CALCO Early Ideas 2013
Warsaw, 2 September 2013
A satellite workshop of CALCO 2013
5th Conference on Algebra and Coalgebra in Computer Science

Selected Papers
edited by
Bartek Klin and Monika Seisenberger
Preface

The International Conference on Algebra and Coalgebra in Computer Science, CALCO, aims to bring together researchers and practitioners with interests in foundational aspects, and both traditional and emerging uses of algebras and coalgebras in computer science. This is a high-level, biennial conference formed by joining the forces and reputations of CMCS (the International Workshop on Coalgebraic Methods in Computer Science), and WADT (the Workshop on Algebraic Development Techniques). Previous very successful CALCO conferences were held 2005 in Swansea, Wales, 2007 in Bergen, Norway, 2009 in Udine, Italy, and 2011 in Winchester, UK. This fifth event took place in Warsaw, Poland.

The CALCO Early Ideas Workshop was a CALCO satellite event dedicated to presentation of work in progress and original research proposals. PhD students and young researchers were particularly encouraged to contribute. Attendance at the workshop was open to all — many CALCO conference participants attended CALCO-EI and vice versa. The workshop had 10 contributions and over 40 participants.

CALCO Early Ideas presentations have been selected on the basis of submitted 2-page short contributions, by the CALCO-EI PC. After the workshop, the authors of each presentation were invited to submit a full 10-15 page paper on the same topic. They were also asked to write anonymous reviews of papers submitted by other authors on related topics. Additional reviewing was organised and the final selection of papers was carried out by the programme committee. The volume of selected papers from the workshop is published as a technical report, and available on arXiv. Authors will retain copyright, and are also encouraged to disseminate the results reported at CALCO-EI by subsequent publication elsewhere.

We would like to thank the workshop authors and participants for their contributions, and the CALCO-EI PC as well as all referees for their scientific work, thereby making this workshop a successful event. Special thanks go the CALCO PC chairs Stefan Milius and Reiko Heikel and the CALCO 2013 local organizers for their efforts and continuous support of the workshop. Finally, the support of all sponsoring institutions is gratefully acknowledged.

April 2014

Bartek Klin
Monika Seisenberger
CALCO-Early Ideas Programme Committee

Bartek Klin (Warsaw University)
Narciso Martí-Oliet (Universidad Complutense de Madrid)
Till Mossakowski (Universität Magdeburg)
John Power (Bath University)
Monika Seisenberger (Chair, Swansea University)
# Table of Contents

Towards a Formal Semantics-Based Technique for Interprocedural Slicing 1  
Irina Măriuca Asăvoae and Mihail Asăvoae (VERIMAG/UJF), and Adrián Riesco (Universidad Complutense de Madrid)

Nominal Sets over Algebraic Atoms .............................................. 17  
Joanna Ochremiak (University of Warsaw)

An Investigation of Preorders and Preorder $\mathbb{T}$-$\mathbb{T}$-Liftings on the Subdistribution Monad ................................................. 33  
Tetsuya Sato (Kyoto University)

Towards (Co)Algebraic Semantics of the Object-Oriented Rewriter Style Pattern ......................................................... 49  
Baltasar Trancón y Widemann (Ilmenau University of Technology) and Markus Lepper (<semantics/> GmbH)

Author Index ............................................................................. 63
Towards a Formal Semantics-Based Technique for Interprocedural Slicing

Irina Măriuca Asăvoae, Mihail Asăvoae, Adrián Riesco

1 VERIMAG/UJF, France
{irina.asavoae, mihail.asavoae}@imag.fr
2 Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Abstract. Interprocedural slicing is a technique applied on programs with procedures and relies on how the information is passed at procedure call/return sites. Such a technique computes program slices (i.e. program fragments restricted w.r.t. a given criterion). The existing approaches to interprocedural slicing exploit the particularities of the underlying language semantics in order to compute program slices. In this paper we propose a generic technique for the problem of interprocedural slicing. More specifically, our approach takes as input a language semantics (given as a rewriting-logic specification) and infers some particularities of it which are further used to compute the program slices.

Keywords: slicing, semantics, Maude, debugging

1 Introduction

Complex software systems are built in a modular fashion where modularity is implemented with functions and procedures in imperative languages, with classes and interfaces in object-oriented programming, with modules in declarative-style programming, or by other means of organizing the code. Besides its structural characteristic, the modularity also carries semantic information. For example, the modules could be parameterized by types and values (e.g. the generic classes of Java and C#, the template classes of C++, or the parameterized modules of Maude and OCaml) or could have specialized usability (e.g. abstract classes in object-oriented languages or functors in functional programming languages).

It is preferable, for efficiency and accuracy reasons, that the modular characteristics of a system are preserved during its development and are used for its analysis. As such, it is advisable to integrate the development and the analysis of a system. One possible solution for this integration is to use a formal executable framework such as rewriting logic [10]. The integration methodology proposed by rewriting logic starts with a formal executable semantics of the programming language used for the system development. The formal executable semantics provides the set of all concrete executions, for any program correctly
constructed w.r.t. the language syntax and for all possible input data. Next, the notion of a concrete execution could be extended to an abstract execution—seen as a program execution with an analysis tool. This is due to the fact that the abstractions and, implicitly, abstract executions are representations of sets of concrete program executions. One particular abstraction is the program slicing [20], which computes safe program fragments (also called slices) w.r.t. a specified set of variables. A complex variant of program slicing, called interprocedural slicing, preserves the modularity of the underlying program and exploits how the program data is passed between the program modules.

The standardized approaches to slicing consider a particular programming language and work around its particularities. We aim to generalize these approaches by giving the slicing method in a generic fashion, using as premises the programming language semantics specification. Namely, we infer specific particularities of the language from the semantics specification of the language and use these particularities for the analysis method. For example, in our previous work from [13] we discover side-effect language constructs (i.e., constructs triggering data-store changes) which we further use for (intraprocedural) program slicing. In the current work we aim to discover, based on the semantics of the language, which are the language constructs producing context-updates. Next, we explain the terminology used, i.e., interprocedural slicing and context-updates.

Interprocedural slicing is the slicing method applied on programs with procedures where the slice is computed for the entire program by taking into account the procedure calls/returns. The main problem that arises in interprocedural slicing is related to the fact that the procedure calls/returns may be analyzed with a too coarse abstraction. Usually, the coarse abstraction relies only on the call graph without taking into account the context changes occurring during a procedure call/return. By context changes we refer to the instantiation of the local variables during and after a procedure execution. Since we develop a generic slicing method which relies on the language semantics specification, we assume that we do not know what language constructs produce these context changes. Hence, we include in our slicing method a step for inferring these language constructs and we denote them as context-update constructs.

Now, the two-step technique for interprocedural slicing proposed in this paper extends our previous work on slicing from [13] and can be summarized as follows: Given a language semantics $S$, in the first step we extract context-update constructs $c$ from $S$ and, in the second step, we use these constructs for the interprocedural slicing of $S$-programs (i.e., programs written in the language specified by $S$, i.e., well-formed terms in $S$). We define $S$ as a rewriting logic theory [10] which is executable and benefits of tool support via the Maude system [2], an implementation of the rewriting logic framework. Our method for obtaining the context-update constructs is, in fact, a meta-analysis of the programming language which uses the Maude metalevel to analyze $S$. The interprocedural program slicing uses the context-updates from $c$ to guide, collect, and propagate abstract information. This technique is to be concretized with an implementation into a generic semantics-based slicing tool developed in Maude. Hereafter,
we give more details about each part of the currently proposed technique: the semantics $S$, the meta-analysis of $S$, and the $S$-program interprocedural slicing.

Firstly, we consider $S$ to be from the family of imperative and object-oriented languages. Many constructs in these languages use the notion of scope which delimits pieces of program, e.g., the loops or function bodies. These types of scoping information are explicitly represented and manipulated in the rewriting logic definitions of C [3] or Java [4]. We use here, as a case study for $S$, an extension of the WHILE language [7] in which we introduce variable scoping (i.e. homonymous variables behave differently w.r.t. the current scope). We call this extension WhileF which is a simple imperative language with conditions and loops, enriched with constructs like functions and local variables.

Secondly, we need to identify the context-update language constructs appearing in $S$. For example, the introduction of variable scoping triggers the necessity of differentiating the variables based on their scopes. Consequently, we propose a meta-analysis to discover the language constructs delimiting such scopes from the semantics specification $S$. The results of the proposed meta-analysis are called context-updates and they trigger modularity features such as variable scoping or parameter passing. Other modularity features present in object-oriented languages are classes and object construction/destruction. We aim to further study the generalization of context-update meta-analysis to all such modularity aspects for different classes of languages.

Thirdly, the program slicing step of our technique receives as input both context-update and side-effect information hence the relatively simple slicing step in [13] is subject to heavy changes. For one, it is necessary to address the representability of the derived context-update constructs w.r.t. the interprocedural program slicing. Namely, a combined representation of context-update and side-effect constructs could be terms denoting generic skeletons for procedure summaries—a succinct representation of the procedure behavior w.r.t. its input variables, as introduced in [16] and advanced in [5]. Our approach is complementary to it because it creates language-specific representations of summaries based on the specific manipulation of context-updates in the formal semantics.

Finally, the interprocedural slicing proposed here has the following status w.r.t. the Maude implementation: Our work in [13] containing the meta-analysis for side-effect constructs and the intraprocedural slicing is already prototyped in Maude. For the current work, we defined WhileF and we set the interprocedural slicing to work with the context-updates, side-effects, and intraprocedural slicing. However, the meta-analysis producing context-updates is, at the moment, at conceptual level, being used as an hand-given input. It also remains to implement the procedure summaries which improve the accuracy of the slicing result as well as the procedure parameter passing inference.

The rest of the paper is organized as follows: Section 2 presents comparisons with some related work organized in two parts w.r.t. standard program slicing and analysis methods in rewriting logic. Section 3 introduces the basic notions about the Maude system and our WhileF Maude specification. Section 4 de-
scribes foundations of program slicing and our proposed interprocedural slicing algorithm on an example. Section 5 concludes and presents the future work.

2 Related Work

Program slicing is a program analysis technique which computes safe program slices (i.e. sequences of program statements) w.r.t. a given criteria (i.e. sets of variables of interest). Program slicing addresses a wide range of applications from code parallelization [18] to program testing [6], debugging [17], and static analysis [9,8]. We note first that our work resides at the intersection of standard program analysis and rewriting logic. As such, we organize the related work presentation in two categories: (A) standard slicing as defined in the already classical program analysis and (B) rewriting logic from the perspective of its preexistent analysis tools and its programming languages semantics specifications. We use this section to better localize our work rather than to give a direct comparison with other researches. The reasons for this setting are: (1) even though program slicing is relatively well established in program analysis, in rewriting logic is rather a new topic; (2) other work on slicing in rewriting logic apply on the counterexamples produced by model checking rather than programs, while other program analysis tools for language semantics specifications in rewriting logic target different topics, e.g., testing and debugging.

(A) Program slicing was introduced in [20] where, for a given program with procedures, a limited form of context information (i.e. procedure call location) is used to compute the program slices. The approach resembles an on-demand procedure inlining, using a backward propagation mechanism. The results of [20] are backward program slices. Moreover, multiple procedure calls are included in the computed slice without distinguishing between them w.r.t. their intraprocedural information. Our proposed approach takes into consideration the context-update constructs (as extracted from the formal semantics) and produces forward slices (via term slicing on the program term). Moreover, the context-update constructs induce the symbolic procedure summaries as in [16,8,5]. A procedure summary is a compact representation of the procedure behavior, parameterized by its input values and, in our proposed framework, is derived from the context-update constructs and intraprocedural slicing. The interprocedural slicing is explicit in [8] and implicit in [16,5] and sets the support for interprocedural program analyses. Next we compare our work with the underlying interprocedural slicing algorithm of each of the three aforementioned approaches.

The work in [16] uses a data-flow analysis to represent how the information is passed between procedure calls. It is applied on a restricted class of programs—restricted by a finite lattice of data values—, while the underlying program representation is a mix of control-flow and call graphs. In comparison, our approach considers richer context information (as in [8]) while working on a term representation of a program, i.e., we have an implicit representation of the control-flow and call graphs. The work in [5] keeps the same working structures but addresses the main data limitation of [16]. As such, the procedure
summaries are represented as sets of constraints on the input/output variables. The underlying interprocedural slicing algorithm of [5] is more refined than our approach just because of the sharper representation of context information. Our approach requires a specialized data abstraction on top of the interprocedural slicing procedure—this is one of our proposed developments. We follow closely the work in [5] which introduces a new program representation for interprocedural slicing. This program representation is a mix program graph with explicit constructs for context information, i.e., specialized nodes and edges. In comparison, our approach does not require the explicit context representation but uses term matching (in the second part of the algorithm) to distinguish between different contexts. We present [8] in more details in Section 4.1. What separates our approach from the three aforementioned techniques is that we work with a formal executable semantics, in the rewriting logic framework. This means, on one hand, that the results of our slicing algorithm are correct w.r.t. the language specification and, on the another hand, that the slicing algorithm is generic, as the slicing information is extracted by meta-processing the formal semantics.

(B) In the rewriting logic environment there are several existing approaches towards program debugging, testing, and analysis. The work in [12] presents an approach to generate test cases similar to the one presented here. Namely, both use the formal specifications of the languages semantics to extract information about programs written in these languages. In this case, the semantic rules are used to instantiate the state with the variables in the given program by using narrowing. In this way, it is possible to compute the values of the variables required to traverse all the statements in the program, the so called coverage.

The recent work in [1] proposes a first slicing technique of rewriting logic computations. It takes as input an execution trace—the result of executing Maude model checker tools—and computes dependency relations using a backward tracing mechanism. While they perform dynamic slicing by executing the semantics for an initial given state, we propose a static approach that is centered around the rewriting logic theory of the language specification. Moreover, our main target application is not counterexamples or execution traces of model checkers, but programs executed by the particular semantics.

The technique proposed in here follows our previous work on language-independent program slicing in rewriting logic environment [13]. Actually, the implementation of the current work is an extension of the slicing tool from [13]. Both approaches share the methodology steps: (1) the initial meta-analysis of $S$ and (2) the program analysis conducted over the $S$-programs. Namely, in [13] we use the classical WHILE language augmented with side-effect constructs (assignments and read/write statements) to exemplify (1) the inference of the set of side-effect language constructs in $S$ and (2) program slicing as term rewriting.

As a semantical framework, Maude [2] and rewriting logic [10] have been used to specify the semantics of several languages, such as LOTOS [19], CCS [19], Java [4], or C [9]. These researches, as well as several other efforts to describe a methodology to represent the semantics of programming languages in Maude, led to the **rewriting logic semantics project** [11]—which presents a comprehensive
compilation of these works—and to the development of K [15]—a tool built upon a continuation-based technique that provides mechanisms to (i) ease language definitions and (ii) translate these definitions into Maude which comes with its “for free” analysis tools. Our interest in these semantics consists in using them as benchmarks to validate different meta-analyses as, e.g., the context-update inference. First, we plan to limit our approach to those semantics directly implemented in Maude, but we would like to further extend our attention to the semantics implemented in K and automatically interpreted in Maude.

3 Maude and Language Semantics Specifications

In this section we present some basic notions about the Maude system and an example of a programming language semantics specification in Maude.

3.1 Preliminaries of Maude

Maude modules are executable rewriting logic specifications. Maude functional modules [2, Chap. 4], introduced with syntax fmod ... endfm, are executable membership equational specifications that allow the definition of sorts (by means of keyword sort(s)); subsort relations between sorts (subsort); operators (op) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (assoc) or commutative (comm), for example; memberships (mb) asserting that a term has a sort; and equations (eq) identifying terms. Both memberships and equations can be conditional (cmb and ceq). Maude system modules [2, Chap. 6], introduced with syntax mod ... endm, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (rl) and conditional rules (crl).

An important feature of rewriting logic is that it is reflective, that is, it can be faithfully interpreted in terms of itself. This feature is efficiently implemented in Maude by means of the META-LEVEL module [2, Chapter 14], which allows us to use Maude modules and terms as usual data. Using these features we can traverse Maude modules and terms and study their characteristics. This is specially important when dealing with semantics of other languages implemented in Maude, since we can study both the syntax and semantics (i.e. the Maude modules implementing them) and the programs written and executed using them (i.e. the terms built using the modules). Hence, we can combine both views to study the programs and the way they are executed.

3.2 WhileF - a Maude specification of language semantics

Here we describe an example of a language specification in Maude which we call WhileF—a simple imperative language with functions. In Fig. 1 we give a snapshot of WhileF which emphasizes on the language constructs that we use, in a subsequent section, to describe the context-updates inference.
sorts FunId Fun FunSet Env ESt Var VarL BExp RWBUF Com Statement.
subsorts Fun < FunSet Env < ESt Var < VarL.

op _(_){_} : FunId VarL Com -> Fun.

op nf : -> FunSet [ctor] .
op __: FunSet FunSet -> FunSet [ctor comm assoc id: nf] .

op nd : -> Value [ctor] .
op _=_ : Variable Value -> Env [ctor] .

op mt : -> Env [ctor] .
op __: Env Env -> Env [ctor comm assoc id: mt] .

op _|_ : ESt ESt -> ESt [ctor assoc] .

op nv : -> VarL [ctor] .
op _,_ : VarL VarL -> VarL [ctor assoc id: nv] .

op rm : Env Variable -> Env .
eq [rm1] : rm(mt, X) = mt .
eq [rm2] : rm(X = V ro, X') = if X == X' then ro else X = V rm(ro, X') fi .

op alloc : VarL ESt -> ESt .
eq [al1] : alloc(nv, mu | ro) = mu | ro .
eq [al2] : alloc((X,Vs), mu | ro) = alloc(Vs, mu | rm(ro,X) X = nd) .

op asgP : VarL VarL ESt -> ESt .
eq [aP1] : asgP(nv, Vs, mu | ro) = alloc(Vs, mu | ro) .
ceq [aP2] : asgP((X,Vs), (Y,Ws), mu | ro) = asgP(Vs, Ws, mu | rm(ro,X) Y = V) if V := mu(X) .

op <_,_> : BExp ESt -> Statement .
op <_,_,_,_> : Com ESt RWBUF FunSet -> Statement .

op skip : -> Com .
op _;_: Com Com -> Com .
op While_Do_ : BExp Com -> Com .
op Call_(_) : FunId VarL -> Com .
op Local_ : VarL -> Com .


crl [WR2] : < While b Do C, st, rw, fs > => < skip, st'', rw', fs > if b st => < Y, st'' > \ < C ;(While b Do C), st', rw, fs > => < skip, st'', rw', fs > .

Fig. 1. Snapshot of WhileF—a Maude specification for a small imperative language.

The language syntax in WhileF is maintained in the sort Com, while the program state is formed by the sorts ESt, RWBUF, and FunSet. The terms in FunSet define the set of the function/procedures definitions in the program represented as Name(Params) {Body} of sort Fun. The sort ESt contains the environments of a program while the sort RWBUF contains the read/write buffer (used in the semantics of read/write instructions). Note that ESt is a list of environments ro : Env linked by the operator _|_ while the sort Env is a set of associations between variables and their values, e.g., v = n.

The operator <_,_,_,_> : Com ESt RWBUF FunSet -> Statement defines the semantics of each instruction in the language. In the snapshot above we give the rules for the while statement (i.e., the operator While_Do_), for local variables declaration (i.e., Local_), and for the function call (i.e., Call_(_)).

The rules [WR1] and [WR2] specify the semantics of the while construct w.r.t. the evaluation of the loop condition, i.e., variable b. In [WR1] the boolean expression b : BExp is evaluated to F (false) by the operator <_,_> so the result of the evaluation is the state containing the environment after the evaluation of b, i.e., the variable st', while the read-write buffer remains the same, i.e., the
variable \( rw \). In \([WR2]\) the value of \( b \) is evaluated to \( T \) (true) and the while body \( C \) is evaluated in the condition of \([WR2]\) following a big-step style. The new state resulted from this evaluation (i.e. variables \( st'' \) and \( rw' \)) is returned.

The declaration of local variables is defined in \( [Loc] \). The new environment \( st' \) is given by the operator \( alloc(_,_) \), a simple recursive operator described by the equations \([al1]\) and \([al2]\). In \([al1]\) the recursion stops because it finds the void list of variables, given by the constant \( nv : \text{VarL} \), while in \([al2]\) the head of the variable list \( X \) is added to \( ro : \text{Env} \), i.e., to the last environment in the list of sort \( \text{ESt} \). Note that the new value of \( X \) is \( \text{nd} \), i.e., \( \text{undefined} \), which overrides any previous value of \( X \) in \( ro \) via the operator \( \text{rm}(ro,X) \).

The rule \( [CFn] \) describes the semantics for calling a function \( fn \) with the parameter values \( \text{actPrms} \). As in the case of rule \([WR2]\), the semantic-execution of the function body takes place in the condition of the rule. First, the function name \( fn \), the parameters \( \text{Prms} \), and the body \( C \) are identified in \( fs : \text{FunSet} \)—the set of functions in the program. Second, the local environment \( lenv \) is created with the function parameters being updated with the actual parameter values. Third, the body is executed using the current environment (both local \( lenv \) and the global one \( st \)), yielding a new state (the variables \( st', lenv', rwb', \) and \( fs \)). The function return is included in this rule as the updated local environment \( lenv' \) is dropped when the rule returns the new program state.

### 4 Semantics-based Interprocedural Slicing

In this section we discuss program slicing with preliminaries—explained on an WhileF program—and our approach which consists in two steps: the meta-analysis and the program slicing as term slicing. We present the meta-analysis step on the WhileF language case study then we describe the run of the second step in our slicing—the term slicing—on the given WhileF program.

#### 4.1 Preliminaries of interprocedural program slicing

Program slicing, as introduced in \([20]\), is a program analysis technique which computes all the program statements that might affect the value of a variable \( v \) at a program point of interest, \( \pi \). It is a common setting to consider \( \pi \) as the last instruction of a procedure or of the entire program. Hence, without restricting the proposed methodology, here we consider slices of the entire program.

A classification of program slicing techniques identifies \textit{intraprocedural} slicing, when the method is applied on a procedure body, and \textit{interprocedural} slicing, when the method is applied across procedure boundaries. The key element of a methodology for interprocedural slicing is the notion of context (i.e. the values of the function/procedure parameters). Next, we elaborate on how context-aware program slicing produces better program slices than a context-forgetful one.

Let us consider, in Fig. 2 the program from \([5]\) written as an WhileF program term upon which we present subtleties of interprocedural slicing. We start the slicing with the set of variables of interest \( \{z\} \).
The first method, from [20], resembles an on-demand inlining of the necessary procedures. In the example in Fig. 2, the variable \( z \) is an argument of the call to procedure \( \text{Add} \) in \( \text{Inc} \) hence, the sliced body of \( \text{Add} \) is included in the slice of \( \text{Inc} \). Note that, when slicing the body of \( \text{Add} \), \( z \) is replaced by \( a \) hence, the slicing of \( \text{Add} \) deems \{a\} and \{b\} as relevant. The return statement of procedure \( \text{Inc} \) is paired with the call to \( \text{Inc} \) in the body of \( A \) so, the variable \{y\} becomes relevant for the computed slice. When the algorithm traces the source of the variable \( y \), it finds the second call to \( \text{Add} \) in the body of \( A \) (with the arguments \( x \) and \( y \)) and includes it in the program slice. When tracing the source of \( x \) and \( y \), it leads to include the entire body of procedure \( \text{Main} \) (through the variables \( \text{sum} \) and \( i \) which are used by the assignments and calls of \( \text{Main} \)). With this method, the program slice w.r.t. the set of variables of interest \{z\} is the original program, as in Fig. 2. This particular slice is a safe over-approximation of a more precise one (which we present next) because the method relies on a transitive-closure, i.e., a fixpoint computation where all the variables of interest are collected at the level of each procedure body. As such, the body of procedure \( \text{Add} \) is included twice in the computed slice.

```
function Main () {
  sum := 0;
  Local i;
  i := 1;
  while i < 11 do
    call A (sum, i)
  }
}
function A (x, y) {
  call Add (x, y);
  call Inc (y)
  i := 1;
  call Add (z, i)
}
function Add (a, b) {
  a := a + b
}
function Inc (z) {
  Local i;
  i := 1;
  call Add (z, i)
}
```

Fig. 2. A WhileF program \( Px \) with procedures \( \text{Main}, A, \text{Add}, \text{and Inc} \)

The second approach in [8] exploits, for each procedure call, the available information w.r.t. the program variables passed as arguments (i.e. the existing context before the procedure call). Again, in the example in Fig. 2, the variable \( z \) is an argument of procedure \( \text{Add} \) hence, upon the return of \( \text{Add} \), its body is included in the slice. However, because of the data dependencies between the variables \( a \) and \( b \) (with \( a \) using an unmodified value of \( b \)) only the variable \( a \) is collected and further used in slicing. Next, upon exiting from \( \text{Add} \) and then \( \text{Inc} \), the call of \( \text{Inc} \) in \( A \) (with parameter \( y \)) is included in the slice. Note that the call to \( \text{Add} \) from \( A \) (with parameters \( x \) and \( y \)) is not included in the slice because it does not modify the context (i.e. the variables of interest at the call point in \( A \)). As such, the slicing algorithm collects only the second parameter of procedure \( A \) and, following the call to \( A \) in \( \text{Main} \), it discovers \( i \) as the variable of interest (and not \( \text{sum} \) as it was the case in the previous method). Hence, the sliced \( A \) with only the second argument is included in the computed slice. Consequently, the variable \( \text{sum} \) from \( \text{Main} \) is left outside the slice. The result is presented in Fig. 3.

Any analysis that computes an interprocedural slice works with the control-flow graph—which captures the program flow at the level of procedures—and the
Fig. 3. The result of a context-dependent interprocedural analysis for \( P_x \) call graph—which represents the program flow between the different procedures. To improve the precision of the computed program slice, it is necessary for the analysis to use explicit representations of procedure contexts (as special nodes and transitions). This is the case of the second method which relies on a program representation called *system dependence graph*.

### 4.2 Meta-analysis for context-updates

Now, we present several notions related to the meta-analysis step of our slicing, i.e., the analysis of the language semantics specification. We consider \( S \) a specification of a programming language semantics given in Maude. We make the distinction between the languages syntax and the program state via their different sorts in \( S \), i.e., \( \text{Com} \) and respectively the pair \( (\text{ESt}, \text{RWBUF}) \). The meta-analysis described in this section is developed in Maude’s metalevel.

We assume the following (standard) transformations over \( S \): All the equations are directed from left to right such that they become rewrite rules. Then, we assume a unique label \([R]\) for each such rule where all the matchings in the conditions \( v := e \) remain the same but the equations are transformed into rewrites. Finally, we denote \( \bar{S} \) the specification \( S \) after these transformations.

Let \( t \) be a term with variables and \( R \) denoting the rule

\[
[R] \ l \Rightarrow r \text{ if } \chi(l_1,r_1) \wedge \ldots \wedge \chi(l_n,r_n),
\]

where \( \chi(A,B) \) is either the matching \( A := B \) or the rewriting \( A \Rightarrow B \), and \( n \) is a positive number. We use the notation \( R :: t \) for “an instance of \( t \) on a non-variable position can be reduced by the rule \( R \)”, i.e., there exist an unifier \( \theta \) and a subterm \( s \) of \( t \) such that \( s\theta = l\theta \).

**Definition 1.** A hypernode for a valid term \( t \), denoted as \( \forall R \in \bar{S}, R :: t \), is a list \( [R_1 \rightarrow \ldots \rightarrow R_m] \) of distinct rules in \( \bar{S} \), with \( m \in \mathbb{N} \), such that \( R_i :: t \) for all \( 1 \leq i \leq m \). We define an inspection tree \( iT \) as a tree of hypernodes where the children of a hypernode \( [R_1 \rightarrow \ldots \rightarrow R_m] \) are defined as:

\[
\text{children}([R_1 \rightarrow \ldots \rightarrow R_m]) = \{ \text{successors}(R_i) \mid 1 \leq i \leq m \}
\]

where the successors of a rule \([R] \ l \Rightarrow r \text{ if } \chi(l_1,r_1) \wedge \ldots \wedge \chi(l_n,r_n)\) are defined as:
successors($R$) = $\forall R_1 \in \mathcal{S}, R_1 :: l_1 \rightarrow \ldots \rightarrow \forall R_n \in \mathcal{S}, R_n :: l_n \rightarrow \forall R_r \in \mathcal{S}, R_r :: r$

We exemplify the meta-analysis of context-updates on the WhileF specification. First, we identify all operators in WhileF which are list connectors simply based on their signature and attributes, i.e., no commutativity. For example, the operators `_` : ESt ESt -> ESt and `_`, `_` : VarL VarL -> VarL are list connectors. Next, we filter the list connectors based on the sort they construct and we keep only those for the program state ESt. Hence, only the operator `_` remains of interest for the meta-analysis of WhileF. Next, we explain how we use the inspection tree of WhileF to check that the operator `_` is responsible for stacking up environments hence, inferring the context-update constructs.

We make the assumptions that the context-update constructs have the standard behavior of creating new environments on top of the previous ones in a stack-like fashion. Namely, upon entering a new context, the context-update adds a new environment at the top of the environments list and, upon exiting the context, the environment is removed from the top of environments list. Hence, the list produced by the operator `_` has a stack-like behavior and this behavior is triggered by the context-update construct. As such, the meta-analysis phase constructs the inference tree and uses colors to label the visited rewrite rules. This coloring is used to classify the rewrite rules while identifying which of them induce the aforementioned changes in the environments list. We set a counter $\ell$ which identifies the level of the stack and we denote as context-entering $\ell \uparrow$ the increment of the counter, and context-exiting $\ell \downarrow$ a decrement of the counter. The notation $\ell \uparrow \downarrow$ is used for a context-enter followed by context-exit at the same level of $iT$ and we name this context-update. We use the following color convention:

- red - those nodes under classification (i.e. unclassified at the current point of the execution);  
- green - those nodes which are classified as not producing context-update; 
- orange - those nodes which could result in context-update; 
- blue - special nodes which are used to cut the recursive calls.

The meta-analysis terminates when all the tree nodes are visited/classified.

Let us consider the inspection tree for a term `<com, est, rwbuf, funset>` where the sorts of the variables com, est, rwbuf, and funset are Com, ESt, RWBUF, and FunSet, respectively. Hence, the top-level hypernode contains the rewrite rules included in our semantics snapshot from Fig. 1—the rules for while, WR1, WR2, the declaration of local variables, Loc, and the function call, CFn.

We start by unfolding the rule WR1. So $l_1$ is the term `< b, st >`, with the variables b:BExp and st:ESt, $r_1$ is the term `< F, st >`, while $r$ is the term `< skip, st, rw, fs >`, where rw:RWBUF and fs:FunSet. Consequently, successors(WR1) is the list of hypernodes $\forall R \in \mathcal{S}, R :: l_1 \rightarrow \ldots \rightarrow \forall R \in \mathcal{S}, R :: r$. However, $r$ is in the normal form of the terms of sort Statement because it contains the constant skip as com and there is no rule in WhileF for rewriting a skip-statement. Hence, successes(WR1) unfolds, via $l_1$, only into the hypernode `[BVR->BOR-> ...]`. The rule `[BVR] : < bx, st > = > < st(bx), st >`. 
Irina Măriuca Asăvoae, Mihail Asăvoae, Adrián Riesco

(where \( bx: \text{BVar} \) is a boolean variable) handles the evaluation of boolean variables based on their associated value in the environment. Note that \( \text{st}(bx) \) represents the look-up operation, i.e., finding the value of the variable \( bx \) in \( \text{st:ESst} \). Meanwhile, the rewrite rule \( \text{BOR} \) addresses the comparison operations between boolean variables but, for brevity, we do not include here the its specification.

In the next iteration the current node is \( \text{BVR} \), with the color-code red. The unfolding of \( \text{BVR} \) produces the hypernode \( \forall R \in \mathcal{S}, R :: r \) where \( r \) is the term \( < \text{st}(bx), \text{st} > \). The rules matching a subterm of \( r \) are the look-up rules which match the operator \( _\cdot(\_\cdot) : \text{ESst Var} \to \text{Num} \) appearing in the current \( r \):

\[
\begin{align*}
[lkp1] & : (\mu | X = V \text{ ro}) (X) \Rightarrow V. \\
[lkp2] & : (X = V \text{ ro}) (X) \Rightarrow V. \\
[lkp3] & : (\mu | \text{ ro}) (X) \Rightarrow \mu (X) [\text{wise}] .
\end{align*}
\]

where \( \mu: \text{ESst}, \text{ro:Env}, X: \text{Variable} \), and \( V: \text{Value} \). Hence, \( iT \) becomes:

\[
\begin{align*}
\text{WR1} & \to \text{WR2} \to \text{Loc} \to \text{CFn} \to \ldots \\
\text{BVR} & \to \text{BOR} \to \ldots \\
\text{lkp1} & \to \text{lkp2} \to \text{lkp3}
\end{align*}
\]

We observe that the unfoldings of \( \text{lkp1} \) and \( \text{lkp2} \) do not produce any hypernodes as the term \( r \) for these rules is in normal form. Next, the rule \( \text{lkp3} \) is labeled with \( \ell \downarrow \) as the stack size is decremented in this rule (from \( \mu | \text{ ro} \) to \( \mu \)) and the rule \( \text{lkp3} \) unfolds into the hypernode \( \forall R \in \mathcal{S}, R :: \mu(X) \) i.e., \( \text{lkp1} \to \text{lkp2} \to \text{lkp3} \) again. The rules in this hypernode are colored \textit{blue} as they are already considered in \( iT \) so the unfolding stops here for this hypernode. Hence, \( \text{lkp3} \downarrow \) is colored with \textit{orange} as it produces stack changes. However, the orange color is not propagated to \( \text{BVR} \) because the environment changes from \( \text{lkp3} \) are transparent to \( \text{BVR} \) as its environment \( \text{st} \) is kept intact from the left-hand-side to the right-hand-side of the rule. We label this transparency with “of” as depicted in the following \( iT \):
Note that the unfolding of WR2 is similar with the already described one of WR1 with only an additional blue hypernode containing \(WR1\rightarrow WR2\rightarrow Loc\rightarrow CFn\rightarrow\ldots\) produced by the term \(<C; (While\ b\ Do\ C),\ st',\ rw,\ fs>\). Still, no context change is detected for the counter \(\ell\). Furthermore, the unfolding of Loc produces the hypernode \(all\rightarrow al2\) which in turn has only one descendent, due to \(al2\), \(all\rightarrow al2\rightarrow rm1\rightarrow rm2\) but none of the descendants produces modifications to \(\ell\).

Finally, the unfolding of CFn produces firstly two hypernodes \(aP1\uparrow\rightarrow aP2\uparrow\) and \(WR1\uparrow\rightarrow WR2\uparrow\rightarrow\ldots\) corresponding to \(\forall R\in S, R::\text{asgP}(\text{actPrms}, Prms, st|mt)\) and \(\forall R\in S, R::<C, st|\text{env}, rw, fs>\), respectively. Note that all the rules in these hypernodes are labeled with \(\ell\uparrow\) because their terms of sort \(\text{ESt}\) receive an additional element in comparison with the \(\text{ES}t\) term from the \(\ell\) term of the rule CFn. The second hypernode is colored blue as it was already unfolded. We draw in the above iT the whole unfolding of the hypernode \(aP1\uparrow\rightarrow aP2\uparrow\) but we do not insist on explaining details of it since, by now it should be superfluous.

After the entire unfolding of CFn is finished, upon processing the hypernode \(\forall R\in S, R::r\) we label it with \(\ell\downarrow\) (actually we label the connectig arrow as this hypernode is void). Hence, in CFn we have \(\ell\downarrow\), i.e., context-entering, followed by \(\ell\uparrow\), i.e., context-exiting, which induces the labeling of CFn with \(\ell\uparrow\downarrow\), i.e. context-update. After the semantics analysis step is applied to the WhileF language, we identify only \(\text{Call}_\_(_\_\_\_\_)\) in \(c\), the set of context-updates constructs.

### 4.3 Program slicing as term slicing

The second slicing step is a fixpoint iteration which applies the current slicing criterion over the program term in order to discover new function-subterms of the program that use the slicing criterion. In [13] we give a semantics-based intraprocedural slicing, denoted \(\sqrt{S/f}\), where \(f\) is a function and \(sc\) is the slicing criterion, i.e., the subset of program variables of interest. The input of the second step of our slicing method is formed by \(p\), the term representation of the program and the slicing criterion \(sc\) together with the set of context-updates \(c\). Along this step, we use the intraprocedural slicing \(\sqrt{S/f}\) for any \(f\) in \(p\).
The context-update terms of the program $p$, i.e. $\text{Call } fn \ (\_)$, direct the slicing towards applying $\sqrt{S}/f$, where $f$ is the definition of the function named $fn$ in the program $p$. Namely, when a context-update from $c$ is encountered for some function, we proceed to a round of intraprocedural slicing which takes into consideration the new scope for this procedure, i.e., the new slicing criterion. We iterate this until the slicing criterion remains unchanged, so no new subterms can be discovered and added to the result, i.e., the slice set. For example, the iterations of the second slicing step for program $P_2$ in Fig. 2 and the slicing criterion \{z\} are listed in Fig. 4. Next, we explain this table.

In iteration 1 from Fig. 4 the slicing starts with variable $z$ and, while going through all the functions in the program $P_2$ with the intraprocedural slicing $\sqrt{S}/f$, it finds an initial slice for function $\text{Inc}$. This slice is given in the column Computed slice and we observe that it maintains only the first parameter of the function $\text{Add}$ since the other parameter is not part of the currently computed slicing criterion, i.e., $i$ has no dependency with the slicing variable $z$. Note that the rows 2, 3, and 4 are grouped together in one box in the table from Fig. 4 with the intended meaning that all these three rows are part of the same fixpoint iteration, i.e., the second iteration.

Firstly, the fixpoint submits to slicing the functions containing context-updates, i.e., calls to functions from the previous slicing iteration, i.e., $\text{Inc}$. Hence, the function $A$ is added to the slice in row 2 as it contains call $\text{Inc}(z)$. The slice of $A$ with the slicing criterion $y$, i.e., $\sqrt{S}/A$, is given in the last column of row 2. We observe that the call to $\text{Add}(y)$ does not appear in the slice of $A$. The reason is that the slice of $\text{Add}$ with the slicing criterion \{b\} (obtained from $y$ in $A$ due to parameter substitution) is empty because there is no variable changing $b$ in $\text{Add}$. Hence, call $\text{Add}(y)$ is not included in the slice for $A$.

Secondly, in row 3, all the context-update elements from the current slice are added to the slicing processing. Note that in Slicing variables we maintain the current variables used for slicing and their replacements in various scopes. As such, $\text{Add}$ is added to the slice computation with only one parameter $a$ which is induced by the actual parameter $z$ from the call to $\text{Add}$ in the slice computed at iteration 1, i.e., call $A(z)$. The substitution of the actual parameter in the

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Slicing variables</th>
<th>Function scopes</th>
<th>Computed slice (identified subterms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$z$</td>
<td>${z} \leadsto \text{Inc}$</td>
<td>$\text{Inc}(z) {\text{call Add}(z)}$</td>
</tr>
<tr>
<td>2</td>
<td>$z/y$</td>
<td>${y} \leadsto A$</td>
<td>$A(y) {\text{call Inc}(y)}$</td>
</tr>
<tr>
<td>3</td>
<td>$z/a$</td>
<td>${a} \leadsto \text{Add}$</td>
<td>$\text{Add}(a, b) {a := a + b}$</td>
</tr>
<tr>
<td>4</td>
<td>$z/a, z/b, i$</td>
<td>${z, i} \leadsto \text{Inc}$</td>
<td>$\text{Inc}(z) {\text{Local } i; \ i := 1; \ \text{call Add}(z, i)}$</td>
</tr>
<tr>
<td>5</td>
<td>$z/y, i$</td>
<td>${i} \leadsto \text{Main}$</td>
<td>$\text{Main}() {\text{Local } i; \ i := 1; \ \text{call A}(i)}$</td>
</tr>
</tbody>
</table>
function call \texttt{call A(z)} with the formal parameter \texttt{a} from the definition of \texttt{z} is represented as \texttt{z/a}. Further, the slicing for function \texttt{Add} brings into the slicing variables set the second formal parameter of \texttt{Add}, i.e., the variable \texttt{b}.

Thirdly, in row 4, the slice for \texttt{Inc} is recomputed with the new slicing criterion \{\texttt{z,i}\} and the new slice of \texttt{Inc} is given in the last column. Note that \texttt{i} was added to the slice because of the formal/actual parameter substitution, denoted as \texttt{b/i} where \texttt{b} is a part of the slicing criterion \{\texttt{a,b}\} computed for \texttt{Add}, i.e., \texttt{a} from \texttt{Add} becomes \texttt{z} in \texttt{Inc} while \texttt{b} becomes \texttt{i}, via parameter substitution.

The fixpoint algorithm continues by adding to the slice the functions that contain a call to any of the functions added to the slice in the second iteration. Hence, in the third iteration, row 5, the function \texttt{Main} is added to the slice with the slicing variable \texttt{i} which comes from the actual/formal parameter substitution \texttt{z} from \texttt{Inc} with \texttt{y} in \texttt{A}, and \texttt{y} from \texttt{A} with \texttt{i} in \texttt{Main} (denoted as \texttt{z/y/i}). The slice computed for function \texttt{Main} is given in the last column of row 5. Finally, the fixpoint is reached as no other new call site exists for the functions currently added to the slice, i.e., \texttt{Inc, Add, A}, and \texttt{Main}.

The fixpoint algorithm—the second slicing step—terminates because there exists a finite set of program subterms and a finite number of slicing criterions for each context-update operator. Note also that there are finitely many context-updates in the program because the slicing in this step is syntactical w.r.t. the program term. This algorithm produces a valid slice, because it exhaustively saturates the slicing criterion but the slice produced is not minimal. Note however that for \texttt{Px} from Fig. 2 we obtain a minimal slice, i.e., the slice in Fig. 3.

5 Concluding Remarks and Ongoing Work

The formal language definitions based on the rewriting logic framework support program executability and create the premises for further development of program analyzers. In this paper we presented a generic interprocedural slicing method which uses the information produced by a meta-level analysis of the language semantics (i.e. context-update constructs) to ensure the generality. In turns, the actual program slice computation is delivered as term slicing. This work intends to complement the recent advances in semantics-constructed tools for debugging [14], automated testing [12], and program analysis [13].

Our immediate working plan is to extend our existing Maude implementation for the intraprocedural slicing from [13] to accommodate the proposed algorithm for interprocedural slicing. While the two slicing analyses share a common general structure (with a generic semantics-based processing engine and a program slicing using term slicing), the implementation of the interprocedural analysis is more challenging due to context-updates. We plan to implement a memoization mechanism to keep track, for a particular procedure, what are the calling contexts, as well as the most general context, and to facilitate symbolic representations of procedure summaries, as in [5].
References

Abstract. Nominal sets, introduced to Computer Science by Gabbay and Pitts, are useful for modeling computation on data structures built of atoms that can only be compared for equality. In certain contexts it is useful to consider atoms equipped with some nontrivial structure that can be tested in computation. Here, we study nominal sets over atoms equipped with both relational and algebraic structure. Our main result is a representation theorem for orbit-finite nominal sets over such atoms, a generalization of a previously known result for atoms equipped with relational structure only.

1 Introduction
Nominal sets [10] are an elegant algebra of name binding. Given an infinite set \( A \) of atoms and the group \( G \) of all its permutations, one can consider an arbitrary \( G \)-set \( X \) (i.e. a set equipped with an action of the group \( G \)) and study the relations between the canonical action of \( G \) on \( A \) and the action of \( G \) on \( X \). Examples of \( G \)-sets include:

- the set \( A \) itself,
- the set \( A^n \) of \( n \)-tuples of atoms,
- the set \( A^{(n)} \) of \( n \)-tuples of distinct atoms,
- the set \( A^* \) of finite words over \( A \), etc.

Each of those sets is acted upon by permutations of the atoms in a natural way, by renaming all atoms that appear in it. For a \( G \)-set \( X \) to be nominal we require the result of applying a permutation of atoms to each of its elements to be determined by a finite set of atoms, called a *support* of this element. Sets \( A \), \( A^n \) and \( A^{(n)} \) are nominal, since each tuple of atoms is supported by the finite set of atoms that appear in it. Another example of a nominal set is the set \( A^* \), where a word is supported by the set of its letters.

Nominal sets were introduced in 1999 by Gabbay and Pitts [5]. Their idea was to use an infinite set of atoms to describe variable names in programs or
logical formulas. In this setting, the permutations of the atoms correspond to renaming of variables. For example, in λ-calculus nominal sets provide a nice mathematical model of α-conversion. In the 90s nominal sets were also implicitly used by Pistore [9] and later on by Tzevelekos [11] in automata theory, under the name of *named sets with symmetries*. The motivation here is that atoms can be used to model sources of infinite data.

In [1] and [2] Bojańczyk, Klin and Lasota noticed that nominal sets provide a more relaxed notion of finiteness, namely *orbit-finiteness*, which can be used to deal with infinite systems such as automata over infinite alphabets. A nominal set is considered orbit-finite if it has finitely many elements, up to permutations of atoms. Such a set can be represented in a concrete, finite way [1]. An example of an orbit-finite set is the set of configurations of a register automaton by Francez and Kaminski [8], which recognizes languages such as “the first letter does not appear any more”. Using orbit-finite sets in the setting of automata with data gives results similar to those by Pistore et al.

Atoms turn out to be a good framework to speak of data that can be accessed only in a limited way. Nominal sets, as defined in [10], intuitively correspond to data with no structure except for equality. To model a device with more access to its alphabet one may use atoms with additional structure.

In [2] atoms are modelled as countable relational structures and considered together with their groups of automorphisms. The definition of a nominal set remains essentially the same. One only needs to replace the group $G$ of all permutations of the set of atoms by the automorphism group. A choice of such automorphisms is called an *atom symmetry*. This setting is more general. Atoms with equality only give rise to “classical” nominal sets. But one can also consider e.g. atoms with order. A typical language recognized by a nominal automaton over such atoms is the language of all monotonic words.

Since interesting nominal sets are usually infinite, to manipulate them effectively we need to represent them in a finite way. In [2] Bojańczyk et al. provide a concrete, finite representation of orbit-finite nominal sets for atoms that are a homogeneous relational structures over finite vocabularies. The representation theorem is obtained using the theory of Fraïssé limits (see e.g. [7]).

Sometimes a relational structure of atoms is not enough. In [3] Bojańczyk and Lasota use the theory of nominal sets to obtain a machine-independent characterization of the languages recognized by deterministic timed automata. To do so they introduce atoms with a function symbol $+1$ and relate deterministic timed automata to automata over these *timed atoms*. An example of a language recognized by such a nominal automaton is the set of all words where the distance between any two consecutive letters is smaller than 1. It is therefore natural to ask if the representation theorem can be generalized to cover also structures with function symbols. Our main result gives a positive answer to this question. We see this as an initial step in a theory of nominal sets over atoms with algebraic structure.

---

1 The equivalence between named sets and nominal sets was proven in [4] and [6].
The proof of the representation theorem for symmetries with function symbols follows the same pattern as the proof given in [2]. There are, however, some subtleties, since instead of finite supports one has to consider finitely generated supports (which can be infinite). As a result it is not always obvious how to define the counterparts of some key notions like fungibility.

The structure of this paper is as follows. In Section 2 we define atom symmetries and introduce the category of $G$-sets. In Section 3, following [1, 2], we focus on the theory of nominal sets for Fraissé symmetries, introduce the category of nominal sets, and explain the notion of least finitely generated support. In Section 4 we define the property of fungibility and finally prove the representation theorem for fungible Fraissé symmetries that admit least supports.

Acknowledgments. I would like to thank Mikołaj Bojańczyk, Bartek Klin, Sławomir Lasota and Szymon Toruńczyk for inspiring discussions and helpful remarks on earlier drafts of this paper, and the anonymous referees for many insightful comments.

2 Atom symmetries

A (right) group action of a group $G$ on a set $X$ is a binary operator $\cdot : X \times G \rightarrow X$ that satisfies following conditions:

for all $x \in X$ $x \cdot e = x$, where $e$ is the neutral element of $G$,

for all $x \in X$ and $\pi, \sigma \in G$ $x \cdot (\pi \sigma) = (x \cdot \pi) \cdot \sigma$.

The set $X$ equipped with such an action is called a $G$-set.

Example 2.1. For a set $X$ let $\text{Sym}(X)$ denote the symmetric group on $X$, i.e. the group of all bijections of $X$. Take any subgroup $G$ of the symmetric group $\text{Sym}(X)$. There is a natural action of the group $G$ on the set $X$ defined by $x \cdot \pi = \pi(x)$.

Definition 2.2. An atom symmetry $\langle A, G \rangle$ is a set $A$ of atoms, together with a subgroup $G \leq \text{Sym}(A)$ of the symmetric group on $A$.

Example 2.3. Examples of atom symmetries include:

- the equality symmetry, where $A$ is a countably infinite set, say the natural numbers, and $G = \text{Sym}(A)$ contains all bijections of $A$,

- the total order symmetry, where $A = \mathbb{Q}$ is the set of rational numbers, and $G$ is the group of all monotone permutations,

- the timed symmetry, where $A = \mathbb{Q}$ is the set of rational numbers, and $G$ is the group of all permutations of rational numbers that preserve the order relation $\leq$ and the successor function $x \mapsto x + 1^2$.

\footnote{The timed symmetry was originally defined in [3] for $A = \mathbb{R}$. Considering the rational numbers instead of the reals makes little difference but is essential for our purposes, since we need the set of atoms to be countable.}
For any element $x$ of a $G$-set $X$ the set $x \cdot G = \{ x \cdot \pi \mid \pi \in G \} \subseteq X$ is called the orbit of $x$. Orbits form a partition of $X$. The set $X$ is called orbit-finite if the partition has finitely many parts. Each of the orbits can be perceived as a separate $G$-set. Therefore we can treat any $G$-set $X$ as a disjoint union of its orbits.

Example 2.4. For any atom symmetry $(A, G)$ the action of $G$ on $A$ extends pointwise to an action of $G$ on the set of tuples $A^n$. In the equality symmetry, the set $A^2$ has two orbits: $\{(a, a) \mid a \in A\}$ and $\{(a, b) \mid a \neq b \in A\}$. In the timed symmetry, the set $A^2$ is not orbit-finite. Notice that for any $a \in \mathbb{Q}$ each of the elements $(a, a + 1), (a, a + 2), \ldots$ is in a different orbit.

Let $X$ be a $G$-set. A subset $Y \subseteq X$ is equivariant if $Y \cdot \pi = Y$ for every $\pi \in G$, i.e., it is preserved under group action. Considering a point-wise action of a group $G$ on the Cartesian product $X \times Y$ of two $G$-sets $X, Y$ we can define an equivariant relation $R \subseteq X \times Y$. In the special case when the relation is a function $f: X \to Y$ we obtain a following definition of an equivariant function

$$ f(x \cdot \pi) = f(x) \cdot \pi \text{ for any } x \in X, \pi \in G. $$

The identity function on any $G$-set is equivariant, and the composition of two equivariant functions is again equivariant, therefore for any group $G$, $G$-sets and equivariant functions form a category, called $G$-Set.

Definition 2.5. For any $x$ in a $G$-set $X$, the group $G_x = \{ \pi \in G \mid x \cdot \pi = x \} \leq G$ is called the stabilizer of $x$.

Lemma 2.6. If $H \leq G$ is the stabilizer of an element $x$ of a $G$-set $X$ then $\pi^{-1}H\pi$ is the stabilizer of $x \cdot \pi$ for each $\pi \in G$.

Proof. Let $K \leq G$ be the stabilizer of $x \cdot \pi$. Obviously $\pi^{-1}H\pi \subseteq K$. On the other hand, $x \cdot (\pi \sigma \pi^{-1}) = x$ for any $\sigma \in K$. Hence $\pi K \pi^{-1} \subseteq H$, which means that $K \subseteq \pi^{-1}H\pi$. As a result $K = \pi^{-1}H\pi$, as required.

Proposition 2.7. Let $x$ be an element of a single-orbit $G$-set $X$. For any $G$-set $Y$ equivariant functions from $X$ to $Y$ are in bijective correspondence with those elements $y \in Y$ for which $G_x \leq G_y$.

Proof. Given an equivariant function $f: X \to Y$, let $y = f(x)$. If $\pi \in G_x$ then

$$ y \cdot \pi = f(x) \cdot \pi = f(x \cdot \pi) = f(x) = y, $$

hence $G_x \leq G_y$. On the other hand, given $y \in Y$ such that $G_x \leq G_y$, define a function $f: X \to Y$ by $f(x \cdot \pi) = y \cdot \pi$. Function $f$ is well-defined. Indeed, if $x \cdot \pi = x \cdot \sigma$ then $\pi \sigma^{-1} \in G_x \subseteq G_y$, hence $y \cdot \pi = y \cdot \sigma$.

It is easy to check that the two above constructions are mutually inverse.
3 Fraïssé symmetries

In the following, we shall consider atom symmetries that arise as automorphism groups of algebraic structures. Such symmetries behave particularly well if those structures arise as so-called Fraïssé limits, which we introduce in this sections.

Fraïssé limits. A signature is a set of relation and function symbols together with (finite) arities. A structure \( A \) over a fixed signature is a set, called a carrier of \( A \), equipped with functions and relations that correspond to symbols in the signature. We will consider structures over a fixed finite signature.

For two structures \( A \) and \( B \), an embedding \( f : A \to B \) is an injective function from the carrier of \( A \) to the carrier of \( B \) that preserves and reflects all relations and functions in the signature. A bijective embedding is called an isomorphism. A class \( K \) of structures over some fixed finite signature is closed under isomorphisms if for each structure \( A \in K \) it contains all structures isomorphic to \( A \).

Let \( B \) be a subset of the carrier of a structure \( A \). The smallest substructure of \( A \) containing \( B \) is called a substructure generated by \( B \). A substructure \( B \) of \( A \) is said to be finitely generated if there exists a finite set \( B \) that generates it.

Definition 3.1. A class \( K \) of finitely generated structures over some fixed signature is called a Fraïssé class if it:

- is closed under isomorphisms as well as finitely generated substructures and has countably many members up to isomorphism,
- has joint embedding property: if \( A, B \in K \) then there is a structure \( C \) in \( K \) such that both \( A \) and \( B \) are embeddable in \( C \),
- has amalgamation: if \( A, B, C \in K \) and \( f_B : A \to B \), \( f_C : A \to C \) are embeddings then there is a structure \( D \) in \( K \) together with two embeddings \( g_B : B \to D \) and \( g_C : C \to D \) such that \( g_B \circ f_B = g_C \circ f_C \).

Examples of Fraïssé classes include:

- all finite structures over an empty signature, i.e. finite sets,
- finite total orders,
- all finite structures over a signature with a single binary relation symbol, i.e. directed graphs,
- finite Boolean algebras,
- finite groups,
- finite fields of characteristic \( p \).

Some Fraïssé classes admit a stronger version of amalgamation property. We say that a class \( K \) has strong amalgamation if it has amalgamation and moreover, \( g_B \circ f_B(A) = g_C \circ f_C(A) = g_B(B) \cap g_C(C) \). It means that we can make amalgamation without identifying any more points than absolutely necessary.

Example 3.2. All the Fraïssé classes listed above, except for the class of finite fields of characteristic \( p \), have the strong amalgamation property.
The age of a structure $U$ is the class $K$ of all structures isomorphic to finitely generated substructures of $U$. A structure $U$ is homogeneous if any isomorphism between finitely generated substructures of $U$ extends to an automorphism of $U$. The following theorem says that for a Fraïssé class $K$ there exists a so-called universal homogeneous structure of age $K$. We shall refer to it also as the Fraïssé limit of the class $K$ (see e.g. [7]).

**Theorem 3.3.** For any Fraïssé class $K$ there exists a unique, up to isomorphism, countable (finite or infinite) structure $U^K$ such that $K$ is the age of $U^K$ and $U^K$ is homogeneous.

**Example 3.4.** The Fraïssé limit of the class of finite total orders is $\langle Q, \leq \rangle$. For finite Boolean algebras it is the countable atomless Boolean algebra.

A structure $U$ is called weakly homogeneous if for any two finitely generated substructures $A$, $B$ of $U$, such that $A \subseteq B$, any embedding $f_A : A \to U$ extends to an embedding $f_B : B \to U$. It turns out that a countable structure $U$ is homogeneous if and only if it is weakly homogeneous (see [7]). Hence, one way to obtain a Fraïssé class $K$ is to take a weakly homogeneous, countable structure $U$ and simply consider its age.

**Fact 3.5** Every countable, weakly homogeneous structure $U$ is a Fraïssé limit of its age.

**Example 3.6.** Consider a signature with a single unary function symbol $+1$ and a single binary relation symbol $\leq$. The structure $\langle Q, \leq, +1 \rangle$ is countable and weakly homogeneous. Therefore it is the Fraïssé limit of its age.

**Definition 3.7.** For a Fraïssé class $K$ an atom symmetry $\langle A^K, G^K \rangle$, where $A^K$ is the carrier of $U^K$ and $G^K = \text{Aut}(U^K)$ is its group of automorphisms, is called a Fraïssé symmetry.

**Example 3.8.** All symmetries in Example 2.3 are Fraïssé symmetries. The equality symmetry arises from the class of all finite sets, the total order symmetry – from the class of finite total orders and the timed symmetry – from the class of all finitely generated substructures of $\langle Q, \leq, +1 \rangle$ (see Example 3.6).

For simplicity we frequently identify the elements of age $K$ with finitely generated substructures of $U^K$.

**Least supports.** From now on, we restrict to $G$-sets for groups arising from Fraïssé symmetries. Consider such a symmetry $\langle A^K, G^K \rangle$ and a $G^K$-set $X$.

**Definition 3.9.** A set $C \subseteq A^K$ supports an element $x \in X$ if $x \cdot \pi = x$ for all $\pi \in G^K$ such that $\pi|_C = \text{id}|_C$. A $G^K$-set is nominal in the symmetry $\langle A^K, G^K \rangle$ if every element in the set is supported by the carrier of a finitely generated substructure $A$ of $U^K$. We call $A$ a finitely generated support of $x$. 
Nominal $G_K$-sets and equivariant functions between them form a category $G_K$-Nom which is a full subcategory of $G_K$-Set. When the symmetry $(A_K,G_K)$ under consideration is the equality symmetry, the category $G_K$-Nom coincides with the category Nom defined in [10].

Example 3.10. For any Fraïssé symmetry $(A_K, G_K)$ the sets $A_K$ and $A^n_K$ are nominal. A tuple $(d_1, ..., d_n)$ is supported by the structure generated by its elements.

Lemma 3.11. The following conditions are equivalent:

1. $C$ supports an element $x \in X$;
2. for any $\pi, \sigma \in G_K$ if $\pi|_C = \sigma|_C$ then $x \cdot \pi = x \cdot \sigma$.

Proof. For the implication $(1) \implies (2)$, notice that if $\pi|_C = \sigma|_C$, then $\pi \sigma^{-1}$ acts as identity on $C$, hence $x \cdot \pi \sigma^{-1} = x$ and $x \cdot \pi = x \cdot \sigma$, as required. The opposite implication follows immediately from the definition if we take $\sigma = \text{id}$.

It is easy to see that if an element $x \in X$ has a finitely generated support $A$ then it is also supported by the finite set $C$ of its generators. Thus we can equivalently require $x$ to be finitely supported.

Fact 3.12 A $G_K$-set is nominal if and only if its every element has a finite support.

Example 3.13. Consider the timed symmetry. If the automorphism $\pi$ preserves an atom $a \in Q$, then it necessarily preserves also $a + i$ for any integer $i$. Therefore, if an element $x$ of a nominal set is supported by a substructure generated by $\{1, 30, 100, 105\}$ it is also supported by a substructure generated by $\{1000, 300, 105\}$. Hence in this case for any finitely generated support $A$ of an element $x$ one can find a finitely generated substructure $B$, which is properly contained in $A$ and still supports $x$.

An element of a nominal set has many supports. In particular, supports are closed under adding atoms. If every element of a nominal set $X$ has a unique least finitely generated support, we say that $X$ admits least supports. As shown in Example 3.13 it is not always the case. It turns out that to check if a single-orbit nominal set admits least supports, one just needs to find out if any element of the set has the least finitely generated support.

Lemma 3.14. If $A \subseteq U_K$ is the least finitely generated support of an element $x \in X$, then $A \cdot \pi$ is the least finitely generated support of $x \cdot \pi$ for any $\pi \in G_K$.

Proof. First we prove that $A \cdot \pi$ supports $x \cdot \pi$. Indeed, if an arbitrary $\rho \in G_K$ is an identity on $A \cdot \pi$, then $\pi \rho \pi^{-1}$ is an identity on $A$, hence $x \cdot (\pi \rho \pi^{-1}) = x$. As a result $(x \cdot \pi) \cdot \rho = x \cdot \pi$, as required.

Now let $B \subseteq U_K$ be any finitely generated support of $x \cdot \pi$. We need to show that $A \cdot \pi \subseteq B$. A reasoning similar to the one above shows that $B \cdot \pi^{-1}$ supports $x$, from which we obtain $A \subseteq B \cdot \pi^{-1}$. Therefore, since $\pi$ is a bijection, $A \cdot \pi \subseteq B$. 

Definition 3.15. A Fraïssé symmetry $(A_K, G_K)$ admits least supports if every nominal $G_K$-set admits least supports.

We call a structure $\mathcal{U}$ locally finite if all its finitely generated substructures are finite. Notice that if the universal structure $\mathcal{U}_K$ is locally finite then admitting least supports is equivalent to finitely generated supports being closed under finite intersections. The same holds under the weaker assumption that any finitely generated structure has only finitely many finitely generated substructures.

Example 3.16. If we have only relation symbols in the signature it is obvious that any finitely generated structure is finite. One can prove that in the equality symmetry the intersection of two supports is a support itself. Hence the equality symmetry admits least supports. The same holds for the total order symmetry. Both facts are proved e.g. in [2].

Example 3.17. From Example 3.13 we learned that the timed symmetry does not admit least supports (even though the finitely generated supports are closed under finite intersections). The situation changes when we add to the signature another unary function symbol $-1$ and consider the structure $\langle Q, \leq, +1, -1 \rangle$. It remains weakly homogeneous and therefore is a Fraïssé limit of its age. The automorphism groups of $\langle Q, \leq, +1, -1 \rangle$ and $\langle Q, \leq, +1 \rangle$ are the same, but now for any atom $a \in Q$ we bind together all the elements $a + i$. As a result we obtain a Fraïssé symmetry that admits least supports.

Proposition 3.18. The Fraïssé symmetry obtained from the universal structure $\langle Q, \leq, +1, -1 \rangle$ admits least supports.

Proof. Notice that any finitely generated substructure of $\langle Q, \leq, +1, -1 \rangle$ has only finitely many substructures. Hence it is enough to show that finitely generated supports are closed under finite intersections.

Take any two finitely generated substructures $\mathfrak{A}, \mathfrak{B}$ of $\langle Q, \leq, +1, -1 \rangle$. Let $A$ and $B$ be the sets of elements of $\mathfrak{A}$ and $\mathfrak{B}$ that are contained in the interval $[0, 1)$. These are (finite) sets of generators. Moreover, the structure $\mathfrak{A} \cap \mathfrak{B}$ is generated by $A \cap B$. Hence, it is enough to show that if an automorphism $\pi$ acts as identity on $A \cap B$, then $\pi$ can be decomposed as

$$\pi = \sigma_1 \tau_1 \sigma_2 \tau_2 \ldots \sigma_n \tau_n,$$

where $\sigma_i$ acts as identity on $A$ and $\tau_i$ acts as identity on $B$. Indeed, since each $\sigma_i, \tau_i$ acts as identity on $\mathfrak{A}$ and $\mathfrak{B}$ respectively, we have $x \cdot \sigma_i = x$ and $x \cdot \tau_i = x$. As a result $x \cdot \pi = x$.

Let $l$ be the smallest and $h$ the biggest element of the set $A \cup B$. Notice that $h - l < 1$. Take two different open intervals $(l_A, h_A), (l_B, h_B)$ of length 1 such that $[l, h] \subseteq (l_A, h_A)$ and $[l, h] \subseteq (l_B, h_B)$. Now, consider sets $A' = A \cup \{l_A, h_A\}$, $B' = B \cup \{l_B, h_B\}$. Take an automorphism $\pi$ that acts as identity on $A \cap B = A' \cap B'$. Obviously $\pi$ is a monotone bijection of the set of rational numbers. Therefore, since the total order symmetry admits least supports,

$$\pi = \sigma_1' \tau_1' \sigma_2' \tau_2' \ldots \sigma_n' \tau_n',$$
where $\sigma'_i$, $\tau'_i$ are monotone bijections of $Q$ and $\sigma'_i$ act as identity on $A'$, $\tau'_i$ act as identity on $B'$. For each of the permutations $\sigma'_i$, $\tau'_i$ take an automorphism $\sigma_i$, $\tau_i$ of the universal structure $\langle Q, \leq, +1, -1 \rangle$, such that

$$\sigma'_i|_{(l_A, h_A)} = \sigma_i|_{(l_A, h_A)}, \quad \tau'_i|_{(l_B, h_B)} = \tau_i|_{(l_B, h_B)}.$$ 

Then $\sigma_i$ act as identity on $A$ and $\tau_i$ act as identity on $B$. Moreover $\pi = \sigma_1 \tau_1 \sigma_2 \tau_2 \ldots \sigma_n \tau_n$, as required.

From now on, we assume a Fraïssé symmetry $(\mathcal{A}_K, G_K)$ that admits least supports.

### 4 Structure representation

For any $C \subseteq A$ and $G \leq \text{Sym}(A)$, the restriction of $G$ to $C$ is defined by

$$G|_C = \{ \pi|_C \mid \pi \in G, \ C \cdot \pi = C \} \leq \text{Sym}(C).$$

**Lemma 4.1.** Let $\mathfrak{A} \in \mathcal{K}$ be a finitely generated structure. The set of embeddings $u: \mathfrak{A} \to \mathfrak{U}_K$ with the $G_K$-action defined by composition: $u \cdot \pi = u\pi$, is a single-orbit nominal set.

**Proof.** First notice that any embedding $u: \mathfrak{A} \to \mathfrak{U}_K$ is supported by its image $u(\mathfrak{A})$. Indeed, if an automorphism $\pi \in G_K$ is an identity on $u(\mathfrak{A})$ then obviously $u \cdot \pi = u$. Hence the set of embeddings is a nominal set. Now take any two embeddings $u$ and $v$. The images $u(\mathfrak{A})$, $v(\mathfrak{A})$ are finitely generated isomorphic substructures of $\mathfrak{U}_K$. By extending any isomorphism between $u(\mathfrak{A})$ and $v(\mathfrak{A})$, we obtain an automorphism $\pi \in G_K$ such that $u \cdot \pi = v$.

As we shall show now, every single-orbit nominal set is isomorphic to one of the above form, quotiented by some equivariant equivalence relation.

Notice that the quotient of a $G$-set by an equivariant equivalence relation $R$ has a natural structure of a $G$-set, with the action defined as follows:

$$[x]_R \cdot \pi = [x \cdot \pi]_R.$$ 

It is easy to see that if $X$ has one orbit, then so does the quotient $X/R$. Moreover, any support $C$ of an element $x \in X$ supports the equivalence class $[x]_R$, hence if $X$ is nominal then $X/R$ is also nominal.

**Definition 4.2.** A structure representation is a finitely generated structure $\mathfrak{A} \in \mathcal{K}$ together with a group of automorphisms $S \leq \text{Aut}(\mathfrak{A})$ (the local symmetry). Its semantics $[\mathfrak{A}, S]$ is the set of embeddings of $u: \mathfrak{A} \to \mathfrak{U}_K$, quotiented by the equivalence relation:

$$u \equiv_S v \iff \exists \tau \in S \tau u = v.$$ 

A $G_K$-action on $[\mathfrak{A}, S]$ is defined by composition:

$$[u]_S \cdot \pi = [u \pi]_S.$$
Proposition 4.3. (1) \([\mathfrak{A}, S]\) is a single-orbit nominal \(G_K\)-set. (2) Every single-orbit nominal \(G_K\)-set \(X\) is isomorphic to some \([\mathfrak{A}, S]\).

Proof. For (1), use Lemma 4.1. The set of embeddings \(u: \mathfrak{A} \to \mathfrak{U}_K\) is a single-orbit nominal \(G_K\)-set, and so is the quotient \([\mathfrak{A}, S]\).

For (2), take a single-orbit nominal set \(X\) and let \(H \leq G_K\) be the stabilizer of some element \(x \in X\). Put \(S = H|_{\mathfrak{A}}\) where \(\mathfrak{A} \in K\) is the least finitely generated support of \(x\). Define \(f: X \to [\mathfrak{A}, S]\) by \(f(x \cdot \pi) = [\pi|_{\mathfrak{A}}]|_S\). The function \(f\) is well defined: if \(x \cdot \pi = x \cdot \sigma\) then \(\pi \cdot \sigma^{-1} \in H\). As \(A \cdot \pi \cdot \sigma^{-1}\) is the least finitely generated support of \(x \cdot \pi \cdot \sigma^{-1} = x\), we obtain \(A \cdot \pi \cdot \sigma^{-1} = \mathfrak{A}\). Therefore for \(\tau = (\pi \cdot \sigma^{-1})|_{\mathfrak{A}} \in S\) we have \(\tau \cdot \pi|_{\mathfrak{A}} = \pi|_{\mathfrak{A}}\), hence \([\pi|_{\mathfrak{A}}]|_S = [\sigma|_{\mathfrak{A}}]|_S\). It is easy to check that \(f\) is equivariant.

It remains to show that \(f\) is bijective. For injectivity, assume \(f(x \cdot \pi) = f(x \cdot \sigma)\). This means that there exists \(\tau \in S\) such that \(\tau \cdot \pi|_{\mathfrak{A}} = \pi|_{\mathfrak{A}}\), then \((\pi \cdot \sigma^{-1})|_{\mathfrak{A}} \in S\), hence \((\pi \cdot \sigma^{-1})|_{\mathfrak{A}} = \rho|_{\mathfrak{A}}\) for some \(\rho \in H\). Therefore \(x \cdot \pi \cdot \sigma^{-1} = x \cdot \rho = x\), from which we obtain \(x \cdot \pi = x \cdot \sigma\). For surjectivity of \(f\), note that by universality of the structure \(\mathfrak{U}_K\) any embedding \(u: \mathfrak{A} \to \mathfrak{U}_K\) can be extended to an automorphism \(\pi\) of \(\mathfrak{U}_K\), for which we have \(f(x \cdot \pi) = [u]|_S\).

Structure representation was defined by Bojańczyk et al. in the special case of \(\mathfrak{U}_K\) being a relational structure. The proposition above generalizes Proposition 11.7 of [2].

Example 4.4. Consider the Fraïssé symmetry obtained from the universal structure \((\mathbb{Q}, \preceq, +1, -1)\) and a structure \(\mathfrak{A}\) generated by \(\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}\). Notice that mapping one of the generators, say \(\frac{1}{4}\), to any element of \(\mathfrak{A}\), say \(\frac{1}{2} \mapsto 3\frac{3}{4}\), uniquely determines an automorphism \(\pi\) of \(\mathfrak{A}\). The automorphism can be seen as a shift. It maps \(\frac{1}{4}\) to \(3\frac{3}{4}\) and \(\frac{3}{4}\) to \(4\frac{1}{4}\). This observation leads to the conclusion that \(\text{Aut}(\mathfrak{A}) = \mathbb{Z}\). Any subgroup \(S\) of \(\text{Aut}(\mathfrak{A})\) is therefore isomorphic to \(\mathbb{Z}\) and generated by a single automorphism \(\pi\) of the form described above. The same holds for any finitely generated substructure \(\mathfrak{A}\). In this particular case Proposition 4.3 provides a very nice finite representation of single-orbit nominal sets.

Fungibility. Even if the symmetry admits least supports it may happen that some finitely generated structure is not the least finitely generated support of anything. Now we will introduce a condition which ensures that any finitely generated structure is the least finitely generated support of some element of some nominal set.

Definition 4.5. A finitely generated substructure \(\mathfrak{A}\) of \(\mathfrak{U}_K\) is fungible if for every finitely generated substructure \(\mathfrak{B} \subseteq \mathfrak{A}\), there exists \(\pi \in G_K\) such that:

\[\pi|_{\mathfrak{B}} = \text{id}|_{\mathfrak{B}},\]
\[\pi(\mathfrak{A}) \neq \mathfrak{A}.\]

A Fraïssé symmetry \((\mathfrak{A}_K, G_K)\) is fungible if every finitely generated substructure \(\mathfrak{A}\) of \(\mathfrak{U}_K\) is fungible.
**Example 4.6.** The equality and total order symmetries are both fungible. The timed symmetry is not fungible. Take a structure $\mathfrak{A}$ generated by $\{0\}$ and its substructure $\mathfrak{B}$ generated by $\{1\}$. Obviously if an automorphism $\pi$ acts as identity on $\mathfrak{B}$ then it acts as identity also on $\mathfrak{A}$. It is easy to see that by adding to the timed symmetry the function symbol $-1$ (see Example 3.17) we obtain a fungible symmetry.

In general, admitting least supports and being fungible are independent properties. Examples are given in [2].

**Lemma 4.7.** (1) If $(\mathfrak{A}, G_\mathfrak{A})$ admits least supports then every finitely generated fungible $\mathfrak{A} \subseteq \mathfrak{U}_\mathfrak{A}$ is the least finitely generated support of $[\text{id}|_\mathfrak{A}]_S$, for any $S \leq \text{Aut}(\mathfrak{A})$.

(2) If $(\mathfrak{A}, G_\mathfrak{A})$ is fungible then every finitely generated $\mathfrak{A} \subseteq \mathfrak{U}_\mathfrak{A}$ is the least finitely generated support of $[\text{id}|_\mathfrak{A}]_S$, for any $S \leq \text{Aut}(\mathfrak{A})$.

**Proof.** For (1), recall from Lemma 4.1 that an embedding $u : \mathfrak{A} \rightarrow \mathfrak{U}_\mathfrak{A}$ is supported by its image. Therefore $\mathfrak{A}$ supports $\text{id}|_\mathfrak{A}$ and hence also $[\text{id}|_\mathfrak{A}]_S$. Now consider any finitely generated structure $\mathfrak{B}$ properly contained in $\mathfrak{A}$. Since $\mathfrak{A}$ is fungible there exists an automorphism $\pi$ from the Definition 4.5. The automorphism $\pi$ acts as identity on $\mathfrak{B}$, but $[\text{id}|_\mathfrak{A}]_S \cdot \pi \neq [\text{id}|_\mathfrak{A}]_S$, which means that also $\pi(\mathfrak{B}) \neq \mathfrak{B}$. Hence $[\text{id}|_\mathfrak{A}]_S \cdot \pi = [\pi|_\mathfrak{A}]_S \neq [\text{id}|_\mathfrak{A}]_S$ and we obtain a contradiction as it turns out that $\mathfrak{B}$ does not support $[\text{id}|_{\mathfrak{B}}]_S$.

Let us focus for a moment on relational structures. In this case to obtain a fungible symmetry it is enough to require an existence of $\pi$ that is not an identity on $\mathfrak{A}$.

**Definition 4.8.** A finitely generated substructure $\mathfrak{A}$ of $\mathfrak{U}_\mathfrak{A}$ is weakly fungible if for every finitely generated substructure $\mathfrak{B} \subsetneq \mathfrak{A}$, there exists $\pi \in G_\mathfrak{A}$ such that:

- $\pi|_{\mathfrak{B}} = \text{id}|_{\mathfrak{B}}$,
- $\pi|_{\mathfrak{A}} \neq \text{id}|_{\mathfrak{A}}$.

A Fraïssé symmetry $(\mathfrak{A}, G_\mathfrak{A})$ is weakly fungible if every finitely generated substructure $\mathfrak{A}$ of $\mathfrak{U}_\mathfrak{A}$ is weakly fungible.

On the other hand, if we restrict ourselves to relational structures, we can also equivalently require an existence of automorphisms $\pi$ that satisfy a stronger condition.

**Definition 4.9.** A finitely generated substructure $\mathfrak{A}$ of $\mathfrak{U}_\mathfrak{A}$ is strongly fungible if for every finitely generated substructure $\mathfrak{B} \subsetneq \mathfrak{A}$, there exists $\pi \in G_\mathfrak{A}$ such that:
A Fraïssé symmetry \((A_K, G_K)\) is strongly fungible if every finitely generated substructure \(\mathfrak{A}\) of \(U_K\) is weakly fungible.

**Fact 4.10** Let \((A_K, G_K)\) be a Fraïssé symmetry over a signature containing only relation symbols. The following conditions are equivalent:

1. \((A_K, G_K)\) is weakly fungible,
2. \((A_K, G_K)\) is fungible,
3. \((A_K, G_K)\) is strongly fungible.

The general picture is more complicated. When we introduce function symbols, the notions of weak fungibility, fungibility and strong fungibility differ from each other. Before showing this let us notice that the condition of strong fungibility is in fact equivalent to the strong amalgamation property.

**Proposition 4.11.** A Fraïssé symmetry \((A_K, G_K)\) is strongly fungible if and only if the age \(K\) of the universal structure \(U_K\) has the strong amalgamation property.

**Proof.** The if part is easily proved using homogeneity. For the only if part take any finitely generated substructures \(\mathfrak{A}, \mathfrak{B}, \mathfrak{C}\) of \(U_K\) and embeddings \(f_\mathfrak{B}: \mathfrak{A} \rightarrow \mathfrak{B}, f_\mathfrak{C}: \mathfrak{A} \rightarrow \mathfrak{C}\). Thanks to amalgamation there exists a finitely generated substructure \(D\) of \(U_K\) together with two embeddings \(g_\mathfrak{B}: \mathfrak{B} \rightarrow D, g_\mathfrak{C}: \mathfrak{C} \rightarrow D\) such that \(g_\mathfrak{B} \circ f_\mathfrak{B}(\mathfrak{A}) = g_\mathfrak{C} \circ f_\mathfrak{C}(\mathfrak{A}) = \mathfrak{A}'\). Take \(\pi \in G_K\) for which \(\pi|_{\mathfrak{A}'} = \text{id}|_{\mathfrak{A}'}\) and \(\pi(D) \cap D = \mathfrak{A}'\). Let \(D'\) be a substructure generated by \(\mathfrak{D} \cup \pi(\mathfrak{D})\). The embeddings \(g_\mathfrak{B}\) and \(g_\mathfrak{C} = \pi \circ g_\mathfrak{C}\) into \(D'\) are as needed:

\[g_\mathfrak{B} \circ f_\mathfrak{B}(\mathfrak{A}) = g_\mathfrak{C} \circ f_\mathfrak{C}(\mathfrak{A}) = g_\mathfrak{B}(\mathfrak{B}) \cap g_\mathfrak{C}(\mathfrak{C}).\]

**Corollary 4.12.** A Fraïssé symmetry \((A_K, G_K)\) over a signature containing only relation symbols is fungible if and only if the age \(K\) of the universal structure \(U_K\) has the strong amalgamation property.

**Example 4.13.** Consider an algebraic signature with unary function symbols \(F\) and \(G\). For any integer \(i\) let \(A_i\) be the set of all infinite, binary sequences \(\langle a_n \rangle_{n \geq i}\) such that \(a_n = 0\) for all but finitely many \(n\). Take \(A = \bigcup A_i\) and define a structure \(X\) with a carrier \(A\), where

\[F(\langle a_i, a_{i+1}, a_{i+2}, \ldots \rangle) = \langle a_{i+1}, a_{i+2}, \ldots \rangle, \quad G(0w) = 1w, \quad G(1w) = 0w.\]

Since the structure is weakly homogeneous, we obtain a Fraïssé symmetry. The symmetry is weakly fungible, but it is not fungible, as the structure generated by \(\{0w, 1w\}\) is not fungible for any \(w \in A\).

**Example 4.14.** Consider a signature with a single unary function symbol \(F\). For any integer \(i\) let \(A_i\) be the set of all infinite sequences \(\langle a_n \rangle_{n \geq i}\) of natural
numbers with \( a_n = 0 \) for all but finitely many \( n \). Take \( \mathbb{A} = \bigcup \mathbb{A}_i \) and define a structure \( \mathfrak{U} \) with a carrier \( \mathbb{A} \), where

\[
F((a_i, a_{i+1}, a_{i+2}, \ldots)) = \langle a_{i+1}, a_{i+2}, \ldots \rangle.
\]

Notice that the age of \( \mathfrak{U} \) is the class \( \mathcal{K} \) of all finitely generated structures that satisfy the following axioms

- for any \( a, b \) there exist \( m, n \in \mathbb{N} \) such that \( F^m(a) = F^n(b) \),
- there are no loops, i.e., \( F^n(a) \neq a \) for all \( n \in \mathbb{N} \).

Since the structure is weakly homogeneous, we obtain a Fraïssé symmetry \( (\mathbb{A}_\mathcal{K}, \mathcal{G}_\mathcal{K}) \). It is easy to check that the symmetry is fungible.

From now on, we assume a Fraïssé symmetry \( (\mathbb{A}_\mathcal{K}, \mathcal{G}_\mathcal{K}) \) that admits least supports and is fungible.

**Representation of functions.** For any finitely generated substructure \( \mathfrak{A} \) of \( \mathfrak{U}_\mathcal{K} \) and any \( S \leq \text{Aut}(\mathfrak{A}) \), the \( \mathcal{G}_\mathcal{K} \)-extension of \( S \) is

\[
\text{ext}_{\mathcal{G}_\mathcal{K}}(S) = \{ \pi \in \mathcal{G}_\mathcal{K} \mid \pi|_\mathfrak{A} \in S \} \leq \mathcal{G}_\mathcal{K}.
\]

Notice that \( \text{ext}_{\mathcal{G}_\mathcal{K}}(S) \) is exactly the stabilizer of \( [\text{id}_\mathfrak{A}]_S \) in \( \mathcal{G}_\mathcal{K} \).

**Lemma 4.15.** For each embedding \( u: \mathfrak{A} \to \mathfrak{U}_\mathcal{K} \) the group \( \text{ext}_{\mathcal{G}_\mathcal{K}}(u^{-1}Su) \), where \( u^{-1}Su \leq \text{Aut}(u(\mathfrak{A})) \), is the stabilizer of an element \( [u]_S \in [\mathfrak{A}, \mathfrak{S}] \).

**Proof.** For any \( \pi \in \mathcal{G}_\mathcal{K} \) that extends \( u \) we have \( [u]_S = [\text{id}_\mathfrak{A}]_S \cdot \pi \). Hence, by Lemma 2.6, the stabilizer of \( [u]_S \) is \( \pi^{-1}\text{ext}_{\mathcal{G}_\mathcal{K}}(S)\pi \). It is easy to check that

\[
\pi^{-1}\text{ext}_{\mathcal{G}_\mathcal{K}}(S)\pi = \text{ext}_{\mathcal{G}_\mathcal{K}}(u^{-1}Su).
\]

**Lemma 4.16.** Let \( \mathfrak{A}, \mathfrak{B} \) be finitely generated substructures of \( \mathfrak{U}_\mathcal{K} \) and let \( S \leq \text{Aut}(\mathfrak{A}) \), \( T \leq \text{Aut}(\mathfrak{B}) \) then \( \text{ext}_{\mathcal{G}_\mathcal{K}}(S) \leq \text{ext}_{\mathcal{G}_\mathcal{K}}(T) \) if and only if \( \mathfrak{B} \subseteq \mathfrak{A} \) and \( S|_\mathfrak{B} \leq T \).

**Proof.** The if part is obvious. For the only if part, we first prove that \( \mathfrak{B} \subseteq \mathfrak{A} \). Notice that if \( \pi|_\mathfrak{A} = \text{id}_\mathfrak{A} \) then \( \pi \in \text{ext}_{\mathcal{G}_\mathcal{K}}(S) \) and hence \( \pi \in \text{ext}_{\mathcal{G}_\mathcal{K}}(T) \), which is the stabilizer of \( [\text{id}_\mathfrak{B}]_T \). Therefore \( \mathfrak{A} \) supports \( [\text{id}_\mathfrak{B}]_T \). By Lemma 4.7 (2) the least support of \( [\text{id}_\mathfrak{B}]_T \) is \( \mathfrak{B} \). Hence \( \mathfrak{B} \subseteq \mathfrak{A} \). Then we have

\[
\text{ext}_{\mathcal{G}_\mathcal{K}}(S) \leq \text{ext}_{\mathcal{G}_\mathcal{K}}(T)
\]

\[
\forall \pi \in \mathcal{G}_\mathcal{K} \pi|_\mathfrak{A} \in S \implies \pi|_\mathfrak{B} \in T.
\]
∀π ∈ \( G \) \( \pi|_A \in S \implies (\pi|_A)|_B \in T \)

∀τ ∈ S \( \tau|_B \in T \).

Similar facts about finite substructures of a universal relational structure \( U_K \) were proven in [2]. The following proposition generalizes the Proposition 11.8.

**Proposition 4.17.** Let \( X = [A, S] \) and \( Y = [B, T] \) be single-orbit nominal sets. The set of equivariant functions from \( X \) to \( Y \) is in one to one correspondence with the set of embeddings \( u: B \to A \), for which \( uS \subseteq Tu \), quotiented by \( \equiv_T \).

**Proof.** By Proposition 2.7 and Lemma 4.15 equivariant functions from \( [A, S] \) to \( [B, T] \) are in bijective correspondence with those elements \( [u]_T \in [B, T] \) for which

\[
\text{ext}_{G_K}(S) \leq \text{ext}_{G_K}(u^{-1}Tu).
\]

Hence, by Lemma 4.16, equivariant functions from \( [A, S] \) to \( [B, T] \) correspond to those elements \( [u]_T \in [B, T] \) for which

\[
u(B) \subseteq A \text{ and } S|_{u(B)} \leq u^{-1}Tu,
\]

which means that \( u \) is an embedding from \( B \) to \( A \) and \( uS \subseteq Tu \), as required.

Let \( G_K\text{-Nom} \) denote the category of single-orbit nominal sets and equivariant functions. Propositions 4.3 and 4.17 can be phrased in the language of category theory:

**Theorem 4.18.** In a Fraïssé symmetry that admits least supports and is fungible, the category \( G_K\text{-Nom} \) is equivalent to the category with:

- as objects, pairs \( (A, S) \) where \( A \in K \) and \( S \leq \text{Aut}(A) \),
- as morphisms from \( (A, S) \) to \( (B, T) \), those embeddings \( u: B \to A \) for which \( uS \subseteq Tu \), quotiented by \( \equiv_T \).

Since a nominal set is a disjoint union of single-orbit sets, this representation extends to orbit-finite sets in an obvious way.

In the special case of relational structures the above theorem was formulated and proved in [2].

### 5 Conclusions and future work

Orbit-finite nominal sets can be used to model devices, such as automata or Turing machines, which operate over infinite alphabets. This approach makes sense only if one can treat objects with atoms as data structures and manipulate them using algorithms. To do so the existence of a finite representation of orbit-finite nominal sets is crucial.
In this paper we have generalized the representation theorem due to Bojańczyk et al. to cover atoms with algebraic structure. The result is however not entirely satisfying. Our representation uses automorphism groups of finitely generated substructures of the atoms. If such groups are finitely presentable Theorem 4.18 indeed provides a concrete, finite representation of orbit-finite nominal sets. But is it always the case? So far we do not know and we regard it as a field for a further research effort.

Another thing left to be done is a characterization of “well-behaved” atom symmetries in terms of Fraïssé classes that induce them. One might think of algebraic atoms that could be potentially interesting from the point of view of computation theory: strings with the concatenation operator, binary vectors with addition, etc. Yet checking the technical conditions, such as the existence of the least finitely generated supports or fungibility, needed for the representation theorem to hold requires each time a lot of effort. This is because these conditions are formulated in terms of Fraïssé limits, and these are not always easy to construct. It would be desirable to have more natural criteria that would be easier to verify.

References

An Investigation of Preorders and Preorder ⊤⊤-Liftings on the Subdistribution Monad

Tetsuya Sato
Research Institute for Mathematical Sciences, Kyoto University, Kyoto, 606-8502, Japan
satoutet@kurims.kyoto-u.ac.jp

Abstract. The subdistribution monad is well-known in the coalgebraic treatment of discrete probabilistic systems. We analyse preorders on the subdistribution monad using a technique called preorder ⊤⊤-lifting, which calculates relational liftings and preorders on monads at the same time. We show that there are $5^2 = 25$ preorders at unit type on the subdistribution monad. We partition all preorders on this monad using preorders at the unit type. Moreover, we show that there are exactly 9 partial orders on the subdistribution monad. We give relational liftings generated from these preorders, and we show that various probabilistic simulations are covered with simulations given by the preorder ⊤⊤-lifting and preorders at the unit type on the subdistribution monad.

1 Introduction

1.1 Background

The monad structures play an important role in coalgebraic modelling of branching of transition systems. Intuitively, the unit of the monad embeds a state in a trivial branching with only one possibility of choice, and the multiplication of the monad flattens successive branchings into one branching. For example, the powerset monad $\mathcal{P}$, the finite multiset monad $\mathcal{M}$, the lifting monad $\mathcal{L}$, and the subdistribution monad $\mathcal{D}$ captures nondeterministic branching, nondeterministic branching with duplication, deterministic branching with an error, and discrete probabilistic branching respectively.

Preorders on monads are important for the coalgebraic study of state-based systems. A suitable partial order on a monad gives a coalgebraic trace semantics [3] and forward/backward simulations between coalgebras [2]. In the study [4] by Hesselink and Thijs and the study [6] by Jacobs and Hughes, a systematic procedure to obtain coalgebraic simulations between coalgebras of preordered functors is developed. Many of these preorders on functors involve preorders on monads.

Preorders on a monad are closely related to relational liftings of monads. Hence, study of preorders on monads helps study of coalgebraic simulations. Levy identified all preorders on $\mathcal{P}$, $\mathcal{L}$ and $\mathcal{PL}$, and he discussed modal logics in the presence of simulations for nondeterministic labelled transition systems with
errors using boolean precongruences [9]. Katsumata and the author developed a technique called preorder \( \Top \)-lifting [7] to generate relational liftings of monads and preorders on monads at the same time. They identified all preorders on \( \mathcal{P} \), \( \mathcal{M} \), \( \mathcal{L} \), and \( \mathcal{PL} \) using the preorder \( \Top \)-lifting, independently from Levy’s results in [9].

1.2 Our Contribution

In this work, we study preorders on \( \mathcal{D} \) using the preorder \( \Top \)-lifting. We show that there are exactly 5\(^2 = 25\) congruent and substitutive preorders on \( \mathcal{D} \), we partition the class of preorders on \( \mathcal{D} \) into 25 partitions using them. We calculate the least preorder and the greatest preorder in each partition. We show that these the least and the greatest preorders are generated from 4 preorders on \( \mathcal{D} \). In addition, we show that there are exactly 9 partial orders on \( \mathcal{D} \). Since the preorder \( \Top \)-lifting gives the preorders on \( \mathcal{D} \) and relational liftings of \( \mathcal{D} \) at the same time, we also study relational liftings that are given by the preorder \( \Top \)-lifting. We obtain coalgebraic simulations between probabilistic systems by preorder \( \Top \)-liftings of \( \mathcal{D} \), which include the probabilistic bisimulations given by Larsen and Skou [8].

1.3 Preliminaries

Throughout this paper, we consider the category \( \text{Set} \) of sets and functions. We denote by \( \text{Pre} \) the category of preordered sets and monotone functions between them. We define the category \( \text{BRel} \) to be the category of binary relations and morphisms between them. Here, a morphism from a binary relation \( R \subseteq X_1 \times Y_1 \) to \( S \subseteq X_2 \times Y_2 \) is a pair of functions \( f: X_1 \to Y_1 \) and \( g: X_2 \to Y_2 \) such that \((f \times g)(R) \subseteq S\).

For each set \( X \), we denote by \( \text{Eq}_X \) the equality relation on \( X \), namely \( \{ (x, x) \mid x \in X \} \). For each pair of a binary relation \( R \subseteq X \times Y \) and a set \( U \subseteq X \), we denote by \( R[U] \) the image \( \{ y \in Y \mid \exists x \in U. (x, y) \in R \} \) of \( U \) under \( R \). In particular, we denote by \( R[x] \) the image \( R[\{ x \}] \) of the singleton set \( \{ x \} \). For each pair of real numbers \( p \) and \( q \) with \( p < q \), real intervals between \( p \) and \( q \) are denoted by \( [p, q] \), \( (p, q) \), \( [p, q) \), or \( (p, q] \).

For a \( \text{Set} \) monad \( (T, \eta, \mu) \) and a function \( f: J \to TI \), the Kleisli Lifting \( f^!: TJ \to TI \) of \( f \) is the composition \( f^! = \mu \circ T(f) \).

2 Preorders on Monads and Preorder \( \Top \)-Liftings

We introduce some results of [7], which we later apply to the subdistribution monad.

Katsumata and the author developed a technique called preorder \( \Top \)-lifting to study preorders on monads categorically. We developed some concrete methods for enumerating and identifying preorders on monads and discuss coalgebraic simulations given by the preorder \( \Top \)-lifting in [7].

First, we define relational liftings and coalgebraic simulations.
Definition 1. Let \( F \) be an endofunctor on \( \text{Set} \). A relational lifting of \( F \) is a functor \( \Gamma: \text{BRel} \to \text{BRel} \) such that \( \Gamma(f, g) = (F(f), F(g)) \) for each morphism \((f, g): R \to S\). We say that a relational lifting \( \Gamma \) of \( F \) is

- reflexive when \( \text{Eq}_{FX} \subseteq \Gamma(\text{Eq}_X) \),
- lax compositional when \( \Gamma(R) \circ \Gamma(S) \subseteq \Gamma(R \circ S) \) for each \( R \subseteq X \times Z \) and \( S \subseteq Z \times Y \),
- monotone if \( \Gamma(R) \subseteq \Gamma(S) \) holds whenever \( R \subseteq S \), and
- conversive when \( \Gamma(R^\circ) = \Gamma(R)^\circ \).

Definition 2. Consider a relational lifting \( \Gamma \) of \( F \), and \( F \)-coalgebras \( \xi: X \to FX \) and \( \xi': Y \to FY \). A \( \Gamma \)-simulation from \( \xi \) to \( \xi' \) is a relation \( R \subseteq X \times Y \) such that \((\xi, \xi')\) is a morphism from \( R \) to \( \Gamma(R) \) in \( \text{BRel} \).

We remark that \( \Gamma \)-simulations are closed under the union of arbitrary size. The \( \Gamma \)-similarity is defined as the union of all \( \Gamma \)-simulations: two states \( x \in X \) and \( y \in Y \) are \( \Gamma \)-similar if \((x, y) \in R \) for some \( \Gamma \)-simulation \( R \). If \( \Gamma \) is lax compositional then \( \Gamma \)-simulations are closed under the composition of relations. If \( \Gamma \) is reflexive, lax compositional, and conversive then \( \Gamma \)-similarity from \( \xi \) to \( \xi' \) is an equivalence relation.

We next introduce the notion of preorder \( \sqsubseteq \)-liftings and some related concepts of monads for further discussions. We now fix a monad \((T, \eta, \mu)\) on \( \text{Set} \).

Definition 3. A functor \( \Gamma: \text{BRel} \to \text{BRel} \) is the relational lifting of monad \( T \) if \((\eta_X, \eta_Y)\) is a morphism from \( R \subseteq X_1 \times X_2 \) to \( \Gamma(R) \) and \((f^2, g^2)\) is a morphism from \( \Gamma(R) \) to \( \Gamma(S) \) whenever the pair of \( f: X_1 \to TY_1 \) and \( g: X_2 \to TY_2 \) is a morphism from \( R \subseteq X_1 \times X_2 \) to \( \Gamma(S) \) where \( S \subseteq Y_1 \times Y_2 \).

We remark that a relational lifting of monad \( T \) is indeed a relational lifting of \( T \).

Definition 4. Let \( I \) be a set, and consider a preorder \( \preceq \) on \( TI \).

1. We call \( \preceq \) congruent if for each set \( J \) and functions \( f, g: J \to TI \),
   \[(\forall j \in J. f(j) \preceq g(j)) \implies (\forall x \in T J. f^2(x) \preceq g^2(x)).\]
2. We call \( \preceq \) substitutive if for each \( f: I \to TI \),
   \[f^2 \text{ is a monotone function on } (TI, \preceq).\]

Definition 5 ([7, Definition 3]). A preorder \( \sqsubseteq \) on a monad \( T \) is an assignment of a preorder \( \sqsubseteq_I \) on \( TI \) to each set \( I \) such that

- \( \forall I. \sqsubseteq_I \) is congruent, and
- for each \( f: I \to TI \), \( f^2 \) is a monotone function from \((TI, \sqsubseteq_I)\) to \((TI, \sqsubseteq_I)\)
  (we also call this property substitutive).

For each preorder \( \sqsubseteq \) on \( T \), we call \( \sqsubseteq_I \) evaluation of \( \sqsubseteq \) at \( I \).

Suppose that the Kleisli category \( \text{Set}_T \) of \( T \) is \( \text{Pre} \)-enriched, and the enrichment is pointwise, that is, the ordering of arrows \( \sqsubseteq_{I,J} \subseteq \text{Set}_T(I, J)^2 \) satisfies
\[f \sqsubseteq_{I,J} g \iff \forall j: 1 \to J. f^2(\eta_J \circ j) \sqsubseteq_{1,I} g^2(\eta_J \circ j).\]
Then the assignment $\subseteq': J \mapsto \subseteq_{1,J}$ is a preorder on $T$ because the left monotonicity and the right monotonicity of the composition in $\text{Set}_T$ gives the congruence and the substitutivity respectively to $\subseteq'$. On the other hand, suppose a preorder $\subseteq'$ on $T$. The assignment $(I,J) \mapsto \{(f,g) \mid \forall i. f(i) \subseteq g(i)\}$ gives a pointwise $\text{Pre}$-enrichment of $\text{Set}_T$. This correspondence is bijective.

Below, we write $(\text{Pre}(T), \subseteq)$ for the class of preorders on $T$, ordered by the partial order $\preceq$ defined by $\subseteq \preceq \subseteq' \iff \forall I. \subseteq_I \subseteq \subseteq'_I$. We write $(\text{CSPre}(T, I), \subseteq)$ for the set of congruent and substitutive preorders on $TI$, ordered by inclusion of relations.

**Definition 6 ([7, Definition 8])**. For each $\preceq \in \text{CSPre}(T, I)$, we define the preorder $\top \top$-lifting $T^{\top \top}(\preceq_I): R \subseteq X \times Y \mapsto T^{\top}(\subseteq_I) R \subseteq TX \times TY$ as follows:

$$
T^{\top}(\subseteq_I) R = \{(f, g) \in (X \Rightarrow TI) \times (Y \Rightarrow TI) \mid \forall (x', y') \in R. f(x') \preceq g(y')\}
$$

$$
T^{\top \top}(\preceq_I) R = \{(x, y) \in TX \times TY \mid \forall (f, g) \in T^{\top}(\subseteq_I) R. f^2(x) \preceq g^2(y)\}.
$$

We simply write $T^{\top}(\subseteq)$ and $T^{\top \top}(\subseteq)$ instead of $T^{\top}(\subseteq_I)$ and $T^{\top \top}(\subseteq_I)$ respectively when the set $I$ is obvious from context.

The preorder $\top \top$-lifting $T^{\top \top}(\preceq_I)$ is a reflexive and monotone relational lifting of the monad $T$. We have the following condition for the lax compositionality of $T^{\top \top}(\preceq_I)$.

**Proposition 1 ([7, Theorem 11])**. For given $\preceq \in \text{CSPre}(T, I)$ such that

$$
\forall X, Y \subseteq TI, (\forall x \in X, y \in Y. x \preceq y) \implies (\exists z \in TI. \forall x \in X, y \in Y. x \preceq z \preceq y),
$$

the lifting $T^{\top \top}(\preceq)$ is lax compositional.

Preorder $\top \top$-liftings give preorders on monads at the same time.

**Proposition 2 ([7, Theorem 2])**. The assignment $J \mapsto T^{\top \top}(\preceq_I) \text{Eq}_J$ is a preorder on $T$ (which we denote by $\preceq_I^J$).

The mapping $(-)_I: \subseteq \mapsto \subseteq_I$ from $(\text{Pre}(T), \subseteq)$ to $(\text{CSPre}(T, I), \subseteq)$ is monotone. The monotone mapping $\preceq \mapsto [\preceq]^I$ is characterised as the right adjoint of $(-)_I$.

**Proposition 3 ([7, Theorem 3])**. For each set $I$, $(-)_I \dashv [\_]^I$ and $[\_]^I_I = \text{id}$.

This proposition gives that the mapping $[\_]^I$ preserves intersections, and $[\preceq]^I$ is the greatest preorder on $T$ whose evaluation at $I$ equals $\preceq$ for each $\preceq \in \text{CSPre}(T, I)$. Therefore, $[\preceq^{op}]^I = ([\preceq]^I)^{op}$ for each $\preceq \in \text{CSPre}(T, I)$. In other words, the mapping $[\_]^I$ preserves opposites.

The mapping $(-)_I$ also has the left adjoint $(-)^I$. For given $\preceq$, we define $(\preceq)^I$ by the intersection of all preorder on $T$ whose evaluation at $I$ equal $\preceq$:

$$
(\preceq)^I = \bigcap \{\subseteq \in \text{Pre}(T) \mid \subseteq_I = \preceq\}.
$$

Since $[\preceq]^I = \preceq$ and $(\text{Pre}(T), \subseteq)$ is closed under intersections, the preorder $(\preceq)^I$ is definable. Moreover, it is the least one whose evaluation at $I$ equals $\preceq$ for each $\preceq \in \text{CSPre}(T, I)$.
Thus $\langle \succeq \rangle^T I = \preceq$. The minimality of $\langle \succeq \rangle^T I$ gives $\langle \preceq^\oplus \rangle^T I = (\langle \succeq \rangle^T I)^\oplus$ for each $\preceq \in \text{CSPre}(T, I)$. Hence, the mapping $\langle - \rangle^T I$ preserves opposites.

It is easy to check the monotonicity of $\langle - \rangle^T I$ and the adjunction $\langle - \rangle^T I \dashv (\langle - \rangle^T I)$.

Since $\langle \preceq \rangle^T I = \text{id}$ holds, $\langle \preceq \rangle^T I$ is an injection and $(\langle - \rangle^T I)$ is a surjection. Therefore, we can partition $\text{Pre}(T)$ using inverse images of the surjection $(\langle - \rangle^T I)$ as

$$\text{Pre}(T) = \bigcup \{ (\langle - \rangle^T I)^{-1}(\preceq) \mid \preceq \in \text{CSPre}(T, I) \}.$$ We call each inverse image $(\langle - \rangle^T I)^{-1}(\preceq)$ of $\preceq$ a partition. The mappings $\langle - \rangle^T I$ and $[\langle - \rangle^T I]$ give the least preorder and the greatest preorder in each partition.

3 Preorders on the Subdistribution Monad

After introducing the subdistribution monad $\mathcal{D}$, we carry out the following steps to analyse $\text{Pre}(\mathcal{D})$.

1. We calculate $\text{CSPre}(\mathcal{D}, 1)$ to partition $\text{Pre}(\mathcal{D})$.
2. We partition $\text{Pre}(\mathcal{D})$ using $\text{CSPre}(\mathcal{D}, 1)$.
3. We analyse each partition $(\langle - \rangle^T I)^{-1}(\preceq)$ of $\text{Pre}(\mathcal{D})$ where $\preceq \in \text{CSPre}(\mathcal{D}, 1)$.

3.1 Definition of the Subdistribution Monad

First, we introduce the subdistribution monad $\mathcal{D}$, which is often used to characterise discrete probabilistic systems coalgebraically.

**Definition 7.** The subdistribution functor $\mathcal{D}$ on $\text{Set}$ is defined as follows:

- For each set $X$, $\mathcal{D}X = \{d : X \rightarrow [0, 1] \mid \sum_{x \in X} d(x) \leq 1\}$, and
- For each function $f : X \rightarrow Y$, $\mathcal{D}(f) : \mathcal{D}X \rightarrow \mathcal{D}Y$ is defined by $\mathcal{D}(f)(d)(y) = \sum_{x \in f^{-1}(y)} d(x)$ for each $d \in \mathcal{D}X$ and $y \in Y$.

We call each $d \in \mathcal{D}X$ a (subprobability) distribution on $X$. We give some notations for further discussions.

- For each $x \in X$, the Dirac distribution $\delta_x \in \mathcal{D}X$ is given by $\delta_x(x) = 1$ and $\delta_x(y) = 0 \ (y \neq x)$.
- For each set $X$, zero distribution $0 \in \mathcal{D}X$ is given by $0(x) = 0$ for any $x \in X$.
- For each $d \in \mathcal{D}X$, we define the support of $d$ as $\text{supp}(d) = \{x \in X \mid d(x) > 0\}$.
- For each $d \in \mathcal{D}X$, we define the summation $d[U]$ of $d$ over $U \subseteq X$ by $\sum_{x \in U} d(x)$.
- For each $d_1, d_2 \in \mathcal{D}X$, we define the minimum $\min(d_1, d_2) \in \mathcal{D}X$ of $d_1$ and $d_2$ by $\min(d_1, d_2)(x) = \min(d_1(x), d_2(x))$ for each $x \in X$. 

For each $d_1, d_2 \in \mathcal{D}X$, we define $d_1 \cdot d_2 \in \mathcal{D}X$ by $(d_1 \cdot d_2)(x) = d_1(x) \cdot d_2(x)$.

For each $f, g: X \rightarrow \mathcal{D}Y$, we define $f \cdot g: X \rightarrow \mathcal{D}Y$ by $(f \cdot g)(x) = f(x) \cdot g(x)$.

For each $U \subseteq X$, we define the characteristic function $\chi_U: X \rightarrow [0, 1]$ by $\chi_U(x) = 1$ ($x \in U$), and $\chi_U(x) = 0$ ($x \notin U$).

We remark that the finiteness of supports are not required, but for each $d \in \mathcal{D}X$, $\text{supp}(d)$ is at most countable since $\mathcal{D}[X] \leq 1$.

Many probabilistic systems are characterised coalgebraically by $\mathcal{D}$:

- A Markov chain $\xi_1: X \rightarrow \mathcal{D}X$ is a $\mathcal{D}$-coalgebra,
- A probabilistic LTS $\xi_2: X \rightarrow \mathcal{D}(A \times X)$ is a $\mathcal{D}(A \times \text{Id})$-coalgebra, and
- A Segala automaton $\xi_3: X \rightarrow \mathcal{P}\mathcal{D}(1+A \times X)$ is a $\mathcal{P}\mathcal{D}(1+A \times \text{Id})$-coalgebra.

For further examples, see [10].

We consider a $\mathcal{D}$-coalgebra $\xi: X \rightarrow \mathcal{D}X$ which has a state $x \in X$ such that $\xi(x)[X] < 1$. We then regard that $\xi$ has a deadlock in the state $x$. Similarly, when a coalgebra functor contains $\mathcal{D}$, we discuss deadlocks of the probabilistic branching.

**Definition 8.** We define the unit $\eta: \text{Id} \Rightarrow \mathcal{D}$ and the multiplication $\mu: \mathcal{D}^2 \Rightarrow \mathcal{D}$ of the subdistribution monad $\mathcal{D}$ as follows:

- for each $x \in X$, $\eta_X(x) = \delta_x$ and,
- for each $\xi \in \mathcal{D}^2 X$, $\mu_X(\xi)(x) = \sum_{d \in \mathcal{D}X} \xi(d) \cdot d(x)$ where $x \in X$.

The triple $(\mathcal{D}, \eta, \mu)$ is a monad on $\text{Set}$.

### 3.2 Congruent and Substitutive Preorders on $\mathcal{D}1$

Next, we identify congruent substitutive preorders on $\mathcal{D}1$ to partition and analyse $\text{Pre}(\mathcal{D})$. We regard each $d \in \mathcal{D}1$ as a real number $d \in [0, 1]$ since $\mathcal{D}1 \cong [0, 1]$. We thus draw a congruent substitutive preorder on $\mathcal{D}1$ as a region in the square $[0, 1] \times [0, 1]$. For example, the shaded triangle of right picture stands for the ordering $\preceq^r$ given by $p \preceq^r q \iff p \leq q$.

The following lemma is crucial to identify congruent and substitutive preorders on $\mathcal{D}1$.

**Lemma 1.** Let $\preceq \in \text{CSPre}(\mathcal{D}, 1)$.

1. $p \preceq q$ for some $0 < p < q < 1$ if and only if $p \preceq q$ for each $0 \leq p < q \leq 1$.
2. $0 \preceq q$ for some $0 < p < 1$ if and only if $p \preceq q$ for each $0 \leq p < q < 1$.
3. $p \preceq 1$ for some $0 < p < 1$ if and only if $p \preceq q$ for each $0 < p < q \leq 1$.
4. $0 \preceq 1$ if and only if $p \preceq q$ for each $0 \leq p < q \leq 1$. 
Proof (Sketch). It suffices to prove the “only if”-part of 1. The statements 2.–4. are proved from 1. together with the congruence and transitivity of $\succeq$.

[Step (1)] First, the reflexivity says that the diagonal line is included in $\succeq$. Since $0 < p < q < 1$, the point $(p, q) \in \succeq$ is drawn in $[0, 1] \times [0, 1]$ as the dot in (1). Starting from this point, we extend the area that is shown to be inside of $\succeq$ as the following steps. [Step (2)] We have $p \succeq q$, $0 \succeq 0$, and $1 \succeq 1$. From the congruence of $\succeq$, $\alpha p + \beta \succeq \alpha q + \beta$ for each $0 \leq \alpha$ and $0 \leq \beta$ such that $\alpha + \beta \leq 1$. Hence, triangle that is filled with stripes is included in $\succeq$. [Step (3)] From Step (2), we have $p^2/q \succeq p$ and $q = \beta p + (1 - \beta) \succeq \beta q + (1 - \beta) = (2q - q^2 - p)/(1 - p)$ where $\beta = (1 - q)/(1 - p)$. From the transitivity, $p^2/q \succeq q$ and $p \succeq (2q - q^2 - p)/(1 - p)$. The two points $(p^2/q, q)$ and $(p, (2q - q^2 - p)/(1 - p))$ are outside of the filled triangle of Step (2) since $(2q - q^2 - p)/(1 - p) > q$ and $p^2/q < p$.

[Step (4) and (5)] Similarly, $\succeq$ includes the polygons that are filled with stripes. [Step (6)] Iterating this extension, we obtain $\{(p, q) \mid 0 < p < q < 1\} \subseteq \succeq$.

We then have an identification of congruent substitutive preorders on $D$.1.

**Theorem 1.** Each $\preceq \in \text{CSPre}(D, 1)$ is an intersection of the following preorders:

$$
\preceq^r = \quad \preceq^s = \quad \preceq^d = \\
\preceq^{r \circ p} = \quad \preceq^{s \circ p} = \quad \preceq^{d \circ p} =
$$

where
\[-p \preceq r \text{ if and only if } (p \preceq q)\]
\[-p \preceq s \text{ if and only if } (p > 0 \implies q > 0)\]
\[-p \preceq d \text{ if and only if } (p = 1 \implies q = 1)\]

The superscripts \(r, s, d\) stand for real values, supports, and deadlocks of distributions respectively. In this paper, we will show that \(\preceq^r\) corresponds to the standard order \(\succeq^r\) between distributions as real functions, and \(\preceq^s\) corresponds to the support-inclusion order \(\subseteq^s\) of distributions, and \(\preceq^d\) gives the ordering \(\subseteq^d\) which relates deadlocks of distributions.

Since \(\preceq^r \subseteq \preceq^d\) and \(\preceq^r \preceq \preceq^s\), the intersection closure of \(\{\preceq^r, \preceq^s, \preceq^d\}\) consists of 5 preorders (including \(\mathcal{D}1 \times \mathcal{D}1\)). Similarly, \(\{\preceq^{r1}, \preceq^{s1}, \preceq^{d1}\}\) consists of 5 preorders. Thus, we obtain the following corollary.

**Corollary 1.** \(\text{CSPre}(\mathcal{D}, 1) \cong 5 \times 5 = 25\).

Therefore, we partition \(\text{Pre}(\mathcal{D})\) into 25 classes using \(\text{CSPre}(\mathcal{D}, 1)\).

### 3.3 An Analysis of \(\text{Pre}(\mathcal{D})\) using \(\text{CSPre}(\mathcal{D}, 1)\)

We analyse each partition \((-)^{-1}(\preceq)\) of \(\text{Pre}(\mathcal{D})\), where \(\preceq \in \text{CSPre}(\mathcal{D}, 1)\).

The following lemma is crucial to compute \((-)^{-1} : \text{CSPre}(\mathcal{D}, 1) \to \text{Pre}(\mathcal{D})\).

**Lemma 2.** Let \(\preceq \in \text{CSPre}(\mathcal{D}, 1)\). If \(d_1, d_2 \in \mathcal{D}X\) satisfy the condition \((\dagger)\):

\[
\forall x \in \text{supp}(d_1), \quad \left(\frac{1 + d_1[X]}{2} \preceq \frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)}\right)
\]

then we have \(d_1 \preceq \frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)}\), where the subdistribution \(\min(d_1, d_2) \in \mathcal{D}X\) is given by \(\min(d_1, d_2)(x) = \min(d_1(x), d_2(x))\).

**Proof.** We may assume \(d_1 \neq 0\) because if \(d_1 = 0\) then \(d_1 = 0 = \min(d_1, d_2)\). We here recall that \((\preceq)^{-1} = \preceq\). From the substitutivity of \((\preceq)^{-1}\), for each \(x \in \text{supp}(d_1)\),

\[
\frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)}
\]

We define the functions \(f, g : X \to \mathcal{D}X\) by for each \(x \in \text{supp}(d_1)\),

\[
f(x) = \frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)} \quad \text{and} \quad g(x) = \frac{2}{1 + d_1[X]} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)}
\]

and \(f(x) = g(x) = 0\) for each \(x \in X \setminus \text{supp}(d_1)\). We obtain \(f(x) \preceq \frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)}\) for each \(x \in X\). From the congruence of \((\preceq)^{-1}\), we obtain the following:

\[
d_1 = f^2 \left(\frac{2}{1 + d_1[X]} \cdot d_1\right) \preceq \frac{1 + d_1[X]}{2} \cdot \frac{\min(d_1, d_2)(x)}{d_1(x)} \preceq g^2 \left(\frac{2}{1 + d_1[X]} \cdot d_1\right)
\]

We remark that \(2d_1/(1 + d_1[X]) \in \mathcal{D}X\) because \(2d_1[X]/(1 + d_1[X]) \leq 1\). \(\square\)
By this lemma, we are able to discuss separately the congruence and the transitivity of \( \preceq \) for a given \( \preceq \in \text{CSPre}(D, 1) \).

The following theorem helps identifying preorders on \( D \).

**Lemma 3.** If \( \preceq \in \text{CSPre}(D, 1) \) satisfies the following condition \((\dagger)\):

\[
\langle \preceq \rangle = \langle \preceq \rangle^d \cap \langle \preceq \rangle^\text{exp} \subseteq \preceq \;
\Rightarrow \;
\langle \preceq \rangle = \langle \preceq \rangle^d \cap \langle \preceq \rangle^\text{exp} \subseteq \preceq,
\]

then \( [\preceq]^1 = \langle \preceq \rangle^1 \), hence, \( (-1)^{1}(\preceq) = \{[\preceq]^1\} \).

**Proof.** It suffices to prove \( [\preceq]^1 \subseteq \langle \preceq \rangle^1 \) since \( \langle \preceq \rangle^1 \subseteq [\preceq]^1 \) holds by the maximality of \( [\preceq]^1 \).

Suppose \( d_1 \langle \preceq \rangle^1 X d_2 \). We prove \( d_1 \langle \preceq \rangle^1 X \min(d_1, d_2) \langle \preceq \rangle^1 X d_2 \). It suffices to show that \( d_1 \) and \( d_2 \) satisfy the condition \((\dagger)\) of Lemma 2 to prove \( d_1 \langle \preceq \rangle^1 X \min(d_1, d_2) \). The following cases and their subcases are exhaustive:

1. **(Case: \( \min(d_1, d_2) = d_1 \))** Trivial.

2. **(Case: \( \min(d_1, d_2) \neq d_1 \) and \( \min(d_1, d_2) = d_2 \))** In this case, we have \( d_2 \leq d_1 \) and \( d_1 \neq d_2 \). Hence, \( \text{supp}(d_2) \subseteq \text{supp}(d_1) \) and \( d_2[X] < 1 \).

   - **(Case: \( d_1[X] < 1 \) and \( \text{supp}(d_1) = \text{supp}(d_2) \))** Define \( O(p, q) \iff 0 < q \leq p < 1 \). We first show that \( O(p, q) \) implies \( p \leq q \).

     From \( d_2 \leq d_1 \) and \( d_1 \neq d_2 \), there exists \( x \in \text{supp}(d_1) \) such that \( 0 < d_2(x) < d_1(x) < 1 \); the first and last inequality follows from \( \text{supp}(d_1) = \text{supp}(d_2) \) and \( d_1[1] < 1 \) respectively. From the substitutivity of \( [\preceq]^1 \), we obtain \( d_1(x) \leq d_2(x) \). By Lemma 1, \( p \leq q \) whenever \( O(p, q) \).

     Next, we obtain \( O \left( \frac{(1+d_1[X])}{2} \frac{\text{min}(d_1, d_2)(x)}{d_1(x)} , \frac{(1+d_1[X])}{2} \right) \) for each \( x \in \text{supp}(d_1) \) from \( d_1[X] < 1 \), \( d_2 \leq d_1 \), and \( \text{supp}(d_1) = \text{supp}(d_2) \).

   - **(Case: \( d_1[X] < 1 \) and \( \text{supp}(d_2) \subseteq \text{supp}(d_1) \))** Analogous to the first case, with \( O(p, q) \iff 0 \leq q \leq p < 1 \).

     - **(Case: \( d_1[X] = 1 \) and \( \text{supp}(d_1) = \text{supp}(d_2) \))** Analogous to the first case, with \( O(p, q) \iff 0 < q \leq p \leq 1 \).

     - **(Case: \( d_1[X] = 1 \) and \( \text{supp}(d_2) \subseteq \text{supp}(d_1) \))** Analogous to the first case, with \( O(p, q) \iff 0 \leq q < p \leq 1 \).

3. **(Case: \( \min(d_1, d_2) \neq d_1, d_2 \))** We first show \( 0 < p, q < 1 \) implies \( p \leq q \).

   From \( \min(d_1, d_2) \neq d_1, d_2 \), there are \( x, y \in X \) such that \( d_1(x) < d_2(x) \) and \( d_1(x) > d_2(x) \). By applying the substitutivity of \( [\preceq]^1 \), we obtain \( d_1(x) \leq d_2(x) \) and \( d_1(x) \geq d_2(x) \).

   By Lemma 1, \( 0 < p, q < 1 \) implies \( p \leq q \). From this together with the assumption \((\dagger)\) of this lemma, we obtain \( p \leq q \) for each \( 0 < p, q \leq 1 \).

   - **(Case: \( \text{supp}(d_1) \subseteq \text{supp}(d_2) \))** As \( 0 \neq d_1 \), \( 0 < \frac{\text{min}(d_1, d_2)(x)}{d_1(x)} \) holds for each \( x \in \text{supp}(d_1) \). Thus, \( \frac{(1+d_1[X])}{2} \leq \frac{(1+d_1[X])}{2} \frac{\text{min}(d_1, d_2)(x)}{d_1(x)} \) for each \( x \in \text{supp}(d_1) \).

   - **(Case: \( \text{supp}(d_1) \not\subseteq \text{supp}(d_2) \))** There exists \( x \in \text{supp}(d_1) \) such that \( d_2(x) = 0 \). From the substitutivity of \( [\preceq]^1 \) together with the transitivity of \( \preceq \), we also obtain \( p \leq 0 \) for all \( 0 \leq p \leq 1 \). Hence, \( \frac{(1+d_1[X])}{2} \leq \frac{(1+d_1[X])}{2} \frac{\text{min}(d_1, d_2)(x)}{d_1(x)} \) for each \( x \in \text{supp}(d_1) \).
Similarly, we have \( \min(d_1, d_2) \langle \leq \rangle_{X} d_2 \). Therefore, we obtain \( \langle \leq \rangle_{1} = \langle \leq \rangle_{1} \).

The evaluation at \( 1 \) of a partial order on \( D \) is a congruent substitutional partial order on \( D_1 \). For each partial order \( \leq \) in \( \text{CSPre}(D, 1) \), \( \leq s \cap \leq \text{op} \cap \leq d \cap \leq \text{dp} \subseteq \leq \) does not hold. Thus, all congruent substitutional partial order on \( D_1 \) satisfy the condition of Lemma 3. Therefore, we identify all partial orders on \( D \). There are exactly 9 partial orders on \( D \) (see the cells marked with * of Table 1).

Let \( \mathcal{E} = [\langle \leq \rangle_{1}] \), \( \mathcal{E}^{*} = [\langle \leq \rangle_{1}] \), and \( \mathcal{E}^{d} = [\langle \leq \rangle_{1}] \). We obtain,

\[
\begin{align*}
d_1 \leq X d_2 & \iff \forall x \in X.d_1(x) \leq d_2(x) \\
d_1 \leq X^{*} d_2 & \iff \text{supp}(d_1) \subseteq \text{supp}(d_2) \\
d_1 \leq X^d d_2 & \iff (d_1[X] = 1 \implies d_2[\text{supp}(d_1)] = 1)
\end{align*}
\]

We postpone the proof of the above equivalences to the next section.

It is known that \( \mathcal{E}^{*} \) is the standard partial order on \( D \). Moreover the partial order \( \mathcal{E}^{*} \) is the unique partial order on \( D \) that satisfies the condition of the [3, Theorem 3.3.] Because, the partial orders \( \mathcal{E}^{*} \cap \mathcal{E}^{\text{op}} \), and \( \mathcal{E}^{*} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{d} \) are not \( \omega \)-complete, and the partial orders = (equality), \( \mathcal{E}^{*} \cap \mathcal{E}^{\text{op}}, \mathcal{E}^{*} \cap \mathcal{E}^{d}, \mathcal{E}^{*} \cap \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}}, \mathcal{E}^{*} \cap \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{\text{dp}} \), and \( \mathcal{E}^{*} \cap \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{\text{dp}} \) do not have the assignment \( \bot \) of the least elements.

The preorder \( \mathcal{E}^{*} \) is the support-inclusion order of distributions. The preorder \( \mathcal{E}^{d} \) gives some kind of conditions of the possibility of choices: if \( d_1 \leq X^d d_2 \) and \( d_1[X] = 1 \) then \( d_1[X] = 1 \) and \( \text{supp}(d_2) \subseteq \text{supp}(d_1) \).

<table>
<thead>
<tr>
<th>( D_1 \times D_1 )</th>
<th>( \langle \leq \rangle^* )</th>
<th>( \langle \leq \rangle^d )</th>
<th>( \langle \leq \rangle^s \cap \langle \leq \rangle^d )</th>
<th>( \langle \leq \rangle^r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle \leq \rangle^{\text{op}} )</td>
<td>( \mathcal{E}^{\text{op}} )</td>
<td>( \mathcal{E}^{\text{op}} \cap \mathcal{E}^{\text{op}} )</td>
<td>( \mathcal{E}^{\text{op}} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{d} )</td>
<td>( \mathcal{E}^{\text{op}} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} )</td>
</tr>
<tr>
<td>( \langle \leq \rangle^{d} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{d} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} )</td>
</tr>
<tr>
<td>( \langle \leq \rangle^{s} \cap \langle \leq \rangle^{d} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} \cap \mathcal{E}^{d} \cap \mathcal{E}^{\text{op}} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} )</td>
<td>( \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} \cap \mathcal{E}^{d} )</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Partitions of \( \text{Pre}(D) \) given by \( \text{CSPre}(D, 1) \)

We show identified preorders on \( D \) using Table 1. Each cell of \( \langle \leq \rangle^{1}, \langle \leq \rangle^{2} \) stands for the partition \( (-)_{1}^{-1}(- \cap \langle \leq \rangle^{2}) \) of \( \text{Pre}(D) \). Each cell filled with a preorder \( \mathcal{E} \) on \( D \) means that the corresponding partition has the unique preorder \( \mathcal{E} \), and each cell filled with "..." means that \( \langle \leq \rangle^{1} \neq \langle \leq \rangle^{1} \) holds for the corresponding \( \leq \in \text{CSPre}(D, 1) \). For example, the cell of \( \langle \leq \rangle^{r}, \langle \leq \rangle^{s} \) is filled with \( \mathcal{E}^{r} \cap \mathcal{E}^{s} \). This means that the partition \( (-)_{1}^{-1}(- \cap \langle \leq \rangle^{s}) \) of \( \text{Pre}(D) \) has the unique preorder \( \mathcal{E}^{r} \cap \mathcal{E}^{s} \).

By Lemma 3, \( \mathcal{E}^{r} = \langle \leq \rangle^{1} \) and \( \mathcal{E}^{s} = \langle \leq \rangle^{1} \). However, \( \mathcal{E}^{d} \neq \langle \leq \rangle^{1} \). To show \( \mathcal{E}^{d} \neq \langle \leq \rangle^{1} \), we give the following explicit form of \( \langle \leq \rangle^{d} \).

**Definition 9.** We define the preorder \( \mathcal{E}^{d} \) on \( D \) by

\[
d_1 \mathcal{E}^{d} d_2 \iff (d_1[X] = 1 \implies d_1 = d_2)
\]
The superscript $m$ means the minimality of $\sqsubseteq^m$.

**Proposition 4.** The equality $\sqsubseteq^m = (\preceq d)^1$ holds.

**Proof.** It suffices to prove $\sqsubseteq^m \preceq (\preceq d)^1$. Suppose $d_1 \sqsubseteq^m_d d_2$. We may assume $d_1[X] < 1$ since $d_1 = d_2$ whenever $d_1[X] = 1$. We then obtain $\min(d_1, d_2)[X] < 1$. From from $d_1[X] < 1$, we obtain $\frac{1 + d_1[X]}{2} \preceq d_1^{[x]} \frac{\min(d_1, d_2) + d_1^{[x]}}{2}$ for each $x \in \supp(d_1)$. From $\min(d_1, d_2)[X] < 1$, we obtain $\frac{1 + d_2[X]}{2} \preceq d_2^{[x]} \frac{\min(d_1, d_2) + d_2^{[x]}}{2}$ for each $x \in \supp(d_2)$. By Lemma 2, we obtain $d_1 \preceq (\preceq d)^1_X^{\min(d_1, d_2)} d_2$.

Generally speaking, the left adjoint $(-)^T$ of the evaluation mapping $(-)_I$ does not preserve intersections. However, $(-)^1$ does preserve intersections.

**Theorem 2.** The mapping $(-)^1 : \text{CSPre}(D, 1) \to \text{Pre}(D)$ preserves intersections.

We prove this theorem in the same way as Proposition 4. Lemma 2 has an important role in the proof this theorem.

**Proof (Sketch).** From Lemma 3, Proposition 4, and $(\preceq^\text{op})^1 = (\preceq^\text{op})^1$, it suffices to consider the following intersections: $\preceq^d \cap \preceq^s$, $\preceq^d \cap \preceq^\text{op}$, $\preceq^d \cap \preceq^s \cap \preceq^\text{op}$, $\preceq^d \cap \preceq^\text{op} \cap \preceq^s$, and $\preceq^d \cap \preceq^\text{op} \cap \preceq^s \cap \preceq^\text{op}$.

We prove only $(\preceq^d \cap \preceq^\text{op})^1 = \sqsubseteq^m \cap \sqsubseteq^\text{op}$, and we omit the proofs for other cases. However, they are analogous to the proof for this case.

It suffices to prove $\sqsubseteq^m \cap \sqsubseteq^\text{op} \leq (\preceq^d \cap \preceq^\text{op})^1$. Suppose $d_1 \sqsubseteq^m \sqsubseteq^\text{op} d_2$, that is, $d_1[X] = 1 \implies d_2[X] = 1$ and $d_2[X] = 1 \implies d_1[X] = 1$. We have $d_1[X] = 1 \iff d_2[X] = 1$. We assume $d_1[X], d_2[X] < 1$ since $d_1[X] = d_2[X] = 1$ implies $d_1 = d_2$. From $d_1[X] < 1$, we obtain $0 \leq \frac{1 + d_1[X]}{2} \frac{\min(d_1, d_2) + d_1^{[x]}}{2} \leq \frac{1 + d_2[X]}{2} \frac{\min(d_1, d_2) + d_2^{[x]}}{2}$ for each $x \in \supp(d_1)$. Thus, $\frac{1 + d_1[X]}{2} \preceq d_1^{[x]} \frac{\min(d_1, d_2) + d_1^{[x]}}{2}$ for each $x \in \supp(d_1)$. By Lemma 2, we obtain $d_1 \preceq (\preceq^d \cap \preceq^\text{op})^1_X^{\min(d_1, d_2)} d_2$.

Similarly, we also obtain $\min(d_1, d_2) \preceq (\preceq^d \cap \preceq^\text{op})^1_X^{\min(d_1, d_2)} d_2$. Thus, $d_1 \preceq (\preceq^d \cap \preceq^\text{op})^1_X^{\min(d_1, d_2)} d_2$.

Therefore, $(\preceq^d \cap \preceq^\text{op})^1 = \sqsubseteq^m \cap \sqsubseteq^\text{op} = (\preceq^d)^1 \cap (\preceq^\text{op})^1$.

Since the mappings $[-]^1, (-)^1 : \text{CSPre}(D, 1) \to \text{Pre}(D)$ preserve intersections, the image of $[-]^1$ and $(-)^1$ are generated with $\{\sqsubseteq^r, \sqsubseteq^s, \sqsubseteq^d\}$ and $\{\sqsubseteq^r, \sqsubseteq^s, \sqsubseteq^m\}$ (by taking opposite preorders and intersections) respectively.

## 4 Preorder T-T-Liftings Generated from Preorders at the Unit Type

In the previous section, we have discussed the preorders of the form $[\preceq]_I$ on $D$ where $\preceq \in \text{CSPre}(D, 1)$, and we have seen that they help to analyse $\text{Pre}(D)$ well. We here recall that the preorder $[-]^T$ on a monad $T$ is given by $J \mapsto$. 
\(T^\top T(\preceq,1)\)\(\mathcal{D}\) for each congruent substitutive preorder \(\preceq\) on \(T I\). Thus, we will show that well-known notions of simulations between probabilistic systems are simulations by preorder \(T^\top\)-liftings of \(\mathcal{D}\) generated from \(\text{CSPre}(\mathcal{D}, 1)\). Also, we will give the postponed proof of characterisation of \(\preceq_r, \preceq_\ast, \preceq^d\) in the previous section.

We give a form of \(\mathcal{D}^T(\preceq)\) without \(\mathcal{D}^T(\preceq)\) for each \(\preceq \in \text{CSPre}(\mathcal{D}, 1)\).

**Theorem 3.** For each \(R \subseteq X \times Y\), the following hold:

\[
\mathcal{D}^T(\preceq)R = \{(d_1, d_2) \mid \forall U \subseteq X, d_1[U] \leq d_2[R[U]]\} \quad (1)
\]
\[
\mathcal{D}^T(\preceq^\ast)R = \{(d_1, d_2) \mid \forall U \subseteq X, d_1[U] > 0 \implies d_2[R[U]] > 0\} \quad (2)
\]
\[
\mathcal{D}^T(\preceq^d)R = \{(d_1, d_2) \mid \forall U \subseteq X, d_1[X] = 1 \implies d_2[R[U]] = 1\} \quad (3)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq^d)R = \mathcal{D}^T(\preceq^\ast)R \cap \mathcal{D}^T(\preceq^d)R \quad (4)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast)R = \mathcal{D}^T(\preceq^\ast)R \cap \mathcal{D}^T(\preceq_\ast)R \quad (5)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d)R = \mathcal{D}^T(\preceq^\ast)R \cap \mathcal{D}^T(\preceq_\ast)R \cap \mathcal{D}^T(\preceq^d)R \quad (6)
\]

Here, for each binary relation \(R\), the difunctional closure \(\overline{R}\) of \(R\) is defined by \(\bigcup_{n=0}^\infty ((R \circ \text{op})^n \circ R)\). Hence, simulations given by \(\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d)(\preceq \in \{\preceq^\ast, \preceq_\ast, \preceq^d\})\) are seen as simulations up to difunctional closure (see also [1]).

The following hold (the \(\cap\)s in right-hand sides are setwise intersections):

\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq_\ast) \cap \mathcal{D}^T(\preceq^d) \quad (7)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq^d) \quad (8)
\]
\[
\mathcal{D}^T(\preceq_\ast \cap \preceq^d) = \mathcal{D}^T(\preceq_\ast) \cap \mathcal{D}^T(\preceq^d) \quad (9)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq_\ast) \quad (10)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq_\ast) \cap \mathcal{D}^T(\preceq^d) \quad (11)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq^d) \quad (12)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast) \cap \mathcal{D}^T(\preceq_\ast) \cap \mathcal{D}^T(\preceq^d) \quad (13)
\]
\[
\mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d) = \mathcal{D}^T(\preceq^\ast \cap \preceq_\ast \cap \preceq^d) \quad (14)
\]

We remark that \(\preceq^\ast \cap \preceq^d\) is the equality relation on \(\mathcal{D}\).

**Proof.** We prove only equality (1), and we omit proofs of other equalities.

We fix \(R \subseteq X \times Y\). (\(\subseteq\)) Suppose \((d_1, d_2) \in \mathcal{D}^T(\preceq)\). For each \(U \subseteq X\), \((\chi_U \cdot \chi_R(U)) \in \mathcal{D}^T(\preceq)R\) since \((\chi_U(x), \chi_R(U)(y))\) is \((1, 1)\), \((0, 1)\) or \((0, 0)\) whenever \((x, y) \in R\). Since \(\chi_U^2(d_1) = d_1[U]\) and \(\chi_R[U]^2(d_2) = d_2[R[U]]\), \(d_1[U] \leq d_2[R[U]]\).

(2) Suppose \(d_1[U] \leq d_2[R[U]]\) for each \(U \subseteq X\) and \((f, g) \in \mathcal{D}^T(\preceq^\ast,1)R\). We prove \(f^2(d_1) \leq g^2(d_2)\). For each finite subset \(U \subseteq \text{supp}(d_1)\), we write \(f^2_U\) for
\[ f \cdot \chi_U. \] Since a summation of nonnegative real numbers is the least upper bound of all of its finite partial sums,

\[
\sup \left\{ f_U \cdot 1(d_1) \mid U \subseteq \text{finite sup}(d_1) \right\} = f^2(d_1).
\]

Thus it is enough to prove \( f_U \cdot 1(d_1) \leq g^2(d_2) \) for each finite subset \( U \) of \( \text{supp}(d_1) \).

We fix a finite subset \( U = \{x_1, x_2, \ldots, x_N\} \) of \( \text{supp}(d_1) \). We have \( \text{supp}(f_U) \subseteq U \). Let \( p_k = f(x_k) \) for \( 1 \leq k \leq N \). We assume \( p_k \leq p_{k+1} \) for \( 1 \leq k \leq N - 1 \), without loss of generality. Let \( p_0 = 0 \) for simplicity. We then decompose \( f_U \) as,

\[
f_U = \sum_{k=1}^{N} (p_k - p_{k-1}) \chi_{\{x_k, x_{k+1}, \ldots, x_N\}}.
\]

For each \( y \in Y \), we define \( m(y) = \max\{ \{ k \mid (x_k, y) \in R, x_k \in U \} \cup \{0\} \} \). We obtain that \( p_m(y) = \max \{ f(x_k) \mid x_k \in U \} \) when \( y \in R[U] \), and \( p_m(y) = 0 \) when \( y \notin R[U] \). Since \( p_m(y) = 0 \) for each \( y \in Y \setminus R[U] \), we have \( p_m(y) \leq g(y) \) for each \( y \in Y \). We give \( g' : Y \to D^1 \) by \( g'(y) = g(y) - p_m(y) \), and therefore \( g \) is decomposed as,

\[
g = g' + \sum_{k=1}^{N} (p_k - p_{k-1}) \chi_{\{x_k, x_{k+1}, \ldots, x_N\}}.
\]

We remark that \( \sum_{k=1}^{N} (p_k - p_{k-1}) \chi_{\{x_k, x_{k+1}, \ldots, x_N\}}(y) = p_m(y) \) for each \( y \in Y \).

Since \( d_1[\{x_k, x_{k+1}, \ldots, x_N\}] \leq d_2[R[\{x_k, x_{k+1}, \ldots, x_N\}]] \) holds for each \( 1 \leq k \leq N \), \( f_U \cdot 1(d_1) \leq g^2(d_2) \) holds. Since \( U \) is arbitrary, \( (d_1, d_2) \in D^{\top \top(\leq,\leq)}R \).

By applying equality relations, we obtain the equivalences:

\[
\begin{align*}
d_1 \subseteq_X d_2 & \iff (d_1, d_2) \in D^{\top \top(\leq)}R \chi_X \iff \forall x \in X, d_1(x) \leq d_2(x) \\
d_1 \subseteq_X d_2 & \iff (d_1, d_2) \in D^{\top \top(\leq)} \chi_X \iff \text{supp}(d_1) \subseteq \text{supp}(d_2) \\
d_1 \subseteq_X^\top d_2 & \iff (d_1, d_2) \in D^{\top \top(\leq)} \chi_X \iff (d_1[X] = 1 \implies d_2[\text{supp}(d_1)] = 1)
\end{align*}
\]

Since \( D^{\top \top(\leq)}R = (D^{\top \top(S)}R \chi^\top)^\chi^\top \) and \( D^{\top \top(D^1 \times D^1)}R = D^X \times D^Y \), this lemma gives a form of \( D^{\top \top(\leq)} \) without \( D^{\top \top(=)} \) for a given \( \leq \in \text{CSPre}(D, 1) \).

From Proposition 1, they are lax compositional. However, they need not to be strictly compositional. For example, the lifting \( D^{\top \top(\leq)} \) is not strictly compositional. Suppose \( X = \{x, w\} \), \( Y = \{y\} \), and \( Z = \{z, v\} \), and we give \( R \subseteq X \times Y \) and \( S \subseteq X \times Z \) by \( R = \{(x, y), (w, y)\} \) and \( S = \{(y, z), (y, v)\} \). We have \((\frac{1}{4}\delta_x + \frac{1}{4}\delta_y + \frac{1}{4}\delta_z) \in D^{\top \top(\leq)}(R \circ S) \), but there is no \( d \in D^Y \) such that \((\frac{1}{4}\delta_x + \frac{1}{4}\delta_y + \frac{1}{4}\delta_z, d) \in D^{\top \top(\leq)}(R) \) and \((d, \frac{1}{4}\delta_z + \frac{1}{4}\delta_z) \in D^{\top \top(\leq)}(S) \). If there was such a \( d \in D^Y \) then we have the contradiction: \( 1 \leq d(y) \) and \( d(y) \leq \frac{1}{2} \).

From the equalities (4) – (14) of Theorem 3, \( D^{\top \top(\leq)} \) does not preserve intersections. However, this does not contradict that \( [-] \) preserves intersections because each equality relation \( \text{Eq}_X \) is symmetric and already difunctional closed.

We focus on the first 3 liftings in Theorem 3. We show that well-known simulations between probabilistic systems are simulations by preorder \( \top \top \) liftings of \( D \) generated from \( \text{CSPre}(D, 1) \).
A $D^{\top\top}(\preceq)$-simulation between $D$-coalgebras is exactly a probabilistic simulations [8]. Similarly, a $D^{\top\top}(\preceq) \cap D^{\top\top}(\preceq^{\op})$-simulation is exactly a probabilistic bisimulation [8].

One may alternatively define (bi) simulation between $D$-coalgebras $(X, \xi)$ and $(X', \xi')$ to be a (bi) simulation relation $R$ between two $\mathcal{P}$-colagebras $(X, \text{supp} \circ \xi)$ and $(X', \text{supp} \circ \xi')$ in the standard sense. This (bi) simulation for $D$-coalgebras coincides with (resp. $D^{\top\top}(\preceq) \cap D^{\top\top}(\preceq^{\op})$-simulations.

Consider a $D^{\top\top}(\preceq)$-simulation $R$ between two $D$-coalgebras. Intuitively, $(x, y) \in R$ and no deadlock happens at $x$ then so does at $y$, and every next state of $y$ are related to some next state of $x$ by $R$.

When $F$ is a polynomial functor, we obtain $D^{\top\top}(\preceq)\hat{F}$-simulations between $DF$-coalgebras, where $\hat{F}$ is the canonical relational lifting of $F$. For example, probabilistic simulations between probabilistic LTSs are exactly $D^{\top\top}(\preceq)\hat{F}$-simulations where $F = A \times \text{Id}$ ($A$ is a set of labels).

5 Conclusion and Future Work

We have analysed $\text{Pre}(D)$ partially using the $\text{CSPre}(D, 1)$, and we have discussed preorder $\top\top$-liftings of $D$ for preorders in $\text{CSPre}(D, 1)$. We have obtained the following results:

- We have identified $\text{CSPre}(D, 1)$. It consists of $5^2 = 25$ preorders.
- We have partitioned $\text{Pre}(D)$ into 25 collections using $\text{CSPre}(D, 1)$, and we have identified the least preorder $(\preceq)^1_1$ and the greatest preorder $[\preceq]^1_1$ of each partition $(-)^1_1(\preceq)$ of $\text{Pre}(D)$. Moreover, they are generated from the following 4 preorders: $\subseteq^r$, $\subseteq^s$, $\subseteq^d$, and $\subseteq^m$. This gives that there are at least 37 preorders on $D$.
- We have enumerated all partial orders on $D$. There are exactly 9 partial orders on $D$: $\subseteq^r$, $\subseteq^r \cap \subseteq^{\op}$, $\subseteq^s \cap \subseteq^{\op}$, $\subseteq^s \cap \subseteq^{\op} \cap \subseteq^{\op}$, $\subseteq^d$, $\subseteq^d \cap \subseteq^{\op}$, $\subseteq^d \cap \subseteq^{\op} \cap \subseteq^{\op}$, $\subseteq^{\op}$, and their opposite orders. Moreover $\subseteq^r$ is the unique $\omega$-complete partial order with $\bot$ in the meaning of preorder enrichment of $\text{Set}_D$.
- We have showed that the (bi)similarities and the probabilistic (bi)similarities are characterised by the preorder $\top\top$-liftings.

Dropping the condition $d[X] \leq 1$, we also have the (countable) valuation monad $\mathcal{V}$. In fact, calculating $\text{Pre}(\mathcal{V})$ is easier than $\text{Pre}(D)$, since $\mathcal{V}$ is characterised as countable multiset monad generated by the semiring $[0, \infty]$. By [7, Lemma 7 and Theorem 8], each preorder on $\mathcal{V}$ is characterised as a pointwise ordering given by a preorder on $[0, \infty]$ which is closed under (countable) addition and constant multiplication.

We have the following future work at this time:

- We leave the question whether a preorder $\subseteq$ on $D$ such that $\subseteq^m \subseteq \subseteq \subseteq \subseteq^d$ exists.
In the paper [11], Sokolova and Woracek proved that there are exactly 5 congruences on the convex algebra \([0, 1]\). This fact corresponds to our result: there are exactly 5 equivalence relations in \(\text{CSPre}(D, 1)\), namely \(\Delta = \preceq \cap \preceq^\text{op}\), \(\preceq^\text{op} \cap \preceq^d \cap \preceq^\text{op}\), \(\preceq^d \cap \preceq^\text{op}\), \(\preceq^d \cap \preceq^d\), and \(D_1 \times D_1\). Our results and the Sokolova and Woracek’s results seem to be closely related. Thus, we intend to relate our work and [11], and we expect to extend our study using Sokolova and Woracek’s framework.

We expect to analyse preorders on other monads. For example, the convex module monad \(CM\) [5, 12] and the distribution monad \(D = 1\) which is given by the restriction \(d[X] = 1\) (for detail, see [10]).

Acknowledgements. The author thanks Shin-ya Katsumata for many valuable comments, corrections and discussions, Paul Blain Levy for stimulating discussions, Masahito Hasegawa, Ichiro Hasuo, and Norihiro Tsumagari for helpful comments.

References

Towards (Co)Algebraic Semantics of the Object-Oriented Rewriter Style Pattern

Baltasar Trancón y Widemann\textsuperscript{1,2} and Markus Lepper\textsuperscript{2}

\textsuperscript{1} Ilmenau University of Technology
\textsuperscript{2} <semantics/> GmbH

Abstract. Conventional object-oriented programming languages provide expressive, declarative means for data abstraction, but they lack similarly powerful counterparts for control abstraction. Style patterns have been developed as informal solutions to the problem. They fall short of real abstraction, because they mostly give no criteria how to use imperative language elements in a disciplined way for denotational semantics to be applied effectively. We discuss the Rewriter pattern, an extension of the well-known Visitor pattern, that expresses type-directed rewriting rules for arbitrary object graphs in a relatively declarative style. We demonstrate how to ground the Rewriter pattern in coalgebraic recursion theory, in particular the course-of-argument coiteration scheme, based on a formal model of object graphs as finite coalgebras.

1 Introduction

1.1 Object-Oriented Data Models

Whenever semantically nontrivial data, such as abstract syntax fragments of a formal language, structured documents, or semantic networks, are stored and processed in software written in the dominant object-oriented paradigm, some kind of formal data model is needed for semantic reasoning and verification.

The primary entities in such a model are, naturally, objects: they have persistent identity and (privately) mutable state, and are usually sorted into class types, possibly related by subtyping and/or inheritance. Objects form a network by referring to each other in their state. Hence, on a global scale, object-oriented data models specify directed graphs of objects. Objects together with their dynamic behaviour are well known to have neat formalizations as coalgebras [6].\textsuperscript{3}

But in reality the graph view is too simple; else we would be content to reduce object-orientation to graph theory: One object can figure in the state of another in many different capacities that are not in general adequately represented by the adjacency relation in a graph representation, even with labeled edges. The usual inner structure of object states consists of two levels: On the upper level, the state is a finite family of named fields. Field names could serve as edge labels if field

\textsuperscript{3} In this sense, data model objects have exceptionally simple and pure behaviour; they can merely be queried for their state.
values were trivial. But on the lower level, each field may contain a structured term, ultimately containing primitive values and object references in various aggregated forms. Hence, on a local scale, object-oriented data models specify adjacency algebras, as opposed to mere adjacency lists, for object graphs.

These algebras contain in the simple case type constructors such as optionally, list-of, set-of or map-between. For instance a Person object might be required to have the following fields:

- **spouse**, containing an optional (reference to a) Person object,
- **buddies**, containing a set of Person objects,
- **debtors**, containing a finite map of Person objects to numbers.

It is the purpose of a data model to specify the field layout and other properties of objects precisely and pointedly. For the sake of regularity and expressive power, it is important that the aggregating operators be compositional; nesting should be permitted in any combination and arbitrary depth.

Even though the local structure of composite field values is usually implemented as trees of auxiliary objects itself, these do not count as “objects” at the model level, and hence play a semantically different role: They are instances of imported library classes, for instance the Java collection framework java.util; and they have algebraic semantics, in the sense that content rather than identity determines their equivalence. Since equivalence checks are implemented by structural induction, this has the notable technical consequence of ruling out non-well-founded local aggregates. These semantics are not free, however: different constructor terms may yield equivalent aggregate values, for instance in the case of set enumeration. Thus, the reduction of the local scale to syntactic trees is ruled out as simplistic, just as the reduction of the global scale to graphs.

In summary, object-oriented data models are an interesting and challenging target for formal research: They constitute an effective middle ground between their competitors, offering more locality than entity–relationship models, richer structure than graph models, and more identity awareness than algebraic models. They have ubiquitous real-world applications in “business” software, document processors, compilers and interpreters, etc. On one hand, they feature sophisticated type systems, with incremental refinement and subtyping by inheritance, and general run-time type distinction by method dispatch. On the other hand, they are afflicted with weaknesses of the object-oriented programming paradigm inherited from its imperative ancestors: Referential transparency is broken by mutable data, and algorithmic expressiveness is poor due to a general value-level procedural approach.

1.2 The Visitor Style Pattern

The Visitor style pattern is a well-known technique for implementing type-directed traversal of object graphs in moderately declarative style. It has been

---

4 Advanced operators, for instance cardinality qualifiers such as no-less-than-three-of, are not considered here.

5 Try to compute the hash code of a Java list that contains itself directly or indirectly.
known informally for a long time; the authoritative reference is in [4], a standard textbook on object-oriented program design, where the Visitor pattern occurs at the high end of control constructs. It is used pervasively in real-world application programming.

A visitor class defines distinct methods for visiting all types of objects in the model and may have internal state to thread through the traversal. Many variants exist. In a fairly usual one, there are two mutually recursive phases: In the match phase, the dynamic type of the visited object is determined (by type case distinction or double dispatch). In the action phase, the fields of the object are “visited” recursively; visiting the fields of superclasses is delegated to the relevant method.

For instance, an abstract syntax model with Binary expression nodes and Plus as a subclass would have two visitor methods:

```java
void action(Binary x) {
    match(x.getLeft());
    match(x.getRight());
}
void action(Plus x) {
    action((Binary)x);
}
```

The user adds side effects by creating subclasses of the base visitor class, overwriting methods of the action phase. For instance, for a pre-order traversal affecting all Binary nodes, and for a post-order traversal affecting only Plus nodes, one would give the left and right redefinitions, respectively:

```java
void action(Binary x) {
    doSomething(x);
    super.action(x);
}
void action(Plus x) {
    super.action(x);
    doSomething(x);
}
```

This style results in highly concise, understandable and modular code. It provides fair robustness against trivial programmer errors and smooth changes in the definition of the underlying model, and is therefore an invaluable tool for agile development and rapid prototyping. Note that unconditional recursive descent into field values is safe for cycle-free object graphs only; to ensure termination is up to the user.

Unfortunately, whereas the Visitor pattern is adequate for data extraction, such as in the semantic actions of a parser, its reliance on side effects makes it ill-suited to declarative data transformation. The basic object-oriented approach of in-place modification of object state, when applied to highly structured data, opens a Pandora’s box of infamous problems and kludgy countermeasures, all related to the breakdown of referential transparency. We do not wish to go into the unpleasant technical details; the keywords are alarming enough: deep versus shallow cloning, concurrent modification detection, defensive copying, etc. This deficiency has lead us to study a benign special case which is inspired by rewriting approaches to declarative data transformation, and consequently called the Rewriter style pattern.
1.3 The Rewriter Style Pattern

The Rewriter style pattern is a much less known refinement of the Visitor pattern. It splits the action phase in several subphases, most of which are administrative and need not be redefined by the user. In marked contrast to the Visitor pattern and, even more so, the simpler patterns of [4], there is no simple Rewriter recipe in terms of code templates and algorithmic rules of thumb. Instead, Rewriter programming is guided by fairly deep and subtle semantic considerations. To formalize the underlying model is the main novel contribution of the present paper.

In a Rewriter, the user specifies rewrite rules, with apparently imperative effect, by overwriting methods to either modify the supplied source object or replace it with some other object. However, the concise and familiar notation of destructive updates is actually a (convenient) illusion, maintained on top of a non-destructive “copy-on-write” strategy: User-defined code manipulates only shallow clones of immutable input data, which are created on the fly. The resulting object graph is created using the clones where they have been modified, or possibly shared references to the original where no change has taken place. The internal state maintains the storage of results, and caches for sharing and cycle detection.

In contrast to traditional term rewriting and graph grammar approaches, an explicit evaluation strategy in the form of the traversal is built in; no issues of confluence or termination arise. This is adequate for tasks that decompose into multiple transformation passes, a ubiquitous feature of compilers and web applications.

For instance, to implement constant folding, a straightforward bottom-up program simplification pass, for the Plus operator, one would redefine:

```java
defining_rewriting(Plus x) {  super.rewriting(x);  if (x.getLeft().isConstant() && x.getRight().isConstant())    substitute(makeConstant(x.getLeft().getValue() +  x.getRight().getValue()));}
```

and to remove the references to the spouses in potentially cyclic graphs of Person objects, non-destructively in an otherwise faithful copy, one would write:

```java
defining_rewriting(Person x) {  x.setSpouse(null);  // x is already a clone  super.rewriting(x);
```

This is a top-down rewriting pass, which enables effective techniques for cycle detection and handling by a technique that can be considered the dual of memoization [9,10].

Note that, in all examples of user-redefined code, recursion is implicit by delegation to framework code. This is a first hint that an adequate formalization separates function generators from the underlying recursion scheme; see Section 2.3 below.
These examples already demonstrate a strategic dilemma: From practical applications, two fundamental types of rewriting tasks with incommensurable properties emerge: The bottom-up type comes with the power of induction and can hence perform evaluations regardless of nesting depth. The top-down type comes with the power of coinduction and can hence deal (theoretically) with infinite and (practically) with circular data. Semantics and implementation strategies for each pure type are fairly well understood. But for hybrid instances, required by nontrivial applications, a semantic theory does not yet exist.

1.4 Outline

It has been demonstrated that formal semantics for style pattern are not merely a theoretical exercise: Introducing a notion of semantic equivalence and the corresponding degrees of implementation freedom can create potential for substantial code optimizations, which can be leveraged automatically by programming tools and environments. For instance, the fact that the predefined action methods of a visitor have no side effect can be exploited to prune irrelevant paths in user-defined visitors, on the basis of operational semantics and a combination of static and dynamic analyses [7].

Unfortunately, programming style patterns are specified too loosely to give useful formal semantics to every application in a conventional, semantics-unfriendly host language. Thus a general descriptive approach is doomed to failure. One solution to this problem is to consider only a restricted subset of pattern instances, arising from an embedded domain-specific language or a code generator. The other solution is to turn the situation around: A prescriptive, particular semantics defines a set of corresponding well-behaved pattern instances. The programmer is given the conscious choice to either adhere to a discipline that satisfy the semantics, thereby reaping all the benefits of an elegant semantic model, or else trade additional programming flexibility for a rougher approximation of the ideal semantics. The prescriptive is the approach taken in the present paper.

To this end, the remainder of this paper is structured as follows: Sections 2.1 and 2.2 establish untyped/typed coalgebraic models of object-oriented data, respectively. Section 2.3 reviews suitable schemes of implicit recursion, based on (co)algebraic generator operations and universal homomorphisms. Section 2.4 deals with the characteristically declarative notion of referential transparency in coalgebraic terms. Section 2.5 introduces the novel concept of horizons in order to abstractly formalize patterns of data access, as far as they are relevant to rewriting strategies. Section 2.6, finally, gives a precise semantic explication of the elusive Rewriter pattern in terms of the established coalgebraic framework. Section 3 wraps up the discussion.

2 Formal Framework

2.1 Untyped Object-Oriented Data

Loosely following [6], we model data objects as elements of coalgebras. The difference to the more general theory of mutable objects exposed there is that we
consider only query operations that cause no state change. This gives a simple intuition of

- the carrier $A$ of a $U$-coalgebra as the space of live object references, and
- the operation $A \xrightarrow{\alpha} U(A)$ of a $U$-coalgebra as the dereferencing operation,

for some suitable object state (set) endofunctor $U$. Since the operation does not create new objects or states on the fly, we need only consider finite $U$-coalgebras.

The two-level structure of objects proper and their field values shall be reflected by the decomposition of $U$ into two distinct functors.

On the global level, an object has a nominal type (a class name), and a finite set of named fields. Let $C, F$ be the sets of class and field names, respectively. Then the global structure is specified by the functor

$$O(X) = C \times (F \Rightarrow X)$$

where $\Rightarrow$ denotes the finite partial map construct.

On the local level, choose a collection of operations for the field value algebra. Then specify a signature functor $L$, omitting the case of object references. For instance, having primitive values $\mathbb{V}$, pairs, finite sequences and finite sets gives:

$$L(X) = \mathbb{V} + (X \times X) + X^* + \mathcal{P}_\omega(X)$$

Now define the object functor $U$ as the composition

$$U = OL^*$$

where $L^*$ is the free monad over $L$. This gives the desired algebraic semantics to aggregate values, with compositional support for list of sets of lists etcetera.

Maps of the form $U(f)$ are essential to rewriting, as they perform substitution of references in field values in a way that compositionally respects the semantics of aggregates: For instance, a data structure representing a set must not be rewritten naively with a non-injective map $f$, lest duplicates arise.

Assuming that no qualitatively different operations are added to $L$, the functor $U$ is well-behaved: It has both initial algebras and final coalgebras, it is finitary, and it preserves weak pullbacks, thus giving rise to relational lifting via spans, and many other useful properties in the context of coalgebras; see [5].

### 2.2 Type Discipline

The above formal account of object-oriented data is effectively untyped, as all references in field values range uniformly over the whole carrier. A type discipline that classifies $U$-coalgebras as either well- or ill-typed, can be added in a very abstract form: Consider a type definition map $t : C \to \mathcal{P}(U(C))$ that relates class names to admissible state templates, where only an expected type is given in place of each reference.

For each $U$-coalgebra $(A, \alpha)$ we have a dynamic type map $d = \pi_1 \circ \alpha : A \to C$. The $U$-coalgebra is well-typed with respect to $t$ if and only if the dynamic type
of each object $x \in A$ is related to the template obtained by retaining only the dynamic type of each reference:

$$U(d)(\alpha(x)) \in t(d(x))$$

The question which subsets of $U(C)$ can appear in the codomain of $t$ depends on the type system provided by the programming language, of which there is a great variety with various levels of semantic elegance. Actual practice is more complicated than the account given here, because of features such as subtyping, inheritance, parametric polymorphism, and run-time type-level calculations via reflection. However, for a discussion of semantic foundations, the simple account given here suffices.

### 2.3 Recursion Schemes for Rewriting

Rewriters are to be understood as reified recursive maps between data. Hence their semantics should be given as maps on the carriers of $U$-coalgebras, arising from some generator that takes the coalgebraic structure into account, preferably via a standard recursion scheme. We base our approach on [11] where a collection of candidate recursion schemes is elaborated.

In a nutshell, inductive schemes that produce a map of type $\mu F \to X$ from a generator $G$-algebra on $X$, are contrasted with coinduction schemes that produce a map of type $X \to \nu F$ from a generator $G$-coalgebra on $X$. Here $\mu F$ and $\nu F$ denote the (carrier of the) initial $F$-algebra and final $F$-coalgebra, respectively. Without loss of generality, we assume the natural embedding $\mu F \to \nu F$ to be an inclusion.

The complexity of a scheme lies in the relation between the data functor $F$ and the generator functor $G$. For the inductive and coinductive side, a succession of three, pairwise dual schemes is considered:

1. The simplest choice $G = F$ gives the familiar schemes of (co)iteration, where the generator is only allowed to recur to immediate subresults/subarguments, respectively.
2. The choice $G = \mu F \times F$ and $G = \nu F + F$, respectively, gives the equally familiar schemes of primitive (co)recursion, where the generator has additional access to the whole argument/result.
3. The choice $G = FF_*$ and $G = FF^*$, respectively, gives the schemes of course-of-value iteration and course-of-argument coiteration. Here $F_*(X) = \nu(X \times F)$ and $F^*(X) = \mu(X + F)$ are the cofree comonad and free monad over $F$, respectively.

The last scheme, course-of-argument coiteration, is by far the least familiar of the six. However, we shall demonstrate that it is the appropriate recursion scheme for rewriters, as it can flexibly accommodate both top-down and bottom-up traversal strategies, and their hybrids. By contrast, the pure types of traversal are already described sufficiently by (co)iteration, respectively. We are confident that the Rewriter pattern can provide interesting nontrivial example applications of the, as yet largely theoretic, course-of-argument scheme.
2.4 Referential Transparency

The fact that data objects have a unique and persistent identity is both a blessing and a curse. On one hand, it allows explicit program control of the graph structure, for dealing with both sharing (desirable for efficiency) and cycles (essential for termination). On the other hand, object identities are too specific for many semantic considerations, as they break referential transparency: programs may associate background information with an object that is not part of its state, and technical layout decisions such as sharing and cloning cannot be contained transparently in a system-level program layer, but are exposed to the user.

From a declarative and semantically motivated viewpoint, however, data queries and transformation should have uniform behavior regardless of storage layout. The solution is to consider what we call coalgebraically natural operations: A family of maps \( f_\alpha : F(A) \to G(A) \), parameterized with a class \( \mathcal{C} \) of \( H \)-coalgebras \((A, \alpha)\), for fixed endofunctors \( F, G, H \), is called coalgebraically natural with respect to \( \mathcal{C} \), if and only if it commutes with homomorphisms on \( \mathcal{C} \):

\[
H(h) \circ \alpha = \beta \circ h \implies G(h) \circ f_\alpha = f_\beta \circ F(h)
\]

for all \((A, \alpha), (B, \beta) \in \mathcal{C}\) and \( h : A \to B \). In analogy to natural transformations proper, we write \( f : F \Rightarrow_C G \).

This concept is flexible enough to express various degrees of referential transparency: If \( \mathcal{C} \) contains a final \( H \)-coalgebra, then no information from the concrete carrier of the domain coalgebra is leaked at all; the natural operation is compatible with final semantics of data objects.

Prominent examples of natural operation families, with respect to the class of all \( H \)-coalgebras, are: natural transformations proper of type \( A \Rightarrow B \), constant maps, the coalgebra operations, the unique homomorphisms to a final coalgebra.

2.5 Rewriting Strategies

Rewriting strategies may be devised using pragmatic information that is not explicit in the data model. The only obvious generally valid strategy is top-down, such as formalized by the coiteration scheme; see the next subsection. But in some known program contexts, certain classes in a model might only ever be used for cycle-free data, even if they are theoretically mutually recursive; for instance, a compiler pass may assume cycle-free expressions for the purpose of simplification. In that restricted case, bottom-up rewriting, such as formalized by the iteration scheme, is perfectly safe.

While such meta-knowledge is difficult to formalize in general, a promising and fairly powerful approach makes computation relative to a loose specification of legal observations that are safe for bottom-up processing.

In a first step, we assign a semantic domain of observations to the information obtainable from data objects by iterated dereferencing, namely \( U^* \), the free monad over \( U \). We write \( e \) and \( m \) for the monad unit and multiplication, respectively. Without loss of generality, we take the natural isomorphism \( \mu U \cong U^*(\emptyset) \) to be
identity, and the natural embedding \( \mu U \to U^*(A) \) to be inclusion. We consider morphisms in the Kleisli category of \( U^* \), where we write composition as \( f \circ g \). In particular, for any \( U \)-coalgebra \( (A, \alpha) \), there is a morphism \( \bar{\alpha} : A \to U^*(A) \), obtained by composing \( \alpha \) with the natural embedding \( U \to U^* \). Note that this natural embedding decomposes into a sequence \( U \Rightarrow UU \Rightarrow U^* \).

We call an observation \( u \in U^*(A) \) ground if and only if also \( u \in \mu U \). It can be shown that a map with codomain \( \nu U \) defined by course-of-argument coiteration is non-recursive, that is, it coincides with its generator, for all arguments where the latter yields a ground result.

We call a \( U \)-coalgebra \( (A, \alpha) \) cycle-free if and only if \( \{ \alpha \} : A \to \nu U \), the unique homomorphism to the final \( U \)-coalgebra, is restricted to codomain \( \mu F \).

For any \( U \)-coalgebra \( (A, \alpha) \), the observation refinement relation \( \preceq \) on \( U^*(A) \) is the smallest relation that is a preorder (reflexive and transitive), subsumes the (monadic embedding of the) graph of \( \alpha \) and is closed under relational \( U^* \)-lifting:

\[
\begin{align*}
  u & \preceq u \\
  u & \preceq v \preceq w \\
  e_A(x) & \preceq \bar{\alpha}(x) \\
  s & \preceq (U^*) t \\
  m_A(s) & \preceq m(t)
\end{align*}
\]

for all \( x \in A; u, v, w \in U^*(A); s, t \in U^*(U^*(A)) \). By point-wise extension, for any maps \( f, g : D \to U^*(A) \) with arbitrary domain \( D \), we write \( f \preceq g \) if and only if \( f(d) \preceq g(d) \) for all \( d \in D \). The set of possible observations from an object \( x \in A \), by arbitrarily nested dereferencing, is precisely the up-set of \( e_A(x) \).

In a second step, we consider maps that take objects to observations with a certain regularity. Let \( \sigma : 1 \to \nu U \) be a coalgebraically natural operation with respect to a class of \( U \)-coalgebras \( C \). It is called a horizon if and only if

\[
e_A \preceq \sigma \preceq \sigma \circ \bar{\alpha}
\]

In prose, if we conceive of \( \sigma(x) \) as the neighbourhood that can be observed from object \( x \), then the first inequation states that observation is consistent with iteration of \( \alpha \), and the second inequation states that observation is convex: From no object we can see farther than from all its successors combined.

Horizons can be attributed with various regularity properties. We call a horizon

- \( k \)-bounded iff \( \sigma \preceq \bar{\alpha}^k \), and dually \( k \)-extended iff \( \bar{\alpha}^k \preceq \sigma \),
- (globally) far iff it is \( k \)-extended for unbounded \( k \),
- locally far iff it is far on cycle-free objects,

respectively for all \( U \)-coalgebras \( (A, \alpha) \in C \). Only a horizon for a class of cycle-free coalgebras can be globally far. A (locally) far horizon yields only ground observations (on cycle-free objects), respectively.

Analogous properties can be defined with respect to a particular class of data objects only, for more fine-grained control. For instance, a horizon that is
globally far with respect to expression nodes would be adequate for the compiler pass example. This implies that \( \mathcal{C} \) may not contain coalgebras with circular expressions.

### 2.6 Rewriters Put Together

As outlined above, the semantics of rewriters shall be given as generators for the course-of-argument coiteration scheme.

Choose a class \( \mathcal{C} \) of finite \( U \)-coalgebras, called admissible. Choose a nonempty class \( \Sigma \) of horizons with respect to \( \mathcal{C} \), likewise called admissible. Define a coalgebraically natural operation \( \varphi_\alpha : A \times U^*(A) \to U(U^*(A)) \) for all \( U \)-coalgebras \((A, \alpha) \in \mathcal{C}\). Then each choice of a horizon \( \sigma \in \Sigma \) gives a generator

\[
\varphi_\alpha \circ \langle \text{id}_A, \sigma_\alpha \rangle : A \to U(U^*(A))
\]

which in turn generates a map \( \Phi_\alpha : A \to \nu U \) via course-of-argument coiteration. If the same map is generated for any choice of \( \sigma \), then \( \varphi \) is called a valid specification for a rewriter. See [11] for a necessary and sufficient condition of equivalence of course-of-argument coiterations.

Concrete implementations of the specified rewriter are supposed to act as maps \( r \) that transform each finite \( U \)-coalgebra \((A, \alpha) \in \mathcal{C}\), which represents a valid memory state, to another finite \( U \)-coalgebra \((B, \beta)\), such that \( \{\beta\} \circ r = \Phi_\alpha \).

In words, \( r \) may create an arbitrary layout of concrete result objects, with final semantics specified by \( \Phi \).

The possible actions of a rewriter, as outlined in section 1.3, translate to a generator \( \varphi : A \times U^*(A) \to U(U^*(A)) \) as follows:

- For top-down traversal, use the first argument (the object identity) and ignore the second.
- For a bottom-up traversal, ignore the first argument and use the second in an indirect way: The free monad structure of \( U^* \) gives a parametric form of the iteration scheme, that produces bottom-up traversing maps with domain \( U^*(A) \) from \( U \)-algebra generators. From the set \( \Sigma \) of admissible horizons, certain restrictions on the subset of \( U^*(A) \) that can occur here can be deduced and exploited.
- The two pure strategies can be hybridized, for instance by class-based switching.
- On the result side, the local state observation of (the provisional clone of) an object \( x \in A \) is specified by \( \hat{x} = \hat{U(e_A)}(\alpha(x)) \in U(U^*(A)) \). This can be transformed by local updates, such as field re-assignments or aggregate mutations, or it can be substituted by another object \( y \), likewise represented by its local state observation \( \hat{y} \).

In order to qualify as a semantic theory, the formal recipe we have given so far needs to be complemented with interesting or useful properties. That part of the work is still in progress, but we have identified a minimal theory, consisting of four cornerstone properties, which we consider as self-evidently required:
1. The operation $\Phi$ is coalgebraically natural with respect to $C$.
2. The operation $\Phi$ takes finite inputs to finite outputs.
   This is a prerequisite for having effective implementations.
3. The rewriter recursion scheme subsumes ordinary iteration. Given a class $C$ of cycle-free coalgebras and a class $\Sigma$ of far horizons, a generic rewriting rule $\psi : \mu U \rightarrow \mu U$ can be transformed isomorphically to type $U(\mu U) \rightarrow \mu U$, and hence gives rise to a bottom-up recursive map $\{\psi\} : \mu U \rightarrow \mu U$ by iteration. The codomain is isomorphic to $U(\mu U)$ and widens to $U(U^*(A))$. By a completion to domain $U^*(A)$, which is irrelevant under composition with a far $\sigma$ (invoking the axiom of choice), and an additional first argument of type $A$, which is ignored altogether, we obtain a valid rewriter specification $\phi$, such that $\Phi_\alpha = \{\psi\} \circ \{\alpha\}$.
   This is the pure bottom-up base case.
4. The rewriter recursion scheme subsumes ordinary coiteration. Given a coalgebraically natural family of generators $\psi : A \rightarrow U(A)$ fitting the coiteration scheme, by widening to codomain $U(U^*(A))$ and an additional second argument of type $U^*(A)$, which is ignored, we obtain a valid rewriter specification $\phi$, such that simply $\Phi_\alpha = \{\psi_\alpha\}$. No specific restriction on $C$ or $\Sigma$ is required.
   This is the pure top-down base case.

An informal proof is available for each, but as long as those do not conform to higher standards of formal rigor, the properties should be regarded as plausible conjectures rather than theorems.

### 3 Conclusion

#### 3.1 Discussion

We have demonstrated a coalgebraic treatment of object-oriented data models. It comes with a formal separation of two levels: The upper one is a global level, where objects have explicit identities, at least insofar as finite cycles exist and can be distinguished effectively from periodically infinite regress. The lower one is a local level, with intended algebraic properties, where a technical implementation by means of objects is rather an accident. The composite formalization gives nice final semantics for data objects, and compositionality for local transformations.

We have discussed the problem that programs may technically observe actual object identities, but not be semantically allowed to deduce any information except what can be gained by dereferencing. We have introduced the notion of coalgebraically natural operations to capture all sorts of such assumptions, explicated by a class of coalgebras on which the operation must act uniformly. Elaborating on the same idea, we have captured the fact that rewriting transformations may come with additional assumptions regarding the shape of data and the amount of deep dereferencing they may carry out, by making rewriting relative to a class of shapely coalgebras, and a class of admissible horizons which formalize consistent, but possibly irregular, bounds to dereferencing.
Some of the constraints expressible by these boundary conditions are also addressed by a type system, for instance which fields an object in a certain position should have. Others are conceptually harmless, but beyond your average type system, for instance the fact that some prima facie recursive object type must not form cycles at all or beyond a certain length, or be nested only to a certain depth. Others again are within the realm of dependently typed programming, such as relational constraints between different field values within an object, or complex modal constraints on paths in an object graph. We have not considered any syntactic means for declaring such constraints in programs, although that is a possible direction for future research. For now, we are content with the state of the art, which states semantic assumptions beyond type in prose form in programming interface documentations.

Finally, we have demonstrated a recursion scheme for the Rewriter style pattern. It also comes with a formal separation of two concerns: The specific generator aspect, which is implemented explicitly by the user when instantiating the pattern, and the generic recursion aspect proper, which is implemented transparently by the programming environment. To this end, we have chosen the particular scheme of course-of-argument coiteration, which has been lacking serious applications so far. Theoretically, it subsumes and unifies the simpler schemes of iteration and coiteration, which are the basis of the pure bottom-up and top-down traversal strategies, respectively. Practically, it maps to proven implementation techniques such as the ones described in [10].

The semantic domain that we propose to assign to instances of the Rewriter pattern consists of total, pure functions, and as such evidently falls short from a full denotational semantics for an imperative programming language, such as our primary target Java. There are several obvious ways in which a program can fail to behave as such a function. We completely ignore external side effects, such as stateful interaction with the world at large, as well as abnormal termination behavior, such as divergent loops and exceptions. All of these are frowned upon in a declarative style such as the one underlying the Rewriter pattern, and should be avoided if at all possible.

We also ignore internal side effects, namely changes in the private state of a Rewriter object during traversal. Under the hood, such effects are essential to the implementation of the recursion scheme, in particular to the handling of sharing and cycles. User-defined mutable state is not explicitly discouraged, but neither is it directly supported by semantic theory. A bad omen for the future of such support is the relatively disappointing nature of the obvious candidate formalism, namely monadic coiteration [8].

There is hope that the behavior of real-world programs, necessarily falling short of any semantic theory idealized for mathematical simplicity, is a graceful approximation: Simply speaking, more disciplined use of technical means should imply more accuracy of elegant semantic models. More aggressively, semantic idealizations remain exact as long as well-understood breaches of declarative programming style are avoided. This would validate our approach from a software engineering perspective, but needs to be corroborated by future empirical research.
3.2 Related Work

The problem of semantic equivalence, or deep equality, of objects has been studied extensively in the realm of object-oriented databases, culminating in work [1] that, with hindsight, strongly suggests a coalgebraic treatment with final semantics, although the framework had not been fully developed at the time.

In between the initial algebra $\mu U$, as the universal semantic domain of cycle-free, or well-founded data, and the final coalgebra $\nu U$, as the universal semantic domain of arbitrary, infinitely nested data, there is the intermediate $\rho U$, the rational fixpoint of $U$ [2,3], which includes cycles but excludes infinite regress. We have not found a need to introduce it here, but some of its associated rich theory might be useful in future work.

The exposition of recursion schemes in [11] derives course-of-argument coiteration briefly from its better-known dual, course-of-value iteration. Consequently, the associated functor and natural transformations are constructed explicitly; the connection to the free monad is not mentioned there.

The course-of-argument variant of coiteration increases the reach of a single step of top-down rewriting, but does not adequately capture the simple act of substituting a constant object. This is provided by an orthogonal extension of plain coiteration, namely primitive corecursion; the dual of primitive recursion where both the original and rewritten subterms are available for a bottom-up step. It is not clear whether the unification of the two extensions is worthwhile: The theoretical accounts of primitive (co)recursion are rather awkward in comparison with the simplicity of their efficient implementation. An equivalent observation was proposed as a baffling open problem already in [12].

References

4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)


Author Index

Asăvoae, Irina Măriuca 1
Asăvoae, Mihail 1
Lepper, Markus 49
Ochremiak, Joanna 17
Riesco, Adrián 1
Sato, Tetsuya 33
Trancón y Widemann, Baltasar 49