots, V_A(e_n)(\sigma))$$
$$W_A(\text{not}(b))(\sigma) = \begin{cases} tt & \text{if } W_A(b)(\sigma) = ff; \\ ff & \text{if } W_A(b)(\sigma) = tt. \end{cases}$$
$$W_A(\text{and}(b_1, b_2))(\sigma) = \begin{cases} tt & \text{if } W_A(b_1)(\sigma) = tt \text{ and } W_A(b_2)(\sigma) = tt; \\ ff & \text{otherwise.} \end{cases}$$

The function $W_A$ is one of the component functions of $V_A$ since $BExp(\Sigma) = Exp_{\text{Bool}}(\Sigma)$, and thus:

$$V_A^{\text{Bool}} : Exp_{\text{Bool}}(\Sigma) \rightarrow (State_{\text{Bool}}(A) \rightarrow \mathbb{B}).$$

Notice that since the semantic functions $V_A^s$ are defined simultaneously for all $s \in S$, $W_A$ (which is, of course, $V_A^{\text{Bool}}$) cannot be defined independently of the term evaluation for the other sorts.

**Example** We can now extend Example 13.5.2 to deal with Boolean expressions. Let

$$\sigma_{\text{Bool}}(b)(\sigma) = tt$$

be the evaluation of the Boolean expression $b$ on the state $\sigma$. The evaluation of the term

$$\text{and}(\text{less_than}(x, \text{zero}), b)$$
on the state $\sigma$ proceeds as follows:

$$W_A(\text{and}(\text{less_than}(x, \text{zero}), b))(\sigma) = \begin{aligned} & \wedge(W_A(\text{less_than}(x, \text{zero}))(\sigma), W_A(b)(\sigma)) \\ &= \wedge(< V_A(x)(\sigma), V_A(\text{zero})(\sigma)>, \sigma_{\text{Bool}}(b)(\sigma)) \\ &= \wedge(< \sigma_{\text{real}}(x), 0>, tt) \\ &= \wedge(1, tt) \\ &= \wedge(ff, tt) \\ &= ff \end{aligned}$$
13.6 Statements and Commands: First Definition

The input-output semantics for commands is given by the following functions:

\[ M^\omega_A : \text{Comm}(\Sigma) \rightarrow (\text{State}(A) \rightarrow \text{State}(A)) \]

where, for \( S \in \text{Comm}(\Sigma) \) the purpose of the function

\[ M^\omega_A(S) : \text{State}(A) \rightarrow \text{State}(A) \]

is, for \( \sigma \in \text{State}(A) \)

\[ M^\omega_A(S)(\sigma) = \text{the final state, if such a state exists, on executing a program } S \text{ on an initial state } \sigma. \]

13.6.1 First Definition

The first definition of the input-output semantics \( M^\omega_A \) of \textbf{while} programs is a formalisation of simple ideas about the constructs. The definition is constructed by induction on the syntactic structure of a program \( S \):

**Base Case** There are two base cases:

- **Identity**
  
  The skip statement does nothing: for \( \sigma \in \text{State}(A) \),
  
  \[ M^\omega_A(\text{skip})(\sigma) = \sigma. \]

- **Assignment**
  
  The assignment statement evaluates an expression and updates a variable: for \( \sigma \in \text{State}(A) \),
  
  \[ M^\omega_A(x := e)(\sigma) = \sigma[V_A(e)(\sigma)/x]. \]

**Induction Step** We suppose that the partial functions \( M^\omega_A(S_0), M^\omega_A(S_1) \) and \( M^\omega_A(S_2) \) are specified on all states. There are three cases:

- **Composition**
  
  The composition operation executes \( S_1 \) and then \( S_2 \):
  
  \[ M^\omega_A(S_1; S_2)(\sigma) \simeq M^\omega_A(S_2)(M^\omega_A(S_1)(\sigma)) \]
  
  More exactly, the equation means: Execute \( S_1 \) on \( \sigma \) and if there is a final state \( \sigma' = M^\omega_A(S_1)(\sigma) \) then execute \( S_2 \) on this state \( \sigma' \). If there is a final state \( \sigma'' = M^\omega_A(S_2)(\sigma') \) then the final state of \( S_1; S_2 \) on \( \sigma \) is \( \sigma'' = M^\omega_A(S_1; S_2)(\sigma) \). Otherwise, there is no final state for \( S_1; S_2 \) on \( \sigma \) and \( M^\omega_A(S_1; S_2)(\sigma) \uparrow. \)
CHAPTER 13. INPUT-OUTPUT SEMANTICS

Conditional

The conditional operation chooses to execute $S_1$ or $S_2$ according to the test $b$:

$$M_A^{io}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma) = \begin{cases} M_A^{io}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\ M_A^{io}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = ff. \end{cases}$$

More exactly, this equation means: If $b$ is true on $\sigma$ then execute $S_1$ on $\sigma$ and if there is a final state $\sigma' = M_A^{io}(S_1)(\sigma)$ then the final state of $\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$ is $\sigma'$. If $b$ is false on $\sigma$ then execute $S_2$ on $\sigma$ and if there is a final state $\sigma'' = M_A^{io}(S_2)(\sigma)$ then the final state of $\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$ is $\sigma''$. Otherwise there is no final state for the conditional.

Iteration

The iteration operator repeats $S_0$ until $b$ is false. We define the semantics of the while command in two cases, depending upon whether or not a computation exits the while loop.

Termination

Suppose the computation exits the while construct and halts. Then the situation is characterised by:

$$M_A^{io}(\text{while } b \text{ do } S_0 \text{ od})(\sigma) \downarrow \text{ and } M_A^{io}(\text{while } b \text{ do } S_0 \text{ od})(\sigma) = \tau$$

if, and only if, there exists $n \geq 0$ and a sequence of states

$$\sigma_0, \sigma_1, \ldots, \sigma_n$$

such that

<table>
<thead>
<tr>
<th>Initial state</th>
<th>$\sigma_0 = \sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final state</td>
<td>$\sigma_n = \tau$</td>
</tr>
<tr>
<td>Iteration</td>
<td>$M_A^{io}(S_0)(\sigma_{i-1}) \downarrow$ and $M_A^{io}(S_0)(\sigma_{i-1}) = \sigma_i$ for $1 \leq i \leq n$</td>
</tr>
<tr>
<td>Continuation condition</td>
<td>$W_A(b)(\sigma_i) = tt$ for $1 \leq i \leq n - 1$</td>
</tr>
<tr>
<td>Exit condition</td>
<td>$W_A(b)(\sigma_n) = ff$</td>
</tr>
</tbody>
</table>

This sequence of states traces out precisely the stages of the computation when the Boolean test $b$ is evaluated and the decision is taken either to exit the while loop or execute the body $S_0$. The loop is traversed $n$ times.
Non-termination
Otherwise, suppose that the computation does not exit the while construct. This means there is no such finite sequence and $M^i_A(\text{while } b \text{ do } S_0 \text{ od})(\sigma)$ is undefined, which we denote by

$$M^i_A(\text{while } b \text{ do } S_0 \text{ od})(\sigma) \uparrow.$$ 

13.6.2 Examples
Let us consider the input-output semantics of some trivial programs over the real numbers. Let $\Sigma_{or-real}$ be the signature of the ordered ring of reals, and let $A$ be the standard $\Sigma_{or-real}$-algebra.

1. Consider the program

$$y := x$$

in $Comm(\Sigma_{or})$. We shall evaluate this program on the state $\sigma \in State(A)$, where

$$\sigma_{\text{real}}(x) = 3.142$$
$$\sigma_{\text{real}}(y) = \pi.$$ 

Thus,

$$M^r_A(y := x)(\sigma) = \sigma[V_A(x)(\sigma)/y]$$
$$= \sigma[\sigma_{\text{real}}(x)/y]$$
$$= \sigma[3.142/y].$$

So the program $y := x$ on the state $\sigma$ has the effect of replacing the value of $y$ (initially $\pi$ in state $\sigma$) with that of the value of $x$ on state $\sigma$, which is 3.142, whilst leaving the values of all the other variables unchanged.

2. Consider the program

$$S = \text{while } x > 0 \text{ do } x := x + 1 \text{ od}$$

in $Comm(\Sigma_{or})$, which we shall evaluate over the state $\sigma \in State(A)$, where

$$\sigma(x) = \pi.$$ 

Then

$$M^i_A(S)(\sigma) = M^i_A(\text{while } x > 0 \text{ do } x := x + 1 \text{ od})(\sigma)$$
$$= \bot$$

because the evaluation of the Boolean expression will always be true, so leading to a non-convergent computation sequence

$$\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n, \ldots$$

in which $\sigma_n(x) = \pi + n$. 

However, if we take the initial state $\sigma$, over which we evaluate $S$ to have

$$\sigma_{\text{read}}(x) = -1$$

then

$$M_A^\omega(S)(\sigma) \downarrow \sigma$$

as

$$W_A(x > 0)(\sigma) = ff$$

giving a computation sequence of one state $\sigma_0 = \sigma$.

### 13.6.3 Non-Termination

Because **while** statements may not terminate, for $S \in \text{Comm}(\Sigma)$, $M_A^\omega(S)(\sigma)$ may be a partial function, and when $M_A^\omega(S)(\sigma)$ converges to a final state $\tau$ we shall write:

$$M_A^\omega(S)(\sigma) \downarrow \tau$$

and

$$M_A^\omega(S)(\sigma) \uparrow$$

if $S$ does not terminate.

It is sometimes convenient to make $M_A^\omega(S)(\sigma)$ a total function, which is achieved by the introduction of a special element $\bot$ (read as *bottom*) by the following operation:

$$\text{State}_\bot(A) = \text{State}(A) \cup \{\bot\}$$

and defining

$$M_A^\omega(\textbf{while } b \textbf{ do } S \textbf{ od})(\sigma) = \begin{cases} \tau & \text{if } M_A^\omega(\textbf{while } b \textbf{ do } S \textbf{ od})(\sigma) \downarrow \tau; \\ \bot & \text{if } M_A^\omega(\textbf{while } b \textbf{ do } S \textbf{ od})(\sigma) \uparrow. \end{cases}$$

In defining $M_A^\omega(S)$ for $S \in \text{Comm}(\Sigma)$ we have formalised conditions for final states to exist. These conditions are straightforward. It is important to note that listing and formalising the conditions under which final states do *not* exist is quite subtle.

Take the case of iteration. For $\sigma \in \text{State}(A)$, suppose that

$$M_A^\omega(\textbf{while } b \textbf{ do } S_0 \textbf{ od})(\sigma) \uparrow$$

This corresponds to the computation failing to exit the **while** loop. This situation arises in two ways. Consider the repeated execution of $S_0$ starting in state $\sigma$. Either:

(i) there is an infinite sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$$

of states such that for each $n \geq 0$,

$$W_A(b)(\sigma_n) = tt$$

and

$$M_A(b)(\sigma_{n-1}) \downarrow \quad \text{and} \quad \sigma_n = M_A(S_0)(\sigma_{n-1}).$$

Here $S_0$ terminates and on each occasion the test $b$ is true.
(ii) Or there is a finite sequence
\[ \sigma_0, \sigma_1, \ldots, \sigma_n \]
of states such that for \( 1 \leq i \leq n \),
\[ W_A(b)(\sigma_i) = tt, \]
and for \( 0 \leq i \leq n - 1 \),
\[ M_A^\sigma(S_0)(\sigma_i) \downarrow \quad \text{and} \quad \sigma_{i+1} = M_A^\sigma(S_0)(\sigma_i), \]
but
\[ M_A^\sigma(S_0)(\sigma_n) \uparrow. \]
Here \( S_0 \) terminates on \( \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \) but fails to terminate on \( \sigma_n \) (i.e., there is no \( \sigma_{n+1} \)).

The analysis of non-termination must now focus on \( S_0 \).

### 13.7 Statements and Commands: Second Definition using Recursion

Although the first definition of the input-output semantics \( M_A^\sigma \) is based on simple intuitions, the definition of the \texttt{while} construct is not as smooth as those of the other constructs. In the case of the \texttt{while} construct we abandon explicit equational definitions, used for sequencing and conditional constructs, and postulate the existence of a finite or infinite sequence of states. In logical terms this seems clumsy and more complicated.

An alternative approach is to develop an equational definition for the case of the \texttt{while} statement. Our intuitions about the processing of a \texttt{while} loop allow us to unfold a \texttt{while} loop using an \texttt{if-then-else} statement: we expect that the statement
\[ S \equiv \texttt{while } b \texttt{ do } S_0 \texttt{ od} \]
has the same effect on a state as the statement
\[ S' \equiv \texttt{if } b \texttt{ then } S_0; \texttt{while } b \texttt{ do } S_0 \texttt{ od} \texttt{ else skip fi} \]
which unfolds the first stage in the \texttt{while} loop.

Now both statements \( S \) and \( S' \) are valid \texttt{while} programs and hence have a formal input-output semantics \textit{according to the first definition}. The input-output semantics of the first definition are the same:

#### 13.7.1 Lemma

\begin{align*}
\text{For any } \sigma \in \text{State}(A), \quad \quad M_A^\sigma(\texttt{while } b \texttt{ do } S_0 \texttt{ od})(\sigma) & \simeq M_A^\sigma(\texttt{if } b \texttt{ then } S_0; \texttt{while } b \texttt{ do } S_0 \texttt{ od} \texttt{ else skip fi})(\sigma) \\
\end{align*}

These semantic observations lead to a new semantic definition for the language in which the \texttt{while} statement case in the first definition is changed into a recursion.
Let this new semantics be denoted

\[ M_A^{rec} : Comm(\Sigma) \rightarrow (State(A) \rightarrow State(A)) \]

and defined for \( S \in Comm(\Sigma) \) and \( \sigma \in State(A) \) by induction on the structure \( S \) as follows:

For the basis cases and induction steps for sequencing and conditionals we use the same clauses as in the first definition.

For the while case we define:

\[
M_A^{rec}(\text{while } b \text{ do } S_0 \text{ od})(\sigma) = \begin{cases} 
M_A^{rec}(\text{while } b \text{ do } S_0 \text{ od})(M_A^{rec}(S_0)(\sigma)) & \text{if } W_A(b)(\sigma) = tt; \\
\sigma & \text{if } W_A(b)(\sigma) = ff. 
\end{cases}
\]

This recursive definition provides for each \( S \) an equation that \( M_A^{rec}(S) \) must satisfy. Thus, the complete mathematical definition for while programs is as follows:

\[
M_A^{rec} : \begin{align*}
M_A^{rec}(S)(\sigma) &= \text{the final state, if such a state exists, on executing a program } S \text{ on an initial state } \sigma. \\
M_A^{rec}(\text{skip})(\sigma) &= \sigma \\
M_A^{rec}(x := e)(\sigma) &= \sigma[V_A(e)(\sigma)/x] \\
M_A^{rec}(S_1; S_2)(\sigma) &= M_A^{rec}(S_2)(M_A^{rec}(S_1)(\sigma)) \\
M_A^{rec}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma) &= \begin{cases} 
M_A^{rec}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\
M_A^{rec}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = ff. 
\end{cases} \\
M_A^{rec}(\text{while } b \text{ do } S_0 \text{ od})(\sigma) &= \begin{cases} 
M_A^{rec}(\text{while } b \text{ do } S_0 \text{ od})(M_A^{rec}(S_0)(\sigma)) & \text{if } W_A(b)(\sigma) = tt; \\
\sigma & \text{if } W_A(b)(\sigma) = ff. 
\end{cases}
\]

By Lemma 13.7.1, we know that the state transformer \( M_A^{st} \) satisfies the equations generalised from \( S \). However,

(i) How many state transformers, in addition to \( M_A^{st}(S) \) satisfy the equations?

(ii) Are there extra properties that allow us to characterise the function \( M_A^{st}(S) \) as a unique solution of the equations?

13.8 Adding Data Types to Programming Languages

We have developed a formal definition of the syntax and semantics for the simple programming language

\[ WP(\Sigma) \]

of while programs that compute over an abstract data type with signature \( \Sigma \).

To compute over and implement the data type, we chose a \( \Sigma \)-algebra

\[ A \]
and defined the input-output semantics

$$M^o_A(S) : State(A) \rightarrow State(A)$$

of every program $$S \in Comm(\Sigma)$$ over $$A$$.

Suppose we want to enhance the power of WP by adding some constructs, such as

(i) dynamic arrays, or

(ii) infinite streams.

This is trivial given our methods. As we have emphasised repeatedly, we have solved the problem for while programming over any algebra $$A$$. In Chapter 6, we showed how to model arrays and streams over $$\Sigma$$ by new signatures and algebras.

### 13.8.1 Adding Dynamic Arrays

For any $$\Sigma$$-algebra $$A$$ we can construct the algebra

$$A_{Array}$$

with signature $$\Sigma_{Array}$$ of dynamic arrays over $$A$$.

So, given $$WP(\Sigma)$$, we can add dynamic arrays to our while programming language over $$\Sigma$$ simply by forming the language

$$WP(\Sigma_{Array}).$$

We can obtain its semantics by applying our input-output model to $$\Sigma_{Array}$$-algebra $$A_{Array}$$.

### 13.8.2 Adding Infinite Streams

For any $$\Sigma$$-algebra $$A$$ we can construct the algebra

$$A_{Stream}$$

with signature $$\Sigma_{Stream}$$ of infinite streams over $$A$$.

So, given $$WP(\Sigma)$$, we can add infinite streams to our while programming language over $$\Sigma$$ simply by forming the language

$$WP(\Sigma_{Stream}).$$

We can then obtain its semantics by applying our input-output semantics model to $$\Sigma_{Stream}$$-algebra $$A_{Stream}$$. 
Exercises for Chapter 13

1. Show for any state $\sigma \in \text{State}(A)$ and $x, y \in \text{Var}$:
   a. $\sigma[\sigma(x)/x] = \sigma$;
   b. $(\sigma[a_1/x])[a_2/x] = \sigma[a_2/x]$; and
   c. $(\sigma[a_1/x])[a_2/y] = (\sigma[a_2/y])[a_1/x]$ if $x \neq y$.

2. Add the following constructs to the $WP(\Sigma)$ and define their semantics:
   (i) concurrent assignments;
   (ii) case statements;
   (iii) repeat-until statements; and
   (iv) for statements.

3. Define the set $\text{Var}(S)$ of variables occurring in a program $S$. If $v \notin \text{Var}(S)$ then is it the case that for all $\sigma \in \text{State}(A)$
\[
M^0_A(S)(\sigma)(v) = \sigma(v)\
\]


5. Using the semantics of while programs work out the final state for the following program:
\[
y := 1 ; \\
c := 1 ; \\
\textbf{while} c <= x \textbf{do} \\
\hspace{1em} y := y + c ; \\
\hspace{1em} c := c + 1 \\
\textbf{od}
\]
with start state $\sigma(x) = 10$.

6. What does the above program return when the start state is $\sigma(x) = v$?

7. Using the semantics evaluate the behaviour of the following program:
\[
y := 1 ; \\
x := 1 ; \\
c := 1 ; \\
\textbf{while} c < z \textbf{do} \\
\hspace{1em} d := x ; \\
\hspace{1em} x := x + y ; \\
\hspace{1em} y := d ; \\
\hspace{1em} c := c + 1 \\
\textbf{od}
\]
with start states $\sigma(z) = 2$, $\sigma(z) = 3$ and $\sigma(z) = 4$.

8. What numbers does the above program produce in terms of $z$?
Chapter 14

Proving Properties of Programs

Structural induction is a simple and essential technique for defining syntax, defining functions on syntax, and proving facts about syntax and its semantics. Structural induction is based on how syntax is built up: The definitions postulate some basic syntax and generate new syntax by repeatedly applying operations. Functions are defined, or properties proved, first for the basic syntax and next by specifying the effect of the syntax forming operations. This type of definition or proof is traditionally called

\textit{definition or proof by induction or recursion on the structure of the syntax.}

In practice, the terms structural induction or recursion are applied rather loosely. Structural induction has a huge range of applications.

In this chapter we will reflect on structural induction and prepare the reader for its extensive use. There are many forms of structural induction since there are many types of syntax. We will introduce only a few simple principles. Although structural induction is based on syntax, it is not confined to syntax. Indeed, it originates in induction and recursion on the set \( \mathbb{N} \) of natural numbers which we discussed in Chapter 7 when analysing the data type of natural numbers.

The syntax of interest to us is essentially that of programs. Structural induction is used to define functions on all programs, or prove properties true of all programs. Already, we have seen several examples of definitions of syntax by structural induction, including those of \( \Sigma \)-terms over a signature \( \Sigma \) (Chapter 12); and expressions, Boolean expressions, and commands for \( WP(\Sigma) \) in Chapter 12. There are plenty of functions to construct and properties to prove.

In the case of functions, the way we defined the semantics of while programs, in the last chapter, involved structural inductions on expressions, Boolean expressions and commands. So we have already introduced and used structural induction to define the most important functions in our semantics!

We will meet many examples of such definitions. For example, also definable by structural induction are program transformations

\[ t : \mathcal{L} \rightarrow \mathcal{L} \]

for a programming language \( PL \), and compilers

\[ c : \mathcal{L} \rightarrow \mathcal{L}' \]
which map the programs of one language $\mathcal{L}$ to equivalent programs of another $\mathcal{L}'$. However, we have yet to meet proofs of program properties based on structural induction. In this chapter, we concentrate on proving properties common to all while programs.

To prove facts about while programs, we need to be able to prove facts about expressions, boolean expressions and commands. Thus, for WP we need three structural induction principles corresponding with expressions, Boolean expressions and statements. After some simple results about side effects, we prove two important theorems.

The first theorem establishes the fact that:

> Each computation by any while program on any $\Sigma$-algebra $A$ takes place inside the $\Sigma$-subalgebra of $A$ generated by the program’s input data.

This is called the Local Computation Theorem. It gives us new insight into the way programs are absolutely dependent upon their underlying data types. Among several consequences of the theorem is the fact that:

> The square root function $\sqrt{x}$ on the set of real numbers cannot be computed by any while program using the operations $x + y$, $-x$, $x.y$, $x^{-1}$ and tests $=$ and $<$. The second theorem establishes the fact that:

> If $A$ and $B$ are isomorphic $\Sigma$-algebras, then the input-output semantics of any while program on $A$ and $B$ are isomorphic.

This fact is called the Isomorphism Invariance Theorem. It gives us further insight into the concept of interface and implementation since it confirms that the semantics of while programs is equivalent over equivalent implementations of a data type. Among many consequences is the fact that

> while computation over decimal numbers is isomorphic to while computation over binary numbers.

## 14.1 Principles of Structural Induction for Programming Language Syntax

Suppose some syntax is defined by giving some basic or atomic syntax, and then generating new syntax by repeatedly applying syntax-forming operations. Then this is the general idea of structural induction for proving properties of the syntax:

Suppose that some property $P$ is true of the basic syntax and that if $P$ is true of some syntax, then $P$ also remains true after applying the syntax-forming operations. Then the property $P$ is true of all the syntax.

We have defined the expressions, Boolean expressions and programs from some basic given syntax by applying syntax-forming operations. Recall the argument about generating the natural numbers from 0 and $\text{succ}$ and the induction principle on numbers, and compare it with the ideas of generating syntax and principles of induction for expressions, Boolean expressions and programs.
14.1. PRINCIPLES OF STRUCTURAL INDUCTION FOR PROGRAMMING LANGUAGE SYNTAX

14.1.1 Principle of Induction for Expressions

Principle of Induction for Expressions

Let \( \Sigma \) be a signature and \( \text{Exp}(\Sigma) \) the set of all expressions over \( \Sigma \). Let \( P \) be a property of expressions, i.e.,

\[
P \subseteq \text{Exp}(\Sigma) \text{ is a set of expressions having that property.}
\]

We write \( P(e) \) for \( e \in P \).

If the following two cases hold:

**Base Case**

- \( P(c) \) is true for each constant \( c \in \Sigma \);
- \( P(x) \) is true for each variable \( x \in \text{Var} \).

**Induction Step**

Let \( e_1, \ldots, e_n \in \text{Exp}(\Sigma) \) be any expressions of sorts \( s(1), \ldots, s(n) \) respectively. Let \( f : s(1) \times \cdots \times s(n) \to s \in \Sigma \) be any function. If \( P(e_1), \ldots, P(e_n) \) are all true then

\[
P(f(e_1, \ldots, e_n))
\]

is also true.

Then

\( P(e) \) is true for all expressions \( e \in \text{Exp}(\Sigma) \).

We now formulate an analogous structural induction principle for Boolean expressions.

14.1.2 Principle of Structural Induction for Boolean Expressions

The Boolean expressions over a signature \( \Sigma \) are contained in \( \text{Exp}_{\text{Bool}}(\Sigma) \); we give them special treatment:
Principle of Structural Induction for Boolean Expressions

Let $\Sigma$ be a signature and $\text{BExp}(\Sigma)$ the set of all Boolean expressions over $\Sigma$. Let $P$ be a property of Boolean expressions, i.e.,

$$P \subseteq \text{BExp}(\Sigma)$$

is a set of Boolean expressions having that property.

We write $P(b)$ for $b \in P$.

If the following two cases hold:

**Base Case**

$P(\text{true})$ is true;

$P(\text{false})$ is true.

**Induction Step**

Let $b, b_1, b_2 \in \text{BExp}(\Sigma)$ be any Boolean expressions. If $P(b)$, $P(b_1)$, $P(b_2)$ are all true then

$$P(\text{not}(b)) \text{ and } P(\text{and}(b_1, b_2))$$

are also true.

Let $e_1, \ldots, e_n \in \text{Exp}(\Sigma)$ be any expressions of sorts $s(1), \ldots, s(n)$ respectively. Let $r : s(1) \times \cdots \times s(n) \to \text{bool} \in \Sigma$ be any relation. If $P(e_1), \ldots, P(e_n)$ are all true then

$$P(r(e_1, \ldots, e_n))$$

is also true.

Then

$P(b)$ is true for all Boolean expressions $b \in \text{BExp}(\Sigma)$.

14.1.3 Principle of Structural Induction for Statements

Finally, there are the programs. The while programs are made from skip and assignments by the application of program-forming operations of composition, conditional and iteration.
14.1. PRINCIPLES OF STRUCTURAL INDUCTION FOR PROGRAMMING LANGUAGE SYNTAX

Principle of Induction for Statements

Let \( \Sigma \) be a signature and \( \text{Comm}(\Sigma) \) the set of all commands over \( \Sigma \). Let \( P \) be a property of statements, i.e.,

\[ P \subseteq \text{Comm}(\Sigma) \] is a set of statements having that property.

We write \( P(S) \) is true for \( S \in P \).

If the following two cases hold:

**Base Case**
- \( P(\text{skip}) \) is true;
- \( P(x:=e) \) is true for each variable \( x \in \text{Var} \) and expression \( e \in \text{Exp}(\Sigma) \).

**Induction Step** If \( P(S_1) \) and \( P(S_2) \) are true then

\[ P(S_1 ; S_2) \]

is true;

if \( P(S_1) \) and \( P(S_2) \) are true then

\[ P(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}) \]

is true for all \( b \in \text{BExp}(\Sigma) \);

if \( P(S_0) \) is true then

\[ P(\text{while } b \text{ do } S_0 \text{ od}) \]

is true for all \( b \in \text{BExp}(\Sigma) \).

Then

\[ P(S) \] is true for all commands \( S \in \text{Comm}(\Sigma) \).

14.1.4 Proving the Principles of Structural Induction

Each principle of structural induction can be proved from the Principle of Induction for the natural numbers. That is, each principle is a theorem, the proof of which is has the form: The technique is to define the height

\[ h : \text{Exp}(\Sigma) \to \mathbb{N} \] of expressions;

\[ h : \text{BExp}(\Sigma) \to \mathbb{N} \] of Boolean expressions; and

\[ h : \text{Comm}(\Sigma) \to \mathbb{N} \] of commands

and transfer the induction to one in terms of the Principle of Induction for the natural numbers.
14.2 Reasoning about Side Effects Using Structural Induction

Now we will use the structural induction principles for expressions, Boolean expressions and commands to prove some basic properties of their semantics. We will show that:

If a variable \( v \) does not appear in a while program \( S \) then its value on any state \( \sigma \) is not changed by executing the program \( S \) on \( \sigma \).

This and related results are not difficult but they are important. First, we must introduce concepts to express these side-effect properties precisely.

Definition Two states \( \sigma, \sigma' \) are equivalent over an algebra \( A \) with respect to a set of variables \( V \), if

\[
\text{for all } v \in V, \sigma(v) = \sigma'(v);
\]

we write this

\( \sigma \simeq A \sigma' \mod V \)

and often, when \( A \) is obviously involved, we will just write this as

\( \sigma \simeq \sigma' \mod V \).

14.2.1 Theorem (No external side effects)

Let \( S \in \text{Comm}(\Sigma) \). Let \( A \) be a \( \Sigma \)-algebra. Let \( y \not\in \text{Var}(S) \). Then for all states \( \sigma \in \text{States}(A) \),

\[
M^o_A(S)(\sigma) \downarrow \quad \text{implies} \quad M^o_A(S)(\sigma) \simeq \sigma \mod \{ y \}.
\]

Proof. We use structural induction for \( \text{Comm}(\Sigma) \).

Base case

Case \( S ::= \text{skip} \).

We first note that \( S \) contains no variables. But, for any state \( \sigma \), as \( M^o_A(\text{skip})(\sigma) = \sigma \) it is immediate that

\[
M^o_A(\text{skip})(\sigma) \downarrow \quad \text{and} \quad M^o_A(\text{skip})(\sigma) \simeq \sigma \mod \{ x \}
\]

for all variables \( x \in \text{Var} \).

Case \( S ::= x := e \).

Suppose that \( y \not\in \text{Var}(S) \). Since

\[
M^o_A(x := e)(\sigma) = \sigma\{ V_A(e)(\sigma)/x \}
\]

entails that \( M^o_A(x := e)(\sigma) \downarrow \) and only the value of \( x \) has been changed (Section 13.4.2) we can say that

\[
M^o_A(x := e)(\sigma) \simeq \sigma \mod \{ y \}
\]

for any state \( \sigma \) and for all variables \( y \in \text{Var} \), with \( y \neq x \), and in particular for any \( y \not\in \text{Var}(S) \).
14.2. REASONING ABOUT SIDE EFFECTS USING STRUCTURAL INDUCTION

Induction Step

Case $S ::= S_1; S_2$.

Suppose that $y \notin \text{Var}(S)$, then $y \notin \text{Var}(S_1)$ and $y \notin \text{Var}(S_2)$. If $M^o_A(S)(\sigma) \downarrow$ then $M^o_A(S_1)(\sigma) \downarrow \tau$ and $M^o_A(S_2)(\tau) \downarrow$ for all terminating states $\sigma$. The Induction Hypothesis for $S_1$ and $S_2$ is:

(i) $M^o_A(S_1)(\sigma) \downarrow \Rightarrow M^o_A(S_1)(\sigma) \simeq \sigma \mod \{y\}$; and

(ii) $M^o_A(S_2)(\sigma) \downarrow \Rightarrow M^o_A(S_2)(\sigma) \simeq \sigma \mod \{y\}$ for all states $\sigma$.

Choose a state $M^o_A(S_1)(\sigma)$ and substituting into the equation for sequential composition we have:

$$M^o_A(S_2)(M^o_A(S_1)(\sigma)) \simeq M^o_A(S_1)(\sigma) \mod \{y\} \quad \text{by hypothesis (ii)}$$

$$\simeq \sigma \mod \{y\} \quad \text{by hypothesis (i)}$$

thus $M^o_A(S)(\sigma) = M^o_A(S_2)(M^o_A(S_1)(\sigma))$ implies that

$$M^o_A(S)(\sigma) = \sigma \mod \{y\}$$

for any state $\sigma$.

Case $S ::= \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$.

Suppose that $y \notin \text{Var}(S)$, then $y \notin \text{Var}(S_1)$ and $y \notin \text{Var}(S_2)$. If $M^o_A(S)(\sigma) \downarrow$ then $M^o_A(S_1)(\sigma) \downarrow$ and $M^o_A(S_2)(\sigma) \downarrow$. The Induction Hypothesis is:

$$M^o_A(S_1)(\sigma) \downarrow \Rightarrow M^o_A(S_1)(\sigma) \simeq \sigma \mod \{y\}$$

and

$$M^o_A(S_2)(\sigma) \downarrow \Rightarrow M^o_A(S_2)(\sigma) \simeq \sigma \mod \{y\}$$

for any state $\sigma$. Since evaluating a Boolean expression does not change the state and

$$M^i_A(S)(\sigma) = \begin{cases} M^o_A(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = \text{tt}; \\ M^o_A(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = \text{ff} \end{cases}$$

we have $M^o_A(S)(\sigma) \simeq \sigma \mod \{y\}$ for any state $\sigma$.

Case $S ::= \text{while } b \text{ do } S_0 \text{ od}$

Suppose that $y \notin \text{Var}(S)$, then $y \notin \text{Var}(S_0)$ and $y \notin \text{Var}(b)$. The Inductive Hypothesis is:

$$M^o_A(S_0)(\sigma) \downarrow \Rightarrow M^o_A(S_0)(\sigma) \simeq \sigma \mod \{y\}$$

for any state $\sigma$.

We define a sequence of states: $\sigma_0 = \sigma$ and $\sigma_{i+1} = M^i_A(S_0)(\sigma_i)$ for $i = 0, 1, 2, \ldots$ By a simple argument using the principle of induction on $\mathbb{N}$ it is clear that for each $i$,

$$\sigma_i \simeq \sigma \mod \{y\}.$$

Now consider $M^i_A(S)(\sigma)$ for any state $\sigma$. If $M^o_A(S)(\sigma) \downarrow$ then such a sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ exists, with $W_A(b)(\sigma_i) = \text{tt}$ for $0 \leq i < n$ and $W_A(b)(\sigma_n) = \text{ff}$. Since $M^o_A(S)(\sigma) = \sigma_n$, we have the required result from $\sigma_n \simeq \sigma \mod \{y\}$. □
14.2.2 Lemma (No internal side effects)
Let \( V \subset \text{Var} \) be a set of variables. Let \( \Sigma \) be a signature with \( \text{Comm}(\Sigma) \) the set of commands over \( \Sigma \). For any command \( S \in \text{Comm}(\Sigma) \) such that \( \text{Var}(S) \subset V \), we have
\[
(\sigma \simeq \sigma' \mod V) \Rightarrow (M^\sigma_A(S)(\sigma) \simeq M^{\sigma'}_A(S)(\sigma') \mod V)
\]
for any states \( \sigma, \sigma' \in \text{State}(A) \).

Proof. Left to the reader. Note the case when \( M^\sigma_A(S)(\sigma) \uparrow \). \( \square \)

14.3 Local Computation Theorem and Functions that Cannot be Programmed

Consider how a \textbf{while} program performs a computation. Notice that

- states store data from a \( \Sigma \)-algebra;
- assignment statements compute and store new data from data in states using the operations named in \( \Sigma \); and
- sequencing, conditional and iteration statements schedule computation by assignments, possibly using tests which do not change the values in states.

In particular:

Only the assignment statements compute new data from old, and they do so via expressions over \( \Sigma \).

Thus, given any \textbf{while} program \( S \) and input data \( a_1, \ldots, a_n \) for its variables, the values of these variables in all the states of the computation contain only data constructed from \( a_1, \ldots, a_n \) using expressions over \( \Sigma \). In particular, if there is a final state, then the

output of the computation are data constructed from \( a_1, \ldots, a_n \) using expressions over \( \Sigma \).

In this section, we are going to formulate these observations precisely and prove them true. The main result is called the \textit{Local Computation Theorem} and it has many important consequences.

14.3.1 Local Computation and Expressions

\textbf{Lemma} Let \( e \in \text{Exp}(\Sigma) \) be an expression whose variables are among
\[ v_1, \ldots, v_n. \]

Let \( A \) be any \( \Sigma \)-algebra. For any state \( \sigma \in \text{State}(A) \), the value \( V_A(e)(\sigma) \) of \( e \) on \( \sigma \) lies in the \( \Sigma \)-subalgebra of \( A \) generated by
\[
\sigma(v_1), \ldots, \sigma(v_n).
\]

In symbols, we have
\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]
14.3. LOCAL COMPUTATION THEOREM AND FUNCTIONS THAT CANNOT BE PROGRAMMED

Proof
We prove this by structural induction on the definition of expressions (Section 14.1.1). We will drop reference to sorts as this will not lead to confusion.

Base case There are two cases for \( e \):

Constant case \( e \equiv c \). Then

\[
V_A(e)(\sigma) = V_A(e)(\sigma) = c^A \quad \text{by definition of } V_A.
\]

The element \( c^A \) of \( A \) named by constant \( c \) belongs to every \( \Sigma \)-subalgebra of \( A \). Thus,

\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

Variable case \( e \equiv v_i \). Then

\[
V_A(e)(\sigma) = V_A(v_i)(\sigma) = \sigma(v_i) \quad \text{by definition of } V_A.
\]

Since \( \sigma(v_i) \) is one of the generators of the subalgebra, clearly

\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

Induction Step There is one case for \( e \).

Function application case \( e \equiv f(e_1, \ldots, e_n) \). As induction hypotheses, suppose that

\[
V_A(e_i)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

holds for the expressions \( e_i \in Exp(\Sigma) \) for \( 1 \leq i \leq n \). Then

\[
V_A(e)(\sigma) = V_A(f(e_1, \ldots, e_n))(\sigma) = f^A(V_A(e_1)(\sigma), \ldots, V_A(e_n)(\sigma)) \quad \text{by definition of } V_A.
\]

By the induction hypothesis, \( V_A(e)(\sigma) \) is obtained by applying the operation \( f^A \) to elements of the \( \Sigma \)-subalgebra \( \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle \) of \( A \). Any \( \Sigma \)-subalgebra is closed under operations, and so the value

\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

Since the Basis and Induction Step are true, we apply the Principle of Structural Induction for Expressions to conclude that for all \( e \in Exp(\Sigma) \), and any state \( \sigma \in State(A) \),

\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

\( \square \)
14.3.2 Local Computation and while Programs

**Theorem (Local Computation)** Let \( S \in WP(\Sigma) \) be a while program whose variables are among \( v_1, \ldots, v_n \).

Let \( A \) be any \( \Sigma \)-algebra. For any state \( \sigma \in \text{State}(A) \), the values of the variables in the final state \( M_\sigma^A(S)(\sigma) \), if it exists, lie in the \( \Sigma \)-subalgebra of \( A \) generated by

\[ \sigma(v_1), \ldots, \sigma(v_n). \]

In symbols, if \( M_\sigma^A(S)(\sigma) \perp \), then for each \( v \in \{v_1, \ldots, v_n\} \),

\[ M_\sigma^A(S)(\sigma)(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle. \]

**Proof**

We prove this by structural induction on the definition of commands.

**Base case** There are two types of atomic statement.

**Case 1:** \( S \equiv x:=e \). Then

\[
M_\sigma^A(S)(\sigma) = M_{\phi}(x:=e)(\sigma) = \sigma[V_A(e)(\sigma)/x] \quad \text{by definition of } M_{\phi}^A
\]

Consider the value of the final state on each variable \( v \in \{v_1, \ldots, v_n\} \). There are two subcases depending on whether \( v \) is \( x \in \{v_1, \ldots, v_n\} \).

**Subcase 1a:** \( v \equiv x \). Then

\[
M_\sigma^A(S)(\sigma)(v) = V_A(e)(\sigma) \quad \text{by definition of variable substitution.}
\]

Since \( \text{var}(e) \subseteq \{v_1, \ldots, v_n\} \), we can apply the Local Evaluation Lemma and deduce that

\[
V_A(e)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_k) \rangle.
\]

**Subcase 1b:** \( v \not\equiv x \). Then

\[
M_\sigma^A(S)(\sigma)(v) = \sigma(v).
\]

This is clearly one of the generators of \( \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle \).

**Case 2:** \( S \equiv \text{skip} \). Then

\[
M_\sigma^A(S)(\sigma) = M_{\phi}(\text{skip})(\sigma) = \sigma \quad \text{by definition of } M_{\phi}^A
\]

Clearly, as in subcase 1b,

\[
M_\sigma^A(S)(\sigma)(v) = \sigma(v)
\]

is one of the generators of \( \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle \).

**Induction Step** The three control constructs determine three cases. As induction hypotheses, suppose that the locality property holds for subprograms \( S_0, S_1 \) and \( S_2 \). That is, for any state \( \sigma \in \text{State}(A) \),

\[
M_\sigma^A(S_i)(\sigma)(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]
for $i = 0, 1, 2$.

Case 3: $S \equiv S_1; S_2$. Then

$$M^A_{i_0}(S)(\sigma) = M^A_{i_0}(S_1; S_2)(\sigma) = M^A_{i_0}(S_2)(M^A_{i_0}(S_1)(\sigma)) \text{ by definition of } M^A_{i_0}.$$ 

By the induction hypothesis for $S_1$, we know that for each $v \in \{v_1, \ldots, v_n\}$,

$$M^A_{i_0}(S_1)(\sigma)(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle. \quad (14.1)$$

By the induction hypothesis for $S_2$, applied to the input state $M^A_{i_0}(S_1)(\sigma)$, we know that for each $v \in \{v_1, \ldots, v_n\}$,

$$M^A_{i_0}(S_2)(M^A_{i_0}(S_1)(\sigma))(v) \in \langle M^A_{i_0}(S_1)(\sigma)(v_1), \ldots, M^A_{i_0}(S_1)(\sigma)(v_n) \rangle. \quad (14.2)$$

By the Local Evaluation Lemma applied to Equation 14.1, we deduce that

$$\langle M^A_{i_0}(S_1)(\sigma)(v_1), \ldots, M^A_{i_0}(S_1)(\sigma)(v_n) \rangle \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle$$

and by Equation 14.2, we may conclude that

$$M^A_{i_0}(S)(\sigma)(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.$$

Case 4: $S \equiv \mathrm{if} \ b \ \mathrm{then} \ S_1 \ \mathrm{else} \ S_2 \ \mathrm{fi}$. Then

$$M^A_{i_0}(S)(\sigma) = M^A_{i_0}(\mathrm{if} \ b \ \mathrm{then} \ S_1 \ \mathrm{else} \ S_2 \ \mathrm{fi})(\sigma) = \begin{cases} M^A_{i_0}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = \text{tt}; \\ M^A_{i_0}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = \text{ff}; \end{cases}$$

By definition of $M^A_{i_0}$. Clearly, there are two subcases.

Subcase 4a: $W_A(b)(\sigma) = \text{tt}$. Then

$$M^A_{i_0}(S)(\sigma)(v) = M^A_{i_0}(S_1)(\sigma)(v)$$

which is in $\langle \sigma(v_1), \ldots, \sigma(v_n) \rangle$ by the induction hypothesis for $S_1$.

Subcase 4b: $W_A(b)(\sigma) = \text{ff}$. Then

$$M^A_{i_0}(S)(\sigma)(v) = M^A_{i_0}(S_2)(\sigma)(v)$$

which is in $\langle \sigma(v_1), \ldots, \sigma(v_n) \rangle$ by the induction hypothesis for $S_2$.

Case 5: $S \equiv \mathrm{while} \ b \ \mathrm{do} \ s_0 \ \mathrm{od}$. First, we consider the potentially infinite sequence

$$\sigma = \sigma_0, \sigma_1, \ldots, \sigma_i, \sigma_{i+1}, \ldots$$

of states generated by applying $S_0$ repeatedly, starting on the initial state $\sigma$. Specifically, for $i \geq 0$, we define

$$\sigma_{i+1} = M^A_{i_0}(S)(\sigma_i).$$
**Claim.** For each \( i \geq 0 \), if \( \sigma_i \) exists, then
\[
\sigma_i(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]

**Proof of Claim.** We prove this using the Principle of Induction on the Natural Numbers (recall Section 7.4) applied to the sequence index \( i \geq 0 \).

**Basis:** \( i = 0 \). Now \( \sigma_0(v) = \sigma(v) \) by definition of \( \sigma_0 \), and \( \sigma(v) \) is one of the generators.

**Induction Step:** As induction hypothesis, suppose that \( \sigma_i \) exists and
\[
\sigma_i(v) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle. \tag{14.3}
\]
Suppose that \( \sigma_{i+1} = M^A_\emptyset(S_0)(\sigma_i) \downarrow \). Since we are reasoning within the scope of Case 5 of the Structural Induction Step, by the induction hypothesis for statements, we have
\[
\sigma_{i+1} = M^A_\emptyset(S_0)(\sigma_i) \in \langle \sigma_i(v_1), \ldots, \sigma_i(v_n) \rangle. \tag{14.4}
\]
By the Local Evaluation Lemma applied to Equation 14.3, we have
\[
\langle \sigma_i(v_1), \ldots, \sigma_i(v_n) \rangle \subseteq \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]
Hence, by Equation 14.4, we have
\[
\sigma_{i+1} \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]
By the Principle of Induction on the Natural Numbers, the claim holds.

Having proved the Claim, it is straightforward to conclude Case 5. Recall that
\[
M^A_\emptyset(S)(\sigma) \simeq M^A_\emptyset(\textbf{while } b \textbf{ do } S_0 \textbf{ od})(\sigma) \\
\simeq \sigma_{k+1}
\]
for some \( k + 1 \) such that
\begin{itemize}
  \item[(i)] for all \( 0 \leq i \leq k + 1 \), \( \sigma_i \) exists;
  \item[(ii)] for all \( 0 \leq i \leq k \), \( W_A(b)(\sigma_i) = tt \);
  \item[(iii)] \( W_A(b)(\sigma_{k+1}) = ff \).
\end{itemize}
By the Claim,
\[
M^A_\emptyset(S)(\sigma) \in \langle \sigma(v_1), \ldots, \sigma(v_n) \rangle.
\]
Having verified all cases of the Basis and Induction Steps, by the Principle of Structural Induction for Statements, we deduce that the theorem is true for all programs. \( \square \)
14.3. LOCAL COMPUTATION THEOREM AND FUNCTIONS THAT CANNOT BE PROGRAMMED

14.3.3 Local Computation and Functions

Corollary Let $A$ be any $\Sigma$-algebra. Let

$$F : A^w \rightarrow A$$

be a function on $A$. If $F$ is computable by a while program over $\Sigma$, then for each $(a_1, \ldots, a_n) \in A^w$,

$$F(a_1, \ldots, a_n) \in \langle a_1, \ldots, a_n \rangle.$$  

Consider the field of real numbers with signature:

<table>
<thead>
<tr>
<th>signature</th>
<th>Reals</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>real, bool</td>
</tr>
<tr>
<td>constants</td>
<td>0, 1 : $\rightarrow$ real</td>
</tr>
<tr>
<td></td>
<td>tt, ff : $\rightarrow$ bool</td>
</tr>
<tr>
<td>operations</td>
<td>$+$ : real $\times$ real $\rightarrow$ real</td>
</tr>
<tr>
<td></td>
<td>$-$ : real $\rightarrow$ real</td>
</tr>
<tr>
<td></td>
<td>$.$ : real $\times$ real $\rightarrow$ real</td>
</tr>
<tr>
<td></td>
<td>$^{-1}$ : real $\rightarrow$ real</td>
</tr>
<tr>
<td></td>
<td>$\text{eq}$ : real $\times$ real $\rightarrow$ bool</td>
</tr>
<tr>
<td></td>
<td>$\text{less}_{\text{than}}$ : real $\times$ real $\rightarrow$ bool</td>
</tr>
</tbody>
</table>

Theorem Let $R$ be the standard $\Sigma_{\text{Reals}}$-algebra. Then the partial function $F : R \rightarrow R$ defined by

$$F(x) = +\sqrt{x}$$

is not while computable over $R$.

Proof Suppose, for a contradiction, that there was a program $S \in \text{WP}(\Sigma)$ that computed $F$. Then, by the Corollary, for every $x \in R$,

$$F(x) \in \langle x \rangle.$$  

Consider $F(2)$. Now $\langle 2 \rangle$ is a two-sorted $\Sigma_{\text{Reals}}$-subalgebra of $R$ whose carriers are as follows:

$$\langle 2 \rangle_{\text{real}} = \mathbb{Q}$$

$$\langle 2 \rangle_{\text{bool}} = \mathbb{B}.$$  

By the irrationality of $\sqrt{2}$, we know that $F(2) = \sqrt{2} \notin \mathbb{Q} = \langle 2 \rangle$. This contradicts the existence of the while program that computes $F$.  

\hfill \Box
14.4 Invariance of Semantics

Our definition, in Chapter 13, of the input-output semantics of while programs on any algebra models the input-output behaviour of any while program on any implementation of any data type.

Now, in Chapter 7, we learned the following. A many-sorted algebra $A$ of signature $\Sigma$ is a model of a concrete implementation of a data type. A class of many-sorted algebras of signature $\Sigma$ is a model of a data type. If two $\Sigma$-algebras $A$ and $B$ are isomorphic then this models their equivalence as implementations of a data type. A class of many-sorted algebras of signature $\Sigma$ closed under isomorphism is a model of an abstract data type.

What is abstract about an abstract data type is the idea that computations with data need not depend on the way data is implemented. Equivalent implementations provide equivalent program behaviour. We will now make this idea precise by combining our input-output semantics with our algebraic models of data types. We ask the following question:

Suppose $A$ and $B$ are isomorphic $\Sigma$-algebras. Is computation by a while program $S \in WP(\Sigma)$ on $A$ equivalent with computation by $S$ on $B$?

The computations should be equivalent. For example, in the context of the abstract data type of natural numbers, the general question includes the simple question:

Given isomorphic algebras $N_{\text{dec}}$ and $N_{\text{bin}}$ of decimal and binary representations of the natural numbers, do while programs — such as Euclid's algorithm — have equivalent semantic behaviour on $N_{\text{dec}}$ and $N_{\text{bin}}$?

We will formulate and prove the Isomorphism Invariance Theorem for the semantics, that

for any while program $S \in WP(\Sigma)$, if $A$ and $B$ are isomorphic $\Sigma$-algebras, then the input-output semantics of $S$ on $A$ and $B$ are isomorphic.

We make precise what we mean by the

equivalence of computations

by a program, or more specifically for the main theorem, the

isomorphism of two input-output semantics

for while programs.

The starting point is to compare the states of two computations. Let us explore the problem by looking at the natural number data type and the question about $N_{\text{dec}}$ and $N_{\text{bin}}$ above.

14.4.1 Example of the Natural Numbers

Recall the two-sorted signature of the natural numbers of Section 7.1:
Consider two specific implementations of the data type of natural numbers:

(i) a decimal interpretation $N_{\text{dec}}$ of the natural numbers; and

(ii) a binary interpretation $N_{\text{bin}}$ of the natural numbers.

Let $E \in \text{Comm}(\Sigma)$ be the \textbf{while} program for Euclid’s algorithm:

\[
\begin{align*}
z &:= x \mod y; \\
\text{while } z \neq 0 \text{ do} & \\
& \quad x := y; \\
& \quad y := z; \\
& \quad z := x \mod y \\
\text{od}
\end{align*}
\]

For an input state $\sigma$ with $\sigma_{\text{nat}}(x) = m$ and $\sigma_{\text{nat}}(y) = n$, the program returns $\gcd(m, n)$ as the value of $y$.

Consider the execution of $E$ over the natural numbers under the two implementations of this data type, $N_{\text{dec}}$ and $N_{\text{bin}}$.

Consider case (i) of executing the program $E$ over a decimal implementation $N_{\text{dec}}$ of the natural numbers, starting from a given state $\sigma_{\text{dec}} \in \text{State}(N_{\text{dec}})$ with

\[
\sigma_{\text{dec}}(x) = 45 \text{ and } \sigma_{\text{dec}}(y) = 12.
\]

<table>
<thead>
<tr>
<th>Step Number</th>
<th>$\sigma_{\text{dec}}(x)$</th>
<th>$\sigma_{\text{dec}}(y)$</th>
<th>$\sigma_{\text{dec}}(z)$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>12</td>
<td>?</td>
<td>Initial state</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>12</td>
<td>9</td>
<td>First assignment</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>Entered loop</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>9</td>
<td>3</td>
<td>Re-enter loop</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>Exit loop</td>
</tr>
</tbody>
</table>

Then we have that

\[
M_{40}^{\text{dec}}(E)(\sigma_{\text{dec}}) = \tau_{\text{dec}}
\]

where

\[
\tau_{\text{dec}}(x) = 9 \quad \tau_{\text{dec}}(y) = 3 \quad \tau_{\text{dec}}(x) = 0
\]
Consider case (ii) of executing the program $E$ over a binary implementation $N_{\text{bin}}$ of the natural numbers, starting from a given state $\sigma_{\text{bin}} \in \text{State}(N_{\text{bin}})$ with

$$\sigma_{\text{bin}}(x) = 101101 \text{ and } \sigma_{\text{bin}}(y) = 1100.$$ 

<table>
<thead>
<tr>
<th>Step Number</th>
<th>$\sigma_{\text{bin}}(x)$</th>
<th>$\sigma_{\text{bin}}(y)$</th>
<th>$\sigma_{\text{bin}}(z)$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101101</td>
<td>1100</td>
<td>?</td>
<td>Initial state</td>
</tr>
<tr>
<td>2</td>
<td>101101</td>
<td>1100</td>
<td>1001</td>
<td>First assignment</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>1100</td>
<td>1001</td>
<td>Entered loop</td>
</tr>
<tr>
<td>4</td>
<td>1100</td>
<td>1001</td>
<td>1001</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1100</td>
<td>1001</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1001</td>
<td>1001</td>
<td>11</td>
<td>Re-enter loop</td>
</tr>
<tr>
<td>7</td>
<td>1001</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1001</td>
<td>11</td>
<td>0</td>
<td>Exit loop</td>
</tr>
</tbody>
</table>

Then we have that

$$M_{\text{to}}^{\text{bin}}(E)(\sigma_{\text{bin}}) = \tau_{\text{bin}}$$

where

$$\tau_{\text{bin}}(x) = 1001 \quad \tau_{\text{bin}}(y) = 11 \quad \tau_{\text{bin}}(z) = 0$$

Clearly, the states correspond exactly in the sense that decimal to binary conversion applied to the values in decimal output state $\tau_{\text{dec}}$ results in binary output state $\tau_{\text{bin}}$.

If

$$\phi : N_{\text{dec}} \rightarrow N_{\text{bin}}$$

is the decimal to binary conversion $\Sigma_{\text{Euclidean Algorithm}}$-isomorphism, then we can construct a conversion map

$$\hat{\phi} : \text{State}(N_{\text{dec}}) \rightarrow \text{State}(N_{\text{bin}})$$

for states such that

$$\hat{\phi}(\sigma_{\text{dec}}) = \sigma_{\text{bin}}$$

where

$$\hat{\phi}(\sigma_{\text{nat}}(v)) = \phi(\sigma_{\text{nat}}(v))$$

and so

$$\hat{\phi}(\tau_{\text{dec}}) = \tau_{\text{bin}}$$

### 14.4.2 Isomorphic State Spaces

Let $\Sigma$ be an $S$-sorted signature and let $A$ and $B$ be $\Sigma$-algebras. Let $\phi : A \rightarrow B$ be a $\Sigma$-map. This induces a map

$$\hat{\phi} : \text{State}(A) \rightarrow \text{State}(B)$$

defined as follows.

Recall that the state space $\text{State}(A)$ is a family

$$\text{State}(A) = \langle \text{State}_s(A) \mid s \in S \rangle$$

and a state $\sigma \in \text{State}(A)$ is a family

$$\sigma = \langle \sigma_s \mid s \in S \rangle$$
where the $s$-sorted state $\sigma_s \in \text{State}_s(A)$.

Furthermore, the map $\phi$ is a family

$$\phi = \langle \phi_s \mid s \in S \rangle$$

where $\phi_s : A_s \to B_s$. The state map $\hat{\phi}$ is a family

$$\hat{\phi} = \langle \hat{\phi}_s \mid s \in S \rangle$$

where $\hat{\phi}_s : \text{State}_s(A) \to \text{State}_s(B)$.

For any $\sigma_s \in \text{State}_s(A)$ and $x \in \text{Var}_s$, we define $\hat{\phi}_s$ by

$$\hat{\phi}_s(\sigma_s)(x) = \phi_s(\sigma_s(x)).$$

Suppose $\text{Var}_s = \{x_1, x_2, x_3, \ldots\}$ and let $\sigma_s \in \text{State}_s(A)$ be depicted by:

<table>
<thead>
<tr>
<th>Var_s</th>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>\ldots</th>
<th>x_i</th>
<th>\ldots</th>
<th>x_j</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\sigma_s</td>
<td>a_1</td>
<td>a_2</td>
<td>a_3</td>
<td>\ldots</td>
<td>a_i</td>
<td>\ldots</td>
<td>a_j</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

then we depict $\hat{\phi}_s(\sigma_s) \in \text{State}_s(B)$ as follows:

<table>
<thead>
<tr>
<th>Var_s</th>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>\ldots</th>
<th>x_i</th>
<th>\ldots</th>
<th>x_j</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>\hat{\phi}_s(\sigma_s)</td>
<td>\phi_s(a_1)</td>
<td>\phi_s(a_2)</td>
<td>\phi_s(a_3)</td>
<td>\ldots</td>
<td>\phi_s(a_i)</td>
<td>\ldots</td>
<td>\phi_s(a_j)</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

**Lemma** If the map $\phi$ is injective, surjective or bijective, then $\hat{\phi}$ is injective, surjective or bijective, respectively.

**Proof.** Suppose $\phi$ is injective. Let $\sigma, \sigma' \in \text{State}(A)$ and

$$\hat{\phi}(\sigma) = \hat{\phi}(\sigma').$$

Then for each $s \in S$, for all $\sigma_s \in \text{State}_s(A)$ and $x \in \text{Var}_s$,

$$\hat{\phi}_s(\sigma_s)(x) = \hat{\phi}_s(\sigma'_s)(x)$$

and by definition of $\hat{\phi}_s$,

$$\phi_s(\sigma_s(x)) = \phi_s(\sigma'_s(x)).$$

Since $\phi_s$ is injective, we have for all $x \in \text{Var}_s$,

$$\sigma_s(x) = \sigma'_s(x)$$

and hence

$$\sigma_s = \sigma'_s.$$ 

Thus, $\hat{\phi}_s$ is injective for each $s \in S$ and $\hat{\phi}$ is injective.

Suppose $\phi$ is surjective. For any $\tau \in \text{State}(B)$ we will construct $\sigma \in \text{State}(A)$ such that $\hat{\phi}(\sigma) = \tau$. Let $s \in S$ and $x \in \text{Var}_s$ and suppose

$$\tau_s(x) = b \in B_s.$$
Since $\phi_s$ is surjective there exists $a \in A_s$ such that
\[ \phi_s(a) = b. \]
For each $s \in S$ and $x \in Var_s$ we define $\sigma_s(x) = a$. Then for each $x \in Var_s$,
\[
\begin{align*}
\hat{\phi}(\sigma)(x) &= \phi_s(\sigma_s(x)) \\
&= \phi_s(\sigma_s(a)) \\
&= b \\
&= \tau_s(x) \\
&= \tau(x).
\end{align*}
\]
Combining these cases gives the case that $\phi$ is bijective. \hfill \Box

14.4.3 Isomorphism Invariance Theorem

The correspondence between computations on $A$ and $B$ under the isomorphic translation $\phi : A \rightarrow B$ is formalised by:

**Theorem (Input-Output Semantics)** Let $A$ and $B$ be $\Sigma$-structures and $\phi : A \rightarrow B$ a $\Sigma$-isomorphism. Let $\hat{\phi} : State(A) \rightarrow State(B)$ be the induced bijection between state spaces. Then for each statement $S \in Comm(\Sigma)$ and for every state $\sigma \in State(A)$,
\[
\hat{\phi}(M^A_B(S)(\sigma)) \simeq M^B_B(S)(\hat{\phi}(\sigma)).
\]

Alternatively, the following diagram commutes:

\[
\begin{array}{ccc}
State(A) & \xrightarrow{M^A_B(S)} & State(A) \\
\phi \downarrow & & \phi \downarrow \\
State(B) & \xrightarrow{M^B_B(S)} & State(B)
\end{array}
\]

As usual since the semantics of statements are constructed from the semantics of expressions and Boolean expressions, we will begin the proof by proving the algebraic invariance of the semantics of expressions and Boolean expressions. Each proof is by structural induction using the relevant principles from Chapter ??.

**Lemma (Isomorphism Invariance for Expressions)** For each expression $e \in Exp(\Sigma)$ and for every state $\sigma \in State(A)$ we have
\[
\phi(V_A(e)(\sigma)) = V_B(e)(\hat{\phi}(\sigma)).
\]
14.4. INVARiance OF SEMANTICS

Proof. We prove this by structural induction on the definition of expressions (recall Section 14.1.1). We will drop the reference to sorts as this will not lead to confusion.

**Base case** There are two cases:

**Constant case** \( e \equiv c \).

\[
\begin{align*}
\phi(V_A(c)(\sigma)) & = \phi(c^A) & \text{by definition of } V_A \\
& = e^B & \text{as } \phi \text{ preserves constants} \\
& = V_B(c)(\hat{\phi}(\sigma)) & \text{by definition of } V_B.
\end{align*}
\]

**Variable case** \( e \equiv x \).

\[
\begin{align*}
\phi(V_A(x)(\sigma)) & = \phi(x) & \text{by definition of } V_A \\
& = V_B(x)(\hat{\phi}(\sigma)) & \text{by definition of } V_B.
\end{align*}
\]

**Induction Step** As induction hypotheses, suppose that

\[
\phi(V_A(e_i)(\sigma)) = V_B(e_i)(\hat{\phi}(\sigma))
\]

holds for the expressions \( e_i \in \text{Exp}(\Sigma) \) for \( 1 \leq i \leq n \). Then for \( e \equiv f(e_1, \ldots, e_n) \),

\[
\begin{align*}
\phi(V_A(f(e_1, \ldots, e_n))(\sigma)) & = \phi(f^A(V_A(e_1)(\sigma), \ldots, V_A(e_n)(\sigma))) & \text{by definition of } V_A \\
& = f^B(\phi(V_A(e_1)(\sigma)), \ldots, \phi(V_A(e_n)(\sigma))) & \text{as } \phi \text{ preserves the operation named by } f \\
& = f^B(V_B(e_1)(\hat{\phi}(\sigma)), \ldots, V_B(e_n)(\hat{\phi}(\sigma))) & \text{by Induction Hypothesis} \\
& = V_B(f(e_1, \ldots, e_n))(\hat{\phi}(\sigma)) & \text{by definition of } V_B.
\end{align*}
\]

\( \square \)

**Lemma (Isomorphism Invariance for Boolean Expressions)** For each Boolean expression \( b \in \text{BExp}(\Sigma) \) and for every state \( \sigma \in \text{State}(A) \) we have

\[
\phi(W_A(b)(\sigma)) = W_B(b)(\hat{\phi}(\sigma)).
\]

Proof. We prove this by structural induction on the definition of Boolean expressions (recall Section 14.1.2).

**Base case** There are two base cases:

**Case** \( b \equiv \text{true} \).

\[
\begin{align*}
\phi(W_A(\text{true})(\sigma)) & = tt & \text{by definition of } W_A \\
& = W_B(\text{true})(\hat{\phi}(\sigma)) & \text{by definition of } W_B.
\end{align*}
\]

**Case** \( b \equiv \text{false} \).

\[
\begin{align*}
\phi(W_A(\text{false})(\sigma)) & = ff & \text{by definition of } W_A \\
& = W_B(\text{false})(\hat{\phi}(\sigma)) & \text{by definition of } W_B.
\end{align*}
\]

**Induction Step** There are three cases. As induction hypothesis, Suppose that

\[
\phi(W_A(b_i)(\sigma)) = W_B(b_i)(\hat{\phi}(\sigma))
\]
holds for the Boolean expressions $b_i \in EXP(\Sigma)$ for $i = 0, 1, 2$. Then:

Case $b \equiv \text{not}(b)$.

\[
\phi(W_A(\text{not}(b_0))(\sigma)) = \begin{cases} 
  tt & \text{if } \phi(W_A(b_0)(\sigma)) = ff; \\
  ff & \text{if } \phi(W_A(b_0)(\sigma)) = tt;
\end{cases}
\]

by definition of $W_A$

\[
= \begin{cases} 
  tt & \text{if } W_B(b_0)(\phi(\sigma)) = ff; \\
  ff & \text{if } W_B(b_0)(\phi(\sigma)) = tt;
\end{cases}
\]

by Induction Hypothesis

\[
= W_B(\text{not}(b_0))(\sigma)
\]

by definition of $W_B$.

Case $b \equiv \text{and}(b_1, b_2)$.

\[
\phi(W_A(\text{and}(b_1, b_2))(\sigma))
\]

\[
= \begin{cases} 
  tt & \text{if } \phi(W_A(b_1) (\sigma)) = tt \text{ and } \phi(W_A(b_2)(\sigma)) = tt; \\
  ff & \text{otherwise}
\end{cases}
\]

by definition of $W_A$

\[
= \begin{cases} 
  tt & \text{if } W_B(b_1)(\phi(\sigma)) = tt \text{ and } W_B(b_2)(\phi(\sigma)) = tt; \\
  ff & \text{otherwise}
\end{cases}
\]

by Induction Hypothesis

\[
= W_B(\text{and}(b_1, b_2))(\phi(\sigma))
\]

by definition of $W_B$.

Case $b \equiv r(e_1, \ldots, e_n)$.

\[
\phi(W_A(r(e_1, \ldots, e_n))(\sigma))
\]

\[
= \phi(r^n(V_A(e_1)(\sigma), \ldots, V_A(e_n)(\sigma)))
\]

by definition of $W_A$

\[
= r^B(\phi(V_B(e_1)(\sigma), \ldots, \phi(V_B(e_n)(\sigma)))
\]

as $\phi$ preserves relation named by $r$

\[
= r^B(V_B(e_1)(\phi(\sigma)), \ldots, V_B(e_n)(\phi(\sigma)))
\]

by Lemma 14.4.3

\[
= W_B(r(e_1, \ldots, e_n))(\phi(\sigma))
\]

by definition of $W_B$.

\[
\square
\]

We will use these results to prove the invariance result for commands.

**Theorem (Isomorphism Invariance Theorem)** For each statement $S \in Comm(\Sigma)$ and for every state $\sigma \in \text{State}(A)$ we have

\[
\hat{\phi}(M^a_A(S)(\sigma)) \simeq M^b_B(S)(\hat{\phi}(\sigma)).
\]

**Proof.** We prove this by structural induction on the definition of commands (recall Section 14.1.3).

**Base case** There are two base cases.

**Identity case** $S \equiv \text{skip}$.

\[
\hat{\phi}(M^a_A(\text{skip})(\sigma)) = \hat{\phi}(\sigma)
\]

by definition of $M^a_A$

\[
= M^b_B(\text{skip})(\hat{\phi}(\sigma))
\]

by definition of $M^b_B$.

**Assignment case** $S \equiv x:=e$.

\[
\hat{\phi}(M^a_A(x:=e)(\sigma)) \simeq \hat{\phi}(\sigma[V_A(e)(\sigma)/x])
\]

by definition of $M^a_A$

\[
\simeq (\hat{\phi}(\sigma))[\phi(V_A(e)(\sigma))/x]
\]

by Exercise ??(3)

\[
\simeq \hat{\phi}(\sigma)[V_B(e)(\phi(\sigma))/x]
\]

by Lemma 14.4.3

\[
\simeq M^b_B(x:=e)(\hat{\phi}(\sigma))
\]

by definition of $M^b_B$. 

14.4. INVARIANCE OF SEMANTICS

Induction Step There are three cases. As induction hypotheses, suppose that

$$\hat{\phi}(M_A^i(S_i)(\sigma)) \simeq M_B^i(S_i)(\hat{\phi}(\sigma))$$

holds for the statements $$S_i \in \text{Comm}(\Sigma)$$ for $$i = 0, 1, 2$$. Then:

Sequencing case $$S \equiv S_1; S_2$$.

$$\hat{\phi}(M_A^i(S_1; S_2)(\sigma)) \simeq \hat{\phi}(M_A^i(S_2)(M_A^i(S_1)(\sigma)))$$ by definition of $$M_A^i$$

$$\simeq M_B^i(S_2)(\hat{\phi}(M_A^i(S_1)(\sigma)))$$ by Induction Hypothesis on $$S_2$$

$$\simeq M_B^i(S_2)(M_B^i(S_1)(\hat{\phi}(\sigma)))$$ by Induction Hypothesis on $$S_1$$

$$\simeq M_B^i(S_1; S_2)(\hat{\phi}(\sigma))$$ by definition of $$W_B$$.

Conditional case $$S \equiv \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$$.

$$\hat{\phi}(M_A(S)(\sigma))$$

$$\equiv \begin{cases} \hat{\phi}(M_A(S_1)(\sigma)) & \text{if } W_A(b)(\sigma) = tt \\ \hat{\phi}(M_A(S_2)(\sigma)) & \text{if } W_A(b)(\sigma) = ff \end{cases}$$ by definition of $$M_A$$

$$\equiv \begin{cases} M_B(S_1)(\hat{\phi}(\sigma)) & \text{if } W_A(b)(\sigma) = tt \\ M_B(S_2)(\hat{\phi}(\sigma)) & \text{if } W_A(b)(\sigma) = ff \end{cases}$$ by Induction Hypothesis on $$S_1$$

$$\equiv \begin{cases} M_B(S_1)(\hat{\phi}(\sigma)) & \text{if } W_B(b)(\hat{\phi}(\sigma)) = tt \\ M_B(S_2)(\hat{\phi}(\sigma)) & \text{if } W_B(b)(\hat{\phi}(\sigma)) = ff \end{cases}$$ by Lemma 14.4.3

$$\equiv M_B(S)(\hat{\phi}(\sigma))$$ by definition of $$M_B$$.

Iteration case $$S \equiv \text{while } b \text{ do } S_0 \text{ od}$$.

First, we consider the iteration of the command $$S_0 \in \text{Comm}(\Sigma)$$ on $$A$$ and $$B$$. For $$\sigma \in \text{State}(A)$$ we define the iteration sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$$

of states by

$$\sigma_0 = \sigma \quad \text{and} \quad \sigma_n \simeq M_A^i(S_0)(\sigma_{n-1}).$$

For $$\tau \in \text{State}(B)$$ we define the iteration sequence

$$\tau_0, \tau_1, \ldots, \tau_n, \ldots$$

of states by

$$\tau_0 = \tau \quad \text{and} \quad \tau_n \simeq M_A^i(S_0)(\tau_{n-1}).$$

Note that either of these sequences can be finite because $$S_0$$ need not terminate.

Claim

If $$\tau = \hat{\phi}(\sigma)$$ then for all $$n \geq 0$$

$$\tau_n \simeq \hat{\phi}(\sigma_n).$$
**Proof.** This is proved by induction on \( n \). The basis case \( n = 0 \) is true by assumption.

As induction hypothesis, suppose that

\[
\tau_{n-1} \simeq \hat{\phi}(\sigma_{n-1}).
\]

This means either \( \tau_{n-1} \) and \( \sigma_{n-1} \) exist and \( \tau_{n-1} = \hat{\phi}(\sigma_{n-1}) \), or \( \tau_{n-1} \) and \( \sigma_{n-1} \) do not exist. Then

\[
\begin{align*}
\tau_n & \simeq M_B^\omega(S_0)(\tau_{n-1}) \quad \text{by definition} \\
& \simeq M_B^\omega(S_0)(\hat{\phi}(\sigma_{n-1})) \quad \text{by induction hypothesis on } n \\
& \simeq \hat{\phi}(M_A^\omega(S_0)(\sigma_{n-1})) \quad \text{by induction hypothesis on } S_0 \\
& \simeq \hat{\phi}(\sigma_n)
\end{align*}
\]

This proves the claim. \( \square \)

Now we consider the iteration of \( S_0 \) in the context of the while loop. There are two cases:

**Termination case**

Suppose \( M_A^\omega(S) \downarrow \). Then the sequence

\[
\sigma_0, \sigma_1, \ldots, \sigma_n
\]

of iterates of \( S_0 \) on \( \sigma \) over \( A \) is finite and

\[
W_A(b)(\sigma_i) = tt \quad 1 \leq i \leq n - 1 \text{ and } W_A(b)(\sigma_n) = ff.
\]

On applying \( \phi \) to these Boolean conditions we get

\[
\phi(W_A(b)(\sigma_i)) = tt \quad 1 \leq i \leq n - 1 \text{ and } \phi(W_A(b)(\sigma_n)) = ff
\]

since \( \phi(tt) = tt \) and \( \phi(ff) = ff \). By Lemma 14.4.3, we have

\[
W_B(b)(\hat{\phi}(\sigma_i)) = tt \quad 1 \leq i \leq n - 1 \text{ and } W_B(b)(\hat{\phi}(\sigma_n)) = ff
\]

Comparing with the sequence

\[
\tau_0, \tau_1, \ldots, \tau_n
\]

of iterates of \( S_0 \) on \( \tau = \hat{\phi}(\sigma) \) over \( B \), by the Claim, we get that

\[
W_B(b)(\tau_i) = tt \quad 1 \leq i \leq n - 1 \text{ and } W_B(b)(\tau_n) = ff
\]

and

\[
M_B^\omega(S)(\hat{\phi}(\sigma)) \simeq \tau_n \simeq \hat{\phi}(\sigma) \simeq \hat{\phi}(M_A^\omega(S)(\sigma)).
\]

This concludes the termination case.
Non-termination case

Suppose $M_A^{s_0}(S) \uparrow$. Then the sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$$

of iterates of $S_0$ on $\sigma$ over $A$ is infinite and

$$W_A(b)(\sigma_i) = tt \quad \text{for all } i \geq 0.$$  

Once again, applying $\phi$ and Lemma 14.4.3, we get

$$W_B(b)(\phi(\sigma_i)) = tt \quad \text{for all } i \geq 0.$$  

Comparing with the sequence

$$\tau_0, \tau_1, \ldots, \tau_n, \ldots$$

of iterates of $S_0$ on $\tau = \hat{\phi}(\sigma)$ over $B$, using the Claim, gives

$$W_B(b)(\tau_i) = tt \quad \text{for all } i \geq 0$$

and

$$M_B^{s_0}(S)(\hat{\phi}(\sigma)) \uparrow.$$  

Hence,

$$\phi(M_A^{s_0}(S)(\sigma)) \simeq M_B^{s_0}(S)(\hat{\phi}(\sigma)).$$

\[\square\]

14.4.4 Isomorphism Invariance and Program Equivalence

There are several conceptions of when two programs may be considered equivalent. Normally, they involve performing a common task, like computing a function.

**Definition** Let $S, S' \in \text{Comm}(\Sigma)$ and $U, V \subseteq \text{Var}$. We say that $S$ and $S'$ are equivalent on $\text{State}(A)$, with respect to input variables $U$ and output variables $V$, if for all $\sigma \in \text{State}(A)$ and for all $x \in U$, $\sigma(x) = \sigma'(x)$ implies

$$M_A^{s_0}(S)(\sigma)(y) \simeq M_A^{s_0}(S')(\sigma)(y)$$

for all $y \in V$. We write $S \equiv_A S'$ mod $U, V$.

For example, the Euclidean program $E$ for $\text{gcd}$,

$$E \equiv E \mod \{x, y\}, \{y\}.$$  

In general, $S$ and $S'$ in $\text{Comm}(\Sigma)$ may be equivalent on some $\Sigma$-algebra $A$ but they need not be equivalent on another $\Sigma$-algebra $B$.

**Lemma** Suppose $A$ is isomorphic to $B$. Then $S \equiv_A S'$ mod $U, V$ if, and only if, $S \equiv_B S'$ mod $U, V$.  

Proof. Let \( \phi : A \rightarrow B \) be an isomorphism, and
\[
\hat{\phi} : \text{State}(A) \rightarrow \text{State}(B)
\]
be the corresponding bijection between state spaces. Suppose \( S \equiv_A S' \mod U, V \). For any \( \tau, \tau' \in \text{State}(B) \) there are unique \( \sigma, \sigma' \in \text{State}(A) \) such that
\[
\hat{\phi}(\sigma) = \tau \quad \text{and} \quad \hat{\phi}(\sigma') = \tau'
\]
If
\[
\tau \simeq \tau' \mod U
\]
then
\[
\hat{\phi}(\sigma) \simeq \hat{\phi}(\sigma') \mod U
\]
and
\[
\sigma \simeq \sigma' \mod U
\]
since \( \phi \) is injective. Now, since \( S \equiv_A S' \mod U, V \),
\[
M_A(S)(\sigma) \simeq M_A(S')(\sigma) \mod V.
\]
Applying \( \hat{\phi} \) to both sides, we have
\[
\hat{\phi}(M_A(S)(\sigma)) \simeq \hat{\phi}(M_A(S')(\sigma)) \mod V.
\]
By the Isomorphism Invariance Theorem,
\[
M_B(S)(\hat{\phi}(\sigma)) \simeq M_B(S')(\hat{\phi}(\sigma)) \mod V
\]
and so
\[
M_B(S)(\tau) \simeq M_B(S')(\tau) \mod V.
\]
Since \( \tau \) was chosen arbitrarily, the last equation means that \( S \equiv_B S' \mod U, V \).

Definition Let \( P \) be a property of \( \text{Comm}(\Sigma) \), i.e.,
\[
P \subseteq \text{Comm}(\Sigma).
\]
Let \( U, V \subseteq \text{Var} \). We say that \( P \) is \( A \)-semantic with respect to \( U, V \), if
\[
P(S) \text{ holds and } S \equiv_A S' \mod U, V \text{ implies } P(S') \text{ holds.}
\]

Lemma If \( P \) is \( A \)-semantic with respect to \( U, V \) and \( A \) is isomorphic with \( B \) then \( P \) is \( B \)-semantic with respect to \( U, V \).

Proof. If \( P \) is \( A \)-semantic with respect to \( U, V \), then \( P(S) \) holds and \( S \equiv_A S' \mod U, V \) implies that \( P(S') \) holds. As \( A \cong B \), from Lemma 14.4.4, \( S \equiv_A S' \mod U, V \) if, and only if, \( S \equiv_B S' \mod U, V \). So we have that \( P(S) \) holds and \( S \equiv_B S' \) implies that \( P(S') \) holds. Thus, \( P \) is \( B \)-semantic if \( P \) is \( A \) semantic and \( A \cong B \).

14.4.5 Review of Terms

To appear.
14.4. INvariance OF Semantics 515

Exercises for Chapter 14

1. Prove structural induction on signatures implies induction on natural numbers, and hence complete the proof of Theorem 12.2.4.

2. Prove that if a variable $x$ does not appear in the left-hand side of an assignment in $S$ then:

$$M_A^o(S)(\sigma) \downarrow \Rightarrow M_A^o(S)(\sigma)(x) = \sigma(x).$$

3. Prove that if $x$ occurs only on the left hand side of a single assignment $x:=e$ in $S$ and all variables $y \in \text{var}(e)$ appear nowhere on the left hand side of an assignment in $S$ then:

$$V_A(e)(\sigma) = M_A^o(S)(\sigma)(x)$$

if $M_A^o(S)(\sigma)$ terminates.

4. Use the Local Computation Theorem to prove that the function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = n + 1$$

cannot be computed by any while program over the algebra

$$A = (\mathbb{N}, \mathbb{B}; 0; n \cdot 1, =, _{=}, _{\mathbb{B}}).$$

5. Consider the algebra

$$A = (\mathbb{N}, \mathbb{B}; 0, 1; 2n, =_{\mathbb{N}}, =_{\mathbb{B}}).$$

Which of the following functions $f : \mathbb{N} \to \mathbb{N}$ are, or are not, while computable on $A$? Justify your answers.

a. $f(x) = 3$;
b. $f(x) = 3x$;
c. $f(x) = 4x^2$;
d. $f(x) = 6x^3$;
e. $f(x) = 2^n x^n$; and
f. $f(n, x) = 2^n x^n$.

6. Evaluate the program $E$

a. over the decimal implementation $\mathbb{N}_{\text{dec}}$ on the start state $\sigma_0$ with

$$\sigma(x) = 10 \text{ and } \sigma(y) = 5;$$
b. over the binary implementation $\mathbb{N}_{\text{bin}}$ on the start state $\sigma_0$ with

$$\sigma(x) = 1010 \text{ and } \sigma(y) = 101.$$

7. Show that

$$\hat{\phi}(\sigma\{a/x\}) = \hat{\phi}(\sigma\{\phi(a)/x\})$$

holds for any state $\sigma : V ar \to A$, any program variable $x \in V ar$ and any value $a \in A$. 
8. Write out the trace of the execution of the program $P \in \text{Comm}(\Sigma)$:

\[
\begin{align*}
&x := 1; \\
y &:= x + 1; \\
&\text{if } x = y \text{ then} \\
&\quad \text{skip} \\
&\text{else} \\
&\quad y := x + y \\
&\text{fi}
\end{align*}
\]

on an initial state $\sigma_0$

a. over a decimal implementation $\mathbb{N}_{\text{dec}}$ of the natural numbers where $\sigma_0(x) = 4$ and $\sigma_0(x) = 3$; and

b. over a binary implementation $\mathbb{N}_{\text{bin}}$ of the natural numbers where $\sigma_0(x) = 100$ and $\sigma_0(x) = 11$.

9. Does the Isomorphism Invariance Theorem 14.4.3 still hold if it is assumed that $\phi : A \to B$ is a $\Sigma$-homomorphism.

10. Does the Isomorphism Invariance Theorem 14.4.3 hold if we add the following constructs to the \textbf{while} language:

a. concurrent assignments;

b. \textbf{case} statements;

c. \textbf{repeat-until} statements; and

d. \textbf{for} statements.
14.4. INVARINACE OF SEMANTICS

Assignment for Chapter 14

A: Performance Measures

The aim of this assignment is to define a method for measuring or estimating the cost of a computation by a while program. Let \( N^+ = \{1, 2, \ldots \} \). We want to define a function

\[
\lambda : \text{Comm}(\Sigma) \rightarrow (\text{State}(A) \rightarrow N^+)
\]

such that, for \( S \in \text{Comm}(\Sigma) \),

\[
\lambda(S) : \text{State}(A) \rightarrow N^+
\]

computes the resources needed for the computation, i.e.,

\[
\lambda(S)(\sigma) = \text{the cost of executing } S \text{ in state } \sigma.
\]

The costs could measure the time or space used in a computation. The function \( \lambda(S) \) may be a partial function and we require for any \( S \) and \( \sigma \)

\[
\lambda(S)(\sigma) \downarrow \quad \text{if, and only if, } \quad M^i_{\lambda}(S)(\sigma) \downarrow.
\]

The strategy for measuring the costs of a computation follows that for defining semantics. We must determine the costs associated with data and states; operations and tests on states; and control and sequencing of actions in commands.

Performance of Data

We begin by modelling the costs of the basic operations on data. Let \( A \) be a many sorted algebra of signature \( \Sigma \). To measure the cost of using the constants and operations of \( A \), we define the concept of a performance measure \( p \) which consists of the following:

(i) For each constant symbol

\[
c : \rightarrow \ s
\]

of \( \Sigma \) naming an element

\[
c^A : \rightarrow A_s
\]

of \( A \), the performance measure \( \rho \) assumes a fixed charge of

\[
c^{A, \rho} : \rightarrow N^+.
\]

(ii) For each function symbol

\[
f : w \rightarrow s
\]

of \( \Sigma \) naming an operation

\[
f^A : A^w \rightarrow A_s
\]

of \( A \), the performance measure \( \rho \) assumes a cost function

\[
f^{A, \rho} : A^w \rightarrow N^+
\]
to compute the number of units charged. Specifically, the cost for any argument \((a_1, \ldots, a_n) \in A^n\) to compute
\[
f^A(a_1, \ldots, a_n) \in A_s
\]
is
\[
f^{A,p}(a_1, \ldots, a_n) \in \mathbb{N}^+
\]
units.

An important example of a performance measure \(p\) for any algebra \(A\) is called \(A\)-time in which data and operations have unit costs. More precisely, this is defined by:

- \(c^{A,p} = 1\) for each constant \(c^A\); and
- \(f^{A,p}(a_1, \ldots, a_n) = 1\) for each operation \(f^A\) on each argument \((a_1, \ldots, a_n)\).

Here the data type \(A\) is considered without reference to any implementation: the operations of \(A\) are seen as atomic and indivisible and therefore define units for evaluating program performance based on \(A\). This performance measure is appropriate whenever we regard the algebra \(A\) as modelling an independent and autonomous level of abstraction for data.

1. Define costs for the operations \(Succ\), \(+\) and \(\times\) and the tests of equality and order on the natural numbers.

**Expressions**

2. Use structural induction on expressions to define the performance for computing expressions:
\[
\lambda_{Exp}^{A,p} : Exp(\Sigma) \rightarrow (State(A) \rightarrow \mathbb{N}^+)
\]
such that for \(e \in Exp(\Sigma)\),
\[
\lambda_{Exp}^{A,p}(e) : State(A) \rightarrow \mathbb{N}^+
\]
computes the resources needed for evaluating an expression \(e\) in state \(\sigma\), i.e.,
\[
\lambda_{Exp}^{A,p}(e)(\sigma) = \text{the cost of evaluating expression } e \text{ in state } \sigma \text{ according to performance measure } p.
\]
Give two formulae to cover parallel and sequential evaluation of subexpressions.

**Tests**

3. Use structural induction on Boolean expressions to define the performance measure for tests
\[
\lambda_{BExp}^{A,p} : BExp(\Sigma) \rightarrow (State(A) \rightarrow \mathbb{N}^+)
\]
such that for \(b \in BExp(\Sigma)\),
\[
\lambda_{BExp}^{A,p}(b) : State(A) \rightarrow \mathbb{N}^+
\]
computes the resources needed for evaluating the Boolean expression \(b\) in state \(\sigma\), i.e.,
\[
\lambda_{BExp}^{A,p}(b)(\sigma) = \text{the cost of evaluating the Boolean expression } b \text{ in state } \sigma \text{ as measured by performance measure } p.
\]
Performance of Programs

4. Use structural induction on commands to define the performance function for commands:

\[ \lambda^{Ap} : Comm(\Sigma) \to (State(A) \to \mathbb{N}^+) \]

such that for \( S \in Comm(\Sigma) \),

\[ \lambda^{Ap}(S) : State(A) \to \mathbb{N}^+ \]

computes the resources needed for evaluating the command \( S \) in state \( \sigma \), so

\[ \lambda^{Ap}(S)(\sigma) = \text{the cost of evaluating the command } S \text{ on state } \sigma \text{ as measured by performance measure } p. \]

(Note this involves the definition of the input-output semantics function \( M_{io}^A \).)

5. Let \( Var(S) \subseteq V \) be the set of variables appearing in the command \( S \). Prove that for all states \( \sigma \) and \( \sigma' \) with \( \sigma \approx \sigma' \mod Var(S) \):

\[ \lambda^{Ap}(S)(\sigma) = \lambda^{Ap}(S)(\sigma'). \]

6. Prove that if \( y \notin Var(S) \) then for all states \( \sigma \) and \( \sigma' \) with \( \sigma \approx \sigma' \mod Var(S) \setminus \{y\} \):

\[ \lambda^{Ap}(S)(\sigma) = \lambda^{Ap}(S)(\sigma'). \]

7. Prove that if a variable \( x \) does not occur in the right-hand side of an expression, or in a boolean expression then for all states \( \sigma \) and \( \sigma' \) with \( \sigma \approx \sigma' \mod Var(S) \setminus \{x\} \):

\[ \lambda^{Ap}(S)(\sigma) = \lambda^{Ap}(S)(\sigma'). \]

Extensions

8. Define \( \lambda(S) \) for a program that involves

   a. concurrent assignments;
   b. case statements;
   c. repeat-until statements; and
   d. for statements.
Chapter 15
Operational Semantics

Incomplete Draft

The input-output semantics of Chapter 13 gives an abstract program behaviour by defining the output state of a computation as the semantics of a program. In this chapter, we extend the input-output semantics to give a new semantics that defines the operation of the program at every step in making computations.

In this way, we can see how a program behaves over time as it executes, just as we would with a debugger or program animator. So now we will have the semantics of a program giving a sequence of states instead of a single output state. Thus, if we execute a program $P$ on an initial state $\sigma_0$, we will either get a finite sequence

$$\sigma_0, \sigma_1, \ldots, \sigma_t$$

of states if it terminates, or else an infinite sequence

$$\sigma_0, \sigma_1, \ldots$$

of states if it does not.

These state sequences represent execution traces. The state $\sigma_t$ is the state of the computation produced by a program at some time cycle $t$. We can recover the input-output behaviour of a program by returning the last state (if it exists) of the computation of a finite sequence.

Thus, to define the operational semantics of a language, we need to define a function

$$\text{Comp} : \text{Prog}(\Sigma) \times \text{State}(A) \times \text{Time} \rightarrow \text{State}(A)$$

to enumerate, by time, the sequence of states produced by a computation, such that

$$\text{Comp}(P, \sigma, t)$$

gives the state produced by the computation of a program $P$ on a state $\sigma$ at time $t$. Thus, the state

$$\sigma_t = \text{Comp}(P, \sigma_0, t)$$

is that which is produced by executing the program $P$ on the initial state $\sigma_0$ for $t$ time cycles. An execution trace is the finite

$$\text{Comp}(P, \sigma_0, 0), \text{Comp}(P, \sigma_0, 1), \ldots, \text{Comp}(P, \sigma_0, t)$$

or infinite

$$\text{Comp}(P, \sigma_0, 0), \text{Comp}(P, \sigma_0, 1), \ldots, \text{Comp}(P, \sigma_0, t), \ldots$$

sequence of states produced by this function, depending on whether $P$ terminates on $\sigma_0$ or not.
Problem of Operational Semantics

To give a precise mathematical definition of Comp.

We shall consider three different methods of defining Comp: we define

(i) execution trace semantics in Section 15.1;
(ii) structural operational semantics in Section 15.2; and
(iii) algebraic operational semantics in an equational manner in Section 15.3.

DISCUSSION ON LEVEL OF ABSTRACTION... CURRENTLY HAVE TIME TO EXECUTE TESTS IN AOS BUT NONE OTHERS

15.1 Execution Trace Semantics

Input-output semantics concentrates on the output of execution. Operational semantics, on the other hand, considers the evolution of program execution. One way of viewing this evolution is via an execution trace. Given an initial state $\sigma_0$, the execution trace of program $S$ on the state $\sigma_0$ will either be finite:

$$\sigma_0, \sigma_1, \ldots, \sigma_n$$

or infinite:

$$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$$

depending on whether the program $S$ terminates or not, on starting from $\sigma_0$.

We shall define a function

$$M^\text{op}_A(S) : \text{State}(A) \rightarrow \text{Seq(State}(A))$$

to give the execution trace

$$M^\text{op}_A(S)(\sigma) \in \text{Seq(State}(A))$$

of a program $M^\text{op}_A(S)$ from a state $\sigma$. Thus, we can determine the state $\text{Comp}(P, \sigma, t)$ produced at time $t$ by executing program $P$ on state $\sigma$ by extracting the $t^{th}$ element of the sequence of states produced by $M^\text{op}_A$:

$$\text{Comp}(P, \sigma, t) = \text{nth}(M^\text{op}_A(P, \sigma), t)$$

Given an operational semantics for a program $S$, we can easily define its input-output semantics by extracting the final state (if it exists) from the execution trace:

$$M^\text{iop}_A(S)(\sigma_0) = \begin{cases} \sigma_n & \text{if } M^\text{op}_A(S)(\sigma_0) = \sigma_0, \sigma_1, \ldots, \sigma_n \\ \perp & \text{if } M^\text{op}_A(S)(\sigma_0) = \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots \end{cases}$$
15.1. EXECUTION TRACE SEMANTICS

15.1.1 Execution Traces

Given a model $\text{State}(A)$ of states storing values from some data type $A$, we define the set

$$\text{Seq}(\text{State}(A))$$

of execution traces to be all possible finite sequences of the form:

$$\sigma_0, \sigma_1, \ldots, \sigma_n$$

and all possible infinite sequences of the form:

$$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$$

Thus, the set

$$\text{Seq}(\text{State}(A)) = \{\sigma_0, \sigma_1, \ldots, \sigma_n \mid \sigma_i \in \text{State}(A)\} \cup \{\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots \mid \sigma_i \in \text{State}(A)\}$$

of execution traces is the set of all possible finite and infinite sequences of states.

Extracting Particular States

We define a projection function

$$\text{nth} : \text{Seq}(\text{State}(A)) \times \mathbb{N} \to \text{State}_\perp(A)$$

on state sequences such that $\text{nth}(\overline{\sigma}, i)$ extracts the $i^{th}$ element:

$$\text{nth}(\sigma_0, \sigma_1, \ldots, \sigma_n, i) = \begin{cases} \sigma_i & \text{if } i \leq n; \\ \perp & \text{otherwise.} \end{cases}$$

$$\text{nth}(\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots, i) = \sigma_i$$

We define a function

$$\text{Final} : \text{Seq}(\text{State}(A)) \to \text{State}_\perp(A)$$

so that $\text{Final}(\overline{\sigma})$ gives the final state of an execution trace $\overline{\sigma}$ if it exists:

$$\text{Final}(\sigma_0, \ldots, \sigma_n) = \sigma_n$$

and otherwise, we flag its non-existence with the element $\perp \in \text{State}_\perp(A)$:

$$\text{Final}(\sigma_0, \ldots, \sigma_n, \ldots) = \perp$$

Thus, we can define the relationship between input-output semantics and operational semantics:

$$M^x_{\perp}(S)(\sigma) = \text{Final}(M^x_{\perp}(S)(\sigma))$$
Concatenating Execution Traces

In order to manipulate execution traces, we introduce a concatenation operator

\[ \circ : \text{Seq}(\text{State}(A)) \times \text{Seq}(\text{State}(A)) \rightarrow \text{Seq}(\text{State}(A)) \]

on execution traces. The concatenation of finite execution traces

\[ \sigma_0, \ldots, \sigma_m \]

and

\[ \sigma'_0, \ldots, \sigma'_n \]

is, as we would expect, simply

\[ \sigma_0, \ldots, \sigma_m \circ \sigma'_0, \ldots, \sigma'_n = \sigma_0, \ldots, \sigma_m, \sigma'_0, \ldots, \sigma'_n. \]

We have to exercise a little care though when infinite execution traces are involved. If the first execution trace is finite:

\[ \sigma_0, \ldots, \sigma_m \]

but the second is infinite,

\[ \sigma'_0, \ldots, \sigma'_n, \ldots \]

the execution trace that results will be infinite:

\[ \sigma_0, \ldots, \sigma_m \circ \sigma'_0, \ldots, \sigma'_n, \ldots = \sigma_0, \ldots, \sigma_m, \sigma'_0, \ldots, \sigma'_n, \ldots \]

If the first execution trace is infinite:

\[ \sigma_0, \ldots, \sigma_m, \ldots \]

then the second execution trace will be immaterial. If the second trace is finite

\[ \sigma'_0, \ldots, \sigma'_n \]

the execution trace that results from concatenation is simply the first:

\[ \sigma_0, \ldots, \sigma_m, \ldots \circ \sigma'_0, \ldots, \sigma'_n = \sigma_0, \ldots, \sigma_m, \ldots \]

and if the second trace is infinite

\[ \sigma'_0, \ldots, \sigma'_n, \ldots \]

the execution trace that results from concatenation is similarly just the first:

\[ \sigma_0, \ldots, \sigma_m, \ldots \circ \sigma'_0, \ldots, \sigma'_n, \ldots = \sigma_0, \ldots, \sigma_m, \ldots \]

Thus, for any computation traces \( \overline{\sigma}, \overline{\sigma}' \in \text{Seq}(\text{State}(A)) \), we define the concatenation

\[ \overline{\sigma} \circ \overline{\sigma}' \]

of execution traces by:

\[ \overline{\sigma} \circ \overline{\sigma}' = \begin{cases} \overline{\sigma} & \text{if } \overline{\sigma} = \sigma_0, \ldots, \sigma_m, \ldots; \\ \sigma_0, \ldots, \sigma_m, \overline{\sigma}' & \text{if } \overline{\sigma} = \sigma_0, \ldots, \sigma_m. \end{cases} \]
15.1. EXECUTION TRACE SEMANTICS

Algebra of Execution Traces

| algebra | ExecutionTraces |
|-------------------------------------------|
| import | State(A), N |
| carriers | Seq(State(A)) = \{σ_0, σ_1, ..., σ_n \mid σ_i \in State(A)\} |
| \quad \cup \{σ_0, σ_1, ..., σ_n, ... \mid σ_i \in State(A)\} |
| constants |
| operations | nth : Seq(State(A)) × N → State(A) |
| \quad final : Seq(State(A)) → State(A) |
| \quad \circ : Seq(State(A)) × Seq(State(A)) → Seq(State(A)) |
| definitions | nth(σ_0, σ_1, ..., σ_n, i) = \begin{cases} σ_i & \text{if } 1 \leq n; \\ ⊥ & \text{otherwise.} \end{cases} |
| \quad nth(σ_0, σ_1, ..., σ_n, i) = σ_i |
| \quad Final(σ_0, ..., σ_n) = σ_n |
| \quad Final(σ_0, ..., σ_n, ...) = ⊥ |
| \quad σ \circ σ' = \begin{cases} σ' & \text{if } σ = σ_0, ..., σ_m, ...; \\ σ_0, ..., σ_m, σ' & \text{if } σ = σ_0, ..., σ_m. \end{cases} |

15.1.2 Operational Semantics of while programs

We define the operational semantics of while programs by induction on their syntax.

Base Case

As is the case for all the execution traces, the first state in the sequence is the initial state. For the atomic programs, the second, and final state is that given by the input-output semantics of Chapter 13:

Identity  Executing a skip statement on a state σ does not affect the state σ:

\[ M_A^{op}(\text{skip})(σ) = σ \circ σ \]

Assignment  Executing an assignment statement \( x := e \) on a state σ just updates the value stored in the variable x with the result \( V_A(e)(σ) \) of evaluating the expression e on the state σ:

\[ M_A^{op}(x := e)(σ) = σ \circ σ[V_A(e)(σ)/x]. \]
**Induction Step**

We suppose that the execution traces $M^p_A(S_0)$, $M^p_A(S_1)$ and $M^p_A(S_2)$ are defined for all states.

**Sequencing** We define the operational semantics

$$M^p_A(S_1; S_2)$$

of a sequenced statement $S_1; S_2$ on an initial state $\sigma$ by concatenating:

- the execution trace $M^p_A(S_1)(\sigma)$ produced by the execution of $S_1$ on the initial state; and
- the execution trace produced by executing $S_2$ on the final state $Final(M^p_A(S_1)(\sigma))$

(if it exists) produced by executing $S_1$ on the initial state $\sigma$.

Thus,

$$M^p_A(S_1; S_2)(\sigma) = M^p_A(S_1)(\sigma) \circ M^p_A(S_2)(Final(M^p_A(S_1)(\sigma)))$$

So, if $S_1$ terminates, it produces a finite sequence of states and the sequencing of $S_1$ and $S_2$ produces first the state sequence from executing $S_1$, then the state sequence from executing $S_2$. If $S_1$ does not terminate, then it produces an infinite sequence of states and we do not get to observe the behaviour of $S_2$.

**Conditional** We define the operational semantics

$$M^p_A(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma)$$

of a conditional statement $\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}$ on an initial state $\sigma$ by cases.

If the evaluation $W_A(b)(\sigma)$ of the test $b$ on the initial state $\sigma$ is true, then we define the execution trace $M^p_A(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma)$ by concatenating:

- the initial state $\sigma$; and
- the execution trace $M^p_A(S_1)(\sigma)$ produced by executing the then-statement $S_1$ on the initial state $\sigma$.

If, however, the evaluation $W_A(b)(\sigma)$ of the test $b$ on the initial state $\sigma$ is false, then we define the execution trace $M^p_A(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma)$ by concatenating:

- the initial state $\sigma$; and
- the execution trace $M^p_A(S_1)(\sigma)$ produced by executing the else-statement $S_2$ on the initial state.

Thus,

$$M^p_A(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma) = \begin{cases} 
\sigma \circ M^p_A(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\
\sigma \circ M^p_A(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = ff. 
\end{cases}$$
15.1. EXECUTION TRACE SEMANTICS

Iteration We define the operational semantics

\[ M_A^{op}(\text{while } b \text{ do } S_0 \text{ od})(\sigma) \]

of a while statement while \( b \) do \( S_0 \) od on an initial state \( \sigma \) in the same manner as we defined
the input-output semantics of while statements in Chapter 13, but we retain all the intermediate states.

We concatenate:

(i) the initial state \( \sigma_0 \); and

(ii) the execution trace

\[ M_A^{op}(S_0)(\sigma_0) \]

produced by executing the body \( S_0 \) on the state \( \sigma_0 \) if the evaluation

\[ W_A(b)(\sigma_0) \]

of the test \( b \) on the state \( \sigma_0 \) is true;

(iii) the execution trace

\[ M_A^{op}(S_0)(M_A^{op}(S_0)(\text{Final}(\sigma_0))) \]

produced by executing the body \( S_0 \) on the final state

\[ \text{Final}(M_A^{op}(S_0)(\sigma_0)) \]

(if it exists) that results from stage (ii) if the evaluation of the test in stage (iii) is true and the evaluation

\[ W_A(b)(\text{Final}(M_A^{op}(S_0)(\sigma_0))) \]

of the test \( b \) on the final state \( \text{Final}(M_A^{op}(S_0)(\sigma_0)) \) (if it exists) that results from stage

(ii) on the state \( \sigma_0 \) is also true;

(iv) and so on whilst the evaluation of the test \( b \) at each stage remains true.

Thus,

\[ M_A^{op}(\text{while } b \text{ do } S_0 \text{ od})(\sigma_0) = \begin{cases} \sigma_0 \circ \sigma_0 & \text{if } W_A(b)(\sigma_0) = \text{ff}; \\ \overline{\sigma_0} \circ \cdots \circ \overline{\sigma_n} & \text{if } \overline{\sigma_0} = \sigma_0, \\ \exists n > 1, \forall i : 0 \leq i \leq n - 1 & [W_A(b)(\text{Final}(\sigma_i)) = \text{tt}, M_A^{op}(S_0)(\text{Final}(\sigma_i)) = \sigma_{i+1}] \\ \text{and } W_A(b)(\text{Final}(\overline{\sigma_i})) = \text{ff}; \\ \overline{\sigma_0} \circ \cdots \circ \overline{\sigma_n} \circ \cdots & \text{if } \overline{\sigma_0} = \sigma_0, \sigma_0 \text{ and} \\ \forall i \geq 0 & [W_A(b)(\text{Final}(\sigma_i)) = \text{tt}, M_A^{op}(S_0)(\text{Final}(\sigma_i)) = \sigma_{i+1}]. \end{cases} \]

Note that we can get an infinite sequence of states from executing a while statement in one of two ways. The first possibility is that the execution of the body \( S_0 \) of the loop does not terminate at some point in the execution. The second possibility is that each iteration of the body \( S_0 \) terminates, but the loop itself does not terminate because the test at each stage is true. In the first case we do not observe any subsequent behaviour of the loop after the first point that the infinite body execution occurs.
Summary  This gives us the following operational semantics for while programs:

\[
M_A^{sp}(\text{skip})(\sigma) = \sigma \circ \sigma \\
M_A^{sp}(x := e)(\sigma) = \sigma \circ \sigma \{V_A(e)(\sigma) / x\} \\
M_A^{sp}(S_1 ; S_2)(\sigma) = M_A^{sp}(S_1)(\sigma) \circ M_A^{sp}(S_2)(\text{Final}(M_A^{sp}(S_1)(\sigma))) \\
M_A^{sp}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ }\text{fi})(\sigma) = \begin{cases} \\ 
\sigma \circ M_A^{sp}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\
\sigma \circ M_A^{sp}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = ff.
\end{cases}
\]

\[
M_A^{sp}(\text{while } b \text{ do } S_0 \text{ od})(\sigma_1) = \begin{cases} \\ 
\sigma_0 \circ \sigma_0 & \text{if } W_A(b)(\sigma_0) = ff; \\
\sigma_0 \circ \cdots \circ \sigma_n & \text{if } \sigma_0 = \sigma_0, \\
\prod_{i > 1, \forall i : 0 \leq i \leq n - 1} [W_A(b)(\text{Final}(\sigma_i)) = tt, M_A^{sp}(S_0)(\text{Final}(\sigma_i)) = \sigma_{i+1}] & \text{and } W_A(b)(\text{Final}(\sigma_n)) = ff; \\
\sigma_0 \circ \cdots \circ \sigma_n \circ \cdots & \text{if } \sigma_0 = \sigma_0, \sigma_0 \text{ and} \\
\forall i \geq 0 & \text{and} \\
[W_A(b)(\text{Final}(\sigma_i)) = tt, M_A^{sp}(S_0)(\text{Final}(\sigma_i)) = \sigma_{i+1}].
\end{cases}
\]

Example  Consider the program \(E\) from Section 14.4.1 to compute Euclid’s algorithm. Suppose we execute \(E\) on the state \(\sigma_0\) where \(\sigma_0(x) = 45\) and \(\sigma_0(y) = 12\). Then the execution trace for \(E\) on \(\sigma_0\) is constructed as follows.

\[
M_A^{sp}(z := x \text{ mod } y \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \text{ mod } y \text{ od})(\sigma_0) \\
= M_A^{sp}(z := x \text{ mod } y)(\sigma_0) \\
\circ M_A^{sp}(\text{while } z \neq 0 \text{ do } x := y; y := z; z := x \text{ mod } y \text{ od})(\text{Final}(M_A^{sp}(z := x \text{ mod } y(\sigma_0))))
\]

by the definition of \(M_A^{sp}\) on sequenced statements.

Let us consider the execution trace \(\sigma_0\) of the assignment:

\[
\sigma_0 = M_A^{sp}(z := x \text{ mod } y)(\sigma_0) \\
= \sigma_0, \sigma_0[V_A(x \text{ mod } y)(\sigma) / z]
\]

by the definition of \(M_A^{sp}\) on assignments

\[
= \sigma_0, \sigma_0[9/z]
\]

by the definition of \(V_A\).

Now let us consider the execution trace of the while loop. Let

\[
\sigma_1 = \text{Final}(\sigma_0) \\
= \sigma_0[9/z].
\]
Then, by the definition of $M^op_A$ on iteration,

\[ M^op_A(\textbf{while } z \neq 0 \textbf{ do } x:=y; y:=z; z:=x \textbf{ mod } y \textbf{ od})(\sigma_1) = \sigma_1 \circ \sigma_2 \circ \sigma_3 \]

where the execution traces

\[
\begin{align*}
\sigma_1 &= \sigma_1 \\
\sigma_2 &= M^op_A(x:=y; y:=z; z:=x \textbf{ mod } y)(\text{Final}(\sigma_1)) \\
\sigma_3 &= M^op_A(x:=y; y:=z; z:=x \textbf{ mod } y)(\text{Final}(\sigma_2))
\end{align*}
\]

The execution trace $\sigma_2$ simplifies to:

\[ M^op_A(x:=y; y:=z; z:=x \textbf{ mod } y)(\text{Final}(\sigma_1)) = M^op_A(x:=y; y:=z; z:=x \textbf{ mod } y)(\sigma_1) \]

by definition of $\sigma_1$ and Final

\[
\begin{align*}
&= M^op_A(x:=y)(\sigma_1) \circ M^op_A(y:=z; z:=x \textbf{ mod } y)(\text{Final}(M^op_A(x:=y)(\sigma_1))) \\
&= M^op_A(x:=y)(\sigma_1) \circ (M^op_A(y:=z)(\text{Final}(M^op_A(x:=y)))) \\
&\circ M^op_A(z:=x \textbf{ mod } y)(\text{Final}(M^op_A(y:=z)(\text{Final}(M^op_A(x:=y)(\sigma_1))))))
\end{align*}
\]

by definition of $M^op_A$ on sequenced statements.

Taking the semantics of $x:=y$:

\[ M^op_A(x:=y)(\sigma_1) = \sigma_1, \sigma_1[V_A(y)(\sigma_1)/x] \]

by the definition of $M^op_A$ on assignments

\[ M^op_A(x:=y)(\sigma_1) = \sigma_1, \sigma_1[12/x] \]

by the definition of $V_A$.

The semantics of $y:=z$:

\[ M^op_A(y:=z)(\text{Final}(M^op_A(x:=y))) = M^op_A(y:=z)(\text{Final}(\sigma_1, \sigma_1[12/x])) \]

\[ = M^op_A(y:=z)(\sigma_1[12/x]) \]

\[ = \sigma_1[12/x], \sigma_1[12/x][V_A(z)(\sigma_1[12/x])/y] \]

by the definition of $M^op_A$ on assignments

\[ = \sigma_1[12/x], \sigma_1[12/x][9/y] \]

by the definition of $V_A$.

The semantics of $z:=x \textbf{ mod } y$:

\[ M^op_A(z:=x \textbf{ mod } y)(\text{Final}(M^op_A(y:=z)(\text{Final}(M^op_A(x:=y)(\sigma_1)))))) \]

\[ = M^op_A(z:=x \textbf{ mod } y)(\text{Final}(\sigma_1[12/x], \sigma_1[12/x][9/y])) \]

\[ = M^op_A(z:=x \textbf{ mod } y)(\sigma_1[12/x][9/y]) \]

\[ = \sigma_1[12/x][9/y], \sigma_1[12/x][9/y][V_A(x \textbf{ mod } y)(\sigma_1[12/x][9/y])/z] \]
by the definition of $M_A^{op}$ on assignments
\[ = \sigma_1[12/x][9/y], \sigma_1[12/x][9/y][3/z] \]
by the definition of $V_A$.
Thus,
\[ \bar{\sigma}_2 = \sigma_1[12/x], \sigma_1[12/x][9/y], \sigma_1[12/x][9/y][3/z] \]
\[ = \sigma_0[9/z][12/x], \sigma_0[9/z][12/x][9/y], \sigma_0[9/z][12/x][9/y][3/z] \]
\[ = \sigma_0[9/z][12/x], \sigma_0[9/z][12/x][9/y], \sigma_0[12/x][9/y][3/z] \]
by definition of $\sigma_1$ and substitution on states; and similarly,
\[ \bar{\sigma}_3 = \sigma_2[9/x], \sigma_2[9/x][3/y], \sigma_2[9/x][3/y][0/z] \]
\[ = \sigma_0[9/x][9/y][3/z], \sigma_0[9/x][3/y][3/z], \sigma_0[9/x][3/y][0/z] \]
Note that the test $z \neq 0$ evaluates to:
\[ W_A(z \neq 0)(Final(\bar{\sigma}_0)) = W_A(z \neq 0)(Final(\sigma_1, \sigma_1)) \]
\[ = W_A(z \neq 0)(\sigma_0[9/z]) \]
\[ = tt \]
on the state $Final(\bar{\sigma}_1)$,
\[ W_A(z \neq 0)(Final(\bar{\sigma}_2)) \]
\[ = W_A(z \neq 0)(Final(\sigma_0[9/z][12/x], \sigma_0[9/z][12/x][9/y], \sigma_0[12/x][9/y][3/z])) \]
\[ = W_A(z \neq 0)(\sigma_0[12/x][9/y][3/z]) \]
\[ = tt \]
on the state $Final(\bar{\sigma}_2)$ and
\[ W_A(z \neq 0)(Final(\bar{\sigma}_3)) \]
\[ = W_A(z \neq 0)(Final(\sigma_0[9/x][9/y][3/z], \sigma_0[9/x][3/y][3/z], \sigma_0[9/x][3/y][0/z])) \]
\[ = W_A(z \neq 0)(\sigma_0[9/x][3/y][0/z]) \]
\[ = ff \]
on the state $Final(\bar{\sigma}_3)$.
Thus, the execution trace for the whole algorithm is:
\[ M_A^{op}(z:=x \mod y; \text{while } z \neq 0 \text{ do } x:=y; y:=z; z:=x \mod y \text{ od})(\sigma_0) \]
\[ = \sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_3 \]
\[ = \sigma_0, \sigma_0[9/z], \sigma_0[9/z], \sigma_0[9/z][12/x], \sigma_0[9/z][12/x][9/y], \sigma_0[12/x][9/y][3/z] \]
\[ \sigma_0[9/x][9/y][3/z], \sigma_0[9/x][3/y][3/z], \sigma_0[9/x][3/y][0/z] \]
which we illustrate in Figure 15.1.
15.2. STRUCTURAL OPERATIONAL SEMANTICS

<table>
<thead>
<tr>
<th>Trace</th>
<th>Tests</th>
<th>Comment</th>
<th>Execute</th>
<th>State Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Evaluate</td>
<td>Result</td>
<td></td>
<td>x  y  z</td>
</tr>
<tr>
<td>σ₀</td>
<td></td>
<td>Initial State</td>
<td></td>
<td>45 12 ?</td>
</tr>
<tr>
<td>σ₁</td>
<td></td>
<td>z ≠ 0 Succeeds</td>
<td>x := x mod y</td>
<td>45 12 9</td>
</tr>
<tr>
<td>σ₂</td>
<td></td>
<td>Enter loop</td>
<td></td>
<td>45 12 9</td>
</tr>
<tr>
<td>σ₃</td>
<td></td>
<td>z ≠ 0 Succeeds</td>
<td>x := y</td>
<td>12 12 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Re-enter loop</td>
<td>y := z</td>
<td>12 9 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exit loop</td>
<td>z := x mod y</td>
<td>12 9 3</td>
</tr>
</tbody>
</table>

Figure 15.1: Execution trace for Euclid's algorithm on initial state σ₀ where σ₀(x) = 45 and σ₀(y) = 12.

15.2 Structural Operational Semantics

To appear: general introduction

Thus, we can determine the state Comp(P, σ, t) produced at time t by executing program P on state σ by extracting the tᵗʰ element of the sequence of states produced by \( M_A^{P} \):

\[
Comp(P, \sigma, t) = nth(M_A^{SOS}(P, \sigma), t)
\]

15.2.1 General approach

A configuration \( \gamma \in Configuration \) represents a program execution in progress, or one that has terminated.

\[ \rightarrow_{SOS} \subseteq Configuration \times Configuration \]

Two possibilities for configuration \( \gamma \).

\[
(S, \sigma) \rightarrow_{SOS} \gamma
\]

The first stage in executing the program \( S \) on the state \( \sigma \) produces the state \( \sigma' \), and the program \( S' \) remains to be executed.

\[
(S, \sigma) \rightarrow_{SOS} \sigma'
\]
The program $S$ terminates on the state $\sigma$ in a single stage, producing the state $\sigma'$.

A derivation sequence of a program $S$ on an initial state $\sigma_0$ is a finite

$$\gamma_0, \gamma_1, \ldots, \gamma_n$$

or infinite

$$\gamma_0, \gamma_1, \ldots, \gamma_n, \ldots$$

sequence of configurations, where the initial configuration $\gamma_0 = (S, \sigma_0)$, and each subsequent configuration $\gamma_{i+1}$ is derived from the previous configuration $\gamma_i$ by the transition relation $\xrightarrow{SOS}$.

In the case of an infinite derivation sequence, $\gamma_i \xrightarrow{SOS} \gamma_{i+1}$ for $i \geq 0$.

In the case of a finite derivation sequence, $\gamma_i \xrightarrow{SOS} \gamma_{i+1}$ for $0 \leq i \leq n$, and $\exists \gamma_{n+1} \in \text{Configuration}$ such that $\gamma_n \xrightarrow{SOS} \gamma_{n+1}$. This may be deliberate, because a program has been completely executed and the configuration is just a state $\gamma_n \in \text{State}$. However, it may also be because there are simply no rules to describe how a configuration may progress.

### 15.2.2 Structural Operational Semantics of while Programs

### (skip, $\sigma$) $\xrightarrow{SOS} \sigma$

### (x:=e, $\sigma$) $\xrightarrow{SOS} \sigma[V^A(e)(\sigma)/x]$

### (S_1, $\sigma$) $\xrightarrow{SOS} \sigma'$

### (S_1; S_2, $\sigma$) $\xrightarrow{SOS} (S_2, \sigma')$

### (S_1, $\sigma$) $\xrightarrow{SOS} (S_1, \sigma')$

### (S_1; S_2, $\sigma$) $\xrightarrow{SOS} (S_1; S_2, \sigma')$

### (if b then S_1 else S_2 fi, $\sigma$) $\xrightarrow{SOS}$ \begin{cases} (S_1, \sigma) & \text{if } W^A(b)(\sigma) = tt; \\ (S_2, \sigma) & \text{if } W^A(b)(\sigma) = ff. \end{cases}

### (while b do S_0 od, $\sigma$) $\xrightarrow{SOS}$ (if b then S_0; while b do S_0 od else skip od, $\sigma$)

### Equivalent Axiomatic Definition
15.2. STRUCTURAL OPERATIONAL SEMANTICS

\[ M_A^{SOS}(\text{skip}, \sigma) = \sigma \circ \sigma \]
\[ M_A^{SOS}(x := e, \sigma) = \sigma[V^A(e)(\sigma)/x] \]
\[ M_A^{SOS}(S_1; S_2, \sigma) = \begin{cases} 
\sigma \circ M_A^{SOS}(S_2, \sigma') & \text{if } M_A^{SOS}(S_1, \sigma) = \sigma \circ \sigma'; \\
\sigma \circ M_A^{SOS}(S_1; S_2, \sigma') & \text{if } M_A^{SOS}(S_1, \sigma) = \sigma \circ M_A^{SOS}(S_1', \sigma'). 
\end{cases} \]
\[ M_A^{SOS}((\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) = \begin{cases} 
M_A^{SOS}(S_1, \sigma) & \text{if } W^A(b)(\sigma) = \text{tt}; \\
M_A^{SOS}(S_2, \sigma) & \text{if } W^A(b)(\sigma) = \text{ff}. 
\end{cases} \]
\[ M_A^{SOS}((\text{while } b \text{ do } S_0 \text{ od}, \sigma) = M_A^{SOS}((\text{if } b \text{ then } S_0; \text{while } b \text{ do } S_0 \text{ od else skip}, \sigma) \]

**Example** Consider the program \( E \) from Section 14.4.1 to compute Euclid’s algorithm. Suppose we execute \( E \) on the state \( \sigma_0 \) where \( \sigma_0(x) = 45 \) and \( \sigma_0(y) = 12 \) using structural operational semantics.

The first statement we execute is the first of the sequence of statements that constitute \( E \). Furthermore, as this is an assignment statement \( z := x \mod y \), we can completely execute it in a single stage. Thus:

\[
\frac{(z := x \mod y, \sigma_0) \xrightarrow{\text{SOS}} \sigma_0[9/z]}{(z := x \mod y; S_{\text{while}}, \sigma_0) \xrightarrow{\text{SOS}} (S_{\text{while}}, \sigma_0[9/z])}
\]

So after one time cycle, we have executed the statement \( z := x \mod y \) on the initial state \( \sigma_0 \), producing the state

\[ \sigma_1 = \sigma_0[9/z] \]

where we have the values:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_1 )</td>
<td>45</td>
<td>12</td>
<td>9</td>
</tr>
</tbody>
</table>

We also have the program \( S_{\text{while}} \) still to execute on the state \( \sigma_1 \).

We now consider the execution of \( S_{\text{while}} \) on the state \( \sigma_1 \). Using the transition rules for structural operational semantics, we express the semantics of the while loop using that of a conditional statement.

\[
(S_{\text{while}}, \sigma_1) \xrightarrow{\text{SOS}} (\text{if } z \neq 0 \text{ then } S_{\text{body}}; S_{\text{while}} \text{ else skip od}, \sigma_1)
\]

So now using the semantics of conditional statements, as \( W^A(z \neq 0)(\sigma_1) = \text{tt} \), we get:

\[
(\text{if } z \neq 0 \text{ then } S_{\text{body}}; S_{\text{while}} \text{ else skip od}, \sigma_1) \xrightarrow{\text{SOS}} (S_{\text{body}}; S_{\text{while}}, \sigma_1)
\]

We now have a sequence formed from the body of the while loop, followed by the whole of the while loop. As the body consists of a sequence of statements, it will be the case that there exists some program \( S' \) and some state \( \sigma_2 \) such that

\[
(S_{\text{body}}, \sigma_1) \xrightarrow{\text{SOS}} (S', \sigma_2)
\]

\[
(S_{\text{body}}; S_{\text{while}}, \sigma_1) \xrightarrow{\text{SOS}} (S', S_{\text{while}}, \sigma_2)
\]
So, let us consider just $S_{\text{body}}$ to determine the state $\sigma_2$ from executing the first stage of $S'$. In so doing, we shall also determine the remaining program $S'$ of $S_{\text{body}}$ that we shall need to execute, in order to fully execute $S_{\text{body}}$:

$$
\frac{(x:=y, \sigma_1) \xrightarrow{SOS} \sigma_1[12/x]}{(S_{\text{body}}, \sigma_1) \xrightarrow{SOS} (y:=z; z:=x \mod y, \sigma_1[12/x])}
$$

Thus, we generate the state $\sigma_2 = \sigma_1[12/x]$ at step 1 of the execution of $S_{\text{body}}$, and we have the remaining program $y:=z; z:=x \mod y$ of $S_{\text{body}}$ to execute. Now, substituting back into our original problem, we have:

$$
\frac{(x:=y, \sigma_1) \xrightarrow{SOS} \sigma_1[12/x]}{(x:=y; y:=z; z:=x \mod y; S_{\text{while}}, \sigma_1) \xrightarrow{SOS} (y:=z; z:=x \mod y; S_{\text{while}}, \sigma_1[12/x])}
$$

I.e., we have the state $\sigma_2 = \sigma_1[12/x]$

where

\[
\begin{array}{ccc}
   x & y & z \\
\sigma_1 & 45 & 12 & 9 \\
\sigma_2 & 12 & 12 & 9 \\
\end{array}
\]

at step 2 of the execution of the whole program $E$, and we have the remaining program $y:=z; z:=x \mod y; S_{\text{while}}$ to execute.

It is a similar story with the configuration

$$(y:=z; z:=x \mod y; S_{\text{while}}, \sigma_2)$$

As we have a program formed by sequencing an assignment statement $y:=z$ with another statement $z:=x \mod y; S_{\text{while}}$, we have:

$$
\frac{(y:=z, \sigma_2) \xrightarrow{SOS} \sigma_2[9/y]}{(y:=z; z:=x \mod y; S_{\text{while}}, \sigma_2) \xrightarrow{SOS} (z:=x \mod y; S_{\text{while}}, \sigma_2[9/y])}
$$

This gives us our third execution state $\sigma_3 = \sigma_2[9/y]$

where

\[
\begin{array}{ccc}
   x & y & z \\
\sigma_1 & 45 & 12 & 9 \\
\sigma_2 & 12 & 12 & 9 \\
\sigma_3 & 12 & 9 & 9 \\
\end{array}
\]
and the remaining program
\[ z := x \text{ mod } y; S_{\text{while}} \]
to execute.

Again, we have a program formed by sequencing an assignment statement \( z := x \text{ mod } y; S_{\text{while}} \), with another statement \( S_{\text{while}} \):
\[
\begin{align*}
(z := x \text{ mod } y, \sigma_3) & \xrightarrow{SOS} \sigma_3[3/z] \\
(z := x \text{ mod } y; S_{\text{while}}, \sigma_3) & \xrightarrow{SOS} (S_{\text{while}}, \sigma_3[3/z])
\end{align*}
\]
This gives us our fourth execution state
\[ \sigma_4 = \sigma_3[3/z] \]
where
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
 & \( x \) & \( y \) \\
\hline
\( \sigma_1 \) & 45 & 12 \\
\hline
\( \sigma_2 \) & 12 & 12 \\
\hline
\( \sigma_3 \) & 12 & 9 \\
\hline
\( \sigma_4 \) & 12 & 9 \\
\hline
\end{tabular}
\end{center}

and the remaining program
\[ S_{\text{while}} \]
to execute.

Now, having executed the whole of the body \( S_{\text{body}} \) of the \textbf{while} loop \( S_{\text{while}} \) once, we are now at the stage of having \( S_{\text{while}} \) to deal with again.

\[ (S_{\text{while}}, \sigma_4) \xrightarrow{SOS} (\text{if } z \neq 0 \text{ then } S_{\text{body}}; S_{\text{while}} \text{ else skip od}, \sigma_4) \]

Again, the test is true:
\[ W^A(z \neq 0)(\sigma_4) = tt, \]
so we execute the body of the \textbf{while} loop once more. By the same reasoning as for the first execution of \( S_{\text{body}}; S_{\text{while}} \), we produce the configurations
\[
(y := z; z := x \text{ mod } y; S_{\text{while}}, \sigma_5), (z := x \text{ mod } y; S_{\text{while}}, \sigma_6)(S_{\text{while}}, \sigma_7)
\]
where the state produced at time 5 is:
\[ \sigma_5 = \sigma_4[9/x] \]
at time 6:
\[ \sigma_6 = \sigma_5[3/y] \]
and time 7:
\[ \sigma_7 = \sigma[0/z] \]
where

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ₁</td>
<td>45</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>σ₂</td>
<td>12</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>σ₃</td>
<td>12</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>σ₄</td>
<td>12</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>σ₅</td>
<td>9</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>σ₆</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>σ₇</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Now we are back at \( S_{\text{while}} \) again. This time though, the test is false:

\[
W^A(z \neq 0)(\sigma_7) = \mathit{ff}.
\]

Consequently, we exit from the \textbf{while} loop:

\[
(\text{if } z \neq 0 \text{ then } S_{\text{body}}; S_{\text{while}} \text{ else skip od}, \sigma_7) \xrightarrow{\text{SOS}} (\text{skip}, \sigma_7)
\]

And at this point, our program execution terminates:

\[
(\text{skip}, \sigma_7) \xrightarrow{\text{SOS}} \sigma_7
\]

Thus, the overall sequence of configuration states that we produce is shown in Figure 15.2.

<table>
<thead>
<tr>
<th>Time</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Remaining Program</td>
</tr>
<tr>
<td>0</td>
<td>( z := x \mod y; \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>1</td>
<td>\textbf{while} ( z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>2</td>
<td>( y := z; z := x \mod y; \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>3</td>
<td>( z := x \mod y; \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{while} ( z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>5</td>
<td>( y := z; z := x \mod y; \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>6</td>
<td>( z := x \mod y; \text{while } z \neq 0 \text{ do } x := y; y := z; z := x \mod y \text{ od} )</td>
</tr>
<tr>
<td>7</td>
<td>9 3 0</td>
</tr>
</tbody>
</table>

Figure 15.2: Derivation trace for Euclid’s algorithm using structural operational semantics on an initial state \( \sigma_0 \) where \( \sigma_0(x) = 45 \) and \( \sigma_0(y) = 12 \).

### 15.3 Algebraic Operational Semantics

We can describe the operational semantics of a language without recourse to infinite sequences of states by describing how each state in the execution trace is generated from the previous state.

We define the operational semantics function

\[
\text{Comp} : \text{Prog}(\Sigma) \times \text{State}(A) \times \text{Time} \rightarrow \text{State}(A)
\]
by considering how an execution trace can be constructed in a step-by-step manner. In particular, we suppose that programs are built from a set of atomic programs

$$AProg(\Sigma) \subseteq Prog(\Sigma)$$

and flow of control constructors that determine in what order the atomic programs are executed. We need a function

$$Act : AProg(\Sigma) \times State(A) \rightarrow State(A)$$

that determines the semantics $$Act(\alpha, \sigma)$$ of any atomic program $$\alpha$$ on any state $$\sigma$$.

Our aim is to produce a finite sequence

$$\alpha_0, \alpha_1, \ldots, \alpha_t$$

or infinite sequence

$$\alpha_0, \alpha_1, \ldots, \alpha_t, \ldots$$

of atomic programs, such that the execution

$$Act(\alpha_t, \sigma_t)$$

of the program $$\alpha_t$$ on the state $$\sigma_t$$ gives the next state $$\sigma_{t+1}$$ in the finite

\[
\begin{array}{c|c|c|c|c}
\sigma_0, & \sigma_1, & \sigma_2, & \ldots, & \sigma_t \\
\hline
 Comp(P, \sigma, 0), & Comp(P, \sigma, 1), & Comp(P, \sigma, 2), & \ldots, & Comp(P, \sigma, t) \\
\hline
 Act(\alpha_0, \sigma_0) & Act(\alpha_1, \sigma_1) & \ldots & Act(\alpha_{t-1}, \sigma_{t-1}) \\
\end{array}
\]

or infinite execution trace

We shall suppose that $$AProg(\Sigma)$$ contains the identity program $$\text{skip}$$, whose execution has no effect on the values stored in a state.

$$Act(\text{skip}, \sigma) = \sigma.$$

15.3.1 Producing Execution Traces

We define the function $$Comp$$ by induction on time.
**Time** $t = 0$

At time 0, we simply want to return the initial state of the computation. So, for any $P \in \text{Prog}(\Sigma)$ and $\sigma \in \text{State}(A)$, we define

$$\text{Comp}(P, \sigma, 0) = \sigma.$$  

**Time** $t = 1$

At time 1, we want to find the first atomic instruction

$$\text{First}(P, \sigma)$$

of the program $P$ on the state $\sigma$ and then execute it on the state $\sigma$,

$$\text{Act}(\text{First}(P, \sigma), \sigma).$$

So, for any $P \in \text{Prog}(\Sigma)$ and $\sigma \in \text{State}(A)$, we define

$$\text{Comp}(P, \sigma, 1) = \text{Act}(\text{First}(P, \sigma), \sigma).$$

**Induction Case**

Suppose we have an atomic program to execute. Then we know that this only takes one step of time to execute. Thus, at times 2, 3, \ldots, we can have no further work to do. We denote that we have finished executing a program with a distinguished state $\ast$, which we presume is contained in the set $\text{State}(A)$ of all possible computation states. (This state allows us to model a finite computation $\sigma_0, \sigma_1, \ldots, \sigma_n$ as an infinite sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_n, \ast, \ast, \ldots$) Thus, if we have an atomic program $\alpha \in \text{AProg}(\Sigma)$, we define that for any $\sigma$ and $t \geq 1$,

$$\text{Comp}(\alpha, \sigma, t + 1) = \ast.$$  

Otherwise, if we have a non-atomic program $P \in \text{Prog}(\Sigma)$ with $P \notin \text{AProg}(\Sigma)$, we shall split the computation into two stages. In the first step of time, we shall produce the state

$$\text{Comp}(P, \sigma, 1)$$

that results at time 1. We shall use this state as the new initial state to execute the remaining instructions

$$\text{Rest}(P, \sigma)$$

of the program in the remaining $t$ cycles of time. Thus, for any non-atomic program $P$, $\sigma$ and $t \geq 1$, we define

$$\text{Comp}(P, \sigma, t + 1) = \text{Comp}(\text{Rest}(P, \sigma), \text{Comp}(P, \sigma, 1), t).$$  

To summarise, we define the function $\text{Comp}$ for any $P \in \text{Prog}(\Sigma)$, $\sigma \in \text{State}(A)$ and $t \in \text{Time}$ by:
\[ \begin{align*}
Comp(P, \sigma, 0) &= \sigma \\
Comp(P, \sigma, 1) &= \text{Act(First}(P, \sigma))(\sigma) \\
\forall t \geq 1 : \ Comp(P, \sigma, t + 1) &= \begin{cases} 
* & \text{if } P \text{ is atomic;} \\
\text{Comp(Rest}(P, \sigma), \text{Comp}(P, \sigma, 1), t) & \text{if } P \text{ is not atomic.}
\end{cases}
\end{align*} \]

### 15.3.2 Deconstructing Syntax

As we have seen, the functions \( \text{First} \) and \( \text{Rest} \) play a crucial part in the definition of the function \( \text{Comp} \). We use these functions to split a program into its constituent parts.

Thus, on an atomic program \( \alpha \in AProg(\Sigma) \subseteq Prog(\Sigma) \),

\[
\text{First}(\alpha, \sigma) = \alpha \\
\text{Rest}(\alpha, \sigma) = \text{skip}.
\]

On compound programs, we determine the first atomic command \( \text{First}(P, \sigma) \) of the program \( P \) that we execute to simulate the execution of the program \( P \) on the state \( \sigma \) over a period of one cycle of time. We use this in conjunction with the remaining instructions \( \text{Rest}(P, \sigma) \) of the program \( P \) that we have to consider to determine any future behaviour of the program \( P \) after the first cycle of time.

### 15.3.3 Algebraic Operational Semantics of while programs

![Diagram](image-url)

**Figure 15.4**: Semantic kernel of while programs.

**Atomic Programs**

We first note that the while programs have two atomic commands, namely

\[
\text{skip} \text{ and } x := e;
\]

every other while program command is dependent on some other sub-command(s). Thus, we can define the semantics of any while program in terms of the effect of some sequence of
identity and assignment statements. Thus,

\[
\begin{align*}
\text{First}(\text{skip}, \sigma) &= \text{skip} \\
\text{Rest}(\text{skip}, \sigma) &= \text{skip} \\
\text{Act}(\text{skip}, \sigma) &= \sigma
\end{align*}
\]

\[
\begin{align*}
\text{First}(x:=e, \sigma) &= x:=e \\
\text{Rest}(x:=e, \sigma) &= \text{skip} \\
\text{Act}(x:=e, \sigma) &= \sigma[V_A(e, \sigma)/x].
\end{align*}
\]

We now proceed by induction on the structure of commands.

**Sequencing**

When two commands \(S_1\) and \(S_2\) are sequenced together, the first atomic statement that we should execute is the first of \(S_1\). If \(S_1\) consists only of this atomic command then the rest of the program that we need to consider after execution of \(S_1\) is simply \(S_2\). Alternatively, if \(S_1\) is itself a compound statement, then after execution of the first atomic command of \(S_1\), we need to consider the rest of the command \(S_1\) and then the whole of \(S_2\). Thus,

\[
\begin{align*}
\text{First}(S_1; S_2, \sigma) &= \text{First}(S_1, \sigma) \\
\text{Rest}(S_1; S_2, \sigma) &= \begin{cases} 
S_2 & \text{if } S_1 \text{ is atomic;} \\
\text{Rest}(S_1, \sigma); S_2 & \text{if } S_1 \text{ is not atomic.}
\end{cases}
\end{align*}
\]

**Conditionals**

The command \(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}\) requires the evaluation of the Boolean expression \(b\) in order to determine whether \(S_1\) or \(S_2\) should be executed. We shall consider that any such evaluation requires one cycle of time. Accordingly, we define

\[
\begin{align*}
\text{First}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) &= \text{skip} \\
\text{Rest}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) &= \begin{cases} 
S_1 & \text{if } W_A(b)(\sigma) = \text{tt}; \\
S_2 & \text{if } W_A(b)(\sigma) = \text{ff}.
\end{cases}
\end{align*}
\]

as evaluating the test \(b\) requires one time cycle, and does not alter the state; this is equivalent to executing the skip statement.

**Iteration**

In the case that we have a while loop,

\[
\text{while } b \text{ do } S_0 \text{ od}
\]

we first evaluate the test \(b\) to determine whether we should execute the body of the loop or not. Just as for the Boolean test in conditional statements, we shall consider that we require one step of time to perform any such test, and we shall model it by executing the program skip.
15.3. ALGEBRAIC OPERATIONAL SEMANTICS

If the test evaluates to true on the initial state, we have to first execute the body \( S_0 \) of the loop. Then we execute the while loop again. Otherwise, if the test evaluates to false, we exit from the while loop, which we model by executing the program skip. Thus,

\[
\begin{align*}
First(\text{while } b \text{ do } S_0 \text{ od}, \sigma) &= \text{ skip} \\
Rest(\text{while } b \text{ do } S_0 \text{ od}, \sigma) &= \begin{cases} \\
S_0; \text{while } b \text{ do } S_0 \text{ od} & \text{if } W_A(b)(\sigma) = tt; \\
\text{skip} & \text{if } W_A(b)(\sigma) = ff.
\end{cases}
\end{align*}
\]

To summarise, the AOS of the while language is given by combining the algebraic definition of \( \text{Comp} \) in Section 15.3.1 with this algebraic definition of First and Rest:

| First \( (\text{skip}, \sigma) \) | = | \text{skip} |
| Rest \( (\text{skip}, \sigma) \) | = | \text{skip} |
| First \( (x:=e, \sigma) \) | = | \( x:=e \) |
| Rest \( (x:=e, \sigma) \) | = | \text{skip} |
| First \( (S_1;S_2, \sigma) \) | = | First \( (S_1, \sigma) \) |
| First \( (S_1;S_2, \sigma) \) | = | \begin{cases} \\
S_2 & \text{if } S_1 \in \text{AProg}(\Sigma); \\
Rest(S_1, \sigma);S_2 & \text{if } S_1 \notin \text{AProg}(\Sigma).
\end{cases} |
| First \( (\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) \) | = | \text{skip} |
| First \( (\text{while } b \text{ do } S_0 \text{ od}, \sigma) \) | = | \text{skip} |
| First \( (\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) \) | = | \begin{cases} \\
S_1 & \text{if } W_A(b)(\sigma) = tt; \\
S_2 & \text{if } W_A(b)(\sigma) = ff.
\end{cases} |
| Rest \( (\text{while } b \text{ do } S_0 \text{ od}, \sigma) \) | = | \begin{cases} \\
S_0;\text{while } b \text{ do } S_0 \text{ od} & \text{if } W_A(b)(\sigma) = tt; \\
\text{skip} & \text{if } W_A(b)(\sigma) = ff.
\end{cases} |

**Example** Consider the program \( E \) from Section 14.4.1 to compute Euclid’s algorithm. Suppose we execute \( E \) on the state \( \sigma \) where \( \sigma(x) = 45 \) and \( \sigma(y) = 12 \). Then, we simulate \( E \) with the program formed by taking the following sequence of identity and assignment statements:
### 15.4 Comparison of Semantics

To appear.

### 15.5 Program Properties

To appear.
Chapter 16

Virtual Machines

A virtual machine is a model of a computer. As a model, it is an abstraction of a real physical computer, in terms of, for example,

(i) its architecture,

(ii) the data it computes on.

(iii) the control constructs that sequence operations;

(iv) memory structures.

Virtual machines are designed to be implemented in software and run on different processors. A specification of how a program or a process should behave on a virtual machine enables implementations to be built for different processors, running different operating systems and environments. Thus, virtual machines afford implementation independence. The degree of independence is determined by the degree of abstraction provided in the model.

Any model that is produced is a virtual machine. For example, a relatively low-level model of a computer may have descriptions of data buses that transport data between different processing and memory units; such a description may be helpful to a hardware engineer who may be concerned, for example, with maximising the performance of assembly level instructions. A higher level model of a computer might be used by a software engineer who may be concerned, for example, with the production of a piece of software written in a high-level language such as Pascal, Prolog or Java.

16.1 Machine Semantics and Operational Semantics

We refine our general model of operational semantics to describe the semantics of low-level programming languages and abstract machines. We call the special case

machine semantics

Our refinement involves an adaptation to the notion of state. Clearly at the level of machine semantics the programming language $Prog$ that we study will be used to control the machine, and the clock $Time$ that enumerates the sequences of states will run at a faster level compared to a high-level language. (We shall explain exactly how we can relate implementations at different levels of abstraction in Chapter 17.)
16.1.1 Generating Machine State Execution Traces

First, we observe that when executing a low-level language the distinction between programs and states is somewhat blurred; the program is stored within the state, and certain aspects of the state are determined by the program to be executed. Of course if we were to look very closely at how a high-level language is executed, we would find that this description also holds, yet with high-level languages it is possible and, indeed, desirable that we model their execution at a higher level of abstraction.

Suppose we have some set

\( \text{MachineState} \)

of machine states with a distinguished element

* \( \in \text{MachineState} \)

to determine when we have completed a computation.

We extend our model \( \text{State} \) of states by supposing that we have a function

\( \text{Load} : \text{Prog} \times \text{State} \rightarrow \text{MachineState} \)

such that

\( \text{Load}(P, \sigma) = \text{the state of the machine that results from loading the program} \)
\( P \in \text{Prog} \text{ into memory for execution on the state whose initial} \)
\( \text{values are determined by } \sigma \in \text{State}. \)

We shall also use a function

\( \text{Reset} : \text{MachineState} \rightarrow \text{State} \)

to clear a program from memory, so that

\( \text{Reset}(\text{Load}(P, \sigma)) = \sigma \)

for all \( P \in \text{Prog} \) and \( \sigma \in \text{State}. \)

Thinking of operational semantics, we expect that executing a machine program \( P \in \text{Prog} \) on an initial state \( \sigma_0 \in \text{State} \) that is stored in an initial machine state \( \tau_0 = \text{Load}(P, \sigma_0) \) produces a finite

\( \tau_0, \tau_1, \ldots, \tau_t \)

or infinite

\( \tau_0, \tau_1, \ldots, \tau_t, \ldots \)

sequence of machine states.

To define the behaviour of a low-level program \( P \), we shall use a function

\( \text{Next} : \text{MachineState} \rightarrow \text{MachineState} \)

such that

\( \text{Next}(\tau_t) = \text{the next state } \tau_{t+1} \text{ that results from executing the program that} \)
\( \text{is loaded in memory on the state } \tau_t. \)
16.2. THE REGISTER MACHINE

The machine semantics of a low-level language can be defined via an iterated map

\[ \text{MachineComp} : \text{MachineState} \times \text{Time} \rightarrow \text{MachineState} \]

such that,

\[ \text{MachineComp}(\tau_0, t) = \text{the state } \tau_1 \text{ that results from executing the program that is loaded} \]
\[ \text{in memory on the state } \tau_0 \text{ for } t \text{ machine cycles of time.} \]

We define \text{MachineComp} by induction on time:

\[ \text{MachineComp}(\tau, 0) = \tau \]
\[ \text{MachineComp}(\tau, t + 1) = \text{MachineComp}(\text{Next}(\tau), t) \]

**Definition (Operational Semantics of a Machine)** We define the operational semantics of a machine for any \( P \in \text{Prog}, \sigma \in \text{State} \) and \( t \in \text{Time} \) by:

\[ \text{Comp}(P, \sigma, t) = \text{Reset}(\text{Next}^t(\text{Load}(P, \sigma))). \]

16.2 The Register Machine

16.2.1 Informal Description

We shall construct a virtual machine with a very simple architecture called an abstract or virtual register machine (VRM).

It consists of a sequence of registers. The structure of the machine is based on distinguishing three types of register according to the data that we are allowed to store in them:

- **program counter** a register to determine the order in which the program instructions are to be executed;
- **data** registers to store values that are manipulated by programs; and
- **test** registers to store values that result from performing tests on the data registers.

We suppose that the machine has a single program counter, and unlimited storage, i.e., the machine has infinitely many data and test registers, see Figure 16.1.

```
          r_1  r_2  \ldots  r_n  \ldots  t_1  t_2  \ldots  t_n  \ldots
PC       d_1  d_2  \ldots  d_n  \ldots  p_1  p_2  \ldots  p_n  \ldots
Program Counter  Data Registers  Test Registers
```

*Figure 16.1: Form of register machine.*

Any computation by the machine will access only finitely many registers. These registers store data and Booleans from an arbitrarily chosen data type and its process will be based
on the operations of the data type. Thus, one can say that what makes this register machine model virtual is the abstract nature of its data.

On these VRMs, we shall execute programs which consist of a sequence of simple instructions.

An instruction has one of four forms:

**copy** instructions duplicate data in one register to another;

**jump** instructions conditionally direct the order in which instructions are executed;

**value** instructions place the value of a constant into a register; and

**evaluate** instructions place in a register the result of applying an operation of the data type to register values.

### 16.2.2 Data Type

To specify formally a register machine we begin by choosing a data type. Let $\Sigma$ be any signature of the form:

<table>
<thead>
<tr>
<th>signature</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorts</td>
<td>data</td>
</tr>
<tr>
<td>constants</td>
<td>$... , c : \rightarrow data, ...$</td>
</tr>
<tr>
<td></td>
<td>true : $\rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>false : $\rightarrow Bool$</td>
</tr>
<tr>
<td>operations</td>
<td>$... , f : data^n \rightarrow data, ...$</td>
</tr>
<tr>
<td></td>
<td>$... , rel : data^n \rightarrow Bool, ...$</td>
</tr>
<tr>
<td></td>
<td>$\neg : Bool \rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>$\land : Bool \times Bool \rightarrow Bool$</td>
</tr>
<tr>
<td></td>
<td>$\lor : Bool \times Bool \rightarrow Bool$</td>
</tr>
</tbody>
</table>

For simplicity, we are assuming that there is just one sort of data and the Booleans.

Let $D$ be any $\Sigma$-algebra of the form:
16.2. THE REGISTER MACHINE

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{algebra} & $D$ \\
\textbf{carriers} & $D, B = \{tt, ff\}$ \\
\textbf{constants} & $\ldots, c^D : \to D, \ldots$ \\
& $tt : \to B$ \\
& $ff : \to B$ \\
\textbf{operations} & $\ldots, f^D : D^n \to D, \ldots$ \\
& $\ldots, rel^D : D^m \to B, \ldots$ \\
& $not : B \to B$ \\
& $and : B \times B \to B$ \\
& $or : B \times B \to B$ \\
\hline
\end{tabular}
\end{center}

Any data and any operations and tests may be chosen.

16.2.3 States

Suppose that a VRM program computes over the data type $D$. We consider in turn each of the three types of register.

Program Counter

The program counter is a single register which stores the address of the next program instruction to be executed.

We shall model addresses in a simple manner using the natural numbers.

Data Registers

The data registers of a VRM store values that are computed by a program during execution. Each data register can hold some value $d \in D$. We have arbitrary many data registers in a machine; each register has a unique address.

We model the data registers as a set

$$Reg^D = [DataReg \to D]$$

of locations which are accessed through addresses $\ldots, r \in DataReg, \ldots$, and which store values $\ldots, d \in D, \ldots$: a data register

$$\rho^D \in Reg^D$$

stores the value

$$\rho^D(r) \in D$$

in the register $r \in DataReg$ as shown in Figure 16.2.
 CHAPTER 16. VIRTUAL MACHINES

\[
\begin{array}{cccc}
  r_1 & r_2 & \ldots & r_n \\
  \rho^D(r_1) & \rho^D(r_2) & \ldots & \rho^D(r_n)
\end{array}
\]

Figure 16.2: Data registers \( \rho^D \in \text{Reg}^D \) of the VRM.

**Test Registers**

The test registers hold the value of tests that can be used to determine the execution order of program instructions.

We model the test registers as a set

\[ \text{Reg}^B = [\text{TestReg} \to B] \]

of locations which are accessed through addresses \( \ldots, t \in \text{TestReg}, \ldots \), and which store values \( \ldots, b \in B, \ldots \): a test register

\[ \rho^B \in \text{Reg}^B \]

stores the value

\[ \rho^B(r) \in B \]

in the register \( r \in \text{TestReg} \) as shown in Figure 16.3.

\[
\begin{array}{cccc}
  t_1 & t_2 & \ldots & t_n \\
  \rho^B(t_1) & \rho^B(t_2) & \ldots & \rho^B(t_n)
\end{array}
\]

Figure 16.3: Test registers \( \rho^B \in \text{Reg}^B \) of the VRM.

**Definition (Modelling States)** A VRM state

\[ \tau = (P, pc, \rho^D, \rho^B) \]

consists of a program \( P \), program counter \( pc \in \mathbb{N} \), a set \( \rho^D \in \text{Reg}^D \) of data registers, and a set \( \rho^B \in \text{Reg}^B \). We model the set

\[ \text{VRMState}(D) = \text{Prog}(\Sigma) \times \mathbb{N} \times \text{Reg}^D \times \text{Reg}^B \]

of VRM states as the collection of all possible combinations of programs and values held in the program counter, data registers, and test registers.

### 16.2.4 Programs

The form of a program is:

\[
\begin{align*}
  l &. & I_l \\
  l + 1 &. & I_{l+1} \\
  l + 2 &. & I_{l+2} \\
  \vdots & & \vdots \\
  l + \lambda &. & I_{l+\lambda}
\end{align*}
\]
where \( l, l+1, l+2, \ldots, l+\lambda \in \mathbb{N} \) are labels and \( I_l, I_{l+1}, I_{l+2}, \ldots, I_{l+\lambda} \in \text{Instrn} \) are instructions.

Thus, the syntax of VRM programs is given by:

\[
\text{VRMProgram} := \langle \text{label} \rangle \cdot \langle \text{instrn} \rangle | \\
\text{VRMProgram} \cdot \langle \text{label} \rangle \cdot \langle \text{instrn} \rangle , \langle \text{VRMProgram} \rangle | \\
\langle \text{label} \rangle \cdot \langle \text{instrn} \rangle <\text{newline}> \langle \text{VRMProgram} \rangle |
\]

where we require the labels to successively increment.

Thus,

\[
\text{VRMProgram} = \{ l, I_l, l + 1, I_{l+1}, l + 2, I_{l+2}, \ldots, I_{l+\lambda}, I_{l+\lambda} | \\
l, l+1, l+2, \ldots, l+\lambda \in \mathbb{N}, I_l, I_{l+1}, I_{l+2}, \ldots, I_{l+\lambda} \in \text{Instrn} \} \\
\subseteq (\mathbb{N} \times \text{Instrn})^*
\]

**Semantics**

The semantics of a VRM program is derived from the semantics of its individual instructions.

The program counter register of the VRM state determines the order of execution of the instructions. Execution ceases when there is no instruction with a label equal to the value stored in the program counter.

Thus, given a program

\[
P = l. \quad I_l \quad l+1. \quad I_{l+1} \quad l+2. \quad I_{l+2} \quad \vdots \quad \vdots \quad l+\lambda. \quad I_{l+\lambda}
\]

and a state

\[
(pc, \rho^P, \rho^B)
\]

the behaviour of \( P \) is

\[
M_{VRM}^P(pc, \rho^P, \rho^B) = \begin{cases} 
M_{VRM}(P)(M_{VRMInstrn}(I_{pc})(pc, \rho^P, \rho^B)) & \text{if } l \leq pc \leq l + \lambda \\
(pc, \rho^P, \rho^B) & \text{if } pc < l \text{ and } pc > l + \lambda.
\end{cases}
\]

**16.2.5 Instructions**

The program instructions are summarised in Figure 16.4.

Let us consider the instructions in more detail.

We consider each of the four types of instruction, giving the behaviour

\[
M_{VRMInstrn}^D(i) : VRMState(D) \rightarrow VRMState(D)
\]

of each type of instruction \( i \).
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Intuitive Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>( r_j \leftarrow r_i )</td>
<td>Copy contents of data register ( r_i ) to data register ( r_j ).</td>
</tr>
<tr>
<td>jump</td>
<td>( n \leftarrow t_i )</td>
<td>If test register ( t_i ) contains the value true, increment program position by ( n \in \mathbb{Z} ). Otherwise, increment current program position by one.</td>
</tr>
<tr>
<td>value</td>
<td>( r_j \leftarrow c )</td>
<td>Place value of constant ( c ) in register ( r_j ).</td>
</tr>
<tr>
<td>evaluate</td>
<td>( r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}) )</td>
<td>Place in register ( r_j ), the result of applying function ( f ) to values held in registers ( r_{i_1}, \ldots r_{i_n} ).</td>
</tr>
</tbody>
</table>

**Figure 16.4:** Virtual machine instructions.

**Copy Instructions**

The form of a copy instruction is

\[ r_j \leftarrow r_i \]

for \( r_i, r_j \in \text{DataReg} \). The behaviour of such a copy instruction is that the value of register \( r_i \) is placed in register \( r_j \) and the value of the the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in \text{VRMState}(D) \]

executing an instruction:

\[ r_j \leftarrow r_i \]

produces the state

\[ M_{\text{VRMInstn}}^D(r_j \leftarrow r_i)(pc, \rho^D, \rho^B) = (pc + 1, \rho^D_{\text{new}}, \rho^B) \in \text{VRMState}(D) \]

where

\[ \rho^D_{\text{new}}(r) = \begin{cases} \rho^D(r_j) & \text{if } r = r_j; \\ \rho^D(r) & \text{otherwise}. \end{cases} \]

**Jump Instructions**

The form of a jump instruction is

\[ n \leftarrow t_i \]

for \( n \in \mathbb{Z}, t_i \in \text{TestReg} \). The behaviour of such a jump instruction is that the value of the program counter is incremented by \( n \) if the value in the test register \( t_i \) is true, and otherwise, the value of the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in \text{VRMState}(D) \]

executing an instruction:

\[ n \leftarrow t_i \]

produces the state

\[ M_{\text{VRMInstn}}^D(n \leftarrow t_i)(pc, \rho^D, \rho^B) = (m, \rho^D_{\text{new}}, \rho^B) \in \text{VRMState}(D) \]
where

\[ m = \begin{cases} pc & \text{if } t_i = \text{true}; \\ pc + 1 & \text{otherwise}. \end{cases} \]

### Value Instructions

There are two forms of value instruction: one for data registers, and one for test registers.

The form of the value instruction for data registers is

\[ r_j \leftarrow c \]

for \( r_j \in DataReg, c : \rightarrow data \in \Sigma^{Data} \). The behaviour of such a value instruction is that the value of the constant \( c^D \) is placed in register \( r_j \) and the value of the the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in VRMState(D) \]

executing an instruction:

\[ r_j \leftarrow c \]

produces the state

\[ M^D_{VRMInstr}(r_j \leftarrow c)(pc, \rho^D, \rho^B) = (pc + 1, \rho^D_{\text{new}}, \rho^B) \in VRMState(D) \]

where

\[ \rho^D_{\text{new}}(r) = \begin{cases} c^D & \text{if } r = r_j; \\ \rho^D(r) & \text{otherwise}. \end{cases} \]

The form of the value instruction for test registers is

\[ t_j \leftarrow c \]

for \( t_j \in TestReg, c : \rightarrow \text{Bool} \in \Sigma^{Data} \). The behaviour of such a value instruction is that the value of the constant \( c^D \) (which will be either true or false) is placed in register \( r_j \) and the value of the the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in VRMState(D) \]

executing an instruction:

\[ r_j \leftarrow c \]

produces the state

\[ M^D_{VRMInstr}(t_j \leftarrow c)(pc, \rho^D, \rho^B)(pc + 1, \rho^D, \rho^B_{\text{new}}) \in VRMState(D) \]

where

\[ \rho^B_{\text{new}}(t) = \begin{cases} c^D & \text{if } t = t_j; \\ \rho^B(t) & \text{otherwise}. \end{cases} \]
CHAPTER 16. VIRTUAL MACHINES

Evaluate Instructions

There are three forms of evaluate instruction: one for data registers, one for test registers, and one for data and test registers.

The form of an evaluate instruction for data registers is

\[ r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}) \]

for \( r_{i_1}, \ldots, r_{i_n}, r_j \in DataReg \) and \( f : data^n \rightarrow data \in \Sigma^{Data} \). The behaviour of such an evaluation instruction is that the function \( f^D \) is applied to the values in the data registers \( r_{i_1}, \ldots, r_{i_n} \), and is placed in the data register \( r_j \); and the value of the the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in VRMState(D) \]

executing an instruction:

\[ r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}) \]

produces the state

\[ M^D_{VRMInstrn}(r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}))(pc, \rho^D, \rho^B) = (pc + 1, \rho^D_{\text{new}}, \rho^B) \in VRMState(D) \]

where

\[ \rho^D_{\text{new}}(r) = \begin{cases} f^D(\rho^D(r_{i_1}), \ldots, \rho^D(r_{i_n})) & \text{if } r = r_j; \\ \rho^D(r) & \text{otherwise}. \end{cases} \]

The form of an evaluate instruction for data and test registers is

\[ t_j \leftarrow \text{rel}(r_{i_1}, \ldots, r_{i_n}) \]

for \( r_{i_1}, \ldots, r_{i_n} \in DataReg, t_j \in TestReg \) and \( f : data^n \rightarrow Bool \in \Sigma^{Data} \). The behaviour of such an evaluation instruction is that the function \( \text{rel}^D \) is applied to the values in the data registers \( r_{i_1}, \ldots, r_{i_n} \), and is placed in the test register \( t_j \); and the value of the the program counter is incremented by one. Thus, given a state

\[ (pc, \rho^D, \rho^B) \in VRMState(D) \]

executing an instruction:

\[ t_j \leftarrow \text{rel}(r_{i_1}, \ldots, r_{i_n}) \]

produces the state

\[ M^D_{VRMInstrn}(t_j \leftarrow \text{rel}(r_{i_1}, \ldots, r_{i_n}))(pc, \rho^D, \rho^B) = (pc + 1, \rho^D, \rho^B_{\text{new}}) \in VRMState(D) \]

where

\[ \rho^B_{\text{new}}(t) = \begin{cases} \text{rel}^D(\rho^D(r_{i_1}), \ldots, \rho^D(r_{i_n})) & \text{if } t = t_j; \\ \rho^B(t) & \text{otherwise}. \end{cases} \]

The form of an evaluate instruction for test registers is

\[ t_j \leftarrow \text{not}(t_i) t_j \leftarrow \text{and}(t_{i_1}, t_{i_2}) t_j \leftarrow \text{or}(t_{i_1}, t_{i_2}) \]
for \( t_i, t_j \in \text{TestReg} \). The behaviour of such an evaluation instruction is that the function \( \neg, \land, \lor \) as appropriate, is applied to the values in the test registers \( t_i, t_j \) and \( t_k \) (where \( \neg \) appropriate), and is placed in the test register \( t_j \); and the value of the the program counter is incremented by one. Thus, given a state

\[
(pc, \rho^D, \rho^B) \in \text{VRMState}(D)
\]

executing an instruction:

\[
t_j \leftarrow \text{not}(t_i)
\]

produces the state

\[
M^D_{\text{VRMInstr}}(t_j \leftarrow \text{not}(t_i))(pc, \rho^D, \rho^B) = (pc + 1, \rho^D, \rho^B_{\text{new}}) \in \text{VRMState}(D)
\]

where

\[
\rho^B_{\text{new}}(t) = \begin{cases} 
-\rho^B(t) & \text{if } t = t_j; \\
\rho^B(t) & \text{otherwise.}
\end{cases}
\]

And

\[
t_j \leftarrow \text{and}(t_i, t_j)
\]

produces the state

\[
M^D_{\text{VRMInstr}}(t_j \leftarrow \text{and}(t_i, t_j))(pc, \rho^D, \rho^B) = (pc + 1, \rho^D, \rho^B_{\text{new}}) \in \text{VRMState}(D)
\]

where

\[
\rho^B_{\text{new}}(t) = \begin{cases} 
\land(\rho^B(t_i), \rho^B(t_j)) & \text{if } t = t_j; \\
\rho^B(t) & \text{otherwise.}
\end{cases}
\]

And

\[
t_j \leftarrow \text{or}(t_i, t_j)
\]

produces the state

\[
M^D_{\text{VRMInstr}}(t_j \leftarrow \text{or}(t_i, t_j))(pc, \rho^D, \rho^B) = (pc + 1, \rho^D, \rho^B_{\text{new}}) \in \text{VRMState}(D)
\]

where

\[
\rho^B_{\text{new}}(t) = \begin{cases} 
\lor(\rho^B(t_i), \rho^B(t_j)) & \text{if } t = t_j; \\
\rho^B(t) & \text{otherwise.}
\end{cases}
\]

**Instruction Syntax**

The form of instructions is:

\[
<\text{insn}> ::= <\text{data_register}> \leftarrow <\text{data_register}> | <\text{increment}> \leftarrow <\text{test_register}> | <\text{data_register}> \leftarrow <\text{data_expression}> | <\text{test_register}> \leftarrow <\text{test_expression}>
\]
In particular, we restrict VRM instructions to the following forms:

\[
\text{Instrn} = \{ r_j \leftarrow r_i \mid r_i, r_j \in \text{DataReg} \} \\
\cup \{ n \leftarrow b_i \mid n \in \mathbb{Z}, b_i \in \text{TestReg} \} \\
\cup \{ r_j \leftarrow c \mid c \in \Sigma_{\text{data}}, r_j \in \text{DataReg} \} \\
\cup \{ r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}) \mid f \in \Sigma_{\text{data} \times \text{data}}, r_{i_1}, \ldots, r_{i_n}, r_j \in \text{DataReg} \} \\
\cup \{ t_j \leftarrow \text{true} \mid t_j \in \text{TestReg} \} \\
\cup \{ t_j \leftarrow \text{false} \mid t_j \in \text{TestReg} \} \\
\cup \{ t_j \leftarrow \text{not}(t_i) \mid t_j, t_i \in \text{TestReg} \} \\
\cup \{ t_j \leftarrow \text{and}(t_{i_1}, t_{i_2}) \mid t_j, t_{i_1}, t_{i_2} \in \text{TestReg} \} \\
\cup \{ t_j \leftarrow \text{or}(t_{i_1}, t_{i_2}) \mid t_j, t_{i_1}, t_{i_2} \in \text{TestReg} \} \\
\cup \{ t_j \leftarrow \text{rel}(r_{i_1}, \ldots, r_{i_m}) \mid \text{rel} \in \Sigma_{\text{data} \times \text{bool}}, r_{i_1}, \ldots, r_{i_m} \in \text{DataReg}, t_j \in \text{TestReg} \}
\]

16.2.6 Example

Recall Euclid’s algorithm from Section 13.1.3:

```
program Euclid(input : x, y; output : y);
signature Naturals for Euclidean Algorithm
sorts nat, bool
constants 0 : nat \rightarrow nat
            true, false : \rightarrow bool
operations mod : nat \times nat \rightarrow nat
                \neq : nat \times nat \rightarrow bool
endsig
body
var x, y, z : nat;
begin
  z:= x mod y;
  while z \neq 0 do
    x:= y;
    y:= z;
    z:= x mod y
  od
end
```
We implement Euclid’s algorithm as a VRM program.

\[
\text{program } \quad \text{Euclid}(\text{input} : r_1, r_2; \text{output} : r_2);
\]

\[
\text{signature } \quad \text{Naturals for Euclidean Algorithm}
\]

\[
\text{sorts } \quad \text{nat}, \text{bool}
\]

\[
\text{constants } \quad 0 : \rightarrow \text{nat}
\]

\[
\text{true, false : } \rightarrow \text{bool}
\]

\[
\text{operations } \quad \text{mod} : \text{nat} \times \text{nat} \rightarrow \text{nat}
\]

\[
\neq : \text{nat} \times \text{nat} \rightarrow \text{bool}
\]

\[
\text{not} : \text{bool} \rightarrow \text{bool}
\]

\[
\text{and} : \text{bool} \times \text{bool} \rightarrow \text{bool}
\]

\[
\text{or} : \text{bool} \times \text{bool} \rightarrow \text{bool}
\]

\[
\text{endsig}
\]

\[
\text{body}
\]

\[
\text{var } \quad r_1, r_2, r_3, r_4 : \text{DataReg}; t_1 : \text{TestReg};
\]

\[
\text{begin}
\]

\[
1. \quad r_4 \leftarrow 0
\]

\[
2. \quad r_3 \leftarrow \text{mod}(r_1, r_2)
\]

\[
3. \quad t_1 \leftarrow \neq (r_3, r_4)
\]

\[
4. \quad t_1 \leftarrow 4
\]

\[
5. \quad r_1 \leftarrow r_2
\]

\[
6. \quad r_2 \leftarrow r_3
\]

\[
7. \quad t_1 \leftarrow -5
\]

\[
\text{end}
\]

Executing Euclid’s algorithm on the initial state

\[
\tau = (1, \rho^D, \rho^B) \quad \text{where} \quad \rho^D(r_1) = 45 \quad \text{and} \quad \rho^D(r_2) = 12,
\]
produces the execution trace:

<table>
<thead>
<tr>
<th>r_1</th>
<th>r_2</th>
<th>r_3</th>
<th>r_4</th>
<th>t_1</th>
<th>pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>12</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>2</td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>tt</td>
<td>4</td>
</tr>
<tr>
<td>45</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>tt</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>tt</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>tt</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>tt</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>tt</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>tt</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>tt</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>tt</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>tt</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>tt</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>ft</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>ft</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>ft</td>
<td>8</td>
</tr>
</tbody>
</table>

16.3 Constructing Programs

Unlike \textbf{while} programs, we cannot construct VRM programs from other VRM programs; we can only build VRM programs by sequential composition of individual VRM instructions. In this section, we consider how we can design operations to construct VRM programs from existing VRM programs.

16.3.1 Operations on Programs

\textbf{Program Start point} The label of the first instruction of a VRM program conventionally indicates the starting point for execution. We can establish this with the operation

\[ \text{Start} : \text{Prog} \rightarrow \mathbb{N} \]

which we define by

\[ \text{Start}(l.I_1, \ldots, l + \lambda.I_{l+\lambda}) = l \]

\textbf{Program Length} The number of instructions present in a program determines its syntactic length. This is not necessarily the same as the number of program instructions that will be executed, because jump instructions can determine the order in which instructions are executed.

The syntactic length of a program is given by the operation

\[ \text{Length} : \text{Prog} \rightarrow \mathbb{N} \]
which we define by

\[ \text{Length}(l, \ldots, l + \lambda) = \lambda \]

**Analysing Register Access** A useful operation on a VRM program is to be able to determine which registers affect its behaviour.

We want to define operations

\[
\begin{align*}
\text{UsedDataReg} : \text{Prog} & \rightarrow \mathcal{P}(\text{DataReg}) \\
\text{UsedTestReg} : \text{Prog} & \rightarrow \mathcal{P}(\text{TestReg}) 
\end{align*}
\]

such that

\[
\begin{align*}
\text{UsedDataReg}(P) & = \text{the data registers used in the program } P \\
\text{UsedTestReg}(P) & = \text{the test registers used in the program } P. 
\end{align*}
\]

We shall also want to distinguish between those registers which are written to, and those which are read from. So, we want operations:

\[
\begin{align*}
\text{WrittenDataRegisters} : \text{Prog} & \rightarrow \mathcal{P}(\text{DataReg}) \\
\text{ReadDataRegisters} : \text{Prog} & \rightarrow \mathcal{P}(\text{DataReg}) \\
\text{WrittenTestRegisters} : \text{Prog} & \rightarrow \mathcal{P}(\text{TestReg}) \\
\text{ReadTestRegisters} : \text{Prog} & \rightarrow \mathcal{P}(\text{TestReg}) 
\end{align*}
\]

such that

\[
\begin{align*}
\text{UsedDataReg}(P) & = \text{WrittenDataRegisters}(P) \cup \text{ReadDataRegisters}(P) \\
\text{UsedTestReg}(P) & = \text{WrittenTestRegisters}(P) \cup \text{ReadTestRegisters}(P) 
\end{align*}
\]

Note that a register may be both read from and written to, during the execution of a program.

An obvious method for calculating which registers are used is to introduce operations

\[
\begin{align*}
\text{WrittenDataReg} : \text{Instrn} & \rightarrow \mathcal{P}(\text{DataReg}) \\
\text{ReadDataReg} : \text{Instrn} & \rightarrow \mathcal{P}(\text{DataReg}) \\
\text{WrittenTestReg} : \text{Instrn} & \rightarrow \mathcal{P}(\text{TestReg}) \\
\text{ReadTestReg} : \text{Instrn} & \rightarrow \mathcal{P}(\text{TestReg}) 
\end{align*}
\]

on individual instructions, such that

\[
\begin{align*}
\text{WrittenDataReg}(I) & = \text{the data register written to by the instruction } I \\
\text{ReadDataReg}(I) & = \text{the data register read by the instruction } I \\
\text{WrittenTestReg}(I) & = \text{the test register written to by the instruction } I \\
\text{ReadTestReg}(I) & = \text{the test register read by the instruction } I. 
\end{align*}
\]

We define these operations for any data register \( r_i, \ldots, r_n, r_i, r_j \in \text{DataReg} \), test register
\( t_{i_1}, t_{i_2}, t_i, t_j, \text{ and } n \in \mathbb{Z} \), by:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>WrittenDataReg</th>
<th>ReadDataReg</th>
<th>WrittenTestReg</th>
<th>ReadTestReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_j \leftarrow r_i )</td>
<td>{r_j}</td>
<td>{r_i}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( n \leftarrow t_i )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\ldots, ( r_j \leftarrow c )</td>
<td>{r_j}</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( r_j \leftarrow f(r_{i_1}, \ldots, r_{i_n}) )</td>
<td>{r_j}</td>
<td>{r_{i_1}, \ldots, r_{i_n}}</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{rel}(r_{i_1}, \ldots, r_{i_m}) )</td>
<td>\emptyset</td>
<td>{r_{i_1}, \ldots, r_{i_m}}</td>
<td>{t_j}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{true} )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{false} )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{not}(t_i) )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{and}(t_{i_1}, t_{i_2}) )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{t_j}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>( t_j \leftarrow \text{or}(t_{i_1}, t_{i_2}) )</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>{t_j}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Thus,

\[
\text{WrittenDataRegisters}(l.I_1, \ldots, l + \lambda.I_{l+\lambda}) = \bigcup_{k=l}^{l+\lambda} \text{WrittenDataReg}(I_k)
\]

\[
\text{ReadDataRegisters}(l.I_1, \ldots, l + \lambda.I_{l+\lambda}) = \bigcup_{k=l}^{l+\lambda} \text{ReadDataReg}(I_k)
\]

\[
\text{WrittenTestRegisters}(l.I_1, \ldots, l + \lambda.I_{l+\lambda}) = \bigcup_{k=l}^{l+\lambda} \text{WrittenTestReg}(I_k)
\]

\[
\text{ReadTestRegisters}(l.I_1, \ldots, l + \lambda.I_{l+\lambda}) = \bigcup_{k=l}^{l+\lambda} \text{ReadTestReg}(I_k).
\]

### 16.3.2 Building Programs

### 16.4 Properties

list of lemmas that satisfies property \( x \) (e.g., iso inv thm) (without proof)

If a register other than the program counter does not appear in an VRM program \( P \), then its value is unchanged by the execution of \( P \):

**Lemma (Register Functionality)** Let \( r \in \mathbb{N} \) be a register and \( P \in \text{VRMProg}(\Delta) \) such that \( r \not\in \text{Reg}(P) \). Then for any state \( \tau \in \text{VRMState}(B) \),

\[
(M_B^\Delta(P)(\tau))(r) = \tau(r).
\]

**Proof** By inspection of the semantics of VRM programs. \( \square \)

We can strengthen this observation to cover registers that are not used as destination registers:

**Lemma (Non-destination Register Functionality)** Let \( r \in \mathbb{N} \) be a register. Let \( P \in \text{VRMProg}(\Delta) \) be a program such that \( r \not\in \text{Reg}_{\text{Dest}}(P) \). Then for any state \( \tau \in \text{VRMState}(B) \),

\[
(M_B^\Delta(P)(\tau))(r) = \tau(r).
\]
16.4. PROPERTIES

Proof By inspection of the semantics of VRM programs. □

Looking at the situation from the converse view, if a register other than the program counter does not appear in an VRM program \( P \), then its value will not affect the execution of \( P \):

Lemma (Non-Participating Register Preservation) Let \( r \in \mathbb{N} \) be a register and \( P \in \text{VRMProg}(\Delta) \) such that \( r \notin \text{Reg}(P) \). Then for any state \( \tau \in \text{VRMState}(B) \) and any value \( v \in B \),

\[
M^B_{i_0}(P)(\tau[v/r]) = M^B_{i_0}(P)(\tau).
\]

Proof By inspection of the semantics of VRM programs. □

Again, we can strengthen this observation, this time to cover registers that are not used as origin registers:

Lemma (Non-Origin Register Preservation) Let \( r \in \mathbb{N} \) be a register and \( P \in \text{VRMProg}(\Delta) \) such that \( r \notin \text{Reg}_{\text{Ori}}(P) \). Then for any state \( \tau \in \text{VRMState}(B) \) and any value \( v \in B \),

\[
M^B_{i_0}(P)(\tau[v/r]) = M^B_{i_0}(P)(\tau).
\]

Proof By inspection of the semantics of VRM programs. □

If we are executing a program \( P \) in which \( r \) does not occur in the source registers of \( P \), then we can delay substituting a value \( v \) in \( r \). If \( r \) is not in the destination registers of \( P \) either, then we can delay the substitution of \( v \) in \( r \) until after the execution of \( P \). If \( r \) is in the destination registers of \( P \), then it may be overwritten by the execution of \( P \). In particular, if the program exhibits sequential instruction behaviour, we are guaranteed that any value in \( r \) will be overwritten by executing \( P \); in this situation, we need not perform the substitution of \( v \) in \( r \).

Lemma (Delaying Substitution)

\[
M^B_{i_0}(P)(\tau[v/r]) = \begin{cases} 
M^B_{i_0}(P)(\tau[v/r]) & \text{if } r \notin P; \\
M^B_{i_0}(P)(\tau) & \text{if } r \notin \text{source}(P), r \in \text{dest}(P) \text{ and } P \text{ exhibits sequential instruction behaviour.}
\end{cases}
\]

Proof ... □

If a jump instruction does not appear in an VRM program \( P \), and the value of the program counter is set to the label of the first instruction of \( P \), then the program will terminate, and it will have executed the instructions of \( P \) sequentially, starting at the first instruction.

Lemma (Sequential Instruction Behaviour) Let \( P \) be a program

\[
m. \quad I_m \\
m+1. \quad I_{m+1} \\
\vdots \\
n. \quad I_n
\]

in which the instructions \( \ldots, I_j \neq d \leftarrow o_1, o_2, \ldots \). Then, for any set \( \rho \in [\mathbb{N} \rightarrow B] \) of registers,

\[
M^B_{i_0}(P)((m, \rho)) \downarrow (n+1, \rho')
\]

where

\[
M^B_{i_0}(n.I_n) \cdots (M^B_{i_0}(m+1.I_{m+1}) (M^B_{i_0}(m.I_m)((m, \rho)))) = (n+1, \rho).
\]
Proof By induction on the length of programs. \[\square\]

A jump instruction which uses the same origin registers will act as an unconditional jump instruction:

**Lemma (Unconditional Jump Behaviour)** For any set \(\rho \in [N \rightarrow B]\) of registers,

\[
M^{B}_{\rho}(l.n \leftarrow m, m)((l, \rho)) = (l + n, \rho)
\]

**Proof** By the semantics of jump instructions. \[\square\]

If we want to preserve the values stored in certain registers, we can ensure that they are recoverable after the execution of a program \(P\), by copying the required values to registers that are not used by \(P\):

**Lemma (Register Restoration)** Let \(P\) be a program which always terminates with the value of the program counter set to \(\text{Last}(P) + 1\). Let \(d_1, \ldots, d_n \not\in \text{Reg}(P)\) be register values that do not appear in \(P\). Let \(P'\) be a program:

\[
\begin{align*}
1. & \quad d_1 \leftarrow o_1 \\
\vdots & \quad \vdots \\
n. & \quad d_n \leftarrow o_n \\
n + 1. & \quad \begin{cases} \\
& \quad P \\
\vdots & \quad \vdots \\
n + l. & \quad \begin{cases} \\
& \quad o_1 \leftarrow d_1 \\
\vdots & \quad \vdots \\
2n + l. & \quad o_n \leftarrow d_n \\
\end{cases}
\end{cases}
\end{align*}
\]

Then for any state \(\tau\),

\[
M^{B}_{\rho}(P')(\tau) = M^{B}_{\rho}(P)(\tau)[\tau(o_1)/o_1] \cdots [\tau(o_n)/o_n][\tau(o_1)/d_1] \cdots [\tau(o_n)/d_n][2n + l/\text{PC}].
\]

**Proof** By the Register Functionality Lemma and the semantics of copy instructions. \[\square\]

If we change the values held in two distinct program registers, it does not matter in which order we make the substitutions. If however, we make more than one change to the value held in a particular program register, then the latest substitution will over-ride any previous ones:

**Lemma (Substitution Lemma)** For any state \(\tau \in \text{VRMState}(D)\) any registers \(r, r' \in N\), and any values \(v, v' \in D\):

\[
\tau[v/r][v'/r'] = \begin{cases} \\
\tau[v'/r'][v/r] & \text{if } r \neq r'; \\
\tau[v'/r'] & \text{otherwise.}
\end{cases}
\]

**Proof** Let \(r'' \in N\) be an arbitrary register. The proof is by the definition of register evaluation
and substitution:

\[
(\tau[v/r][v'/r'])(r'') = \begin{cases} 
  v' & \text{if } r' = r''; \\
  \tau[v/r](r'') & \text{otherwise.} 
\end{cases}
\]

\[
= \begin{cases} 
  v & \text{if } r = r'', r' \neq r''; \\
  \tau(r'') & \text{if } r \neq r' \neq r''. 
\end{cases}
\]

\[
= \begin{cases} 
  v & \text{if } r = r'', r' \neq r''; \\
  \tau[v'/r'](r'') & \text{if } r \neq r'. 
\end{cases}
\]

\[
= \begin{cases} 
  \tau[v'/r'][v/r](r'') & \text{if } r \neq r'; \\
  \tau[v'/r'](r'') & \text{otherwise.} 
\end{cases}
\]

\(\square\)

The behaviour of a program on a set of registers is not affected by the start point of the instruction labels; if the start point and the program counter coincide, the program behaviour is independent of the precise value for the first label:

**Lemma (Relabelling Invariance)** For any program \(P\), labels \(l, m, n \in \mathbb{N}\) and registers \(\rho \in \text{VRMState}(B)\),

\[M^B_{i_0}(P)((m, \rho)) \simeq (n, \rho') \Rightarrow M^B_{i_0}(\text{Relabel}(l, P))((l, \rho)) \simeq (n - m + l, \rho').\]

**Proof** By induction on the length of programs. \(\square\)

We execute a program whilst the program counter points to a program instruction:

**Lemma (Execution Cessation)** For any program \(P \in \text{VRMProg}(\Delta)\) with \(l \leq \text{Start}(P)\) or \(l > \text{Last}(P)\),

\[M^B_{i_0}(P)((l, \rho)) = (l, \rho)\]

**Proof** By induction on the length of programs. \(\square\)

The behaviour of an VRM program is determined by the composition of the behaviour of single instructions:

**Lemma (Single Instruction Execution)** For any program \(P = \ldots, l.I_l, \ldots \in \text{VRMProg}(\Delta)\) and label \(l \in \mathbb{N}\) with \(\text{Start}(P) \leq l \leq \text{Last}(P)\), and any set \(\rho \in [\mathbb{N} \rightarrow B]\) of registers,

\[M^B_{i_0}(P)((l, \rho)) = M^B_{i_0}(P)(M^B_{i_0}(l.I_l)((l, \rho))).\]

**Proof** By induction on the length of programs. \(\square\)

Whilst we execute an VRM program one instruction at a time, we typically want to reason about an VRM program using larger steps, i.e., in terms of more than one instruction at a time:

**Lemma (Execution Decomposition)** For any VRM programs \(P_1, P_2 \in \text{VRMProg}(\Delta)\), and any states \(\tau_1, \tau_2, \tau_3 \in \text{VRMState}(D)\),

\[M^B_{i_0}(P_1)(\tau_1) \simeq \tau_2 \text{ and } M^B_{i_0}(P_2)(\tau_2) \simeq \tau_3 \Rightarrow M^B_{i_0}(P_2)(M^B_{i_0}(P_1)(\tau_1)) \simeq \tau_3.\]

**Proof** By induction on the lengths of the programs \(P_1\) and \(P_2\). \(\square\)
Chapter 17

Compilation

Incomplete Draft

Let us reflect on the rôle of a programming language in the process of making computations. In Chapter 1, we explained our concept of a programming language: a programming language is a formal description or definition of what might be loosely called a programming notation. A programming notation is used to describe algorithms and is based on some model of computing. A computation establishes a level of abstraction through its choice of data, algorithmic operations and control.

In even the simplest of computations, several distinct levels of abstraction, models of computation and programming notations are involved. The complexities of computer implementations require these different levels to manage their automation.

First, there are the levels of the problem domain. The computational problem or task needs to be formulated in terms of functions on data, and algorithms need to be chosen to compute the functions. Secondly, there are the software levels: a programming language must be chosen, and specific programs written that encode the algorithms. These programs are subsequently processed and executed by further software systems, including operating systems and networks. Finally, there are the levels of the machine. These are also characterised by programs and include assembly programs that are based on the machine architecture and machine code programs.

The point is this:

*The different models of computation define a hierarchy of different levels of computational abstraction.*

*The levels of computational abstraction are defined formally by means of specification and programming languages.*

From the early days of computing, the problem of constructing abstractions and supporting them by systematic methods of translating between each level of abstraction has been a fundamental problem that has led directly to fundamental discoveries (recall Chapter 2). Levels of abstraction are ubiquitous in computing. Although widely used, they are not well understood theoretically.

Indeed, the same can be said of notions of abstraction in other sciences. Everywhere there are mathematical models that have been designed to abstract different features of natural
phenomena at different levels of spatial or temporal abstraction. It is a difficult problem to show that one model is an abstraction of another.

In fact, modelling with levels of abstraction is better understood in Computer Science than elsewhere! One reason is that we have general frameworks — the programming languages — which define formally the levels of abstraction in general terms. Against this background, the idea of an

integrated hierarchy of levels of computational abstraction

qualifies as another Big Idea in Computer Science. In this final technical chapter we will explore this idea in an extremely simple special case.

Compilation is ubiquitous in computing. Typically, compilation is associated with the translation from a high-level user-oriented language into a low-level machine-oriented language. However, compilation is much more general. Simply stated, a compiler translates each program \( P \) in one language to an “equivalent” program \( P' \) in another language. The theoretical problem is:

In what precise sense are the programs \( P \) and the compiled program \( P' \) equivalent?

Or, said differently,

What exactly does it mean for a compiler to be correct?

We will examine in very general terms what it means to compile one language into another. Each of the types of semantics we have met — the input-output semantics, operational semantics, and machine semantics — require different criteria for correctness. We will show how to

- formulate correctness equations that express the fact that a compiler is correct; and
- structure the definition of a compiler on the basis of the structure of the languages.

Then we will apply this general discussion to the two main languages we have studied. We think of the while language as a high-level language that may be compiled into the register language, which is based on lower level data and control constructs. We will show in detail how to

- define a compiler between the two languages using structural induction; and
- prove its correctness equations using structural induction.

Thus, in this final chapter we will use many of the concepts and techniques we have developed in our studies of data, syntax and semantics.

17.1 What is Compilation?

A compiler is a translator between programming languages. In particular, it takes programs written in one language, the source language, and translates them into programs that are in another language, the target language.

Whilst a compiler simply processes syntax, we cannot define the purpose of a compiler without considering semantics. A compiler needs to “preserve” semantics: the behaviour of a program and its compiled version should be “equivalent”.

Thus, for any program \( P \), and any state \( \sigma \), we need
the execution of the program $P$ on the state $\sigma$

to be "equivalent" to

the execution of the compiled version $P'$ of the program $P$ on a state $\sigma'$ equivalent to $\sigma$.

Let us model the situation. Suppose we have sets

$$Progs \quad \text{and} \quad Prog_T$$

of source and target programs, and a function

$$\text{Compile} : Progs \rightarrow Prog_T$$

from source programs in $Progs$ to target programs in $Prog_T$.

We want to compare the behaviours of source programs $P \in Progs$ and target programs $\text{Compile}(P) \in Prog_T$.

17.1.1 Input-Output Semantics

Suppose both source and target languages have input-output semantics. In the case of input-output semantics, we have functions

$$M^S_{io} : Progs \rightarrow [\text{States} \sim \text{States}]$$

$$M^T_{io} : Prog_T \rightarrow [\text{States} \sim \text{States}]$$

that describe how $Progs$-programs and $Prog_T$-programs behave in terms of inputs and outputs only. We must compare the input-output behaviour

$$M^S_{io}(P) : \text{States} \sim \text{States}$$

of a source program $P \in Progs$, and the input-output behaviour

$$M^T_{io}(\text{Compile}(P)) : \text{States} \sim \text{States}$$

of the compiled program $\text{Compile}(P) \in Prog_T$. In particular, we shall want

$$M^S_{io}(P) \quad \text{and} \quad M^T_{io}(\text{Compile}(P))$$

to produce equivalent results if they are executed on equivalent inputs.

Comparing States

To describe this equivalence, we need first to consider the set of states. Suppose that the source programs compute over some set

$$\text{States}$$

of states, and the target programs over some set

$$\text{States}$$
of states. We need some means

\[ \text{Encode} : \text{State}_S \to \text{State}_T \]

of relating the state sets, such that for any state \( \sigma \in \text{State}_S \),

\[ \sigma \text{ and } \text{Encode}(\sigma) \]

represent equivalent input states. We will also find it useful to have an inverse function

\[ \text{Decode} : \text{State}_T \to \text{State}_S \]

such that for any state \( \tau \in \text{State}_T \),

\[ \tau \text{ and } \text{Decode}(\tau) \]

represent equivalent states. Thus,

\[ \text{Decode}(\text{Encode}(\sigma)) = \sigma \]

but interestingly,

\[ \text{Encode}(\text{Decode}(\tau)) \neq \tau \]

in many cases. Mathematically, this means that \text{Encode} is an injective map with left-inverse \text{Decode} or, equivalently that \text{Decode} is a surjective map with right-inverse \text{Encode}. This is because

(i) all states in \( \text{State}_S \) must have some representative in \( \text{State}_T \); and

(ii) for any source state, there need not be a unique representative state in \( \text{State}_T \).

So, if we execute the source program on a state \( \sigma \), this should give us an equivalent output state to that of executing the compiled program on an equivalent input state \( \text{Encode}(\sigma) \) as shown in Figure 17.1.

![Figure 17.1: Commutative diagram illustrating the correctness of a compiler for a single program \( P \in \text{Progs}_S \)](image)

This gives us a definition of the correctness of compiling a single program; for the correctness of a compiler, we want to consider all possible programs. We need to compare the executions of

\[ M_{w_0}^S(P)(\sigma) \text{ and } M_{w_0}^T(\text{Compile}(P))(\text{Encode}(\sigma)) \]

on equivalent input states. We want their outputs to be equivalent.
17.1. WHAT IS COMPILATION?

**Definition (Compiler Correctness for Input-Output Semantics)** The compiler

\[ \text{Compile} : \text{Progs} \rightarrow \text{ProgsT} \]

is said to be *correct* with respect to the input-output semantics \( M^S_{io} \) and \( M^T_{io} \) and state transformation maps *Encode* and *Decode*, if for all programs \( P \in \text{Progs} \) and all states \( \sigma \in \text{States} \),

\[
\text{Correctness Equation} \quad M^S_{io}(P)(\sigma) \simeq \text{Decode}(M^T_{io}(\text{Compile}(P))(\text{Encode}(\sigma))).
\]

Equivalently, we require the diagram shown in Figure 17.2 to commute.

![Diagram representing compiler correctness for input-output semantics](image)

Figure 17.2: Commutative diagram representing compiler correctness for input-output semantics.

### 17.1.2 Operational Semantics

In the case of operational semantics, we have functions

\[
\begin{align*}
\text{Comp}_S : & \quad \text{Progs} \times \text{States}_S \times \text{Time}_S \rightarrow \text{States}_S \\
\text{Comp}_T : & \quad \text{ProgsT} \times \text{States}_T \times \text{Time}_T \rightarrow \text{States}_T
\end{align*}
\]

that describe how *Progs*-programs and *ProgsT*-programs behave. We must compare the behaviour

\[
\text{Comp}_S(P, \sigma, t)
\]

of a source program \( P \in \text{Progs} \), over all states \( \sigma \in \text{States}_S \) and all time cycles \( t \in \text{Time}_S \), with the behaviour

\[
\text{Comp}_T(\text{Compile}(P), \tau, r)
\]

of the compiled program \( \text{Compile}(P) \in \text{ProgsT} \) on *certain* states \( \tau \in \text{States}_T \) and at *certain* time cycles \( r \in \text{Time}_T \). This comparison is much more concrete and technically sensitive than that for input-output semantics because of the role of time in operational semantics. However, we can follow the ideas we used for input-output semantics.

For a program \( P \) to be correctly compiled, we need its behaviour

\[
\text{Comp}(P, \sigma, t)
\]

to be equivalent to some

\[
\text{Comp}(\text{Compile}(P), \tau, r)
\]
where $\tau$ and $r$ are to be determined from $P$, $\sigma$ and $t$ in some way. Following the case for input-output semantics, we assume some means of relating the state sets by functions

$$\text{Encode} : \text{State}_S \rightarrow \text{State}_T$$

and

$$\text{Decode} : \text{State}_T \rightarrow \text{State}_S.$$  

We can assume that $\text{Encode}(\sigma) = \tau$.

The question of relating timing is new and we assume some function

$$\lambda : \text{Progs} \times \text{State}_S \times \text{Times}_S \rightarrow \text{Time}_T$$

to calculate

$$r = \lambda(P, \sigma, t).$$

**Definition (Compiler Correctness for Operational Semantics)** The compiler

$$\text{Compile} : \text{Progs} \rightarrow \text{Progs}_T$$

is said to be *correct* with respect to the operational semantics $\text{Comp}_S$ and $\text{Comp}_T$, state transformation maps $\text{Encode}$ and $\text{Decode}$, and time transformation maps $\lambda$, if:

**Correctness Equation**

$$\text{Comp}_S(P, \sigma, t) = \text{Decode}(\text{Comp}_T(\text{Compile}(P), \text{Encode}(\sigma), \lambda(P, \sigma, t)))$$

for all programs $P \in \text{Progs}$, all states $\sigma \in \text{State}_S$ and all time cycles $t \in \text{Times}_S$. Equivalently, we require the diagram shown in Figure 17.3 to commute, in which the function

$$\Phi : (\text{Progs} \times \text{State}_S \times \text{Times}_S) \rightarrow (\text{Progs}_T \times \text{State}_T \times \text{Time}_T)$$

is defined by

$$\Phi(P, \sigma, t) = (\text{Compile}(P), \text{Encode}(\sigma), \lambda(P, \sigma, t)).$$

![Figure 17.3: Commutative diagram representing compiler correctness for operational semantics.](image)

### 17.2 Structuring the Compiler

We now have two definitions of the correctness of a compiler, one for input-output semantics and one for operational semantics, but how are we to define Compile? We want a systematic way of translating all the programs in $\text{Progs}_S$ that we can model to help us

(i) implement a compiler; and

(ii) reason about the behaviour of a compiler, for example, to prove its correctness.
17.2. STRUCTURING THE COMPILER

17.2.1 Defining a Compiler

From the concrete syntax of a high level language, an abstract syntax is derived that defines the semantically meaningful constructs. The abstract syntactic structure of the higher level language is used to structure the definition of the compiler. We shall define a compiler by structural induction on the abstract syntax of the higher level programming language.

Suppose the abstract syntax of $\text{Progs}$ is defined by some atomic programs and some program-forming operations that put together old programs to create new ones. Thus, if a source program

$$P \in \text{Progs}$$

is dependent on source sub-programs

$$P_1, \ldots, P_n \in \text{Progs},$$

via some operator, then we shall construct the compiled version

$$\text{Compile}(P) \in \text{Progs}$$

of $P$ from compiled versions

$$\text{Compile}(P_1), \ldots, \text{Compile}(P_n) \in \text{Progs}$$

of $P_1, \ldots, P_n$.

If a source program is atomic it is independent of any sub-programs, then we shall need to translate it into a target program that is independent of any other compiled source programs.

If we are to automate the compilation though, the way in which we put together the target sub-programs

$$\text{Compile}(P_1), \ldots, \text{Compile}(P_n)$$

to yield $\text{Compile}(P)$ must be systematic. Thus, for each source program

$$P$$

that is atomic, we shall need a target program

$$\text{Compile}(P)$$

which is independent of any other compiled source program.

For each way in which we can construct a source program

$$P = f^S(P_1, \ldots, P_n)$$

by means of a program-forming operator $f^S$, from sub-components $P_1, \ldots, P_n$, we shall construct a target program

$$\text{Compile}(P) = f^T(\text{Compile}(P_1), \ldots, \text{Compile}(P_n))$$

from sub-components $\text{Compile}(P_1), \ldots, \text{Compile}(P_n)$ using target language “glue” $f^T$. The target programs for atomic source programs and the operators will not necessarily be built-in target language constructs: they may require many individual target language constructs.
17.2.2 Algebraic Model of Compilation

This ideal view of compiler construction is algebraic. We have, or must build,

an algebra of source programs and an algebra of target programs.

In particular, the algebra of syntax of source programs and the algebra of syntax of compiled target programs both have the same signature.

A compiler translates programs, so it acts on syntax. In particular it acts on program language syntax. What do we know about the syntax of programming languages that we can use to define a compiler? Recall from Chapter 11 that although most programming languages are not context-free, it can be helpful to use a context-free grammar to describe the underlying structure of a language. In Chapter 12, we considered abstract syntax algebraically at some length. We saw how context-free grammar structures can also be modelled algebraically as term algebras, so Progs can be expected to be structured algebraically.

As Compile is a function defined by structural induction which is between algebras of the same signature, Compile is a homomorphism.

17.2.3 Structuring the States

Our definitions of compiler correctness all depend on some notion of equivalence between source and target states. A state of a computation stores data, so to compare a source state $\sigma \in \text{State}_S$ and a target state $\tau \in \text{State}_T$, we need to know

(i) where the data that is stored in $\sigma$ is stored in $\tau$; and

(ii) how we are to compare the data.

Thus, given a location $x$ in a source state $\text{State}_S$ which stores the value $a$, we need to know the location $y$ in the target state $\text{State}_T$ that is meant to perform the same task. And we need a notion of equivalence on the data that is stored in there.

From an algebraic point of view, we have or must build,

an algebra of source states and an algebra of target states.

If the algebra of source states has operations of access and store data, then the algebra of target states will need operations of accessing and storing data. In particular, accessing equivalent locations should yield equivalent data values.

17.3 Proof Techniques

If a compiler is defined by structural induction, we can exploit this to reason about its correctness. As a basis, we consider first the correctness of the compiler on atomic source programs. Then we use structural induction to reason about the correctness of the compiler on programs that are constructed from sub-components.
17.3. PROOF TECHNIQUES

17.3.1 Other Formulations of Correctness

The definitions of compiler correctness for both input-output semantics and operational semantics are intuitive. They compare the behaviour of source programs and their compiled versions. There are other ways of representing this principle.

A more general definition of compiler correctness is based on the observation that there may be many target states that are equivalent to a given source state. For example, if the target state has a program counter, then does it matter what value the program counter is initially set to, provided that it is the same as the first instruction of the program that it is meant to execute?

In the definitions of correctness given in Sections 17.1.1 and 17.1.2, the target state may be over-specified by the function $Encode : States_S \rightarrow States_T$. This can be rectified for example, by relaxing $Encode$ from a function to a relation between source and target states. Alternatively, the function $Decode$ can be used to relate both the input source and target states, as well as the output source and target states.

17.3.2 Data Types

When we are comparing the behaviour of two programs, ultimately the comparison is on the data that is manipulated.

If the source programs operate over some data type $A$ and the target programs over some data type $B$, then at some point we shall need to compare values

$$a_1, a_2, \ldots \in A$$

that are produced by source programs, and values

$$b_1, b_2, \ldots \in B$$

that are produced by target programs.

The purpose of the compiler is to be able to produce the values $b_1, b_2, \ldots$ that are equivalent, in some sense, to the values $a_1, a_2, \ldots$. At this point we can separate out two concerns:

1. How do we compare values from $A$ and $B$?

2. How do we construct operations on $B$: given values $b_1, \ldots, b_n$ that are equivalent to $a_1, \ldots, a_n$, how can we produce a value that is equivalent to $f^A(a_1, \ldots, a_n)$, but by using $b_1, \ldots, b_n$?

From an algebraic point of view, we have or must build,

an algebra of source data and an algebra of target data.

17.3.3 Recursion and Structural Induction

If we define a function on the natural numbers using recursion, we can reason about it by using induction. More generally, if we define a function on some data type by using recursion, we can reason about it by using structural induction. If we have a function between two data
types that have the same structure, and which is defined by recursion, this function will be a homomorphism. Thus, we can reason about a homomorphism using using structural induction.

From an algebraic point of view, we have algebras of syntax, data and state. The compiler can be modelled by a homomorphism, and so we can reason about its correctness using structural induction.

Suppose the algebra of source programs has

(i) constants \( \ldots, c^S, \ldots \) representing source programs that are independently constructed, and

(ii) program constructing functions \( \ldots, f^S, \ldots \) which create new source programs

\[
\ldots, f^S(P_1, \ldots, P_n), \ldots
\]

from existing source programs \( \ldots, P_1, \ldots, P_n \).

Then we define the compiler \( \text{Compile} \) recursively:

\[
\ldots, \text{Compile}(c^S) = c^T, \ldots
\]

\[
\ldots, \text{Compile}(f^S(P_1, \ldots, P_n)) = f^T(\text{Compile}(P_1), \ldots, \text{Compile}(P_n)), \ldots
\]

where \( \ldots, c^T, \ldots \) are constants representing target programs, and \( \ldots, f^T, \ldots \) are target program constructing functions. And we reason about the behaviour of \( \text{Compile} \) using structural induction:

1. We prove that \( \text{Compile} \) behaves as required on the base cases \( \ldots, c^S, \ldots \).

2. We assume the structural induction hypotheses that \( \text{Compile} \) behaves as required on the programs \( \ldots, P_1, \ldots, P_n, \ldots \).

3. Using the structural induction hypotheses, we prove that \( \text{Compile} \) behaves as required on the cases \( \ldots, f^S(P_1, \ldots, P_n), \ldots \).

17.4 Comparing while Statements and Register Machine Programs

We will now start our case study of a compiler that translates the \textit{while} programming language \( WP \) to the register machine language \( RMProg \). Following our general discussion, we will:

1. Choose an abstract syntax for \( WP \).

2. Compare data types in \( WP \) and \( RMProg \), choose the input-output semantics for \( WP \) and the input-output semantics derived from the machine semantics of \( RMProg \), and define appropriate maps \( Encode \) and \( Decode \) between state spaces, to obtain a precise compiler correctness criterion.

3. Define a map

\[
\text{Compile} : WP \rightarrow RMProg
\]

using structural induction on the abstract syntax of \( WP \).
4. Prove Compile is correct under the compiler correctness criterion.

We will also

5. Establish algebraic structures for the syntax of the languages so that

\[ \text{Compile : } WP \rightarrow RMProg \]

is a homomorphism.

What are the differences between the two languages? Essentially, they are as follows.

**Data Types** Because we have emphasised the rôle of abstract user-defined data types in the notion of a virtual machine, our register machine language \( RMProg \) and while language both operate over arbitrary data, modelled by signatures and algebras.

**Atomic Statements** In \( WP \), we have skip and assignments \( x:=e \) with arbitrary expression evaluation. In \( RMProg \), we have the copy instructions \( x \leftarrow y \) and instructions that perform single applications of atomic data type operations \( x \leftarrow c \) and \( x \leftarrow f(y_1, \ldots, y_n) \). Clearly, in computing, we will need to construct register machine programs to evaluate expressions.

**Control Statements** In \( WP \), we have constructs that control arbitrary subprograms, and in \( RMProg \) instructions are scheduled one at a time. For example, in \( WP \), we can write

\[ S_1;S_2 \]

which schedules “blocks” of constructs \( S_1 \) and \( S_2 \); in \( RMProg \), we can write lists of single instructions.

In \( WP \), we can write

\[ \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \]

and

\[ \text{while } b \text{ do } S_0 \text{ od} \]

which evaluates arbitrary Boolean-valued tests and then schedules “blocks” of constructs, by selection and iteration respectively.

In \( RMProg \), there is the unconstrained jump control construct \( n \leftarrow b_1, b_2 \) that allows any choice of next instruction. Clearly, in compilation, we will need to construct register machine programs to evaluate Boolean expressions and reschedule blocks of instructions.

### 17.5 Memory Allocation

#### 17.5.1 Data Equivalence

Both our while programs and register machine programs compute over arbitrary data types. Let us fix arbitrary data types \( A \) with the Booleans for while programs and \( D \) with the Booleans for register machine programs to compute over.
We shall suppose that any value we can compute in $A$, we can compute an equivalent value in $D$. The syntactic representations of these values need not be the same, but we shall want their values to be the same under some representation.

We shall use functions

\[
\begin{align*}
\text{Encode}_{\text{Data}} : & \ A \to D \\
\text{Decode}_{\text{Data}} : & \ D \to A
\end{align*}
\]

to convert between the data types $A$ and $D$, such that

1. $\text{Encode}_{\text{Data}}$ is a surjective function;
2. $\text{Decode}_{\text{Data}}(\text{Encode}_{\text{Data}}(a)) = a$; and
3. $\text{Encode}_{\text{Data}}$ is a homomorphism.

### 17.5.2 Memory Allocation and Structuring the Register Machine States

Recall from ..., that in a register machine

\[(pc, \rho) \in \text{RMState}(D)\]

we have a program counter $pc \in \mathbb{N}$ and an infinite number of registers

\[\rho \in [\text{Reg} \to D]\]

in which we can access and update data of type $D$ according to the instructions of the RM program.

Because we want to construct RM programs to simulate while programs, we shall impose an artificial construction on RM architectures to help us simulate the architecture of while program machines. We think of while programs that compute over some data type $A$ as operating over an abstract state space

\[\text{State}(A) = [\text{Var} \to A].\]

We shall reserve some of the registers of the RM to store the data associated with while program variables. We shall also categorise the remaining RM registers according to different tasks that we shall need to do to simulate the execution of while programs.

### Register Allocation

Let

\[
\begin{align*}
\text{Allocate} : & \ Var \to \text{Reg}_{\text{var}}^D \\
\text{Restore} : & \ \text{Reg}_{\text{var}}^D \to \text{Var}
\end{align*}
\]

be mappings between the while variables and their RM registers, such that $\text{Allocate}$ is an injective function and $\text{Restore}$ is a surjective function, with

\[\text{Restore}(\text{Allocate}(x)) = x.\]

We shall distinguish in the register machine between four types of register:
(i) the register

\[ PC \]

that we use to store

\[ RMState(D)(PC) \in \mathbb{N} \]

the value of the program counter;

(ii) the registers

\[ Reg^D_{Var} \]

that we use to store

\[ RMState_{Var}(D) = [Reg^D_{Var} \rightarrow D] \]

the values of variables appearing in a \textit{while} program;

(iii) the registers

\[ Reg^D_{IO} \quad \text{and} \quad Reg^B_{IO} \]

that we use to store

\[ RMState_{IO}(D) = [Reg^D_{IO} \rightarrow D] \times [Reg^B_{IO} \rightarrow B] \]

input and output data and Booleans; and

(iv) the registers

\[ Reg^D_{Temp} \quad \text{and} \quad Reg^B_{Temp} \]

that we use to store

\[ RMState_{Temp}(D) = [Reg^D_{Temp} \rightarrow D] \times [Reg^B_{Temp} \rightarrow B] \]

values on a temporary basis.

This gives us the set

\[ Reg = PC \cup Reg^D_{Var} \cup Reg^D_{IO} \cup Reg^B_{IO} \cup Reg^D_{Temp} \cup Reg^B_{Temp} \]

of registers, which we use to define the set

\[ RMState(D) = [PC \rightarrow \mathbb{N}] \times RMState_{Var}(D) \times RMState_{IO}(D) \times RMState_{Temp}(D) \]

of RM states.

**Retrieving Values**

An RM state

\[ \tau = (pc, \rho_{Var}, \rho_{IO}, \rho_{Temp}) \]

consists of a collection of stores.
(i) We use the register 

\[ \text{PC} \]

\[ p_c = \tau(\text{PC}) \]

of the program counter.

(ii) We use the store 

\[ \rho_{\text{var}} : \text{RMState}_{\text{var}}(D) \]

for the values 

\[ \rho_{\text{var}}(\text{Allocate}(x)) \in D \]

of the \textbf{while} program variables \( x \in \text{Var} \).

(iii) A store 

\[ \rho_{\text{IO}} : \text{RMState}_{\text{IO}}(D) \]

of input and output values consists of 

\[ \rho_{\text{IO}} = (\rho_{\text{IO}}^D, \rho_{\text{IO}}^B) \]

where 

\[ \rho_{\text{IO}}^D : [\text{Reg}_{\text{IO}}^D \rightarrow D] \]

is a store for data and 

\[ \rho_{\text{IO}}^B : [\text{Reg}_{\text{IO}}^D \rightarrow D] \]

is a store for Booleans.

We use \( \rho_{\text{IO}}^D \) to store the values 

\[ \rho_{\text{IO}}^D(\text{in}_i^D) \in D \]

of data input registers \( \text{in}_i^D \in \text{Reg}_{\text{IO}}^D \), and the values 

\[ \rho_{\text{IO}}^D(\text{out}_i^D) \in D \]

of data output registers \( \text{out}_i^D \in \text{Reg}_{\text{IO}}^D \).

We use \( \rho_{\text{IO}}^B \) to store the values 

\[ \rho_{\text{IO}}^B(\text{in}_i^B) \in B \]

of Boolean input registers \( \text{in}_i^B \in \text{Reg}_{\text{IO}}^B \), and the values 

\[ \rho_{\text{IO}}^B(\text{out}_i^B) \in B \]

of Boolean output registers \( \text{out}_i^B \in \text{Reg}_{\text{IO}}^B \).

(iv) A store 

\[ \rho_{\text{Temp}} : \text{RMState}_{\text{Temp}}(D) \]

of temporary working values consists of 

\[ \rho_{\text{Temp}} = (\rho_{\text{Temp}}^D, \rho_{\text{Temp}}^B) \]
where
\[ \rho^D_{\text{Temp}} : [\text{Reg}^D_{\text{Temp}} \rightarrow D] \]
is a store for data and
\[ \rho^B_{\text{Temp}} : [\text{Reg}^B_{\text{Temp}} \rightarrow D] \]
is a store for Booleans.

We use \[ \rho^D_{\text{Temp}} \] to store the values
\[ \rho^D_{\text{Temp}}(w^D_{k_i}) \in D \]
of data registers \[ w^D_{k_i} \in \text{Reg}^D_{\text{Temp}} \] during calculations. and \[ \rho^B_{\text{Temp}} \] to store the values
\[ \rho^B_{\text{Temp}}(w^B_{k_i}) \in B \]
of Boolean registers \[ w^B_{k_i} \in \text{Reg}^B_{\text{Temp}} \] during calculations.

We want to define what it means for \textbf{while} states to be equivalent to RM states.

### 17.5.3 State Equivalence

We shall define a function
\[ \text{Encode} : \text{State}(A) \rightarrow \text{RMState}(D) \]
so that
\[ \text{Encode}(\sigma) = (1, \rho^\text{var}, \rho^\text{to}, \rho^\text{Temp}) \]
gives an RM program state equivalent to \( \sigma \): in particular, we set the program counter to be 1 and we initialise the set \( \rho^\text{var} \subset \rho \) of register values so that the value of a variable in a \textbf{while} program state and the value of a register used to store the corresponding variable, contain equivalent values:
\[ \rho^\text{var}(\text{Allocate}(x)) = \text{Encode}_{\text{Data}}(\sigma(x)) \]

Thus, for any state \( \tau \in \text{RMState}(D) \) and any variable \( x \in \text{Var} \), we require that
\[ \text{Decode}(\tau)(x) = \text{Decode}(\tau(\text{Allocate}(x))). \]

We shall also define a function
\[ \text{Decode} : \text{RMState}(D) \rightarrow \text{State}(A) \]
so that
\[ \text{Decode}(\tau) = \sigma \]
gives an equivalent \textbf{while} program state. in particular, we are interested only in the correspondence of those registers of the RM state that we use to store the values of compiled \textbf{while} program variables:
\[ (\forall x \in \text{Var})[\tau(\text{Allocate}(x)) = \text{Encode}_{\text{Data}}(\sigma(x))] \Leftrightarrow \text{Decode}(\tau) = \sigma \]
for any state \( \sigma \in \text{State}(A), \tau \in \text{RMState}(D) \).

We can substitute a value for a variable on a decoded target state, or we can translate the state that results from substituting the equivalent value on the appropriate register on the target state:
Lemma (State Substitution Transfer) For any variable \(x \in \text{Var}\), values \(v \in A\) and \(d \in D\), and state \(\tau \in R\text{MState}(D)\),

(i) \((\text{Decode}(\tau))[v/x] = \text{Decode}(\tau[\text{Encode}_{\text{Data}}(v)/\text{Allocate}(x)])\).

(ii) \((\text{Decode}(\tau))[\text{Decode}_{\text{Data}}(d)/x] = \text{Decode}(\tau[d/\text{Allocate}(x)])\).

Proof (i) Choose a variable \(y \in \text{Var}\). Then by the definition of the substitution function on while states,

\[
(\text{Decode}(\tau))[v/x](y) = \begin{cases} v & \text{if } x = y; \\ (\text{Decode}(\tau))(y) & \text{otherwise}. \end{cases}
\]

And by the definition of \(\text{Decode}\) on the evaluation function on states,

\[= \text{Decode}_{\text{Data}}(\tau(\text{Allocate}(y))).\]

As \(\text{Allocate}\) is an injective function, and by the definition of the substitution function on RM states,

\[= \text{Decode}_{\text{Data}}(\tau[\text{Encode}_{\text{Data}}(v)/\text{Allocate}(x)])(\text{Allocate}(y)).\]

(ii) Similarly. \(\square\)

Observe that \(\text{Decode}(\text{Encode}(\sigma)) = \sigma\)

but that

\[\text{Encode}(\text{Decode}(\tau)) = \tau \iff \tau(\text{PC}) = 1.\]

The execution of a compiled while program, is independent of the registers other than the program counter and those used to store variable values:

17.6 Defining the Compiler

Recall that while statements fall into two categories:

(i) atomic statements (skip and assignments);

(ii) composite statements constructed from simpler statements, that are concerned with flow-of-control (sequencing, conditionals and iteration).

When we compile the atomic statements, their compilation will be independent of any other statements. But because assignment statements are constructed from expressions, the compilation of an assignment statement \(x := e\) will be dependent on the compilation of the expression \(e\).

Thus, the RM program we produce for \(\text{Compile}(x := e)\) will be constructed from basic RM instructions and the RM program \(\text{Compile}_{\text{Exp}}(e)\) for the expression \(e\).

The constructed statements by their nature are dependent on other statements, so the compilation of a constructed statement will be dependent on the compilation of its component
17.6. DEFINING THE COMPILERS

statements. Thus, the compilation \( \text{Compile}(S_1; S_2) \) of a sequenced statement will depend on
the compilation \( \text{Compile}(S_1) \) and \( \text{Compile}(S_2) \) of its components \( S_1 \) and \( S_2 \).

Similarly, the compilation \( \text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}) \) of a conditional statement will depend on
the compilation \( \text{Compile}(S_1) \) and \( \text{Compile}(S_2) \) of its component statements, but in
addition will depend on the compilation \( \text{Compile}_{B \text{Exp}}(b) \) of its component Boolean expression
test \( b \).

And the compilation \( \text{Compile}(\text{while } b \text{ do } S_0 \text{ od}) \) of an iterative statement will depend on
the compilation \( \text{Compile}_{B \text{Exp}}(b) \) of its Boolean test \( b \) and the compilation \( \text{Compile}(S_0) \) of its
body \( S_0 \).

We will construct the compiled programs \( \text{Compile}(S) \) from the compilation of the con-
stituent elements of the statement \( S \) as described above. We join the compiled components by
sequencing them and basic RM program instructions together.

17.6.1 Compiling the Identity Statement

The identity, or

\[
\text{skip}
\]

statement simply requires that no variable values are altered by its execution. An algorithm
for the compiled code is:

1. Compare the value held in the Boolean register \( \text{out}^B_1 \) to itself.
2. Increment the program counter register by 1.

We compile the \( \text{skip} \) statement by:

\[
\text{Compile}(\text{skip}) = 1. \ +1 \leftarrow (\text{out}^B_1, \text{out}^B_1)
\]

This simply forces the program counter to be incremented by one, which will trigger the next
(if any) program instruction be executed. No registers other than the program counter are
updated, and it is simply the Boolean output register \( \text{out}^B_1 \) that is (artificially) consulted to
determine the execution of the instruction. Note that the value held in the register \( \text{out}^B_1 \) does
not affect the execution as is shown by the execution trace:

<table>
<thead>
<tr>
<th>PC</th>
<th>out(^B_1)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>program counter is always incremented</td>
</tr>
</tbody>
</table>

17.6.2 Compiling Assignment Statements

We compile an assignment statement

\[
x := e
\]

by compiling the expression \( e \), then setting the value of the register used for the variable \( x \). In
algorithmic terms, we:

1. Evaluate the expression \( e \), storing the value in the register \( \text{out}^D_1 \).
2. Place a copy of the value in the register \( \text{out}^D_1 \) into that allocated for
the variable \( x \).
Expressing this as an RM program, we get:

\[
\begin{align*}
\text{Compile}(x=e) &= \begin{cases} 
1. & \text{Compile}_{\text{Exp}}(e) \\
\alpha. & \\
\alpha + 1. & \text{Allocate}(x) \leftarrow \text{out}^D_1
\end{cases}
\end{align*}
\]

The execution trace of this code is:

<table>
<thead>
<tr>
<th>PC</th>
<th>out^D_1</th>
<th>Allocate(x)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>?</td>
<td>evaluate the expression for e</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>\alpha + 1</td>
<td>v</td>
<td>?</td>
<td>assign the result to the register for the variable x</td>
</tr>
<tr>
<td>\alpha + 2</td>
<td>v</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

17.6.3 Compiling Sequencing

We compile a sequenced statement

\[ S_1; S_2 \]

by placing the code for the compiled version of \( S_2 \) immediately following the code for the compiled version of \( S_1 \), and relabelling. Algorithmically, this gives us:

Execute the compiled code for the program \( S_1 \).
Then execute the compiled code for the program \( S_2 \) which has been relabelled to follow \( S_1 \).

Rendering this algorithm into RM code gives:

\[
\begin{align*}
\text{Compile}(S_1; S_2) &= \begin{cases} 
1. & \text{Compile}(S_1) \\
\gamma_1. & \\
\gamma_1 + 1. & \text{Compile}(S_2) \\
\vdots & \\
\gamma_1 + \gamma_2. & 
\end{cases}
\end{align*}
\]

The execution trace for such a program will depend on the nature of \( S_1 \) and \( S_2 \) as is shown in Figure 17.4. More precisely, because of the way in which we define our compiled programs, if the code for \( S_1 \) terminates, then it will do so with a value for the program counter that is the first instruction for the code for \( S_2 \). Otherwise, \( S_1 \) does not terminate. Similarly, if \( S_2 \) is reached, then if \( S_2 \) terminates, then it will do so with a value for the program counter that is one more than the last instruction for the compiled code (i.e., the length of \( S_1 + \) the length of \( S_2 + 1 \)). Alternatively, \( S_2 \) is reached, but may not terminate.
17.6. DEFINING THE COMPILER

![Diagram](image.png)

Figure 17.4: Execution of compiled sequenced statements.

17.6.4 Compiling Conditionals

We compile a conditional statement

```
if b then S₁ else S₂ fi
```

by placing jump instructions around the compiled forms of \( b \), \( S₁ \) and \( S₂ \) to direct the flow of control. First, we evaluate the compiled code for the Boolean expression \( b \). Then, we check the output of this against a Boolean output register which we have set to be false. If these two values are the same, we jump to the compiled code for the else-statement \( S₂ \). Otherwise, the compiled code for the then-statement \( S₁ \) is automatically executed. We do though have to ensure that after we have executed \( S₁ \) that we set the program counter to skip over the compiled code for \( S₂ \). In algorithmic terms, we get:

1. Evaluate the Boolean expression \( b \).
2. Set a test register to false.
3. Compare the result of the compiled code for \( b \) and the test register.
   - If they are the same then jump over the compiled code for \( S₁ \) to execute the compiled code for the else-statement \( S₂ \).
   - If they differ, i.e., \( b \) evaluated to true, then execute the compiled code for the then-statement \( S₁ \) and exit the compiled code by jumping over the compiled code for \( S₂ \).
Writing this algorithm as an RM program, we get:

\[
\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}) = \begin{cases} 
1. & \text{Compile}_{BExp}(b) \\
\beta. & \text{Compile}(S_1) \\
\beta + 1. & \text{out}_2^B \leftarrow \text{ff} \\
\beta + 2. & +(\gamma_1 + 2) \leftarrow (\text{out}_1^B, \text{out}_2^B) \\
\beta + 3. & \text{Compile}(S_2) \\
\vdots & \\
\beta + \gamma_1 + 2. & \text{Compile}(S_1) \\
\beta + \gamma_1 + 3. & +(\gamma_2 + 1) \leftarrow (\text{out}_1^B, \text{out}_1^B) \\
\beta + \gamma_1 + 4. & \text{Compile}(S_2) \\
\vdots & \\
\beta + \gamma_1 + \gamma_2 + 3. & \\
\end{cases}
\]

The execution trace for the code is shown in Figure 17.5. Again, the situation is complicated a little by the fact that we cannot guarantee the termination of either sub-statement \(S_1\) or \(S_2\).

\[
\begin{array}{|c|c|c|}
\hline
PC & \text{out}_1^B & \text{out}_2^B \\
\hline
1 & ? & ? \\
\hline
\end{array}
\]

**Execution of Compile\textsubscript{BExp}(b)**

\[
\begin{array}{|c|c|c|}
\hline
\beta + 1 & tt & ? \\
\beta + 2 & tt & ff \\
\beta + 3 & tt & ff \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|}
\hline
\beta + 1 & ff & ? \\
\beta + 2 & ff & ff \\
\beta + \gamma_1 + 4 & ff & ff \\
\hline
\end{array}
\]

**Execution of Compile(S_1)**

\[
\begin{array}{|c|c|c|}
\hline
\beta + \gamma_1 + 3 & ? & ? \\
\beta + \gamma_1 + \gamma_2 + 4 & ? & ? \\
\hline
\end{array}
\] Infinite loop

**Execution of Compile(S_2)**

\[
\begin{array}{|c|c|c|}
\hline
\beta + \gamma_1 + \gamma_2 + 4 & ? & ? \\
\hline
\end{array}
\] Infinite loop

\[\text{Figure 17.5: Execution of compiled conditional statements.}\]
17.6. DEFINING THE COMPILER

17.6.5 Compiling Iterative Statements

The cause of potential non-terminating programs is, of course, iterative statements of the form:

\[ \textbf{while } b \textbf{ do } S_0 \textbf{ od} \]

We compile such statements by using jump statements to effect the flow-of-control around the compiled Boolean expression for \( b \), and the compiled statement for \( S_0 \).

The algorithm for the compiled code is:

1. Evaluate the Boolean expression \( b \).
2. Set a test register to false.
3. Compare the result of the compiled code for \( b \) and the test register.
4. If they are the same, exit this code.
5. If they differ, i.e., \( b \) evaluated to true, then execute the compiled code for the body statement \( S_0 \).
6. Repeat the code from the start.

We implement this algorithm by:

\[
\text{Compile}(\textbf{while } b \textbf{ do } S_0 \textbf{ od}) = \begin{cases} 
\beta & 1. \\
\vdots & \vdots \\
\beta + 1. & \text{Compile}_{\text{EXP}}(b) \\
\beta + 2. & \text{out}_2^B \leftarrow \text{ff} \\
\beta + 3. & (\gamma_0 + 2) \leftarrow (\text{out}_1^B, \text{out}_2^B) \\
\vdots & \text{Compile}(S_0) \\
\beta + \gamma_0 + 2. & \text{out}_2^B \leftarrow (\text{out}_1^B, \text{out}_2^B) \\
\beta + \gamma_0 + 3. & 
\end{cases}
\]

The execution trace for this code is shown in Figure 17.6. Note the backwards loop in the code that sets up the potentially infinite repetition. Note also that this code may fail to terminate, either because of non-termination in this \textbf{while} loop, or because of non-termination in the sub-statement \( S_0 \) (due to a nested \textbf{while} loop not terminating). If the code does terminate though, then it will do so with the value of the program counter set to one more than the last instruction.
Figure 17.6: Execution of compiled iterative statements.
17.6.6  Summary of the Compiler
**Basis**

\[
\text{Compile}(\text{skip}) = \begin{array}{l}
1. \quad +1 \leftarrow (out_{1}^{B}, out_{2}^{B}) \\
\end{array}
\]

\[
\begin{array}{l}
1. \\
\end{array}
\]

\[
\text{Compile}(x := e) = \begin{array}{l}
\vdots \\
a. \end{array}
\]

\[
\begin{array}{l}
\text{Compile}_{\text{Exp}}(e) \\
\alpha + 1. \\
\end{array}
\]

\[
\begin{array}{l}
\text{Allocate}(x) \leftarrow out_{1}^{D} \\
\end{array}
\]

**Induction Step**

\[
\text{Compile}(S_{1} ; S_{2}) = \begin{array}{l}
\vdots \\
\gamma_{1}. \\
\gamma_{1} + 1. \end{array}
\]

\[
\begin{array}{l}
\text{Compile}(S_{1}) \\
\gamma_{1} + \gamma_{2}. \\
\end{array}
\]

\[
\text{Compile}(S_{2})
\]

\[
\text{Compile(}\text{if } b \text{ then } S_{1} \text{ else } S_{2} \text{ fi} ) = \begin{array}{l}
\vdots \\
\beta. \\
\beta + 1. \end{array}
\]

\[
\begin{array}{l}
out_{2}^{B} \leftarrow \text{ff} \\
\beta + 2. \\
\beta + 3. \\
\vdots \\
\beta + \gamma_{1} + 2. \\
\beta + \gamma_{1} + 3. \\
\beta + \gamma_{1} + 4. \\
\vdots \\
\beta + \gamma_{1} + \gamma_{2} + 3. \\
\end{array}
\]

\[
\begin{array}{l}
\text{Compile}(S_{1}) \\
\beta + \gamma_{0} + 2. \\
\beta + \gamma_{0} + 3. \\
\end{array}
\]

\[
\begin{array}{l}
\text{Compile}(S_{2}) \\
\beta + \gamma_{0} \\
\end{array}
\]

\[
\text{Compile}_{\text{Exp}}(b)
\]

\[
\begin{array}{l}
\beta. \\
\beta + 1. \\
\beta + 2. \\
\beta + 3. \\
\vdots \\
\beta + \gamma_{0} + 2. \\
\beta + \gamma_{0} + 3. \\
\end{array}
\]

\[
\begin{array}{l}
\text{Compile}(S_{0}) \\
\beta + \gamma_{0} \\
\end{array}
\]

\[
\text{Compile}_{\text{Exp}}(b)
\]

\[
\begin{array}{l}
\beta. \\
\beta + 1. \\
\beta + 2. \\
\beta + 3. \\
\vdots \\
\beta + \gamma_{0} + 2. \\
\beta + \gamma_{0} + 3. \\
\end{array}
\]

\[
\begin{array}{l}
\text{Compile}(S_{0}) \\
\beta + \gamma_{0} \\
\end{array}
\]

\[
\text{Compile}_{\text{Exp}}(b)
\]
17.6. DEFINING THE COMPILER

17.6.7 Constructing Compiled Programs

We can define the compiled while programs more precisely by introducing an operation

\[ \text{Join} : \text{RMProg} \times \text{RMProg} \rightarrow \text{RMProg} \]

such that

\[ \text{Join}(P_1, P_2) \]

is the RM program created by joining together the RM programs \( P_1, P_2 \in \text{RMProg} \). The program \( P_1 \) forms the first part of \( \text{Join}(P_1, P_2) \) and the relabelled program \( P_2 \) the remainder. The relabelling process renumbers the labels of \( P_2 \)'s instructions so that the labels of \( \text{Join}(P_1, P_2) \) are contiguous.

Using \( \text{Join} \), we can define \( \text{Compile} \) by:

\[
\begin{align*}
\text{Compile}(\text{skip}) &= 1. + 1 \leftarrow (\text{out}_1^B, \text{out}_1^B) \\
\text{Compile}(x:=e) &= \text{Join}(\text{Compile}_{\text{Exp}}(e), 1.\text{Allocate}(x) \leftarrow \text{out}_1^D) \\
\text{Compile}(S_1;S_2) &= \text{Join}(\text{Compile}(S_1), \text{Compile}(S_2)) \\
\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}) &= \text{Join}(\text{Compile}_{\text{Exp}}(b), \\
& \quad \text{Join}(1.\text{out}_2^B \leftarrow \text{ff}) \\
& \quad 2. + (|\text{Compile}(S_1)| + 1) \leftarrow (\text{out}_1^B, \text{out}_2^B), \\
& \quad \text{Join}(\text{Compile}(S_1), \\
& \quad \text{Join}(1. + |\text{Compile}(S_2)| \leftarrow (\text{out}_1^B, \text{out}_2^B), \\
& \quad \text{Compile}(S_2)))) \\
\text{Compile}(\text{while } b \text{ do } S_0 \text{ od}) &= \text{Join}(\text{Compile}_{\text{Exp}}(b), \\
& \quad \text{Join}(1.\text{out}_2^B \leftarrow \text{ff}) \\
& \quad 2. + (|\text{Compile}(S_0)| + 1) \leftarrow (\text{out}_1^B, \text{out}_2^B) \\
& \quad \text{Join}(\text{Compile}(S_0), \\
& \quad 1. - (|\text{Compile}_{\text{Exp}}(b)| + |\text{Compile}(S_2)|) \leftarrow (\text{out}_1^B, \text{out}_2^B))))
\end{align*}
\]

17.6.8 Compiled Statements Behaviour

The register programs that result from compiling while statements reflect the structured nature of the source while programs. Accordingly, they have some useful properties which are not true of RM programs in general.

Observation (Sequential Labelling) The labels of the instructions in a compiled while program \( \text{Compile}(S) \) are consecutive, starting at 1 and ending at \( |\text{Compile}(S)| \).

Observation (Restricted Jumps) Providing that any jump instructions in compiled expressions or Boolean expressions only set the program counter to a value within the expression, then any jump instructions in a compiled while program \( \text{Compile}(S) \) can only set the program counter to a value between 1 and \( |\text{Compile}(S)| + 1 \).

The value of the program counter after executing a terminating compiled while program \( \text{Compile}(S) \), will be \( |\text{Compile}(S)| + 1 \):
Lemma (Controlled Exiting) For all while programs $S$, and for all RM states $(1, \rho), \tau \in RMState$,

$$M^{\omega}_{RM(D)}(\text{Compile}(S))((1, \rho)) \downarrow \tau \Rightarrow \tau(\text{PC}) = |\text{Compile}(S)| + 1.$$ 

Proof By the Sequential Labelling Observation, the Restricted Jumps Observation and the Sequential Instruction Lemma of Chapter 16.

Lemma (Subprogram Execution) For all while programs $S$ and any RM programs $P, P_1, P_2$ and RM state $\tau$,

(i) $\tau(\text{PC}) = 1 \Rightarrow$

$$M^{\omega}_{RM(D)}(\text{Join}(\text{Compile}(S), P))(\tau) = M^{\omega}_{RM(D)}(\text{Join}(\text{Compile}(S), P)), (M^{\omega}_{RM(D)}(\text{Compile}(S))(\tau));$$

(ii) $\tau(\text{PC}) = \text{Last}(P) + 1 \Rightarrow$

$$M^{\omega}_{RM(D)}(\text{Join}(P, \text{Compile}(S)))(\tau) = M^{\omega}_{RM(D)}(\text{Join}(P, \text{Compile}(S))), (M^{\omega}_{RM(D)}(\text{Compile}(S))(\tau[1/\text{PC}])[\text{Last}(P) + |\text{Compile}(S)| + 1/\text{PC}]);$$

and

(iii) $\tau(\text{PC}) = \text{Last}(P_1) + 1 \Rightarrow$

$$M^{\omega}_{RM(D)}(\text{Join}(P_1, \text{Join}(\text{Compile}(S), P_2)))(\tau) = M^{\omega}_{RM(D)}(\text{Join}(P_1, \text{Join}(\text{Compile}(S), P_2))), (M^{\omega}_{RM(D)}(\text{Compile}(S))(\tau[1/\text{PC}])[\text{Last}(P_1) + |\text{Compile}(S)| + 1/\text{PC}]).$$

Proof

(i) As the value of the program counter is 1, the first instruction that will be executed will come from Compile$(S)$. Whilst the value of the program counter lies within 1 and $|\text{Compile}(P)|$, the instructions that are executed will come from Compile$(P)$.

But, by the Controlled Exiting Lemma, we know that the value of the program counter can only be set to $|\text{Compile}(P)| + 1$ when the program Compile$(P)$ terminates.

So, by induction on the length of programs, we can deduce that once the program counter is set to 1, the only instructions that we can execute will be from Compile$(P)$ until Compile$(P)$ terminates.

(ii) As the program counter is set to Last$(P) + 1$, by the definition of Join, the first instruction we will execute will be the first of Relabel(Last$(P) + 1, \text{Compile}(P)$).

By the semantics of RM programs, the execution of any non-jump instruction in Compile$(P)$ can only increment the program counter. Jump instructions can decrement the program counter, but by the Restricted Jumps Observation, in the program Compile$(P)$, they can only decrement the program counter to a value which lies within Compile$(P)$. 
17.7 Correctness of Compilation: Statements

So, by induction on the length of programs, we can deduce that once the program counter is set to $\text{Start}(\text{Compile}(P))$ the only instructions that we can execute will be from $\text{Compile}(P)$.

Thus, by the Relabelling Invariance Lemma, once the program counter is set to $\text{Last}(P) + 1$, the only instructions that we can execute will be from $\text{Relabel}(\text{Last}(P) + 1, \text{Compile}(P))$.

(iii) Follows from the previous two cases.

\[\Box\]

17.7 Correctness of Compilation: Statements

We shall consider the correctness of the compilation of statements with as much independence from the compilation of expressions and Boolean expressions as is possible. However, we shall need to suppose the compiler satisfies certain behavioural requirements.

17.7.1 Requirements for Compiled Expressions

We shall prove that our compiler from while statements to Register Machine programs is correct, providing that we have a correct compiler for expressions and Boolean expressions.

**Definition (Correct Expression Compilation)** A compiler

$\text{Compile}_{\text{Exp}} : \text{Exp}(\Sigma) \rightarrow \text{RMProg}(\Sigma)$

will be correct for any expression $e \in \text{Exp}(\Sigma)$ if, some value equivalent to the value

$V_{i_0}^A(e)(\sigma)$

of the expression $e$ on the state $\sigma$ is stored in the state that results from executing the program $\text{Compile}_{\text{Exp}}(e)$ on a state $\text{Encode}(\sigma)$ equivalent to $\sigma$.

In particular, we want the evaluation of the expression stored in the register $\text{out}_1^D$ that has been allocated expressly for this purpose.

Just as importantly, the execution of the program $\text{Compile}_{\text{Exp}}(e)$ should not alter the values of any variable-storing registers nor those of any of the designated input/output registers.

The program $\text{Compile}_{\text{Exp}}(e)$ should terminate and should do so in a controlled manner, i.e., the value of the program counter should be set at a value $|\text{Compile}_{\text{Exp}}(e)| + 1$ of one more than the length of the program.

**Lemma (Compiled Expression Requirements)** For all expressions $e \in \text{Exp}(\Sigma)$ and all states $\sigma \in \text{State}(A)$, a compiler $\text{Compile}$ satisfies the Compiled Expression Requirements if it satisfies all of the following:

(i) $\text{Decode}_{\text{Data}}(M^D_{i_0}(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))(\text{out}_1^D)) = V_{i_0}^A(e)(\sigma)$;

(ii) $\text{Decode}(M^D_{i_0}(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))) = \sigma$;

(iii) $r \in \text{Regio} \land r \neq \text{out}_1^D \Rightarrow M^D_{i_0}(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))(r) = r$; and

(iv) $\text{Eval}^{\text{PC}}(M^D_{i_0}(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))) = |e| + 1$. 


17.7.2 Requirements for Compiled Boolean Expressions

A compiler

\[ \text{Compile}_{\text{BE}} : BExp(\Sigma) \rightarrow RMProg(\Sigma) \]

will be correct for any Boolean expression \( b \in BExp(\Sigma) \) if, some value equivalent to the value

\[ W^A_{io}(b)(\sigma) \]

of the Boolean expression \( b \) on the state \( \sigma \) is stored in the state that results from executing the program \( \text{Compile}_{\text{BE}}(b) \) on a state \( \text{Encode}(\sigma) \) equivalent to \( \sigma \). In particular, we want the evaluation of the expression stored in the register

\[ \text{out}_{1}^{B} \]

that has been allocated expressly for this purpose.

As for expressions, we require that the execution of the program \( \text{Compile}_{\text{BE}}(b) \) should not alter the values of any variable-storing registers, nor those of any of the designated input/output registers.

The program \( \text{Compile}_{\text{BE}}(b) \) should terminate and should do so in a controlled manner, i.e., the value of the program counter should be set at a value \( |\text{Compile}_{\text{BE}}(b)| + 1 \) of one more than the length of the program.

**Lemma (Compiled Boolean Expression Requirements)** For all Boolean expressions \( b \in BExp(\Sigma) \) and all states \( \sigma \in \text{State}(A) \),

(i) \( \text{Decode}_{\text{Data}}(M^D_{io}(\text{Compile}_{\text{BE}}(b))(\text{Encode}(\sigma))(\text{out}_{1}^{B})) = W^A_{io}(b)(\sigma) \);

(ii) \( \text{Decode}(M^D_{io}(\text{Compile}_{\text{BE}}(b))(\text{Encode}(\sigma))) = \sigma \);

(iii) \( r \in \text{Reg}_{io} \land r \neq \text{out}_{1}^{B} \Rightarrow M^D_{io}(\text{Compile}_{\text{BE}}(b))(\text{Encode}(\sigma))(r) = r \); and

(iv) \( \text{Eval}^{PC}(M^D_{io}(\text{Compile}_{\text{BE}}(b))(\text{Encode}(\sigma))) = |b| + 1 \).

17.7.3 Correctness of Statements

A compiler

\[ \text{Compile} : Stmt(\Sigma) \rightarrow RMProg(\Sigma) \]

will be correct for any statement \( S \in Stmt(\Sigma) \) if, the execution of the compiled statement \( \text{Compile}(S) \) on a state \( \text{Encode}(\sigma) \) produces a state

\[ M^D_{io}(\text{Compile}(S))(\text{Encode}(\sigma)) \]

that is equivalent to the execution

\[ M^A_{io}(S)(\sigma) \]

of the statement \( S \) on an equivalent initial state \( \sigma \).

**Theorem (Compiler Correctness)** If a compiler \( \text{Compile} \) satisfies the Compiled Expression Requirements and the Compiled Boolean Expression Requirements Lemma, then for all \textbf{while} programs \( S \in WP(\Sigma) \) and all states \( \sigma \in \text{State}(A) \),

\[ M^A_{io}(S)(\sigma) \simeq \text{Decode}(M^D_{io}(\text{Compile}(S))(\text{Encode}(\sigma))). \]
17.7. CORRECTNESS OF COMPILATION: STATEMENTS

Proof We shall prove the Correctness Equation by structural induction on while programs. Let \( \sigma \in \text{State}(A) \) be some arbitrary while state and let

\[
\text{Encode}(\sigma) = (1, \rho)
\]

for some RM state \( \rho \in \text{RM State}(D) \).

Basis

Skip Statement By definition of Compile and Encode,

\[
\text{Decode}(M^D_{i_0}(\text{Compile(skip)})(\text{Encode}(\sigma))) = \text{Decode}(M^D_{i_0}(1. + 1 \leftarrow (\text{out}^B_1, \text{out}^B_1)((1, \rho)))).
\]

The behaviour of this instruction will always just increment the value of the program counter by one. More formally, by the Unconditional Jump Behaviour Lemma,

\[
= \text{Decode}((2, \rho)).
\]

The decode function is independent of the value of the program counter, so by the definition of Decode,

\[
= \text{Decode}((1, \rho)).
\]

But this is just our initial state

\[
= \sigma
\]

by the definition of Decode. In turn, this is the same state that we get by executing the program skip. Thus, by the semantics \( M^A_{i_0} \) of while programs,

\[
= M^A_{i_0}(\text{skip})(\sigma).
\]

Assignment Statements By definition of Compile and Encode,

\[
\text{Decode}(M^D_{i_0}(\text{Compile}(x := e))(\text{Encode}(\sigma))) = \text{Decode}(M^D_{i_0}(\text{Compile}_{\text{exp}}(e), \alpha + 1. \text{Allocate}(x) \leftarrow \text{out}^D((1, \rho)))).
\]

The first instruction we execute is the first of \( \text{Compile}_{\text{exp}}(e) \) as the program counter is 1. By the Compiled Expression Requirements Lemma, we know that executing the program \( \text{Compile}_{\text{exp}}(e) \) is guaranteed to terminate, in a state where the value of the program counter is \( \gamma + 1 \). So by induction on the length of programs, we know that the value of the program counter must lie between 1 and \( \gamma + 1 \) during the execution of \( e \). Thus,

\[
= \text{Decode}(M^D_{i_0}(\text{Compile}(x := e))(M^D_{i_0}(\text{Compile}_{\text{exp}}(e)((1, \rho))))).
\]

By the Compiled Expression Requirements Lemma, we know that executing the program \( \text{Compile}_{\text{exp}}(e) \) will guarantee to exit in a controlled manner with the evaluation of the expression \( e \) stored in the output register \( \text{out}^D_1 \). We can tolerate other register values changing, provided that these registers are not variable-storing.

Let us suppose that the execution of \( \text{Compile}_{\text{exp}}(e) \) on the state \( (1, \rho) \) produces some state \( (\alpha + 1, \rho') \) satisfying the above conditions. Thus, by the Compiled Expression Requirements Lemma,

\[
= \text{Decode}(M^D_{i_0}(\text{Compile}(x := e))((\alpha + 1, \rho))).
\]
By the Single Instruction Execution Lemma,

\[ D_{\text{Decode}}(M_{I_0}^{D}(\text{Compile}(x:=e))(M_{I_0}^{D}(\alpha + 1.\text{Allocate}(x) \leftarrow \text{out}^{D}))(\alpha + 1, \rho')) \]

Executing the instruction \( \alpha + 1.\text{Allocate}(x) \leftarrow \text{out}^{D} \) copies the value held in the register \( \text{out}^{D} \) to the register \( \text{Allocate}(x) \), and increments the program counter by one. Thus, by the definition of \( M_{I_0}^{D} \)

\[ = D_{\text{Decode}}(M_{I_0}^{D}(\text{Compile}(x:=e))(\alpha + 1, \rho')[[\rho'(PC) + 1][\rho'(\text{out}^{D})/\text{Allocate}(x)])] \]

The value of the program counter on the state \( \rho' \) is \( \alpha + 1 \). So, by the Register Substitution Lemma,

\[ = D_{\text{Decode}}(M_{I_0}^{D}(\text{Compile}(x:=e))(\alpha + 2, \rho')[\rho'(\text{out}^{D})/\text{Allocate}(x)])] \]

The program counter now exceeds all the instruction labels, so the program has terminated. Thus, by the Execution Cessation Lemma,

\[ = D_{\text{Decode}}((\alpha + 2, \rho')[\rho'(\text{out}^{D})/\text{Allocate}(x)])] \]

By the Compiled Expression Requirements Lemma, the evaluation of the expression \( e \) is not allowed to alter any variable-storing registers. The decode function is only dependent on variable-storing registers, so by the Compiled Expression Execution Lemma and the definition of \( D_{\text{Decode}} \),

\[ = D_{\text{Decode}}((1, \rho)[\rho'(\text{out}^{D})/\text{Allocate}(x)])] \]

Expanding the definition of \( \rho'(\text{out}^{D}) \):

\[ = D_{\text{Decode}}((1, \rho)[M_{I_0}^{D}(\text{Compile}_{Exp}(e))(\text{Encode}(\sigma))(\text{out}^{D})/\text{Allocate}(x)])] \]

\[ = (D_{\text{Decode}}((1, \rho))[D_{\text{Decode}}_{data}(M_{I_0}^{D}(\text{Compile}_{Exp}(e))(\text{Encode}(\sigma))(\text{out}^{D}))/\text{Allocate}(x)])] \]

Using the Compiled Expression Requirements Lemma,

\[ = D_{\text{Decode}}((1, \rho)[\text{Encode}_{Data}(V_{I_0}^{A}(e)(\sigma))/\text{Allocate}(x)])] \]

By the State Substitution Transfer Lemma,

\[ = D_{\text{Decode}}((1, \rho)[V_{I_0}^{A}(e)(\sigma)/x]) \]

And decoding the state \( (1, \rho) \),

\[ \sigma[V_{I_0}^{A}(e)(\sigma)/x] \]

But this is the semantics of the \textbf{while} assignment statement \( x:=e \). So, by the definition of \( M_{I_0}^{A} \),

\[ = M_{I_0}^{A}(x:=e)(\sigma) \]

\section*{Induction Step}

Suppose that our correctness statement holds for programs \( S_0 \), \( S_1 \) and \( S_2 \).
17.7. CORRECTNESS OF COMPILATION: STATEMENTS

Sequenced Statements  By the definition of Encode and Compile,

\[
\text{Decode}(M^D_{\rho}(\text{Compile}(S_1;S_2))(\text{Encode}(\sigma))) = \text{Decode}(M^D_{\rho}(\text{Join}(\text{Compile}(S_1), \text{Compile}(S_2)))(1, \rho))).
\]

By the Subprogram Execution Lemma, we can split the execution of \(\text{Compile}(S_1;S_2)\) up:

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_1;S_2))(M^D_{\rho}(\text{Compile}(S_1))((1, \rho)))).
\]

Case: Compile\((S_1)\) terminates If \(\text{Compile}(S_1)\) terminates, then by the Controlled Exiting Lemma, we know that it will do so with the program counter set at one more than the length of \(\text{Compile}(S_1)\). Suppose that \(M^D_{\rho}(\text{Compile}(S_1))(\rho) \downarrow (\gamma_1 + 1, \rho_1)\):

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_1;S_2))((\gamma_1 + 1, \rho_1))).
\]

By the Subprogram Execution Lemma, we can again split this execution up:

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_1;S_2))(M^D_{\rho}(\text{Compile}(S_2))((\gamma_1 + 1, \rho_1)[PC/1][\gamma_1 + \gamma_2 + 1/PC])).
\]

By the Execution Cessation Lemma,

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))((\gamma_1 + 1, \rho_1)[1/PC][\gamma_1 + \gamma_2 + 1/PC]).
\]

And now if we expand the state \((\gamma_1 + 1, \rho_1)\), we get

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))(M^D_{\rho}(\text{Compile}(S_1))((1, \rho))[1/PC][\gamma_1 + \gamma_2 + 1/PC]).
\]

But decoding is independent of the program counter, so by the definition of Decode,

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))(M^D_{\rho}(\text{Compile}(S_1))((1, \rho))[1/PC])).
\]

But the state \((1, \rho)\) is the encoding of the initial state:

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))(M^D_{\rho}(\text{Compile}(S_1))(\text{Encode}(\sigma))[1/PC])).
\]

As we reset the program counter to 1 after executing \(\text{Compile}(S_1)\), we can rewrite using the definition of Encode and Decode,

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))(\text{Encode}(\text{Decode}(M^D_{\rho}(\text{Compile}(S_1))(\text{Encode}(\sigma)))))).
\]

By Induction Hypothesis on \(\text{Compile}(S_1)\),

\[
= \text{Decode}(M^D_{\rho}(\text{Compile}(S_2))(\text{Encode}(M^A_{\rho}(S_1)(\sigma)))).
\]

And now by induction hypothesis on \(\text{Compile}(S_2)\),

\[
\simeq M^A_{\rho}(S_2)(M^A_{\rho}(S_1)(\sigma)).
\]

This is just the semantics of the sequenced while program \(S_1;S_2\), so by the definition of \(M^A_{\rho}\),

\[
= M^A_{\rho}(S_1;S_2)(\sigma).
\]
Case: Compile($S_1$) does not terminate If Compile($S_1$) does not terminate, then execution of the whole does not terminate, which is what results from $M^A_{io}(S_1;S_2)$ also.

Conditional Statements By definition of Encode and Compile,

\[
\begin{align*}
\text{Decode}(M^D_{io}(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(\text{Encode}(\sigma))) \\
= \text{Decode}(M^D_{io}(1. \\
\vdots \\
\beta. \\
\beta + 1. \quad \text{Compile}_{\text{Exp}}(b) \\
\beta + 2. \\
\beta + 3. \\
\vdots \\
\beta + \gamma_1 + 2. \\
\beta + \gamma_1 + 3. \\
\beta + \gamma_1 + 4. \\
\vdots \\
\beta + \gamma_1 + \gamma_2 + 3. \\
)((1, \rho))
\end{align*}
\]

As the program counter is set to 1, the first instruction that will be executed will be from $b$. By the Compiled Boolean Expression Requirements Lemma, the execution of $b$ is guaranteed to terminate, in a state where the value of the program counter is $\beta + 1$. So by induction on the length of programs, we know that the value of the program counter must lie between 1 and $\beta + 1$ during the execution of $b$:

\[
= \text{Decode}(M^D_{io}(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(M^D_{io}(\text{Compile}(b))((1, \rho)))).
\]

By the Compiled Boolean Expression Requirements Lemma, we know that executing the program $\text{Compile}_{\text{Exp}}(b)$ will guarantee to exit in a controlled manner with the evaluation of the Boolean expression $b$ stored in the output register $\text{out}_1^B$. We can tolerate other register values changing, provided that these registers are not variable-storing.

Let us suppose that the execution of $\text{Compile}_{\text{Exp}}(b)$ on the state $(1, \rho)$ produces some state $(\beta + 1, \rho')$ satisfying the above conditions:

\[
= \text{Decode}(M^D_{io}(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))((\beta + 1, \rho'))).
\]

As the program counter is set to $\beta + 1$, the next instruction we execute is $\beta + 1.\text{out}_2^B \leftarrow \text{ff}$. So, by the Single Instruction Execution Lemma,

\[
= \text{Decode}(M^D_{io}(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(M^D_{io}(\beta + 1.\text{out}_2^B \leftarrow \text{ff})((\beta + 1, \rho')))),
\]

which by the definition of $M^D_{io}$,

\[
= \text{Decode}(M^D_{io}(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})))((\beta + 1, \rho')[(\beta + 1, \rho)(PC)/PC][\text{ff}/\text{out}_2^B]))
\]
and simplifying,

\[ = \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(\beta + 2, \rho)[ff/output^B_2])). \]

Now the value of the program counter is \( \beta + 2 \), so the next instruction we execute is \( \beta + 2 + (\gamma_1 + 2) \leftarrow (out^B_1, out^B_2) \). So, by the Single Instruction Execution Lemma,

\[ = \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})))
\]

\[ (M^D_1(\beta + 2 + (\gamma_1 + 1) \leftarrow (out^B_1, out^B_2))(\beta + 2, \rho)[ff/output^B_2])). \]

There are two possible outcomes for the jump instruction. Either the source comparison registers \( out^B_1 \) and \( out^B_2 \) hold the same value (i.e., the evaluation of the Boolean test \( \text{Compile}(b) \) is false), or they differ (i.e., the evaluation of the Boolean test \( \text{Compile}(b) \) is true).

**Case:** \( b \) evaluates to true. Suppose that the Boolean test \( b \) evaluates to true on the state \( \sigma \). Then, by the definition of \( M^D_1 \),

\[ \text{Encode}_{Data}(W^A(b)(\sigma)) = \text{tt} \]

\[ \Rightarrow \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(\text{Encode}(\sigma))) \]

\[ = \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}))(\beta + 3, \rho)[ff/output^B_2])). \]

Now the value of the program counter is set to \( \beta + 3 \), the first instruction of \( \text{Compile}(S_1) \). Thus, we can apply the Subprogram Execution Lemma,

\[ = \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})))
\]

\[ (M^D_1(\text{Compile}(S_1))(\beta + 3, \rho)[ff/output^B_2][(1/PC)][\beta + \gamma_1 + 3/PC]), \]

which by the Labelling Invariance Lemma is:

\[ = \text{Decode}(M^D_1(\text{Compile}(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi})))
\]

\[ (M^D_1(\text{Relabel}(\beta + 3, \text{Compile}(S_1))((\beta + 3, \rho)[ff/output^B_2)))) \]

The sub-program \( \text{Compile}(S_1) \) for the then-clause will either terminate or not. We can check for termination

\[ M^D_1(\text{Compile}(S_1))((1, \rho)) \downarrow \]

of \( \text{Compile}(S_1) \) on the state \((1, \rho)\) rather than

\[ M^D_1(\text{Compile}(S_1))((1, \rho)[ff/output^B_2]) \downarrow \]

because the registers with changed values are always set by compiled statements before being accessed:

- The evaluation of expressions and Boolean expressions are not allowed to alter the values of input/output registers other than \( out^D_1 \) and \( out^B_1 \), by the Expression Requirements Lemma and the Boolean Expression Requirements Lemma.

- The evaluation of expressions and Boolean expressions are not allowed to alter the values of variable-storing registers by the Expression Requirements Lemma and the Boolean Expression Requirements Lemma.
The register \( \text{out}_{2}^{B} \) is only used by compiled conditional and iteration statements. In both cases, the instruction to set the value is independent of any registers (it is only ever set to \texttt{false}) and it always occurs immediately prior to a test. As there are no jump instructions to the test itself, the instruction to set the value in \( \text{out}_{2}^{B} \) is always executed before the value is determined.

So, consider the case that \( \text{Compile}(S_{1}) \) terminates. By the Controlled Exiting Lemma, we know that if \( \text{Compile}(S_{1}) \) terminates, then it will do so with a value of \( \gamma_{1} + 1 \), and so by the Relabelling Invariance Lemma, we know that if \( \text{Relabel}(\beta + 3, \text{Compile}(S_{1})) \) terminates, it will do so with a value of \( \beta + \gamma_{1} + 3 \). Suppose then, that \( \text{Relabel}(\beta + 3, \text{Compile}(S_{1})) \) terminates in a state \((\beta + \gamma_{1} + 3, \rho_{1})\).

\[
\begin{align*}
\text{Decode}_{\text{Data}}(W^{A}(b)(\sigma)) &= tt \land M^{P}_{\text{io}}(\text{Compile}(S_{1}))(\text{Encode}(\sigma)) \downarrow \\
&\Rightarrow \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(\text{if } b \text{ then } S_{1} \text{ else } S_{2} \text{ fi}))(\text{Encode}(\sigma))) \\
&= \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(\text{if } b \text{ then } S_{1} \text{ else } S_{2} \text{ fi}))(\beta + \gamma_{1} + 3, \rho_{1}))
\end{align*}
\]

As the value of the program counter is \( \beta + \gamma_{1} + 3 \), the next instruction that we execute is \( \beta + \gamma_{1} + 3 + (\gamma_{2} + 1) \mapsto (\text{out}_{1}^{B}, \text{out}_{1}^{B}) \). So, by the Single Instruction Execution Lemma,

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(\text{if } b \text{ then } S_{1} \text{ else } S_{2} \text{ fi}))(\beta + \gamma_{1} + 3, \rho_{1})[\beta + \gamma_{1} + \gamma_{2} + 4/\text{PC}]).
\]

This terminates the program: by the Execution Cessation Lemma,

\[
= \text{Decode}((\beta + \gamma_{1} + 3, \rho_{1})[\beta + \gamma_{1} + \gamma_{2} + 4/\text{PC}]).
\]

As \text{decode} is independent of the value of the program counter, by the definition of \text{Decode},

\[
= \text{Decode}((\beta + \gamma_{1} + 3, \rho_{1})).
\]

Now substituting back for the state \((\beta + \gamma_{1} + 3, \rho_{1})\),

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Relabel}(\beta + 3, \text{Compile}(S_{1}))))((\beta + 3, \rho'))
\]

and substituting back for the state \((\beta + 3, \rho')\),

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Relabel}(\beta + 3, \text{Compile}(S_{1}))))(M^{P}_{\text{io}}(\text{Compile}(b))((1, \rho'))).
\]

But the state \((1, \rho)\) is the encoding of the initial state:

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Relabel}(\beta + 3, \text{Compile}(S_{1}))))(M^{P}_{\text{io}}(\text{Compile}(b))(\text{Encode}(\sigma))).
\]

By the Relabelling Invariance Lemma,

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(S_{1}))(M^{P}_{\text{io}}(\text{Compile}(b))(\text{Encode}(\sigma))))[1/\text{PC}].
\]

As we set the program counter back to 1 after evaluating the Boolean expression,

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(S_{1})))\text{Encode}(\text{Decode}(M^{P}_{\text{io}}(\text{Compile}(b))(\text{Encode}(\sigma))))).
\]

Now applying the Boolean Expression Requirements Lemma,

\[
= \text{Decode}(M^{P}_{\text{io}}(\text{Compile}(S_{1}))(\text{Encode}(\sigma))).
\]

And applying the Induction Hypothesis to \( S_{1} \),

\[
\simeq M^{A}_{\text{io}}(S_{1}, \sigma).
\]

But this is the semantics of the \texttt{while} program \texttt{if } b \texttt{ then } S_{1} \texttt{ else } S_{2} \texttt{ fi} when the Boolean expression evaluates to true. So, by the definition of \( M^{A}_{\text{io}} \),

\[
= M^{A}_{\text{io}}(\text{if } b \text{ then } S_{1} \text{ else } S_{2} \text{ fi})(\sigma).
\]
Case: Compile($S_1$) does not terminate Now consider the case that the then-clause Compile($S_1$) does not terminate:

$$
\text{Encode}_{\text{Data}}(W^A(b)(\sigma)) = tt \land M^D_{io}(\text{Compile}(S_1))(\text{Encode}(\sigma)) \uparrow \\
\Rightarrow \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(\text{Encode}(\sigma))) \uparrow 
$$

which is the semantics of $\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}.$

Case: $b$ evaluates to false Now, returning to the possibility that the else-clause is executed. This occurs when the values in the registers $out^B_1$ and $out^B_2$ contain the same values, so the jump instruction causes the program counter to be incremented to the location of the sub-program corresponding to $S_2$.

So by the definition of $M^D_{io}$,

$$
\text{Encode}_{\text{Data}}(W^A(b)(\sigma)) = ff \\
\Rightarrow \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(\text{Encode}(\sigma))) \\
= \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(\beta + \gamma_1 + 4, \rho')[ff/out^B_2])
$$

Now that the value of the program counter is $\beta + \gamma_1 + 4$, the instruction that is executed next is the first instruction of Compile($S_2$). So applying the Subprogram Execution Lemma and the Labelling Invariance Lemma, Thus,

$$
= \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(M^D_{io}(\text{Relabel}(\beta + \gamma_1 + 4, \text{Compile}(S_2)))(\beta + \gamma_1 + 4, \rho')[ff/out^B_2]))
$$

Again, there are now two cases to consider, depending on whether Compile($S_2$) terminates or not. We can check for termination

$$
M^D_{io}(\text{Compile}(S_2))((1, \rho)) \downarrow
$$

of Compile($S_2$) on the state $(1, \rho)$ rather than

$$
M^D_{io}(\text{Compile}(S_2))((1, \rho')[ff/out^B_2]) \downarrow
$$

for the same reasons outlined for determining the termination in the then-case.

First, suppose the else-clause Compile($S_2$) does terminate: By the Controlled Exiting and Execution Cessation Lemmas,

$$
\text{Encode}_{\text{Data}}(W^A(b)(\sigma)) = ff \land M^D_{io}(\text{Compile}(S_2))(\text{Encode}(\sigma)) \downarrow \\
\Rightarrow \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(\text{Encode}(\sigma))) \\
= \text{Decode}(M^D_{io}(\text{Relabel}(\beta + \gamma_1 + 4, \text{Compile}(S_2)))(\beta + \gamma_1 + 4, \rho')[ff/out^B_2])
$$

And then the result follows by similar reasoning for the then case.

And now suppose the else-clause Compile($S_2$) does not terminate:

$$
\text{Encode}_{\text{Data}}(W^A(b)(\sigma)) = ff \land M^D_{io}(\text{Compile}(S_2))(\text{Encode}(\sigma)) \uparrow \\
\Rightarrow \text{Decode}(M^D_{io}(\text{Compile}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}))(\text{Encode}(\sigma))) \uparrow
$$

which is the semantics of $\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}.$
Iterative Statements
The execution of the statement \texttt{while } b \texttt{ do } S_0 \texttt{ od} on a state \( \sigma \in \text{State}(A) \) produces the sequence \( \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots \) of states from the repeated execution of the body \( S_0 \) on the initial state \( \sigma_0 = \sigma \). The sequence of states produced may be infinite if either the execution of \( S_0 \) does not terminate, or if the controlling condition \( b \) of the \texttt{while} loop determines non-termination.

We need to show that the execution of the program

\[
\text{Compile(while } b \texttt{ do } S_0 \texttt{ od)}
\]

on a state \( \tau \in \text{RMState}(D) \) that is equivalent to \( \sigma \in \text{State}(A) \) produces an equivalent sequence \( \tau_0, \tau_1, \ldots, \tau_n, \ldots \) of states. I.e., we need to show that

\[
\text{if } Decode(\tau) = \sigma \text{ then } W^A(b)(\text{Decode}(\tau_i)) = tt \text{ for } 0 \leq i < n
\]

\[
W^A(b)(\text{Decode}(\tau_n)) = ff
\]

and \( \text{Decode}(\tau_i) = \sigma_i \).

Let us first examine the execution of compiled \texttt{while} statements. Let \( \tau = (1, \rho) \in \text{RMState}(D) \) be an arbitrary RM state. Then, following the same reasoning as for conditional statements:

\[
M^D_\rho(\text{Compile(while } b \texttt{ do } S_0 \texttt{ od)}(\tau) = M^D_\rho(\text{Compile(while } b \texttt{ do } S_0 \texttt{ od)}((\beta + 2, \rho)[ff/out^B_2])
\]

where the values of the variable-storing registers in \( \rho \) remain the same as they were in \( \rho \), the value in the output register \( \text{out}^D_1 \) contains the evaluation of the Boolean expression \( b \), and the program counter is set to \( |b| + 1 \). We shall use the state produced by executing the program \text{Compile}_{BE}\exp(b) \) for the evaluation of the Boolean expression as our state \( \tau_0 \):

\[
\tau_0 = (\beta + 2, \rho'[ff/out^B_2])
\]

This corresponds to the initial state \( \sigma_0 \):

\[
\text{Decode}(\tau_0) = \text{Decode}(\tau[ff/out^B_2][\beta + 2/PC])
\]

by definition of \( \tau_0 \)

\[
= \text{Decode}(\tau) \text{ by definition of } \text{Decode}
\]

\[
= \sigma \text{ by assumption}
\]

\[
= \sigma_0 \text{ by definition of } \sigma_0.
\]

Continuing with our execution, if the Boolean test evaluated to false, then

\[
\text{Encode}_{Data}(W^A(b)(\text{Decode}(\tau))) = ff \Rightarrow M^D_\rho(\text{Compile(while } b \texttt{ do } S_0 \texttt{ od)}(\tau)
\]

\[
= M^D_\rho(\text{Compile(while } b \texttt{ do } S_0 \texttt{ od))}
\]

\[
(M^D_\rho(\beta + 2, + (\gamma_0 + 2) \Rightarrow (\text{out}^B_1, \text{out}^B_2))((\beta + 2, \rho'[ff/out^B_2]))
\]

by the same reasoning as for conditional statements

\[
= M^D_\rho(\text{Compile(while } b \texttt{ do } S_0 \texttt{ od}))(\beta + \gamma_0 + 4, \rho'[ff/out^B_2])
\]

by definition of \( M^D_\rho \)

\[
= (\beta + \gamma_0 + 4, \rho'[ff/out^B_2])
\]

by the Execution Cessation Lemma.
17.7. CORRECTNESS OF COMPILATION: STATEMENTS

Thus, if

\[ \text{Encode}_{\text{Data}}(W^A(b)(\text{Decode}(\tau_n))) = \text{ff} \]

then execution of the compiled \texttt{while} statement will cease. But we can simplify this termination condition to:

\[
\text{Encode}_{\text{Data}}(W^A(b)(\text{Decode}(\tau_n))) = \text{ff} \\
\iff \text{Decode}_{\text{Data}}(\text{Encode}_{\text{Data}}(W^A(b)(\text{Decode}(\tau_n)))) = \text{Decode}_{\text{Data}}(\text{ff}) \\
\text{by applying \text{Decode}_{\text{Data}} to both sides} \\
\iff W^A(b)(\text{Decode}(\tau_n)) = \text{ff} \\
\text{by definition of \text{Decode}_{\text{Data}}} 
\]

as we required.

Now, we return to the execution, but considering the case where the Boolean test evaluates to true.

\[
W^D_i(\text{Compile}_{\text{BE}}(b))(\tau) = \text{tt} \implies M^D_{i_0}(\text{Compile(while } b \text{ do } S_0 \text{ od)}(\tau) \\
= M^D_{i_0}(\text{Compile(while } b \text{ do } S_0 \text{ od)} \\
(M^D_{i_0}(\text{Relabel}(\beta + 3, \text{Compile}(S_0))))((\beta + 3, \rho)[\text{ff/out}^B]) \\
\text{by the same reasoning as for conditional statements} 
\]

Using the same reasoning contained in the conditionals case, we can ask whether \text{Compile}(S_0) terminates on the state \( \tau \) or not. If it does not terminate, then

\[
W^D_i(\text{Compile}_{\text{BE}}(b))(\tau) = \text{tt} \land M^D_{i_0}(\text{Compile}(S_0))(\tau) \uparrow \\
\implies M^D_{i_0}(\text{Compile(while } b \text{ do } S_0 \text{ od)}(\tau) \uparrow .
\]

If it does terminate, then

\[
W^D_i(\text{Compile}_{\text{BE}}(b))(\tau) = \text{tt} \implies M^D_{i_0}(\text{Compile(while } b \text{ do } S_0 \text{ od)}(\tau) \\
= (\text{Compile(while } b \text{ do } S_0 \text{ od)} \\
(M^D_{i_0}(\beta + \gamma_0 + 3. - (\beta + \gamma_0 + 2) \leftarrow (\text{out}^B_1, \text{out}^B_1)) \\
(M^D_{i_0}(\text{Relabel}(\beta + 3, \text{Compile}(S_0))))((\beta + 3, \rho)[\text{ff/out}^B]) \\
\text{by the Execution Cessation and Single Instruction Execution Lemmas} \\
= M^D_{i_0}(\text{Compile(while } b \text{ do } S_0 \text{ od)} \\
(M^D_{i_0}(\text{Compile}(S_0))((1, \rho)[\text{ff/out}^B]))[1/\text{PC}] \\
\text{by arithmetic and the Relabelling Isomorphism Invariance Theorem} 
\]

This tells us how our states in the iteration sequence will be produced.

Claim For \( 0 \leq i < n \), the sequence of iterates is determined by

\[ \tau_{i+1} \simeq M^D_{i_0}(\text{Compile}(S_0))(\tau_i[\text{Encode}_{\text{Data}}(W^A(b)(\text{Decode}(\tau_i))/\text{out}^B_1)] [1/\text{PC}] . \]

Proof We prove the claim by induction.

For the base case \( i = 0 \),

\[ \tau_1 \simeq M^D_{i_0}(\text{Compile}(S_0))((1, \rho)[\text{ff/out}^B]) [1/\text{PC}] \\
\text{by the semantics of compiled \texttt{while} programs} \\
\simeq M^D_{i_0}(\text{Compile}(S_0))(\tau_0[\text{tt/out}^B]) [1/\text{PC}] \\
\text{by the definition of } \tau_0 \\
\simeq M^D_{i_0}(\text{Compile}(S_0))(\tau_0[\text{Encode}_{\text{Data}}(W^A(b)(\text{Decode}(\tau_0))/\text{out}^B_1)] [1/\text{PC}] \\
\text{by assumption} \]
For the Induction Hypothesis, we assume the case holds up to \( i \). Then, for the Induction Step, \( i + 1 \):

\[
M^D_\bullet (\texttt{Compile(while } b \texttt{ do } S_0 \texttt{ od)})(\tau) \\
\simeq \texttt{Compile(while } b \texttt{ do } S_0 \texttt{ od)}(\tau_i) \\
\quad \text{by induction} \\
\simeq (\texttt{Compile(while } b \texttt{ do } S_0 \texttt{ od)}) \\
\quad (M^D_\bullet (\texttt{Compile}(S_0))(\tau_i)[\texttt{ff }/\texttt{out}^B_2)]\{1/PC\} \\
\quad \text{by the semantics of compiled while statements} \\
\simeq (\texttt{Compile(while } b \texttt{ do } S_0 \texttt{ od)}) \\
\quad (M^D_\bullet (\texttt{Compile}(S_0))(\tau_i)[\texttt{Encode}_{Data}(W^A(b)(\texttt{Decode}(\tau_i))/\texttt{out}^B_1)]\{1/PC\} \\
\quad \text{by the definition of } \tau_i \\
\]

And finally, we examine the iteration condition

\[
\texttt{Encode}_{Data}(W^A(b)(\texttt{Decode}(\tau_i))) = tt
\]

which we can simplify to

\[
\texttt{Encode}_{Data}(W^A(b)(\texttt{Decode}(\tau_i))) = tt \\
\iff \texttt{Decode}_{Data}(\texttt{Encode}_{Data}(W^A(b)(\texttt{Decode}(\tau_i)))) = \texttt{Decode}_{Data}(tt) \\
\quad \text{by applying } \texttt{Decode}_{Data} \text{ to both sides} \\
\iff W^A(b)(\texttt{Decode}(\tau_i)) = tt \\
\quad \text{by definition of } \texttt{Decode}_{Data}
\]

as we required.

\[\square\]

17.8 Compiling Expressions into Register Machine Programs

17.8.1 The Compiler

Compiling Constants

We have already assumed that much of the hard work has already been performed for compiling constants: we assume for each constant \( c^A \) of the source algebra \( A \) that we have some equivalent constant \( c^D \) in the target algebra \( D \). We can set a register to the value \( c^D \) by taking the \( \Sigma \)-RM instruction \( c(r) \). In particular, we set the designated data-output register \( \texttt{out}^D_1 \) to \( c^D \) by:

\[
\texttt{Compile}_{Exp}(c) = 1. \quad \texttt{out}^D_1 \leftarrow c
\]

Thus, the execution of a compiled constant has the form:

<table>
<thead>
<tr>
<th>PC</th>
<th>( \texttt{out}^D_1 )</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>( c^D )</td>
<td>set the output register</td>
</tr>
</tbody>
</table>
17.8. COMPILING EXPRESSIONS INTO REGISTER MACHINE PROGRAMS 601

Compiling Variables

For each variable appearing in a **while** program, we shall designate a register to store any values it may contain during the execution of the compiled version. We assume we have a function

\[ \text{Allocate} : \text{Var} \rightarrow \mathbb{N} \]

such that

\[ \text{Allocate}(v) \] gives the index to the register for the variable \( v \).

The precise nature of \( \text{Allocate} \) is irrelevant for our purposes, but there are some general conditions that it should satisfy:

(i) \( \text{Allocate}(v) \) for any \( v \in \text{Var} \) should not coincide with any designated input or output registers, temporary storage registers, or with the program counter.

(ii) \( \text{Allocate} \) should be an injective function, so that different variables \( v \neq x \in \text{Var} \) should map to different registers \( \text{Allocate}(v) \neq \text{Allocate}(x) \).

So to set the designated data-output register \( \text{out}^D_1 \) to the value of a variable \( x \), we just copy its value over:

\[ \text{Compile}_{\text{exp}}(x) = 1. \quad \text{out}^D_1 \leftarrow \text{Allocate}(x) \]

Thus, the execution of a compiled constant has the form:

<table>
<thead>
<tr>
<th>PC</th>
<th>( \text{Allocate}(x) )</th>
<th>( \text{out}^D_1 )</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( v )</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>( v )</td>
<td>( v )</td>
<td>set the output register</td>
</tr>
</tbody>
</table>

Compiling Function Application

Again, we assume that for each function \( f^A \) of the source algebra \( A \) we are able to construct some equivalent function \( f^D \) in the target algebra \( D \).

We want to be able to compile expressions of the form

\[ f(e_1, \ldots, e_n) \]

where \( e_1, \ldots, e_n \) are nested sub-expressions. We recursively compile each of the sub-expressions, then apply the function \( f^D \) to the results.

We need to perform some book-keeping operations to ensure that everything proceeds smoothly. First, we need to transfer each of the results \( v_1, \ldots, v_n \in D \) of the evaluation of the sub-expressions into designated input registers \( \text{in}^D_1, \ldots, \text{in}^D_n \). We need to perform this process as we evaluate each sub-expression \( e_i \), as the output of expression evaluation is always placed in the designated output register \( \text{out}^D_1 \); we need to make sure the evaluation of a sub-expression \( e_i \) does not over-write the evaluation of a previous sub-expression \( e_{i-1} \).

We also need to ensure that when we store the intermediate results \( v_1, \ldots, v_n \) of the sub-expressions \( e_1, \ldots, e_n \), that we do not lose information from the designated input registers \( \text{in}^D_1, \ldots, \text{in}^D_n \). This could occur if we had one function nested inside another; e.g., in evaluating

\[ f(e_1, \ldots, e_{i-1}, g(e'_1, \ldots, e'_m), e_{i+1}, \ldots, e_n) \]
we would evaluate the expressions $e_1, \ldots, e_{i-1}$ and store their results $v_1, \ldots, v_{i-1}$ in the registers $in_1^D, \ldots, in_{i-1}^D$. Then, in evaluating the $i^{th}$ expression $e_i = g(e'_1, \ldots, e'_m)$, we would evaluate its sub-expressions $e'_1, \ldots, e'_m$, storing their results $v'_1, \ldots, v'_m$ in the registers $in_1^D, \ldots, in_m^D$, before evaluating $g$ on these registers. Thus, we would over-write and lose (some of) the values $v_1, \ldots, v_{i-1}$ of the initially evaluated expressions $e_1, \ldots, e_{i-1}$.

Again, we must transfer values to avoid losing information. Before we evaluate an expression $f(e_1, \ldots, e_n)$ involving function application, we copy the values held in the designated input registers $in_1^D, \ldots, in_n^D$ to a temporary storage area that is not used in the evaluation of the sub-expressions $e_1, \ldots, e_n$. Then, when we have finished the evaluation, we restore the input register values.

How do we find a safe storage area for copying the initial values of any input registers? The evaluation of any particular expression will require the use of only a finite number of registers. Furthermore, we can even determine which registers it will use simply by code inspection. Let us suppose we have a function

$$WorkReg : Exp(\Sigma) \rightarrow N^*$$

such that

$WorkReg(e)$ gives the indices of the registers used in the compilation of an expression $e$.

Then, given an expression

$$f(e_1, \ldots, e_n)$$

we can determine free slots

$$Max \{WorkReg(e_i) \mid 1 \leq i \leq n\} + 1, \ldots, Max \{WorkReg(e_i) \mid 1 \leq i \leq n\} + n$$

that we can use to store the values initially held in the input registers $in_1^D, \ldots, in_n^D$.

Thus, in algorithmic terms, we compile an expression

$$f(e_1, \ldots, e_n)$$

by:

Copy the input registers $in_1^D, \ldots, in_n^D$ to a safe temporary storage area.
Evaluate the sub-expression $e_1$.
Move the evaluation of $e_1$ from $out_1^D$ to $in_1^D$.
Evaluate the sub-expression $e_2$.
Move the evaluation of $e_2$ from $out_2^D$ to $in_2^D$.

Evaluate the sub-expression $e_n$.
Move the evaluation of $e_n$ from $out_n^D$ to $in_n^D$.
Evaluate $f^D$ on $in_1^D, \ldots, in_n^D$.
Restore the initial input register values.
17.8. Compiling Expressions Into Register Machine Programs

Expressing this algorithm in RM code:

\[ Compile_{Exp}(f(e_1, \ldots, e_n)) = \]

1. \[ \max\{\text{WorkReg}(e_i) \mid 1 \leq i \leq n\} + 1 \leftarrow in_1^D \]
2. \[ \vdots \]
3. \[ n. \]
4. \[ n + 1. \]
5. \[ \vdots \]
6. \[ n + \alpha_1. \]
7. \[ n + \alpha_1 + 1. \]
8. \[ n + \alpha_1 + 2. \]
9. \[ \vdots \]
10. \[ n + \alpha_1 + \alpha_2 + 1. \]
11. \[ n + \alpha_1 + \alpha_2 + 2. \]
12. \[ \vdots \]
13. \[ 2n + \alpha_1 + \cdots + \alpha_{n-1}. \]
14. \[ \vdots \]
15. \[ 2n + \alpha_1 + \cdots + \alpha_n - 1. \]
16. \[ 2n + \alpha_1 + \cdots + \alpha_n. \]
17. \[ 2n + \alpha_1 + \cdots + \alpha_n + 1. \]
18. \[ 2n + \alpha_1 + \cdots + \alpha_n + 2. \]
19. \[ \vdots \]
20. \[ 3n + \alpha_1 + \cdots + \alpha_n + 1. \]

Thus, the evaluation of a compiled function-application expression

\[ f(e_1, \ldots, e_n) \]

has the form shown in Figure 17.7:
<table>
<thead>
<tr>
<th>PC</th>
<th>$i_{in}^1$</th>
<th>$i_{in}^2$</th>
<th>...</th>
<th>$i_{in}^n$</th>
<th>$out^D$</th>
<th>$w_1$</th>
<th>...</th>
<th>$w_n$</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>?</td>
<td>?</td>
<td>...</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>?</td>
<td>?</td>
<td>$x_1$</td>
<td>...</td>
<td>copy input registers to safe area</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$n+1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>?</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>evaluate sub-expression for $e_1$</td>
</tr>
<tr>
<td>$n+2$</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>$v_1$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>copy $e_1$’s evaluation to $i_{in}^D$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$n+\alpha_1 + 1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>$v_1$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>evaluate sub-expression for $e_2$</td>
</tr>
<tr>
<td>$n+\alpha_1 + 2$</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>$v_1$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>copy $e_2$’s evaluation to $i_{in}^D$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$2n+\alpha_1 + \ldots + \alpha_n + 1$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>?</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>evaluate sub-expression for $e_n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$2n+\alpha_1 + \ldots + \alpha_n$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>...</td>
<td>$x_n$</td>
<td>$v_n$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>copy $e_n$’s evaluation to $i_{in}^D$</td>
</tr>
<tr>
<td>$2n+\alpha_1 + \ldots + \alpha_n + 1$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>...</td>
<td>$v_n$</td>
<td>$v_n$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>apply $f^D$ to evaluated sub-expressions</td>
</tr>
<tr>
<td>$2n+\alpha_1 + \ldots + \alpha_n + 2$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>...</td>
<td>$v_n$</td>
<td>$f^D(v_1, \ldots, v_n)$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td></td>
</tr>
<tr>
<td>$2n+\alpha_1 + \ldots + \alpha_n + 3$</td>
<td>$x_1$</td>
<td>$v_2$</td>
<td>...</td>
<td>$v_n$</td>
<td>$f^D(v_1, \ldots, v_n)$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td>restore initial input register values</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$3n+\alpha_1 + \ldots + \alpha_n + 2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
<td>$v_n$</td>
<td>$f^D(v_1, \ldots, v_n)$</td>
<td>$x_1$</td>
<td>...</td>
<td>$x_n$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 17.7: Execution trace for expressions formed by function application.
17.8. COMPILING EXPRESSIONS INTO REGISTER MACHINE PROGRAMS

17.8.2 Correctness of Expression Compilation

We need to show that

\[ \text{Decode}_{\text{data}}(V^D_\sigma(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))) = V^A_\sigma(e)(\sigma) \]

holds for all expressions \( e \in \text{Exp}(\Sigma) \) and all states \( \sigma \in \text{State}(A) \), where

\[ V^D_\sigma : \text{Exp}(\Sigma) \rightarrow D \]

is the RM expression evaluation function defined by

\[ V^D_\sigma(P)(\tau) = \text{Decode}_{\text{data}}(M^D_\sigma(P)(\tau)(\text{out}^1)) \]

We shall prove this result as a corollary to:

**Lemma (Expression Execution)** For all expressions \( e \in \text{Exp}(\Sigma) \) and all states \( \sigma \in \text{State}(A) \),

\[ M^D_\sigma(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma)) = \text{Encode}(\sigma)[|e| + 1/PC][\text{Encode}_{\text{Data}}(V^A_\sigma(e)(\sigma))/\text{out}^P][\text{in}^D_{\text{Orig}}(\text{Encode}(\sigma))/\text{work}^D(e)] \]

where

\[ \text{in}^D_{\text{Orig}}(\tau)/\text{work}^D(e_1, \ldots, e_n) = [\tau(\text{in}^P_1)/\text{Max}\{Wk\text{Reg}(e_i) \mid 1 \leq i \leq n\} + 1] \]

\[ \cdots [\tau(\text{in}^P_n)/\text{Max}\{Wk\text{Reg}(e_i) \mid 1 \leq i \leq n\} + n] \]

**Proof** Suppose, for induction hypotheses, that the lemma holds for expressions \( e_1, \ldots, e_n \).

**Constants**

By definition of \( \text{Compile}_{\text{Exp}} \) and \( \text{Encode} \),

\[ M^D_\sigma(\text{Compile}_{\text{Exp}}(c))(\text{Encode}(\sigma)) = M^D_\sigma(1.\text{out}^1 \leftarrow c)((1, \rho)) \]

By definition of \( M^D_\sigma \),

\[ = (2, \rho)[c^D/\text{out}^1] \]

As \( \text{Encode}_{\text{data}} \) is a homomorphism,

\[ = (2, \rho)[\text{Encode}_{\text{data}}(c^A)/\text{out}^P] \]

By definition of \( V^A_\sigma \),

\[ = (2, \rho)[\text{Encode}_{\text{data}}(V^A_\sigma(c)(\sigma))/\text{out}^P] \]

By definition of register substitution,

\[ = (1, \rho)[|c|/PC][\text{Encode}_{\text{data}}(V^A_\sigma(c)(\sigma))/\text{out}^P] \]

and as \( \text{work}^D(c) = \emptyset \), we can trivially say that

\[ = (1, \rho)[|c|/PC][\text{Encode}_{\text{data}}(V^A_\sigma(c)(\sigma))/\text{out}^P][\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(c)] \].
CHAPTER 17. COMPILATION

Variables

The case for variable compilation is similar to that for constants.

Function Application

By definition of $Encode$,

$$M^D_{io}(\text{Compile}_{Exp}(f(e_1, \ldots, e_n)))(Encode(\sigma)) = M^D_{io}(\text{Compile}_{Exp}(f(e_1, \ldots, e_n)))(1, \rho)$$

As $work^D(e_1, \ldots, e_n) \cap (\bigcup_i^\rho \text{Reg}(e_i)) = \emptyset$, we can apply the Register Restoration Lemma,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))((1, \rho)) \frac{in^D_{\text{Orig}}(\rho)/in^D(e)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)} [f(e_1, \ldots, e_n)]/PC$$

where

$$\frac{in^D_{\text{Orig}}(\rho)/in^D(e)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)} = [\rho(in^D_1)/in^D] \cdots [\rho(in^D_{\text{arity}(e)})/in^D_{\text{arity}(e)}].$$

By the Subprogram Execution Lemma and definitions of $Body$ and $Encode$,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))(M^D_{io}(\text{Compile}(e_1))(Encode(\sigma)) \frac{in^D_{\text{Orig}}(\rho)/in^D(e)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}[[f(e_1, \ldots, e_n)]/PC].$$

Applying the Induction Hypothesis to $e_1$,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))(Encode(\sigma) \frac{\alpha_1 + 1/PC} {Encode_{Data}(V^A_{io}(e_1)(\sigma))/out^P} \frac{in^D_{\text{Orig}}(\rho)/out^P}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}[[f(e_1, \ldots, e_n)]/PC].$$

By definition of $Encode$ and register substitution,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))\
(M^D_{io} (\alpha_1 + 1.in^D \leftarrow out^P) \frac{in^D_{\text{Orig}}(\rho)/in^D(e)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}[[f(e_1, \ldots, e_n)]/PC].$$

By the Single Instruction Execution Lemma,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))\
(M^D_{io} (\alpha_1 + 1.in^D \leftarrow out^P) \frac{in^D_{\text{Orig}}(\rho)/in^D(e)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}[[f(e_1, \ldots, e_n)]/PC].$$

By the definition of $M^D_{io}$ and register evaluation,

$$= M^D_{io}(Body(\text{Compile}_{Exp}(f(e_1, \ldots, e_n))))\
((\alpha_1 + 2, \rho) \frac{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}{in^D_{\text{Orig}}(\rho)/work^D(e_1, \ldots, e_n)}[[f(e_1, \ldots, e_n)]/PC].$$
Having executed the expression \( e_1 \), we show that we can tidy the resulting state ready for subsequent expression evaluations. By the Execution Decomposition and Relabelling Invariance Lemmas,

\[
\begin{align*}
M^D_w(\text{Body}(\text{CompileExp}(f(e_1, \ldots, e_n)))) &= M^D_w(\text{CompileExp}(e_2)) \\
&= M^D_w(\text{CompileExp}(e_2)) \\
&\equiv (1, \rho) \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}].
\end{align*}
\]

By the Delaying Substitution Lemma,

\[
\begin{align*}
M^D_w(\text{Body}(\text{CompileExp}(f(e_1, \ldots, e_n)))) &= M^D_w(\text{CompileExp}(e_2)) \\
&= M^D_w(\text{CompileExp}(e_2)) \\
&\equiv (1, \rho) \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}].
\end{align*}
\]

and again,

\[
\begin{align*}
M^D_w(\text{Body}(\text{CompileExp}(f(e_1, \ldots, e_n)))) &= M^D_w(\text{CompileExp}(e_2)) \\
&= M^D_w(\text{CompileExp}(e_2)) \\
&\equiv (1, \rho) \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}].
\end{align*}
\]

Continuing this process, we get:

\[
\begin{align*}
M^D_w(\text{Body}(\text{CompileExp}(f(e_1, \ldots, e_n)))) &= M^D_w(\text{CompileExp}(e_2)) \\
&= M^D_w(\text{CompileExp}(e_2)) \\
&\equiv (1, \rho) \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}].
\end{align*}
\]

By the Single Instruction Execution Lemma,

\[
\begin{align*}
M^D_w(\text{Body}(\text{CompileExp}(f(e_1, \ldots, e_n)))) &= M^D_w(\text{CompileExp}(e_2)) \\
&= M^D_w(\text{CompileExp}(e_2)) \\
&\equiv (1, \rho) \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv \left[ \text{EncodeData}(V^A_{10}(e_1)(\sigma))/\text{out}^D_1 \right] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}] \\
&\equiv [\alpha_1 + \alpha_2 + 1/\text{PC}].
\end{align*}
\]
By definition of $\mathcal{M}^D_\text{io}$ and register substitution,

$$= M^D_\text{io}(\text{Body}(\text{Compile}_{\text{Exp}}(f(e_1, \ldots, e_n))))$$

$$= (2n + \alpha_1 + \cdots + \alpha_n + 2, \rho)$$

$$\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_n)(\sigma))/\text{out}^D}{\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}{\cdots}\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_n)}}$$

By the Register Substitution Lemma,

$$= M^D_\text{io}(\text{Body}(\text{Compile}_{\text{Exp}}(f(e_1, \ldots, e_n))))$$

$$= (2n + \alpha_1 + \cdots + \alpha_n + 2, \rho)$$

$$\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_1)(\sigma))/\text{in}^D}{\cdots}\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_n)(\sigma))/\text{in}^D}{\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}{\cdots}\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_n)}}$$

By the Execution Cessation Lemma,

$$= (1, \rho)\frac{f(e_1, \ldots, e_n)/PC}{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}$$

$$\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_1)(\sigma))/\text{out}^D}{\cdots}\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_n)(\sigma))/\text{out}^D}{\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}{\cdots}\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_n)}}$$

By the Register Substitution Lemma,

$$= (1, \rho)\frac{f(e_1, \ldots, e_n)/PC}{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}$$

$$\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_1)(\sigma))/\text{out}^D}{\cdots}\frac{\text{Encode}_{\text{Data}}(V^A\text{io}(e_n)(\sigma))/\text{out}^D}{\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_1)}{\cdots}\frac{\text{in}^D_{\text{Orig}}(\rho)/\text{work}^D(e_n)}}$$

**Corollary (Expression Evaluation)** For all expressions $e \in \text{Exp}(\Sigma)$ and all states $\sigma \in \text{State}(A)$,

$$\text{Decode}_{\text{data}}(V^D\text{io}(\text{Compile}_{\text{Exp}}(e))(\text{Encode}(\sigma))) = V^A\text{io}(e)(\sigma).$$

**Proof** Follows by simple register evaluation from the Expression Execution Lemma.

---

### 17.9 Compiling Boolean Expressions into Register Machine Programs

#### 17.9.1 The Compiler

As Boolean expressions are just a particular type of expressions, their compilation follows the same principles as that for expressions.
We designate certain registers to have particular purposes for the evaluation of Boolean
expressions; these registers are separate from those used for expression compilation. We suppose
we have three sets of distinct registers:

(i) \ldots , \textit{Allocate}(b) , \ldots \text{ for storing the values of Boolean variables } \ldots , b , \ldots \text{ present in a source}
programs;

(ii) \textit{in}\_1^B , \ldots \text{ for storing the Boolean values used for inputs; and}

(iii) \textit{out}\_1^B , \ldots \text{ for storing the Boolean values used for outputs.}

\textbf{Compiling True and False}

We set a register to the value \textit{tt} with the \textit{\Sigma-RM} instruction \texttt{true}(r). Thus, we compile the
constant for true by:

\[\text{Compile}_{BExp} (\texttt{true}) = 1. \textit{out}\_1^B \leftarrow \texttt{true}\]

which, when executed yields the trace:

<table>
<thead>
<tr>
<th>\textit{PC}</th>
<th>\textit{out}_1^B</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>\textit{tt}</td>
<td>set the output register</td>
</tr>
</tbody>
</table>

And similarly, for false, the compiled code is:

\[\text{Compile}_{BExp} (\texttt{false}) = 1. \textit{out}\_1^B \leftarrow \texttt{false}\]

which, when executed yields the trace:

<table>
<thead>
<tr>
<th>\textit{PC}</th>
<th>\textit{out}_1^B</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>\textit{ff}</td>
<td>sets the output register</td>
</tr>
</tbody>
</table>

\textbf{Compiling Boolean Variables}

Just as for expressions, we designate a register with the function \textit{Allocate} to store the values
of any Boolean variable appearing in a \texttt{while} program.

\textbf{Compiling Negation}

To compile the Boolean expression

\begin{equation}
\text{not } (b)
\end{equation}

we compile the sub-expression \(b\) and then negate the result. We choose to follow the pattern
for compiling general expressions, so we also add (superfluously in this case, as negation is a
unary operator) the book-keeping instructions detailed in Section 17.8.

\[
Compile_{B_{expr}}(\text{not } b) = \begin{align*}
1. & \quad \text{Max}\{WorkReg(b)\} + 1 \leftarrow in_1^B \\
2. & \quad \vdots \quad Compile_{B_{expr}}(b) \\
& \quad \beta + 1. \\
& \quad \beta + 2. \quad in_1^B \leftarrow out_1^B \\
& \quad \beta + 3. \quad out_1^B \leftarrow \text{not}(in_1^B) \\
& \quad \beta + 4. \quad in_1^B \leftarrow \text{Max}\{WorkReg(b)\} + 1
\end{align*}
\]

The execution of compiled negated expressions yields:

<table>
<thead>
<tr>
<th>PC</th>
<th>(in_1^B)</th>
<th>(out_1^B)</th>
<th>(w_1)</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>?</td>
<td>x</td>
<td>copy input register to safe area</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>?</td>
<td>x</td>
<td>evaluate the sub-expression for (b)</td>
</tr>
<tr>
<td>(\vdots)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\beta + 2)</td>
<td>x</td>
<td>v</td>
<td>x</td>
<td>copy output to input register</td>
</tr>
<tr>
<td>(\beta + 3)</td>
<td>v</td>
<td>v</td>
<td>x</td>
<td>negate evaluation of (b)</td>
</tr>
<tr>
<td>(\beta + 4)</td>
<td>v</td>
<td>-(v)</td>
<td>x</td>
<td>restore input register value</td>
</tr>
<tr>
<td>(\beta + 5)</td>
<td>x</td>
<td>-(v)</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Compiling Conjunction

We compile a Boolean expression

\(b_1 \text{ and } b_2\)

formed as the conjunction of two Boolean sub-expressions \(b_1\) and \(b_2\) by compiling first \(b_1\), then \(b_2\), and then applying the conjunction operator. Here, as for general expressions we also have to take care not to over-write results from nesting Boolean expressions.

\[
Compile_{B_{expr}}(b_1 \text{ and } b_2) = \begin{align*}
1. & \quad \text{Max}\{WorkReg(b_1), WorkReg(b_2)\} + 1 \leftarrow in_1^B \\
2. & \quad \text{Max}\{WorkReg(b_1), WorkReg(b_2)\} + 2 \leftarrow in_2^B \\
3. & \quad \vdots \quad Compile_{B_{expr}}(b_1) \\
& \quad \beta_1 + 2. \\
& \quad \beta_1 + 3. \\
& \quad \beta_1 + 4. \\
& \quad \vdots \quad Compile_{B_{expr}}(b_2) \\
& \quad \beta_1 + \beta_2 + 3. \\
& \quad \beta_1 + \beta_2 + 4. \\
& \quad \beta_1 + \beta_2 + 5. \\
& \quad \beta_1 + \beta_2 + 6. \\
& \quad \beta_1 + \beta_2 + 7. \\
\end{align*}
\]
When we execute this code, we get a trace:

<table>
<thead>
<tr>
<th>PC</th>
<th>( \text{in}_1^\mathbb{B} )</th>
<th>( \text{in}_2^\mathbb{B} )</th>
<th>( \text{out}_1^\mathbb{B} )</th>
<th>( \text{w}_1 )</th>
<th>( \text{w}_2 )</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>?</td>
<td>( x_1 )</td>
<td>?</td>
<td>copy input register to safe area</td>
</tr>
<tr>
<td>3</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>?</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>copy input register to safe area</td>
</tr>
<tr>
<td>4</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>?</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>execute sub-expression for ( b_1 )</td>
</tr>
<tr>
<td>( \beta_1 + 3 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( v_1 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>execute sub-expression for ( b_1 )</td>
</tr>
<tr>
<td>( \beta_1 + 4 )</td>
<td>( v_1 )</td>
<td>( x_2 )</td>
<td>( v_1 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>copy ( b_1 )'s output to first input register</td>
</tr>
<tr>
<td>( \beta_1 + 5 )</td>
<td>( v_1 )</td>
<td>( x_2 )</td>
<td>?</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>execute sub-expression for ( b_2 )</td>
</tr>
<tr>
<td>( \beta_1 + \beta_2 + 4 )</td>
<td>( v_1 )</td>
<td>( x_2 )</td>
<td>( v_2 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>execute sub-expression for ( b_2 )</td>
</tr>
<tr>
<td>( \beta_1 + \beta_2 + 5 )</td>
<td>( v_1 )</td>
<td>( v_2 )</td>
<td>( v_1 \land v_2 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>copy ( b_2 )'s output to second input register</td>
</tr>
<tr>
<td>( \beta_1 + \beta_2 + 6 )</td>
<td>( v_1 )</td>
<td>( v_2 )</td>
<td>( v_1 \land v_2 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>take conjunction of ( b_1 ) and ( b_2 )'s values</td>
</tr>
<tr>
<td>( \beta_1 + \beta_2 + 7 )</td>
<td>( x_1 )</td>
<td>( v_2 )</td>
<td>( v_1 \land v_2 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>restore first input register</td>
</tr>
<tr>
<td>( \beta_1 + \beta_2 + 8 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( v_1 \land v_2 )</td>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>restore second input register</td>
</tr>
</tbody>
</table>

**Compiling Disjunction**

We follow the same pattern for conjunction to compile the disjunction

\[
\text{\textbf{or}(} \begin{equation}
\end{equation}\text{)}
\]

of two sub-expressions \( b_1 \) and \( b_2 \):

\[
\text{Compile}_{BE}^x(p \text{ or } q) =
\]

1. \( \text{Max}\{\text{WorkReg}(b_1), \text{WorkReg}(b_2)\} + 1 \leftarrow \text{in}_1^\mathbb{B} \)
2. \( \text{Max}\{\text{WorkReg}(b_1), \text{WorkReg}(b_2)\} + 2 \leftarrow \text{in}_2^\mathbb{B} \)
3. 
   \begin{align*}
   \beta_1 + 2. & \\
   \beta_1 + 3. & \\
   \beta_1 + 4. & \\
   \end{align*}
   \begin{align*}
   \text{Compile}_{BE}^x(b_1) & \\
   \text{\textit{in}_1^\mathbb{B} \leftarrow \text{out}_1^\mathbb{B} } & \\
   \end{align*}
4. 
   \begin{align*}
   \beta_1 + \beta_2 + 3. & \\
   \beta_1 + \beta_2 + 4. & \\
   \beta_1 + \beta_2 + 5. & \\
   \beta_1 + \beta_2 + 6. & \\
   \beta_1 + \beta_2 + 7. & \\
   \end{align*}
   \begin{align*}
   \text{\textit{in}_2^\mathbb{B} \leftarrow \text{out}_1^\mathbb{B} } & \\
   \text{\textit{out}_1^\mathbb{B} \leftarrow \text{or}(\text{in}_1^\mathbb{B}, \text{in}_2^\mathbb{B})} & \\
   \text{\textit{in}_1^\mathbb{B} \leftarrow \text{Max}\{\text{WorkReg}(b_1), \text{WorkReg}(b_2)\} + 1} & \\
   \text{\textit{in}_2^\mathbb{B} \leftarrow \text{Max}\{\text{WorkReg}(b_1), \text{WorkReg}(b_2)\} + 2} & \\
   \end{align*}

The only difference in the code for conjunction and disjunction is that we apply the instruction \text{or}(r)\ in place of \text{and}.

The trace then for the execution of disjunction is as would be expected:
<table>
<thead>
<tr>
<th>$\text{PC}$</th>
<th>$\text{in}_1^B$</th>
<th>$\text{in}_2^B$</th>
<th>$\text{out}_1^B$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>initial state</td>
</tr>
<tr>
<td>2</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>?</td>
<td>$x_1$</td>
<td>?</td>
<td>copy input register to safe area</td>
</tr>
<tr>
<td>3</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>?</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>copy input register to safe area</td>
</tr>
<tr>
<td>4</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>?</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>execute sub-expression for $b_1$</td>
</tr>
<tr>
<td>$\beta_1 + 3$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$v_1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>execute sub-expression for $b_2$</td>
</tr>
<tr>
<td>$\beta_1 + 4$</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>$v_1$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>copy $b_1$’s output to first input register</td>
</tr>
<tr>
<td>$\beta_1 + 5$</td>
<td>$v_1$</td>
<td>$x_2$</td>
<td>?</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>copy $b_2$’s output to second input register</td>
</tr>
<tr>
<td>$\beta_1 + 5$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>?</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>take disjunction of $b_1$ and $b_2$’s values</td>
</tr>
<tr>
<td>$\beta_1 + 6$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$v_1 \lor v_2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>restore first input register</td>
</tr>
<tr>
<td>$\beta_1 + 7$</td>
<td>$x_1$</td>
<td>$v_2$</td>
<td>$v_1 \lor v_2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>restore second input register</td>
</tr>
<tr>
<td>$\beta_1 + 8$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$v_1 \lor v_2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td></td>
</tr>
</tbody>
</table>

### Compiling Relation Application

In general, we can form Boolean expressions

$$r(e_1, \ldots, e_n)$$

using function application applied to expressions as input. We compile such Boolean expressions in an exactly analogous manner as to that of general expressions formed by function application. The only difference is that we store the output of the calculation in the Boolean output register $out_1^B$.

#### 17.9.2 Correctness of Boolean Expression Compilation

We need to show that

$$Decode_{data}(W_{io}^D(\text{Compile}_{BE_{Exp}}(b))(\text{Encode}(\sigma))) = W_{io}^A(b)(\sigma)$$

holds for all expressions $e \in Exp(\Sigma)$ and all states $\sigma \in State(A)$, where

$$W_{io}^D : BE_{Exp}(\Sigma) \rightarrow B$$

is the RM expression evaluation function defined by

$$W_{io}^D(P)(\tau) = Decode_{data}((M_{io}^D(P)(\tau))(out_1^D)).$$

We shall prove this result as a corollary to:

**Lemma (Boolean Expression Execution)** For all expressions $e \in Exp(\Sigma)$ and all states $\sigma \in State(A)$,

$$M_{io}^D(\text{Compile}_{BE_{Exp}}(b))(\text{Encode}(\sigma)) = Encode(\sigma)[[b + 1/PC][\text{Encode}_{Data}(V_{io}^A(b)(\sigma))/out_1^B][\text{mOrig}(b, \rho)/\text{work}(b)]$$

**Proof** Suppose, for induction hypotheses, that the lemma holds for Boolean expressions $b$, $b_1$ and $b_2$. 


17.9. COMPILING BOOLEAN EXPRESSIONS INTO REGISTER MACHINE PROGRAMS

**Constants**

The compilation of the Boolean constants true and false follows the same pattern as for general expression constants.

**Negated Boolean Expressions**

By definition of $Encode$, 

$$M_{io}^{D}(Compile_{BEexp}(\text{not}(b)))(Encode(\sigma)) = M_{io}^{D}(Compile_{BEexp}(\text{not}(b)))(1, \rho).$$

By the Register Restoration Lemma, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))((1, \rho))$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1].$$

By the Execution Decomposition Lemma, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))(M_{io}^{D}(Compile_{BEexp}(b))((1, \rho))$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1].$$

Applying the Induction Hypothesis to $b$, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))$$

$$(((1, \rho)[\beta + 1/PC][\text{Encode}_{\text{Data}}(W^{A}(b)(\sigma))/\text{out}^{B}][\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)]$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].$$

By the Single Instruction Execution Lemma and definition of substitution, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))$$

$$((\beta + 1, \rho)[\text{Encode}_{\text{Data}}(W^{A}(b)(\sigma))/\text{out}^{B}][\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)]$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].$$

By definition of $M_{io}^{D}$, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))$$

$$((\beta + 2, \rho)[\text{Encode}_{\text{Data}}(W^{A}(b)(\sigma))/\text{out}^{B}][\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)]$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].$$

By the Single Instruction Execution Lemma, 

$$= M_{io}^{D}(Body(Compile_{BEexp}(\text{not}(b))))$$

$$((\beta + 2, \rho)[\text{Encode}_{\text{Data}}(W^{A}(b)(\sigma))/\text{out}^{B}][\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)]$$

$$[\rho(in^{B})/\rho(in^{B})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].$$
By definition of $M^D_{\omega b}$ and the Substitution Lemma,

$$
= M^D_{\omega b}(\text{Compile}_{\text{Exp}}(\text{not} \ (b)))
\begin{align*}
&=((\beta + 3, \rho)[\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{out}^B_{1}] \\
&[\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)] \\
&[\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{in}^B_{1}] - (\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{out}^B_{1}] \\
&[\rho(\text{in}^B_{1})/\text{in}^B_{1}] [\rho(\text{in}^B_{1})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].
\end{align*}
$$

By the Execution Cessation Lemma,

$$
= (\beta + 3, \rho)[\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{out}^B_{1}] \\
[\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)] \\
[\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{in}^B_{1}] - (\text{Encode}_{\text{Data}}(W^A(b)(\sigma))/\text{out}^B_{1}] \\
[\rho(\text{in}^B_{1})/\text{in}^B_{1}] [\rho(\text{in}^B_{1})]/\text{Max}\{\text{WorkReg}(b)\} + 1][\beta + 4/PC].
$$

By the Substitution Lemma,

$$
= (1, \rho)[\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)] \\
[-(\text{Encode}_{\text{Data}}(W^A(b)(\sigma)))/\text{out}^B_{1}] \\
[\rho(\text{in}^B_{1})]/\text{Max}\{\text{WorkReg}(b)\} + 1] \\
[\beta + 4/PC].
$$

By the definitions of $\text{Encode}$ and $W^A,$

$$
= (1, \rho)[\text{in}_{\text{Orig}}(b, \rho)/\text{work}(b)] \\
[\text{Encode}_{\text{Data}}(W^A(\text{not}(b))(\sigma))/\text{out}^B_{1}] \\
[\rho(\text{in}^B_{1})]/\text{Max}\{\text{WorkReg}(b)\} + 1] \\
[\text{not} \ b]/PC.
$$

**Conjunction**

The proof, which we have omitted, is along similar lines to that of negation. Thus, it is proved by structural induction on the two sub-expressions $b_1$ and $b_2.$

**Relation Application**

The proof, is analogous to function application.

□

**Corollary (Boolean Expression Evaluation)** For all Boolean expressions $b \in B\text{Exp}(\Sigma)$ and all states $\sigma \in \text{State}(A),$

$$
\text{Decode}_{\text{Data}}(W^D_{\omega b}(\text{Compile}_{\text{Exp}}(b))(\text{Encode}(\sigma))) = W^A_{\omega b}(b)(\sigma).
$$

**Proof** Follows by simple register evaluation from the Boolean Expression Execution Lemma. □
Chapter 18

Further Reading

18.1 Basic Topics

The first two texts are recommended reading:


The following texts give accounts of the material covered in the course:


Data is covered in:


The work on syntax and grammars is covered in


Other relevant undergraduate books on semantics are:


For graduate studies on semantics there are:


Specification and correctness is well surveyed in:


...Compilation...

...Logical Verification...

...Efficiency...

### 18.2 Advanced Topics

#### 18.2.1 Abstract Data Types and Equational Specifications

#### 18.2.2 Computable Data Types

#### 18.2.3 Fixed Points and Domain Theory

#### 18.2.4 λ-Calculus and Type Theory

#### 18.2.5 Concurrency
Bibliography

Bar-Hillel et al. [1961]

Birkhoff and MacLane [1965]

Borst [1993]

Burrill [1967]

Chen and Tucker [2000]

Chen et al. [2000]

Chomsky [1956]

Cohn [1982]

Comer [1995]

Dale and Walker [1996]

Dedekind [1888]
R Dedekind. Was Sind und was Sollen die Zahlen? Braunschweig, 1888.
Dijkstra [1962]

Dijkstra [1976]

Fauvel and Gray [1987]

Fraleigh [1967]

Herstein [1964]

Hopcroft and Ullman [1979]

Huntington [1904]

Kaufman et al. [1993]

Napier [1616]
J Napier. *A Description of the Admirable Table of Logarithms.* Nicholas Okes, London, 1616. Translated from original Latin by Edward Wright.

Struik [1969]

van der Waerden [1949]

Whitrow [1980]