Electronic Notes in Theoretical Computer Science

Mathematical Foundations of Programming Semantics
Twenty-eighth Annual Conference

University of Bath
Bath, England
June 6 – 9, 2012

Guest Editors:

Ulrich Berger and Michael Mislove
Contents

Preface ................................................................. v

MIKE BEHRSCH, SEBASTIAN KERKHOFF AND JOHN POWER (Invited Speaker)
Category Theoretic Understandings of Universal Algebra and its Dual: Monads and Lawvere Theories, Comonads and ? .............. 1

DEXTER KOZEN (Invited Speaker)
New ................................................................. 13

INGO BATTENFELD
Observationally-induced Effects in Cartesian Closed Categories ... 39

ULRICH BERGER AND TIE HOU
Typed vs. Untyped Realizability ........................................ 53

FILIPPO BONCHI, MARCELLO BONSANGUE, GEORGIANA CALTAIS, JAN RUTTEN AND ALEXANDRA SILVA
Final Semantics for Decorated Traces ............................... 67

STEPHEN BROOKES
Syntactic Control of Interference and Concurrent Separation Logic 81

MATTEO CIMINIA, MOHAMMANDREZA MOUSAVI, MICHEL A. RENIERS AND MURDOCH J. GABBAY
Nominal SOS ......................................................... 97

ROBIN COCKET, JOAQUÍN DÁZ-BOÍLS JONATHAN GALLAGHER AND PAVEL HRUBE
Timed Sets, Functional Complexity, and Computability .......... 111

ROBERT DOCKINS AND AQUINAS HOBOR
Time Bounds for General Function Pointers ....................... 132

ROY DYCKHOFF, MEHRNOOSH SADRZADEH AND JULIEN TRUFFAULT
Algebra, Proof Theory and Applications for a Logic of Propositions, Actions and Adjoint Modal Operators ......................... 149
Murdoch Gabbay and Dan Ghica

Game Semantics in the Nominal Model .......................... 165

Dan Ghica and Nikos Tzevelekos

A System-level Semantics ........................................ 180

Naohiko Hoshino

A Representation Theorem for Unique Decomposition Categories .. 201

Wolfgang Jeltsch

Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming .................. 215

Joachim Kock

Data Types with Symmetries and Polynomial Functors over Groupoids ......................................................... 229

Andrew Lawrence, Ulrich Berger and Monika Seisenberger

Extracting a DPLL Algorithm ....................................... 244

Extracting a DPLL Algorithm

A Simply Typed λ-Calculus of Forward Automatic Differentiation . 259

Guy McCusker, John Power and Cai Wingfield

A Graphical Foundation for Schedules .............................. 274

Ugo Montanari and Matteo Sammartino

Network Conscious π-calculus: a Concurrent Semantics ............ 290

Barbara Petit

Continuation Models for the Lambda Calculus with Constructors .. 306

Dan Teodosiu

A Truly Concurrent Process Semantics over Multi-Pomsets of Consumable Resources ...................................... 320

Toby Wilkinson

A Characterisation of Expressivity for Coalgebraic Bisimulation and Simulation ................................................. 335
Preface

This volume collects papers presented at the 28th Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXVII), held on the campus of the University of Bath, England, from Wednesday, June 6 through Saturday, June 9, 2012.

The MFPS conferences are devoted to those areas of mathematics, logic, and computer science that are related to models of computation, in general, and to the semantics of programming languages, in particular. The series particularly stresses providing a forum where researchers in mathematics and computer science can meet and exchange ideas about problems of common interest. As the series also strives to maintain breadth in its scope, the conference strongly encourages participation by researchers in neighboring areas.

The Organizing Committee for MFPS consists of Stephen Brookes (CMU), Achim Jung (Birmingham), Catherine Meadows (NRL), Michael Mislove (Tulane), and Prakash Panangaden (McGill). The local arrangements for MFPS XXVII were overseen by John Power and Guy McCusker.

The MFPS XXVIII Program Committee members are:

Thorsten Altenkirch (Nottingham)   Achim Jung (Birmingham)
Steve Awodey (CMU)               Daniel Leivant Indiana
Andrei Bauer (Ljubljana)           Guy McCusker (Bath)
Ulrich Berger (Swansea (Chair))  Catherine Meadows (NRL)
Bob Coecke (Oxford)              Michael Mislove (Tulane)
Stephen Brooks (CMU)             Peter O’Hearn (Queen Mary)
Martin Escardo (Birmingham)     Luke Ong (Oxford)
Marcelo Fiore (Cambridge)         Prakash Panangaden (McGill)
Neil Ghani (Strathclyde)           John Power (Bath)
Alexey Gotsman (IMDEA, Madrid) Jan Rutten (CWI)

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
Hugo Herbelin (INRIA, Rocquencourt)  Alex Simpson (Edinburgh)

James Worrell (Oxford)

The following people gave invited plenary talks at the meeting:

Steve Awodey (CMU)  Michael Clarkson (GWU)
Patricia Johann (Strathclyde)  Dexter Kozen (Cornell)
M. Andrew Moshier (Chapman)  John Power (Bath)

In addition, there were three special sessions:

(i) A Special Session Logic, computation and algebraic topology, organized by Steve Awodey and Michael Mislove. In addition to Professor Awodey’s plenary lecture, the session will include invited talks by Andrej Bauer (Slovenia), Nicola Gambino (Palermo), Eric Goubault (CSA) and Sanjeevi Krishnan (Penn).

(ii) A Special Session on Computational effects, organized by John Power. In addition to Professor Power’s plenary address, the session will include invited talks by Andrej Bauer (Slovenia), Neil Ghani (Strathclyde), Alex Simpson (Edinburgh) and Sam Staton (Cambridge).

(iii) A Special Session on Computability on Continuous Data, organised by Drew Moshier. In addition to Professor Moshier’s plenary talk, the session will include invited talks by Nick Bezhanishili (Imperial), Jens Blanck (Swansea), Martin Escard (Birmingham) and Paul Taylor. This session is part of the ATY.

Finally, the meeting included a series of Tutorial Lectures Game Semantics. These are being organized by Andrea Schalk (Manchester) and Paul-Andre Mélies (Paris VII), The speakers will include the organizers as well as Martin Hyland (Cambridge) and Luke Ong (Oxford).

It remains for us to thank all authors and speakers, the organizers of special sessions, the program committee, and the external referees for their contribution to the success of the conference.

We also would like to thank the US Office of Naval Research for its continued support of the MFPS series.

_Ulrich Berger_

_Michael Mislove_
Abstract

Universal algebra is often known within computer science in the guise of algebraic specification or equational logic. In 1963, it was given a category theoretic characterisation in terms of what are now called Lawvere theories. Unlike operations and equations, a Lawvere theory is uniquely determined by its category of models. Except for a caveat about nullary operations, the notion of Lawvere theory is equivalent to the universal algebraist’s notion of an abstract clone. Lawvere theories were soon followed by a further characterisation of universal algebra in terms of monads, the latter quickly becoming preferred by category theorists but not by universal algebraists. In the 1990’s began a systematic attempt to dualise the situation. The notion of monad dualises to that of comonad, providing a framework for studying transition systems in particular. Constructs in universal algebra have begun to be dualised too, with different leading examples. But there is not yet a definitive dual of the concept of Lawvere theory, or that of abstract clone, or even a definitive dual of operations and equations. We explore the situation here.

Keywords: Universal algebra, Lawvere theory, abstract clone, monad, comonad.

1 Introduction

There have been two main category theoretic formulations of universal algebra. The earlier was by Bill Lawvere in 1963 [16]. Nowadays, his central construct is usually called a Lawvere theory, more prosaically a single-sorted finite product theory [1,2].

Lawvere made a careful distinction between the notions of Lawvere theory and equational theory. Equational theories are a form of presentation for Lawvere theories: every equational theory determines a Lawvere theory and every Lawvere theory is determined by an infinite class of equational theories. Choosing good presentations for a Lawvere theory and deriving an invariant description of the theory from...
a presentation are important, but the semantics of a Lawvere theory can be considered independently of that [16]. We give the definitions and outline the situation in Section 2.

Universal algebraists have had the same concerns about the lack of invariance of presentations as category theorists have had. They have long used the notion of clone, of which Lawvere was aware, and they have gradually moved towards that of abstract clone [4]. Subject to a caveat about nullary operations, the notion of abstract clone is equivalent to that of Lawvere theory. So, subject to the caveat, universal algebraists readily recognise the definition of Lawvere theory and accept its significance for universal algebra. We give the definitions and constructions in Section 3.

The second category-theoretic formulation of universal algebra, which was in terms of monads, has a more complicated history and is much less accepted by universal algebraists. Monads typically arise from adjoint pairs of functors. The notion of monad (or triple or standard construction) arose in algebraic topology for reasons distinct from universal algebra, see for instance [6]. In 1965, Eilenberg and Moore noted that in case $T$ is the free group monad, their category of $T$-algebras is the category of groups [5].

In 1966, Linton made the general connection between monads and Lawvere theories: every Lawvere theory gives rise to a monad on $Set$ whose category of algebras is equivalent to the category of models of the Lawvere theory, and, subject to a generalisation in the definition of Lawvere theory, every monad arises thus, uniquely up to coherent isomorphism [17]. So Linton focused on a generalisation of the notion of a Lawvere theory, one that corresponds exactly to the notion of monad. We give the details in Section 4.

Monads have been the more common category theoretic formulation of universal algebra, see for example [19]. But Lawvere theories relate more closely to universal algebra; they arose directly from universal algebra; and they allow natural constructions that arise in universal algebra, such as the sum or tensor of theories, while monads do not [9]. So it seems little wonder that, although many universal algebraists are aware of monads, they seem generally not to have found them, or an equivalent notion, very helpful. Much of the relevant historical development has been summarised in [9].

Over the past decade or so, category theorists, computer scientists and universal algebraists have all become interested in the dual of this situation, for a variety of reasons [7,10,14,24,26,29].

A leading example for the interest by computer scientists arises from transition systems, which play a fundamental role in, for example, concurrency [21]. A finitely branching transition system is given by a set $S$ together with a function $t : S \rightarrow P_f(S)$, where $P_f(S)$ is the set of finite subsets of $S$. The functor $P_f$ on $Set$ generates a cofree comonad $G(P_f)$, and $t$ corresponds to the $G(P_f)$-coalgebra that sends an element $\sigma$ of $S$ to the set of all possible streams of transitions generated by $t$ with source $\sigma$. We would like to develop a body of theory that is dual to universal algebra and includes this example, but it is not easy. We outline some of the issues
in Section 5.

One approach to dualising the theory of Lawvere theories is by defining a co-
model of a Lawvere theory $L$ in $\text{Set}$ to be a model of $L$ in $\text{Set}^{\text{op}}$. That line of thought
has proved to be valuable for category theorists, for computer scientists, and for uni-
versal algebraists [14,24,26]. The category of comodels induces a comonad and is
the category of coalgebras for the induced comonad on $\text{Set}$. But this approach does
not include transition systems.

Alternatively, one can dualise definitions associated with presentations, carefully
dualising the structure of [13,27]. That line of thought has value too [7]. But the
fact that $\text{Set}$ is a locally finitely presentable category while $\text{Set}^{\text{op}}$ is not, leads to less
elegant results than one would wish [7], and one loses the presentation independence
that is central to the notions of Lawvere theory and abstract clone.

Linton’s generalised notion of Lawvere theory, corresponding exactly to the no-
tion of monad on $\text{Set}$, suggests a third approach: Linton did not require size con-
ditions [17,18], so dualising his definition is immediate, yielding a definition that is
equivalent to that of comonad on $\text{Set}$, thus including transition systems. But sums
of monads need not exist, so products of comonads need not exist either; similarly
for tensors.

So the question is open, hence the question mark in the title of this paper. We
outline the above three proposals in Section 6, and we propose a tentative definition
of a dual Lawvere theory in Section 7, leaving its development for further work.

2 Lawvere theories

In his 1963 PhD thesis, Lawvere gave a category theoretic formulation of universal
algebra along the following lines.

**Definition 2.1** Let $\aleph_0$ denote a skeleton of the category of finite sets and all func-
tions between them, considered as a category with strictly associative coproducts.

Since $\aleph_0$ is equipped with strictly associative finite coproducts given by the
ordinal sum of natural numbers, the opposite category $\aleph_0^{\text{op}}$ is equipped with strictly
associative finite products. It is equivalent to the free category with finite products
on 1 as $\aleph_0$ is equivalent to the free category with finite coproducts on 1.

**Definition 2.2** A **Lawvere theory** consists of a small category $L$ with (necessarily
strictly associative) finite products and a strict finite-product preserving identity-
on-objects functor $I : \aleph_0^{\text{op}} \rightarrow L$. A map of Lawvere theories from $L$ to $L'$ is a
(necessarily strict finite-product preserving) functor from $L$ to $L'$ that commutes
with the functors $I$ and $I'$.

Thus the objects of any Lawvere theory $L$ are exactly the objects of $\aleph_0$, and
every function between such objects yields a map in $L$. Note that the functor $I$
need not be an inclusion. One often refers to the maps of a Lawvere theory as
operations. The notion of map between Lawvere theories encapsulates the idea of
an interpretation of one theory in another.
The definitions of Lawvere theory and map between them yield a category $\text{Law}$, with composition given by ordinary composition of functors. The category $\text{Law}$ is complete and cocomplete, indeed a locally finitely presentable category.

Given an equational theory, one generates a Lawvere theory by putting $L(n, 1) = F_n$, the free algebra on $n$ generators. This determines $L(n, m)$ for any $m$ as $L(n, m)$ must be the product of $m$ copies of $L(n, 1)$. The composition of $L$ is fully determined by the family of maps

$$(F_p)^n \times F_n \rightarrow F_p$$

determined by substitution of $n$ terms of $p$ variables into a term of $n$ variables.

For most mathematical purposes, one understands a Lawvere theory by study of its models.

**Definition 2.3** A model of a Lawvere theory $L$ in a category $C$ with finite products is a finite-product preserving functor $M : L \rightarrow C$.

Note that one has preservation of finite products here, not strict preservation. Preservation rather than strict preservation of finite products is fundamental: if one demanded strict preservation, the category of models for the Lawvere theory for a monoid would be empty, rather than being the category of monoids as one wants. The reason is that, with the usual set-theoretic definitions, finite products in $\text{Set}$ are not strictly associative, whereas they are strictly associative in any Lawvere theory. Preservation rather than strict preservation also allows a smooth account of change of base category along a finite product preserving functor $H : C \rightarrow C'$.

The requirement that $M$ preserves projections, which is part of what preservation of products means, determines the behaviour of $M$ on all operations of the form $If$: projections in $L$ amount to coprojections in $\aleph_0$, and every function $f$ is given by a family of coprojections. So what determines a model is the interpretation of the other operations.

There is a mild difference here between universal algebra and category theory traditions. Some universal algebraists only admit non-empty models [3]. So, for example, a category theorist would regard the empty set as a carrier for the structure of a semigroup, whereas some universal algebraists would not. For a category theorist, the empty semigroup is important as it is the initial object in the category of semigroups.

**Definition 2.4** For any Lawvere theory $L$ and any category $C$ with finite products, the category $\text{Mod}(L, C)$ is defined to have objects given by all models of $L$ in $C$, with maps given by all natural transformations between them.

The definition of map in $\text{Mod}(L, C)$ is subtle. One can readily prove that any natural transformation between models respects finite products: for any natural transformation $\alpha$ between models $M$ and $N$, and for any $n$ in $\aleph_0$, the map $\alpha_n : Mn \rightarrow Nn$ is given by the product of $n$ copies of the map $\alpha_1 : M1 \rightarrow N1$. Thus the maps in $\text{Mod}(L, C)$ could equally be defined to be those natural transformations that respect the product structure of $L$. 
The semantic category $C$ of primary interest is $\text{Set}$. Consider a model $M$ of a Lawvere theory $L$ in $\text{Set}$. The set $M1$ determines $Mn$ up to coherent isomorphism for every $n$ in $L$: for $M$ preserves finite products of $L$, equivalently of $\aleph_0^{op}$; these are finite coproducts of $\aleph_0$, which are given by finite sums; and so $Mn$ must be a product of $n$ copies of $M1$. Thus to give a model $M$ is equivalent to giving a set $X = M1$ together with, a function from $Xm$ to $X$ for each map of the form $f : m \rightarrow 1$ in the category $L$, subject to the equations given by the composition and product structure of $L$; and Mod($L, \text{Set}$) is equivalent to the evident category of such structures. This analysis routinely extends to any category $C$ with finite products.

The category Mod($L, \text{Set}$) is always complete and cocomplete, with the initial object given by the empty set if $L$ has no nullary operations.

Unlike equational theories, Lawvere theories are semantically invariant. The precise sense in which that is so is as follows. With each Lawvere theory $L$, we associate the underlying set functor $ev_1 : \text{Mod}(L, \text{Set}) \rightarrow \text{Set}$ given by evaluation at 1. This is the semantics functor of Lawvere [16]. We say that the categories Mod($L, \text{Set}$) and Mod($L', \text{Set}$) of models are coherently equivalent if there is an equivalence of categories between them that respects the underlying set functor.

**Proposition 2.5** [16] Given Lawvere theories $L$ and $L'$, if the categories Mod($L, \text{Set}$) and Mod($L', \text{Set}$) are coherently equivalent, then the Lawvere theories $L$ and $L'$ are isomorphic in the category Law.

## 3 Abstract clones

Let $C$ be an arbitrary category and let $X$ be an object of $C$ for which all finite powers of $X$ also exist in $C$.

**Definition 3.1** Setting $O_X = \coprod_{n \geq 0} C(X^n, X)$, a subset $Cl \subseteq O_X$ is called a clone of operations over the object $X$ if it contains all projections $\pi_i : X^n \rightarrow X$ and is closed under composition, i.e., writing $Cl_n$ for those elements of $Cl$ that lie in $C(X^n, X)$, given $f \in Cl_n$ and $f_1, \ldots, f_n \in Cl_k$, the composite $f(f_1, \ldots, f_n)$ is in $Cl_k$.

This seems the most straightforward possible generalisation of a clone to abstract categories [14]. It is almost verbatim the definition in universal algebra [4] except that the composition under which the clones must be closed is written with the help of tuplings. In particular, putting $C = \text{Set}$, this is exactly the notion of a clone as studied in universal algebra [25,28,15].

Note that nullary operations are excluded from the definition, i.e., $C(X^0, X) \not\subseteq O_X$. This follows a convention in universal algebra, which has its advantages but also disadvantages. The notion is naturally connected with that of models of Lawvere theories, although it would be even more natural if one would allow nullary operations, which are less often encountered in universal algebraic literature, e.g.,
Proposition 3.2 \cite{14} A subset $\text{Cl} \subseteq O_X$ is a clone of operations over $X$ if and only if there exists a model $M : L \to C$ of a Lawvere theory $L$ in $C$ such that $M(1) = X$ and $\text{Cl} = \coprod_{n > 0} \{ M(f) \mid f \in L(n, 1) \}$.

The notion of clone is standard within universal algebra and has been so for many years. What is less standard is the abstraction from a base set $X$ to a notion corresponding exactly to that of a Lawvere theory, modulo the above caveat about nullary operations. In order to state a precise equivalence result, in the following, we shall allow abstract clones to have nullary operations.

Definition 3.3 cite \cite{4, 14} An abstract clone consists of
- for each $n \geq 0$, a set $\text{Cl}_n$, the elements of which are called $n$-ary operations
- for $1 \leq i \leq n$, an $n$-ary operation $\pi_i$ (allowing overloading of notation as, strictly speaking, we have a $\pi_i$ for each $n$)
- for each $n$-ary operation $g$ and $m$-ary operations $f_1, \ldots, f_n$, an $m$-ary operation $g(f_1, \ldots, f_n)$ such that, subject to the composites being defined,
  - $(h(g_1, \ldots, g_n))(f_1, \ldots, f_m) = h(g_1(f_1, \ldots, f_m), \ldots, g_n(f_1, \ldots, f_m))$
  - $\pi_i(f_1, \ldots, f_n) = f_i$ for all $1 \leq i \leq n$
  - $f(\pi_1, \ldots, \pi_n) = f$.

Proposition 3.4 To give an abstract clone is equivalent to giving a Lawvere theory.

Proof. Given an abstract clone $\text{Cl}$, put $L_{\text{Cl}}(n, m) = \text{Cl}_m^n$, with composition determined by the composition of operations in $\text{Cl}$, and with the identity on $n$ given by $(\pi_1, \ldots, \pi_n)$. Observe that $L_{\text{Cl}}$ forms a category with strictly associative finite products given by ordinal sum of natural numbers together with tuples of the $\pi_i$’s, and use the fact that $\aleph_0^{op}$ is the free category with finite products on $1$ to generate the functor $I : \aleph_0^{op} \to L_{\text{Cl}}$. Thus $L_{\text{Cl}}$ is a Lawvere theory.

For the converse, given a Lawvere theory $L$, put $(\text{Cl}_L)_n = L(n, 1)$, define the $\pi_i$’s using projections of $L$, and define composition of operations by the composition of $L$ together with the universal property of finite products. This data readily satisfies the axioms for an abstract clone.

The two constructions are routinely checked to be mutually inverse. \hfill $\Box$

4 Monads

Soon after Lawvere theories were defined, Linton showed that every Lawvere theory yields a monad on $\text{Set}$ \cite{17}. The construction extends to a fully faithful functor from $\text{Law}$ to the category $\text{Mnd}$ of monads on $\text{Set}$. The functor is not an equivalence of categories. So in this precise sense, a monad on $\text{Set}$ is a more general notion than that of Lawvere theory.
Linton also gave a partial converse. One can generalise the definition of Lawvere theory to allow for arities of arbitrary size, with a generalised theory no longer a small category or fully determined by one. The construction of a monad from a Lawvere theory then generalises to an equivalence of categories between the category of generalised Lawvere theories and $Mnd$. In [18], Linton accordingly generalised Lawvere’s treatment of semantics and algebraic structure.

In more detail, for any Lawvere theory $L$, let $U_L : Mod(L, C) \to C$ denote evaluation at 1, cf. Proposition 2.5. If $U_L$ has a left adjoint $F_L$, as it does whenever $C$ is locally presentable, it exhibits $Mod(L, C)$ as equivalent to the category $T_L$-Alg for the induced monad $T_L$ on $C$ [1]. Since $Set$ is locally finitely presentable, every Lawvere theory $L$ induces a monad $T_L$ on $Set$.

**Proposition 4.1** The monad $T_L$ may be described by the following colimit:

$$T_LX = \int_{n \in \mathbb{N}_0} L(n, 1) \times X^n$$

This colimit can be constructed by taking the coproduct

$$\coprod_{n \in \mathbb{N}_0} L(n, 1) \times X^n$$

then factoring by identifying elements determined by taking projections and diagonal maps of $\mathbb{N}_0^{op}$. So it is the set of all equivalence classes of terms generated by the operations of $L$, with variables among the elements of $X$, subject to the equalities determined by $L$.

**Proposition 4.2** The construction sending a Lawvere theory $L$ to the monad $T_L$ extends to a fully faithful functor from $Law$ to $Mnd$. Moreover, the comparison functor exhibits an equivalence between $Mod(L, Set)$ and $T_L$-Alg.

One can readily check that $T_L$ is always finitary. When the base category is $Set$, finitariness characterises the image of the construction, but that was an observation of a later time [12].

For a converse, observe that for any monad $T$ on $Set$, the Kleisli category $Kl(T)$ has all small coproducts and the canonical functor $I : Set \to Kl(T)$ preserves them: the canonical functor $I$ has a right adjoint and is identity-on-objects. Restricting $I$ to the full subcategory $\mathbb{N}_0$, we obtain (the opposite of) a Lawvere theory as in the diagram.

![Diagram](image)

It is straightforward to show the following.
Proposition 4.3 The construction sending a monad $T$ on $\text{Set}$ to the category $Kl(T)_{\aleph_0}^{\text{op}}$ determined by restricting $Kl(T)$ to the objects of $\aleph_0$ extends to a functor $L_\_ : \text{Mnd} \to \text{Law}$.

Given a Lawvere theory $L$, one can readily prove that $L(T_L)$ is isomorphic to $L$, but the converse is false: the only monads of the form $T_L$ are the finitary ones. Thus we have the following.

Theorem 4.4 The constructions sending $L$ to $T_L$ and that sending $T$ to $L_T$ exhibit Law as a full coreflective subcategory of $\text{Mnd}$, the category of monads on $\text{Set}$.

Because this is not an equivalence of categories, Linton generalised the definition of Lawvere theory to consider a locally small category $L$ with all small products, together with a strict product preserving identity-on-objects functor from the opposite of a skeleton of $\text{Set}$ to $L$. With this generalised notion of Lawvere theory, Linton showed that the construction of Proposition 4.2 extends, and in the corresponding version of Theorem 4.4, one has an equivalence of categories [17].

The different range of generality of the ideas of monads and Lawvere theories extends to the connection between (generalised) Lawvere theories and monads. One can consider monads on any category, while Lawvere theories correspond to (finitary) monads on $\text{Set}$. On the other hand, a monad on a category $C$ has algebras, i.e., models, only in $C$, while a Lawvere theory naturally has models in any category with products. So while monad maps between monads on $\text{Set}$ (see [1] for this notion of monad map) correspond directly to maps of Lawvere theories, there is nothing in the world of monads (at least nothing to which one has immediate access) corresponding to the functoriality of $\text{Mod}(L,C)$ in $C$.

5 Comonads and transition systems

The notions of monad and algebra dualise to those of comonad and coalgebra: that is easy.

Definition 5.1 A comonad on a category $C$ is a monad on $C^{\text{op}}$. Given a comonad $G$ on $C$, a $G$-coalgebra is a $G$-algebra for $G$ qua monad on $C^{\text{op}}$.

So the body of abstract theory initiated by Eilenberg and Moore for algebra immediately yields a body of abstract theory for coalgebra [5], and that has proved to be of considerable importance for computer science over the past twenty years or so [10]. A leading example is as follows.

Definition 5.2 Given a set $A$, a finitely branching $A$-labelled transition system is a pair $(S,t)$ consisting of a set $S$ and a function $t : S \to P_f(A \times S)$, where $P_f(X)$ is the set of finite subsets of a set $X$.

The transition function $t$ tells you, given a machine in state $\sigma$, to what states it might pass in one $A$-labelled move. The notion is fundamental to the theory of concurrency, for instance, playing a central role in $\text{CCS}$ [21].
The axiomatic situation is that in a category $C$ such as $\text{Set}$, one considers a pair $(X,x)$ consisting of an object $X$ of $C$ and a map $x : X \to H(X)$, where $H$ is an endofunctor on $C$. Such a pair is called an $H$-coalgebra [10]. In such axiomatic terms, a finitely branching $A$-labelled transition system is precisely a $P_f(A \times -)$-coalgebra.

The structure of the category of $P_f(A \times -)$-coalgebras can be used to characterise the critical notion of bisimulation in concurrency [21]. In fact, the body of theory of bisimulation can be defined and developed axiomatically in terms of $H$-coalgebras for an arbitrary endofunctor $H$ satisfying axiomatically defined conditions [10].

The most fundamental construct one makes in coalgebra is the construction of the cofree comonad $G(H)$ on $H$ if it exists. To give an $H$-coalgebra is equivalent to giving a $G(H)$-coalgebra, where the term coalgebra is overloaded, as $H$ is treated as an endofunctor while $G(H)$ is treated as a comonad.

**Theorem 5.3** [7] For any finitary endofunctor $H$ on any locally finitely presentable category $C$, a cofree comonad $G(H)$ on $H$ exists.

Although the statement of this theorem is dual to the statement of a theorem about algebras [11], the proof is different, not dual to the proof of the corresponding theorem for algebras. The reason for the difference is that $\text{Set}$ is a locally finitely presentable category, while $\text{Set}^{op}$ is not. Much of the category-theoretic effort involved with coalgebra revolves around handling that fact.

Although a cofree comonad on a finitary endofunctor on a locally finitely presentable category necessarily exists, it typically is not finitary.

**Example 5.4** [7] Given a set $A$, consider the endofunctor $H = A \times -$ on $\text{Set}$. The cofree comonad $G(H)$ sends a set $X$ to the set of infinite streams of elements of $A \times X$. So, for any countably infinite set $X$, the set $G(H)(X)$ contains a stream involving infinitely many different elements of $X$. Such a stream cannot be given by a finite subset of $X$, and so although $G(H)$ has a rank, that rank is necessarily greater than $\aleph_0$.

This phenomenon has been studied extensively and generalised by Worrell [29]. The key consequence of these issues for us is that dualising notions such as that of Lawvere theory is subtle, and we cannot expect to obtain as neat a relationship as that between Lawvere theories and monads.

### 6 Dualising Lawvere theories

One approach to dualising the body of theory of Lawvere theories is as follows.

**Definition 6.1** A comodel of a Lawvere theory $L$ in a category with finite coproducts $C$ is a model of $L$ in $C^{op}$.

Comodels in $C$ generate a category $\text{Comod}(L,C)$, with maps given by natural transformations, and a forgetful functor $U_L : \text{Comod}(L,C) \to C$ given by evaluation at 1, just as for models as in Section 2. Moreover, albeit with a different proof
Theorem 6.2 [26] For any Lawvere theory $L$ and locally finitely presentable category $C$, the forgetful functor $U_L$ has a right adjoint, generating a comonad $G_L$ on $C$, whereupon $\text{Comod}(L, C)$ is coherently equivalent to $G_L\text{-Coalg}$.

This dual of the theory of Lawvere theories appeared in [26], where it was used to model arrays. If $L_S$ be the (countable) Lawvere theory for global state, as described in [23,8], the category $\text{Comod}(L_S, \text{Set})$ is equivalent to the category of arrays [26]. Lawvere theories and their comodels have also been used, extending [23], to model operational semantics [24]. And they have been used, in terms of coclones, in [14]. But this dualisation of the theory of Lawvere theories does not include transition systems [10].

A second approach to dualising algebra is to start not with the notion of Lawvere theory but rather with that of equational theory, dualise it to a notion of co-equational theory, then look for an invariant, allowing us to mimic Proposition 2.5. There is an axiomatic account of the notions of operations, equations, algebras and monads in [13,27] that has been dualised in [7]. The idea is as follows.

Example 6.3 Consider three binary operations and no equations. This may be seen as a single binary operation with codomain 3, cf. the way in which one generates a Lawvere theory from an equational theory [13,27]. A model is a set $X$ together with a function of the form $X^2 \rightarrow X^3$. Dualise this to consider a function of the form $2^X \rightarrow 3^X$.

Axiomatically, in [7], one retained the natural numbers as arities, defined the notions of co-operation, co-equation and coalgebra in the spirit of Example 6.3, and proved that any family of co-operations and co-equations generates a comonad $G$ on $\text{Set}$ such that the category of $G$-coalgebras is isomorphic to the category of coalgebras for the co-operations and co-equations. Finitely branching $A$-labelled transition systems provided a leading example.

A third approach to a dual notion of Lawvere theory is generated by Linton’s work [17,18], as his generalised notion of Lawvere theory as discussed in Section 4 does not depend upon $\text{Set}$ being locally finitely presentable, and so generates a dual just as the notion of monad does. As for algebra, this has the drawback of not allowing duals of basic universal algebraic constructs such as sum.

These three approaches, together with a recent characterisation of Lawvere theories in [22], collectively suggest a tentative definition of dual Lawvere theory, which we give in Section 7.

7 Further Work: a proposal for dualising

There are several different but equivalent formulations of the notions of Lawvere theory and model. Starting with the usual definition of Lawvere theory as in Definition 2.2, to give a model of $L$ in $\text{Set}$ (Definition 2.3) is equivalent to giving a functor $M : L \rightarrow \text{Set}$ such that the composite $MI$ is of the form $\text{Set}(J- , X)$, where $J$
is the inclusion of $\aleph_0$ in $\text{Set}$: it follows from this definition that $M$ preserves finite products; the converse is given by putting $X = M1$.

With considerably more effort, one can prove that to give a Lawvere theory is equivalent to giving a small category $L$ together with an identity-on-objects functor $I : \aleph_0^{op} \to L$ such that $I$ preserves all finite limits in $\aleph_0^{op}$: this does not imply that $L$ has all finite limits, although it does follow that $L$ has all finite products [22].

So one possible notion of dual to investigate is as follows: a dual Lawvere theory is a small category $L$ together with an identity-on-objects functor $I : \aleph\to L$ that preserves all finite limits in $\aleph$. Note the dropping of $(−)^{op}$. A comodel in $\text{Set}$ is a functor $C : L \to \text{Set}$ for which $CI$ is of the form $\text{Set}(X, J−)$.

This seems to bear comparison with the definitions in [7] and seems to restrict Linton’s abstract work [17,18], which does not require size. It is not clear yet how it relates to comodels of Lawvere theories qua finite coproduct preserving functors. We propose this as further work.

References


Abstract

We propose a theoretical device for modeling the creation of new indiscernible semantic objects during program execution. The method fits well with the semantics of imperative, functional, and object-oriented languages and promotes equational reasoning about higher-order state.

Keywords: Operational semantics, functional programming, imperative programming, object-oriented programming, indiscernibles

1 Introduction

There are many situations in computing in which we want to create something new. Often we do not really care exactly what is created, as long as it has the right properties. For example, when allocating a new heap cell, we do not care exactly what its address in memory is, but only that we can store and retrieve data there. For that purpose, any heap cell is as good as any other. In object-oriented programming, when we create a new object of a class, we only care that it has the right fields and methods and is different from every other object of that class previously created. In the $\lambda$-calculus, when we $\alpha$-convert to rename a bound variable, we do not care what the new variable is as long as it is fresh.

As common as it is, the intuitive act of creating a new object out of nothing does not fit well with set-theoretic foundations. Such situations are commonly modeled as an allocation of one of a previously existing collection of equivalent candidates. One often sees statements such as, “Let $\text{Var}$ be a countable set of variables...,” or, “Let $\mathcal{L}$ be a countable set of heap cells...” The set is assumed to exist in advance.

---

1 Thanks to Kamal Aboul-Hosn, Robert Constable, Nate Foster, Jean-Baptiste Jeannin, Konstantinos Mamouras, Andrew Myers, Dirk Pattinson, Mark Reitblatt, Alexandra Silva, Aaron Stump, and all the members of the PLDG seminar at Cornell.

2 Email: kozen@cs.cornell.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
of its use and is assumed to be large enough that fresh elements can always be obtained when needed. Standard references on the semantics of objects also tend to treat object creation as allocation [1,23,25].

The difficulty here is that the candidates for allocation should be theoretically indiscernible, whereas real implementations must somehow make a deterministic choice. But to choose requires some way of distinguishing the chosen from the unchosen, thus the candidates cannot be indiscernible after all. Moreover, cardinality constraints often interfere with closure conditions on the language. For example, we only need a countable set of variables to represent an infinitary $\lambda$-term, but if all available variables already occur in the term, there would be none left over in case we needed a fresh one for $\alpha$-conversion. One could permute the variables to free one up, but that is awkward.

The issue is related to the philosophical problem of the identity of indiscernibles. Leibniz proposed that objects that have all the same properties must in fact be the same object. Although the subject of much debate in the philosophical literature [8,9,14], it is certainly desirable in programming language semantics, especially object-oriented programming, to allow the existence of distinct but indiscernible semantic objects. But it can also be the source of much confusion, as is well known to anyone who has ever tried to explain to introductory Java students why one should never compare strings with $==$.

The issue also arises in systems involving terms with variable binders, such as quantificational logic and the $\lambda$-calculus. We would like to treat bound variables as indiscernible for the purposes of $\alpha$-conversion and safe (capture-avoiding) substitution. Several devices for the generation of fresh variables have been proposed, both practical and theoretical, the earliest possibly being the gensym facility of LISP. Popular variable-avoiding alternative representations of $\lambda$-terms include de Bruijn indices and Stoy diagrams [5]. The NuPrl system [3,7] has a facility for generating nonces, or objects for which nothing can be tested except identity. The $\nu$-calculus of Pitts and Stark has a similar objective [6,26]. Nominal logic [11,12,27] is a logical system for reasoning about syntactic terms with binders.

In this paper we propose a device for creating new indiscernible objects in a semantic domain. Simply put, a semantic object is created by allocating a name for it. The object itself is defined to be the congruence class of all its names. A system such as nominal logic or the $\nu$-calculus can be used to handle the generation of names in the syntactic domain.

The idea can be illustrated with a very simple example. Consider a domain of semantic objects $D = \{a, b, c, \ldots\}$. Let $\phi$ be a first-order formula with free variables, say $x = y \land y \neq z$. According to the usual Tarskian definition of truth, we could interpret $\phi$ relative to a valuation $\sigma : \text{Var} \rightarrow D$, provided $\{x, y, z\} \subseteq \text{dom} \sigma$, and the judgment $\sigma \models \phi$ would have a well-defined truth value. For example, if $\sigma(x) = \sigma(y) = c$ and $\sigma(z) = a$, then $\sigma$ would satisfy $\phi$, along with many other other valuations over $D$.

However, suppose we did not specify the actual values of $x, y, z$, but only which variables represent the same values. Thus instead of $\sigma : \text{Var} \rightarrow D$, we would
have a set of equations $\alpha \subseteq \text{Var} \times \text{Var}$ specifying aliasing relationships between the variables. For the $\sigma$ above, $\alpha$ would consist of the single equation $x = y$. The free algebra generated by $\{x, y, z\}$ modulo the congruence induced by $x = y$ has two elements, namely the two congruence classes $\{x, y\}$ and $\{z\}$. Under the canonical interpretation

$$
\begin{align*}
x & \mapsto \{x, y\} \\
y & \mapsto \{x, y\} \\
z & \mapsto \{z\},
\end{align*}
$$

the formula $\phi$ is satisfied. The presentation $\alpha$ of the free algebra contains enough information to determine the truth of the formula; there is no need to represent the actual values.

The relation $\alpha$ is called an aliasing relation. It generates a congruence, that is, the smallest relation on terms that contains $\alpha$ and is reflexive, symmetric, transitive, and a congruence with respect to any operations defined on the elements. To represent the creation of a new object, we simply update $\alpha$ in a way that ensures that there is no aliasing between the variable instantiated with the new object and others of the same type currently represented in the state. We do not need to worry about how to select a new semantic object from a previously defined set or whether there are enough of them available; in essence, that responsibility is completely borne by the allocation of syntactic names. The advantage of this approach is that objects in the semantic domain can be generated ex nihilo and are truly indiscernible. An added benefit is that we can reason equationally with $\alpha$, and this appears to align well with popular approaches for reasoning about higher-order program state involving logical relations and bisimulation [2,6,10,13,16,20,26,28].

In this paper we develop this basic idea into an operational semantics for a higher-order functional programming language with imperative and object-oriented features. We give a set of operational rules that describe how the state, as represented by $\sigma$ and $\alpha$, should be updated as each atomic action is performed. The semantics is an extension of capsules [17,18,19]. We show how objects, nonces, references, arrays, and records fit into this framework. As an illustration, we show how to model safe substitution in the $\lambda$-calculus with nonces as variables and show that $\alpha$-conversion is an idempotent operation.

## 2 Capsules

Capsules [17,18,19] are a precursor to the system introduced here. Capsule semantics does not rely on heaps, stacks, or any other form of explicit memory, but only on names and bindings.

### 2.1 Syntax

A capsule is a pair $\langle e, \sigma \rangle$, where $e$ is a $\lambda$-term or constant and $\sigma$ is a partial function from variables to irreducible $\lambda$-terms or constants such that

(i) $\text{FV}(e) \subseteq \text{dom } \sigma$, and

(ii) if $x \in \text{dom } \sigma$, then $\text{FV}(\sigma(x)) \subseteq \text{dom } \sigma$, 

15
where $\text{FV}(e)$ is the set of free variables of $e$. Thus the free variables of a capsule are not really free; every variable in $\langle e, \sigma \rangle$ either occurs in the scope of a $\lambda$ or is bound by $\sigma$ to a constant or irreducible expression, which represents its value. A capsule represents a closed $\lambda$-coterm (infinitary $\lambda$-term). The closure conditions (i) and (ii) preclude catastrophic failure due to access of unbound variables. There may be circularities, which enables a representation of recursive functions.

Capsules may be $\alpha$-converted. Abstraction operators $\lambda x$ and the occurrences of $x$ bound to them may be renamed as usual. Variables in $\text{dom} \sigma$ may also be renamed along with all free occurrences. Capsules that are equivalent in this sense represent the same value.

Values are also preserved by garbage collection. A monomorphism of capsules $h : \langle d, \sigma \rangle \to \langle e, \tau \rangle$ is an injective map $h : \text{dom} \sigma \to \text{dom} \tau$ such that

- $\tau(h(x)) = h(\sigma(x))$ for all $x \in \text{dom} \sigma$, and
- $h(d) = e$,

where $h(e) = e[x/h(x)]$ (safe substitution). The set of monomorphic preimages of a given capsule contains an initial object that is unique up to a permutation of variables. This is the garbage-collected version of the capsule.

2.2 Semantics

Capsule evaluation semantics looks very much like the original evaluation semantics of LISP, with the added twist that a fresh variable is substituted for the parameter in function applications. The relevant small-step rule is

$$\langle (\lambda x. e) v, \sigma \rangle \to \langle e[y/v], \sigma[y/v] \rangle,$$

where $y$ is fresh. In the original evaluation semantics of LISP, the right-hand side is $\langle e, \sigma[x/v] \rangle$, which gives dynamic scoping. This simple change faithfully models $\beta$-reduction with safe substitution in the $\lambda$-calculus, providing static scoping without closures [17,18]. It also handles local variable declaration in recursive functions correctly.

Another evaluation rule of particular note is the assignment rule:

$$\langle x := v, \sigma \rangle \to \langle (), \sigma[x/v] \rangle$$

where $v$ is irreducible. The closure condition (i) of §2.1 ensures that $x$ is already bound in $\sigma$, and the assignment rebinds $x$ to $v$. Assignment is also used to create recursive functions via backpatching, also known as Landin’s knot, without the use of fixpoint combinators.

See [17,18,19] for further details and examples.

3 Syntax

In this section we define the syntax of our language. We use the same notation for rebinding and substitution. Given a function $\sigma$, we write $\sigma[x/v]$ for the function
such that $\sigma[x/v](y) = \sigma(y)$ for $y \neq x$ and $\sigma[x/v](x) = v$. Given an expression $e$, we write $e[x/d]$ for the expression $e$ with $d$ substituted for all free occurrences of $x$, renaming bound variables as necessary to avoid capture.

3.1 Types

Our type system distinguishes between constructive types and creative types. Constructive objects are constants or are constructed from other objects using constructors. They are represented directly in the state, bound by an environment $\sigma$ to a variable of the same type. Creative objects, on the other hand, do not exist in advance and are not built from constructors, but are created on the fly during program execution using new. They can be used to model objects (in the sense of object-oriented programming), references, arrays, records, and nonces. Creative objects have a weaker ontological status than constructive objects in that they have no direct representation in the state, but only indirect representation in the form of an aliasing relation $\alpha$.

The collection of all types is denoted $\text{Type}$. Let $\text{Var} = \{x, y, z, \ldots\}$ be an unlimited supply of variables. A type environment is a partial function $\Gamma : \text{Var} \rightarrow \text{Type}$ with finite domain $\text{dom} \Gamma$.

3.1.1 Constructive Types

Constructive types are built from type constructors. We have the function space constructor $\rightarrow$, products and coproducts, and coinductive types defined with finite systems of fixpoint equations.

Products and coproducts are of the form

$$\prod \Gamma = \prod_{x \in \text{dom} \Gamma} \Gamma(x) \quad \sum \Gamma = \sum_{x \in \text{dom} \Gamma} \Gamma(x)$$

where $\Gamma$ is a type environment. The corresponding projections and injections have type

$$\pi_x : \prod \Gamma \rightarrow \Gamma(x) \quad \iota_x : \Gamma(x) \rightarrow \sum \Gamma$$

for $x \in \text{dom} \Gamma$. The unit type 1 is the empty product, and the type of booleans is $2 = 1 + 1$.

Our product and coproduct types are not dependent types, as $\text{Var}$ is not a type. All function, product, and coproduct types are constructive.

3.1.2 Creative Types

In addition to constructive types, we have creative types $\mathcal{C}(\prod \Gamma)$, where $\Gamma : \text{Var} \rightarrow \text{Type}$ is a type environment. The type $\mathcal{C}(\prod \Gamma)$ represents a class of objects having fields named $x$ for $x \in \text{dom} \Gamma$ in the sense of object-oriented programming. Values of type $\mathcal{C}(\prod \Gamma)$ are creative objects. The field $x$ has type $\Gamma(x)$, which can be either constructive or creative.
3.1.3 Coinductive Types

Coinductive types are defined by finite systems of fixpoint equations. For example, the natural numbers are $\mathbb{N} = 1 + \mathbb{N}$, where $\mathbb{N}$ is a type variable. Formally, $\mathbb{N} = \sum \Gamma$, where $\Gamma(z) = 1$ and $\Gamma(s) = \mathbb{N}$. The number 3 would be represented by $\iota_s(\iota_s(\iota_s(\iota_z())))$. Since the type is coinductive, there is also an infinite element $\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_s(\iota_z())))$).

For another example, lists and streams over $\mathbb{N}$ are defined by $\text{intlist} = 1 + (\mathbb{N} \times \text{intlist})$ where intlist is a type variable. Formally, $\text{intlist} = \sum \Delta$, where

$$
\Delta(\text{nil}) = 1 \quad \Delta(\text{cons}) = \prod \Gamma \quad \Gamma(\text{hd}) = \mathbb{N} \quad \Gamma(\text{tl}) = \text{intlist}
$$

Then

$$
\sum \Delta = 1 + \prod \Gamma \quad \iota_{\text{nil}} : 1 \to \sum \Delta \quad \iota_{\text{cons}} : \prod \Gamma \to \sum \Delta \\
\prod \Gamma = \mathbb{N} \times \text{intlist} \quad \pi_{\text{hd}} : \prod \Gamma \to \mathbb{N} \quad \pi_{\text{tl}} : \prod \Gamma \to \text{intlist}
$$

Both constructive and creative types may appear in a coinductive type definition.

3.2 Expressions

Expressions $d, e, \ldots$ are defined inductively. Variables are expressions, as are typed projections $\pi_x$ and injections $\iota_x$. The unit object () is the null tuple of type $1$, and booleans $0 = \iota_{\text{false}}()$ and $1 = \iota_{\text{true}}()$ are of type $2$. We might also include other typed constants.

Compound expressions are formed with the following constructs, subject to typing constraints.

- $\lambda$-abstraction $\lambda x. e$
- application $(d \ e)$
- assignment $x := e$ or $d.x := e$
- tupling $(e_x \mid x \in \text{dom } \Gamma)$
- case analysis $[e_x \mid x \in \text{dom } \Gamma]$
- projection $e.x$
- object creation $\text{new } \Gamma(e)$
- identity test $d = e$

We also have defined expressions

- booleans $0, 1$
- composition $d \ ;
\ e$
- conditional $\text{if } b \text{ then } d \text{ else } e$
- while loop $\text{while } b \text{ do } e$

Let $w = \lambda x. \text{if } b \text{ then } e \text{ else } ()$ in $w()$.
• local definition \[ \text{let } x = d \text{ in } e \] \((\lambda x.e) d\)

• recursive definition \[ \text{let rec } x = d \text{ in } e \]

It is not necessary to worry about the capture of free occurrences of \(x\) in the composition, conditional, and while loop because the type of \(x\) is \(\bot\) in all cases.

The \(\bot\) in the definition of \texttt{let rec} is a special constant of the appropriate type designated for this purpose. This technique is known as Landin’s knot. The constant \(\bot\) is treated specially in the small-step operational semantics (see §4.6) in that a variable bound to it is considered irreducible, effectively allowing Landin’s knot to create self-referential objects.

The \texttt{let rec} construct is used to create recursive functions and values of coinductive types. It specifies a value that is the unique solution of the given equation in a certain final coalgebra. For recursive functions, this is a \(\lambda\)-coterm (infinitary \(\lambda\)-term), as in capsules (see §2). For coinductive datatypes, it is a multigraph realization as defined in [22]. In both cases, the infinite object is regular and has a finite representation. For example, the type of integer lists and streams was defined in §3.1.3. An element of this type is the infinite stream of alternating 0s and 1s, which can be defined by

\[ \text{let rec } x = t_{\text{cons}}(0, t_{\text{cons}}(1, x)) \text{ in } e \]

The mutually recursive definition

\[ \text{let rec } x_1 = d_1 \text{ and } \ldots \text{ and } x_n = d_n \text{ in } e \]

can be coded using the single-variable form of \texttt{let rec} with products and projections or with nested \texttt{let recs}.

The case analysis construct \([e_x \mid x \in \text{dom } \Gamma]\) corresponds to a \texttt{case} or \texttt{match} statement of functional languages. It is used to extract the elements of a coproduct based on their types. For example, the \texttt{map} function that maps a given function \(f : \mathbb{N} \to \mathbb{N}\) over a given integer list would be defined by \texttt{let rec} to satisfy the equation

\[ \text{map } = \lambda(f : \mathbb{N} \to \mathbb{N}). [t_{\text{nil}}, \lambda x. t_{\text{cons}}(f(\pi_{\text{hd}} x), \text{map } f (\pi_{\text{tl}} x))] \]

This would be written more conventionally as

\[ \text{map } (f : \mathbb{N} \to \mathbb{N}) (\ell : \text{intlist}) : \text{intlist} = \]

\begin{align*}
\text{case } \ell \text{ of } \\
\mid t_{\text{nil}} \to t_{\text{nil}} \\
\mid t_{\text{cons}} x \to t_{\text{cons}}(f(\pi_{\text{hd}} x), \text{map } f (\pi_{\text{tl}} x))
\end{align*}

\subsection{3.2.1 Typing Rules}

Let \(\Delta : \text{Var} \rightarrow \text{Type}\) be a type environment. We write \(\Delta \vdash e : \alpha\) if the type \(\alpha\) can be derived for the expression \(e\) by the typing rules of Fig. 1. The constructs \texttt{let rec } x = d \text{ in } e \text{ and } \texttt{let } x = d \text{ in } e \text{ are typed as } (\lambda x.e) d.\]

If \(\Delta \vdash e : \alpha\) for some \(\alpha\), we say that \(e\) is \(\Delta\text{-well-typed}, or just well-typed if
Kozen

\[ \Delta \vdash c : \text{type}(c), \ c \in \text{Const} \quad \Delta \vdash x : \Delta(x), \ x \in \text{dom} \Delta \]
\[ \Delta \vdash x : \alpha \quad \Delta \vdash e : \beta \quad \Delta \vdash d : \alpha \rightarrow \beta \quad \Delta \vdash e : \alpha \quad \Delta \vdash (d \ e) : \beta \]
\[ \Delta \vdash x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x := e : \alpha \]
\[ \Delta \vdash b : 2 \quad \Delta \vdash d : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash b : 2 \quad \Delta \vdash e : 1 \quad \Delta \vdash \text{while } b \text{ do } e : \alpha \]
\[ \Delta \vdash d : 1 \quad \Delta \vdash e : \alpha \quad \Delta \vdash d : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash d : e : 2 \]
\[ \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Gamma \vdash x : \beta \quad \Delta \vdash e.x : \beta \quad \Delta \vdash \text{new } \Gamma^e(e) : \mathcal{C}(\prod \Gamma) \]
\[ \Delta \vdash e_x : \Gamma(x), \ x \in \text{dom} \Gamma \quad \Delta \vdash (e_x \mid x \in \text{dom} \Gamma) : \prod \Gamma \quad \Delta \vdash e_x : \Gamma(x) \rightarrow \beta, \ x \in \text{dom} \Gamma \quad \Delta \vdash [e_x \mid x \in \text{dom} \Gamma] : \sum \Gamma \rightarrow \beta \]

Fig. 1. Typing Rules

\[ \Delta \vdash e : \text{type}(e), \ e \in \text{Const} \]
\[ \Delta \vdash x : \Delta(x), \ x \in \text{dom} \Delta \]
\[ \Delta \vdash d : \alpha \rightarrow \beta \quad \Delta \vdash e : \alpha \quad \Delta \vdash (d \ e) : \beta \]
\[ \Delta \vdash x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x := e : \alpha \]
\[ \Delta \vdash b : 2 \quad \Delta \vdash d : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash b : 2 \quad \Delta \vdash e : 1 \quad \Delta \vdash \text{while } b \text{ do } e : \alpha \]
\[ \Delta \vdash d : 1 \quad \Delta \vdash e : \alpha \quad \Delta \vdash d : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash d : e : 2 \]
\[ \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Gamma \vdash x : \beta \quad \Delta \vdash e.x : \beta \quad \Delta \vdash \text{new } \Gamma^e(e) : \mathcal{C}(\prod \Gamma) \]
\[ \Delta \vdash e_x : \Gamma(x), \ x \in \text{dom} \Gamma \quad \Delta \vdash (e_x \mid x \in \text{dom} \Gamma) : \prod \Gamma \quad \Delta \vdash e_x : \Gamma(x) \rightarrow \beta, \ x \in \text{dom} \Gamma \quad \Delta \vdash [e_x \mid x \in \text{dom} \Gamma] : \sum \Gamma \rightarrow \beta \]

\[ \Delta \vdash e : \text{type}(e), \ e \in \text{Const} \quad \Delta \vdash x : \Delta(x), \ x \in \text{dom} \Delta \]
\[ \Delta \vdash d : \alpha \rightarrow \beta \quad \Delta \vdash e : \alpha \quad \Delta \vdash (d \ e) : \beta \]
\[ \Delta \vdash x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash d.x := e : \alpha \]
\[ \Delta \vdash b : 2 \quad \Delta \vdash d : \alpha \quad \Delta \vdash e : \alpha \quad \Delta \vdash b : 2 \quad \Delta \vdash e : 1 \quad \Delta \vdash \text{while } b \text{ do } e : \alpha \]
\[ \Delta \vdash d : 1 \quad \Delta \vdash e : \alpha \quad \Delta \vdash d : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Delta \vdash d : e : 2 \]
\[ \Delta \vdash e : \mathcal{C}(\prod \Gamma) \quad \Gamma \vdash x : \beta \quad \Delta \vdash e.x : \beta \quad \Delta \vdash \text{new } \Gamma^e(e) : \mathcal{C}(\prod \Gamma) \]
\[ \Delta \vdash e_x : \Gamma(x), \ x \in \text{dom} \Gamma \quad \Delta \vdash (e_x \mid x \in \text{dom} \Gamma) : \prod \Gamma \quad \Delta \vdash e_x : \Gamma(x) \rightarrow \beta, \ x \in \text{dom} \Gamma \quad \Delta \vdash [e_x \mid x \in \text{dom} \Gamma] : \sum \Gamma \rightarrow \beta \]
• constants,
• \(\lambda\)-abstractions,
• creative assignable expressions, i.e. elements of \(\mathsf{CA}_\Delta\),
• expressions \(v_x \mid x \in \mathsf{dom} \Gamma\), where all \(v_x\) are irreducible,
• expressions \([v_x \mid x \in \mathsf{dom} \Gamma]\), where all \(v_x\) are irreducible,
• expressions \(\iota_x v\), where \(v\) is irreducible.

In addition, for the purpose of Landin’s knot, a constructive variable \(x\) is considered irreducible if \(\sigma(x) = \bot\); see §4.6. Constructive assignable expressions are not irreducible in general. The set of constructive irreducible expressions is denoted \(\mathsf{NVal}_\Delta\).

### 3.3 Aliasing Relations

Let \(\alpha \subseteq \mathsf{CA}_\Delta \times \mathsf{CA}_\Delta\) be a set of pairs of creative assignable expressions such that if \((u, v) \in \alpha\), then \(\Delta \vdash u : \mathsf{C}(\prod \Gamma)\) iff \(\Delta \vdash v : \mathsf{C}(\prod \Gamma)\). The set \(\alpha\) is called an aliasing relation. It represents a set of well-typed equations between creative assignable expressions.

The congruence generated by \(\alpha\) is the smallest binary relation on \(\mathsf{A}_\Delta\) containing \(\alpha\) and closed under the rules of Fig. 2. There is some redundancy among the premises of the last rule (congruence), as one can show inductively that if \(\alpha \vdash u = v\), then \(u\) and \(v\) have the same type. Note that \(u\) and \(v\) can be constructive, even though the elements of \(\alpha\) are all creative. The congruence class of \(v \in \mathsf{A}_\Delta\) is denoted \([v]_\alpha\).

We can form the free algebra \(\mathsf{A}_\Delta / \alpha = \{[u]_\alpha \mid u \in \mathsf{A}_\Delta\}\). It is an algebra in the sense that the projections \(x\), regarded as unary operations, are well-defined on congruence classes; that is, if \([u]_\alpha = [v]_\alpha\), then by congruence, \([u.x]_\alpha = [v.x]_\alpha\) whenever \(u.x\) is well-typed, so it makes sense to define \([u]_\alpha.x = [u.x]_\alpha\). Intuitively, if \(\Delta \vdash u = v\), then \(u\) and \(v\) are aliases for the same object, so the values of the fields \(u.x\) and \(v.x\) should also be the same.

As mentioned, the set \(\mathsf{A}_\Delta\) can be infinite in general, but the computational rules will maintain the invariant that \(\mathsf{A}_\Delta / \alpha\) is finite. One can regard \(\mathsf{A}_\Delta / \alpha\) as a finite graph with nodes \([u]_\alpha\) and labeled edges \([u]_\alpha \xrightarrow{\iota} [u.x]_\alpha\).

We denote by \(\mathsf{CA}_\Delta / \alpha\) and \(\mathsf{NA}_\Delta / \alpha\) the sets of creative and constructive elements
of $A_\Delta/\alpha$, respectively; that is, the sets
\[ CA_\Delta/\alpha = \{ [u]_\alpha | u \in CA_\Delta \} \]
\[ NA_\Delta/\alpha = \{ [u]_\alpha | u \in NA_\Delta \} = A_\Delta/\alpha - CA_\Delta/\alpha. \]

### 3.4 Equational Reasoning

The congruence generated by $\alpha$ extends inductively to effect-free constructors with the obvious syntactic congruence rule for each constructor. For example, for products and injections,
\[
\alpha \vdash d = e \\
\alpha \vdash \lambda x.d = \lambda x.e.
\]
The only nonobvious rule is $\lambda$-abstraction, in which we must treat the bound variable specially.
\[
\alpha \vdash d[x/z] = e[x/z], \ z \text{ fresh} \\
\alpha \vdash \lambda x.d = \lambda x.e.
\]

There is no sound congruence rule for assignment := or new, as these constructs have side effects. For example, it would never be the case that new $C(\prod \Gamma) = \text{new } C(\prod \Gamma)$, because evaluation of each side creates an object.

These rules, along with $\alpha$-conversion, renaming by a permutation, and garbage collection (§3.6) can be used in equational reasoning on program states.

### 3.5 Program States

A program state is represented by a quadruple $(e, \Delta, \sigma, \alpha)$, where:
- $\Delta : \text{Var} \rightarrow \text{Type}$ is a type environment
- $\alpha \subseteq CA_\Delta \times CA_\Delta$ is a $\Delta$-well-typed equational presentation
- $\sigma : NA_\Delta/\alpha \rightarrow NVal_\Delta$ is a $\Delta$-well-typed valuation
- $e$ is a $\Delta$-well-typed expression

The domain of $\sigma$ is officially $NA_\Delta/\alpha$, but we will often abuse notation and write $\sigma(u)$ for $\sigma([u]_\alpha)$.

The component $e$ is the expression to be evaluated. The typing of expressions is determined by $\Delta$. The components $\sigma$ and $\alpha$ comprise an environment that determines the interpretation of free variables. Conditions (i) and (ii) of §2.1 for capsules are implied by the facts that $e$ and $\sigma([u]_\alpha)$ are well-typed and the domain of $\sigma$ is $NA_\Delta/\alpha$. Formally, $\sigma$ is also defined on bound variables, but that is unnecessary and could be relaxed.

The set of states is a nominal set over the set of names $\text{Var}$ in the sense of nominal logic [11,12,27].
3.6 Garbage Collection

Our notions of \(\alpha\)-conversion and garbage collection are based on capsules (see §2.1) with appropriate modifications to account for the aliasing relation \(\alpha\). As with capsules, values are preserved.

Any variable declared in \(\Delta\) may be \(\alpha\)-converted. If a fresh variable is needed for \(\alpha\)-conversion, its type is first declared in \(\Delta\). Renaming variables in some type environment \(\Gamma\) used in the declaration of a product or sum does not constitute \(\alpha\)-conversion and does not result in an equivalent state.

As with capsules, garbage collection is defined in terms of monomorphisms. A monomorphism

\[
h : \langle e, \Delta, \sigma, \alpha \rangle \to \langle e', \Delta', \sigma', \alpha' \rangle
\]

is an injective map \(h : \text{dom} \Delta \to \text{dom} \Delta'\) such that

(i) \(h\) is type-preserving, that is, \(\Delta(x) = \Delta'(h(x))\);

(ii) modulo \(\alpha\) and \(\alpha'\), \(h\) is an algebra monomorphism \(A_{\Delta/\alpha} \to A_{\Delta'/\alpha'}\);

(iii) \(\sigma'(\[h(x)\]_{\alpha'}) = h(\sigma([x]_{\alpha}))\) for all \([x]_{\alpha} \in \text{dom} \sigma\); and

(iv) \(e' = h(e)\),

where \(h(e) = e[x/h(x)]\). Like capsules, every state has an initial monomorphic preimage, which is its garbage-collected version and which is unique up to a permutation of variables and variation in the presentation \(\alpha\) of \(A_{\Delta/\alpha}\).

However, unlike capsules, we cannot collect garbage simply by removing variables inaccessible from \(e\), because some of them may be needed in the equational presentation \(\alpha\) of \(A_{\Delta/\alpha}\). Removing the equations containing them could cause property (ii) to be violated; \(h\) would be a homomorphism but not a monomorphism. To ensure (ii), we show that \(A_{\Delta/\alpha}\) has a canonical presentation in which \(\alpha\) is minimal and the pairs are of a certain form. This form will also be used in the semantics of assignment (§4.5).

**Lemma 3.1** Given an aliasing relation \(\alpha\) on \(A_{\Delta}\), there is a set of variables \(X\), an extension \(\Delta'\) of \(\Delta\) with domain \(X \cup \text{dom} \Delta\), and an aliasing relation \(\alpha'\) on \(A_{\Delta'}\) with the following properties:

(i) \(A_{\Delta/\alpha}\) and \(A_{\Delta'/\alpha'}\) are isomorphic;

(ii) all pairs in \(\alpha'\) are of the form \((x, z)\) or \((x, y, z)\), where \(x, z \in X\);

(iii) every congruence class in \(CA_{\Delta'/\alpha'}\) contains exactly one variable of \(X\).

Moreover, \(\Delta'\) and \(\alpha'\) can be computed from \(\Delta\) and \(\alpha\) in time \(O(n \alpha(n))\), where \(\alpha(n)\) is the inverse of Ackermann’s function.

**Proof.** Let \(A\) be the set of subterms of terms appearing in \(\alpha\). Form the congruence closure \(\hat{\alpha}\) of \(\alpha\) on \(A\). The congruence closure is the smallest relation on \(A\) that contains \(\alpha\) and is closed under the rules of Fig. 2 applied only to terms in \(A\). It is shown in [21] that for \(s, t \in A\), \(\alpha \vdash s = t\) iff \((s, t) \in \hat{\alpha}\); that is, one need not go outside of \(A\) to prove congruence between two terms in \(A\).
One can form the congruence closure for a signature involving only unary func-
tions in time $O(n \alpha(n))$. The algorithm is essentially the same as that used to
minimize deterministic finite-state automata [4,15,24]. By “forming the congru-
ence closure,” we do not mean computing the relation $\hat{\alpha}$ itself—that would take
too long to write down—but rather forming the congruence classes and associating
each element of $A$ with its respective congruence class so that we can subsequently
determine whether $(s,t) \in \hat{\alpha}$ (that is, $\alpha \vdash s = t$) for $s,t \in A$ in constant time.

Let $X$ be a set of variables such that each creative $\alpha$-congruence class contains
exactly one element of $X$. If $[u]_\alpha$ does not contain a variable, we can add a fresh
variable $x$ and the equation $(x,u)$ to $\alpha$, although this step is not strictly necessary,
as our operational semantics maintains the invariant that every creative congruence
class contains a variable. Let $\Delta'$ be $\Delta$ extended as necessary with the appropriate
typings for $x \in X$.

Now let

$$\alpha' = \hat{\alpha} \cap \{(x,z) \mid x \in X, z \in \text{Var} \} \cup \{(x.y,z) \mid x, z \in X\}.$$

The set $\alpha'$ has the following properties:

- For each $u \in A$, there is exactly one $x \in X$ such that $\alpha' \vdash x = u$.
- For each $x \in X$ and $y \in \text{dom} \ \Gamma$, where $\Delta'(x) = \mathcal{C}(\prod \Gamma)$, there is exactly one $z \in X$
such that $(x.y,z) \in \alpha'$.

It follows that $\alpha$ and $\alpha'$ generate the same congruence closure $\hat{\alpha}$, thus $A_{\Delta/\alpha}$ and
$A_{\Delta'/\alpha'}$ are isomorphic.

Now we can collect garbage by forming the reduced presentation as described in
Lemma 3.1 and removing inaccessible variables from $\Delta$, $\sigma$, and $\alpha$, where a variable
is accessible if it is in the smallest set of variables containing the variables of $e$ and
closed under the following operations:

- If $x$ is accessible, $(x,z) \in \alpha$ or $(x.y,z) \in \alpha$, and $z \in X$, then $z$ is accessible;
- if $x$ is accessible and $z$ occurs in $\sigma([x]_\alpha)$ or $\sigma([x.y]_\alpha)$, then $z$ is accessible.

The monomorphism $h$ is defined on the subalgebra of $A_{\Delta/\alpha}$ generated by the ac-
cessible variables.

\section{Operational Semantics}

The operational semantics of the language is defined by the small-step rules given be-
low. In addition, there are context rules that define a standard shallow applicative-
order evaluation strategy (leftmost innermost, call-by-value) and left-to-right eval-
uation of tuples and expressions $e.x$. 
4.1 Function Application

Our rule for function application is adapted from the rule for capsules (see §2.2):

\[
\langle (\lambda x. e) \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e[y/x], \Delta[y/\Delta(x)], \sigma', \alpha' \rangle,
\]

where \( y \) is fresh and

\[
(\sigma', \alpha') = \begin{cases} 
(\sigma[y]_\alpha/v, \alpha) & \text{if } \Delta(x) \text{ is constructive} \\
(\sigma, \alpha \cup \{(y, v)\}) & \text{if } \Delta(x) \text{ is creative.}
\end{cases}
\]

As with capsules, a fresh variable \( y \) is conjured and given the same type as \( x \), resulting in a new global type environment \( \Delta[y/\Delta(x)] \). If the type is constructive, \( \sigma \) is updated with the value \( v \), and \( \alpha \) is unchanged. If the type is creative, \( \sigma \) is unchanged, but \( \alpha \) is updated with the new alias \( (y, v) \).

4.2 Creation

The following rule creates a new creative object:

\[
\langle \text{new } \Gamma(v), \Delta, \sigma, \alpha \rangle \rightarrow \langle y, \Delta[y/\xi(\prod \Gamma)], \sigma', \alpha' \rangle,
\]

where \( y \) is fresh and

\[
\alpha' = \alpha \cup \{(y.x, v_x) \mid x \in \text{dom } \Gamma, \Gamma(x) \text{ creative}\}
\]

\[
\sigma' = \sigma[y.x]_{\alpha'/v_x} \mid x \in \text{dom } \Gamma, \Gamma(x) \text{ constructive}
\]

The object is represented by a fresh variable \( y \), which is added to the domain of \( \Delta \) with the appropriate creative type. The value \( v \) is a tuple supplying the initial values of the fields. The entities \( \alpha \) and \( \sigma \) are updated to assign the fields of the new object their initial values.

4.3 Assignment to Constructive Expressions

Assignment for constructive types is essentially the same as for capsules. For \( u \in \text{NA}_\Delta \) and \( v \) irreducible of the same constructive type,

\[
\langle u := v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \Omega, \Delta, \sigma[u]_\alpha/v, \alpha \rangle.
\]

Here \( \Delta \) does not need to be updated, because \( u \) is already well-typed.

4.4 Assignment to Creative Variables

Before we can define the semantics of assignments to creative assignable expressions, we need to lay some groundwork. The issue is that assignment to a creative expression may change the free algebra presented by \( \alpha \) if the expression to be assigned is involved in the presentation.
First we consider the case of an assignment \( x := v \) to a creative variable \( x \in \text{dom } \Delta \). Let \( \Delta' = \Delta[z/\Delta(x)] \), where \( z \notin \text{dom } \Delta \). Define \( g : \text{dom } \Delta \rightarrow \mathbb{A}_{\Delta'} \) by

\[
g(x) = z \quad \quad g(u) = u, \quad u \in \text{dom } \Delta - \{x\}.
\]

Define \( h : \text{dom } \Delta' \rightarrow \mathbb{A}_{\Delta} \) by

\[
h(z) = x \quad \quad h(x) = v \quad \quad h(u) = u, \quad u \in \text{dom } \Delta' - \{z,x\}.
\]

Extend \( h \) uniquely to a homomorphism \( h : \mathbb{A}_{\Delta'} \rightarrow \mathbb{A}_{\Delta} \) by inductively defining \( h(u.y) = h(u).y \) for \( y \in \text{dom } \Gamma \), where \( \Delta' \vdash u : c(\prod \Gamma) \). Likewise, extend \( g \) uniquely to a homomorphism \( g : \mathbb{A}_{\Delta} \rightarrow \mathbb{A}_{\Delta'} \).

Define a new set of axioms on \( \mathbb{A}_{\Delta'} \):

\[
\alpha' = \{(x,g(v))\} \cup \{(g(s),g(t)) \mid (s,t) \in \alpha\}.
\]

**Lemma 4.1** Modulo \( \alpha \) and \( \alpha' \), the homomorphisms \( g \) and \( h \) are well defined and are inverses, thus the quotient algebras \( \mathbb{A}_{\Delta}/\alpha \) and \( \mathbb{A}_{\Delta'}/\alpha' \) are isomorphic.

**Proof.** First we observe that \( h \) is a left inverse of \( g \):

\[
h(g(x)) = h(z) = x \quad \quad h(g(u)) = h(u) = u, \quad u \in \text{dom } \Delta - \{x\}.
\]

Moreover, \( g \) is a left inverse of \( h \) modulo \( \alpha' \):

\[
g(h(z)) = g(x) = z \quad \quad g(h(u)) = g(u) = u, \quad u \in \text{dom } \Delta' - \{z,x\},
\]

and since \( (x,g(v)) \) is an axiom of \( \alpha' \) and \( g(h(x)) = g(v) \),

\[
\alpha' \vdash g(h(x)) = x.
\]

Since \( h \) is a left inverse of \( g \) on generators \( \text{dom } \Delta \) of \( \mathbb{A}_{\Delta} \), and since \( h \) and \( g \) are homomorphisms, \( h \) is a left inverse of \( g \) on all elements of \( \mathbb{A}_{\Delta} \). Similarly, \( g \) is a left inverse of \( h \) modulo \( \alpha' \) on all elements of \( \mathbb{A}_{\Delta'} \).

\[
\alpha' \vdash s = t \quad \Rightarrow \quad \alpha \vdash h(s) = h(t),
\]

thus \( h \) is well-defined modulo \( \alpha \) and \( \alpha' \). By general considerations of universal algebra, it suffices to show that (4) holds for the axioms \( (s,t) \in \alpha' \). For the axiom \( (x,g(v)) \), we wish to show \( \alpha \vdash h(x) = h(g(v)) \). This follows immediately from the facts that \( h(x) = v \) and \( h \) is a left-inverse of \( g \). For the axioms \( (g(s),g(t)) \) for \( (s,t) \in \alpha \), we have \( \alpha \vdash s = t \), and since \( h \) is a left-inverse of \( g \), \( \alpha \vdash h(g(s)) = h(g(t)) \).

We have shown that \( h \) composed with the canonical map \( \mathbb{A}_{\Delta} \rightarrow \mathbb{A}_{\Delta}/\alpha \) is well-defined on \( \alpha' \)-congruence classes, therefore reduces to a homomorphism

\[
h' : \mathbb{A}_{\Delta'}/\alpha' \rightarrow \mathbb{A}_{\Delta}/\alpha.
\]
Likewise, one can show that $\alpha \vdash s = t$ implies $\alpha' \vdash g(s) = g(t)$ by the same argument, thus $g$ reduces to a homomorphism

$$g' : A_\Delta/\alpha \rightarrow A_{\Delta'}/\alpha'.$$

Finally, since $h$ is a left inverse of $g$ and $g$ is a left inverse of $h$ modulo $\alpha'$, it follows that $g'$ and $h'$ are inverses, thus constitute an isomorphism between $A_\Delta/\alpha$ and $A_{\Delta'}/\alpha'$.

Lemma 4.1 allows us to define the semantics of assignment to a creative variable:

$$\langle x := v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \sigma([x]_\alpha), \Delta[z/\Delta(x)], \sigma', \alpha' \rangle,$$

where $z$ is fresh, $\sigma' = \sigma \circ h'$, and $\alpha'$ and $h'$ are as defined in (3) and (5), respectively.

4.5 Assignment to Creative Fields

Now we treat the case of an assignment $u.y := v$, where both $u$ and $u.y$ are creative. As before, we need to ensure that $u.y$ is not involved in the axiomatization $\alpha$ of the quotient structure so that the assignment will have no unintended consequences. However, unlike the previous case, if $\alpha \vdash u = v$, then assigning to $u.y$ also assigns the same value to $v.y$ due to the aliasing. Moreover, there is not necessarily an isomorphism between the two structures.

We first put $\alpha$ into the reduced form of Lemma 3.1. Let $X$ be the set defined in that lemma. We can find variables $x, z, w \in X$ such that $\alpha \vdash u = x$, $\alpha \vdash v = w$, and $(x.y, z) \in \alpha$. We then define

$$\langle u.y := v, \Delta, \sigma, \alpha \rangle \rightarrow \langle 1, \Delta, \sigma', \alpha' \rangle$$

where $\alpha' = (\alpha - \{(x.y, z)\}) \cup \{(x.y, w)\}$ and $\sigma'$ is defined to agree with $\sigma$ on all constructive expressions of the form $r$ or $r.s$, where $r$ is a variable. By the form of the reduced presentation, this determines $\sigma'$ completely.

4.6 Other Small-Step Rules

(i) $\langle x, \Delta, \sigma, \alpha \rangle \rightarrow \langle \sigma([x]_\alpha), \Delta, \sigma, \alpha \rangle$, $\sigma([x]_\alpha) \neq \bot$, $x$ constructive

(ii) $\langle x.y, \Delta, \sigma, \alpha \rangle \rightarrow \langle \sigma([x.y]_\alpha), \Delta, \sigma, \alpha \rangle$, $x.y$ constructive

(iii) $\langle u = v, \Delta, \sigma, \alpha \rangle \rightarrow \langle 0, \Delta, \sigma, \alpha \rangle$, $\alpha \not\vdash u = v$

(iv) $\langle u = v, \Delta, \sigma, \alpha \rangle \rightarrow \langle 0, \Delta, \sigma, \alpha \rangle$, $\alpha \not\vdash u = v$

(v) $\langle \pi_y(v_x | x \in \text{dom } \Gamma), \Delta, \sigma, \alpha \rangle \rightarrow \langle v_y, \Delta, \sigma, \alpha \rangle$

(vi) $\langle [g_x | x \in \text{dom } \Gamma](v_y), \Delta, \sigma, \alpha \rangle \rightarrow \langle (g_y v), \Delta, \sigma, \alpha \rangle$

Defined rules are

(vii) $\langle \sigma : e, \Delta, \sigma, \alpha \rangle \rightarrow \langle e, \Delta, \sigma, \alpha \rangle$

(viii) $\langle \text{if } 1 \text{ then } d \text{ else } e, \Delta, \sigma, \alpha \rangle \rightarrow \langle d, \Delta, \sigma, \alpha \rangle$
(ix) \( \langle \text{if } 0 \text{ then } d \text{ else } e, \Delta, \sigma, \alpha \rangle \rightarrow \langle e, \Delta, \sigma, \alpha \rangle \)

\[ \langle \text{while } b \text{ do } e, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{if } b \text{ then } (e ; \text{while } b \text{ do } e) \text{ else } (), \Delta, \sigma, \alpha \rangle \]

The proviso “\( \sigma([x]_\alpha) \not= \bot \)” in (i) effectively makes \( x \) irreducible when this property holds. This is to allow Landin’s knot to form self-referential terms. Recall that \texttt{let rec } x = d in e abbreviates \texttt{let } x = \bot \texttt{ in } (x := d) ; e. The object \( \bot \) is meant for this purpose only, and is not meant to be visible as the final value of a computation. In a real implementation one would prevent \( \bot \) from becoming visible by imposing syntactic guardedness conditions on the form of \( d \), as done for example in OCaml, or by raising a runtime error if the value of \( \bot \) is ever required in the evaluation of \( d \).

5 Applications

Nonces

A nonce is a creative object of type \( \mathcal{C}(\mathbb{1}) \). These are objects with no fields. They correspond to the objects created by the \texttt{new} operator in the \( \nu \)-calculus \cite{6,26}. They can be used as unique identifiers. We illustrate the use of nonces as variables in §6.

Records

A record with fields of type \( \Gamma \) is an object of type \( \mathcal{C}(\prod \Gamma) \). Note that this is different from \( \prod \Gamma \). The difference is that if \( x_1 = y_1 \) and \( x_2 = y_2 \), then \( \langle x_1, x_2 \rangle = \langle y_1, y_2 \rangle \), whereas there can be distinct creative objects \( x \) and \( y \) with \( x.1 = y.1 \) and \( x.2 = y.2 \).

References

A reference is a record with a single field named \( ! \). The type of the reference is \( \mathcal{C}(\prod \Gamma) \), where \( \text{dom} \Gamma = \{!\} \), and \( \Gamma(!) \) is the type of the datum. For example, an integer reference, which would be represented by the type \texttt{int ref} in OCaml, would have \( \Gamma(!) = \mathbb{Z} \). The following OCaml expressions would translate to our language as indicated:

<table>
<thead>
<tr>
<th>OCaml</th>
<th>our language</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{let } x = \texttt{ref 3 in} ...</td>
<td>\texttt{let } x = \texttt{new } \Gamma(3) \texttt{ in} ...</td>
</tr>
<tr>
<td>( !x )</td>
<td>( x! )</td>
</tr>
<tr>
<td>( x := 4 )</td>
<td>( x! := 4 )</td>
</tr>
</tbody>
</table>

Arrays

An integer array of length \( m \) is a record with fields \( \{0,1,\ldots,m-1,\text{length}\} \). This would have type \( \mathcal{C}(\prod \Gamma) \), where \( \text{dom} \Gamma = \{0,1,\ldots,m-1,\text{length}\} \), \( \Gamma(i) = \mathbb{Z} \) for \( 0 \leq i \leq m-1 \), and \( \Gamma(\text{length}) = \mathbb{N} \). The following Java expressions would translate to our language as indicated:
<table>
<thead>
<tr>
<th>Java</th>
<th>our language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int[] x = new int[3];</code></td>
<td><code>let x = new Γ(0, ..., 0, 3) in ...</code></td>
</tr>
<tr>
<td><code>x.length</code></td>
<td><code>x.length</code></td>
</tr>
<tr>
<td><code>x[0]</code></td>
<td><code>x.0</code></td>
</tr>
<tr>
<td><code>x[2] = x[3];</code></td>
<td><code>x.2 := x.3</code></td>
</tr>
</tbody>
</table>

**Objects**

A creative type $\mathcal{C}(\prod \Gamma)$ can be regarded as a class with fields whose types are specified by $\Gamma$. If $\textit{self}$ is a variable of type $\Delta(\textit{self}) = \mathcal{C}(\prod \Gamma)$, then other fields $x \in \text{dom} \Gamma$ of the object can be accessed from within the object as $\textit{self}.x$. To create a new object of the class, we would say

\[
\text{let rec } \textit{self} = \text{new } \Gamma(\textit{v}) \text{ in } \textit{self}
\]  

(7)

The value of this expression is a new object in which the references to $\textit{self}$ in $v$ have been backpatched via Landin’s knot to refer to the object just created. If we like, we can even have $\textit{self} \in \text{dom} \Gamma$ with $\Gamma(\textit{self}) = \mathcal{C}(\prod \Gamma)$. The component of $v$ corresponding to $\textit{self}$ should be $\textit{self}$. In order to have $\Gamma(\textit{self}) = \mathcal{C}(\prod \Gamma)$, the type must be coinductive.

Note that the use of Landin’s knot is essential here. The traditional approach involving fixpoint combinators does not work, as the $\text{new}$ operator is not referentially transparent.

Here is an example to demonstrate (7). Let $\text{dom} \Gamma = \{\textit{self}, f, n\}$ with

\[
\Gamma(\textit{self}) = \mathcal{C}(\prod \Gamma) \quad \Gamma(f) = \mathbb{N} \rightarrow \emptyset \quad \Gamma(n) = \mathbb{N}.
\]

Let us evaluate (7) with $v = (\textit{self}, \lambda y. (\textit{self}.n := y), 3)$. Substituting the definitions of $\text{let rec}$ and $\text{let}$, we have

\[
\begin{align*}
\text{let rec } \textit{self} & = \text{new } \Gamma(\textit{v}) \text{ in } \textit{self} \\
& = \text{let } \textit{self} = \bot \text{ in } (\textit{self} := \text{new } \Gamma(\textit{v}) \text{ in } \lambda y. (\textit{self}.n := y), 3) ; \textit{self} \\
& = (\lambda \textit{self}. (\textit{self} := \text{new } \Gamma(\textit{v}) \text{ in } \lambda y. (\textit{self}.n := y), 3) ; \textit{self}) \perp.
\end{align*}
\]

Evaluating this expression in a state with $\Delta$, $\sigma$, and $\alpha$ would result in the state

\[
\langle (x := \text{new } \Gamma(\textit{x}, \lambda y. (\textit{x}.n := y), 3)) ; x, \Delta', \sigma, \alpha' \rangle
\]

where $x$ is fresh, $\Delta' = \Delta[x/\mathcal{C}(\prod \Gamma)]$, and $\alpha' = \alpha \cup \{(x, \bot)\}$. One more step of the evaluation would yield

\[
\langle (x := v) ; x, \Delta'', \sigma', \alpha'' \rangle
\]
where \( v \) is fresh and
\[
\Delta'' = \Delta'[\sigma / \mathcal{C}(\prod \Gamma)] \quad \alpha'' = \alpha' \cup \{(v, \text{self}, x)\}.
\]

Now performing the assignment leaves the expression \( x \) and changes the aliasing relation to \((\alpha'' - \{(x, \bot)\}) \cup \{(x, v)\} \). Applying Lemma 3.1 with \( x \in X \) and collecting garbage, we are left with the final state
\[
\langle x, \Delta[x/\mathcal{C}(\prod \Gamma)], \sigma[x.f/\lambda y.(x.n := y)][x.n/3], \alpha \cup \{(x, \text{self}, x)\} \rangle.
\]

To accommodate nominal classes in the sense of [25, §19.3], one could augment the new construct to allow new \( C(e) \), where \( C = \mathcal{C}(\prod \Gamma) \) is a class declaration, although we have not done so here.

6 Substitution and \( \alpha \)-Conversion

In this section we demonstrate how syntactic equivalence of computational states gives rise to indiscernability in the semantic domain. We show how to model \( \lambda \)-terms semantically as elements of a coinductive datatype in which variables are nonces. In the semantic domain, \( \alpha \)-conversion is an idempotent operation; that is, \( \alpha \)-converting twice is the same as \( \alpha \)-converting once.

A \( \lambda \)-term is either a \( \lambda \)-variable, an application, or an abstraction. An application is a pair of \( \lambda \)-terms, an abstraction consists of a \( \lambda \)-variable (the parameter) and a \( \lambda \)-term (the body), and \( \lambda \)-variables are nonces. We can thus model \( \lambda \)-terms with the coinductive type
\[
\begin{align*}
\lambda \text{Term} &= \lambda \text{Var} + \lambda \text{App} + \lambda \text{Abs} \quad \text{\( \lambda \)-coterms} \\
\lambda \text{App} &= \lambda \text{Term} \times \lambda \text{Term} \quad \text{applications} \\
\lambda \text{Abs} &= \lambda \text{Var} \times \lambda \text{Term} \quad \text{abstractions} \\
\lambda \text{Var} &= \mathcal{C}(1) \quad \text{\( \lambda \)-variables}
\end{align*}
\]

The type also contains \( \lambda \)-coterms (infinitary \( \lambda \)-terms), although they do not figure in our development.

The free variables of a \( \lambda \)-term are defined inductively by
\[
\text{FV}(y) = \{y\} \quad \text{FV}(t_1 t_2) = \text{FV}(t_1) \cup \text{FV}(t_2) \quad \text{FV}(\lambda y.t_0) = \text{FV}(t_0) - \{y\}
\]

They can be computed (for well-founded terms) by the following recursive program:

\[
\text{let rec isFreeIn}(x : \lambda \text{Var}) (t : \lambda \text{Term}) : 2 =
\begin{align*}
\text{case } t \text{ of} \\
| t_0 y \rightarrow y = x & \text{false} \\
| t_1 (t_1, t_2) \rightarrow \text{isFreeIn } x \ t_1 \lor \text{isFreeIn } x \ t_2 & \text{false} \\
| t_2 (y, t_0) \rightarrow y \neq x \land \text{isFreeIn } x \ t_0 & \text{true}
\end{align*}
\]

Likewise, safe (capture-avoiding) substitution is defined as a fixpoint of a system of equations. The result of substituting \( e \) for \( x \) in \( t \) is denoted \( t[x/e] \) and is defined
inductively by

\[
y[x/e] = \begin{cases} 
  e & \text{if } y = x \\
  y & \text{if } y \neq x
\end{cases}
\]

\[
(t_1 \cdot t_2)[x/e] = (t_1[x/e] \cdot t_2[x/e])
\]

\[
(\lambda y.t_0)[x/e] = \begin{cases} 
  \lambda y.t_0 & \text{if } y = x \\
  \lambda y.(t_0[x/e]) & \text{if } y \neq x \text{ and } y \not\in \text{FV}(e) \\
  \lambda z.(t_0[y/z][x/e]) & \text{otherwise, where } z \not\in \{x\} \cup \text{FV}(t_0) \cup \text{FV}(e)
\end{cases}
\]

In the last rule, to satisfy the proviso \( z \not\in \{x\} \cup \text{FV}(t_0) \cup \text{FV}(e) \), it suffices to take \( z \) fresh. This leads to the following recursive program:

```ocaml
let rec subst (t : \lambda Term) (x : \lambda Var) (e : \lambda Term) : \lambda Term =
  case t of 
  | \iota_0 y \to \text{if } y = x \text{ then } e \text{ else } t 
  | t_1 (t_1, t_2) \to t_1 (\text{subst} t_1 x e, \text{subst} t_2 x e) 
  | t_2 (y, t_0) \to \begin{cases} 
    t & \text{if } y = x \\
    \text{else if } \neg(\text{isFreeIn} y e) \text{ then } t_2 (y, \text{subst} t_0 x e) \\
    \text{else let } z = \text{new } \lambda \text{Var in } t_2 (z, \text{subst} (\text{subst} t_0 y (\iota_0 z)) x e)
  \end{cases}
```

If \( e \) is a variable \( w \), this simplifies to

\[
y[x/w] = \begin{cases} 
  w & \text{if } y = x \\
  y & \text{if } y \neq x
\end{cases}
\]

\[
(t_1 \cdot t_2)[x/w] = (t_1[x/w]) (t_2[x/w])
\]

\[
(\lambda y.t_0)[x/w] = \begin{cases} 
  \lambda y.t_0 & \text{if } y = x \\
  \lambda y.(t_0[x/w]) & \text{if } y \neq x \text{ and } y \neq w \\
  \lambda z.(t_0[y/z][x/w]) & \text{if } w = y \neq x, \text{ where } z \not\in \{x, w\} \cup \text{FV}(t_0).
\end{cases}
\]

```ocaml
let rec subst' (t : \lambda Term) (x : \lambda Var) (w : \lambda Var) : \lambda Term =
  case t of 
  | \iota_0 y \to \text{if } y = x \text{ then } \iota_0 w \text{ else } t 
  | t_1 (t_1, t_2) \to t_1 (\text{subst}' t_1 x w, \text{subst}' t_2 x w) 
  | t_2 (y, t_0) \to \begin{cases} 
    t & \text{if } y = x \\
    \text{else if } y \neq w \text{ then } t_2 (y, \text{subst}' t_0 x w) \\
    \text{else let } z = \text{new } \lambda \text{Var in } t_2 (z, \text{subst}' (\text{subst}' t_0 y z) x w)
  \end{cases}
```

**Lemma 6.1** Modulo \( \alpha \)-equivalence and garbage collection, the following big-step
rules are sound:

\[ \alpha \vdash x = y \]
\[ (\text{subst'}(\iota_0 y) x v, \Delta, \sigma, \alpha) \rightarrow (\iota_0 y v, \Delta, \sigma, \alpha) \] (8)
\[ (\text{subst'}(\iota_0 y) x v, \Delta, \sigma, \alpha) \rightarrow (\iota_0 y, \Delta, \sigma, \alpha) \] (9)
\[ (\text{subst'} e_0 x v, \Delta, \sigma, \alpha) \rightarrow (\iota_0 e_0, \Delta, \sigma, \alpha) \rightarrow (\iota_0 e_1 x v, \Delta, \sigma, \alpha) \rightarrow (\iota_1 (e_0 v), \Delta, \sigma, \alpha) \] (10)
\[ (\text{subst'}(\iota_1 (e_0 e_1)) x v, \Delta, \sigma, \alpha) \rightarrow (\iota_1 (v_0, v_1), \Delta, \sigma, \alpha) \] (11)
\[ \alpha \vdash x = y \]
\[ (\text{subst'}(\iota_2 (y t) x v, \Delta, \sigma, \alpha) \rightarrow (\iota_2 (y t) x v, \Delta, \sigma, \alpha) \rightarrow (\iota_2 (y u), \Delta, \sigma, \alpha) \] (12)

**Proof.** We start with rule (8). Suppose \( \alpha \vdash y = x \). Let

\[
\Delta' = \Delta[t'/\lambda \text{Term}][x'/\lambda \text{Var}][v'/\lambda \text{Var}] \\
\alpha' = \alpha \cup \{(x,x'), (v,v')\} \\
\alpha'' = \alpha' \cup \{(y,y')\}
\]
\[ (13) \]

where \( t', x', v', y' \) are fresh. We will first give the steps of the derivation, then give a brief justification of each step afterwards.

\[ (\text{subst'}(\iota_0 y) x v, \Delta, \sigma, \alpha) \]
\[ \rightarrow (\lambda x w. [\lambda y. \text{if } y = x \text{ then } \iota_0 w \text{ else } \iota_0 y, \ldots ] t)(\iota_0 y) x v, \Delta, \sigma, \alpha) \] (14)
\[ \rightarrow (\lambda y. \text{if } y = x' \text{ then } \iota_0 v' \text{ else } \iota_0 y, \ldots ] t', \Delta', \sigma', \alpha') \] (15)
\[ \rightarrow (\text{if } y' = x' \text{ then } \iota_0 v' \text{ else } \iota_0 y', \Delta'', \sigma', \alpha'') \] (16)
\[ \rightarrow (\iota_0 v', \Delta'', \sigma', \alpha'') \] (17)
\[ = (\iota_0 v, \Delta'', \sigma', \alpha'') \] (18)
\[ = (\iota_0 v, \Delta, \sigma, \alpha). \] (19)

For (14), we have just replaced \( \text{subst'} \) with its definition. This is just an application of small-step rule (i) of §4.6.

We obtain (15) from (14) by doing three successive function applications as defined in §4.1. The first allocates a fresh constructive variable \( t' \) of type \( \lambda \text{Term} \), substitutes it for \( t \) in the body of the function, and binds it to the argument \( \iota_0 y \) in \( \sigma \) to get \( \sigma' \). The second and last allocate fresh creative variables \( x' \) and \( v' \) of type \( \lambda \text{Var} \), substitute them for \( x \) and \( v \), respectively, in the body of the function, and equate them to the arguments \( x \) and \( v \), respectively, thereby extending \( \alpha \) to \( \alpha' \). The new type environment is \( \Delta' \).

We obtain (16) from (15) by rule (vi) of §4.6, the small-step rule for the case statement. After lookup of \( t' \), its value \( \iota_0 y \) is analyzed and the function corresponding to index 0 in the tuple (the one shown) is dispatched. That function is
applied to \(y\), which causes a fresh creative variable \(y'\) of type \(\lambda \text{Var}\) to be allocated, substituted for \(y\) in the body, and equated with the argument \(y\) in \(\alpha'\) to get \(\alpha''\). The new type environment is \(\Delta''\).

For (17), since \(\alpha \vdash y = x\) by assumption, we have \(\alpha'' \vdash y' = x'\), therefore the conditional test succeeds, resulting in the value \(\iota_0 v'\). Since \(\alpha'' \vdash v = v'\), (17) is equivalent to (18). Finally, (19) is obtained by garbage collection, observing that \(t', x', y'\), and \(y'\) are no longer accessible from \(\iota_0 v\).

The proof of rule (10), let

\[
\Delta' = \Delta[t'/\lambda \text{Term}]x'/\lambda \text{Var}]v'/\lambda \text{Var}] \quad \Delta'' = \Delta'[y'/\lambda \text{App}]
\]

\[
\alpha' = \alpha \cup \{(x, x'), (v, v')\}
\]

\[
\sigma' = \sigma[t'/\iota_1 (e_0, e_1)] \quad \sigma'' = \sigma'[y'/(e_0, e_1)],
\]

where \(t', x', y'\) are fresh. By reasoning similar to the above, we have

\[
\langle \text{subst}' (\iota_1 (e_0, e_1)) \rangle x v, \Delta, \sigma, \alpha \\
\rightarrow [\ldots, \lambda \text{y.}\iota_1 (\text{subst}' (\pi_0 y)) x' v', \text{subst}' (\pi_1 y) x' v'), \ldots] t', \Delta', \sigma', \alpha'
\rightarrow \iota_1 (\text{subst}' (\pi_0 y') x' v'), \text{subst}' (\pi_1 y') x' v'), \Delta'', \sigma'', \alpha'
\]

where \(\alpha' \vdash x = x'\) and \(\alpha' \vdash v = v'\). Now evaluating \(\pi_0 y'\) gives \(e_0\), and by the left-hand premise of (10), \(\text{subst}' e_0 x v\) reduces to \(v_0\) in context. Similarly, by the right-hand premise, \(\text{subst}' (\pi_0 y') x v\) reduces to \(v_1\) in context. This leaves us with

\[
\langle \iota_1 (v_0, v_1), \Delta'', \sigma', \alpha'' \rangle = \langle \iota_1 (v_0, v_1), \Delta, \sigma, \alpha \rangle,
\]

where the right-hand side is obtained from the left by garbage collection.

Finally, for rule (12), let

\[
\Delta' = \Delta[t'/\lambda \text{Term}]x'/\lambda \text{Var}]v'/\lambda \text{Var}]y'/\lambda \text{Abs}
\]

\[
\alpha' = \alpha \cup \{(x, x'), (v, v')\} \quad \sigma' = \sigma[t'/\iota_2 (y, t)]y'/(y, t)],
\]

where \(t', x', y'\) are fresh. As above, we have

\[
\langle \text{subst}' (\iota_2 (y, t)) \rangle x v, \Delta, \sigma, \alpha \\
\rightarrow \iota_2 (\pi_0 y', \text{subst}' (\pi_1 y') x' v'), \Delta', \sigma', \alpha'
\rightarrow \iota_2 (y, \text{subst}' t x' v'), \Delta', \sigma', \alpha'
\rightarrow \iota_2 (y, \text{subst}' t x v), \Delta, \sigma, \alpha
\rightarrow \iota_2 (y, u), \Delta, \sigma, \alpha
\]

with (20) from the fact that \(\alpha' \vdash x = x'\), \(\alpha' \vdash v = v'\), and garbage collection, and (21) from the premise of (12) applied in context. \(\square\)
Lemma 6.2 Let $\Delta \vdash e : \lambda \text{Term}$ and $x, u, v \in \text{dom } \Delta$. Assume that $\alpha \not\vdash y = u$ and $\alpha \not\vdash y = v$ for $y = x$ or any $y$ occurring in $e$. The states

$$\langle \text{subst}' (\text{subst}' e \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \quad \langle \text{subst}' e \, x \, v, \Delta, \sigma, \alpha \rangle$$

reduce to equivalent states modulo $\alpha$-equivalence and garbage collection.

Proof. For the case $e = \iota_0 \, y$ and $\alpha \vdash y = x$, by rule (8) both states reduce to $\langle \iota_0 \, v, \Delta, \sigma, \alpha \rangle$:

$$\langle \text{subst}' (\text{subst}' (\iota_0 \, y) \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' (\iota_0 \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_0 \, v, \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_0 \, y) \, x \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_0 \, v, \Delta, \sigma, \alpha \rangle.$$ 

If $\alpha \not\vdash y = x$, by rule (9) both states reduce to $\langle \iota_0 \, y, \Delta, \sigma, \alpha \rangle$:

$$\langle \text{subst}' (\text{subst}' (\iota_0 \, y) \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' (\iota_0 \, y) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_0 \, y, \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_0 \, y) \, x \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_0 \, y, \Delta, \sigma, \alpha \rangle.$$ 

For the case $\iota_1 (e_0, e_1)$, we have

$$\langle \text{subst}' e_0 \, x \, u, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_0', \Delta, \sigma, \alpha \rangle \quad \langle \text{subst}' e_0' \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_0'', \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' e_1 \, x \, u, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_1', \Delta, \sigma, \alpha \rangle \quad \langle \text{subst}' e_1' \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_1'', \Delta, \sigma, \alpha \rangle,$$

thus

$$\langle \text{subst}' (\text{subst}' e_0 \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' e_0' \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_0'', \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\text{subst}' e_1 \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' e_1' \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_1'', \Delta, \sigma, \alpha \rangle.$$ 

By the induction hypothesis,

$$\langle \text{subst}' e_0 \, x \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_0'', \Delta, \sigma, \alpha \rangle \quad \langle \text{subst}' e_1 \, x \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle e_1'', \Delta, \sigma, \alpha \rangle.$$ 

By rule (10),

$$\langle \text{subst}' (\iota_1 (e_0, e_1)) \, x \, u, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_1 (e_0', e_1'), \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_1 (e_0', e_1')) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_1 (e_0'', e_1''), \Delta, \sigma, \alpha \rangle,$$

therefore

$$\langle \text{subst}' (\text{subst}' (\iota_1 (e_0, e_1)) \, x \, u) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' (\iota_1 (e_0', e_1')) \, u \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_1 (e_0'', e_1''), \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_1 (e_0', e_1')) \, x \, v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_1 (e_0'', e_1''), \Delta, \sigma, \alpha \rangle.$$
For the case $\iota_2(y, t)$, if $\alpha \vdash y = x$, by rule (11) and the fact that $\alpha \not\vdash u = w$ for any $w$ occurring in $t$, we have

$$\langle \text{subst}' (\text{subst}' (\iota_2(y, t)) x u) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' (\iota_2(y, t)) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_2(y, t), \Delta, \sigma, \alpha \rangle.$$ 

If $\alpha \not\vdash y = x$, we have $\alpha \not\vdash y = u$ and $\alpha \not\vdash y = v$ by the assumptions of the lemma, and

$$\langle \text{subst}' t x u, \Delta, \sigma, \alpha \rangle \rightarrow \langle t', \Delta, \sigma, \alpha \rangle \quad \langle \text{subst}' t' u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle t'', \Delta, \sigma, \alpha \rangle,$$

thus

$$\langle \text{subst}' (\text{subst}' t x u) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' t' u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle t'', \Delta, \sigma, \alpha \rangle.$$ 

By the induction hypothesis,

$$\langle \text{subst}' t x v, \Delta, \sigma, \alpha \rangle \rightarrow \langle t'', \Delta, \sigma, \alpha \rangle.$$ 

By rule (12),

$$\langle \text{subst}' (\iota_2(y, t)) x u, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_2(y, t'), \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_2(y, t')) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_2(y, t''), \Delta, \sigma, \alpha \rangle$$

$$\langle \text{subst}' (\iota_2(y, t)) x v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_2(y, t''), \Delta, \sigma, \alpha \rangle,$$

therefore

$$\langle \text{subst}' (\text{subst}' (\iota_2(y, t)) x u) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{subst}' (\iota_2(y, t')) u v, \Delta, \sigma, \alpha \rangle \rightarrow \langle \iota_2(y, t''), \Delta, \sigma, \alpha \rangle.$$ 

To $\alpha$-convert, we would map $\lambda x.e$ to $\lambda z.(e[x/z])$, where $z \notin \text{FV}(e) - \{x\}$. We choose $z \notin \text{FV}(e) - \{x\}$ to avoid the capture of a free occurrence of $z$ in $e$ as a result of the renaming. Usually we would simply choose a fresh $z$.

In our language, this would be implemented by a function

$$\text{alpha} : \lambda \text{Abs} \rightarrow \lambda \text{Abs}$$

$$\text{alpha} = \lambda t. \text{let } z = \text{new } \lambda \text{Var in } (z, \text{subst}' (\pi_1 t) (\pi_0 t) z),$$

or more informally,

$$\text{alpha}(x, e) = \text{let } z = \text{new } \lambda \text{Var in } (z, \text{subst}' e x z).$$
The following theorem illustrates how syntactic equivalence of computational states gives rise to indiscernability in the semantic domain. It states that $\alpha$-conversion is an idempotent operation; that is, performing it twice gives the same result as performing it once.

**Theorem 6.3** Modulo $\alpha$-equivalence and garbage collection,

$$\text{alpha}(\text{alpha}(x,e)) = \text{alpha}(x,e).$$

**Proof.** In the evaluation of $\langle \text{alpha}(x,e), \Delta, \sigma, \alpha \rangle$, let $t,u,v$ be fresh variables and let

$$\Delta' = \Delta[t/\lambda\text{Abs}], \quad \sigma' = \sigma[t/(x,e)], \quad \alpha' = \alpha \cup \{(u,v)\}.$$

Suppose

$$\langle \text{subst'} e x u, \Delta[u/\lambda\text{Var}], \sigma, \alpha \rangle \rightarrow \langle e', \Delta[u/\lambda\text{Var}], \sigma, \alpha \rangle.$$

The evaluation yields the following sequence of states:

$$\langle \text{alpha}(x,e), \Delta, \sigma, \alpha \rangle \rightarrow \langle \text{let} \ z = \text{new} \ \lambda\text{Var} \ in \ \langle z, \ \text{subst'} (\pi_1 t) (\pi_0 t) z \rangle, \ \Delta', \ \sigma', \ \alpha \rangle$$
$$\rightarrow \langle (\lambda z. (z, \ \text{subst'} (\pi_1 t) (\pi_0 t) z)) v, \ \Delta'[v/\lambda\text{Var}], \ \sigma', \ \alpha \rangle$$
$$\rightarrow \langle (u, \ \text{subst'} (\pi_1 t) (\pi_0 t) u), \ \Delta'[v/\lambda\text{Var}][u/\lambda\text{Var}], \ \sigma', \ \alpha' \rangle$$
$$\rightarrow \langle (u, \ \text{subst'} e x u), \ \Delta'[v/\lambda\text{Var}][u/\lambda\text{Var}], \ \sigma', \ \alpha' \rangle$$
$$= \langle (u, \ \text{subst'} e x u), \ \Delta[u/\lambda\text{Var}], \ \sigma, \ \alpha \rangle \quad (22)$$
$$\rightarrow \langle (u, e'), \ \Delta[u/\lambda\text{Var}], \ \sigma, \ \alpha \rangle.$$

Step (22) is by garbage collection. Using this,

$$\langle \text{alpha}(\text{alpha}(x,e)), \Delta, \sigma, \alpha \rangle$$
$$= \langle \text{alpha}(u,e'), \ \Delta[u/\lambda\text{Var}], \ \sigma, \ \alpha \rangle$$
$$\rightarrow \langle (v, \ \text{subst'} e x u), \ \Delta[u/\lambda\text{Var}][v/\lambda\text{Var}], \ \sigma, \ \alpha \rangle \quad (23)$$
$$= \langle (v, \ \text{subst'} e x u), \ \Delta[u/\lambda\text{Var}][v/\lambda\text{Var}], \ \sigma, \ \alpha \rangle \quad (24)$$
$$= \langle (v, \ \text{subst'} e x u), \ \Delta[v/\lambda\text{Var}], \ \sigma, \ \alpha \rangle \quad (25)$$
$$= \langle (u, \ \text{subst'} e x u), \ \Delta[u/\lambda\text{Var}], \ \sigma, \ \alpha \rangle \quad (26)$$
$$\rightarrow \langle (u, e'), \ \Delta[u/\lambda\text{Var}], \ \sigma, \ \alpha \rangle.$$

Step (23) is by the same argument as (22). Step (24) is by Lemma 6.2. Steps (25) and (26) are by garbage collection and renaming of a creative variable. □

7 Conclusion and Future Directions

We have shown how to model the creation of new indiscernible semantic objects during program execution and how to incorporate this device in a higher-order
Kozen

functional language with imperative and object-oriented features. Modeling indiscernables is desirable because it abstracts away from properties needed to allocate objects from a preexisting set, thus allowing the representation of semantic objects at a higher level of abstraction.

We have also shown that the explicit aliasing relation $\alpha$ and the congruence closure algorithm are useful techniques in equational reasoning about higher-order state. An interesting question for further study is the extent to which they can be assimilated in equational deduction systems based on logical relations and bisimulation [2,6,10,13,16,20,26,28].

References


[9] Boolos, G., To be is to be a value of a variable (or to be some values of some variables), J. Philosophy 81 (1984), pp. 430–450.


37


Observationally-induced Effects in Cartesian Closed Categories

Ingo Battenfeld

Fakultät für Informatik
TU Dortmund
Dortmund, Germany

Abstract

Alex Simpson has suggested an observationally-induced approach towards obtaining monads for computational effects in denotational semantics. The underlying idea of this approach is to use a single observation algebra as computational prototype and to obtain a computational monad as a free algebra construction derived from this prototype. Recently, it has been shown that free observationally-induced algebras exist in the category of continuous maps between topological spaces for arbitrary pre-chosen computational prototypes.

In this work we transfer these results to cartesian closed categories. In particular, we show that, provided the category under consideration satisfies suitable completeness conditions, it supports a free observationally-induced algebra construction for arbitrary computational prototypes. We also show that the free algebras are obtained as certain subobjects of double exponentials involving the computational prototype as result type. Finally, we apply these results to show that in topological domain theory an observationally-induced lower powerspace construction over a QCB-space $X$ is given by the space of nonempty closed subsets of $X$ topologised suitably.

Keywords: denotational semantics, computational effects, powerdomains, topological domain theory

1 Introduction

Computational effects occur whenever a computer program interacts with its physical environment. The most common examples for computational effects are interactive user in- and output, reading from and writing to memory cells, nondeterministic and probabilistic features of the underlying system. Thus, when giving a denotational semantics to a programming language, no matter how tight this language follows an abstract calculus, one has to provide means for modelling computational effects. The first general denotational treatment of computational effects has been proposed by Moggi in the form of computational monads [7], which had significant impact on the subsequent design of functional programming languages, perhaps...

1 Email: battenfeld@ls10.cs.tu-dortmund.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
most prominently in Haskell. This work has since been refined by Plotkin and Power’s proposal to obtain computational monads as free algebra constructions for suitable algebraic theories [8]. The principal idea of this approach is that computational effects are triggered by algebraic operations, which therefore have to be modelled on the corresponding datatypes.

In recent years, Alex Simpson has furthermore refined the Plotkin and Power approach by suggesting to derive a computational monad from the algebraic properties of a single observation algebra instead of using a free algebra construction for a general equational algebraic theory [10]. The observation algebra that induces this construction can be thought of as computational prototype, giving rise to computational observations. Therefore, one may call this construction an observationally-induced approach to computational effects. Simpson and Schröder have used the observationally-induced approach to recover a probabilistic powerspace construction as space of probability valuations in \( \text{Top} \), the category of continuous maps between topological spaces [10,11]. In recent work, Schröder and the author have shown that in \( \text{Top} \), free observationally-induced algebras exist for arbitrary pre-chosen computational prototypes, and how to derive and characterise lower and upper powerspace constructions [2].

In the work at hand we investigate free observationally-induced algebra constructions in cartesian closed categories. Cartesian closure is essential when it comes to model function types in denotational semantics. Hence this work aims at answering some naturally arising questions concerning existence and characterisation of observationally-induced effect types in such frameworks. It turns out that under suitable completeness requirements on the category under investigation, free observationally-induced algebras exist for arbitrary computational prototypes. A particularly interesting result is that the free algebras are obtained as subobjects of a double exponential, reminiscent of continuation types where the result type of the continuation is the respective computational prototype. Furthermore, we show that under this construction one obtains a lower powerspace construction in the category \( \text{QCB} \), of continuous maps between quotients of countably-based spaces, which is one of the central categories in topological domain theory [3]. This observationally-induced lower powerspace over a \( \text{QCB} \)-space \( X \) is given by the set of nonempty closed subsets of \( X \) topologised suitably. This shows that in topological domain theory one can recover a classical intrinsic lower powerspace construction via observationally-induced free algebras, which is known not to be the case for the upper or probabilistic powerspaces.

The paper is organised as follows. In section 2, we recall the basic definitions of observationally-induced free algebras. In section 3, we show that under suitable completeness conditions free observationally-induced algebras exist in cartesian closed categories for arbitrary pre-chosen computational prototypes. Section 4 applies these results to investigate a lower powerspace construction in topological domain theory. The paper is concluded in section 5.
2 Observationally-induced algebras

Throughout the paper, we assume $\Sigma$ to be an algebraic signature, i.e. a set of operation symbols $\{\sigma \in \Sigma\}$ each of which has an arity $|\sigma| \in \mathbb{N}$. Then in any category $C$ with finite products we can apply the usual definition of $\Sigma$-algebras and $\Sigma$-homomorphisms and get a corresponding category $C_{\Sigma}$. This holds in particular for cartesian closed categories and for the purpose of this paper we assume all our categories to be such.

Remark 2.1 For simplicity we assume the arity of an operation to be a natural number, i.e. we consider finitary signatures. However, the results below can easily be adjusted to more general settings, such as arities given by cardinals up to some fixed $\kappa$, objects of the ambient category or parameterised operations as in [1] (which correspond to operations with co-arities). The important point is that a category of $\Sigma$-algebras in $C$ has to be well-defined.

In section 4.6 of [1], it has been shown that:

Lemma 2.2 If $(A, \{\sigma_A\})$ is a $C_{\Sigma}$-algebra and $X$ any object of $C$, then there exists a canonical $\Sigma$-algebra structure on the exponential $A^X$, giving the $C_{\Sigma}$-algebra $(A^X, \{\sigma_{A^X}\})$.

Proof. The operations $\sigma_{A^X}$ are given by $\sigma_A^X \circ \iota_{|\sigma|}$ where $\iota_{|\sigma|} : (A^{[|\sigma|]}X) \cong (A^X)^{|\sigma|}$ is the canonical isomorphism. $\square$

This allows us to define the notion of left and right homomorphisms.

Definition 2.3 Suppose $(A, \{\sigma_A\})$ and $(B, \{\sigma_B\})$ are $C_{\Sigma}$-algebras and $Z$ is any object of $C$. Then a right $\Sigma$-homomorphism is a map $h : Z \times A \to B$ for which the exponential transpose $\hat{h} : (A, \{\sigma_A\}) \to (B^Z, \{\sigma_B^Z\})$ is a $\Sigma$-homomorphism. We usually write $h : Z \times (A, \{\sigma_A\}) \to (B, \{\sigma_B\})$ for such a right $\Sigma$-homomorphism.

Notice that for every canonical exponential algebra $(A^X, \{\sigma_{A^X}\})$, the evaluation map $\text{eval} : X \times (A^X, \{\sigma_{A^X}\}) \to (A, \{\sigma_A\})$ becomes a right homomorphism, since it is the exponential transpose of the identity map $A^X \to A^X$.

Following Schröder and Simpson [10], we define observationally-induced algebras in a parameterised setting.

Definition 2.4 Suppose $(O, \{\sigma_O\})$ is a fixed $C_{\Sigma}$-algebra, serving as computational prototype (or observational prototype).

- An abstract $(O, \{\sigma_O\})$-structure over an object $X$ is a tuple $(A, \{\sigma_A\}, \eta)$ where $(A, \{\sigma_A\})$ is a $C_{\Sigma}$-algebra and $\eta : X \to A$ a $C$-morphism such that for every $C$-object $Z$, every $C$-morphism $f : Z \times X \to O$ extends to a unique right $\Sigma$-
homomorphism $\tilde{f} : Z \times (A, \{\sigma_A\}) \rightarrow (O, \{\sigma_O\})$ along $Z \times \eta$, as in:

\[
\begin{array}{ccc}
Z \times A & \xrightarrow{\tilde{f}} & O \\
\downarrow \eta & & \downarrow \\
Z \times X & & \\
\end{array}
\]

- A complete $(O, \{\sigma_O\})$-algebra is a $C_\Sigma$-algebra $(B, \{\sigma_B\})$ such that for every $C$-object $X$ and every abstract $(O, \{\sigma_O\})$-structure $(A, \{\sigma_A\}, \eta)$ over $X$, every morphism $f : Z \times X \rightarrow B$ extends to a unique right $\Sigma$-homomorphism $\tilde{f} : Z \times (A, \{\sigma_A\}) \rightarrow (B, \{\sigma_B\})$ along $Z \times \eta$, as in:

\[
\begin{array}{ccc}
Z \times A & \xrightarrow{\tilde{f}} & B \\
\downarrow \eta & & \downarrow \\
Z \times X & & \\
\end{array}
\]

The category of complete $(O, \{\sigma_O\})$-algebras and $\Sigma$-homomorphisms between them is denoted by $C_{(O,\{\sigma_O\})}$.

- Provided it exists, the free complete $(O, \{\sigma_O\})$-algebra over an object $X$ is called the free observationally-induced $(O, \{\sigma_O\})$-algebra over $X$. It is uniquely determined (up to isomorphism) and given by a complete $(O, \{\sigma_O\})$-algebra $(A, \{\sigma_A\})$ such that there exists a map $\eta : X \rightarrow A$ making $(A, \{\sigma_A\}, \eta)$ an abstract $(O, \{\sigma_O\})$-structure $(A, \{\sigma_A\}, \eta)$ over $X$.

To motivate the use of observationally-induced algebras for modelling effect types consider the following. According to Plotkin and Power [8] most computational effects are algebraic, i.e. triggered by algebraic operations. These operations define a signature $\Sigma$ and, in principle, every $\Sigma$-algebra is a potential denotation of an effect type. However, as pointed out by Plotkin and Power, some coherence conditions are needed in order to define a well-behaved effect type. In op. cit. they formulate these coherence conditions in form of algebraic equations, e.g. expressing that a binary nondeterministic choice operation should be commutative, associative and idempotent. Thus, they propose effect types to be modelled by algebras for the corresponding equational theory. The observationally-induced approach differs in the respect that the coherence conditions are completely determined by the properties of the observational prototype. Indeed, the following results show that it suffices that complete $(O, \{\sigma_O\})$-algebras behave like the prototype $(O, \{\sigma_O\})$ with respect to abstract $(O, \{\sigma_O\})$-structures, in order to inherit many properties of the prototype, including but not limited to equational laws. Therefore, one might consider the observationally-induced approach towards modelling computational effects as a refinement of the work of Plotkin and Power.
Let us start by observing the important fact that complete \((O, \{\sigma_O\})\)-algebras are closed under exponentiation and taking limits in \(C\).

**Lemma 2.5** If \((B, \{\sigma_B\})\) is a complete \((O, \{\sigma_O\})\)-algebra, then for every \(C\)-object \(Y\), the canonical \(C_\Sigma\)-algebra \((B^Y, \{\sigma_{B^Y}\})\) is a complete \((O, \{\sigma_O\})\)-algebra, as well.

**Proof.** Suppose \((A, \{\sigma_A\}, \eta)\) is an abstract \((O, \{\sigma_O\})\)-structure over \(X\), and \(f : Z \times X \to A^Y\) a \(C\)-morphism. The exponential transpose yields a morphism \(Y \times Z \times X \to A\) which has a unique right \(\Sigma\)-homomorphism extension \(Y \times Z \times (A, \{\sigma_A\}) \to (B, \{\sigma_B\})\) whose exponential transpose is the required unique right \(\Sigma\)-homomorphism extension \(\overrightarrow{f} : Z \times (A, \{\sigma_A\}) \to (B^Y, \{\sigma_{B^Y}\})\).

**Lemma 2.6** The forgetful functor \(C_{(O, \{\sigma_O\})} \to C\) creates limits.

**Proof.** It is well-known that the forgetful functor \(C_\Sigma \to C\) creates limits. Essentially the argument is that whenever \(D\) is a diagram consisting of \(C_\Sigma\)-algebras and \(\Sigma\)-homomorphisms for which a limit exists in \(C\), then for any \(\sigma \in \Sigma\), the diagram \(D^{|\sigma|}\) also has a limit in \(C\), namely \(\text{Lim}(D)^{|\sigma|}\). Moreover, for every \(\sigma \in \Sigma\) the universal property of a limit yields a \(C\)-morphism \(\text{Lim}(D)^{|\sigma|} \cong \text{Lim}(D^{|\sigma|}) \to \text{Lim}(D)\) which gives the required operation \(\sigma_{\text{Lim}(D)}\).

If \((A, \{\sigma_A\}, \eta)\) is an abstract \((O, \{\sigma_O\})\)-structure over \(X\), every \(C\)-morphism \(f : Z \times X \to \text{Lim}(D)\) induces a family of morphisms \(Z \times X \to D\) which have unique right homomorphism extensions \(Z \times (A, \{\sigma_A\}) \to D\). Again by the universal property of a limit, these induce a unique right homomorphism \(\overrightarrow{f} : Z \times (A, \{\sigma_A\}) \to \text{Lim}(D)\) extending \(f\) along \(\eta\), as required.

With these observations in hand, we can show that complete \((O, \{\sigma_O\})\)-algebras satisfy all \(\Sigma\)-equations which hold in \((O, \{\sigma_O\})\), provided the category under consideration supports a free algebra construction for equational theories.

Let us recall that a \(\Sigma\)-equation \(e\) is given by a pair of \(\Sigma\)-terms \((t, t')\) of the same arity, say \(|e|\). These \(\Sigma\)-terms give rise to families of maps \(t_{(-)}, t'_{(-)} : (-)^{|e|} \to (-)\) indexed by \(C_\Sigma\)-algebras which commute with \(\Sigma\)-homomorphisms. A \(C_\Sigma\)-algebra \((A, \{\sigma_A\})\) is said to satisfy the equation if \(t_A = t'_A\) as maps in \(C\).

**Lemma 2.7** Let \(E\) be a set of \(\Sigma\)-equations and assume that \((O, \{\sigma_O\})\) satisfies all equations in \(E\) and that \(C\) supports a free \((\Sigma, E)\)-algebra construction. Then every complete \((O, \{\sigma_O\})\)-algebra \((A, \{\sigma_A\})\) satisfies the equations in \(E\).

**Proof.** Let \((A, \{\sigma_A\})\) be a complete \((O, \{\sigma_O\})\)-algebra, and \((FA, \{\sigma_{FA}\})\) be the free \((\Sigma, E)\)-algebra over \(A\) with free algebra inclusion \(\eta\). Furthermore, let \(e := (t, t')\) be an equation in \(E\) with arity \(|e|\). Since \((O, \{\sigma_O\})\) satisfies all equations in \(E\), we get that \((FA, \{\sigma_{FA}\}, \eta)\) is an abstract \((O, \{\sigma_O\})\)-structure over \(A\). Moreover, the previous lemmas prove that this also holds for the corresponding \(|e|\)-powers, and that \((A^{[e]}, \{\sigma_{A^{[e]}}\})\) is a complete \((O, \{\sigma_O\})\)-algebra.
Hence, we obtain the following commutative diagram:

\[
\begin{array}{c}
\begin{array}{ccc}
FA & \xrightarrow{\bar{r}} & A \\
\downarrow^{t_{FA}} & & \downarrow^{t_A} \\
FA[e] & \xrightarrow{id[e]} & A[e]
\end{array} \\
\begin{array}{c}
\xi[e] \\
\end{array}
\end{array}
\]

Thus, the equality of \(t_{FA}\) and \(t'_{FA}\) implies the equality of \(t_A\) and \(t'_A\), and so \((A, \{\sigma_A\})\) satisfies the equation \(e\).

Finally we observe that complete \((O, \{\sigma_O\})\)-algebras deserve the term complete by showing that their underlying \(C\)-objects belong to any full reflective subcategory to which \(O\) belongs, again provided that \(C\) supports a free algebra construction.

**Lemma 2.8** Suppose \(C\) supports a free \(\Sigma\)-algebra construction. Let \(D\) be a full reflective subcategory of \(C\), for which the reflection functor \(R : C \to D\) preserves (finite) products. If \(O\) is an object of \(D\), then so is the underlying space \(A\) of every complete \((O, \{\sigma_O\})\)-algebra \((A, \{\sigma_A\})\).

**Proof.** Let \(F\) denote the free \(\Sigma\)-algebra functor. In [1, Chapter 5] it is shown that a product-preserving reflection functor lifts to the categories of algebras and that \(F \circ R \cong R \circ F\). This can be used to show that for an arbitrary \(C\)-object \(X\), we obtain an abstract \((O, \{\sigma_O\})\)-structure \((FRX, \{\sigma_{FRX}\}, \beta)\) over \(X\), where \(\beta\) is the composition of the units of the reflection and the free \(\Sigma\)-algebra construction. In particular this yields a one-to-one correspondence between maps \(X \to A\) and \(RX \to A\) along the unit of the reflection. Instantiating \(X\) with \(A\) yields the required result.

These results show our claim that the category \(C_{(O, \{\sigma_O\})}\) of complete \((O, \{\sigma_O\})\)-algebras inherits many desired properties from the computational prototype. Hence, a free observationally-induced algebra construction can indeed be considered as a refinement of Plotkin and Power’s algebraic approach to computational effects.

### 3 Existence and characterisation in cartesian closed categories

In this section we show that under fairly mild conditions on the ambient cartesian closed category \(C\), free observationally-induced algebras exist for arbitrary computational prototypes, and that they can be obtained as subobjects of certain double-exponentials.

Let us again assume the computational prototype \((O, \{\sigma_O\})\) to be given. The mild conditions on \(C\) we impose are the following two requirements.
Requirement 1 The category $\mathcal{C}$ supports a free $\Sigma$-algebra construction, i.e. the forgetful functor $\mathcal{C}_\Sigma \to \mathcal{C}$ has a left adjoint.

Observe that by Lemma 2.2, the free $\Sigma$-algebras are automatically parametrically free, i.e. for every $\mathcal{C}_\Sigma$-algebra $(A, \{\sigma_A\})$ and every morphism $f : Z \times X \to A$ there exists a unique right $\Sigma$-homomorphism extension $\overline{f} : Z \times (F_X, \{\sigma_{F_X}\}) \to (A, \{\sigma_A\})$ along the free algebra unit $X \to F_X$.

Requirement 2 The category $\mathcal{C}$ has equalizers and infima of regular subobjects, i.e. every family $\{m_i : X_i \to Y\}$ of regular monos with the same codomain has a limit in $\mathcal{C}$.

These two requirements are closely related to completeness of the category $\mathcal{C}$. Indeed, Requirement 2 holds in every complete category and completeness can be of great help when applying the Adjoint Functor Theorem to establish Requirement 1. However, in the next section we apply our results in $\mathcal{QCB}$, the central category of topological domain theory, and $\mathcal{QCB}$ does satisfy Requirements 1 and 2 although it fails to be complete in general.

As mentioned in the introduction, the double-exponential of which the free observationally-induced algebra is carved out is a continuation-like type with the computational prototype as result type. For a given object $X$ it is the object $O^O^X$, which by Lemma 2.5 carries a canonical $\Sigma$-algebra structure making it a complete $(O, \{\sigma_O\})$-algebra. We make the following straightforward observation.

**Proposition 3.1** For every $\mathcal{C}$-object $X$, let $\gamma_X : X \to O^O^X$ denote the map given by $\lambda x.\lambda t.x$, i.e. the exponential transpose of the evaluation map $X \times O^X \to O$. Then every morphism $f : Z \times X \to O^O^X$ has a (not necessarily unique) right $\Sigma$-homomorphism extension $\overline{f} : Z \times (O^O^X, \{\sigma_{O^O^X}\}) \to (O, \{\sigma_O\})$ along $\gamma_X$.

**Proof.** Consider the exponential transpose $\widehat{f} : Z \to O^X$ of $f$, and define the required map to be the composite:

$$Z \times O^O^X \xrightarrow{\widehat{f} \times O^O^X} O^X \times O^O^X \xrightarrow{\text{eval}} O$$

In $\lambda$-terms this map is given by $\lambda z.\lambda g.g(\widehat{f} z)$. It is a routine verification via manipulations of $\lambda$-terms to show that this map is a right $\Sigma$-homomorphism and that it extends $f$ along $\gamma_X$. $\square$

Since $(O^O^X, \{\sigma_{O^O^X}\})$ is a complete $(O, \{\sigma_O\})$-algebra, the map $\gamma_X : X \to O^O^X$ from the proposition above has a unique homomorphism extension $\overline{\gamma}_X : (F_X, \{\sigma_{F_X}\}) \to (O^O^X, \{\sigma_{O^O^X}\})$, where the domain is the free $\Sigma$-algebra over $X$.

**Definition 3.2** For a $\mathcal{C}_\Sigma$-algebra $(B, \{\sigma_B\})$ we define a $\Sigma$-subalgebra to be a $\mathcal{C}_\Sigma$-algebra $(A, \{\sigma_A\})$ for which there exists a regular mono $A \to B$ in $\mathcal{C}$ which becomes a $\Sigma$-homomorphism on the corresponding algebra structures.

If $(A, \{\sigma_A\})$ is a $\Sigma$-subalgebra of $(B, \{\sigma_B\})$ and both are complete $(O, \{\sigma_O\})$-algebras, we call $(A, \{\sigma_A\})$ a complete $(O, \{\sigma_O\})$-subalgebra of $(B, \{\sigma_B\})$. 45
Under Requirement 2, Lemma 2.6 shows that $C_{(O, \{\sigma_O\})}$ has infima of complete $(O, \{\sigma_O\})$-subalgebras. In particular this yields that (up to isomorphism) there exists a smallest complete $(O, \{\sigma_O\})$-subalgebra $(A, \{\sigma_A\})$ of $(O^{O^X}, \{\sigma_{O^{O^X}}\})$ through which $\overline{\gamma_X} : (FX, \{\sigma_{FX}\}) \to (O^{O^X}, \{\sigma_{O^{O^X}}\})$ factors via homomorphisms as in:

$$
\begin{array}{c}
A \\
\overline{\gamma_X} \\
FX
\end{array} \to
\begin{array}{c}
O^{O^X} \\
\uparrow \\
\end{array}
$$

We show that this yields the free complete $(O, \{\sigma_O\})$-algebra over X.

**Theorem 3.3** For every $C$-object $X$, the free complete $(O, \{\sigma_O\})$-algebra over $X$ exists and is given by the smallest complete $(O, \{\sigma_O\})$-subalgebra $(A, \{\sigma_A\})$ of $(O^{O^X}, \{\sigma_{O^{O^X}}\})$ through which $\overline{\gamma_X}$ factors.

**Proof.** Clearly, $(A, \{\sigma_A\})$ is a complete $(O, \{\sigma_O\})$-algebra. Thus it suffices to show that one can obtain it as part of an abstract $(O, \{\sigma_O\})$-structure $(A, \{\sigma_A\}, \eta)$ over $X$.

The morphism $\eta : X \to A$ is given by the free algebra inclusion $X \to FX$ followed by the mediating morphism of the factorization of $\overline{\gamma_X}$. Now suppose $f : Z \times X \to O$ is a $C$-morphism. We obtain the following diagram which consists of right $\Sigma$-homomorphisms where appropriate and commutes in all possible directions:

$$
\begin{array}{ccc}
Z \times A & \longrightarrow & Z \times O^{O^X} \\
\downarrow \quad \downarrow \quad \downarrow \\
Z \times FX & \longrightarrow & O
\end{array}
$$

Thus we get a right $\Sigma$-homomorphism extension of $f$ along $Z \times \eta$. It remains to show that this extension is unique.

Assume that $h, h' : Z \times (A, \{\sigma_A\}) \to (O, \{\sigma_O\})$ are right $\Sigma$-homomorphism extensions of $f$ along $Z \times \eta$. Consider their exponential transposes $\tilde{h}, \tilde{h}' : (A, \{\sigma_A\}) \to (O^{Z^2}, \{\sigma_{O^{Z^2}}\})$ and the corresponding equalizer $e : A' \to A$ which, by Lemma 2.6, carries a complete $(O, \{\sigma_O\})$-algebra structure. Thus, it is a complete $(O, \{\sigma_O\})$-subalgebra of $(O^{O^X}, \{\sigma_{O^{O^X}}\})$ through which $\overline{\gamma_X}$ factors. Since $(A, \{\sigma_A\})$ was the smallest such subalgebra, it follows that $e$ must be an isomorphism, hence $h = h'$. □

Obviously, this construction yields a left adjoint to the forgetful functor $C_{(O, \{\sigma_O\})} \to C$, hence it induces a monad on $C$. Using the argument of Lemma 4.5.7 in [1], one can show that this monad is strong.
Corollary 3.4 The free complete \((O, \{\sigma_O\})\)-algebra construction induces a strong monad on the underlying cartesian closed category \(C\).

Thus, given a cartesian closed category \(C\) and an observational prototype \((O, \{\sigma_O\})\), \(C\) supports a free complete \((O, \{\sigma_O\})\)-algebra construction whenever it satisfies Requirements 1 and 2. More specifically, in that case the free complete \((O, \{\sigma_O\})\)-algebra over \(X\) is given by a subobject (or subalgebra) of \(O^{OX}\). This result is not limited to ordinary finitary algebraic signatures, but applies to signatures with infinitary operations, parameterised operations as in [1] or operations whose arity is given by an object of the ambient category. Of course, care has to be taken with respect to Requirement 1.

4 The lower powerspace construction in QCB

In this section we investigate an observationally-induced lower powerspace construction in the category \(QCB\) of continuous maps between topological quotients of countably-based topological spaces. This category lies at the core of topological domain theory [3], an extension of classical domain theory with excellent closure properties. The results in [1] show that \(QCB\) satisfies Requirements 1 and 2, provided that the signature \(\Sigma\) consists of operations with finite or countable arity which are allowed to be parameterised. For the paper at hand it suffices to consider ordinary finitary operations.

Corollary 4.1 Let \(\Sigma\) be a signature consisting of finitary operations and \((O, \{\sigma_O\})\) be a \(QCB_{\Sigma}\)-algebra. Then the forgetful functor \(QCB_{(O, \{\sigma_O\})} \rightarrow QCB\) has a left adjoint giving a free observationally-induced \((O, \{\sigma_O\})\)-algebra construction. Moreover, for a given \(QCB\)-space \(X\), the free \((O, \{\sigma_O\})\)-algebra over \(X\) is given by a subalgebra of the canonical \((O^{OX}, \{\sigma_{OX}\})\).

In [4] it has been shown that \(QCB\) supports a free observationally-induced probabilistic powerspace construction by taking \((I, \oplus)\) as observational prototype, where \(I\) is the unit interval equipped with the Scott-topology for the usual ordering and \(\oplus\) the average sum operation. The proof in op.cit. differs from the method in the previous section, in that it uses an Adjoint Functor Theorem applied internally inside a realizability topos into which \(QCB\) embeds. Results from [10] could be used to show that for countably-based spaces \(X\), the free observationally-induced probabilistic powerspace over \(X\) in \(QCB\) is given by a space whose underlying set consists precisely of the probability valuations of \(X\), resembling the classical domain-theoretic construction. However, an example of Gruenhage and Streicher [5] was used to show that this classical intrinsic construction cannot be recovered by the observationally-induced approach on all \(QCB\)-spaces. The same example can easily be adapted to show that the classical intrinsic upper powerspace construction, which for a topological space \(X\) is given by a space whose underlying set consists precisely of the (proper) Scott-open filters of the topology of \(X\), cannot be recovered in \(QCB\) as an observationally-induced free algebra construction. In fact, neither can the intrinsic upper powerspace construction be recovered via observationally-
induced algebras in the category of continuous maps between topological spaces, as an example in [2] shows. In contrast to these results we show that there is an observationally-induced free algebra construction which for all QCB-spaces $X$ yields a space whose underlying set is given precisely by the nonempty closed subsets of $X$, hence resembles the classical intrinsic lower powerdomain construction.

As in [2], the observational prototype for the observationally-induced lower powerspace construction in QCB is given by $(S, \lor)$, the Sierpinski space $S$ with a binary operation (representing a nondeterministic choice) given by join. The results of the previous section show that we obtain the observationally-induced lower powerspace over a QCB-space $X$ as a complete $(S, \lor)$-subalgebra of $(S^S^X, \lor_{S^S^X})$. In the following we identify the elements of $S^S^X$ with collections of open subsets of $X$ (which are Scott-open with respect to the inclusion order). It is then an easy observation that the canonical $\lor_{S^S^X}$ is precisely given by set-union $\cup$ on these collections.

Observe that the nonempty closed subsets of $X$ embed into $S^S^X$ by the mapping:

$$B \mapsto \tilde{B} := \{ U \in S^X | U \cap B \neq \emptyset \}.$$ 

We equip the nonempty closed subsets of $X$ with the sequential (or equivalently compactly-generated) subspace topology for this embedding $(\tilde{\cdot})$ and obtain the space $C(X)$. Notice that $(\tilde{\cdot})$ preserves all unions/joins although in general it fails to preserve intersections/meets. Let us identify the image of $(\tilde{\cdot})$ in $S^S^X$.

**Lemma 4.2** An element $T \in S^S^X$ is in the image of $(\tilde{\cdot}) : C(X) \to S^S^X$ if and only if it satisfies the following two conditions:

(i) $X \in T$,

(ii) if $U \cup V \in T$, then $U \in T$ or $V \in T$.

**Proof.** It is a straightforward observation that for all $B \in C(X)$, $\tilde{B}$ satisfies (i) and (ii).

Conversely suppose $T$ satisfies (i) and (ii). Then it is easy to see that

$$B_T := X \setminus \bigcup \{ V \in S^X | V \notin T \}$$

is a nonempty closed subset of $X$ such that $T \equiv \tilde{B}_T$. \hfill $\Box$

Next, observe that for every $U \in S^X$ the corresponding evaluation map $\text{eval}_U : S^S^X \to S$, given by $\text{eval}_U(T) = T$ if and only if $U \in T$, is a continuous $\lor$-homomorphism. This observation is a key to the next result which shows that $(C(X), \cup)$ forms a complete $(S, \lor)$-algebra.

**Proposition 4.3** For every QCB-space $X$, $(C(X), \cup)$ is a complete $(S, \lor)$-algebra.

**Proof.** Suppose $(A, \oplus, \eta)$ is an abstract $(S, \lor)$-structure over $Y$ and $f : Z \times Y \to \{ 0, 1 \}^S$.
\[ C(X) \text{ a continuous map. We obtain the following commuting diagram:} \]

\[
\begin{array}{ccc}
Z \times A & \overset{(-)}{\longrightarrow} & \mathcal{C}(X) \\
\uparrow \quad & & \searrow \gamma_X \\
Z \times Y & \underset{f}{\longrightarrow} & \mathbb{S}^{S_X}
\end{array}
\]

where \( h \) is the unique right homomorphism extension of \((-) \circ f\) obtained from the fact that \((\mathbb{S}^{S_X}, \vee_{\mathbb{S}^{S_X}})\) is a complete \((\mathbb{S}, \vee)\)-algebra by Lemma 2.5. It suffices to show that the elements in the image of \( h \) satisfy conditions (i) and (ii) of Lemma 4.2.

For (i) observe that \( X \in \tilde{\gamma} f(z, y) \) for all \((z, y) \in Z \times Y\), hence by the unique extension property we must have \( X \in h(z, a) \) for all \((z, a) \in Z \times A\). This is the case, because the map \( \text{eval}_X \circ h \) is a continuous right homomorphism extension of the constant map \( \top : Z \times X \to S \) along \( Z \times \eta \). Thus, if there exists \((z_0, a_0) \in Z \times A\), then \( \top : Z \times X \to S \) had at least two right homomorphism extensions along \( Z \times \eta \).

For (ii) assume \( U \cup V \in h(z_0, a_0) \). This yields that \( \text{eval}_{U \cup V} \circ h(z_0, a_0) = \top \). Moreover, we have that \( \text{eval}_{U \cup V} \circ (-) \circ f \equiv (\text{eval}_U \circ (-) \circ f) \vee (\text{eval}_V \circ (-) \circ f)\), since every \( f(z, y) \) satisfies (ii). Thus these two maps have the same continuous right homomorphism extension along \( Z \times \eta \). But \( \vee \) composes continuous right homomorphisms, hence we get that:

\[
(\text{eval}_U \circ h)(z_0, a_0) \vee (\text{eval}_V \circ h)(z_0, a_0) = (\text{eval}_{U \cup V} \circ h)(z_0, a_0) = \top,
\]

and thus it holds that \( U \in h(z_0, a_0) \) or \( V \in h(z_0, a_0) \), as required.

Notice that in particular it follows from this proof, that the observationally-induced lower powerspace over a \textbf{QCB}-space \( X \) is a complete \((\mathbb{S}, \vee)\)-subalgebra of \((\mathcal{C}(X), \cup)\), for the following reason. Consider the diagram:

\[
\begin{array}{ccc}
FX & \overset{\gamma_X}{\longrightarrow} & \mathbb{S}^{S_X} \\
\eta_X \uparrow & & \gamma_X \uparrow \\
X & \underset{\gamma_X}{\longrightarrow} & \mathbb{S}^{S_X}
\end{array}
\]

where \( \gamma_X \) is the canonical embedding, \( FX \) denotes the underlying space of the absolutely-free \( \Sigma \)-algebra over \( X \) (for a signature \( \Sigma \) consisting of a binary operation symbol) with the free algebra inclusion map \( \eta_X \) and \( \gamma_X \) is the unique homomorphism extension of \( \gamma_X \) along \( \eta_X \). Then, since \((FX, \star, \eta_X)\) is an abstract \((\mathbb{S}, \vee)\)-structure over \( X \), the argument of the proof above shows that it factors via homomorphisms through \((-) : \mathcal{C}(X) \to \mathbb{S}^{S_X}\). Of course, Theorem 3.3 applies.
We want to show that there exists no proper complete \((\mathbb{S}, \lor)\)-subalgebra of \((\mathcal{C}(X), \cup)\) through which this inclusion factors.

**Proposition 4.4** The only complete \((\mathbb{S}, \lor)\)-subalgebra of \((\mathcal{C}(X), \cup)\) that contains the image of \(\overline{\pi_X} : FX \to \mathbb{S}^{SX}\) is \((\mathcal{C}(X), \cup)\) itself.

**Proof.** By Theorem 2.8 we know that the underlying set of any complete \((\mathbb{S}, \lor)\)-subalgebra of \((\mathcal{C}(X), \cup)\) belongs to any full reflective subcategory that \(\mathbb{S}\) belongs to. In particular, that means it must be a monotone convergence space, i.e. under the specialization order it forms a dcpo.

Moreover, the image of \(\overline{\pi_X} : FX \to \mathbb{S}^{SX}\) in \(\mathcal{C}(X)\) consists exactly of the closures of nonempty finite subsets of \(\mathcal{C}(X)\). Notice that every closed set can be obtained as a directed supremum of the closures of finite subsets it contains. Suppose we find a complete \((\mathbb{S}, \lor)\)-subalgebra of \((\mathcal{C}(X), \cup)\) that contains all closures of nonempty finite subsets of \(X\). We denote it by \((A, \cup)\). Since the embedding \(j : (A, \cup) \hookrightarrow (\mathcal{C}(X), \cup)\) must be a continuous homomorphism, it must preserve directed (continuous) and binary (homomorphism) joins. Thus, it preserves all joins and we get:

\[
\bigvee_{F \subseteq fin, B} j(cl(F)) = \bigvee_{F \subseteq fin, B} j(cl(F)) = B.
\]

where \(cl(F)\) denotes the closure of the finite subset \(F\), the first join is taken in \(A\), the second join is taken in \(\mathcal{C}(X)\).

As mentioned, every nonempty closed subset can be obtained in this way. Hence, \(j\) is surjective and the claim follows. \(\square\)

Of course the corresponding abstract \((\mathbb{S}, \lor)\)-structure \((\mathcal{C}(X), \cup, (\overline{\_}))\) is obtained from the point closure map \((\overline{\_}) : X \to \mathcal{C}(X)\).

**Theorem 4.5** The observationally-induced lower powerspace over a QCB-space \(X\) is given by \((\mathcal{C}(X), \cup)\).

In Section 4.4.2 of [9], Schröder has shown that on \(\mathcal{C}(X)\), the subspace topology of \(\mathbb{S}^{SX}\) coincides with the sequentialisation of the lower Vietoris topology, which is generated by sets of the form:

\[
\langle U \rangle := \{B \in \mathcal{C}(X) | B \cap U \neq \emptyset\},
\]

for \(U\) ranging over the open subsets of \(X\). This relates the observationally-induced lower powerspace construction in QCB to the one in the category of continuous maps between topological spaces given in [2]. There it was shown that for every topological space \(X\), the free complete \((\mathbb{S}, \lor)\)-algebra over \(X\) is given by the set of nonempty closed subsets of \(X\) under the lower Vietoris topology with set-union as operation. We do not know whether there exists a QCB-space \(X\) for which the lower Vietoris topology on the nonempty closed subsets is not sequential. Also Schröder’s work does not give any insights for this question.

On the other hand, we also do not know exactly for which QCB-spaces the topology on \(\mathcal{C}(X)\) coincides with the Scott-topology for the inclusion order. Despite
the fact that as a complete lattice $\mathcal{C}(X)$ has a rich structure and the embedding into $\mathbb{S}^{\mathbb{S}^X}$, which carries the Scott-topology, preserves all suprema, we can do no better than concluding that our construction at least coincides with the classical domain-theoretic construction for continuous dcpos.

**Corollary 4.6** For a countably-based continuous dcpo $X$ the observationally-induced lower powerspace $(\mathcal{C}(X), \cup)$ in QCB carries the Scott-topology.

## 5 Conclusions

We have shown that free observationally-induced algebras exist in cartesian closed categories for arbitrary computational prototypes, provided the category under consideration satisfies certain closure requirements. Moreover, in this case free observationally-induced algebras can be characterised as subalgebras of a double exponential corresponding to a type of continuations with the prototype as results type. The closure requirements we have identified are that the category needs to have infima of regular subobjects and support an ordinary free algebra construction for the signature defining the algebra structure of the prototype. These requirements are fairly mild and satisfied in most complete categories, such as the categories of monotone maps between preordered sets, continuous maps between dcpos, or continuous maps between compactly generated spaces. They are also satisfied in QCB, the central category in topological domain theory, provided the arity of operations in the signature is at most countable. Thus, the observationally-induced approach to modelling computational effects extends the toolbox of denotational semantics for functional programming languages nicely. Unfortunately, the requirements are not satisfied in any cartesian closed category of continuous or algebraic dcpos, since these categories do not have infima of regular subobjects in general. Hence, this work does not give any insights into the well-known problem of modelling computational effects in cartesian closed categories of domains. On the other hand, our results can be applied in the category of Scott-continuous maps between general dcpos.

Furthermore, we have shown that in QCB one has an observationally-induced lower powerspace construction which for a QCB-space $X$ is characterised as the space of nonempty closed subsets of $X$ equipped with the subspace topology of the double exponential $\mathbb{S}^{\mathbb{S}^X}$ which itself carries the Scott-topology. This confirms the well-behavedness of the lower powerspace construction in comparison to the upper powerspace construction, for which the observationally-induced approach does not have such a neat characterisation as the counterexample in [2] shows. Also Gruenhage and Streicher’s counterexample [5] is easily seen to adapt to the upper powerspace construction.

The characterisation of observationally-induced free algebras as subobjects of double exponentials is a very interesting result. It reveals a deeper connection between algebraic effects and continuation types. Thus, it somehow appears to be an orthogonal result to Plotkin and Power’s observation that continuations fall outside the scope of algebraic effects. However, deeper investigations are necessary.
for determining in how far this characterisation of observationally-induced algebras can be of help, e.g. in modelling the interplay of algebraic effects and continuations [6] or whether it is possible to apply results of the diverse research on continuations to the world of computational effects.

Finally, we remark that the construction of free observationally-induced algebras as subobjects of double exponentials bears resemblance to the work of Vickers and Townsend [12] on powerlocales. They show that lower and upper powerlocale constructions are both obtained as subobjects of a certain double exponential and their composition commutes and yields the full double exponential. It should be investigated under which conditions free observationally-induced algebra constructions commute and in how far it is possible to obtain full double exponentials with our approach. The category of locales is not cartesian closed, hence it is not covered by the present paper. However, certain exponentials do exist in locales and there are certainly strong connections to the work on free observationally-induced algebras over topological spaces [2], which should be investigated further.

Acknowledgement

The author wishes to thank Ernst-Erich Doberkat, Matthias Schröder, Alex Simpson and Thomas Streicher for helpful discussions and encouragement.

References


Abstract

We study the domain-theoretic semantics of a Church-style typed \( \lambda \)-calculus with constructors, pattern matching and recursion, and show that it is closely related to the semantics of its untyped counterpart. The motivation for this study comes from program extraction from proofs via realizability where one has the choice of extracting typed or untyped terms from proofs. Our result shows that under a certain regularity condition, the choice is irrelevant. The regularity condition is that in every use of a fixed point type \( \text{fix} \, \alpha.\rho \), \( \alpha \) occurs only positively in \( \rho \).

Keywords: Scott domain, typed and untyped \( \lambda \)-calculus, program extraction, logical relation, realizability.

1 Introduction

This paper is part of a research project aiming at a semantical foundation for program extraction from proofs [7]. It contributes to a soundness proof for a language of realizers of proofs involving inductive and coinductive definitions. The natural language of realizers for inductive and coinductive definitions is a typed lambda calculus with types modeling initial algebras and final coalgebras, and terms modeling structural recursion and corecursion. In this paper we study a more general calculus that allows fixed points of arbitrary type operators and definitions of functions by general recursion. The advantage of this generality is that our results will apply to all conceivable extensions of our theory of realizers of inductive and coinductive definitions.

We study the domain-theoretic semantics of a Church-style typed \( \lambda \)-calculus with constructors, pattern matching and recursion, and compare it with its untyped counterpart. We work with polymorphic types that allow fixed points of arbitrary type operators. A type \( \rho \) is interpreted as (the image of) a finitary projection \( \langle \rho \rangle \),
following [2]. The main result (Theorem 4.15) relates the semantics of a typed term $M$ with its untyped variant $M^-$: if $M$ has type $\rho$, where $\rho$ is a regular type, that is, fixed points are only taken of positive operators, then

$$\langle \rho \rangle [M^-] = [M].$$

The proof uses logical relations. We do not know whether the result also holds if $\rho$ is not regular.

A similar problem was studied by Reynolds [13,14] who established a coherence between the typed and untyped meanings of expressions based on cpo models of a version of PCF. The main differences to our work are as follows: Reynolds considers simple types over the base types of natural numbers and booleans while we allow arbitrary recursive types. On the other hand, he includes subtyping which we do not. Regarding the typed semantics, Reynolds interprets typing derivations in a typed model while we interpret terms with a typed abstraction in an untyped model.

The motivation for this study comes from program extraction from proofs via realizability (see e.g. [6,4,7,8] for applications in constructive analysis) where one has the choice of extracting typed or untyped terms from proofs. Our result shows that if the extracted type is regular, the choice is irrelevant. In fact, regularity is a harmless restriction because in the intended realizability interpretation the types of realizing terms will always be regular. In [5] the soundness of a realizability interpretation based on a fragment of the untyped version of our calculus was proven, and the calculus was shown to be computationally adequate with respect to a domain-theoretic semantics (the same semantics we are considering here). In [9] it was shown that the extracted programs admit a Curry-style typing. In the present paper we provide the missing semantical link to Curry-style typing.

The application to realizability is also our motivation for working with Scott domains (instead of arbitrary cpos, as Reynolds does): the adequacy proof in [5] uses the fact that all semantic objects can be approximated by compact ones, hence we have to ensure that types are interpreted in a cartesian closed category of algebraic domains. This is achieved by interpreting types as finitary projections. Apart from that, the results of this paper could also be obtained using arbitrary cpos and embedding-retraction pairs.

The problem of relating typed and untyped realizability was also studied by Longley [10]. He used a condition called (constructive) logical full abstraction to connect realizability over typed and untyped structures by means of partial combinatorial algebra.

Our paper is organized as follows. Section 2 introduces the syntax of types and typed terms, and typing rules for typed terms. Next, in Section 3, the semantics of types and typed terms are described in the setting of Scott domains and its correctness is proved. And then, in Section 4, the relation between the domain-theoretic semantics of typed terms and the semantics of their untyped counterparts are studied. The proof uses logical relations, which are related to Tait’s computability method and Girard’s method of reducibility candidates. Finally, Section 5 concludes the
paper and outlines some future work.

2 Types and terms

In this and the next section we study the syntax and semantics of types and typed terms. Untyped terms will be introduced in Section 4.

**Definition 2.1** [Types] The set of types is defined by the following grammar:

\[
\text{Type} \ni \rho, \sigma, \tau := \alpha \mid \rho \rightarrow \sigma \mid 1 \mid \rho \times \sigma \mid \rho + \sigma \mid \text{fix}\, \alpha.\rho.
\]

where \(\alpha\) ranges over a set \(\text{TVar}\) of type variables. A fixed-point construction, \(\text{fix}\, \alpha.\rho\), binds all free occurrences of \(\alpha\) in \(\rho\).

We work with a Church-style typed lambda-calculus with constructors, pattern matching and recursion which we call *Language of Realizers* (LoR) because its terms are intended to be used as extracted programs from proofs obtained by a realizability interpretation.

We consider only the constructors Nil (nullary), Pair (binary), and Left, Right, In (unary). The intention behind the first four constructors should be obvious. The constructor In is used to model type fixed points up to isomorphism. Many definitions and results could be extended to an arbitrary set of constructors.

**Definition 2.2** [Terms] The set of (Church-style typed) terms is defined by

\[
\text{LoR} \ni M, N, R := x \mid \lambda x : \rho.\, M \mid MN \mid \text{rec } x : \rho.\, M \mid C(M_1, \ldots, M_n) \mid \text{case } M \text{ of } \{C_i(x_i) \rightarrow R_i\}_{i \in \{1, \ldots, n\}}.
\]

where \(x\) ranges over a set of variables \(\text{Var}\), \(C\) is a constructor of arity \(n\), and in \(\text{case } M \text{ of } \{C_i(x_i) \rightarrow R_i\}_{i \in \{1, \ldots, n\}}\) all constructors \(C_i\) are distinct and each \(x_i\) is a vector of distinct variables whose length coincide with the arity of \(C_i\). Lambda abstraction, \(\lambda x : \rho.\, M\), and recursion, \(\text{rec } x : \rho.\, M\), bind all free occurrences of \(x\) in \(M\), and a pattern matching clause, \(C_i(x_i) \rightarrow R_i\), binds all free occurrences of \(x_i\) in \(R_i\).

We introduce typing rules for LoR-terms. A *type context* is a set of pairs \(\Gamma := x_1 : \rho_1, \ldots, x_n : \rho_n\) (for notational convenience we omit the curly braces) where \(\rho_i\) are types and \(x_i\) are distinct variables. The set of variables \(\{x_1, \ldots, x_n\}\) (which may be empty) is denoted by \(\text{dom}(\Gamma)\).

The relation \(\Gamma \vdash M : \rho\) (\(M\) is a LoR term of type \(\rho\) in context \(\Gamma\)) is inductively defined as follows. Note that in the definition of terms (Definition 2.2), case expression can have in general many clauses, but typing rules only allow two or one clauses which contains respectively binary sum constructors (Left and Right), or
other constructors (Pair or In).

\[\begin{align*}
\Gamma \vdash \text{Nil} : 1 & \quad \Gamma, x : \rho \vdash x : \rho \\
\Gamma, x : \rho \vdash M : \sigma & \quad \Gamma, x : \tau \vdash M : \tau \\
\Gamma \vdash \lambda x : \rho. M : \rho \to \sigma & \quad \Gamma \vdash \text{rec } x : \tau. M : \tau \\
\Gamma \vdash M : \rho \to \sigma & \quad \Gamma \vdash N : \rho \\
\Gamma \vdash M \cdot N : \sigma & \quad \Gamma \vdash \text{Pair}(M, N) : \rho \times \sigma \\
\Gamma \vdash M : \rho & \quad \Gamma \vdash M : \sigma \\
\Gamma \vdash \text{Left}(M) : \rho + \sigma & \quad \Gamma \vdash \text{Right}(M) : \rho + \sigma \\
\Gamma \vdash M : \rho + \sigma & \quad \Gamma, x_1 : \rho \vdash L : \tau \\
\Gamma, x_2 : \sigma \vdash R : \tau & \quad \Gamma \vdash \text{case } M \text{ of } \{\text{Left}(x_1) \to L; \text{Right}(x_2) \to R\} : \tau \\
\Gamma \vdash M : \rho \times \sigma & \quad \Gamma, x, y : \sigma \vdash N : \tau \\
\Gamma \vdash \text{case } M \text{ of } \{\text{Pair}(x, y) \to N\} : \tau \\
\Gamma \vdash M : \rho[\text{fix } \alpha. \rho/\alpha] \\
\Gamma \vdash \text{In}(M) : \text{fix } \alpha. \rho \\
\Gamma \vdash M : \text{fix } \alpha. \rho & \quad \Gamma, x : \rho[\text{fix } \alpha. \rho/\alpha] \vdash N : \sigma \\
\Gamma \vdash \text{case } M \text{ of } \{\text{In}(x) \to N\} : \sigma
\end{align*}\]

3 Domain-theoretic semantics

We assume familiarity with the basic theory of Scott domains and the method of defining domains by recursive domain equations [15,1,3]. We omit the proofs of the most basic results since they are rather elementary, or can be found in the above cited literature. The reason for working with Scott domains is that all the semantic constructions we need are readily available, e.g. cartesian closure, solutions to recursive domain equation, recursive definition of functions, interpretation of types, including recursive types, as finitary projections. All these constructions are very elementary and do not require a heavy category-theoretical machinery.

By a Scott-domain, or domain for short, we mean a bounded complete \(\omega\)-algebraic dcpo with least element. We will denote the least element of a domain by \(\bot\). By \(\mathbf{1}\) we denote the sole-element domain \(\{\text{Nil}\}\), and by \((D_1 + \ldots + D_n)\bot, D \times E, [D \to E]\) the separated sum, cartesian product, and continuous function space of domains\(^3\).

\(^3\) These domain operations should not be confused with the syntactic constructors for types which for simplicity we denoted by the same symbols.
Due to $\omega$-algebraicity, every element $x$ of a domain $D$ is the directed countable supremum of compact elements, where $y \in D$ is called compact if for every directed $A \subseteq D$, $A$ has a supremum $\sqcup A$ and $y \sqsubseteq \sqcup A$ s.t. $y \sqsubseteq z$ for some $z \in A$. By $D_c$ we denote the set of compact elements of $D$.

**Definition 3.1** [Subdomain] $E \subseteq D$ is a subdomain of $D$ if

1. $\bot_D \in E$.
2. If $A \subseteq E$ and $\sqcup_D A$ exists in $D$, then $\sqcup_D A \in E$ and $\sqcup_D A = \sqcup_E A$.
3. If $x$ is compact in $E$, then $x$ is compact in $D$.
4. $\forall y \in D_c \forall x \in E (y \subseteq x \rightarrow \exists y' \in E_c (y \subseteq y' \sqsubseteq x))$.

**Lemma 3.2** Let $E \subseteq D$ be a subdomain of $D$. Then $E$ is a domain.

**Proof.** By verifying clauses (directed complete, algebraic, bounded complete) of the definition of domain. \hfill $\Box$

Following [2] we interpret types as finitary projections in $D$. Since the range of a finitary projection is a subdomain of $D$ the semantics of types can be viewed as a domain. This approach provides an easy solution to the problem of defining the semantics of a fixed point type: one can simply take the least fixed point of a suitable continuous function on the domain $[D \rightarrow D]$.

**Definition 3.3** [Finitary projection] $f : D \rightarrow D$ is a projection if

- $f$ is continuous,
- $f \sqsubseteq \text{id}$, i.e. $\forall x \in D. f(x) \sqsubseteq x$,
- $f \circ f = f$, i.e. $\forall x \in D. f(f(x)) = f(x)$.

A projection $f$ is finitary if the range of $f$, denoted by $f(D)$, is a subdomain of $D$.

For $f : D \rightarrow D$ we set $\text{Fix}(f) := \{ x \in D \mid f(x) = x \}$. Obviously, if $f \circ f = f$, then $f(D) = \text{Fix}(f)$.

In the following two lemmas we assume that $p : D \rightarrow D$ is a projection, and we set $p(D)_c := D_c \cap p(D)$. We omit their proofs since they are easy.

**Lemma 3.4** $E$ is a subdomain of $D$ if and only if there exists a finitary projection $p : D \rightarrow D$ such that $E = p(D)$.

**Lemma 3.5** The following are equivalent:

(a) $p$ is finitary
(b) $\forall x \in D (A_x := \{ a \in p(D)_c \mid a \sqsubseteq x \})$ is directed and $p(x) = \sqcup A_x$.
(c) $\exists A \subseteq D_c \forall x \in D (p(x) = \sqcup \{ a \in A \mid a \sqsubseteq x \})$.

Now we define by a recursive domain equation a particular domain $D$ which we will use to interpret types and terms.
Definition 3.6 We define the Scott domain $D$ by the recursive domain equation:

$$D \simeq (1 + D + D + D + D \times D + [D \to D])_\bot$$

Using the constructors of LoR as names for the injections into the sum, each element in $D$ has exactly one of the following forms: $\bot$, Nil, Left($a$), Right($a$), In($a$), Pair($a, b$), Fun($f$), where $a$ and $b$ range over $D$, and $f$ ranges over continuous functions from $D$ to $D$.

It will be convenient to use the following continuous functions: $\text{case}_{C_1,\ldots,C_n} : D \to [D^{\text{arity}(C_1)} \to D] \to \cdots \to [D^{\text{arity}(C_n)} \to D] \to D$ defined by

$$\text{case}_{C_1,\ldots,C_n} a f_1 \ldots f_n := \begin{cases} f_i(b_i) & \text{if } a = C_i(b_i), \\ \bot & \text{otherwise.} \end{cases}$$

We also use an informal lambda-notation $\lambda a. f(a)$ and composition $f \circ g$ to define continuous functions on $D$. We do not prove the continuity in each case since this follows from well-known fact about the category of Scott domains and continuous functions. We also let $\text{LFP} : [D \to D] \to D$ be the continuous least fixed point operator, which can be defined by $\text{LFP}(f) = \bigsqcup_n f^n(\bot)$.

The following definition gives an unexpected interpretation of a type $\rho$ as a finitary projection $\langle \rho \rangle$, but from this one can derive the usual definition as the image (or, equivalently, set of fixed points) of $\langle \rho \rangle$. Also $\langle \rho \rangle$ will be used later in Theorem 4.15, and act as a function (not only a type), providing a link between the two semantics.

Definition 3.7 [Semantics of types] For every type $\rho$ we define $\langle \rho \rangle : ([D \to D]^{\text{TVar}} \to [D \to D])$

$$\langle 1 \rangle(\alpha)(a) = \text{case}_{\text{Nil}} a \text{ Nil} \quad \left(\begin{array}{l} \text{Nil if } a = \text{Nil} \\ \bot \text{ otherwise} \end{array}\right)$$

$$\langle \alpha \rangle(\alpha)(a) = \zeta(\alpha)(a)$$

$$\langle \rho + \sigma \rangle(\alpha)(a) = \text{case}_{\text{Left,Right}} a \text{ Left}(\sigma \circ \rho)(\sigma)$$

$$\langle \rho \times \sigma \rangle(\alpha)(a) = \text{case}_{\text{Pair}} a \text{ Pair}(\lambda b_1. \text{Pair}(\rho)(\sigma)(b_1), \sigma)(b_2))$$

$$\langle \rho \to \sigma \rangle(\alpha)(a) = \text{case}_{\text{Fun}} a \text{ Fun}(\lambda f. \sigma(\rho)(\sigma) \circ f \circ \rho)$$

$$\langle \text{fix } \alpha. \rho \rangle := \text{LFP}(\lambda \rho. \lambda a. \text{case}_{\text{In}} a \text{ In}(\langle \rho \rangle(\alpha)(a) := p(b)))$$

We set $[\rho] := (\langle \rho \rangle)\langle D \rangle$.

We call $\zeta : [D \to D]^{\text{TVar}}$ a finitary projection if $\zeta(\alpha)$ is a finitary projection for all $\alpha \in \text{TVar}$.

Lemma 3.8 If $\zeta$ is a finitary projection, then $\langle \rho \rangle \zeta$ is a finitary projection.

\footnote{[D \to D]^{\text{TVar}} \text{ is the set of type environments, i.e., functions from TVar to } [D \to D].}
Proof. By induction on $\rho$ using Lemma 3.4 and 3.5.

Now we are ready to define the semantics of LoR-terms. The leading idea in the definition of the value of a typed lambda-abstraction $\lambda x : \rho. M$ is that the domain of the resulting function is (the semantics of) $\rho$. Therefore, the incoming argument $a$ is first projected down to $\rho$.

Definition 3.9 [Semantics of terms] For every environment $\zeta : [D \rightarrow D]^{tVar}$, $\eta : Var \rightarrow D$, and every LoR term $M$ we define the value $\llbracket M \rrbracket^\zeta \eta \in D$.

\[
\llbracket x \rrbracket^\zeta \eta = \eta(x)
\]
\[
\llbracket C(M_1, \ldots, M_n) \rrbracket^\zeta \eta = C(\llbracket M_1 \rrbracket^\zeta \eta, \ldots, \llbracket M_n \rrbracket^\zeta \eta)
\]
\[
\llbracket [\lambda x : \rho.M] \rrbracket^\zeta \eta = \text{caseFun}(\llbracket M \rrbracket^\zeta \eta, \lambda f. (\llbracket f \rrbracket^\zeta \eta))
\]
\[
\llbracket [\text{rec } x : \tau. M] \rrbracket^\zeta \eta = \text{LFP}(\lambda x. \llbracket M \rrbracket^\zeta \eta[x := \langle \tau \rangle(\zeta(a))])
\]
\[
\llbracket [\text{case } M \text{ of } \{C_i(x_i) \rightarrow R_i\}_{i \in \{1, \ldots, n\}}] \rrbracket^\zeta \eta = \text{case}_{C_1, \ldots, C_n}(\llbracket M \rrbracket^\zeta \eta, \lambda a. \llbracket R_i \rrbracket^\zeta \eta[x_i := a])
\]

One can prove the following soundness theorem, stating that if from a context $\Gamma$ we can derive LoR term $M$ with type $\rho$, and for every variable $x \in \text{dom}(\Gamma)$, $\eta(x)$ is an element of $\llbracket \Gamma(x) \rrbracket^\zeta$, then the value of term $M$ is an element of the value of type $\rho$. We write $\eta \in \llbracket \Gamma \rrbracket^\zeta \zeta$ as an abbreviation for $\eta(x) \in \llbracket \Gamma(x) \rrbracket^\zeta$.

Theorem 3.10 (Soundness For LoR terms) Let $\zeta$ be a finitary projection. If $\Gamma \vdash M : \rho$ and $\eta \in \llbracket \Gamma \rrbracket^\zeta$, then $\llbracket M \rrbracket^\zeta \eta \in [\rho]^\zeta$.

Proof. By induction on the structure of the relation $\Gamma \vdash M : \rho$.

4 Relating typed and untyped terms

We now relate the semantics of typed terms with the semantics of untyped terms which are defined exactly as typed terms except that the type annotations for abstraction and recursion are omitted:

Definition 4.1 [Untyped terms]

\[
\text{LoR}^- \ni M, N, R_i ::= x | \lambda x.M | MN | \text{rec } x.M | C(M_1, \ldots, M_n) | \text{case } M \text{ of } \{C_i(x_i) \rightarrow R_i\}_{i \in \{1, \ldots, n\}}
\]

The same provisions made in Definition 2.2 for typed terms apply here.

The semantics of untyped terms is straightforward. It can be defined exactly as in the typed case except that the type environment $\zeta : [D \rightarrow D]^{tVar}$ and finitary projections involved in typed abstraction and recursion are omitted.

Definition 4.2 [Semantics of untyped terms] For every environment $\eta : Var \rightarrow D$ and every LoR$^-$ term $M$ we define the value $\llbracket M \rrbracket \eta \in D$.
\[ [x] \eta = \eta(x) \]
\[ [C(M_1, \ldots, M_n)] \eta = C([M_1] \eta, \ldots, [M_n] \eta) \]
\[ [MN] \eta = \text{case} \eta \to F (\eta) (\lambda f. f([N] \eta)) \]
\[ [\lambda x. M] \eta = \text{Fun}(\lambda a. [M] \eta[x := a]) \]
\[ [\text{rec} \ x. M] \eta = \text{LFP}(\lambda a. [M] \eta[x := a]) \]
\[ [\text{case} M \text{ of } \{ C_i(x_i) \to R_i \}] \eta = \text{case} C_1,\ldots,C_n ([M] \eta) (\lambda a. [R_i] \eta[x_i := a]) \]

Our main result, the Coincidence Theorem 4.15, only applies to terms that are typed w.r.t. to a restricted notion of types where fixed point types fix \( \alpha, \rho \) are allowed only if \( \rho \) is positive in \( \alpha \).

**Definition 4.3** [\( \rho \) positive/negative in \( \alpha \)] We give the following definitions.

- \( \alpha \) is positive in \( \alpha \).
- \( 1 \) is positive and negative in \( \alpha \).
- \( \rho \to \sigma \) is positive in \( \alpha \) if \( \rho \) is negative in \( \alpha \) and \( \sigma \) is positive in \( \alpha \).
- \( \rho \to \sigma \) is negative in \( \alpha \) if \( \rho \) is positive in \( \alpha \) and \( \sigma \) is negative in \( \alpha \).
- \( \rho + \sigma \) and \( \rho \times \sigma \) are positive in \( \alpha \) if \( \rho \) and \( \sigma \) are positive in \( \alpha \).
- \( \rho + \sigma \) and \( \rho \times \sigma \) are negative in \( \alpha \) if \( \rho \) and \( \sigma \) are negative in \( \alpha \).
- fix \( \beta, \rho \) is positive in \( \alpha \) if \( \alpha = \beta \) or \( \rho \) is positive in \( \alpha \).
- fix \( \beta, \rho \) is negative in \( \alpha \) if \( \alpha = \beta \) or \( \rho \) is negative in \( \alpha \).

**Definition 4.4** [Regular types] We define *regular* types \( \rho \) as follows.

- \( 1 \) is regular.
- \( \alpha \) is regular.
- \( \rho + \sigma, \rho \times \sigma, \rho \to \sigma \) are regular if \( \rho \) and \( \sigma \) are regular.
- fix \( \alpha, \rho \) is regular if \( \rho \) is regular and \( \rho \) is positive in \( \alpha \).

**Example 4.5** fix \( \alpha.(\alpha \to \alpha) \) is not regular, since \( \alpha \to \alpha \) is regular but not positive in \( \alpha \).

In the following all types are assumed to be regular.

To prove our main result we define a logical relation \( \sim_R^\rho \subseteq D \times D \) which can intuitively be understood as a notion of equivalence of elements of a regular type \( \rho \). We use the informal (second-order) lambda abstraction \( \Lambda r \subseteq D^2 \) to define functions on the set \( \mathcal{P}(D^2) \) of binary relations on \( D \).

Definition 4.6 and Lemma 4.8 below should be considered simultaneously, since in the clause of fix \( \alpha, \rho \), the clause is well-defined only if \( \rho \) is positive in \( \alpha \).

**Definition 4.6** [Logical Relation] In the following definition it assumed that \( R : \mathcal{P}(D^2)^\text{Var} \).
Remark 4.7 Logical relations [12] have been used successfully to prove properties of typed systems. Famous examples are the strong normalization proofs by Tait and Girard using logical relations called computability predicates or reducibility candidates. The crucial feature of a logical relation is that it is a family of relations indexed by types and defined by induction on types such that all type constructors are interpreted by their logical interpretations, e.g. $\to$ is interpreted as logical implication.

Lemma 4.8

(1) If $\rho$ is positive in $\alpha$, then $\Lambda r \subseteq D^2. \sim^{R[\alpha:=r]}_{\rho}$ is monotone.

(2) If $\rho$ is negative in $\alpha$, then $\Lambda r \subseteq D^2. \sim^{R[\alpha:=r]}_{\rho}$ is anti-monotone.

Proof. By induction on $\rho$. 

The notion of admissibility has been used in [1] and generalized in [11], where is used to prove properties of least fixed points. An admissible relation holds for the least upper bound of a chain, if it contains $(\bot, \bot)$ and it holds for every element of the chain.

Definition 4.9 [Admissible relation] A relation $r \subseteq D^2$ on $D$ is called admissible if it satisfies

(i) $(\bot, \bot) \in r$.

(ii) If $(d_n, d'_n) \in r$ and $(d_n, d'_n) \subseteq (d_{n+1}, d'_{n+1})$ for all $n$, then $\sqcup_{n \in N}(d_n, d'_n) \in r$.

Note that a finite relation $r \subseteq D^2$ with $(\bot, \bot) \in r$ is always admissible.

Let $\text{Ad} := \{ r \subseteq D^2 \mid r \text{ is admissible} \}$.

Lemma 4.10 $\text{Ad}$ is a complete lattice. Moreover, if $A$ is a directed set of admissible sets, then $\bigcup A$ is admissible.

Proof. Easy. 

We call $R : (D^2)^{\text{TVar}}$ admissible if If $R(\alpha)$ is admissible for all $\alpha \in \text{TVar}$.

Lemma 4.11 If $R$ is admissible, then $\sim^{R}_\rho$ is admissible.

Proof. By induction on $\rho$. 

61
We only look at the most interesting case, which is fix $\alpha, \rho$.
We have $\sim_{\text{fix}, \alpha, \rho}^R = \text{LFP}(\Phi)$ where

$$\Phi : \mathcal{P}(D^2) \to \mathcal{P}(D^2)$$

$$\Phi(r) = \{((\bot, \bot)) \cup \{(\text{In}(a), \text{In}(b)) \mid a \sim^R_{\rho} b\}.\$$

Clearly $(\bot, \bot) \in \Phi(r)$ for every $r \subseteq D^2$. Hence, we have $\text{LFP}(\Phi) = r_{\beta_0}$ for some ordinal $\beta_0$, where for every ordinal $\beta$, $r_\beta$ is defined by $r_0 = \{\} \cup r_{\beta_0} = \Phi(r_\beta)$, $r_\lambda = \bigcup\{r_\beta \mid \beta < \lambda\}$ ($\lambda$ a limit ordinal).

It is easy to see that if $r \in \text{Ad}$, then $\Phi(r) \in \text{Ad}$, i.e. $\Phi : \text{Ad} \to \text{Ad}$. Hence by induction on ordinals and Lemma 4.10 it follows that all $r_\beta$ where $\beta > 0$ are admissible.

**Definition 4.12** [Compatibility] Let $r \subseteq D^2$ and $p \in [D \to D]$. We call $r$ and $p$ compatible, in symbols $r \approx p$, if

(i) $\forall a, b \in D \ (r(a, b) \to p(a) = p(b))$.
(ii) $\forall a \in D \ r(p(a), p(a))$.
(iii) $\forall a, b \in D \ (r(a, b) \to r(p(a), b))$.

We call $R : \mathcal{P}(D^2)^{\text{TVar}}$ and $\zeta : [D \to D]^\text{TVar}$ compatible, in symbols $R \approx \zeta$ if $R(\alpha) \approx \zeta(\alpha)$ holds for all $\alpha \in \text{TVar}$.

To obtain an example of compatibility one may take any idempotent $p \in [D \to D]$ and define $r \subseteq D^2$ by $r := \{(a, b) \in D^2 \mid p(a) = p(b)\}$. Then, clearly, $r \approx p$.

**Lemma 4.13** If $R$ is admissible, $\zeta$ is a finitary projection, and $R \approx \zeta$, then $\sim_R^R \approx (\rho)\zeta$.

**Proof.** We write $r \approx_{\text{i,ii}} p$ for the notion of compatibility obtained by deleting property (iii) in Definition 4.12. Similarly, $r \approx_{\text{i,iii}} p$ means compatibility where properties (i) and (ii) are deleted. The notions $R \approx_{\text{i,ii}} \zeta$ and $R \approx_{\text{i,iii}} \zeta$ are defined mutatis mutandis as in Definition 4.12.

We show that if $R$ is admissible and $\zeta(\alpha)$ is a finitary projection, then:

(1) If $R \approx_{\text{i,ii}} \zeta$, then $\sim_R^R \approx_{\text{i,ii}} (\rho)\zeta$.
(2) If $R \approx_{\text{i,iii}} \zeta$, then $\sim_R^R \approx_{\text{i,iii}} (\rho)\zeta$.

Both statements are proved by induction on $\rho$. We only look at the interesting cases which are function and fixed point types.

(1) **Case** $\rho \to \sigma$: (i) Assume $a \sim_{\rho \to \sigma} b$. We have to show $\langle \rho \to \sigma \rangle(\zeta(a)) = \langle \rho \to \sigma \rangle(\zeta(b))$. If $a = b = \bot$. Then $\langle \rho \to \sigma \rangle(\zeta(\bot)) = \langle \rho \to \sigma \rangle(\bot) = \langle \rho \to \sigma \rangle(\zeta(b))$. If $a = \text{Fun}(f)$ and $b = \text{Fun}(g)$. By the definition of $\sim^R_{\rho \to \sigma}$ (Definition 4.6), we have

$$\forall c, d \in D (c \sim_R^R d \Rightarrow f(c) \sim_R^R g(d)) \tag{*}$$

By Definition 3.7, we have $\langle \rho \to \sigma \rangle(\zeta(f)) = \text{Fun}(\langle \sigma \rangle(\zeta \circ f \circ (\rho)\zeta))$ and $\langle \rho \to \sigma \rangle(\zeta(g)) = \text{Fun}(\langle \sigma \rangle(\zeta \circ g \circ (\rho)\zeta))$. Let $c \in D$. Now we need to show $\langle \sigma \rangle(\zeta(f(\langle \rho \rangle(\zeta(c)))) = \langle \sigma \rangle(\zeta(g(\langle \rho \rangle(\zeta(c))))$. By induction hypothesis (ii) for $\rho$, we have $\langle \rho \rangle(\zeta(c)) \sim_R^R (\rho)\zeta(c)$. 62
Then by (*) we have \( f(\langle \rho \rangle \zeta(c)) \sim^R_\sigma g(\langle \rho \rangle \zeta(c)) \). And then by induction hypothesis (i) for \( \sigma \), we have \( \langle \sigma \rangle \zeta(f(\langle \rho \rangle \zeta(c))) = \langle \sigma \rangle \zeta(g(\langle \rho \rangle \zeta(c))) \).

(ii) Let \( a \in D \). We have to show \( \langle \rho \rightarrow \sigma \rangle \zeta(a) \sim^R_{\rho \rightarrow \sigma} \langle \rho \rightarrow \sigma \rangle \zeta(a) \). If \( a = \bot \). Then \( \langle \rho \rightarrow \sigma \rangle \zeta(a) = \bot \). Thus \( \bot \sim^R_{\rho \rightarrow \sigma} \bot \). If \( a = \text{Fun}(f) \). Then by Definition 3.7, we have \( \langle \rho \rightarrow \sigma \rangle \zeta(a) = \text{Fun}(\langle \sigma \rangle \zeta \circ f \circ \langle \rho \rangle \zeta) \). Now we need to show \( \text{Fun}(\langle \sigma \rangle \zeta \circ f \circ \langle \rho \rangle \zeta) \sim^R_{\rho \rightarrow \sigma} \text{Fun}(\langle \sigma \rangle \zeta \circ f \circ \langle \rho \rangle \zeta) \). By the definition of \( \sim^R_{\rho \rightarrow \sigma} \) (Definition 4.6), it is to show

\[
\forall c, d \in D(c \sim^R_\rho d \Rightarrow \langle \sigma \rangle \zeta(f(\langle \rho \rangle \zeta(c))) \sim^R_\sigma f(c) \sim^R_\sigma g(d))
\]
Case fix \( \alpha, \rho \): Set \( r := \sim_{\text{fix } \alpha, \rho}^{R} \) and \( p := \{ \text{fix } \alpha, \rho \} \). We have to show that \( r(a, b) \) implies \( r(p(a), b) \), i.e. \( r \subseteq s \) where \( s := \{(a, b) \mid r(p(a), b)\} \). Since \( r \) is the least fixed point of the operator \( \Phi := \Lambda r.\{(\bot, \bot)\cup\{(\text{In}(a), \text{In}(b)) \mid a \sim_{\rho}^{R[\alpha:=r]} b\}\} \) we can attempt to prove the inclusion \( r \subseteq s \) by induction. In fact we use the strong induction principle (see for example, [8]) according to which it suffices to show \( \Phi(r \cap s) \subseteq s \) (instead of \( \Phi(s) \subseteq s \)). Clearly \( (\bot, \bot) \in s \). Hence we assume \( a \sim_{\rho}^{R[\alpha:=r \cap s]} b \) and have to show \( r(p(In(a)), In(b)) \). Since \( p(In(a)) = In(\langle \rho\rangle \langle \alpha := p(a) \rangle, In(b)) \), i.e., \( \langle \rho\rangle \langle \alpha := p(a) \rangle \sim_{\rho}^{R[\alpha:=r]} b \). We verify that \( r \cap s \approx_{\alpha}^{\rho} p \) holds. Indeed, if \( (r \cap s)(a, b) \), then \( r(p(a), b) \) and hence, since by Lemma 3.8 \( p \) is idempotent, \( r(p((a)), b) \), i.e. \( (r \cap s)(p(a), b) \). By induction hypothesis (iii), \( \langle \rho\rangle \langle \alpha := p(a) \rangle \sim_{\rho}^{R[\alpha:=r \cap s]} b \) and hence \( \langle \rho\rangle \langle \alpha := p(a) \rangle \sim_{\rho}^{R[\alpha:=r]} b \), by monotonicity (Lemma 4.8).

Let \( \eta \sim_{\Gamma}^{R} \eta' \) denote the following: for all \( x \in \text{dom}(\Gamma) \), if \( \Gamma(x) = \sigma \), then \( \eta(x) \sim_{\sigma}^{R} \eta'(x) \).

Let \( \Gamma \vdash_{\text{reg}} M : \rho \) mean that \( \Gamma \vdash M : \rho \) has been derived using regular types only.

Let \( M' \) be the untyped term obtained from Church-style term \( M \) by deleting the type information in lambda-abstractions. The following lemma is the core of the proof of the Coincidence Theorem.

**Lemma 4.14** Assume \( R \) is admissible, \( \zeta \) is a finitary projection, and \( R \approx \zeta \). If \( \Gamma \vdash_{\text{reg}} M : \rho \) and \( \eta \sim_{\Gamma}^{R} \eta' \), then \([M]\zeta \sim_{\rho}^{R} [M']\eta'\).

**Proof.** By induction on the structure of the relation \( \Gamma \vdash_{\text{reg}} M : \rho \).

The interesting cases are lambda abstraction and recursion.

\[
\begin{align*}
\frac{}{\Gamma, x : \rho \vdash_{\text{reg}} M : \sigma} & \quad \text{(a)}
\frac{\Gamma \vdash_{\text{reg}} \lambda x : \rho. M : \rho \rightarrow \sigma}{\Gamma \vdash_{\text{reg}} \lambda x : \rho. M : \rho \rightarrow \sigma} \quad \text{(b)}
\end{align*}
\]

We have to show \([\lambda x : \rho. M]_{\zeta} \sim_{\rho \rightarrow \sigma}^{R} [\lambda x. M']_{\eta'}\). By Definition 3.9 and 4.2, we have

\[
\begin{align*}
[M]_{\zeta} \eta & = \text{Fun}(f) \text{ where } f(a) = [M]_{\zeta} \eta[x := \langle \rho \rangle \zeta(a)] \\
[\lambda x. M']_{\eta'} & = \text{Fun}(g) \text{ where } g(b) = [M']_{\eta'}[x := b]
\end{align*}
\]

Then it is to show \( \text{Fun}(f) \sim_{\rho \rightarrow \sigma} \text{Fun}(g) \). By definition of our logical relation (Definition 4.6), it is to show

\[
\forall a, b \in D(a \sim_{\rho}^{R} b \Rightarrow f(a) \sim_{\sigma}^{R} g(b)) \quad \text{(*)}
\]

By induction hypothesis for \( \sigma \) we have

\[
\forall a, b \in D(a \sim_{\rho}^{R} b \Rightarrow [M]_{\zeta} \eta[x := a] \sim_{\sigma}^{R} [M']_{\eta'}[x := b]) \quad \text{(IH)}
\]

To prove (\textit{*}), assume \( a \sim_{\rho}^{R} b \). By Lemma 4.13 (iii) we have \( \langle \rho \rangle \zeta(a) \sim_{\rho}^{R} b \). Hence \([M]_{\zeta} \eta[x := \langle \rho \rangle \zeta(a)] \sim_{\rho}^{R} [M']_{\eta'}[x := b]) \).
We have to show \([\text{rec } x : \tau. M] \xi \eta \sim^R \tau [\text{rec } x. M^-] \eta'\). By Definition 3.9 and 4.2, we have

\[
[\text{rec } x : \tau. M] \xi \eta = \text{LFP}(f) \text{ where } f(a) = [M] \xi \eta[x := \langle \tau \rangle \zeta(a)]
\]

\[
[\text{rec } x. M^-] \eta' = \text{LFP}(g) \text{ where } g(b) = [M^-] \eta'[x := b]
\]

Now we have to show \(\text{LFP}(f) \sim^R \tau \text{LFP}(g)\). By definition it is to show \(\sqcup_n f^n(\bot) \sim^R \tau \sqcup_n g^n(\bot)\). Then it is to show the following two statements.

- \(\forall n. f^n(\bot) \sim^R \tau g^n(\bot)\).
  
  By induction on \(n\).
  
  \(n = 0\): To show \(f^0(\bot) \sim^R \tau g^0(\bot)\), i.e. \(\bot \sim^R \bot\). This holds by Definition 4.6.
  
  \(n + 1\): Assume \(f^n(\bot) \sim^R \tau g^n(\bot)\), to show \(f^{n+1}(\bot) \sim^R \tau g^{n+1}(\bot)\). We have

\[
f^{n+1}(\bot) = f(f^n(\bot)) = [M] \xi \eta[x := \langle \tau \rangle \zeta(f^n(\bot))],
g^{n+1}(\bot) = g(g^n(\bot)) = [M^-] \eta'[x := g^n(\bot)].
\]

Then it is to show \([M] \xi \eta[x := \langle \tau \rangle \zeta(f^n(\bot)))] \sim^R \tau [M^-] \eta'[x := g^n(\bot)]\). By I.H. we get \(f^n(\bot) \sim^R \tau g^n(\bot) \Rightarrow [M] \xi \eta[x := \langle \tau \rangle \zeta(f^n(\bot))] \sim^R \tau [M^-] \eta'[x := g^n(\bot)]\).

- \(\sim^R \tau\) is admissible. It follows by Lemma 4.11.

\(\Box\)

The above lemma (Lemma 4.14) yields as an immediate consequence our main result that if from a context \(\Gamma\) we can derive a \textbf{LoR} term \(M\) with regular type \(\rho\), and for every variable \(x \in \text{dom}(\Gamma)\), \(\eta(x)\) is an element of \([\Gamma(x)]\xi\), then the value of \(M\) and its corresponding untyped term \(M^-\) coincide up to the finitary projection \(\langle \rho \rangle \zeta\).

**Theorem 4.15 (Coincidence)** If \(\Gamma \vdash_{\text{reg}} M : \rho\) and \(\eta \in [\Gamma]\xi\) where \(\zeta\) is a finitary projection, then \([M] \xi \eta = \langle \rho \rangle \zeta([M^-] \eta)\).

**Proof.** Given a finitary projection \(\zeta\), we define \(R(\alpha) := \{(a, b) \in \mathbb{D}^2 \mid \zeta(\alpha)(a) = p(b)\}\). Then \(R \approx \zeta\), as explained in the example following Definition 4.12. By Lemma 4.14, we then have \([M] \xi \eta \sim^R \tau [M^-] \eta\). By Lemma 4.13 (i), we get \(\langle \rho \rangle \zeta([M] \xi \eta) = \langle \rho \rangle \zeta([M^-] \eta)\). Then by Soundness Theorem 3.10 and the definition of \(\langle \rho \rangle \zeta(D)\), we have \([M] \xi \eta = \langle \rho \rangle \zeta([M] \xi \eta)\). Thus, \([M] \xi \eta = \langle \rho \rangle \zeta([M^-] \eta)\). \(\Box\)

5 Conclusion

We have studied a domain-theoretic semantics for Church-style system \textbf{LoR} of typed lambda terms and proved that, when restricted to regular types, it is closely related to its untyped counterpart. The proof uses hybrid logical relations. The reason for studying this domain-theoretic semantics is that it allows for very simply and elegant proofs of computational adequacy, and hence the correctness of program extraction.

65
Our results could be easily extended to also include full second-order polymorphism \( \forall \alpha. \rho, \exists \alpha. \rho \) as in [2], but for our application, simple parametric and recursive types are sufficient.

As future work we intend to investigate whether the requirement of regularity is indeed necessary for our result to hold. Furthermore, we plan to compare the Church-style system with a corresponding Curry-style system.

Acknowledgement

We are grateful to the anonymous referees for providing valuable suggestions and drawing our attention to related work.

References


Final Semantics for Decorated Traces

Filippo Bonchi\textsuperscript{a}, Marcello Bonsangue\textsuperscript{b,c}, Georgiana Caltais\textsuperscript{d,c}, Jan Rutten\textsuperscript{e,c}, Alexandra Silva\textsuperscript{e,c,f}

\textsuperscript{a} ENS Lyon, Universit\'e de Lyon, LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)
\textsuperscript{b} LIACS - Leiden University, The Netherlands
\textsuperscript{c} Centrum voor Wiskunde en Informatica, The Netherlands
\textsuperscript{d} School of Computer Science - Reykjavik University, Iceland
\textsuperscript{e} Radboud University Nijmegen, The Netherlands
\textsuperscript{f} HASLab / INESC TEC, Universidade do Minho, Braga, Portugal

Abstract

In concurrency theory, various semantic equivalences on labelled transition systems are based on traces enriched or decorated with some additional observations. They are generally referred to as decorated traces, and examples include ready, failure, trace and complete trace equivalence. Using the generalized powerset construction, recently introduced by a subset of the authors \cite{13}, we give a coalgebraic presentation of decorated trace semantics. This yields a uniform notion of canonical, minimal representatives for the various decorated trace equivalences, in terms of final Moore automata. As a consequence, proofs of decorated trace equivalence can be given by coinduction, using different types of (Moore-) bisimulation (up-to), which is helpful for automation.

1 Introduction

The study of systems equivalence has been an interesting research topic for many years now. Several equivalences have been proposed throughout the years, each of which suitable for use in different contexts of application. Many of the equivalences that are important in the theory of concurrency were described in the well-known paper by van Glabbeek \cite{14}.

Proof methods for the different equivalences are an important part of this research enterprise. In this paper, we propose coinduction as a general proof method.

\footnote{The work of Georgiana Caltais has been partially supported by a CWI Internship and by the project 'Meta-theory of Algebraic Process Theories' (nr. 100014021) of the Icelandic Research Fund. The work of Alexandra Silva is partially funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/122240/2010 and SFRH/BPD/71956/2010. We thank Luca Aceto and Anna Ing"olfsd"ottir for comments and references to the literature.}

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
for what van Glabbeek calls decorated trace semantics, which includes (complete) trace, ready and failure semantics.

Coinduction is a general proof principle which has been uniformly defined in the theory of coalgebras for different types of state-based systems and infinite data types. Given a functor $\mathcal{F}: \text{Set} \to \text{Set}$, an $\mathcal{F}$-coalgebra is a pair $(X, f)$ consisting of a set of states $X$ and a function $f: X \to \mathcal{F}(X)$ defining the dynamics of the system. The functor $\mathcal{F}$ determines the type of the transition system or data type under study. For a large class of functors $\mathcal{F}$, there exists a final coalgebra into which every $\mathcal{F}$-coalgebra is mapped by a unique homomorphism. Intuitively, one can see the final coalgebra as the universe of all behaviours of systems and the unique morphism as the map assigning to each system its behaviour. This provides a standard notion of equivalence called $\mathcal{F}$-behavioural equivalence. Moreover, these canonical behaviours are minimal, by general coalgebraic considerations \[10\], in that no two different states are equivalent.

Labelled transition systems (LTS’s) can be modelled as coalgebras for the functor $\mathcal{F}(X) = (\mathcal{P}_\omega X)^A$ and the canonical behavioural equivalence associated with $\mathcal{F}$ is precisely the finest equivalence of the spectrum in \[14\]. In the recent past, other equivalences of the spectrum have been also cast in the coalgebraic framework. Notably, trace semantics was widely studied \[5,13\] and, more recently, decorated trace semantics was recovered via a coalgebraic generalization of the classical powerset construction \[12\].

To get some intuition on the type of distinctions the equivalences above encompass, consider the following labelled transition systems over the alphabet $A = \{a, b, c\}$:

The traces of the states $p, q, r$ and $s$ are $\{a, ab, ac\}$, and therefore they are all trace equivalent. Complete trace semantics identifies states that have the same set of complete traces, that is, traces that lead to states where no further action are possible. Of the four states above, $q$ and $r$ and $s$ are complete trace equivalent, but not $p$ since it is the only state that has $a$ as a complete trace. Failure semantics takes into account the set of actions that cannot be fired immediately after the execution of a certain trace. Only $r$ and $s$ are failure equivalent, since after $a$, state $p$ might not be able to fire actions $b$ and $c$, whereas $p$, $r$ and $s$ might not be able to fire only one of $b$ or $c$ and $q$ never fails with those two actions. Ready semantics identifies states according to the set of actions they can trigger immediately after a certain trace has been executed. None of the states above are ready equivalent, since after $a$ only $p$ has the option of not executing any action, $q$ and $s$ can choose from $b$ or $c$ but $r$ cannot and $q$ always has two options $b, c$ whereas $s$ can end in a state where only $b$ or $c$ can be taken.

The contributions of the paper are three-fold. First, we prove that the coalgebraic decorated trace semantics, which are mentioned without proof in \[12\] as
examples, are equivalent to the corresponding set-theoretic notions from [14]. Second, we show how the coalgebraic semantic leads to canonical representatives for the various decorated trace equivalences. Third, we show how to prove decorated trace equivalence using coinduction, by constructing bisimulations (up-to congruence) that witness the desired equivalence. The latter is interesting also from the point of view of tool development: construction of bisimulations is known to be particularly suitable for automation. Moreover, the up-to congruence technique also increases the efficiency of reasoning, as verifications are performed under certain closure properties, which means the bisimulations that are built are smaller (see Section 3, and Section 4 for an example). The techniques we use here for up-to reasoning are an extension of the recent work by the first author [2].

The paper is organized as follows. In Section 2, we provide the basic notions from coalgebra and recall the generalized powerset construction. In Section 3, we show how the powerset construction can be applied for determinizing LTS’s in terms of Moore automata \((X, f: X \rightarrow B \times X^A)\), in order to coalgebraically characterize decorated trace semantics. A detailed description of coalgebraic ready semantics is provided in Section 4. Here we also prove that the obtained coalgebraic model is equivalent to the original definition, and illustrate how one can reason about ready equivalence by constructing bisimulations up-to congruence. By following the approach in Section 4, similar results can be easily shown for (complete) trace and failure semantics coalgebraically modelled as in [12]. Section 5 discusses that the canonical representatives of LTS’s we obtain coalgebraically coincide with the minimal LTS’s one would obtain by identifying all states equivalent w.r.t. a particular decorated trace semantics. Section 6 contains concluding remarks and discusses future work.

2 Preliminaries

In this section, we briefly recall basic notions from coalgebra and the generalized powerset construction [5,13]. We first introduce some notation on sets.

We denote sets by capital letters \(X, Y, \ldots\) and functions by lower case letters \(f, g, \ldots\). The cartesian product of two sets \(X\) and \(Y\) is denoted by \(X \times Y\), and has the projection maps \(X \xrightarrow{\pi_1} X \times Y\) and \(X \xrightarrow{\pi_2} Y\). The disjoint union of \(X\) and \(Y\) is written \(X + Y\) and has the injection maps \(X \xleftarrow{k_1} X + Y\) and \(Y \xleftarrow{k_2} Y\). By \(X^Y\) we represent the family of functions \(f: Y \rightarrow X\), whereas the collection of finite subsets of \(X\) is denoted by \(\mathcal{P}_\omega X\). For each of these operations defined on sets, there is an analogous one on functions (for details see for example [1]). This turns the operations above into (bi)functors, which we shall use throughout this paper.

For an alphabet \(A\), we denote by \(A^*\) the set of all words over \(A\) and by \(\varepsilon\) the empty word. The concatenation of words \(w_1, w_2 \in A^*\) is written \(w_1w_2\).

**Coalgebras:** We consider coalgebras of functors \(\mathcal{F}\) defined on \(\text{Set}\) – the category of sets and functions. An \(\mathcal{F}\)-coalgebra (or coalgebra, when \(\mathcal{F}\) is understood) is a pair \((X, c: X \rightarrow \mathcal{F}X)\), where \(X \in \text{Set}\). We call \(X\) the state space, and we say that \(\mathcal{F}\) together with \(c\) determine the dynamics, or the transition structure of the
\(\mathcal{F}\)-coalgebra.

An \(\mathcal{F}\)-homomorphism between two \(\mathcal{F}\)-coalgebras \((X, f)\) and \((Y, g)\), is a function \(h: X \to Y\) preserving the transition structure, i.e., \(g \circ h = \mathcal{F}(h) \circ f\).

An \(\mathcal{F}\)-coalgebra \((\Omega, \omega)\) is final if for any \(\mathcal{F}\)-coalgebra \((X, f)\) there exists a unique \(\mathcal{F}\)-homomorphism \([\cdot]_X: X \to \Omega\). A final coalgebra represents the universe of all possible behaviours of \(\mathcal{F}\)-coalgebras. The unique morphism \([\cdot]_X: X \to \Omega\) maps each state in \(X\) to its behaviour. Using this mapping, behavioural equivalence can be defined as follows: for any two coalgebras \((X, f)\) and \((Y, g)\), the states \(x \in X\) and \(y \in Y\) are behaviourally equivalent, written \(x \sim_{\mathcal{F}} y\), if and only if they have the same behaviour, that is

\[
x \sim_{\mathcal{F}} y \text{ iff } [x]_X = [y]_Y.
\]  

We think of \([x]_X\) as the canonical representative of the behaviour of \(x\). Also it can be viewed as the minimization of \((X, f)\), since the final coalgebra contains no pairs of equivalent states.

For an example we consider deterministic automata (DA). A deterministic automaton over the input alphabet \(A\) is a pair \((X, \langle o, t \rangle)\), where \(X\) is a set of states and \(\langle o, t \rangle: X \to 2 \times X^2\) is a function with two components: \(o\), the output function, determines if a state \(x\) is final \((o(x) = 1)\) or not \((o(x) = 0)\); and \(t\), the transition function, returns for each input letter \(a\) the next state. DA’s are coalgebras for the functor \(\mathcal{D}(X) = 2 \times X^2\). The final coalgebra of this functor is \((2^{A^*}, \langle \varepsilon, (-)_a \rangle)\) where \(2^{A^*}\) is the set of languages over \(A\) and \(\langle \varepsilon, (-)_a \rangle\), given a language \(L\), determines whether or not the empty word is in the language \((\varepsilon(L) = 1\) or \(\varepsilon(L) = 0\), resp.) and, for each input letter \(a\), returns the derivative of \(L\): \(\langle w \rangle = \{ w \in A^* \mid aw \in L\}\). From any DA, there is a unique map \([\cdot]_X\) into \(2^{A^*}\) which assigns to each state its behaviour (that is, the language that the state recognizes).

\[
\begin{array}{c}
X \xrightarrow{\langle o, t \rangle} 2 \times X^2 \xrightarrow{\text{id} \times [\cdot]_X} 2 \times (2^{A^*})^A \\
\end{array}
\]

Therefore, behavioural equivalence for the functor \(\mathcal{D}\) coincides with the classical language equivalence of automata.

Another example (fundamental for the rest of the paper) is given by Moore automata. Moore automata with inputs in \(A\) and outputs in \(B\) are coalgebras for the functor \(\mathcal{M}(X) = B \times X^A\), that is pairs \((X, \langle o, t \rangle)\) where \(X\) is a set, \(t: X \to X^A\) is the transition function (like for DA) and \(o: X \to B\) is the output function which maps every state in its output. Thus DA can be seen as a special case of Moore automata where \(B = 2\). The final coalgebra for \(\mathcal{M}\) is \((B^{A^*}, \langle \varepsilon, (-)_a \rangle)\) where \(B^{A^*}\) is the set of all functions \(\varphi: A^* \to B\), \(\varepsilon: B^{A^*} \to B\) maps each \(\varphi\) into \(\varphi(\varepsilon)\) and \((-)_a: B^{A^*} \to (B^{A^*})^A\)

70

Bonchi, Bonsangue, Caltais, Rutten, Silva
is defined for all \( \varphi \in B^A \), \( a \in A \) and \( w \in A^* \) as \( (\varphi)_a(w) = \varphi(aw) \).

\[
\begin{align*}
X & \xrightarrow{\{ \cdot \}} B^A^* \\
\downarrow^{(\cdot,\cdot)} & \downarrow^{(\epsilon,(-)_a)} \\
B \times X^A & \xrightarrow{id \times \{ \cdot \}} B \times (B^A)^A \\
\end{align*}
\]

\[
[x]_X(\epsilon) = o(x) \\
[x]_X(aw) = [t(x)(a)]X(w)
\]

Coalgebras provide a useful technique for proving behavioural equivalence: \emph{bisimulation}. Let \((X, f)\) and \((Y, g)\) be two \(\mathcal{F}\)-coalgebras. A relation \( R \subseteq X \times Y \) is a bisimulation if there exists a function \( \alpha_R \colon R \to \mathcal{F}R \) such that \( \pi_1 \colon R \to X \) and \( \pi_2 \colon R \to Y \) are coalgebra homomorphisms. In \cite{Bonchi2018}, it is shown that under certain conditions on \( \mathcal{F} \) (which are met by all the functors in this paper), bisimulations are a sound and complete proof technique for behavioural equivalence, namely,

\[
x \sim_\mathcal{F} y \text{ iff there exists a bisimulation } R \text{ such that } x R y.
\] (2)

The \emph{generalized powerset construction}: As shown above, every functor \( \mathcal{F} \) induces both a notion of \( \mathcal{F}\)-coalgebra and a notion of behavioural equivalence \( \sim_\mathcal{F} \). Sometimes, it is interesting to consider different equivalences than \( \sim_\mathcal{F} \) for reasoning about \( \mathcal{F}\)-coalgebras. This is the case of labeled transition systems which are coalgebras for the functor \( \mathcal{L}(X) = (\mathcal{P}_\omega X)^A \). The induced behavioural equivalence \( \sim_\mathcal{L} \) coincides with the standard notion of bisimilarity by Milner and Park \cite{Milner1980, Park1981}. However, in concurrency theory, many other equivalences have been studied, notably, \emph{decorated trace equivalences} \cite{Rutten1991}. Another example is given by non-deterministic automata which are coalgebras for the functor \( \mathcal{N}(X) = 2 \times (\mathcal{P}_\omega X)^A \). The associated equivalence \( \sim_\mathcal{N} \) strictly implies language equivalence, which is often taken as an intended semantics.

For this reason, a subset of the authors has introduced in \cite{Bonchi2018} the \emph{generalized powerset construction}, for coalgebras \( f \colon X \to \mathcal{F}T(X) \) for a functor \( \mathcal{F} \) and a monad \( T \), with the proviso that that \( \mathcal{F}T(X) \) is an algebra for the monad \( T \). In \cite{Bonchi2018}, all the technical details are explored and many interesting instances of the construction are shown. In this paper, we will only be interested in the case where \( T = \mathcal{P}_\omega \) and \( M(X) = B \times X^A \), for \( B \) a semilattice, and we will therefore only explain the concrete picture for the functor and monad of interest. The fact that we take \( B \) to be a semilattice is enough to guarantee that \( MT(X) = B \times (\mathcal{P}_\omega X)^A \) is a semilattice. This fulfills then the proviso above, since semilattices are precisely the algebras of the monad \( \mathcal{P}_\omega \).

Given a coalgebra \( f \colon X \to M\mathcal{P}_\omega X \), and because \( M \) has a final coalgebra, we can extend it uniquely to \( f^\sharp \colon \mathcal{P}_\omega X \to M\mathcal{P}_\omega X \) and consider the unique coalgebra homomorphism into the final coalgebra, as summarised by the following diagram:

\[
\begin{align*}
X & \xrightarrow{\{ \cdot \}} \mathcal{P}_\omega X \xrightarrow{\{ \cdot \}} B^A^* \\
\downarrow^{f} & \downarrow^{f^\sharp} \\
B \times (\mathcal{P}_\omega X)^A & \xrightarrow{id \times \{ \cdot \}} B \times (B^A)^A
\end{align*}
\] (3)
With this construction, one can co-algebraically characterize language equivalence for Moore automata and, in particular, for non-deterministic automata. Take \( T = \mathcal{P}_\omega \) and \( \mathcal{F} = \mathcal{D} \), which is an instance of \( \mathcal{M} \) for \( B = 2 \), the two-element semi-lattice. An \( \mathcal{MT} \)-coalgebra is a pair \((X, f)\) with \( f: X \to 2 \times (\mathcal{P}_\omega X)^A \), i.e., an NDA. Therefore every NDA \((X, f)\) is transformed into \((\mathcal{P}_\omega X, f^\sharp)\) which is a DA. This corresponds to the classical powerset construction for determinizing non-deterministic automata. The language recognized by a state \( x \) can be defined by precomposing the unique morphism \([-]\): \( \mathcal{P}_\omega X \to 2^A \) with the unit of \( \mathcal{P}_\omega \), which is the function \( \{\}\): \( X \to \mathcal{P}_\omega X \) mapping each \( x \in X \) into the singleton set \( \{x\} \in \mathcal{P}_\omega X \).

### 3 Decorated trace semantics via determinization

Our aim is to reason about decorated trace equivalences of labelled transition systems. In this section, we use the generalized powerset construction and show how one can determinize arbitrary labelled transition systems obtaining particular instances of Moore automata (with different output sets) in order to model ready, failure, trace and complete trace equivalences. This paves the way to building a general framework for reasoning on decorated trace equivalences in a uniform fashion, in terms of bisimulations up-to congruence.

A **labeled transition system** is a pair \((X, \delta)\) where \( X \) is a set of states and \( \delta: X \to (\mathcal{P}_\omega X)^A \) is a function assigning to each state \( x \in X \) and to each label \( a \in A \) a finite set of possible successors states. We write \( x \xrightarrow{a} y \) whenever \( y \in \delta(x)(a) \). We extend the notion of transition to words \( w = a_1 \ldots a_n \in A^* \) as follows: \( x \xrightarrow{w} y \) if and only if \( x \xrightarrow{a_1} \ldots \xrightarrow{a_n} y \). For \( w = \varepsilon \), we have \( x \xrightarrow{\varepsilon} y \) if and only if \( y = x \).

We now define in a nutshell the equivalences we will be dealing with in this paper. For a function \( \varphi \in (\mathcal{P}_\omega X)^A \), \( I(\varphi) \) denotes the set of all labels “enabled” by \( \varphi \), given by \( I(\varphi) = \{a \in A \mid \varphi(a) \neq \emptyset\} \), while \( \text{Fail}(\varphi) \) denotes the set \( \{Z \subseteq A \mid Z \cap I(\varphi) = \emptyset\} \).

Let \((X, \delta)\) be a LTS and \( x \in X \) be a state. A **trace** of \( x \) is a word \( w \in A^* \) such that \( x \xrightarrow{w} y \) for some \( y \). A trace \( w \) of \( x \) is **complete** if \( x \xrightarrow{w} y \) and \( y \) stops, i.e., \( I(\delta(y)) = \emptyset \).

A **failure pair** of \( x \) is a pair \((w, Z)\) such that \( x \xrightarrow{w} y \) and \( Z \in \text{Fail}(\delta(y)) \).

A **ready pair** of \( x \) is a pair \((w, Z)\) such that \( x \xrightarrow{w} y \) and \( Z = I(\delta(y)) \).

(See [14] for more details on the classical definition of traces, complete traces, ready and failure pairs.) We use \( \mathcal{T}(x), \mathcal{CT}(x), \mathcal{F}(x) \) and \( \mathcal{R}(x) \) to denote, respectively, the set of all traces, complete traces, failure pairs and ready pairs of \( x \).

For \( \mathcal{I} \) ranging over \( \mathcal{T}, \mathcal{CT}, \mathcal{F} \) and \( \mathcal{R} \), two states \( x \) and \( y \) are **\( \mathcal{I} \)-equivalent** iff \( \mathcal{I}(x) = \mathcal{I}(y) \) [14].

Intuitively, these equivalences can be described as follows:

- **ready semantics** identifies states of LTS’s according to the set \( Z \) of actions they can trigger immediately after a certain action sequence \( w \) has been “consumed”; we call a pair \((w, Z)\) a **ready pair**,
- **failure semantics** takes into account the set \( Z \) of actions that cannot be fired immediately after the execution of sequences \( w \); we call a pair \((w, Z)\) a **failure pair**,
- **trace semantics** identifies system states if and only if they can execute the same
sets of action sequences \( w \),

- **complete trace semantics** identifies system states that perform the same sets of “complete” traces \( w \); we call an action sequence \( w \) a complete trace of a state \( p \) if and only if \( p \xrightarrow{w} q \) and \( q \) cannot execute any further action.

The slight difference between trace and complete trace semantics consists in the fact that trace semantics does not detect stagnation, whereas the latter semantics takes into consideration deadlock states.

The coalgebraic characterization of the equivalences above was obtained in [12] in the following way. Given an arbitrary LTS \((X, \delta : X \to (\mathcal{P}_\omega X)^A)\), we associate a decorated LTS represented by a co-algebra of the functor \( \mathcal{T}_\mathcal{I}(X) = B_{\mathcal{I}} \times (\mathcal{P}_\omega X)^A \), namely \((X, (\overline{\sigma}_\mathcal{I}, id) \circ \delta : X \to B_{\mathcal{I}} \times (\mathcal{P}_\omega X)^A)\), where the output operation \(\overline{\sigma}_\mathcal{I} : (\mathcal{P}_\omega X)^A \to B_{\mathcal{I}} \times (\mathcal{P}_\omega X)^A\) provides the observations of interest corresponding to the original LTS and depending on the equivalence we want to study. (At this point, \(B_{\mathcal{I}}\) represents an arbitrary semilattice with a \(\lor\) operation, instantiated for each of the semantics under consideration as in [12].) Then, we determinize the decorated LTS, as depicted in Figure 1.

![Fig. 1. The powerset construction for decorated LTSs.](image)

Note that both the output operation and its image are parameterized by \(\mathcal{I} \in \{\mathcal{R}, \mathcal{F}, \mathcal{T}, \mathcal{CT}\}\), depending on the type of decorated trace semantics under consideration. The explicit instantiations of \(\overline{\sigma}_\mathcal{I}\) and \(B_{\mathcal{I}}\) for ready semantics are provided in Section 4, where we will also show that the coalgebraic modelling in fact coincides with the original definition of ready equivalence. (Note that the same result can easily be derived in the same style also for the case of trace, complete trace and failure semantics.) A fact that was not formally shown in [12].

The coalgebraic representation of ready, failure, trace and complete trace models as illustrated in Fig. 1 enables the definition of the corresponding equivalences as Moore bisimulations (i.e., bisimulations for a functor \(M = B_{\mathcal{I}} \times X^A\)). This way, checking behavioural equivalence of \(x_1\) and \(x_2\) reduces to checking the equality of their unique representatives in the final coalgebra: \(\llbracket\{x_1\}\rrbracket\) and \(\llbracket\{x_2\}\rrbracket\).

Moreover, it is worth observing that when reasoning on behavioural equivalence it is preferable to use relations as small as possible, that are not necessarily bisimulations, but contained in a bisimulation relation. These relations are referred to as bisimulations up-to [11].
In what follows we exploit the generalized powerset construction summarized in Fig. 1 and get an extension of bisimulation up-to congruence in [2] to the context of decorated LTS’s determined in terms of Moore automata.

Let $L_{\text{dec}} = (X, (\sigma_I, \text{id}) \circ \delta : X \to B_I \times (\mathcal{P}_\omega X)^A)$ be a decorated LTS and $(\mathcal{P}_\omega X, \langle o, t \rangle : \mathcal{P}_\omega X \to B_I \times (\mathcal{P}_\omega X)^A)$ its associated Moore automaton, as in Fig. 1. A bisimulation up-to congruence for $L_{\text{dec}}$ is a relation $R \subseteq (\mathcal{P}_\omega X) \times (\mathcal{P}_\omega X)$ such that:

$$X_1 R X_2 \Rightarrow \begin{cases} o(X_1) = o(X_2) \\ (\forall a \in A). t(X_1)(a) \ c(R) t(X_2)(a) \end{cases} \quad (\spadesuit)$$

where $c(R)$ is the smallest equivalence relation which is closed with respect to set union and which includes $R$, defined as in [2].

Remark 3.1 Observe that by replacing $c(R)$ with $R$ in (\spadesuit) one gets the definition of Moore bisimulation.

Theorem 3.2 Any bisimulation up-to congruence for decorated LTS’s is included in a bisimulation relation.

Proof. The proof consists in showing that for any bisimulation up-to congruence $R$, $c(R)$ is a bisimulation relation (recall that $R \subseteq c(R)$). The result follows immediately by structural induction. \hfill \Box

Remark 3.3 Based on (1), (2) and Theorem 3.2, verifying behavioural equivalence of two states $x_1, x_2$ in a decorated LTS consists in identifying a bisimulation up-to congruence $R^c$ relating $\{x_1\}$ and $\{x_2\}$:

$$\llbracket \{x_1\} \rrbracket = \llbracket \{x_2\} \rrbracket \iff \{x_1\} R^c \{x_2\}. \quad (4)$$

Also note that Theorem 3.2 is not a very different, but useful generalization of Theorem 2 in [2] to the context of decorated LTS’s.

More insight on how to derive canonical representatives of decorated trace semantics and how to apply the bisimulation up-to congruence proof technique is provided in Section 4, for the case of ready semantics.

4 Ready semantics

In this section we show how the ingredients of Fig. 1 in Section 3 can be instantiated in order to provide a coalgebraic modelling of ready semantics, as introduced in [12]. Moreover, we prove that the resulting coalgebraic characterization of this semantics is equivalent to the original definition.

Consider an LTS $(X, \delta : X \to (\mathcal{P}_\omega X)^A)$ and recall that, for a function $\varphi : A \to \mathcal{P}_\omega X$, the set of actions enabled by $\varphi$ is given by

$$I(\varphi) = \{ a \in A \mid \varphi(a) \neq \emptyset \}. \quad (5)$$
For the particular case $\varphi = \delta(x)$, $I(\delta(x))$ denotes the set of all (initial) actions ready to be fired by $x \in X$.

Recall also that a ready pair of $x$ is a pair $(w, Z) \in A^* \times \mathcal{P}_\omega A$ such that $x \xrightarrow{w} y$ and $Z = I(\delta(y))$. We denote by $\mathcal{R}(x)$ the set of all ready pairs of $x$.

Intuitively, ready semantics identifies states in $X$ based on the actions $a \in A$ they can immediately trigger after performing a certain action sequence $w \in A^*$, i.e., based on their ready pairs. It was originally defined as follows:

**Definition 4.1** [$\mathcal{R}$-equivalence [14]] Let $(X, \delta: X \to (\mathcal{P}_\omega X)^A)$ be an LTS and $x, y \in X$ two states. States $x$ and $y$ are ready equivalent ($\mathcal{R}$-equivalent) if and only if they have the same set of ready pairs, that is $\mathcal{R}(x) = \mathcal{R}(y)$.

Next, we instantiate $\sigma_I$ of Fig. 1 to ready semantics, where $I = \mathcal{R}$.

First note that in the setting of ready semantics, the observations provided by the output operation, which we denote by $\sigma_\mathcal{R}$, refer to the sets of actions ready to be executed by the states of the LTS. Therefore, $\sigma_\mathcal{R}$ is defined as follows:

$$
\sigma_\mathcal{R}: (\mathcal{P}_\omega X)^A \to \mathcal{P}_\omega (\mathcal{P}_\omega A)
$$

$$
\sigma_\mathcal{R}(\varphi) = \{I(\varphi)\}.
$$

For the case $\varphi = \delta(x)$, where $x \in X$, it holds that:

$$
\sigma_\mathcal{R}(\delta(x)) = \{I(\delta(x))\} = \{\{a \in A \mid \delta(x)(a) \neq \emptyset\}\}.
$$

In this particular instance, $B_I = B_\mathcal{R} = \mathcal{P}_\omega (\mathcal{P}_\omega A)$ and the final Moore coalgebra

$$
((\mathcal{P}_\omega (\mathcal{P}_\omega A))^A, (\epsilon, (-)_\omega))
$$

associates to each state $\{x\}$ the set of action sequences $w \in A^*$ such that $x \xrightarrow{w} x'$, together with the sets of actions ready to be triggered by (all such) $x'$, for $x, x' \in X$.

Next, we will prove the equivalence between the coalgebraic modelling of ready semantics and the original definition, presented above. More explicitly, given an arbitrary LTS $(X, \delta: X \to (\mathcal{P}_\omega X)^A)$ and a state $x \in X$, we want to show that $\llbracket\{x\}\rrbracket$ is equal to $\mathcal{R}(x)$.

The first remark is that the behaviour of a state $x \in X$ is a function $\llbracket\{x\}\rrbracket: A^* \to \mathcal{P}_\omega (\mathcal{P}_\omega A)$, whereas $\mathcal{R}(x)$ is defined as a set of pairs in $A^* \times \mathcal{P}_\omega A$. However, this is no problem since the set of functions $A^* \to \mathcal{P}_\omega (\mathcal{P}_\omega A)$ and $\mathcal{P}(A^* \times \mathcal{P}_\omega A)$ are isomorphic.

The set of all ready pairs $\mathcal{R}(x)$ associated to $x \in X$ is equivalently represented by $\varphi^R_{\{x\}}$, where, for $w \in A^*$ and $Y \subseteq X$,

$$
\varphi^R_{Y}(A^* \to \mathcal{P}_\omega (\mathcal{P}_\omega A))
$$

$$
\varphi^R_{Y}(w) = \{Z \subseteq A \mid \exists y \in t(Y)(w) \land Z = I(\delta(y))\}
$$

At this point, showing the equivalence between the coalgebraic and the original


definition of ready semantics reduces to proving that

\[(\forall x \in X). [[\{x\}]] = \varphi^R_{\{x\}}.\]  \(6\)

Equality (6) is a direct consequence of the following theorem:

**Theorem 4.2** Let \((X, \delta: X \rightarrow (\mathcal{P} \omega X)^A)\) be an LTS. Then for all \(Y \subseteq X\) and \(w \in A^*\), \([[Y]](w) = \varphi^R_Y(w)\).

**Proof.** We proceed by induction on words \(w \in A^*\).

- **Base case.** \(w = \varepsilon\). Consider an arbitrary set \(Y \subseteq X\). We have:

\[[Y](\varepsilon) = o(Y) = \bigcup_{y \in Y} \{I(\delta(y))\}\]

\[\varphi^R_Y(\varepsilon) = \{Z \subseteq A | \exists y \in Y \land Z = I(\delta(y))\} \text{ (by def., } (\forall y \in Y). y \xrightarrow{\varepsilon} y)\]

\[= \bigcup_{y \in Y} \{I(\delta(y))\}\]

Hence, \([[Y]](\varepsilon) = \varphi^R_Y(\varepsilon)\), for all \(Y \subseteq X\).

- **Induction step.**

Consider \(w \in A^*\) and assume \([[Y]](w) = \varphi^R_Y(w)\), for all \(Y \subseteq X\). We want to prove that \([[Y]](aw) = \varphi^R_Y(aw)\), where \(a \in A\).

\[[Y](aw) = [[t(Y)(a)](w)\]

\[\varphi^R_Y(aw) = \{Z | \exists y \in t(Y)(aw) \land Z = I(\delta(y))\}\]

\[= \{Z | \exists y \in t(t(Y)(a))(w) \land Z = I(\delta(y))\}\]

\[= \varphi^R_{t(Y)(a)}(w)\]

By the induction hypothesis, it follows that \([[Y]](aw) = \varphi^R_Y(aw)\), for all \(Y \subseteq X\).
We have that \([[Y]](w) = \varphi^R_Y(w)\), for all \(Y \subseteq X\) and \(w \in A^*\). \(\square\)

**Example 4.3** In what follows we illustrate the equivalence between the co algebraic and the original definitions of ready semantics by means of an example. Consider the following LTS.

![LTS Diagram](image)

We write \(a^n\) to represent the action sequence \(aa \ldots a\) of length \(n \geq 1\), with
$n \in \mathbb{N}$. The set of all ready pairs associated to $p_0$ is:

$$R(p_0) = \{(\varepsilon, \{a\}), (a^n, \{a\}), (a^n, \{b\}), (a^n, \{c\}), (a^n, \{d\}),$$

$$(a^n, \emptyset), (a^n, \emptyset) \mid n \in \mathbb{N} \land n \geq 1\}.$$  

We can construct a Moore automaton, for $S = \{p_0, p_1, \ldots, p_5\}$,

$$(\mathcal{P}_\omega S, (o, t) : \mathcal{P}_\omega S \rightarrow \mathcal{P}_\omega (\mathcal{P}_\omega A) \times (\mathcal{P}_\omega S)^A)$$

by applying the generalized powerset construction on the LTS above. The automaton will have $2^6 = 64$ states. We depict the accessible part from state $\{p_0\}$, where the output sets are indicated by double arrows:

![Diagram](https://via.placeholder.com/150)

The output sets of a state $Y$ of the Moore automaton in Fig. 2 is the set of actions associated to a certain state $y \in Y$ which can immediately be performed. For example, process $p_0$ in the original LTS above is ready to perform action $a$, whereas $p_1$ can immediately perform $b$. Therefore it holds that $o(\{p_0\}) = \{\{a\}\}$ and $o(\{p_0, p_1\}) = \{\{a\}, \{b\}\}$.

At this point, by simply looking at the automaton in Fig. 2, one can easily see that the set of action sequences $w \in A^*$ the state $\{p_0\}$ can execute, together with the corresponding possible next actions equals $R(p_0)$. Therefore, the automaton generated according to the generalized powerset construction captures the set of all ready pairs of the initial LTS.

As we remarked in Section 3, ready equivalence of LTS’s can be established in terms of bisimulation up-to congruence on Moore automata with output in $\mathcal{P}_\omega(\mathcal{P}_\omega A)$, representing the sets of actions ready to be triggered.

Next, we will explain how one can reason on ready equivalence of two LTS’s, by constructing bisimulations up-to congruence on the associated Moore automata generated according to the powerset construction in Fig. 1.
Example 4.4 Consider the following LTS.

\[
\text{\begin{tikzpicture}
    \node (q0) at (0,0) {$q_0$};
    \node (q1) at (1,1) {$q_1$};
    \node (q2) at (2,0) {$q_2$};
    \node (q3) at (3,1) {$q_3$};
    \node (q4) at (4,0) {$q_4$};
    \node (q5) at (5,1) {$q_5$};
    \node (q6) at (6,0) {$q_6$};
    \node (q7) at (7,1) {$q_7$};
    \node (q8) at (3,-2) {$q_8$};
    \node (q9) at (5,-2) {$q_9$};
    \node (q10) at (7,-2) {$q_{10}$};

    \draw[->] (q0) -- (q1) node[midway,above] {$a$};
    \draw[->] (q1) -- (q2) node[midway,above] {$a$};
    \draw[->] (q2) -- (q3) node[midway,above] {$a$};
    \draw[->] (q3) -- (q4) node[midway,above] {$a$};
    \draw[->] (q4) -- (q5) node[midway,above] {$a$};
    \draw[->] (q5) -- (q6) node[midway,above] {$a$};
    \draw[->] (q6) -- (q7) node[midway,above] {$a$};
    \draw[->] (q7) -- (q8) node[midway,above] {$c$};
    \draw[->] (q8) -- (q9) node[midway,above] {$d$};
    \draw[->] (q9) -- (q10) node[midway,above] {$d$};
\end{tikzpicture}}
\]

It is easy to check that \(q_0\) and \(p_0\) have the same ready pairs, that is \(\mathcal{R}(q_0) = \mathcal{R}(p_0)\), where \(p_0\) is the state in the LTS of the previous example.

Since we have shown the coincidence between the original definition involving equality of ready pairs and the coalgebraic representation, we can now prove that \(q_0\) and \(p_0\) are ready equivalent by building a bisimulation up-to congruence relating \(\{p_0\}\) and \(\{q_0\}\).

First, we have to determinize the LTS above. We show below the accessible part of the determinized automaton starting from state \(\{q_0\}\):

\[
\begin{align*}
\{q_0\} &\xrightarrow{a} \{a\} \\
\{a\} \cup \{b\} &\xrightarrow{b} \{q_1, q_2, q_3, q_7\} \xrightarrow{b} \{\{a\}, \{b\}\} \\
\{a\} \cup \{b\} &\xrightarrow{c} \{q_4, q_5, q_6\} \xrightarrow{c} \{\{c\}, \{d\}\} \\
\{q_0, q_1, q_2, q_3, q_7\} &\xrightarrow{d} \{\emptyset\} \\
\{q_0, q_{10}\} &\xrightarrow{d} \{\emptyset\}
\end{align*}
\]

Fig. 3. Ready determinization when starting from \(\{q_0\}\).

The next step is to build a bisimulation up-to congruence \(R\) on the sets of states of the generated Moore automata in Fig. 2 and Fig. 3, such that \(\{(p_0), \{q_0\}\}\) \(\in R\).

We start by taking \(R = \{(\{p_0\}, \{q_0\}\}\) and check whether this is already a bisimulation up-to congruence, by considering the output values and transitions, and check whether no new states appear in \(c(R)\) in the process. If new pairs of states appear, we add them to \(R\) and repeat the process.

Eventually, we end-up with a bisimulation up-to congruence

\[
R = \{(\{p_0\}, \{q_0\}\), \(\{(p_0, p_1), \{q_1, q_2, q_3, q_7\}\), 
\(\{(p_2, p_3), \{q_4, q_5, q_6\}\), \(\{(q_4), \{q_8\}\), \(\{(p_5\}, \{q_9, q_{10}\}\}\)
\]

By construction \(\{(p_0), \{q_0\}\}\) \(\in R\), so by (4) it follows that \([\{p_0\}] = [[q_0]]\).

Note that \(R\) is not a bisimulation relation since \(\{p_0, p_1\} \xrightarrow{a} \{p_0, p_1\}\) and \(\{q_1, q_2, q_3, q_7\} \xrightarrow{a} \{q_0, q_1, q_2, q_3, q_7\}\) but \(\{(p_0, p_1)\}, \{q_0, q_1, q_2, q_3, q_7\}\) \(\notin R\). Nevertheless, observe that \(R\) is a bisimulation up-to congruence since

78
\[ (\{p_0, p_1\}, \{q_0, q_1, q_2, q_3, q_7\}) \in c(R): \]

\[
\begin{align*}
\{p_0, p_1\} & = \{p_0\} \cup \{p_0, p_1\} \\
c(R) \{q_0\} \cup \{p_0, p_1\} & = ((\{p_0\}, \{q_0\}) \in R) \\
c(R) \{q_0\} \cup \{q_1, q_2, q_3, q_7\} & = ((\{p_0, p_1\}, \{q_1, q_2, q_3, q_7\}) \in R) \\
& = \{q_0, q_1, q_2, q_3, q_7\}
\end{align*}
\]

Also observe that the bisimulation up-to congruence given above is one pair smaller than the Moore bisimulation relating the automata in Fig. 2 and Fig. 3, which would also include \((\{p_0, p_1\}, \{q_0, q_1, q_2, q_3, q_7\})\).

5 **Canonical representatives**

Given a decorated LTS \((X, \langle \overline{\delta}, \delta \rangle, id) \circ \delta\), we showed in the previous section how to construct a determinized decorated LTS \((\mathcal{P}_{\omega}X, \langle o, t \rangle)\). The map \([-] : \mathcal{P}_{\omega}X \rightarrow B_{\mathcal{I}}^{A^*}\) provides us with a canonical representative of the behaviour of each state in \(\mathcal{P}_{\omega}X\). The image \((C, \overline{\delta})\) of \((\mathcal{P}_{\omega}X, \langle o, t \rangle)\), via the map \([-] \), can be viewed as the minimization w.r.t. the equivalence \(\mathcal{I}\).

Recall that the states of the final coalgebra \((B_{\mathcal{I}}^{A^*}, \langle \epsilon, (\cdot)_a \rangle)\) are functions \(\varphi : A^* \rightarrow B_{\mathcal{I}}\), and that their decorations and transitions are given by the functions \(\epsilon : B_{\mathcal{I}}^{A^*} \rightarrow B_{\mathcal{I}}\) and \((\cdot)_a : B_{\mathcal{I}}^{A^*} \rightarrow (B_{\mathcal{I}}^{A^*})^A\), defined in Section 2. The states of the canonical representative \((C, \overline{\delta})\) are also functions \(\varphi : A^* \rightarrow B_{\mathcal{I}}\), i.e., \(C \subseteq B_{\mathcal{I}}^{A^*}\). Moreover, the function \(\overline{\delta} : C \rightarrow B_{\mathcal{I}} \times C^A\) is simply the restriction of \(\langle \epsilon, (\cdot)_a \rangle\) to \(C\), that means \(\overline{\delta}(\varphi) = \langle \varphi(\epsilon), (\varphi)_a \rangle\) for all \(\varphi \in C\).

Finally, it is interesting to observe that \(B_{\mathcal{I}}^{A^*}\) carries a semilattice structure (inherited by \(B_{\mathcal{I}}\)) and that \([-] : \mathcal{P}_{\omega}X \rightarrow B_{\mathcal{I}}^{A^*}\) is a semilattice homomorphism. From this observation, it is immediate to conclude that also \(C\) is a semilattice, but it is not necessarily freely generated, i.e., it is not necessarily a powerset.

6 **Conclusions and future work**

In this paper, we have proved that the coalgebraic characterizations of decorated trace semantics in [12] are equivalent with the corresponding standard definitions. More precisely, for the case of ready equivalence, we have shown that for a state \(x\) in a labelled transition system, the coalgebraic canonical representative \([\{x\}]\), given by determinization and finality, coincides with the classical semantics \(R(x)\) representing the ready pairs of \(x\). In addition, we have illustrated how to reason about decorated trace equivalence using coinduction, by constructing suitable bisimulations up-to congruence. This is a very efficient sound and complete proof technique, and represents an important step towards automated reasoning, as it opens the way for the use of, for instance, coinductive theorem provers such as CIRC [9]. Note that even though in this paper we provided explicit proofs and examples only for the case of ready equivalence, similar results can be easily derived.
in the same style for (complete) trace and failure semantics.

A similar idea of system determinization was also applied in [4], in a non-coalgebraic setting, for the case of testing semantics where must testing coincides with failure semantics in the absence of divergence. A coalgebraic characterization of the spectrum was also attempted in [7], in a somewhat ad hoc fashion. Connections with these works are still to be explored.

There are two possible directions for future works. On the one hand, we would like to investigate to what extent the coalgebraic treatment of decorated trace semantics can be applied in the context of probabilistic systems. On the other hand, we would like to understand how our approach can be combined with [3] to obtain a coinductive approach to denotational (linear-time) semantics of different kinds of processes calculi.

References


Syntactic Control of Interference and Concurrent Separation Logic

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract
At last year’s MFPS conference we introduced a revised version of Concurrent Separation Logic in which assertions are tagged with a “rely set” of variables assumed to be unmodified by other processes. We showed that this logic is compositional and sound with respect to an action trace semantics. The revision was motivated by a subtle issue concerning soundness of the original version of the logic, discovered by Ian Wehrman and Josh Berdine. The revised logic fixes this problem and also relaxes the Owicki-Gries constraints on variables, allowing shared variables to be protected by multiple resources rather than a single one, but requiring that a process writing to a shared variable must acquire all resources that protect it, while a process reading a shared variable need only acquire one such resource. This generalization brings concurrent separation logic closer in spirit to permission-based logics, although our formulation makes no explicit mention of permissions. At the same conference, Uday Reddy introduced a concurrent separation logic with static permissions for variables, generalizing John Reynolds’s ideas on syntactic control of interference to a concurrent setting. Here we show that there is an extremely close relationship between these two logics. Essentially, every provable assertion in Reddy’s logic corresponds to a provable assertion in CSL with the same semantic content; and every provable assertion in CSL corresponds to a multitude of assertions in Reddy’s logic, differing only in the choice of specific permission values. We show how to construct, for a given CSL derivation, a family of corresponding derivations in Reddy’s logic that differ only in inessential permission choices. These results also imply that one can establish soundness of Reddy’s logic by appealing to soundness of CSL, leading to a simpler soundness proof than the one given in Reddy’s original paper, which used an augmented form of action trace semantics.

Keywords: concurrency, shared memory, resources, separation logic, permissions

1 Introduction
Concurrent Separation Logic is a logic for fault-free partial correctness of shared-memory concurrent programs, combining separation logic [10] with Owicki-Gries inference rules for pointer-free shared-memory programs [8], as proposed by Peter O’Hearn [7]. The original Owicki-Gries and O’Hearn logics apply only to programs with rigid parallel structure, because of a constraint that “no other process modifies” certain variables, imposed as a side condition in the rule for conditional critical regions.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
In prior work we formulated a more general concurrent separation logic [3], using resource contexts in an attempt to avoid these limitations. We gave a semantic formalization of O'Hearn’s notion of “ownership transfer” based on resource invariants, and of O'Hearn’s principle that provable processes “mind their own business” [7]. Subsequently, Ian Wehrman and Josh Berdine [12] discovered that the soundness analysis in [3] contains a hidden assumption tantamount to “no other process modifies”, so this logic was also only sound for rigid programs.

In response, at last year’s MFPS [5] we introduced a revised version of this logic in which assertions are augmented with a “rely set” representing a set of variables deemed to be left unmodified by “other” processes. By making this set an integral part of assertions, we avoid the need for a non-compositional side condition. The revised logic, which we call CSL, deals with these no-modifies assumptions in a syntax-directed and modular manner, without forcing the prover to know the rest of the program in advance. In addition the revised logic relaxes the Owicki-Gries constraints on use of critical variables, allowing such variables to be protected by several resources instead of a single one, thus embodying a more general protection discipline. A process writing a shared variable must acquire all resources that protect it, whereas reading a shared variable merely requires acquisition of some resource that protects it. We proved soundness of the revised CSL, this time without the hidden assumption and without requiring rigid program structure [5].

At the same MFPS Uday Reddy presented a permission-based version of concurrent separation logic, in which variables (but not heap cells) are given statically controlled permissions. This material appeared later as a joint paper with John Reynolds [9], including an algorithm (due to Reynolds) for inferring permission assignments. This work was motivated by a desire to generalize earlier ideas of Reynolds on syntactic control of interference to the concurrent setting. Reddy’s assertions deal elegantly with statically scoped permission contexts, and his formulation elides some of the syntactic side conditions that govern CSL rules dealing with variables, instead replacing them with implicit well-formedness constraints on the syntax of judgements. In this logic, writing to a shared variable requires acquisition of “total” permission for that variable, while reading is allowed with any permission. Static permissions, used this way, enforce syntactic control of interference: every provable program is free of race conditions involving variables (because of permission constraints), and free of races involving heap cells (by judicious use of separate conjunction).

Reddy commented [9] that there seems to be a close relationship between CSL and static permissions, and we made a similar observation at the time [5]. Indeed we deliberately used similar terminology to describe the disciplines for shared variable usage in the two frameworks: total permission surely seems “equivalent” to possession of all protecting resources, since both preclude any other process from interfering. CSL keeps track of the set of resources owned by processes, whereas Reddy’s logic maintains book-keeping information on the amount of permission for variables gathered by processes as they acquire and release resources.

In this paper we make a precise and rigorous connection between the two logics,
confirming these intuitions. Informally, each provable assertion in Reddy’s logic corresponds to a provable assertion in CSL; and a provable CSL assertion corresponds to a (non-empty) set of provable assertions in Reddy’s logic, differing only in essentially irrelevant details concerning specific choices of permission values. To establish these claims formally we first summarize some of the key concepts and definitions from the two logics. We make a careful analysis of the way proof derivations work in the two logics. We will also make clear what we mean by irrelevant permission decisions.

We assume familiarity with separation logic, as defined by Reynolds [10]. The reader can consult [5] for semantic foundations. To make this paper self-contained we recapitulate some material from [5] and [9].

2 Syntax

The syntax of commands (or processes) is given by the following abstract grammar:

$$C ::= \text{skip} \mid i := e \mid [e] := e' \mid i := \text{cons } E \mid \text{dispose } e$$

$$\mid C_1; C_2 \mid \text{if } b \text{ then } C_1 \text{ else } C_2 \mid \text{while } b \text{ do } C$$

$$\mid \text{with } r \text{ when } b \text{ do } C \mid \text{resource } r \text{ in } C \mid C_1 || C_2$$

where \(e\) ranges over integer expressions, \(b\) over boolean expressions, and \(E\) over list expressions of form \([e_1, \ldots, e_n]\). Expressions are pure, i.e. independent of the heap. Further, \(i\) ranges over identifiers (or program variables) and \(r\) over resource names; resources behave like binary semaphores. We use the standard abbreviation \(\text{with } r \text{ do } C\) for \(\text{with } r \text{ when true do } C\).

Let \(\text{free}(C) \subseteq \text{Ide}\) be the set of identifiers with a free occurrence in \(C\). Let \(\text{mod}(C)\) be the set of identifiers with a free write occurrence in \(C\), defined as usual, by structural induction:

\[
\text{mod}(\text{skip}) = \{\}
\]
\[
\text{mod}(i := e) = \text{mod}(i := \text{cons } E) = \text{mod}(i := [e]) = \{i\}
\]
\[
\text{mod}([e] := e') = \text{mod}(\text{dispose } e) = \{\}
\]
\[
\text{mod}(C_1; C_2) = \text{mod}(C_1 || C_2) = \text{mod}(C_1) \cup \text{mod}(C_2)
\]
\[
\text{mod}(\text{if } b \text{ then } C_1 \text{ else } C_2) = \text{mod}(C_1) \cup \text{mod}(C_2)
\]
\[
\text{mod}(\text{while } b \text{ do } C) = \text{mod}(C)
\]
\[
\text{mod}(\text{with } r \text{ when } b \text{ do } C) = \text{mod}(C)
\]
\[
\text{mod}(\text{resource } r \text{ in } C) = \text{mod}(C)
\]
3 Concurrent Separation Logic

As in Owicki-Gries [8], we associate to each resource name \( r \) a set \( X \subseteq \text{Ide} \) of “protected variables” and a “resource invariant” \( R \) [6], which is required to be a precise separation logic formula as in [7]. A separation logic formula \( R \) is precise iff, for all stores \( s \) and heaps \( h \), there is at most one sub-heap \( h' \subseteq h \) such that \( (s, h') \models R \). Instead of assuming a statically fixed association of resource invariants and protection sets to resource names, we extend the syntax of partial correctness assertions to include a resource context, as in [3]. We also relax the variable usage constraints from the earlier logics by not insisting that protection sets be pairwise disjoint.

Definition 3.1 A well-formed resource context has the form

\[
    r_1(X_1) : R_1, \ldots, r_n(X_n) : R_n
\]

where \( r_1, \ldots, r_n \) are distinct resource names, \( X_1, \ldots, X_n \) are sets of identifiers, each \( R_i \) is precise, and \( \text{free}(R_i) \subseteq X_i \) for each \( i \).

We let \( \Gamma \) range over the set of well-formed resource contexts. We say that \( r \) protects \( x \) in \( \Gamma \) when \( r(X) : R \) is in \( \Gamma \) and \( x \in X \). Let \( \text{owned}(\Gamma) = \bigcup_{i=1}^{n} X_i \), and \( \text{inv}(\Gamma) = R_1 * \cdots * R_n \). Let \( \text{dom}(\Gamma) \) be \( \{ r_1, \ldots, r_n \} \). We let \( \Gamma, r(X) : R \) be the context that combines \( \Gamma \) with \( r(X) : R \), when well-formed.

Definition 3.2 A well-formed CSL assertion has form

\[
    \Gamma \vdash A \{ P \} C \{ Q \},
\]

where \( A \) is a (finite) set of identifiers, \( \Gamma \) is a well-formed resource context, \( \text{free}(P, Q) \subseteq A \), and \( \text{free}(C) \subseteq \text{owned}(\Gamma) \cup A \).

In an assertion \( \Gamma \vdash A \{ P \} C \{ Q \} \) we refer to \( A \) as the rely set, \( P \) as the pre-condition, and \( Q \) as the post-condition. The constraints allow \( P \) and \( Q \) to mention identifiers owned by resources in \( \Gamma \), but only if they belong to the rely set; the command \( C \) may use variables owned by resources or belonging to the rely set. The rules use rely sets to keep track of variables used (outside of critical regions, so without protection): for soundness and the avoidance of race conditions such variables must not be modified by any other process, and this requirement is enforced as a side condition in the CSL parallel rule. The CSL inference rules will constrain where \( C \) is allowed to read and write protected variables: every write in \( C \) to a protected variable must be inside (nested) critical regions naming all resources that protect it; and every read occurrence in \( C \) of a protected variable must be inside a critical region naming some resource that protects it. In the special case where the protection sets are pairwise disjoint, this coincides with the usual Owicki-Gries discipline.

Definition 3.3 The assertion \( \Gamma \vdash A \{ P \} C \{ Q \} \) is valid iff every finite interactive computation of \( C \), from a state (with values for all variables in \( \Gamma, A \)) satisfying
P \ast \mathit{inv}(\Gamma)$, in an environment that respects $\Gamma$ and does not write to variables in $A$, is fault-free, respects $\Gamma$, and ends in a state satisfying $Q \ast \mathit{inv}(\Gamma)$.

Respect for $\Gamma$ means obedience to the protection regime and preservation of each resource invariant (separately). Fault-freedom means no runtime errors such as dangling pointers, and no race conditions involving concurrent writes to shared variables or heap cells.

Validity of $\Gamma \vdash_A \{P\}C\{Q\}$ also implies fault-free partial correctness: every terminating execution of $C$ in isolation, from a state satisfying $P \ast \mathit{inv}(\Gamma)$, is fault-free and ends in a final state satisfying $Q \ast \mathit{inv}(\Gamma)$. This is because the empty environment vacuously respects $\Gamma$ and does not write to any variable.

The CSL rules for assignment, parallel composition, critical regions, and local resource blocks are summarized below, together with the Frame rule and the Rule of Consequence, which allows weakening of post-conditions and strengthening of pre-conditions as usual and allows expansion of a rely set.

Only well-formed instances of these rules are allowed.

- **Assignment**

  $\Gamma \vdash_A \{e/i\}P_i := e\{P\}$ if $i \notin \text{owned}(\Gamma)$, $\text{free}(P, e) \subseteq A$

- **Parallel**

  $\Gamma \vdash_{A_1 \cup A_2} \{P_1 \ast P_2\}C \{Q_1 \ast Q_2\}$

  if $\text{mod}(C_1) \cap \text{free}(C_2) \subseteq \text{owned}(\Gamma)$, $\text{mod}(C_2) \cap \text{free}(C_1) \subseteq \text{owned}(\Gamma)$, $\text{mod}(C_1) \cap A_2 = \{\}$ and $\text{mod}(C_2) \cap A_1 = \{\}$.

- **Region**

  $\Gamma, r(X) : R \vdash_A \{P\} \text{with } r \text{ when } b \text{ do } C\{Q\}$

- **Resource**

  $\Gamma, r(X) : R \vdash_A \{P\}C\{Q\}$

- **Frame**

  $\Gamma \vdash_A \{P\}C\{Q\}$

  $\Gamma \vdash_{A \cup \text{free}(R)} \{P \ast R\}C\{Q \ast R\}$ if $\text{mod}(C) \cap \text{free}(R) = \{\}$

- **Consequence**

  $\Gamma \vdash_A \{P\}C\{Q\}$

  $\Gamma \vdash_{A'} \{P'\}C\{Q'\}$ if $A \subseteq A'$, $P' \Rightarrow P$, $Q \Rightarrow Q'$

In **Assignment** note that well-formedness implies that $i \in A$.

In **Parallel**, the side condition ensures that neither process modifies any identifiers in the other process’s rely set.
In Region the premiss relies on $A \cup X$ because mutual exclusion for $r$ implies that the identifiers in $X$ cannot be modified by other processes while the current process is inside the critical region.

In Resource the conclusion relies on $A \cup X$, which ensures well-formedness of the conclusion whenever the premiss is well-formed, because $\text{free}(R) \subseteq X$.

In Frame, as usual, $C$ must not write to any identifier occurring free in $R$. There is no need to insist, as was done in [3], that $\text{free}(R) \cap \text{owned}(\Gamma) = \{\}$; instead we add the variables occurring free in $R$ to the rely set, reflecting the assumption that no concurrent processes modify these variables.

4 Example

Here is an example that facilitates the coming comparison with Reddy’s logic.

The following assertion is provable in CSL. Note the rely set $\{x\}$.

$$\vdash_{\{x\}} \{42 \mapsto \_\}$$

resource $r_1$ in

resource $r_2$ in

$$\begin{align*}
&\text{(with } r_1 \text{ do (with } r_2 \text{ do } x:=1); [42]:=1) \\
&\| \text{(with } r_2 \text{ do (with } r_1 \text{ do } x:=2); [42]:=2) \\
&\{42 \mapsto \_\}
\end{align*}$$

Validity of this assertion implies that this program, when executed from a state in which $x$ has a value and 42 is a heap cell, is error-free, provided no other process modifies $x$; in particular, there is no race condition involving $x$, and no race condition involving the heap cell. Of course, the rely assumption is needed: if the code is run concurrently with a process that modifies $x$ there could be a race condition. Let $C_1$ and $C_2$ be:

$$C_1 :: \text{with } r_1 \text{ do (}(\text{with } r_2 \text{ do } x:=1); [42]:=1) \text{) }$$

$$C_2 :: \text{with } r_2 \text{ do (}(\text{with } r_1 \text{ do } x:=2); [42]:=2) \text{).}$$

Let $R_1$ and $R_2$ be the assertions

$$R_1 :: (x = 1 \land 42 \mapsto 1) \lor (x = 2 \land \text{emp})$$

$$R_2 :: (x = 1 \land \text{emp}) \lor (x = 2 \land 42 \mapsto 2).$$

As shown in [5] the following assertions are derivable in CSL:

$$r_1(x) : R_1, r_2(x) : R_2 \vdash_{\{} \{\text{emp}\} C_1 \{\text{emp}\} \}$$

$$r_1(x) : R_1, r_2(x) : R_2 \vdash_{\{} \{\text{emp}\} C_2 \{\text{emp}\}. \}$$
By Parallel we can then get
\[ r_1(x) : R_1, r_2(x) : R_2 \vdash \{\text{emp}\} C_1 \parallel C_2 \{\text{emp}\} \]
and finally, by Resource,
\[ \vdash \{\text{emp}\} (R_1 \ast R_2) \text{resource } r_1 \text{ in resource } r_2 \text{ in } (C_1 \parallel C_2) \{R_1 \ast R_2\}. \]

This derivation employs a resource context \( r_1(x) : R_1, r_2(x) : R_2 \) in which the protection lists are not disjoint.

## 5 Reddy’s logic

Reddy’s version of Concurrent Separation Logic uses static permissions to ensure proper usage of shared variables.

Following [1], a permission algebra \((P, \oplus, \top)\) is a partial commutative, cancellative semi-group such that
\[
\begin{align*}
\forall p_1, p_2 \in P. & \quad p_1 \oplus p_2 \neq p_2 \quad \text{(non-zero)} \\
\forall p \in P. & \quad p \oplus \top \text{ is undefined} \quad \text{(top)} \\
\forall p \in P. & \quad \exists p_1, p_2 \in P. \quad p = p_1 \oplus p_2 \quad \text{(divisibility)}
\end{align*}
\]

A permission context \( \Sigma \) has the form \( x_1^{p_1}, \ldots, x_n^{p_n} \) and is well-formed if and only if whenever \( x \) occurs multiple times in \( \Sigma \), with permissions \( p_i, \ldots, p_k \), then \( p_i \oplus \cdots \oplus p_k \) is defined. We write \( \Sigma_1, \ldots, \Sigma_n \) for the obvious composite context combining the permission entries of the \( \Sigma_i \), when this is well-formed.

Let \( \text{dom}(\Sigma) = \{ x \in \text{Ide} \mid \exists p. x^p \in \Sigma \} \) and, for \( x \in \text{dom}(\Sigma) \), let \( \Sigma(x) \) be the sum of all permissions occurring for \( x \) in the entries of \( \Sigma \). Let \( |\Sigma| = \{(x, \Sigma(x)) \mid x \in \text{dom}(\Sigma)\} \). Permission contexts \( \Sigma_1 \) and \( \Sigma_2 \) are equivalent iff they give the same permissions to all identifiers, i.e. when \( |\Sigma_1| = |\Sigma_2| \).

Reddy introduces syntactic judgements, defined with structural inference rules, with the following forms:
- \( \Sigma \vdash i \text{ Var} \)
- \( \Sigma \vdash e \text{ Exp} \)
- \( \Sigma \vdash P \text{ Assert} \)

In each case \( \Sigma \) is required to be a well-formed permission context. Using validity here to mean derivability from Reddy’s inference rules, the following key facts are easy to establish:
- If \( \Sigma \vdash i \text{ Var} \) is valid, then \( i \in \text{dom}(\Sigma) \) and \( \Sigma(i) = \top \).
- If \( \Sigma \vdash e \text{ Exp} \) is valid, then \( \text{free}(e) \subseteq \text{dom}(\Sigma) \).
- If \( \Sigma \vdash P \text{ Assert} \) is valid, then \( \text{free}(P) \subseteq \text{dom}(\Sigma) \).
Although Reddy’s paper does not provide a full set of inference rules for these judgements, it seems reasonable to assume that the converse implications also hold, since the purpose of these syntactic judgements is to formalize the indicated static constraints on variables and permissions.

Reddy also introduces *permissive resource contexts*, of the form

$$r_1(\Sigma_1) : R_1, \ldots, r_n(\Sigma_n) : R_n$$

where $r_1, \ldots, r_n$ are distinct resource names, $\Sigma_1, \ldots, \Sigma_n$ is a well-formed permission context, each $R_i$ is precise, and for each $i$, $\Sigma_i \vdash R_i$ Assert is valid. We will use $\Delta$ to range over (well-formed) permissive resource contexts.

Reddy’s proof system uses assertions of the form

$$\Sigma \mid r_1(\Sigma_1) : R_1, \ldots, r_n(\Sigma_n) : R_n \vdash \{P\}C\{Q\},$$

subject to the constraint that $\Sigma, \Sigma_1, \ldots, \Sigma_n$ is a well-formed permission context, $r_1, \ldots, r_n$ are distinct resource names, each $R_i$ is precise, and $\Sigma_i \vdash R_i$ Assert is valid, for each $i$. Note that this implies that for each $i$, $\text{free}(R_i) \subseteq \text{dom}(\Sigma_i)$.

The Reddy rules for assignment, parallel composition, critical regions, and local resource blocks are listed below, together with the Frame Rule. There are also rules for the other program constructs and a Rule of Consequence. We use the same labels as for the corresponding CSL rules. Only well-formed instances are allowed.

- **Assignment**

  $$\Sigma \vdash i \text{ Var} \quad \Sigma \vdash e \text{ Exp} \quad \Sigma \vdash \{P\}C\{Q\}$$

  $$\Sigma \mid \Delta \vdash \{[e/i]P\}i := e\{P\}$$

- **Parallel**

  $$\Sigma_1 \mid \Delta \vdash \{P_1\}C_1\{Q_1\} \quad \Sigma_2 \mid \Delta \vdash \{P_2\}C_2\{Q_2\}$$

  $$\Sigma_1, \Sigma_2 \mid \Delta \vdash \{P_1 \ast P_2\}C_1\|C_2\{Q_1 \ast Q_2\}$$

- **Region**

  $$\Sigma \vdash P, Q \text{ Assert} \quad \Sigma, \Sigma_0 \mid \Delta \vdash \{(P \ast R) \land b\}C\{Q \ast R\}$$

  $$\Sigma \mid \Delta, r(\Sigma_0) : R \vdash \{P\} \text{with } r \text{ when } b \text{ do } C\{Q\}$$

- **Resource**

  $$\Sigma_0 \vdash R \text{ Assert} \quad \Sigma, \Sigma_0 \mid \Delta \vdash \{P \ast R\}r(\Sigma_0) \text{ resource } r(\Sigma_0) \text{ in } C\{Q \ast R\}$$

- **Frame**

  $$\Sigma \mid \Delta \vdash \{P\}C\{Q\} \quad \Sigma' \vdash R \text{ Assert}$$

  $$\Sigma, \Sigma' \mid \Delta \vdash \{P \ast R\}C\{Q \ast R\}$$

The implicit well-formedness constraints on these rules codify many of the static side conditions that occur in the corresponding CSL rules. For example, in **Frame** the premiss $\Sigma' \vdash R \text{ Assert}$ implies that $\text{free}(R) \subseteq \text{dom}(\Sigma')$, and when $\Sigma \mid \Delta \vdash \{P\}C\{Q\}$ is provable and $\Sigma, \Sigma'$ is well-formed this implies $\text{mod}(C) \cap \text{free}(R) = \{\}$, the side condition imposed in the CSL Frame rule.

---

1 Reddy uses $\Gamma$ for permissive contexts, but we prefer $\Delta$ to avoid confusion with CSL.
We should point out a subtle issue that differentiates Reddy’s set-up from ours. Although this may seem to be only a minor difference, it has significant consequences. In Reddy’s framework we must attach permission contexts to local resource names. This is evident in the Resource rule: unless $|\Sigma_0| = |\Sigma_1|$, the decorated programs $\text{resource } r(\Sigma_0) \text{ in } C$ and $\text{resource } r(\Sigma_1) \text{ in } C$ are logically distinguishable, since they are usable in different proof contexts. To prove $\Sigma' \mid \Delta \vdash \{P\}$-resource $r(\Sigma_0)$ in $C\{Q\}$ it must be possible to “slice out” $\Sigma_0$ from $\Sigma'$, whereas it may not be possible to slice out $\Sigma_1$ instead. Each of these judgements concerns a decorated version of the same original program, and in establishing connections between the two logics we should distinguish between commands (in the original programming language) and decorated commands that arise in this manner. We will use $C', C''$ to range over decorated commands, $C$ over commands; and we say that $C'$ is a decoration of $C$ if $C$ is obtained from $C'$ by erasing permission contexts. The rules of Reddy’s logic presented above should really employ $C'$ rather than $C$ as meta-variable, since the judgements involve decorated commands rather than commands, although we refrain from re-stating them in amended form.

Obviously we can characterize the decoration relationship by structural induction. In particular, if $C$ contains no local resource blocks then the only decoration of $C$ is $C$ itself; $C''$ is a decoration of $\text{resource } r \text{ in } C$ if and only if $C''$ has form $\text{resource } r(\Sigma) \text{ in } C'$, where $C'$ is a decoration of $C$. Similarly $C'_1; C'_2$ is a decoration of $C_1; C_2$ if and only if $C'_1$ is a decoration of $C_1$ and $C'_2$ is a decoration of $C_2$; and $C'_1 || C'_2$ is a decoration of $C_1 || C_2$ if and only if $C'_1$ is a decoration of $C_1$ and $C'_2$ is a decoration of $C_2$. It is obvious that when $C'$ is a decoration of $C$, $\text{mod}(C') = \text{mod}(C)$ and $\text{free}(C') = \text{free}(C)$.

6 Example

Let $P = (0, 1]$ be fractional permissions, with $p_1 \oplus p_2 = p_1 + p_2$ iff $p_1 + p_2 \leq 1$, and $\top = 1$, as in [2]. One can derive the following assertion in Reddy’s logic, as shown in [9]:

$\vdash \{42 \mapsto \_\}$

$\text{resource } r_1(x^{1/2}) \text{ in }$

$\text{resource } r_2(x^{1/2}) \text{ in }$

$(\text{with } r_1 \text{ do (with } r_2 \text{ do } x:=1); [42]:=1)$

$\| (\text{with } r_2 \text{ do (with } r_1 \text{ do } x:=2); [42]:=2)$

$\{42 \mapsto \_\}$
But equally well for each pair \((p, q)\) such that \(0 < p, q < 1\) and \(p + q = 1\) one can derive

\[\vdash \{42 \mapsto \_\}\]

\[\text{resource } r_1(x^p) \text{ in }\]

\[\text{resource } r_2(x^q) \text{ in }\]

\[(\text{with } r_1 \text{ do (with } r_2 \text{ do } x:=1); [42]:=1)\]

\[\parallel (\text{with } r_2 \text{ do (with } r_1 \text{ do } x:=2); [42]:=2)\]

\[\{42 \mapsto \_\}\]

and in each case the derivations are essentially the same, up to minor details concerning fractions.

The results of the next section will establish that all of these derivations correspond in a precise manner to the single CSL derivation shown earlier as an example.

## 7 Connecting the logics

We first show that every provable judgement in Reddy’s logic corresponds to an assertion provable in CSL. To start, we note without proof following general properties, which are echoed in Reddy’s development.

**Theorem 7.1**

(i) If \(\Sigma \mid r_1(\Sigma_1) : R_1, \ldots, r_n(\Sigma_n) : R_n \vdash \{P\}C'\{Q\}\) is provable, and \(x \in \text{free}(C')\), then \(x \in \text{dom}(\Sigma, \Sigma_1, \ldots, \Sigma_n)\).

(ii) If \(\Sigma \mid r_1(\Sigma_1) : R_1, \ldots, r_n(\Sigma_n) : R_n \vdash \{P\}C'\{Q\}\) is provable, and \(x \in \text{mod}(C')\), then \((\Sigma, \Sigma_1, \ldots, \Sigma_n)(x) = \top\).

When \((\Sigma, \Sigma_1, \ldots, \Sigma_n)\) is a tuple of permission contexts we can construct a rely set and a tuple of protection lists by applying \text{dom} to each component and rearranging, to get \((\text{dom}(\Sigma), (\text{dom}(\Sigma_1), \ldots, \text{dom}(\Sigma_n)))\). Similarly, given a (well-formed) Reddy context \(\Delta\) we can construct a (well-formed) CSL resource context by applying \text{dom} inside each term of \(\Delta\), leaving resource names and invariants unchanged. When \(\Delta\) has the form \(r_1(\Sigma_1) : R_1, \ldots, r_n(\Sigma_n) : R_n\) this produces the resource context \(\Gamma\) of form \(r_1(\text{dom } \Sigma_1) : R_1, \ldots, r_n(\text{dom } \Sigma_n) : R_n\). Let us write \(\Gamma = \text{map dom } \Delta\) when this relationship holds.

**Theorem 7.2**

If \(\Sigma \mid \Delta \vdash \{P\}C'\{Q\}\) is provable in Reddy’s logic, \(C'\) is a decoration of \(C\), \(\Gamma = \text{map dom } \Delta\) and \(A = \text{dom}(\Sigma)\), then \(\Gamma \vdash_A \{P\}C\{Q\}\) is provable in CSL.

Proof: by induction on the proof height of the derivation. We show that for each (well-formed instance of an) inference rule in Reddy logic, if the relevant side conditions hold and the premisses are provable in Reddy’s logic, then the CSL assertion corresponding to the rule’s conclusion is provable from the CSL assertions representing the rule’s premisses. We sketch the details for the rules listed above.
We make use of Theorem 7.1 in various places.

- Every well-formed instance of the Reddy Assignment rule has the form

\[
\begin{align*}
\Sigma & \vdash P \text{ Assert} & \Sigma & \vdash \text{e Exp} & \Sigma & \vdash e \text{ Var} \quad \Sigma \mid \Delta \vdash \{[e/i]P\}i:=e\{P\}
\end{align*}
\]

Since we assume that the premisses are provable, we have \(|\Sigma|(i) = T\), free(e) \(\subseteq\) dom(\(\Sigma\)), and free(P) \(\subseteq\) dom(\(\Sigma\)). Let \(A = \text{dom}(\Sigma)\) and let \(\Gamma = \text{map dom}\ \Delta\) be the CSL resource context determined by \(\Delta\). By well-formedness of \(\Sigma, \Delta\) it follows that \(i \not\in \text{owned}(\Gamma)\). So the side conditions for the appropriate instance of the CSL Assignment rule hold, and the CSL assertion \(\Delta \vdash A \{[e/i]P\}i:=e\{P\}\) is provable.

- Consider a well-formed instance of Parallel, where \(C'_1\mid C'_2\) is a decoration of \(C_1\mid C_2\):

\[
\begin{align*}
\Sigma_1 & \mid \Delta \vdash \{P_1\}C'_1\{Q_1\} & \Sigma_2 & \mid \Delta \vdash \{P_2\}C'_2\{Q_2\} & \Sigma_1, \Sigma_2 & \mid \Delta \vdash \{P_1 \ast P_2\}C'_1\mid C'_2\{Q_1 \ast Q_2\}
\end{align*}
\]

Let \(A_1 = \text{dom}(\Sigma_1)\), \(A_2 = \text{dom}(\Sigma_2)\), \(\Gamma = \text{map dom}\ \Delta\). The CSL assertions corresponding to the premisses of this rule are then \(\Gamma \vdash \Delta A_1 \{P_1\}C_1\{Q_1\}\) and \(\Gamma \vdash \Delta A_2 \{P_2\}C_2\{Q_2\}\). By Theorem 7.1 and well-formedness it follows that \(\text{mod}(C_2) \cap A_1 = \{\}\), \(\text{mod}(C_1) \cap A_2 = \{\}\), \(\text{mod}(C_1) \cap \text{free}(C_2) \subseteq \text{owned}(\Gamma)\), and \(\text{mod}(C_1) \cap \text{free}(C_1) \subseteq \text{owned}(\Gamma)\). So by the CSL Parallel rule we get \(\Gamma \vdash \Delta A_1 \cup A_2 \{P_1 \ast P_2\}C_1\mid C_2\{Q_1 \ast Q_2\}\). This is the CSL assertion corresponding to \(\Sigma_1, \Sigma_2 \mid \Delta \vdash \{P_1 \ast P_2\}C'_1\mid C'_2\{Q_1 \ast Q_2\}\).

- Consider a well-formed instance of Region in which \(C'\) is a decoration of \(C\):

\[
\begin{align*}
\Sigma & \vdash P, Q \text{ Assert} & \Sigma, \Sigma_0 \mid \Delta \vdash \{(P \ast R) \land b\}C'\{Q \ast R\} & \Sigma \mid \Delta, \text{r}(\Sigma_0) : R \vdash \{P\} \text{ with r when b do C'\{Q\}}
\end{align*}
\]

Let \(A = \text{dom}(\Sigma)\), \(X = \text{dom}(\Sigma_0)\), and \(\Gamma = \text{map dom}\ \Delta\). The CSL assertion corresponding to the (second) premiss is \(\Gamma \vdash \Delta A \cup X \{(P \ast R) \land b\}C\{Q \ast R\}\). Using the CSL Region rule we can deduce

\[
\Gamma, \text{r}(X) : R \vdash A \{P\} \text{ with r when b do C\{Q\}}.
\]

This is the CSL assertion corresponding to the conclusion of the Reddy rule, and is well-formed; the first premiss implies free(P, Q) \(\subseteq\) A.

- Consider a well-formed instance Resource in which \(C'\) decorates \(C\):

\[
\begin{align*}
\Sigma_0 & \vdash R \text{ Assert} & \Sigma \mid \Delta, \text{r}(\Sigma_0) : R \vdash \{P\}C'\{Q\} & \Sigma, \Sigma_0 \mid \Delta \vdash \{P \ast R\} \text{ resource r(\Sigma_0) in C'\{Q \ast R\}}
\end{align*}
\]

Assume that the premisses are provable. Let \(A = \text{dom}(\Sigma)\), \(X = \text{dom}(\Sigma_0)\), \(\Gamma = \text{map dom}\ \Delta\). Then free(R) \(\subseteq\) X, because \(\Sigma_0 \vdash R \text{ Assert}\) is provable. The CSL assertion determined by the second premiss is

\[
\Gamma, \text{r}(X) : R \vdash A \{P\}C\{Q\}.
\]
Using the CSL Resource rule we can deduce from this the assertion

\[ \Gamma \vdash_{\text{Aux}} \{ P * R \text{ resource } r \text{ in } C \{ Q * R \}, \]

which corresponds to the conclusion in the Reddy rule.

- For the Frame rule suppose that \( \Sigma \mid \Delta \vdash \{ P \} C' \{ Q \} \) and \( \Sigma' \vdash R \) Assert are provable in Reddy’s logic, so that \( \Sigma, \Sigma' \mid \Delta \vdash \{ P * R \} C' \{ Q * R \} \) is provable from the Frame rule. Let \( C' \) decorate \( C \). Then \( \text{free}(R) \subseteq \text{dom}(\Sigma') \) and \( \text{mod}(C') \cap \text{dom}(\Sigma') = \{ \} \), using Theorem 7.1 again. Let \( A = \text{dom}(\Sigma) \) and \( \Gamma = \text{map} \text{ dom} \Delta \).

The CSL version of the premiss is \( \Gamma \vdash_{A} \{ P \} C' \{ Q \} \). From the above, we have \( \text{mod}(C) \cap \text{free}(R) = \{ \} \). So we can use the CSL Frame rule to derive \( \Gamma \vdash_{\text{Aux} \text{free}(R)} \{ P * R \} C' \{ Q * R \} \). Using Consequence we can then deduce \( \Gamma \vdash_{\text{Aux} \text{dom}(\Sigma')} \{ P * R \} C' \{ Q * R \} \), which corresponds as required to \( \Sigma, \Sigma' \mid \Delta \vdash \{ P * R \} C' \{ Q * R \} \).

(End of Proof)

To establish a converse connection between the logics we argue as follows.

For a tuple of identifier sets \( (A, X_1, \ldots, X_n) \) and a subset \( Y \subseteq A \cup \bigcup_{i=1}^{n} X_i \), the set of permission contexts \( \Sigma, \Sigma_1, \ldots, \Sigma_n \) such that \( \text{dom}(\Sigma) = A \), \( \text{dom}(\Sigma_i) = X_i \) for each \( i \), and \( (\Sigma, \Sigma_1, \ldots, \Sigma_n)(x) = \top \) for all \( x \in Y \), is non-empty. We use this fact to guide the choices of permission contexts when constructing a Reddy judgement \( \Sigma \mid \Delta \vdash \{ P \} C' \{ Q \} \) to match a CSL assertion \( \Gamma \vdash_{A} \{ P \} C' \{ Q \} \).

When \( \Delta \) is \( \text{r}_1(\Sigma_1) : R_1, \ldots, \text{r}_n(\Sigma_n) : R_n \) and \( \Sigma \) is a permission context we may use the abbreviation \( \Sigma, \Delta \) for the permission context \( \Sigma, \Sigma_1, \ldots, \Sigma_n \). We will also say that the combination \( \Sigma, \Delta \) is well-formed when this permission context is well-formed and \( \Delta \) is a well-formed permissive resource context.

**Theorem 7.3**

Let \( \Gamma \vdash_{A} \{ P \} C \{ Q \} \) be a provable assertion in CSL. Let \( \Sigma, \Delta \) be well-formed and suppose that \( \text{dom}(\Sigma) = A \), \( \text{map} \text{ dom} \Delta = \Gamma \), and for all \( x \in \text{mod}(C) \), \( (\Sigma, \Delta)(x) = \top \). Then there is a decoration \( C' \) of \( C \) such that the judgement \( \Sigma \mid \Delta \vdash \{ P \} C' \{ Q \} \) is provable in Reddy’s logic.

Proof: For simplicity we will assume that \( P \) is the fractional permissions algebra, although it is easy to adjust the proof details to encompass a general permissions algebra; in the general proof divisibility plays a crucial rôle, and with fractional permissions we can get by with division by 2.

The proof is by induction on proof height. We show that for each CSL inference rule, if the premises have this property and the side conditions hold, then the conclusion has this property. We give the details for assignment, for parallel composition (where division is needed), regions, and local resource blocks (where we must choose an appropriate decoration); the other rules are simpler and can be handled in similar manner.

**Assignment:** Consider a well-formed instance of the CSL assignment rule:

\[ \overline{\Gamma} \vdash_{A} \{ [e/i]P : = e \{ P \} \} \]

92
Parallel

For instance of Reddy’s assignment rule are valid, hence provable, and 
map dom Σ = A, |Σ|(i) = T, and 
map dom Δ = Γ. (The set of such combinations is non-empty.) Then we 
have Σ ⊢ i Var, Σ ⊢ e Exp, and Σ ⊢ P Assert. So the premisses of the relevant 
instance of Reddy’s assignment rule are valid, hence provable, and

\[ \Sigma | Δ ⊢ \{[e/i]P\}i:=e\{P\} \]

is provable.

• **PARALLEL:** Consider a well-formed instance of the CSL parallel rule:

\[
\frac{\Gamma \vdash A}{\Gamma \vdash A_1 \{P_1\}C_1\{Q_1\} \quad \Gamma \vdash A_2 \{P_2\}C_2\{Q_2\}} \quad \frac{\Gamma \vdash A_1 \cup A_2 \{P_1 * P_2\}C_1\|C_2\{Q_1 * Q_2\}}
\]

with \( mod(C_1) \cap A_2 = mod(C_2) \cap A_1 = \emptyset \), \( mod(C_1) \cap free(C_2) \subseteq owned(\Gamma) \) and 
\( mod(C_2) \cap free(C_1) \subseteq owned(\Gamma) \). We can choose Σ and Δ such that dom(Σ) = 
A_1 ∪ A_2, map dom Δ = Γ, and ∀x ∈ mod(C_1∩C_2), (Σ, Δ)(x) = T. Define permis-
sion contexts Σ_1 and Σ_2 as follows:

\[
\Sigma_1 = \{x^p ∈ Σ \mid x ∈ A_1 - A_2\} \cup \{x^{p/2} \mid x^p ∈ Σ, x ∈ A_1 ∩ A_2\}
\]

\[
\Sigma_2 = \{x^p ∈ Σ \mid x ∈ A_2 - A_1\} \cup \{x^{p/2} \mid x^p ∈ Σ, x ∈ A_1 ∩ A_2\}.
\]

Then |Σ| = |Σ_1, Σ_2|, and dom(Σ_1) = A_1, dom(Σ_2) = A_2. By assumption \( mod(C_1) \cap 
A_2 = mod(C_2) \cap A_1 = \emptyset \), so ∀x ∈ mod(C_1). (Σ_1, Δ)(x) = T and ∀x ∈ mod(C_2). (Σ_2, Δ)(x) = 
T. It follows by the induction hypothesis that there are decorations C_1' of C_1 
and C_2' of C_2 such that the judgements Σ_1 | Δ ⊢ \{P_1\}C_1'\{Q_1\} and Σ_2 | Δ ⊢ 
\{P_2\}C_2'\{Q_2\} are provable. So by the Reddy PARALLEL rule we can deduce

\[ \Sigma_1, \Sigma_2 | Δ ⊢ \{P_1 * P_2\}C_1\|C_2\{Q_1 * Q_2\}, \]

and hence Σ | Δ ⊢ \{P_1 * P_2\}C_1\|C_2\{Q_1 * Q_2\}. Finally, note that C_1'\|C_2' is a 
decoration of C_1\|C_2, so the result holds as required.

• For **REGION**, consider a well-formed instance

\[
\frac{\Gamma \vdash A \cup X\{(P * R) \land b\}C\{Q * R\}}{\Gamma, r(X) : R \vdash_A \{P\} \text{with } r \text{ when } b \text{ do } C\{Q\}}
\]

Let Σ, Σ_0, Δ be well-formed and dom(Σ) = A, dom(Σ_0) = X, map dom Δ = Γ, and ∀x ∈ mod(C). (Σ, Σ_0, Δ)(x) = T. Then dom(Σ, Σ_0) = A∪X. So by induction 
hyothesis there is a decoration C' of C such that

\[ \Sigma, Σ_0 | Δ ⊢ \{(P * R) \land b\}C'\{Q * R\} \]

is provable. By well-formedness we have free(P, Q) ⊆ A, so Σ ⊢ P, Q Assert is 
valid. Hence

\[ \Sigma | Δ, r(Σ_0) : R \vdash \{P\} \text{with } r \text{ when } b \text{ do } C'\{Q\} \]
is provable by the Reddy region rule. This judgement corresponds to the conclusion of the CSL rule.

- For Resource, consider a well-formed instance

\[ \Gamma, r(X) : R \vdash A \{ P \} C \{ Q \} \]

\[ \Gamma \vdash A \cup X \{ P \} C \{ Q \} \]

By well-formedness we have \( \text{free}(R) \subseteq X \), and \( \text{mod}(C) \subseteq A \cup X \cup \text{owned}(\Gamma) \). We can choose \( \Sigma' \) and \( \Delta \) such that \( \text{dom}(\Sigma') = A \cup X \), \( \text{map dom} \Delta = \Gamma \), and \( (\Sigma', \Delta)(x) = \top \) for all \( x \) in \( \text{mod}(C) \). We can then split \( \Sigma' \) as \( \Sigma, \Sigma_0 \) with \( \text{dom}(\Sigma_0) = X \), \( \text{dom}(\Sigma) = A \). Then by the induction hypothesis there is a decoration \( C' \) of \( C \) such that such that \( \Sigma \mid \Delta, r(\Sigma_0) : R \vdash \{ P \} C' \{ Q \} \) is provable and matches the CSL assertion \( \Gamma, r(X) : R \vdash A \{ P \} C \{ Q \} \). Using the Reddy Resource rule we can deduce

\[ \Sigma \mid \Delta \vdash \{ P \} r(\Sigma_0) \text{ resource } r \text{ in } C' \{ Q \} \]

and since \( \text{resource } r(\Sigma_0) \text{ in } C' \) is a decoration of \( \text{resource } r \text{ in } C \), that completes the proof.

(End of Proof)

The choices of \( \Sigma_0 \) and so on in the above proof details are arbitrary up to some explicit constraints on their domains (i.e. on which variables are given a permission) and the insistence that certain variables get total permission, collectively.

8 Conclusions

CSL, using rely sets, relaxes the rather stringent syntactic constraints on shared variable usage from Owicki-Gries [8,7]. This logic can handle programs in which shared variables are protected by multiple resources; the inference rules require that a process must acquire all protecting resources before writing a shared variable, and must acquire some protecting resource before reading a shared variable. In Owicki-Gries logic each shared variable was protected by a single resource. The use of rely sets allows us to avoid the need for non-compositional side conditions dealing with “no other process modifies” constraints.

We have demonstrated a strong connection between CSL and Reddy’s logic based on static permissions for variables: each provable judgement in Reddy’s logic corresponds to a provable assertion in CSL, and vice versa. The relationship is asymmetric: in general many Reddy judgements correspond to the same CSL assertion, the differences arising because of arbitrary choices of permission or different distributions of permission among the permission contexts appearing in a judgement. Arguably this abundance of derivations is unattractive: a single proof, without book-keeping concerning arbitrary choices, is preferable to a plethora. This connection also shows that CSL provides a form of syntactic control of interference in the same sense as Reddy’s logic does.

Our analysis, in the proof details that establish the above connection, indicates a systematic strategy for choosing permission contexts to match a given resource
context and rely set while ensuring that all variables written by a program get total permission, starting from a proof for that program in CSL. In contrast the algorithm in [9] infers which resources need to protect which identifiers (i.e. discovers a suitable resource context) while looking for permission assignments that ensure that each variable occurrence in the program gets “enough” permission, starting from a putative judgement for that program. Our strategy starts with more information, a proven program rather than a potentially provable program, so solves a simpler problem. It would be interesting to explore the Reynolds algorithm further in the light of our results.

Although Reddy’s logic was originally formulated in [9] with inference rules that deal with decorated commands, our results show that it is possible to dispense entirely with decorations and use Reddy-style rules for undecorated commands, relying instead implicitly on the structure of a derivation to keep track of permission choices. Essentially, the only place where this makes a difference is in the resource rule, which we can replace by

\[
\Sigma_0 \vdash R \text{ Assert } \Sigma \mid \Delta, r(\Sigma_0) : R \vdash \{P\}C\{Q\} \\
\Sigma, \Sigma_0 \mid \Delta \vdash \{P \ast R\} \text{ resource } r \text{ in } C\{Q \ast R\}
\]

The choice of \(\Sigma_0\) to “decorate” the local resource name \(r\) in the original Reddy rule does not need to be made explicit here; instead we can infer \(\Sigma_0\) from the premisses. Of course, the cost of doing this would be that the undecorated command is less informative by itself. With this reformulation of Reddy’s logic we would again obtain an analogous connection with CSL: there is a many-one relationship between Reddy derivations and CSL derivations for the same command. This version of Reddy’s logic has the advantage of involving only conventional commands, not decorated commands; consequently we can use conventional semantic models, such as action traces, to establish soundness. In fact it is straightforward, having done this, to combine action trace semantics with permissions [4] and thereby obtain a soundness proof for Reddy’s logic, adjusting the notion of local enabling to manage permissions appropriately. One can also deduce soundness by appealing to the inter-derivability of the two logics, and our existing soundness proof of CSL [5]. Either approach seems simpler than the method of [9], which deals with decorated commands and seeks to generalize action traces by introducing pre-actions, pre-traces, busy markers, and “extended” contexts. We believe it is more natural to work with a semantic model in which logical concepts such as permissions, invariants, protection lists, and decorated commands play no rôle.

References


Nominal SOS

Matteo Cimini\textsuperscript{a} MohamamdReza Mousavi\textsuperscript{b} Michel A. Reniers\textsuperscript{c} Murdoch J. Gabbay\textsuperscript{d}

\textsuperscript{a} Department of Computer Science, Reykjavík University, Reykjavík, Iceland
\textsuperscript{b} Department of Computer Science, TU/e, Eindhoven, The Netherlands
\textsuperscript{c} Department of Mechanical Engineering, TU/e, Eindhoven, The Netherlands
\textsuperscript{d} Computer Science Department, Heriot-Watt University, Edinburgh, United Kingdom

Abstract

Plotkin’s style of Structural Operational Semantics (SOS) has become a de facto standard in giving operational semantics to formalisms and process calculi. In many such formalisms and calculi, the concepts of names, variables and binders are essential ingredients. In this paper, we propose a formal framework for dealing with names in SOS. The framework is based on the Nominal Logic of Gabbay and Pitts and hence is called Nominal SOS. We define nominal bisimilarity, an adaptation of the notion of bisimilarity that is aware of binding. We provide evidence of the expressiveness of the framework by formulating the early $\pi$-calculus and Abramsky’s lazy $\lambda$-calculus within Nominal SOS. For both calculi we establish the operational correspondence with the original calculi. Moreover, in the context of the $\pi$-calculus, we prove that nominal bisimilarity coincides with Sangiorgi’s open bisimilarity and in the context of the $\lambda$-calculus we prove that nominal bisimilarity coincides with Abramsky’s applicative bisimilarity.

Keywords: SOS, Nominal SOS, Nominal calculi, $\lambda$-calculus, $\pi$-calculus.

1 Introduction

The development of a formal semantics for programming and specification languages is a necessary first step towards rigorous reasoning about them. For instance, a formal semantics allows one to prove the correctness of language implementations, and is a prerequisite for proving the validity of program optimizations. Operational semantics is a widely-used methodology to define formal semantics for computer languages, which represents the execution of programs as their step-by-step development on an abstract machine. Structural Operational Semantics (SOS) was introduced by Gordon Plotkin in [24], reprinted in [25], as a logical and structural approach to defining operational semantics. The logical structure of SOS specifications supports a variety of reasoning principles that can be used to prove properties of programs whose semantics is given using SOS. Moreover, SOS language spec-
ifications can be used for rapid prototyping of language designs and to provide experimental implementations of computer languages.

SOS has become the de facto standard for defining operational semantics, and a wealth of programming and executable specification languages have been given formal semantics using it. In the past two decades much work on the underlying theory as well as on the practice of SOS has been carried out—see, e.g., [3,21] and [6,13,20], respectively. Many programming and specification languages make use of the concepts of names and binders. For example, in the $\pi$-calculus [18,19,29], names are first-class objects and the whole language is built on the idea that concurrent agents communicate by exchanging names. Incorporating nominal notions within SOS has received some attention in recent years, but nevertheless the meta-theory of SOS is not sufficiently adapted for these new frameworks. In this paper we propose a formal framework for the handling of names in SOS, called Nominal SOS, which is based on the nominal techniques of Gabbay, Pitts, and Urban [11,31].

The most important notion of equivalence of programs in the context of SOS is bisimilarity [22]. We argue that this notion, taken as it is, is not satisfactory and we adapt bisimilarity in order to better suit our context with binders. We call this equivalence nominal bisimilarity.

Basic notions such as $\alpha$-conversion and substitution are essential parts of most nominal calculi. We show that these notions can be naturally captured in the Nominal SOS framework. Moreover, we give evidence of the expressiveness of our framework by modeling two of the most prominent examples of nominal calculi, namely the lazy $\lambda$-calculus and the early $\pi$-calculus. For both we show that our specifications coincide operationally with the original definitions of [2] and [29], respectively. We moreover prove that in the case of the $\pi$-calculus our notion of nominal bisimilarity coincides with the well-known open bisimilarity of Sangiorgi [29,28]. Finally, we show that nominal bisimilarity in the context of our formulation of the lazy $\lambda$-calculus coincides with the applicative bisimilarity of Abramsky [2].

Proofs are omitted in the main text and the reader can find them, together with a fully elaborated account of Nominal SOS, in [8]. This document extends the last chapter of Cimini’s Ph.D. thesis [7].

Structure of the paper. The rest of the paper is organized as follows. In Section 2 we define nominal terms and in Section 3 we define the framework of Nominal SOS. We show in Section 4 how $\alpha$-conversion and different types of substitution can be accommodated in Nominal SOS. Section 5.1 is devoted to our formulation of the $\pi$-calculus and Section 5.2 addresses the lazy $\lambda$-calculus. In Section 6 we discuss related work and Section 7 concludes the paper.

## 2 Nominal terms

The following definitions of sorts and nominal signature are familiar from [31].

**Definition 2.1 (Sorts)** Sorts are defined inductively as follows: $\sigma ::= 1 | \delta | A | [A]\sigma | \sigma \times \sigma$, where $1$ is the unit sort, $\delta$ is a base sort, $A$ is an atom sort, $[A]\sigma$ denotes an abstraction sort, and $\times$ denotes pairing.
Intuitively, \([A]_\sigma\) is a sort whose elements are functions from objects of sort \(A\) to objects of sort \(\sigma\). As is standard, pair sorts will associate to the left, so that \(\sigma_1 \times \sigma_2 \times \sigma_3\) stands for \((\sigma_1 \times \sigma_2) \times \sigma_3\).

**Definition 2.2 (Nominal signature)** A nominal signature \(\Sigma\) is a triple \((\Delta, A, F)\), where

(i) \(\Delta\) is a set of base sorts ranged over by \(\delta\),
(ii) \(A\) is a set of atom sorts ranged over by \(\forall A\), and
(iii) \(F\) is a set of operators \(f_{(\sigma_1\times\ldots\times\sigma_n)}^\delta\), denoting a function symbol \(f\) with arity \((\sigma_1 \times \ldots \times \sigma_n) \to \delta\), where \(n \geq 0\).

For each atom sort \(A\), we fix a countably infinite set of atoms \(a_\forall, b_\forall, c_\forall, d_\forall\ldots\), and for each sort \(\sigma\), we assume a countably infinite set \(V_\sigma\) of variable symbols \(x_\sigma, y_\sigma, z_\sigma\ldots\). We sometimes write just \(f, a, b, c, d, n, m, \) and \(x, y, z\), leaving arities and sorts implicit (but still present). We assume that all these sets of symbols are pairwise disjoint.

**Definition 2.3 (Nominal terms)** Given a signature \(\Sigma = (\Delta, A, F)\), the set of nominal terms over the signature \(\Sigma\) is denoted by \(T(\Sigma)\) and it is defined as follows, where we write \(t_\sigma\) for a term \(t\) of sort \(\sigma\):

\[
t ::= x_\sigma | a_\forall | ([a_\forall]t_\sigma)_{[A]_\sigma} | (f_{(\sigma_1\times\ldots\times\sigma_n)}^\delta(t_{\sigma_1}, \ldots, t_{\sigma_n}))_{\delta}
\]

where \(A \in A\), \(a_\forall \in A\), \(x_\sigma \in V_\sigma\) and \(f \in F\) with arity \((\sigma_1 \times \ldots \times \sigma_n) \to \delta\).

The subscripts of nominal terms control sorting and we tend to omit them when they are clear from the context or immaterial. We call \([a] t\) an abstraction. A syntactic constructor for pairs is not really needed and is therefore omitted.

For a nominal term \(t\), the following definitions will be useful in the remainder of the paper. In what follows, \(f\) and \(g\) are a unary and a binary function symbol.

- \(A(t)\) stands for the set of atoms that occur in \(t\). For example, \(A(f([a]g(a,b))) = \{a, b\}\).
- \(ba(t)\) is the set of atoms \(a\) for which there exists a subterm \([a]t'\) in \(t\), i.e., the set of abstracted atoms in \(t\). For example, \(ba(f([a]g(a,b))) = \{a\}\).
- \(fa(t)\) is the set of atoms \(a\) in \(A(t)\) that have an occurrence in \(t\) that is not within the scope of an abstraction \([a]t'\), for some term \(t'\). We call \(fa(t)\) the set of free atoms of \(t\). For example, \(fa(f([a]g(a,b))) = \{b\}\) and also \(fa(g(f([a]a,a)) = \{a\}\).
- An atom \(a\) is fresh in \(t\) whenever \(a \not\in fa(t)\). We also say that a term \(t\) is binding-closed if \(fa(t) = \emptyset\), i.e., the term \(t\) does not contain free atoms.\(^1\)
- We say that a nominal term is closed if it contains no variables. It is called open otherwise.

\(^1\) Binding-closed terms corresponds to those that in literature are usually called closed. For instance, in the context of the \(\lambda\)-calculus the \(\lambda\)-term \(\lambda a.\lambda b.(a \; b)\) is closed, as it does not contain free atoms, see [2,4]. We adopt a different nomenclature in order to avoid confusion with the standard concept of closed term of SOS, i.e., a term that contains no variables.
For example, \(a\) and \([a]f(b)\) are closed terms, but \(x\) and \([a]y\) are open terms. Note that neither \(a\) nor \([a]f(b)\) is binding-closed.

3 Nominal SOS

Suppose \(a\) is an atom and \(t\) is a term of some sort. We call a formula \(a \# t\) a freshness assertion. In what follows we give a derivation system in order to derive freshness assertions.

**Definition 3.1 (Freshness derivation rules)** Let \(\Sigma\) be a nominal signature, and let the atom \(a\) and the term \(t\) be over the signature \(\Sigma\). We say that \(a \# t\) is derivable when it may be derived using the following rules, where \(a\) and \(b\) are distinct atoms.

\[
\begin{array}{c}
  a \# b \\
  a \# t_1, \ldots, a \# t_n \\
  a \# f(t_1, \ldots, t_n) \\
  a \# [a] \! t \\
  a \# [b] \! t
\end{array}
\]

These derivation rules are familiar from existing work [31,9].

We are now ready to define the notion of nominal transition system specification whose rules employ the freshness assertions defined above.

**Definition 3.2 (Nominal Transition System Specification)** A nominal transition system specification (NTSS) is a triple \((\Sigma, R, D)\) consisting of:

(i) A nominal signature \(\Sigma\);

(ii) A set of (transition) relation symbols \(R\). To each \(r \in R\) we associate a (transition relation) arity which is a sort of the form \(\sigma \times \sigma_l \times \sigma'\). We may call: \(\sigma\) the ‘sort of the source of the transition’, \(\sigma'\) the ‘sort of the target of the transition’, and \(\sigma_l\) the ‘sort of the label of the transition’.

(iii) A set of derivation rules \(D\) (see below).

Given an NTSS \(T\), we denote with \(A(T)\) the set of atoms of the signature of \(T\). For a relation \(r \in R\) with arity \(\sigma \times \sigma_l \times \sigma'\), if \(\sigma_l\) is the unit sort \(1\) then we say that \(r\) has no label. If \(\sigma'\) is also the unit sort, then \(r\) is a predicate symbol. We may silently drop \(\sigma_l\) (and \(\sigma'\)) if they are the unit sort.

For a relation \(r \in R\) with arity \(\sigma \times \sigma_l \times \sigma'\), a positive transition formula is written \(t \xrightarrow{l} r t'\), where \(t\) is a possibly open term of sort \(\sigma\) (we call it the source term), \(l\) is a possibly open term of sort \(\sigma_l\) (we call it the label), and \(t'\) is a possibly open term of sort \(\sigma'\) (we call it the target term).

For the same relation \(r\), we write \(t \xleftarrow{l} r t'\) for a negative transition formula, where \(t\) is of sort \(\sigma\) and \(l\) is of sort \(\sigma_l\). A transition formula is a positive or negative transition formula.
Definition 3.3 (Derivation rule) A derivation rule is of the form

\[ \{ t_i \xrightarrow{r_i} t'_i \mid i \in I \} \cup \{ t_j \xrightarrow{r_j} \mid j \in J \} \cup \{ a_k \# t_k \mid k \in K \} \]

\[ t \xrightarrow{r} t' \]

where

- \( I, J \) and \( K \) are indexing sets,
- \( \{ t_i \xrightarrow{r_i} t'_i \mid i \in I \} \) is a set of positive transition formulae, called the positive premises of the rule,
- \( \{ t_j \xrightarrow{r_i} \mid j \in J \} \) is a set of negative transition formulae, called the negative premises of the rule,
- \( \{ a_k \# t_k \mid k \in K \} \) is a set of freshness assertions, called the freshness premises of the rule, and
- \( t \xrightarrow{r} t' \) is a positive transition formula, called the conclusion of the rule.

We call \( t, l, \) and \( t' \) the source, the label and the target of the rule, respectively.

We call a derivation rule an axiom if \( I, J, \) and \( K \) are empty. A derivation rule is positive when the index set \( J \) is empty. An NTSS is positive when all its deduction rules are positive. Positive NTSS’s come with a natural notion of semantics, i.e., the set of provable closed transitions formulae; the same notion is adopted for the semantics of freshness formulae by means of the derivation rules given in Definition 3.1. In this paper we restrict ourselves to positive NTSS’s; the semantics of full Nominal SOS can be found in [7, Section 5.3.1].

The most important notion of equivalence between programs defined in SOS is bisimilarity [17,22]. Unfortunately, this equivalence turns out not to be satisfactory in a context with binders. This point is carefully explained, in the context of the \( \pi \)-calculus, on pages 64-65 of [29], where it is shown that the two processes \( P = \nu c. \pi c \) and \( Q = \nu c. ( \alpha c \parallel \nu d. \bar{b} ) \) are distinguished since \( P \) can perform a transition \( \pi(b) \rightarrow \) while \( Q \) is not able to perform any \( \pi(b) \rightarrow \) transitions; the reason is that \( Q \) is not able to change its bound variable to \( b \) since \( b \) is not fresh in \( Q \).

Nominal bisimilarity is thus introduced below as an adaptation of the ordinary bisimilarity that is aware of binding. What happens in the theory of the \( \pi \)-calculus is that bisimilarity is adjusted and transitions of the form \( a(b) \rightarrow \) are matched only for those bound variables that are fresh in both terms. Our notion of bisimilarity revisits the ordinary bisimilarity in such a manner. In the remainder of the paper we relate our nominal bisimilarity to important notions of equivalence in the literature.

Definition 3.4 (Nominal bisimilarity) Let \( T \) be an NTSS. Nominal bisimilarity \( \equiv_T \) is the largest symmetric binary relation \( \sim \) over closed terms of \( T \) such that for all closed terms \( P \) and \( Q \) and labels \( l \) such that \( P \sim Q \) and \( a\# P \) and \( a\# Q \) for all \( a \in ba(l) \), it holds that if \( P \xrightarrow{l} P' \) then there exists \( Q' \) such that \( Q \xrightarrow{l} Q' \) and \( P' \sim Q' \).
4 Substitution and $\alpha$-conversion

4.1 Substitution transitions

Substitution and $\alpha$-equivalence play a key role in the definition of the semantics of calculi with binders. We will now show how those notions can be accommodated in a uniform fashion within the framework of Nominal SOS.

Term-for-atom substitutions are typically employed by higher-order calculi, such as the $\lambda$-calculus, the Calculus of Higher Order Communicating Systems (CHOCS) [30] and the Higher-Order $\pi$-calculus [26]. Given a nominal signature, we generate deduction rules with the goal of proving transitions of the form $t_1 \xrightarrow{a_T} t_2 \xrightarrow{} t_3$ for some atom $a$ and terms $t_1$, $t_2$ and $t_3$, where terms $t_1$ and $t_3$ are bound to be of the same sort. This type of transition should be read as the term $t_2$ replaces the atom $a$ in the term $t_1$ leading to the term $t_3$. For all atoms $a$ and function symbols $f$, we have the following rules.

$$
\begin{align*}
  a & \xrightarrow{T} z \quad \text{(a1-Ts)} \\
  a & \xrightarrow{} a \quad \text{(abs1-Ts)} \\
  x & \xrightarrow{T} y \quad \text{(a2_Ts)} \\
  y & \xrightarrow{T} x \quad \text{(abs2_Ts)} \\
  [a]x & \xrightarrow{T} [a]x \quad \text{(ftTs)}
\end{align*}
$$

The reader can infer the sort of the variables used in the rules by their usage. The reader may be more familiar with the syntactic substitution operation, defined below, where $M$ and $N$ are closed terms and $a$ and $b$ are distinct atoms.

$$
\begin{align*}
a[N/a] &= N \\
[a][b] &= a \\
([a]M)[N/b] &= [a](M[N/b]) \quad \text{if $a$ is fresh in $N$} \\
f(M_1, M_2, \ldots, M_n[N/a]) &= f(M_1[N/a], M_2[N/a], \ldots, M_n[N/a])
\end{align*}
$$

The following theorem states that the two notions (substitution transitions and syntactic substitutions) correspond.

**Theorem 4.1 (Correctness of substitution transitions)** Let $T$ be an NTSS. Let $M$ and $N$ be closed terms, and $a$ be an atom. Then, it holds that $M \xrightarrow{a_T} M'$ if and only if $M' = M[N/a]$.

Atom-for-atom substitution is used in calculi such as the $\pi$-calculus [29,19] and its variants. The same set of rules provided for the term-for-atom substitution are

---

2 Theorem 4.1 is stated as Theorem 2 in Section 4.1 of [8] and proved in Section 13 of that paper.
able to model the atom-for-atom substitution. These transitions are denoted $x \xrightarrow{a^b} y$ and the first and second argument of this label range over atoms.

A syntactic atom-for-atom substitution over nominal terms, together with its corresponding correctness theorem, can be provided. It turns out to be just a straightforward adaptation of Theorem 4.1.

### 4.2 $\alpha$-conversion transitions

The notion of $\alpha$-conversion is a natural equivalence guaranteeing that the exact atom chosen in binders is not important and can be indeed replaced by any other atom (while avoiding capture). Thanks to freshness assertions, we can accommodate $\alpha$-conversion in our framework as an ordinary transition relation. Given a nominal signature, the following deduction rules define $\rightarrow_{\alpha}$. For all atoms $a$ and $b$ and function symbols $f$, we have the following rules.  

\[
\begin{align*}
    x & \rightarrow_{\alpha} x \quad (\text{id}_\alpha) \\
    x & \xrightarrow{a^b} y \quad b \# x \quad (\text{abs1}_\alpha) \\
    [a]x & \rightarrow_{\alpha} [b]y \quad (\text{abs2}_\alpha) \\
    \{ x_i \rightarrow_{\alpha} x'_i \mid 0 < i \leq n \} & \rightarrow_{\alpha} f(x'_1, x'_2, \ldots, x'_n) \quad (f_\alpha) \\
    x & \rightarrow_{\alpha} y \quad y \rightarrow_{\alpha} z \\n    x & \rightarrow_{\alpha} z \quad (\alpha \cdot \text{upTo}_\alpha)
\end{align*}
\]

The reader will notice that $\alpha$-conversion transitions rely on those for the atom-for-atom substitution (rule $\text{abs1}_\alpha$). Throughout the paper, whenever we say that the rules above for $\alpha$-conversion transitions are present in an NTSS, this implies that also the rules for atom-for-atom substitutions are present.

The reader is perhaps familiar with the syntactic version of $\alpha$-conversion, as stated in the following theorem.

**Theorem 4.3 (Correctness of $\alpha$-conversion transitions)** Let $T$ be an NTSS. For all closed terms $M$ and $N$ over the signature of $T$, it holds that $M \rightarrow_{\alpha} N$ if and only if $M =_{\alpha} N$.

Calculi with binders usually consider a term as a representative of the equivalence class of all the terms that are $\alpha$-convertible to it. In Nominal SOS, it is possible to achieve this by augmenting the NTSS with a deduction rule, given below.

---

3 In the nominal world, the standard definition of $\alpha$-equivalence is based on permutations, see [11] for instance. However, we preferred to model the standard definition of $\alpha$-equivalence.

4 Theorem 4.3 is stated as Theorem 3 in Section 4.2 of [8] and proved in Section 14 of that paper.
Definition 4.4 (Transitions up to $\alpha$-equivalence) Let $T$ be an NTSS and $l$ be a label of the signature of $T$. The transition relation $\xrightarrow{l}$ is up to $\alpha$-equivalence whenever the deduction rules of $T$ contain the rules for $\alpha$-conversion transitions, as defined above, the rules for atom-for-atom substitution transitions, as defined in Section 4.1, and the deduction rule:

$$
\frac{x \rightarrow_{\alpha} y \quad y \overset{l}{\rightarrow} z}{x \overset{l}{\rightarrow} z} (l \cdot \text{upTo}\alpha) .
$$

Depending on the peculiarities of the calculus at hand, the modeller might want to consider defining some of the transition relations to be up to $\alpha$-equivalence.

5 Examples

In this section we provide some evidence of the expressiveness and applicability of Nominal SOS by formulating in our framework two classical calculi, namely the early $\pi$-calculus \cite{29,19} and the lazy $\lambda$-calculus \cite{2}.

5.1 The early $\pi$-calculus and open bisimilarity

The signature $\Sigma^{\pi}$ of our $\pi$-calculus is modelled by a base sort $P$ and atom sort $C$ (for processes and channels, respectively) and the following function symbols.

(i) $0 : \rightarrow P$ for inaction (deadlock),
(ii) $\tau : P \rightarrow P$ for $\tau$-prefix,
(iii) $\text{out}(\cdot, \cdot, \cdot) : (C \times C \times P) \rightarrow P$ for output prefix,
(iv) $\text{in}(\cdot, \cdot) : (C \times [C]P) \rightarrow P$ for input prefix,
(v) $\nu(\cdot) : [C]P \rightarrow P$ for restriction,
(vi) $\cdot || \cdot : (P \times P) \rightarrow P$ for parallel composition,
(vii) $\cdot + \cdot : (P \times P) \rightarrow P$ for nondeterministic choice,
(viii) $! : P \rightarrow P$ for parallel replication.

The syntax employed for input and output prefixes differs slightly from the standard notation used in the $\pi$-calculus. In particular, our term $\text{out}(a, b, P)$ corresponds to the process $\overline{ab}.P$ of the $\pi$-calculus, and $\text{in}(a, [b]P)$ corresponds to $a(b).P$. The same choice is adopted for the labels.

Below, we specify the semantics of the early $\pi$-calculus in Nominal SOS. Since in our framework labels are open terms, we display an input transition label as $\text{in}(a, b)$, assuming a different operator $\text{in}$ accepting two atoms as arguments. For presentation purposes, we use the same names to stipulate the meaning of the transitions. For the same reasons, we model an output transition label as $\text{out}(a, b)$ and a bound output transition label as $\text{bout}(a, [b]0)$, abbreviated as $\text{bout}(a, [b])$ throughout the text. The set of rules of the signature $\Sigma^{\pi}$ contains the following rules, we use $\alpha$ to range over labels and $a$, $b$ and $c$ to range over atoms.
\text{Theorem 5.1} \text{(Operational correspondence: early π-calculus)} \text{ For all } \pi, Q \in \Pi, P \xrightarrow{\alpha} Q \Leftrightarrow [P]^{\pi} \xrightarrow{[\alpha]^{\pi}} [Q]^{\pi}, \text{ where } \alpha \text{ ranges over the labels of the form } \tau, ab, \bar{a}b \text{ and } \pi(b) \text{ from the original early π-calculus}.

The reader may wonder what is the equivalence over π-calculus terms that corresponds to nominal bisimilarity. The next theorem provides us with an answer: nominal bisimilarity in our formulation of the early π-calculus coincides with Sangiorgi’s open bisimilarity, see [29, Section 4.2] and [28].
Definition 5.2 (Open bisimilarity) Open bisimilarity $\leftrightarrow^o$ is the largest symmetric relation $\sim$ on $\Pi$ such that whenever $P \sim Q$, and $\sigma$ is a substitution (see Definition 1.1.3 on page 14 of [29]), if $P\sigma \xrightarrow{\alpha} P'$, then there exists $Q'$, such that $Q\sigma \xrightarrow{\alpha} Q'$ and $P' \sim Q'$.

The reader should notice that this definition is the very basic formulation of open bisimilarity, which does not involve distinctions, see [29] and [28]. In Definition 5.2, it is important to note that the ranging over all the substitutions is performed at each step of the bisimulation game.

Theorem 5.3 (Open bisimilarity and bisimilarity coincide) For all $P, Q \in \Pi$, $P \leftrightarrow^o Q$ if, and only if, $[P]^\pi \leftrightarrow [Q]^\pi$.

Theorem 5.3 essentially holds because in our nominal formulation of the $\pi$-calculus, nominal bisimilarity also takes into account the substitutions transitions. In [7, Section 5.21], it is shown that if nominal bisimilarity is adapted not to match the substitution transitions it would coincide with the ordinary bisimilarity over the $\pi$-calculus.

5.2 The lazy $\lambda$-calculus and applicative bisimilarity

The signature $\Sigma^\lambda$ of the lazy $\lambda$-calculus is constructed using a base sort $L$ for $\lambda$-terms and an atom sort $A$. The signature also contains the following function symbols.

(i) $(\_): A \rightarrow L$: A unary function symbol for creating terms from atoms;
(ii) $\lambda(\_): [A]L \rightarrow L$: A unary function symbol for abstractions;
(iii) $\_\_\_\_: (L \times L) \rightarrow L$: A binary function symbol for application.

The semantics includes a reduction transition $\rightarrow$, here displayed with no label to remain in line with the standard notation from [2], transitions $P \xrightarrow{\tau}$ for terms $P$ of sort $L$, the rules for term-for-atom substitution transitions as generated in Section 4.1, and additionally the rules $(a) \xrightarrow{a\#x} z$ and $(a) \xrightarrow{a} (a)$.

The set of rules of the signature $\Sigma^\lambda$ contains the following derivation rules, which define the operational semantics of our version of the lazy $\lambda$-calculus, for all atoms $a$.

\[
\begin{align*}
\frac{\lambda([a]x) \rightarrow \lambda([a]x)}{\text{(abs1AP)}} \\
\frac{x_0 \xrightarrow{a\#y} x_1 \quad \forall b. (b\#y) \rightarrow \lambda([a]x_0) \xrightarrow{y} x_1}{\text{(abs2AP)}} \\
\frac{x_0 \rightarrow y_0 \quad y_0 \xrightarrow{y} y_1 \quad y_1 \rightarrow y_2}{(x_0 \ x_1) \rightarrow y_2} \quad \frac{(x_0 \ x_1) \rightarrow y_1 \quad y_1 \xrightarrow{y} y_2}{(x_0 \ x_1) \xrightarrow{y} y_2} \quad \text{(app1AP)} \\
\frac{(x_0 \ x_1) \rightarrow y_1 \quad y_1 \xrightarrow{y} y_2}{(x_0 \ x_1) \xrightarrow{y} y_2} \quad \text{(app2AP)}
\end{align*}
\]

7 Theorem 5.3 is stated as Theorem 7 in Section 6 of [8] and proved in Section 22 of that paper.
Moreover, the transition relations $\to$, $\xrightarrow{P}$ for any binding-closed term $P^8$, and the term-for-atom substitution transitions are up to $\alpha$-equivalence. Recall that, by Definition 4.4, this means that the set of rules of $\Sigma^\lambda$ contains also the rules for $\alpha$-conversion transitions and the rules for atom-for-atom substitution transitions and these transitions are set to be up to $\alpha$-equivalence as well.

We denote by $\Lambda$ the set of $\lambda$-terms of $[2,4]$, and by $\Lambda^0$ the set of those that do not contain free variables. The encoding $\llbracket \cdot \rrbracket^\lambda$ is a map from $\Lambda$ into terms of our nominal $\lambda$-calculus. The mapping is straightforward and not presented here; it can however be found in [8].9 The following theorem establishes the operational correctness of our formulation of the lazy $\lambda$-calculus with respect to its original formulation for $\lambda$-terms in $\Lambda^0$.10

**Theorem 5.4 (Operational correspondence: lazy $\lambda$-calculus)** For all $M, N \in \Lambda^0$, $M \to N \iff \llbracket M \rrbracket^\lambda \to \llbracket N \rrbracket^\lambda$.

The reader should notice that if we ruled out the premises $\forall b. (b \# y)$ from rule (abs2AP) the operational correspondence of Theorem 5.4 would hold for the set of all $\lambda$-terms. The reason we restrict the parameter passing to binding-closed terms only is that it ties up directly with the study that follows.

In the context of the lazy $\lambda$-calculus, one of the most interesting notions of bisimilarity is the *applicative bisimilarity* due to Samson Abramsky [2]. Below we recall the definition of this equivalence.

**Definition 5.5 (Applicative bisimilarity in the $\lambda$-calculus)** Applicative bisimilarity is the largest symmetric relation $\simeq$ on $\Lambda^0$ such that whenever $M \simeq N$, if $M \to \lambda a.M'$ for some variable $a$ and $M' \in \Lambda$, then there exist some variable $b$ and $N' \in \Lambda$ such that
- $N \to \lambda b.N'$, and
- $M'[P/a] \simeq N'[P/b]$, for all $P \in \Lambda^0$.

It is important to remark that the applicative bisimilarity of the $\lambda$-calculus is defined over closed $\lambda$-terms. Indeed, this equivalence is very unsatisfactory over terms that contain free variables. For instance, for all variables $a, b$ and $c$, it holds that $a \simeq b$ and $\lambda a.b \simeq \lambda a.c$.

The following theorem states that applicative bisimilarity in the lazy $\lambda$-calculus coincides with nominal bisimilarity in the nominal formulation of the lazy $\lambda$-calculus given above.11

**Theorem 5.6 (Applicative and nominal bisimilarity coincide)** For all $M, N \in \Lambda^0$, $M \simeq N$ if, and only if, $\llbracket M \rrbracket^\lambda \leftrightarrow \llbracket N \rrbracket^\lambda$.

---

8 The premises $\forall b. (b \# y)$ of rule (abs2AP) ensure that the parameter passing is performed with binding-closed terms only. This characterization already appeared in [9, Section 9.2].

9 The encoding can be found in Section 5.1 of [8]. By way of example the reader can consider that $\llbracket \lambda x. \lambda y. (x y) \rrbracket^\lambda = \lambda (\llbracket x \rrbracket^\lambda (\llbracket y \rrbracket^\lambda (x y)))$.

10 Theorem 5.4 is stated and proved as Theorem 18 in Section 25 of [8].

11 Theorem 5.6 is stated as Theorem 13 in Section 7.1 of [8] and proved in Section 25 of that paper.
6 Related work

We are aware of a number of existing approaches that accommodate names and binders inside the SOS framework. The frameworks that are most relevant to the work presented in this paper are by Miller and Tiu in [16] (FOλ∆∇), by Lakin and Pitts in [15] (MLSOS) and in [14] (αML) and by Gacek, Miller and Nadathur in [12] (Abella).

As a first difference, Nominal SOS is the only approach that directly extends the formal framework of SOS. We identify some benefits supporting this choice. First, users that are familiar with the SOS framework will find Nominal SOS easy to use. Secondly, although a meta-theory of SOS for calculi with binders can be carried out with the frameworks mentioned above, Nominal SOS seems to be close enough to ordinary SOS. In this respect, a meta-theory for binders can follow by and large the same lines of the meta-theory of ordinary SOS, which has been successfully developed for over 20 years, see [21]. We also expect that already existing results from the meta-theory of SOS would lift to Nominal SOS with relatively little effort.

Some technical differences between Nominal SOS and the systems mentioned above are worth a mention. For instance in MLSOS and α-ML, only restricted operations are allowed on atoms and programs do not depend upon concrete atoms. In Nominal SOS, languages can instead be defined in a way that a particular atom may affect the computation. FOλ∆∇ and Abella are based on the so-called λ-tree approach to syntax where a logic that has its roots in the λ-calculus takes care of the binding management. In Nominal SOS, the management of binders is not delegated to an underlying layer and users need to specify the treatment of binders completely. As another difference, in the mentioned approaches α-conversion is built-in and guaranteed on the meta-level. In Nominal SOS α-conversion is not built-in but it can be automatically generated from the signature. The user can replace it and experiment with alternative notions at will. Moreover, we prefer to model α-equivalence as a transition like any other, so that it can be the subject of meta-theorems based on the shape of rules that may be developed in the future.

We refer the reader to Section 5.8 of [7], where related works are considered in much more detail.

7 Conclusions and future work

In this paper, we have introduced a framework, called Nominal SOS, for modelling the operational semantics of nominal calculi. The framework comes equipped with the basic features used in defining such calculi, namely, substitution and α-conversion. We used the framework to specify the semantics of the early π-calculus and lazy λ-calculus and showed that our formulations of the semantics coincide with the original ones. A notion of nominal bisimilarity arises naturally from our framework. Moreover, we showed that the notion of nominal bisimilarity in our semantics of the early π-calculus coincides with open bisimilarity in the original semantics.

12This property is known in the nominal world as the equivariance property, see [11] and especially [23].
We also proved that nominal bisimilarity coincides with Abramsky’s applicative
bisimilarity in the context of the lazy \( \lambda \)-calculus.

The main goals of our future work are to provide further evidence that Nominal
SOS is expressive enough to capture the original semantics of nominal calculi, such
as variants of the \( \pi \)-calculus and its higher-order version \cite{27}, the psi-calculi \cite{5} and
the object calculi \cite{1}. Also, we plan to address different notions of equivalence
between terms. To begin with, we plan to adapt nominal bisimilarity in order for
it to coincide with open bisimilarity with distinctions \cite{29,28}, when applied to \( \pi \)-
calculus terms. Our main goal is however to develop the meta-theory of Nominal
SOS. By way of example, it would be worth providing congruence formats for
behavioural semantics, possibly generalizing those proposed in \cite{32} and \cite{10}, for
instance.

Acknowledgement

A special thanks to Luca Aceto for his thoughtful comments on this paper at many
different stages of its production.

References


109


Timed Sets, Functional Complexity, and Computability

Robin Cockett\textsuperscript{a,1,5} Joaquín Díaz-Boils \textsuperscript{b,4,6} 
Jonathan Gallagher\textsuperscript{a,3,7} Pavel Hrubes\textsuperscript{a,2,8}

\textsuperscript{a} Department of Computer Science  
University of Calgary  
Calgary, Canada

\textsuperscript{b} Departamento de Lógica y Filosofía  
Universidad de Valencia  
Valencia, Spain

Abstract

The construction of various categories of “timed sets” is described in which the timing of maps is considered modulo a “complexity order”. The properties of these categories are developed: under appropriate conditions they form discrete, distributive restriction categories with an iteration. They provide a categorical basis for modeling functional complexity classes and allow the development of computability within these settings. Indeed, by considering “program objects” and the functions they compute, one can obtain models of computability – i.e. Turing categories – in which the total maps belong to specific complexity classes. Two examples of this are introduced in some detail which respectively have their total maps corresponding to PTIME and LOGSPACE.

Keywords: Restriction categories, complexity measures, functional complexity, computability, Turing categories

1 Introduction

The goal of this paper is to provide a general construction of categories of “timed sets” which may be used as a categorical basis for modeling functional complexity
classes. A motivating example concerns the modeling of polynomially timed functions: this example is of additional interest as it is possible to show one obtains a full model of computability – a Turing category [2,10] – in which the total functions are precisely the PTIME functions in the usual complexity sense. One may therefore identify the computability theory of this Turing category with the theory of PTIME complexity.

To achieve such a modeling, a number of ingredients must be collected. The first and simplest ingredient is a notion of “timing”: here we take a very basic approach supposing that a (partial) map between sets is timed if for every input (on which the map defined) there is a measurement of the “running time” of the function on that input. Of course, “time” here could be replaced by any resource which one might wish to measure. Indeed, at the end of the paper we include a discussion using space as a resource in order to obtain a Turing category whose total functions belong to LOGSPACE.

Formally, the measurement of the resources used by a partial function, \( f \), when applied to an element, \( x \), in its domain, is given by associating to that element a value in a size monoid. The canonical example of a size monoid is, of course, the natural numbers, \( \mathbb{N} \), under addition. Thus, for measuring time complexity, we associate to each \( x \) in the domain of \( f \), a time \( |x|_f \in \mathbb{N} \). If one had wished to measure space, a natural size monoid might be the natural numbers under maximum as the space usage of the composite of two functions is often\(^9\) the maximum of the individual space usages.

Sets with “timed maps” in this sense certainly form a category in which two timed maps are considered to be equal if and only if they have \emph{exactly} the same timing. Typically, however, in complexity theory one wants to regard maps whose timings are “similar” to be equal. So for example, if the functions have timings which asymptotically differ only by a linear factor (e.g. as in “big O” notation), one may want to regard them, from a complexity standpoint, to be equivalent. To capture this idea, which is fundamental to complexity theory, it is clear that one must pass to a quotient of the basic category of timed sets. However, there is a technical problem: one must describe precisely when two timed maps should be equivalent.

Unfortunately, one cannot simply use asymptotic behavior (e.g. “big O” notation) as, \emph{a priori}, it is only defined for maps from the natural numbers to the real numbers. The timing functions in which we are interested have domain an arbitrary set and codomain an arbitrary size monoid. Thus, some other mechanism must be sought: we introduce here the notion of a \emph{complexity order} to serve this purpose. A complexity order is a set of monotone endomorphisms of a size monoid, \( \mathcal{C} \subseteq \text{Mon}(M, M) \) satisfying certain properties. Given a complexity order, \( \mathcal{C} \) one can preorder timed maps: one map, \( f \), being of better \( \mathcal{C} \)-complexity than another, \( g \), if it is defined whenever the latter is, and its cost is bounded by the latter’s cost modified by a map from the order \( \mathcal{C} \), that is \( |x|_f \leq F(|x|_g) \) for all \( x \) in the domain.

\(^9\) For LOGSPACE it turns out (see later) that the appropriate size monoid is actually the natural numbers under addition: however, for example for PSPACE it does seem that maximum is more appropriate.
of \( g \) and for some \( F \in \mathcal{C} \). The maps equivalent under this preorder are then taken as being equal.

Of course, for this to make good sense, \( \mathcal{C} \) must satisfy some basic properties: it must certainly be closed to composition, but also, more technically, it must be "laxly additive." A canonical example of a complexity order is generated by the polynomial maps, \( \mathcal{P} \subseteq \text{Mon}(\mathbb{N}, \mathbb{N}) \), on the natural numbers. Another important example, closely related to "big \( O \)" notation, is generated by the linear maps, \( \mathcal{L} \subseteq \text{Mon}(\mathbb{N}, \mathbb{N}) \).

After the maps are quotiented by this \( \mathcal{C} \)-equivalence, a category of timed sets is obtained whose maps are partially ordered by their \( \mathcal{C} \)-complexity. If the complexity order, \( \mathcal{C} \), is closed to addition – which certainly implies that \( \mathcal{C} \) is laxly additive – something quite striking happens: the resulting category of timed sets naturally has the structure of a restriction category. Restriction categories are important as they provide a completely algebraic way of expressing partiality. The completeness theorem for restriction categories asserts that every restriction category is canonically a full subcategory of a partial map category: the canonical partial map category is actually formed by splitting the restriction idempotents.

In complexity theory, one always considers the running time of a function in terms of the size of its input. Yet, in the development so far, no mention of input size has been made. The effect of splitting the restriction idempotents elegantly corrects this defect. When one considers a timed map whose starting point is an idempotent the timing of that idempotent can be viewed as providing the "size" of its input. One can view the restriction idempotents as giving the cost of "reproducing" or "reading" the input both of which are, intuitively, measures of its size. In fact, these idempotents can provide a surprisingly sophisticated interpretation of size as witnessed by their use in modeling space complexity.

Splitting the idempotents has another remarkable effect. The total maps of this split category become exactly the maps whose complexity belongs to the complexity order, \( \mathcal{C} \). Thus, for example, if \( \mathcal{C} = \mathcal{P} \), then one obtains, as the total maps of the split category, exactly the polynomially timed maps.

This is striking: it is not at all obvious that there should be any relationship between complexity, given in this very concrete manner, and partiality. Yet, in a very straightforward way, one can exhibit a direct relationship. Of course, the polynomially timed maps, obtained in this manner, are by no means the PTIME maps as understood by complexity theorists. The category of timed sets allows all possible timings for all partial maps given between sets whether they are realizable by machine or not. To obtain the classical PTIME maps we need to link these ideas up with computability: for to be a PTIME map, one must also be able to realize the map by a computation on a machine.

This introduces a further technical problem: one must explain what a reasonable notion of computation is in timed sets. One approach to this is to view a computable map as one obtained by iterating a basic (total) machine step. For this approach to work requires that one can produce a well-defined notion of iteration – that is a trace on the coproduct – for categories of timed sets. To achieve this requires that the complexity order, \( \mathcal{C} \), which determines the notion of equivalence, satisfies
an additional requirement: namely, that the order is generated by maps which are “lax” with respect to addition. When this is the case there is a canonical notion of iteration which allows one to talk about iterating the state change of a machine (such as a Turing machine) which, in turn, allows one to “time” computations in a natural way.

In this manner one can obtain a categorical model of computability – in the sense of being a Turing category [2,10] – whose total maps are precisely the PTIME maps as standardly understood by complexity theorists. In particular, this suggests that one can identify the computability theory of this Turing category with the theory of PTIME complexity.

These results are explained in the last section: the development relies on a number of standard results from complexity theory and on a basic understanding of Turing categories. A similar technique allows one to obtain models of Turing categories of other low complexity classes such as ELEMENTARY, PRIM (primitive recursive). These techniques can also be applied when space is taken as the resource: to illustrate this we briefly discuss LOGSPACE computations which also organize themselves into a Turing category. Finally, we illustrate the possibility of interpreting complexity hierarchy problems as questions about functors between these Turing categories.

2 Timed Maps

A size monoid, $M = (U(M), +, 0, \leq)$ is an ordered commutative monoid (thus $(U(M), +, 0)$ is a commutative monoid, where $U(M)$ is the underlying set of $M$ and $\leq$ a partial order on $U(M)$) such that

- $0 \leq m$ for all $m \in M$;
- if $a \leq a'$ and $b \leq b'$ then $a + b \leq a' + b'$.

A basic example of a size monoid is the natural numbers under addition. Of course, the natural numbers also form a size monoid under maximum and multiplication. In general, given any commutative monoid $M$ we may place a preorder on it by setting $x \leq y$ if and only $\exists a. x + a = y$. The equivalence relation determined by $x \sim y \iff x \leq y \& y \leq x$ is then a congruence on the monoid: the quotient by this congruence is the universal size monoid associated with that commutative monoid. Formally this gives a left adjoint to the underlying functor, $\text{SzMon} \to \text{CMon}$, from size monoids to commutative monoids. Notice that under this adjunction the universal size monoid associated to a commutative group is always trivial: so size monoids are, in this sense, orthogonal to groups. In addition, this is a Galois adjunction: the category of size monoids such that $x \leq y \iff \exists a. x + a = y$ (one may think of these as “positive cones”) forms a coreflective subcategory of sized monoids and a reflective subcategory of commutative monoids.
Given a size monoid \( M \) we can form a category \( TSet(M) \)\(^{10}\), the category of \( M \)-timed sets:

**Objects:** Sets;

**Maps:** A timed map \( f : X \to Y \) is a pair \( f = (U(f), |.|_f) \) where \( U(f) : X \to Y \) is a partial function and \( |.|_f : X \to M \) is the **timing** or **cost** of the map which is defined precisely when \( U(f) \) is defined.

**Composition:** Given timed maps \( f : X \to Y \) and \( g : Y \to Z \) the composite \( fg \) has \( U(fg) = U(f)U(g) \) and \( |x|_{fg} = |x|_f + |f(x)|_g \);

**Identities:** The timed identity \( 1_X : X \to X \) has \( U(1_X) = 1_X \) and \( |x|_{1_X} = 0 \) for all \( x \in X \).

This is clearly a category whose maps have attached timings in the size monoid \( M \). Two maps in this category are equal only if they have precisely the same timings. Our objective is to now relax this latter constraint: to do this we introduce the notion of a complexity order. A **complexity order** for a size monoid, \( M \), is given by a class \( \mathcal{C} \) of monotone endomorphisms (i.e., satisfying \( P(x) \leq P(y) \) whenever \( x \leq y \), \( \mathcal{C} \subseteq \text{Mon}(M, M) \), such that:

- \( \mathcal{C} \) is **down closed**: if \( P \in \mathcal{C} \) and \( Q \in \text{Mon}(M, M) \) with \( Q(m) \leq P(m) \) for every \( m \in M \) then \( Q \in \mathcal{C} \).
- \( \mathcal{C} \) is closed to **composition**: the identity map, \( I(x) = x \) is in \( \mathcal{C} \) and, if \( P, Q \in \mathcal{C} \) then \( PQ \in \mathcal{C} \) (where \( PQ(x) = Q(P(x)) \) – as we are writing composition in diagrammatic order rather than applicative order).
- \( \mathcal{C} \) is **laxly additive**: if \( P_1, P_2 \in \mathcal{C} \) then there is a \( Q \in \mathcal{C} \) such that for all \( x \) and \( y \), \( P_1(x) + P_2(y) \leq Q(x + y) \).

We shall say a complexity order \( \mathcal{C} \) is **additive** in case we replace the last axiom by the requirement that whenever \( P, Q \in \mathcal{C} \) then \( P + Q \in \mathcal{C} \). Clearly, being additive implies being laxly additive as \( P_1(x) + P_2(y) \leq P_1(x+y) + P_2(x+y) = (P_1 + P_2)(x+y) \), so with this modification, it is still a complexity order. It is not hard to see, conversely, that a laxly additive order is additive if and only if it contains \( I + I \).

Given a complexity order \( \mathcal{C} \) and two \( M \)-timed partial maps maps \( f, g : X \to Y \) we shall say \( f \) has **better \( \mathcal{C} \)-complexity** than \( g \), denoted \( f \leq \mathcal{C} g \), in case as partial maps \( U(f) \geq U(g) \) (that is whenever \( U(f) \) is defined \( U(f) \) is defined and equal to \( U(g) \)) and there is a \( F \in \mathcal{C} \) such that for all \( x \) for which \( g \) is defined \( |x|_f \leq F(|x|_g) \).\(^{11}\)

Intuitively, one should think that a map \( f \) has better complexity than \( g \), when it is not only as least as defined as \( g \), but also its timing is no worse than \( g \)’s (up to a \( \mathcal{C} \) increment). We say \( f =_\mathcal{C} g \), that is \( f \) and \( g \) have the same \( \mathcal{C} \)-complexity if \( f \leq \mathcal{C} g \) and \( f \geq \mathcal{C} g \). This means that \( U(f) = U(g) \) and there are \( P, Q \in \mathcal{C} \) such that \( |x|_f \leq P(|x|_g) \) and \( |x|_g \leq Q(|x|_f) \).

\(^{10}\)There is a monad involved in this construction for which \( TSet(M) \) is the Kleisli category. Using a monad to model timing was described by Doug Gurr in his PhD thesis [7].

\(^{11}\)It would be more in the spirit of the asymptotic definition of order to require \( |x|_f \leq P(|x|_g) \) holds *almost everywhere* – that is it holds for all but a finite set of \( x \). While the development certainly works with this definition, as we shall see, most complexity orders already accommodate finitely many exceptions thus the gain of this complication is limited.
A complexity order $\mathcal{C}$ is said to be generated by a subset $\mathcal{S}$ of its maps, denoted $\mathcal{C} = \langle \mathcal{S} \rangle$, when every map in the order is dominated by one in the subset. Here are some examples of complexity orders:

(i) The smallest complexity order for any size monoid $M$ is generated by the identity map. In this case $f \leq_{\mathcal{C}} g$ if and only if $f$ is more defined than $g$ and $|x|_f \leq |x|_g$ whenever $g(x)$ is defined.

(ii) Given any size monoid $M$ there is always the constant complexity order generated by translations:

$$\mathcal{K}_M = \langle K_m | K_m = \lambda x. x + m, m \in M \rangle.$$  

Here $f \leq_{\mathcal{K}_M} g$ if and only if $f$ is more defined than $g$ and there is an $m \in M$ such that for all $x \in X$ with $g(x)$ defined, $|x|_f \leq |x|_g + m$.

(iii) Given any size monoid $M$ there is always the linear complexity order:

$$\mathcal{L}_M = \langle \lambda x. n \cdot x | n \cdot x = x + \ldots + x, n \in \mathbb{N} \rangle.$$  

When $M = \mathbb{N}$, note that $f \leq_{\mathcal{L}} g$ implies that, using “big O” notation, $f \in O(g)$.

(iv) When $M = \mathbb{N}$ (or $M = \mathbb{R}_{\geq 0}$) then we may consider the orders:

$$\mathcal{P} = \langle \lambda x. \sum_{i=0}^{n} a_i x^i | a_i, n \in \mathbb{N} \rangle \quad \mathcal{P}^* = \langle \lambda x. \sum_{i=1}^{n} a_i x^i | a_i, n \in \mathbb{N} \rangle.$$  

These are the polynomial complexity orders. Note that the second does not include the constant functions. If $M = \mathbb{N}$, the sums are not necessary as they are dominated by the largest power. Hence, for example, $\mathcal{P}^* = \langle \lambda x. a x^i | a, i \in \mathbb{N}, i > 0 \rangle$.

We start by observing:

**Lemma 2.1** Given any complexity order, $\mathcal{C}$, on a size monoid $M$:

(i) Each homset of $\text{TSet}(M)(X,Y)$ is preordered by $f \leq_{\mathcal{C}} g$;

(ii) If $f \leq_{\mathcal{C}} g$ and $h \leq_{\mathcal{C}} k$ in $\text{TSet}(M)$ then $fh \leq_{\mathcal{C}} gk$ (i.e. this is a preorder enrichment for $\text{TSet}(M)$);

(iii) $=_\mathcal{C}$ is a congruence on $\text{TSet}(M)$.

**Proof.**

(i) It is immediate that $f \leq_{\mathcal{C}} f$ the only difficulty is to show transitivity. So suppose $f \leq_{\mathcal{C}} g \leq_{\mathcal{C}} h$ then for $P,Q \in \mathcal{C}$ we have $|x|_f \leq P(|x|_g)$ and $|x|_g \leq Q(|x|_h)$ but this means $|x|_f \leq P(|x|_g) \leq P(Q(|x|_h))$ as $P$ is monotone. But $QP \in \mathcal{C}$ so we are done.

(ii) Suppose $|x|_f \leq P_1(|x|_g)$ and $|y|_h \leq P_2(|y|_k)$ whenever they are defined. Then,
as $\mathcal{C}$ is laxly additive there is a $Q \in \mathcal{C}$ with $P_1(x) + P_2(y) \leq Q(x + y)$ and we have:

$|x|_h = |x|_f + |f(x)|_h$

$\leq P_1(|x|_g) + P_2(|f(x)|_k)$

$\leq Q(|x|_g + |f(x)|_k)$

$= Q(|x|_{gk}).$

(iii) $\_ = \__{\mathcal{C}}$ is a congruence on $\text{TSet}(M)$ as this is so for the equivalences in any preorder enriched category.

This means that we can pass to the category $\text{TSet}(M)/\mathcal{C}$ where partial map equality is determined by $f =_{\mathcal{C}} g$. Thus, in $\text{TSet}(\mathbb{N})/\mathcal{C}$ two timed maps will be counted as being equal if they are equal as partial maps and their timings differ by at most a constant. Similarly, in $\text{TSet}(\mathbb{N})/\mathcal{L}$ two maps are counted to be equal if their timings are linearly comparable.

## 3 Timed Sets and Partial Map Categories

In this section we consider $\text{TSet}(M)/\mathcal{C}$, where $\mathcal{C}$ is an additive complexity order.

Some of the orders mentioned above are not additive: in particular, neither the order generated by the identity map, nor the order $\mathcal{K}_M$ generated by translations, is additive. On the other hand, the linear and polynomial orders, $\mathcal{L}, \mathcal{P}$ and $\mathcal{P}^*$, are additive.

The purpose of this section is to establish:

**Proposition 3.1** When $\mathcal{C}$ is additive then $\text{TSet}(M)/\mathcal{C}$ is a discrete distributive restriction category with the restriction given by:

$$f : X \to Y \quad \overset{\mathcal{F}}{\longrightarrow} \quad \overline{f} = (U(f), |f| : X \to X)$$

A distributive restriction category is a Cartesian restriction category – that is it has finite products in an appropriate partial sense (explained below) – with coproducts over which these products distribute (see [4]). It is discrete in case all diagonal maps $\Delta : A \to A \times A$ have partial inverses\(^{12}\) $\Delta^{(-1)}$.

The significance of Proposition 3.1 lies in the fact that restriction categories are abstract categories of partial maps. In particular, every restriction category is a full subcategory of a (real) partial map category (see [3,11]). Furthermore, the canonical partial map category, of which it is a full subcategory, is obtained by splitting the “restriction” idempotents to obtain a larger restriction category. From this one can extract the subcategory of total maps which can be used to form the partial map category in which the original restriction category sits (see [3]).

\(^{12}\) A map $f : A \to B$ in a restriction category has a partial inverse if there is a map $f^{(-1)} : B \to A$ with $ff^{(-1)} = \mathcal{F}$ and $f^{(-1)}f = f^{(-1)}$; partial inverses are unique.
In section 5 we carry out these formal steps for \( \text{TSet}(M)/C \). While abstractly these steps are quite routine, their realization in this case is quite striking. As we shall see, this allows the construction of a restriction category whose total maps are precisely those with timings bounded by the complexity order \( C \). This provides a direct link between complexity and partiality.

A restriction category is a category with a restriction combinator which, to any map, assigns an endomorphism of its domain:

\[
\begin{align*}
  f : A \to B \\
  \overline{f} : A \to A
\end{align*}
\]

This must satisfy just four identities:

\[
\begin{align*}
  [R.1] & \quad \overline{f}f = f \\
  [R.2] & \quad \overline{f}\overline{g} = \overline{g}\overline{f} \\
  [R.3] & \quad \overline{f}\overline{g} = \overline{g}\overline{f} \\
  [R.4] & \quad f\overline{g} = \overline{f}g
\end{align*}
\]

As \( \overline{f}\overline{f} = \overline{\overline{f}f} = \overline{f} \), the restriction of a map is always an idempotent, called a restriction idempotent, which should be thought of as a partial identity. In a restriction category a map \( f : A \to B \) is said to be total in case \( f = 1_A \); the total maps always form a subcategory.

The proof of proposition 3.1 can be broken into parts: proving that it is a restriction category, that it is Cartesian, distributive, and discrete. The verification that it is a restriction category is straightforward: we shall prove \([R.1]\) and \([R.4]\) as these show why the additive assumption is needed:

\([R.1]\) We know already that the behavior at the level of partial maps is correct: the problem is to provide a bounding function from \( C \) for \( \overline{f}f \) in terms of \( f \). We have

\[
|x|_{\overline{f}f} = |x|_f + |x|_f \leq (I + I)(|x|_f)
\]

where we use the additivity to produce the bound.

\([R.4]\) Again it is obtaining the bound for \( \overline{f}g \) in terms of that of \( f\overline{g} \) which is the only difficulty. We have:

\[
\begin{align*}
  |x|_{\overline{f}g} &= |x|_{\overline{f}g} + |x|_f = |x|_{f\overline{g}} + |x|_f = 2 \cdot |x|_f + |f(x)|_g \\
  &\leq 2 \cdot (|x|_f + |f(x)|_g) = (I + I)(|x|_f + |f(x)|_g)
\end{align*}
\]

Clearly, again, we have to use the fact that \( C \) is additive.

Thus, \( \text{TSet}(M)/C \) is a restriction category. A restriction category is Cartesian when it has a restriction terminal object and restriction (binary) products. A restriction terminal object is an object \( 1 \) with, for each object \( A \), a unique total map \( !_A : A \to 1 \) such that any map \( f : A \to 1 \) has \( f = \overline{f}!_A \).

In \( \text{TSet}(M)/C \) the terminal object is the one element set \( 1 = \{()\} \) and \( !_A \) is determined by setting \( U(!_A)(a) = () \) for every \( a \in A \) and \( |a|_A = 0 \). One then has, given an \( f : A \to 1 \), that whenever \( f \) is defined on \( a \) that \( |a|_{\overline{f}!_A} = |a|_f + |a|_{!_A} = |a|_f \).

An object, \( A \times B \), with two total projections, \( \pi_0 : A \times B \to A \) and \( \pi_1 : A \times B \to B \), is a restriction product in case for any pair of maps \( f : X \to A \) and \( g : X \to B \)
there is a unique map \(\langle f, g \rangle : X \to A \times B\) such that \(\langle f, g \rangle \pi_0 = \overline{gf}\) and \(\langle f, g \rangle \pi_1 = \overline{fg}\).

In \(\text{TSet}(M)/C\), the restriction product \(A \times B\) is the Cartesian product of the sets, where \(\pi_i, i = 0 \text{ or } 1\), has \(U(\pi_i)\) the appropriate projection in sets, and \(|(x, y)|\pi_i = 0\).

We set \(U((f, g))\) to be the pairing map, which, on \(x \in X\), is only defined when both \(f\) and \(g\) are defined, and set \(|x|_{(f, g)} = |x|_f + |x|_g\). To show that this is well-defined, that is that if \(f\) \(\Rightarrow\) \(f'\) and \(g\) \(\Rightarrow\) \(g'\) that \((f, g) \Rightarrow (f', g')\), it suffices to show that if \(f \leq_c f'\) and \(g \leq_c g'\) then \((f, g) \leq_c (f', g')\). So suppose \(P_1, P_2 \in C\) with \(|x|_f \leq P_1(|x|_{f'})\) and \(|x|_g \leq P_2(|x|_{g'})\) then

\[
|x|_{(f, g)} = |x|_f + |x|_g \leq P_1(|x|_{f'}) + P_2(|x|_{g'}) \leq Q(|x|_{f'} + |x|_{g'}),
\]

where we are using the fact that \(C\) is laxly additive.

To verify that \((f, g) \pi_0 = \overline{gf}\), first recall that this is so for the underlying partial maps between sets, so we need only check the bounds. However, for this we have:

\[
|x|_{(f, g)} \pi_0 = |x|_{(f, g)} + |(f(x), g(x))| \pi_0 = |x|_{(f, g)} + |x|_f + |x|_g = |x|_g + |x|_f = |x|_{gf}.
\]

To establish uniqueness, suppose \(h : X \to A \times B\) satisfies these conditions, then certainly \(U(h) = U((f, g))\); so that it remains to show that there are \(P, Q \in C\) with \(|x|_{(f, g)} \leq P(|x|_h)\) and \(|x|_h \leq Q(|x|_{(f, g)})\). We have \(h \pi_0 = \overline{gf}\) so we have a \(P\) and \(Q\) such that \(|x|_{h \pi_0} \leq P(|x|_{gf})\) and \(|x|_{gf} \leq Q(|x|_{h \pi_0})\). Now observe \(|x|_{gf} = |x|_f + |x|_g = |x|_{(f, g)}\) and \(|x|_{h \pi_0} = |x|_h + |h(x)| \pi_0 = |x|_h\), so that these bounds suffice.

We have now established that \(\text{TSet}(M)/C\) is a Cartesian restriction category.

Given this structure, the diagonal map \(\Delta = \langle 1_A, 1_A \rangle : A \to A \times A\) is clearly a zero cost map and so has a zero cost partial inverse whose domain of definition is exactly the diagonal elements and which has \(\Delta^{-1}(x, x) = x\).

Finally, we must show that \(\text{TSet}(M)/C\) has coproducts: the empty set is clearly the initial object. The binary coproduct \(A + B\) is just the disjoint union of the sets: the injections \(\sigma_0\) and \(\sigma_1\) are the inclusion maps in sets with a zero timing. The copairing map for \(f : A \to X\) and \(g : B \to X\) is the map \(\langle f, g \rangle : A + B \to X\), where \(U((f, g))\) is the usual copairing for sets and partial functions \((U((f, g)))(a) = f(a)\) for \(a \in A\) and \((U((f, g)))(b) = g(b)\) for \(b \in B\) with timing \(|a|_{(f, g)} = |a|_f\) for \(a \in A\) and \(|b|_{(f, g)} = |b|_g\) for \(b \in B\). As before we must show this is well-defined: this requires finding a bounding function which works for both components and - the sum of the bounding functions of the individual components will clearly work. It is now routine to show one has a coproduct. Furthermore, as the canonical map

\[
\langle 1_A \times \sigma_0 | 1_A \times \sigma_1 \rangle : A \times B + A \times C \to A \times (B + A)
\]

has zero cost and is an isomorphism in sets, it follows easily that \(\text{TSet}(M)/C\) is distributive.

This completes the proof of proposition 3.1.

\[\text{In order to make the product and coproduct well-defined, a simultaneous bound for all the components is required. While for finite products and coproducts this is simply given by adding the individual bounds, this technique does not work for infinite products or coproducts and, indeed, }\text{TSet}(M)/C\text{ does not have these limits.}\]
It should be noted that in $\text{TSet}(M)/C$, the total maps are (up to equivalence) zero cost maps. Thus, even when $C = P$ this allows only maps with timings which are bounded by a constant to be total. Furthermore, these maps take no account of the size of their inputs (or outputs for that matter). These are major defects from the complexity perspective: however, as we shall see shortly, they are all corrected by the formal construction of splitting idempotents (see section 5).

4 Joins, Ranges, and Iteration

Given any restriction category there is always an induced restriction partial order on parallel maps; it is defined by $f \leq g$ if and only if $\overline{f}g = f$. This makes any restriction category a partial order enriched category. Observe first that this restriction ordering is actually the converse of the complexity ordering:

**Lemma 4.1** In $\text{TSet}(M)/C$, for additive $C$, $f \leq g$ in the restriction order if and only if $g \leq_C f$ in the complexity order.

**Proof.** For suppose $f \leq g$ then $\overline{f}g = f$ so $U(g)$ is at least as defined as $U(f)$ but, furthermore, on the timing we have a $P \in C$ such that $|x|_g \leq |x|_f + |x|_g = (I + P')(|x|_f)$ where the desired bound is $I + P'$ which, as $C$ is additive, is certainly in $C$.

In any restriction category we say that two parallel maps $f$ and $g$ are compatible, $f \sim g$, in case $\overline{f}g = \overline{gf}$. In sets and partial maps this means that they take the same value when they are both defined: thus, clearly, the join of the partial maps in sets, $f \lor g$, exists. It is not the case that in $\text{TSet}(M)/C$, for an arbitrary $M$, that one can join maps in this manner. Technically a restriction category has joins if whenever $f$ and $g$ are parallel maps with $f \sim g$ then there is a parallel map $f \lor g$ such that, with respect to the restriction order, it is the join – that is $f \leq f \lor g$, $g \leq f \lor g$, and if $f \leq h$ and $g \leq h$ then $f \lor g \leq h$ – and that join is stable – that is $h(f \lor g) = (hf) \lor (hg)$: in a restriction category this then implies universality, $(f \lor g)k = (fk) \lor (gk)$.

In fact, from the general theory [4], *any* distributive restriction category is an “extensive” restriction category. Extensive restriction categories always have disjoint joins of partial maps. This provides:

**Proposition 4.2** When $C$ is additive, $\text{TSet}(M)/C$ has finite disjoint joins of parallel maps.

This means that when $f, g : A \rightarrow B$ are parallel maps with disjoint domains, that is $\overline{f}g = \emptyset = \overline{gf}$, where $\emptyset$ is the empty partial map then the join of $f$ and $g$ exists: it
is written \( f \sqcup g \) to emphasize the disjointness. For a general \( M \) it will not be the case that joins of compatible maps exist, however, if we assume \( M \) has \textit{binary} infima, which are preserved by addition (in the sense that \((x \wedge y) + z = (x + z) \wedge (y + z)\)), then we have:

**Proposition 4.3** When \( M \) has \textit{binary} infima which are preserved by addition and \( C \) is additive, then \( \text{TSet}(M)/C \) has finite joins of all compatible maps.

**Proof.** We define \( f \lor g \) to have \( U(f \lor g) = U(f) \lor U(g) \) and

\[
|x|_{f \lor g} = \begin{cases} 
|x|_f & f(x) \downarrow \text{ and } g(x) \uparrow \\
|x|_g & f(x) \uparrow \text{ and } g(x) \downarrow \\
|x|_f \land |x|_g & f(x) \downarrow \text{ and } g(x) \downarrow 
\end{cases}
\]

It is left to the reader to check this is well-defined. That it has all the required properties is also straightforward to check. \qed

Certainly when \( M = N \) the requirements\(^{14}\) of Proposition 4.3 are satisfied so, in this case, we have finite joins of compatible maps. As \( N \) is well ordered, we may also form “ranges”\(^{10,5}\): the range associates to each map \( f : A \to B \) a restriction idempotent \( \hat{f} : B \to B \) such that \( f \hat{f} = f \), \( \hat{f} g = \hat{f} g \), and \( \hat{f} \hat{g} = \hat{f} g \). A range operator provides an algebraic description of the image of a map. Sets and partial maps have ranges because every map has a partial section. For an additive \( C \), \( \text{TSet}(N)/C \), also has partial sections for every map: however, the section must be chosen carefully!

We briefly outline these ideas. Given a partial map \( f : A \to B \) a partial section is a map \( g : B \to A \) such that \( g f = \text{\textellipsis} \) and \( f g f = f \): if a map has a partial section \( g \) then \( \hat{f} = \hat{f} \). Given \( f : A \to B \) in \( \text{TSet}(N)/C \) we may define a zero cost section \( g \) so that \( g(y) \) is defined only if \( y = f(x) \) for some \( x \in X \), further, when it is defined, \( f(g(y)) = y \) and \( |g(y)|_f = \min\{|x|_f \mid f(x) = y\} \) (here we use the well-ordering of \( N \)).

This last requirement is necessary as can be seen from the proof that \( f g f = f \).

The key steps in the bounding argument are

\[
|x|_{g f} = |x|_f + |f(x)|_g + |g(f(x))|_f = |x|_f + 0 + |g(f(x))|_f \leq |x|_f + |x|_f
\]

where the last step is possible because of the way we chose \( g(y) \) to have \( |g(y)|_f \) minimal. We state without further justification – because it is quite remarkable that these categories are so well-structured:

**Proposition 4.4** When \( C \) is additive, \( \text{TSet}(N)/C \) is a discrete range restriction category with finite joins.

The last aspect of structure we require from these categories is that it should be possible to iterate maps. A category has iteration precisely when it is traced on

---

\(^{14}\)In this case, as \( N \) is well-ordered, it may be tempting to think that as we have arbitrary non-empty infima that we will have arbitrary joins for any set of compatible parallel maps (i.e. not just for finite sets). However, this is \textit{not} the case: the join of an infinite collection will in general not be well-defined as finding uniform bounds is only possible for finite families.
the coproduct \([8,9]\). The iteration combinator has the following form:

\[
\frac{h : A \to A + B}{h^\uparrow : A \to B}
\]

Iteration

However, in an extensive restriction category any map \(h : A \to A + B\) can be broken down into two disjoint maps \(h = f \sqcup g : A \to A + B\) where \(f : A \to A\) and \(g : A \to B\) allowing the iteration to be re-expressed as a “Kleene wand”:

\[
\frac{f : A \to A \quad g : A \to B \quad f \perp g}{f^\uparrow g : A \to B}
\]

Kleene Wand

where \(f \perp g\) means \(\bar{f} \bar{g} = \emptyset\). Intuitively \(f\) is iterated until the guard \(g\) is encountered at which point an output is produced. In sets and partial functions the canonical Kleene wand may be expressed as \(f^\uparrow g = \bigsqcup_{i=0}^{\infty} f^i g\). Of course, as both \(f\) and \(g\) are partial \(f^\uparrow g\) will certainly be partial. However, note that there is an added source of partiality as the iteration of \(f\) may never hit the guard.

The Kleene wand, in order to correspond to a well-behaved trace, must satisfy some basic equations:

- **[W.1]** When \(f \perp h\) then \((fg)^\uparrow h = h \sqcup f((gf)^\uparrow(gh))\);
- **[W.2]** When \(f \perp g\), \(g \perp h\), and \(f \perp h\) then \((f \sqcup g)^\uparrow h = (f^\uparrow g)^\uparrow(f^\uparrow h)\);
- **[W.3]** When \(f \perp g\) then \((f^\uparrow g)^\uparrow h = f^\uparrow(gh)\);
- **[W.4]** When \(f \perp g\) then \(1_A \times (f^\uparrow g) = (1_A \times f)^\uparrow(1_A \times g)\);
- **[W.5]** When \(f \leq f', g \leq g',\) and \(f' \perp g'\) then \(f^\uparrow g \leq f'^\uparrow g'\).

The first identity allows finite unwavings of the iteration:

\[
f^\uparrow g = g \sqcup (f^\uparrow fg) = g \sqcup fg \sqcup (f^\uparrow ffg) = ...
\]

Note that the third identity tells us that \(f^\uparrow g = (f^\uparrow \bar{g})g\) where \(f^\uparrow \bar{g}\) provides a more primitive form in which \(\bar{g}\) may be regarded as implementing a predicate guard. The fourth identity allows one to trace in a context \(A\).

Sets and partial maps satisfy all these identities and we shall use this to establish that \(\text{TSet}(M)/C\) also has iteration which is defined in the obvious manner: \(f^\uparrow g(x) := g(f^n(x))\) when this is defined for some \(n\) (at most one \(n\) will work) and \(|x|_{f^\uparrow g} := (\sum_{i=0}^{n-1} |f^i(x)|_f) + |f^n(x)|_g\). The only technical problem is then to show that iteration, as we have defined it, is actually well-defined with respect to the equivalence on maps. For this it is necessary to demand more of the complexity order \(C\).

We say that a complexity order is **lax** if it generated by a class of functions which are lax in the sense that \(P(m) + P(n) \leq P(m+n)\) (we always have \(0 \leq P(0)\)). Note that sums and composites of lax functions are lax.

**Proposition 4.5** When \(C\) is additive and lax \(\text{TSet}([N])/C\) has iteration.

**Proof.** The equations all follow easily from the fact that sets and partial maps
satisfy the equations. The only difficulty is to prove that if \( f \leq_C f' \) and \( g \leq_C g' \) then \( f \uparrow g \leq_C f' \uparrow g' \). Because of these inequalities we have:

\[
\begin{align*}
    f'(x) \downarrow \Rightarrow |x|_f & \leq P(|x|_{f'}) \\
    g'(x) \downarrow \Rightarrow |x|_g & \leq Q(|x|_{g'})
\end{align*}
\]

Suppose now that \( f' \uparrow g'(x) \) is defined then clearly \( f \uparrow g(x) \) will be defined and in fact their evaluations on this element will be the same. That is there is a unique \( n \) so that \( g(f^n(x)) = g'(f^n(x)) \) and they are both defined. Consider the cost of these:

\[
|x|_{f \uparrow g} = |x|_{f^n g} \\
= |x|_f + |f(x)|_f + \ldots + |f^n(x)|_g \\
\leq P(|x|_{f'}) + P(|f'(x)|_{f'}) + \ldots + Q(|f^n(x)|_{g'}) \\
\leq (P + Q)(|x|_{f'}) + (P + Q)(|f'(x)|_{f'}) + \ldots + (P + Q)(|f^n(x)|_{g'}) \\
\leq (P + Q)(|x|_{f'} + |f'(x)|_{f'} + \ldots + |f^n(x)|_{g'}) \\
\leq (P + Q)(|x|_{f \uparrow g'}). \\
\]

\[
\square
\]

The class \( \mathcal{L} \) (linear time) is lax. Significantly the class \( \mathcal{P} \) is not lax: the problem is that constant functions are not lax. However, the class \( \mathcal{P}^* \) is lax as we removed the constant functions! Recall that constant functions are only important when the input sizes can be zero: we shall remove this possibility shortly.

It is worth remarking that categories of timed sets often have more than one trace. Consider, for example, \( TSet/\mathcal{L} \) where \( \mathcal{L} \) is the linear order on the natural numbers with addition. As described above \( \mathcal{L} \) is lax so that there is a trace as defined above. However, on any size monoid (such as \( \mathbb{N} \)) which has a maximum \( \mathcal{L} \) is also an order for that monoid where addition is replaced by the maximum (e.g. for \( \mathbb{N} \) regarded as size monoid with addition given by \( \max(x, y) \)). Furthermore, the two categories of timed sets are then actually isomorphic. However, significantly, the traces given by the above, with respect to addition and maximum, are not the same. This not only demonstrates that a category can have more than one trace but also is a reminder that when one wishes to talk of iteration one may have a choice of iteration!

### 5 Splitting Restriction Idempotents

Our next objective is to split the restriction idempotents of \( TSet(M)/\mathcal{C} \) to produce \( \text{Split}(TSet(M)/\mathcal{C}) \). This is a standard construction (also – for all idempotents – known as the Karoubi envelope or Cauchy completion) which turns the idempotents \( e \) and \( e' \) into objects and takes as the maps \( f : e \to e' \) those maps in the original category such that \( ef e' = f \). The construction is remarkable as it preserves almost all the properties of the original category. In particular, if one starts with a distributive restriction category then the result will be a distributive restriction category. Furthermore, ranges, joins, and iteration are all transferred onto the splitting.
Let us consider in detail what an object of $\text{Split}(\text{TSet}(M)/C)$ looks like. Recall that a restriction idempotent is just a timed partial identity: it is easy to see that the elements which are not in the domain of the underlying partial identity will not play any role. So it suffices to consider the objects whose underlying partial identity is actually the identity: this makes the only important information the timing of the idempotent. Thus, such an object is essentially a set $A$ with a map which assigns to each element $a \in A$ an “size” $\|a\|_e$ (these are called sized sets in [6]). A timed map $f : e \to e'$ between two sized sets, in this sense, is a partial map such that

$$\|x\|_e + |x|_f + \|f(x)\|_{e'} \leq P(|x|_f) \quad (P \in C)$$

(recall we must have $efe' =_C f$). We may think of this as requiring that the timing of a map cannot be “faster” than what is required to read the input and produce the output! Stated like this it seems to be a reasonable requirement, however, it does mean that even “doing nothing”, in the sense of just passing on the input unchanged involves actually reading the input and writing an output.

In particular, note that a map $f : e \to e'$ is total in case $f = e$ which means in terms of timing that

$$|x|_f \leq P(\|x\|_e) \quad (P \in C).$$

If this condition holds, we say that $f$ has time complexity in $C$. (The other bounding identity holds already from the requirement above.) To have time complexity in $C$ means that the timing of $f$ is $C$-bounded by the size of its input: so, intuitively, $f$ “runs” in time $P(n)$ where $P \in C$ and $n \in M$ is the size of the input.

Summarizing this discussion we have:

**Proposition 5.1** When $C$ is additive $\text{Split}(\text{TSet}(M)/C)$ is a discrete distributive restriction category in which the total maps are precisely the maps whose time complexity lies in $C$. Furthermore, if $\text{TSet}(M)/C$ has joins, ranges, or iteration then so does $\text{Split}(\text{TSet}(M)/C)$.

There is a technical move we must make at this stage to avoid elements with zero size – which are undesirable in complexity calculations. Every object in $\text{Split}(\text{TSet}(M)/C)$ has a unique total map to the terminal object $!: X \to 1; x \mapsto ()$ and $|x|_! = \|x\|$. Now there is a subobject of the object $1 = \{()\}$ given by splitting the total idempotent $\star : 1 \to 1$ where $\|()\| = 1$. By slicing the category over the object $\star$, $\text{Split}(\text{TSet}(M)/C)/\star$ (note here the objects are total maps $h_X : X \to e$ with maps $f : X \to Y$ with $fh_Y \leq h_X$) we make $\star$ the restriction final object.

We shall say that $C$ is a **pointed** complexity order if $P(0) = 0$ for all $P \in C$. Note that $L$ and $P^*$ are pointed complexity orders. We observe:

**Lemma 5.2** If $C$ is a pointed and additive complexity order then an object $Y \in \text{Split}(\text{TSet}(M)/C)$ has a total map to $\star$ (it must be unique) if and only if each element of $Y$ has a non-zero size.

From now on we shall avoid zero size elements by working in $\text{Split}(\text{TSet}(M)/C)/\star$ for a pointed complexity order $C$.  

124
6 Computability in Timed Sets

The total maps in PTSet := \text{Split}(\text{TSet}(\mathbb{N})/\mathbb{P}^*)/\ast$ are by no means the standard PTIME maps of complexity theory as there was no requirement that these maps be realizable by a computation. We have only managed, by construction, to arrange that the “timings” – which are quite arbitrarily given – of all total maps must be polynomially bounded on their input sizes. To obtain, for example, the standard notion of PTIME we must restrict to those timed maps which can be realized by a Turing machine (or some variant thereof – possibly with input and output tapes) in a number of steps which is $\mathcal{P}$-equivalent to the given timing.

Our objective, however, is not merely to cut out the PTIME maps from PTSet but to show how they can be seen to sit in a model of computability in which the total maps are the PTIME maps. More precisely we shall show that the “computable” maps in PTSet form a Turing category in which the total maps are exactly the PTIME maps.

A Turing category may be described as a Cartesian restriction category with a special object $T$, called a Turing object, such that:

- Every object in the category is a retract of $T$.
- There is an application map, also called a Turing morphism, $\bullet : T \times T \to T$ such that for every (partial) map $f : T \times T \to T$ there is a total map $\tilde{f} : A \to T$, called an index of $f$, such that:

$$
\begin{array}{c}
T \times T \xrightarrow{\bullet} T \\
\downarrow{\tilde{f} \times 1_T} \downarrow{f} \\
T \times T
\end{array}
$$

Turing categories provide a unifying formulation of abstract computability (see [2]), and when the category has joins and is discrete, one can obtain many standard results from computability theory.

We show how to obtain a Turing category in two stages: first, we shall view Turing machines (in the standard sense) as providing a “program object” which acts on lists of bits. Second, we shall describe when a program object makes its values a Turing object so that the programmable maps form a Turing category. Thus, our discussion starts by introducing how the set of Turing machines may be viewed as a “program object”, $P$. Programs can be evaluated on values (inputs), $A$, to produce new values (outputs), thus, associated with a program object is an evaluation (partial) map $\text{ev} : P \times A \to A$: we then refer to $P$ as being an $A$-program object. We say a map is “$P$-programmable” if it is of the form $f : A \to A; x \mapsto \text{ev}([f], x)$, where $[f] : 1 \to P$ is an element of $P$, that is a program. Clearly we would like the programmable maps to form a Cartesian restriction category: to achieve this, however, requires more structure.

In order, for an $A$-program object to be able to describe maps from $A^n \to A^m$ it suffices to describe programmable maps $A^n \to A$ (and when $m = 0$ the total map $A^n \to 1$ must be programmable). To describe programmable maps $A^n \to A$ it suffices to be able to encode $A^2$ in $A$ in a programmable way and this leads us to
demanding that the values themselves have a special property.

In a Cartesian restriction category $X$ an object $A$ is a powerful in case there are total maps $s_X : A \times A \to A$, $s_1 : 1 \to A$ and partial maps $P_0, P_1 : A \to A$ such that $s_X(P_0, P_1) = 1_{A \times A}$ (and $s_1 A = 1_l$). The terminal object $1$ is always a trivial example of a powerful object. A more interesting example, which we shall use below, is $\text{List}(\text{Bool})$ (lists of Booleans) in $\text{LTSet}$, with size given by $\| x \| = 2 \cdot (1 + \text{len}(x))$ (i.e., twice the length of the list plus one). The map $s_1 : 1 \to \text{List}(\text{Bool})$ picks out the empty list. There are then linear time maps $s_X$, $P_0$, and $P_1$ which code and decode pairs:

$$s_X(b : bs, b' : bs') = 1 : b : 1 : b' : s_X(bs, bs') \quad P_0(1 : b : \_ : \_ : rs) = b : P_0(rs)$$
$$s_X([], b' : bs') = 0 : 0 : 1 : b' : s_X([], bs') \quad P_0(0 : 0 : \_ : \_ : rs) = []$$
$$s_X(b : bs, []) = 1 : b : 0 : 0 : s_X(bs, []) \quad P_1(\_ : \_ : 1 : b' : rs) = b' : P_1(rs)$$
$$P_1(\_ : \_ : 0 : 0 : rs) = []$$

Given a powerful object $A$, an $A$-program object is an object $P$ which has total operations $\text{comp}, \text{pair} : P \times P \to P$ together with total points $l_0, l_1, l, J : 1 \to P$ and a partial evaluation map $ev : P \times A \to A$ such that:

![Diagram](image-url)

Program object requirements can be explained as follows. The first diagram states that there must be an identity program. The second says that one must be able to compose programs. The third says that there is an operation, pair, for combining two programs which you wish to apply to a single input in order to produce a pair of answers encoded. The last three diagrams require that there are programs for projecting from an encoded pair and to the terminal object. Clearly the codes of a Turing object include these required elements and can be combined in these ways: so every Turing object is automatically a program object. The converse, of course, is not true.

A map $f : A \to A$ is said to be $P$-programmable in case there is an element $[f] : 1 \to P$ such that

$$P \times A \xrightarrow{ev} A$$

We shall indicate that $X$ is retract of $A$ by writing $X \triangleleft^s_r A$, where $sr = 1_X$ and
rs is an idempotent of A. A retract of A, \( X \triangleleft^r_s A \), is a P-programmable retract in case the idempotent \( rs : A \rightarrow A \) is P-programmable. If \( X \triangleleft^r_s A \) and \( Y \triangleleft^u_v A \) are P-programmable retracts of A then a map \( h : X \rightarrow Y \) is P-programmable if the map \( A \rightarrow^r X \rightarrow^h Y \rightarrow^u A \) is P-programmable.

Note that the conditions of P being an A-program object ensure that \( 1_A \) is computable and if \( f \) and \( g \) are computable \( fg \) is computable: \( \langle\langle [g], [f] \rangle\rangle \comp, 1_A \rangle \ev = \langle\langle [g], [f], 1_A \rangle\rangle \ev = fg \). Furthermore, these conditions ensure that all powers \( A^n \) are P-programmable retracts of A (including when \( n = 0 \)) and, thus, we may form a category, \( \text{Prog}_{P/A}(X) \), of P-programmable maps in \( X \) whose objects are just the powers of \( A \). In fact, it is not hard to prove:

**Proposition 6.1** If \( A \) is a powerful object and \( P \) is an A-program object in a Cartesian restriction category then the subcategory of P-programmable maps, \( \text{Prog}_{P/A}(X) \), forms a Cartesian restriction subcategory.

One thing which is perhaps less obvious concerns how the restriction is defined: one uses the fact that \( \overline{f} = \langle 1, f \rangle \pi_0 \); if \( f \) is programmable then \( \overline{f} = \langle 1, f \rangle \pi_0 = \langle\langle (1, [f]) \rangle\rangle \comp, 1_A \rangle \ev \).

Finally, in order to obtain a model of computability, we wish to understand when an A-program \( P \) object turns \( A \) into a Turing object in the subcategory of P-programmable maps. The following is the main formal observation of this section:

**Proposition 6.2** If \( X \) is a Cartesian restriction category with a powerful object \( A \) and an A-program object \( P \) such that \( P \) is a P-programmable retract of \( A \) and \( \comp, \ev, \pi_0, \pi_1, \) and \( 1 \) are all P-programmable then \( \text{Prog}_{P/A}(X) \) is a Turing category.

**Proof.** Define the program \( Q := \langle(P_0, P_1) f \rangle \) then

\[
\begin{array}{c}
\xymatrix{
A \times A \ar[rr]^{(P_0, P_1) \times 1} \ar[rrd]_{1 \times s_x} & & A \times A \ar[rr]^{S_x \times \overline{P}} \ar[rrd]_{1 \times s_x} & & A \times A \ar[rr]^{r_P \times s_x} \ar[rrd]_{1 \times s_x} & & P \times A \\
A \times A \ar[rr]_{s_x \times 1} \ar[rrd]_{s_P \times 1} & & A \times A \ar[rr]_{s_x \times 1} \ar[rrd]_{s_P \times 1} & & A \times A \ar[rr]_{r_P \times 1} \ar[rrd]_{s_x} & & A \\
(Qs_P) \times 1 \times 1 \ar[rr]_{s_P \times 1} \ar[rrd]_{Q \times 1} & & A \ar[rr]_{s_x} \ar[rrd]_{(P_0, P_1)} & & A \times A \ar[rr]_{(P_0, P_1)} \ar[rrd]_{s_x} & & A \\
A \times A \ar[rr]_{s_x} \ar[rrd]_{s_x} & & A \ar[rr]_{(P_0, P_1)} \ar[rrd]_{s_x} & & A \ar[rr]_{(P_0, P_1)} \ar[rrd]_{s_x} & & A \\
} \end{array}
\]

where \((Q \times 1)s_PS_x\) is the required total map and the Turing morphism is \( \bullet := ((P_0, P_1) \times 1)(r_P \times s_x)\ev \).

Notice that the proof uses all the structure of a programming object and the powerfulness of \( A \). In particular, the definitions of the Turing morphism and of the index use pairing and composition non-trivially.

Our aim is to apply this to the set of Turing machines regarded as List(Bool)-program object in PTSet with the usual timing of evaluation. First note that regarding the program object, \( P \), as the set of specifications of Turing machines...
certainly means that $P$ may be viewed a programmable retract of $A = \text{List}(\text{Bool})$. Some bit strings may not be legal specifications of a Turing machine but recognizing the legal specifications can certainly be achieved in polynomial time – indeed in linear time as the legal specification can be given by a regular language. The composition and pairing function must take in two specifications of Turing machines and modify them to produce, respectively, the composite or the interleaved pairing. Composition is clearly programmable map on pairs of programs: it basically involves identifying the starting state of the second machine with the final state of the first. Thus, it can be done in linear time. Somewhat trickier is to see that pairing is programmable: the trick is to view pairing as a composite of duplicating the input (onto “odd” and “even” positions on the tape) modifying the first program to act on only “even” bits and composing it with the second program modified to only act on “odd” bits. This can be done in linear time and is certainly programmable and total.

The only remaining difficulty is to show that the map $ev$ is programmable. Recall that the way we defined evaluation was to run the specified Turing machine on the input: now we are asking that there be a universal Turing machine which can interpret any Turing machine specification in a manner so as to make the two maps equivalent with respect to the complexity order (here $P^*$). Fortunately, that any Turing machine can be simulated by a universal Turing machine with only a loss of an $O(\log n)$ factor in performance is well-known (see [1] Theorem 1.9 or [12] in the proof of the “Time Hierarchy Theorem” 9.10). This tells us that evaluation is programmable.

This discussion shows that the conditions of Proposition 6.2 are satisfied and yields:

**Corollary 6.3** The maps which are programmable by a Turing machine in polynomial time in $\text{PTSet}$ form a Turing category, $\mathcal{T}_{\text{ptime}}$, whose total maps are precisely the $\text{PTIME}$ maps.

As the best known simulation of a Turing machine by a universal Turing machine has an order $\log n$ overhead, one cannot quite make the argument (as presented) work for linear time computations. However, for any additive complexity order which contains $n \log n$ the argument works verbatim. Hence, this provides a number of examples of Turing categories whose total maps have low time complexity.

We have in these arguments relied on having a priori a grasp of how a Turing machine is timed. However, it is worth remarking that this too can be explained within this formalism using the trace. A Turing machine has the property that its evaluation map is given by iterating a total transition map, that is $ev = \text{step} \lor \text{halt}$ where $\text{step} \lor \text{halt} = 1_{P \times A}$. The existence of a total transition map is a key ingredient of being a (deterministic) machine. By modeling the machine steps in $\text{PTSet}$ one adds timing. Notice, however, that to arrange that each step has unit cost we are essentially forced to assume that the input has unit cost. This means these maps do not live in $\mathcal{T}_{\text{ptime}}$. On the other hand, $\mathcal{T}_{\text{ptime}}$ does inherit the trace of $\text{PTSet}$ and, in fact, the evaluation map – the Turing morphism we consider above – is given by such an iteration (up to the equivalence determined by $P^*$). However, because
the timing of a single step is now dominated by the cost of “reading” its input (i.e. the size of the tape) this internal machine, while being present – and in the end having the same effect – does not reflect well the usual step-counting intuition for measuring time complexity. Thus, it seems that the usual intuition is better served as an iteration in \( \text{PTSet} \) with unit timing (as above) together with input and output functions to lists of bits to which restore the usual sizing of the input and output.

The arguments above all rely, mimicking the approach taken in complexity theory, heavily on the machine model and the way resources are measured. Changing the machine model, of course, changes what can be computed and also the overhead of simulation. To illustrate this we briefly discuss \( \text{LOGSPACE} \) computations. Recall that a transducer is a Turing machine with a read only input tape (on which one can go backward and forward), a read/write working tape, and a write only output tape (on which one can only write and move right): the maximum length that the work tape attains during a computation is the space resource which is measured – this is abstractly now to be the “timing” of the map. The manner of composing transducers is crucial: this, in particular, determines how we set the “timing” of the identity on a list of bits. A transducer thinks of its input as being generated “by need”: if it requires the \( n^{\text{th}} \) bit it simply runs the transducer which generates its input, throwing away all the bits it generates, until the required bit is produced. It then proceeds with its calculation having obtained the required bit. This is extremely time inefficient but it is space efficient. Minimally to do this one must be able to actually count the bits one must throw away and, for this, one needs roughly \( O(\log n) \) space. Therefore, the cost of the “reading” the input (the idempotent) on a list of bits is set to be the logarithm (base 2) of the length of the list (rounded away from zero). Furthermore, somewhat surprisingly, the “timing” of composition is given by addition under the linear complexity order, \( \mathcal{L} \). Thus, the category of \( \text{LOGSPACE} \) computations is carved out from \( \text{LTSet} := \text{Split} \left( \text{TSet} \left( \mathbb{N} \right) / \mathcal{L} \right) / \ast \). Furthermore, the program object we use this time is the set of transducer specifications on lists of bits.

Again we wish to use Proposition 6.2 to show this category is a Turing category. For this we need to check the requirements. Clearly legal transducer specifications, represented as list of bits, can be recognized using a log space computation so \( P \) is a programmable retract of \( A \). Clearly, also, \( A \) is still a powerful object with respect to the same maps as above which are all clearly log space. It is less obvious how composition and pairing of transducers can be performed in log space. Composition is implemented by the second program calling on the first program whenever it moves the input head, the first program provides the input required by running itself until the required output is generated. This is achieved by modifying the second program’s read instructions to call the first program and to record a position and can be achieved in log space. Once one has composition the pairing is straightforward using the technique described above: one expresses it as a composite of duplicating the input, modifying the first program to work on “odd” bits and the second to work on “even” bits. Clearly, these modifications can all be managed in \( \text{LOGSPACE} \). For the program evaluation map itself we recall that it is well-known that a universal
Turing machine (or transducer) can simulate a specified transducer with only a constant factor of degradation in space efficiency (see exercises in [12,1]).

We therefore have:

**Corollary 6.4** The maps which are programmable by transducers in logarithmic time in LTSet form a Turing category, $\mathcal{T}_{\text{logspace}}$, whose total maps are precisely the LOGSPACE maps.

It is also worth remarking that there is a functor $V : \mathcal{T}_{\text{logspace}} \to \mathcal{T}_{\text{ptime}}$ which takes a LOGSPACE computation to a PTIME computation using the fact that a computation in space $s$ can be performed in time $2^{O(s)}$ (using [1] Theorem 4.2 for example): the only subtlety is that one must actually “slow down” space computations which do not use the full exponential time so that equivalent programs are taken to equivalent programs. Clearly, this functor preserves the meaning in sets and partial maps:

![Diagram](image)

This shows how relationships between functional complexity classes give rise to functorial relationships between their Turing categories. Whether there exists an isomorphism between $\mathcal{T}_{\text{logspace}}$ and $\mathcal{T}_{\text{ptime}}$ is, of course, an intriguing question equivalent to the open problem whether PTIME = LOGSPACE.

7 Conclusion

The objective of this work was to provide concrete models of computability, as embodied in Turing categories, in which the total maps belong to a specific functional complexity class (such as PTIME and LOGSPACE). The constructions we have provided do achieve this. Furthermore, they closely mimic the standard approach taken in complexity theory. This suggests that the results of complexity theory may be mapped fairly directly into categorical facts.

References


---

15It is immediately clear that if this functor is an isomorphism then PTIME = LOGSPACE. The converse not obvious but, nonetheless, is true. From the assumption that PTIME = LOGSPACE, just for total functions, one can prove that this $V$ must be an isomorphism for all “computable” functions. Here is a sketch of the proof: the idea is to view an arbitrary computation, $f$, as a function which has a time bound provided by an additional argument, so that $f(x) = \exists t. f'(x,t)$. The computation $f'(x,t)$ is total and, in fact, when $t$ is represented in unary, actually linear time. This means, by assumption, this program can be transformed into a LOGSPACE program. However one can then do an iterative search over time using this program to show that $f(x)$ can be implemented in LOGSPACE.
Cockett et al.


Time Bounds for General Function Pointers

Robert Dockins  Aquinas Hobor
rdockins@cs.princeton.edu  hobor@comp.nus.edu.sg
Princeton University  National University of Singapore

Abstract

We develop a logic of explicit time resource bounds for a language with function pointers and semantic assertions. We apply our logic to examples containing nontrivial "higher-order" uses of function pointers and we prove soundness with respect to a standard operational semantics. Our core technique is very compact and may be applicable to other resource bounding problems, and is the first application of step-indexed models in which the outermost quantifier is existential instead of universal. Our results are machine checked in Coq.

Keywords: Step-indexed models, Termination

1 Introduction

We define a minimal Halting Assert Language with two distinctive features: function pointers and semantic assertions. Semantic assertions are program commands that assert the truth of a formula in logic at a program point. Although semantic assertions have runtime behavior equivalent to skip, they are useful during static analysis (e.g., [5]) and as a mechanism for ensuring that the intermediate states of programs meet set invariants. Semantic assertions may seem benign, but their inclusion in a language with function pointers leads to an unpleasant contravariant circularity. Most domains containing a similar circularity in their semantic models (e.g., concurrency with first-class locks, self-modifying code) are quite complex in ways unrelated to the circularity. We consider HAL to be a test bed for semantic techniques that may be applicable in richer settings in the future.

We design a program logic of explicit time resource bounds for HAL. Programs verified in our logic are guaranteed to satisfy all invariants given in assert statements and are verified against an explicit bound on the number of function calls they make before safely halting. We hope this kind of logic will be applicable to real-time systems, where one is interested in concrete bounds rather than simple termination.

We are unaware of any other logic of resource bounds for languages containing either function pointers or the kind of contravariant circularity present in HAL.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
\[\chi(\tau) \equiv x := \ell\]
\[| x_3 := (x_1, x_2)\]
\[| x_2 := x_1.1\]
\[| x_2 := x_1.2\]
\[| \chi_1(\tau); \chi_2(\tau)\]
\[| \text{ifnil } x \text{ then } \chi_1(\tau) \text{ else } \chi_2(\tau)\]
\[| \text{call } x\]
\[| \text{return}\]
\[| \text{assert } P\]
\[\phi(\tau) \equiv \text{label } \rightarrow \chi(\tau)\]

load constant \(\ell\) into \(x\)
allocate a fresh pair
project first component
project second component
sequence two commands
test if \(x = 0\) and branch
call function pointer \(x\)
return from function
semantic assertion, wherein \(P : \tau\)
parametrized program

Fig. 1. Parameterized commands and programs

We can handle programs that exhibit nontrivial use of function pointers including mutually recursive function groups and higher-order functions. Each recursive group is verified as a whole and combined into proofs of whole-program termination, which makes the logic compositional. Higher-order functions are verified independently of the context in which they will be used and we are able to apply such functions to themselves without trouble (e.g., map of map). Our semantic model demonstrates how step indexing can be applied to logics of resource bounds in a compact manner.

Contributions. We design a language containing function pointers and semantic assertions, develop an associated logic of time resource bounds, and apply the logic to example programs. We develop a step-indexed model of the Hoare judgment and prove the logic sound. Our results are checked in Coq.

Associated material. Interested readers can find more details, particularly on the semantic models, in unreviewed previous versions of this work [7]. The Coq development is at \texttt{http://msl.cs.princeton.edu/termination/}.

2 An Introduction to HAL

We present HAL commands and programs in Figure 1. Our syntax is parameterized over the type of assertions \(\tau\) (i.e., the metatype of our parameterized syntax is \texttt{Type\rightarrow Type} instead of \texttt{Type}). Most commands are unexciting: load a constant, allocate a fresh pair, project from a pair, sequence, and branch-if-0. Subcommands for sequences and branches are parameterized over the same type variable \(\tau\). Our call instruction is noteworthy because \(x\) is a variable instead of a constant—i.e., \(x\) is a function pointer. Functions do not take explicit arguments; instead, a programmer must establish an ad-hoc calling convention. The unusual command is the semantic assertion assert; here \(P\) has the type of the argument \(\tau\). A parameterized program \(\phi(\tau)\) is a partial function from code labels to parameterized commands.

We give the basic semantic definitions for HAL in Figure 2a. We use natural numbers for program variables (for readability we use \(r_i\) instead of \(i\) for concrete
variable $x \equiv \mathbb{N}$
label $\ell \equiv \mathbb{N}$
value $v \equiv$
label + (value $\times$ value)
store $\rho \equiv$
variable $\mapsto$ value
measure $t \equiv$ store $\mapsto \mathbb{N}$
predicate $P \approx$
(program $\times$ store) $\rightarrow \mathcal{T}$
command $c \equiv \chi$(predicate)
stack $s \equiv$ list command
program $\Psi \equiv$
label $\mapsto$ command

| $\top$, $\bot$ | truth and falsehood |
| $P \land Q$, $P \lor Q$ | conjunction and disjunction |
| $P \Rightarrow Q$, $\neg P$ | implication and negation |
| $\forall a : \tau$, $\exists a : \tau$ | impredicative quantification |
| $\mu X. P$ | contravariant equirecursion |
| $x \downarrow v$ | variable $x$ evaluates to value $v$ |
| $[x \leftarrow v]P$ | $P$ will hold if $x$ is updated to $v$ |
| closed($P$) | $P$ holds on all stores |
| $P \vdash Q$ | entailment |
| $\langle|t|\rangle$ | measure $t$ on the current store |
| funptr $\ell [P][Q]$ | terminating function pointer |

(a) Basic semantic definitions
(b) A variety of predicates

Fig. 2. Basic semantic definitions and assertions in our separation logic

program variables in our examples). We also use natural numbers for code labels (addresses). We define values as trees having labels as leaves. A store (a.k.a. register bank) is a partial function from variables to values. A measure is a partial function from stores to natural numbers; we will require measures to decrease during function calls. A predicate is (essentially) a function from pairs of program and store to truth values $\mathcal{T}$ (Prop in Coq). A command is a specialization of a parameterized command $\chi$ with predicate; a stack is a list of commands. A program is a partial function from labels to commands—i.e., program $= \phi$(predicate). Notice that the metatypes predicate, command, and program contain a contravariant cycle. The real semantic definition for predicate, which is similar in flavor but with the pleasing addition of being sound, is the subject of §5.

We give a variety of predicates in Figure 2b. We have constants ($\top$, $\bot$) and the standard logical connectives ($\land$, $\lor$, $\Rightarrow$, $\neg$). Our quantification ($\forall$, $\exists$) is impredicative—that is, the metavariable $\tau$ ranges over all of the types in our metalogic ($\tau : \text{Type}$ in Coq), including predicate itself. We provide a contravariant-capable equirecursive $\mu$ to describe recursive program invariants as long as the recursion is contractive [9]. The assertion $x \downarrow v$ means that the variable $x$ evaluates to value $v$ in the current store. We write $[x \leftarrow v]P$ to mean that the predicate $P$ will be true if the current store is updated so that variable $x$ maps to value $v$; $[x \leftarrow v]$ is therefore a kind of modal operator—the modality of store update. We define another modal operator, closed($P$), meaning $P$ holds on all stores.

We write $P \vdash Q$ for predicate entailment. We also introduce a notational convenience for reasoning about measures in the context of a predicate. Since a predicate is more-or-less a function taking (among other things) a store $\rho$ as an argument, and since a measure $t$ is a partial function from stores to $\mathbb{N}$, it is simple to evaluate $t(\rho)$ and then compare the result against other naturals with the usual operators $=, <$, etc. To indicate this kind of evaluation and comparison, we will write e.g.,
“\( \langle t \rangle < n \)”—that is, evaluate \( t \) with the current store and require that the result be less than \( n \). When \( t(\rho) \) is not defined, terms containing \( \langle t \rangle \) are equivalent to \( \bot \).

The assertion of particular interest is the terminating function pointer assertion “\( \text{funptr} \ell t [P] [Q] \)”, wherein \( \ell \) is a function address, \( t \) is a termination measure, \( P \) is a precondition, and \( Q \) is a postcondition. The precondition \( P \) and postcondition \( Q \) are actually functions from some shared type \( A \) to \emph{predicate}, i.e., \( P = \lambda a : A. (\ldots) \) and \( Q = \lambda a : A. (\ldots) \). The type parameter \( A \) is actually part of the function pointer assertion but has been elided for the presentation. When \( \text{funptr} \ell t [P] [Q] \) holds:

(i) The program has code \( c \) at address \( \ell \) (recall \emph{programs} are functions from labels to \emph{commands}); this is why we need \emph{predicates} to take \emph{programs} as arguments.

(ii) When \( c \) is called from a context with an initial store \( \rho \), if \( t(\rho) \) is defined, then \( c \) makes at most \( t(\rho) \) function calls before returning to its caller.

(iii) If \( t(\rho) \) is defined, then for all \( a \), if \( P(a) \) holds prior to executing \( c \), then \( Q(a) \) will hold when \( c \) returns. The parameter \( a \) is thus able to relate pre- and postconditions to each other over the function call without auxiliary state.

### 3 Total Correctness for HAL

Our program logic is divided into two parts. Hoare rules verify commands in HAL; a strength of our approach is that these are natural. Function rules use the verification of a function’s body to prove that the function satisfies its specification.

**Hoare Rules.** Our Hoare judgment, written \( \Gamma, R \vdash_n \{ P \} c \{ Q \} \), where \( P, Q \), and \( R \) are \emph{predicates} (assertions), \( \Gamma \) is a \emph{closed predicate} that only looks at \emph{programs}, \( n \) is a natural number and \( c \) is a \emph{command}. We defer the formal model until §6, but the informal meaning is straightforward. \( P, Q \), and \( R \) are the standard precondition, instruction, and postcondition triple common in Hoare logics. The return assertion \( R \) is the postcondition of the current function; \( R \) must hold before the function can return. We collect \emph{funptr} assertions in \( \Gamma \). Finally, starting from precondition \( P \), \( n \) is an upper bound on the number of function calls \( c \) will execute before it terminates. Our logic is powerful enough that all of these parameters (including the time bound \( n \)) can take logical variables instead of concretes; indeed, the second example from §4 demonstrates that we can verify higher-order polymorphic functions independently of their call sites.

We present the Hoare rules for total correctness in Figure 3. The four rules Hlabel, Hcons, Hf Fetch1, and Hf Fetch2 are the standard weakest precondition forms for local variable updates for constants, fresh pairs, and first/second projections respectively. For brevity we only show the Hlabel rule; see [7] for Hcons, Hf Fetch1, and Hf Fetch2. Since these rules do not make any function calls \( n = 0 \).

The sequence rule Hseq looks standard; the key point is that the upper bounds on the subcommands \( c_1 \) and \( c_2 \) are summed for the sequence. For the conditional rule Hif, both \( c_1 \) and \( c_2 \) must share the same bound \( n \), which is then used for the whole. If the natural bounds differ, one harmonizes them via weakening.

The weakening/consequence rule Hweaken allows covariance in the preconditions.
\[
\begin{align*}
&\Gamma, R \vdash_0 \{[x \leftarrow \ell]\} x := \ell \{Q\} \quad \text{Hlabel} \\
&\Gamma, R \vdash_n \{P\} c_1 \{Q\} \quad \Gamma, R \vdash_n' \{Q\} \quad \text{Hseq} \\
&\quad \Gamma, R \vdash_{n+n'} \{P\} \quad \Gamma, R \vdash_n \{P_1\} \quad \Gamma, R \vdash_n \{P_2\} \quad \text{Hnif} \\
&\quad \Gamma, R \vdash_n \{(x \downarrow 0 \land P_1) \lor ((\neg (x \downarrow 0)) \land P_2)\} \quad \text{ifnil x then c_1 else c_2 \{Q\}} \\
&\quad n \leq n' \quad \Gamma' \land R \vdash R' \quad \Gamma' \land P' \vdash P \quad \text{Hweaken} \\
&\quad \Gamma' \vdash \Gamma' \land Q \vdash Q' \quad \Gamma, R \vdash_n \{P\} \quad \text{Hassert} \\
&\quad \Gamma', R' \vdash_{n'} \{P'\} \quad \Gamma, R \vdash_0 \{R\} \quad \text{Hreturn} \\
&\quad \Gamma \land P \vdash Q \quad \text{Hcall} \\
&\quad \{P\} \quad \text{assert Q \{P\}} \\
&\quad \{P\} \quad \text{call x \{Q\}} \\
&\quad P \equiv \ x \downarrow \ell \quad \land \quad \text{funptr} \ell \ t \ [P_\ell] [Q_\ell] \quad \land \quad \ll t \rr = n \\
&\quad \exists t(a) \quad \land \quad \text{closed}(Q_\ell(a) \Rightarrow Q) \\
&\quad \Gamma, R \vdash_{n+1} \{P\} \quad \text{Hcall} \\
\end{align*}
\]

Fig. 3. Hoare rules

\((P, P')\) and contravariance in the postconditions \((Q, Q')\) and return conditions \((R, R')\). The function assertions \((\Gamma, \Gamma')\) are related covariantly and incorporated in the other entailments in the most general way. We allow the bound on the number of function calls \((n, n')\) to increase during weakening since the bound is not strict.

Although semantic assertions caused significant headaches in the semantic model due to the contravariance outlined in §2, the Hassert rule is pleasingly direct. We simply ensure that the precondition \(P\) (including the function assertions in \(\Gamma\)) entails \(Q\). We use \(n = 0\) since the assert command does not make any function calls. The Hreturn rule requires that the precondition match the return assertion. After a function returns the remainder of the function is not executed, so we provide the postcondition \(\bot\). Since return does not make any function calls \(n = 0\).

The most important rule is Hcall, for verifying a function pointer call. The precondition \(P\) has five conjuncts. First, the variable \(x\) must point to a code label \(\ell\). Second, \(\ell\) must be a function pointer to some code with termination measure \(t\), function precondition \(P_\ell\), and function postcondition \(Q_\ell\). Third, the termination measure \(t\) must be defined on the current store and evaluate to some \(n\). That is, starting from the current store, the function \(\ell\) will make no more than \(n\) function calls before returning. Fourth, the function precondition \(P_\ell\) must hold when applied to some \(a\). Finally, the function postcondition \(Q_\ell\), when applied to the same \(a\), must imply the postcondition \(Q\) in all stores \((i.e., \text{in particular, in the store after the function call is completed})\). The metavariable \(a\) is chosen to relate the function
Fig. 4. Single and mutually recursive function verification

pre- and postconditions to each other over the call. Consider the pair:

\( P_\ell \equiv \lambda(x,v). \; (r_0 \downarrow 4) \land ((x \neq r_0) \Rightarrow x \downarrow v) \)
\( Q_\ell \equiv \lambda(x,v). \; (r_0 \downarrow 8) \land ((x \neq r_0) \Rightarrow x \downarrow v) \)

If we need to know that the invariant \( r_{15} \downarrow (16,(23,42)) \) is preserved over the call then we set \( a = (r_{15},(16,(23,42))) \). The key point of the HCall rule is that if we satisfy \( P \) then we can verify a function pointer call with a bound of \( n + 1 \) calls.

**Precondition generator.** Our update rules are in weakest-precondition style and our predicates include general quantification. Our Coq development defines a precondition generator that computes \( P \) from \( R, n, c, \) and \( Q \), which we use to cut down on the tedium of mechanically verifying the example programs from §4.

**Function Verification.** The whole-function rules in Figure 4 form the heart of our program logic. Although the symbol count is daunting, the core idea is natural.

Functions are normally verified one at a time, although mutually recursive function groups are verified as a set. One begins with Vstart, which says that program \( \Psi \) has specification \( \top \) (i.e., no functions in \( \Psi \) have been verified to terminate). The Vsimple and Vfull rules verify the addition of terminating function specifications into the context \( \Gamma \). Vsimple is sufficient to handle simple recursive functions that take non-polymorphic function pointers as arguments. Vfull handles mutually recursive function groups and polymorphic function pointers; Vsimple is just a special case of Vfull. After verifying the first function/group, one continues with another Vsimple/Vfull until all of \( \Psi \) has been verified.

The Vsimple rule assumes that \( \Psi \) already has specification \( \Gamma \); we wish to add the
Dockins, Hobor

specification for the function at \( \ell \) using termination measure \( t \), precondition \( P \), and postcondition \( Q \). The key premise is the second: we must verify, using the \( \Pi \)-rules, that for any \( n \) and \( a \), the function body \( \Psi(\ell) \) meets the specification

\[
\ldots, \quad Q(a) \vdash_n \{ P(a) \land \langle t \rangle = n \} \quad \Psi(\ell) \{ \bot \}
\]

That is, starting from a state that satisfies \( P(a) \) and in which the termination measure \( t \) evaluates to \( n \), the function will \textbf{return} in a state satisfying \( Q(a) \) after having made no more than \( n \) function calls. We use \( \bot \) as the postcondition since the function is not allowed to “fall off the bottom”. The key to doing recursive functions is how we set up the function specifications: we verify \( \Psi(\ell) \) using the previously-verified function specifications in \( \Gamma \) as well as a modified specification for \( \ell \) itself:

\[
\text{funptr} \quad \ell \quad t \quad [\lambda a'. \quad P(a') \land \langle t \rangle < n] \quad [Q]
\]

That is, the function body \( \Psi(\ell) \) can call to other functions specified in \( \Gamma \) as well as recursive calls to itself as long as the termination measure decreases.

The \( \text{Vfull} \) rule generalizes the \( \text{Vsimple} \) rule in two orthogonal ways. First, \( \text{Vfull} \) can verify a mutually recursive set of functions. Second, \( \text{Vfull} \) can verify function specifications where the specifications take parameters. The universally-quantified variable \( b \) in the \( \text{Vfull} \) rule represents the specification parameters; \( b \) ranges over an arbitrary type chosen by the verifier. The variable \( \Phi \) appearing in the \( \text{Vfull} \) rule represents a finite set of function specifications, i.e., a set of tuples with a label, a termination measure and a pre- and postcondition. The specifications in \( \Phi \) represent the set of mutually recursive functions we are going to verify. The quantification over \( \Phi(b) \) in the premise of the rule means that we will have to construct a Hoare derivation for each function body represented in \( \Phi \). Correspondingly, the quantification in the conclusion means that subsequent verifications may rely on each of the specifications in \( \Phi \). In other words, the \( \text{Vfull} \) rule establishes the specifications of a set of mutually recursive functions simultaneously.
Note that $\Phi$ takes an argument; thus the function specifications can depend on the parameter $b$. In the premise of the Vfull rule, the value $b$ is bound once and the same $b$ is used to construct both the recursive assumptions and the verification obligations. In other words, the value of the parameter, $b$, is a constant throughout the recursion. Contrast this with the value $a$ which connects pre- and postconditions, which is allowed to vary at each recursive call. An interesting case occurs when $b$ is allowed to range over function specifications. In this case, the specifications in $\Phi$ take on a higher-order flavor. We shall use this power in the following section.

4 Examples of Verified Programming in HAL

Our logic has three distinctive features: time bounds on recursive function pointers, time bounds on polymorphic function pointers, and semantic assertions. Here we cover two examples, the first demonstrating recursive function pointers, and the second demonstrating polymorphic function pointers and the use of a semantic assertion whose truth cannot be checked at run-time. In our Coq development we have examples that combine both simultaneously; also see [7].

Example 4.1 [Unary addition] Here we examine a simple recursive function which “adds” two lists representing natural numbers in unary notation (lists terminated by the 0 label). Figure 5a defines the listnat predicate that relates natural numbers to their unary encoding. The code itself is given in Figure 5b. The idea is that starting from two unary-encoded naturals in registers $r_1$ and $r_2$, we strip cons cells from $r_1$ and add them to $r_2$ until there are no cells left in $r_1$, and then return. Line 1 simply asserts the precondition of the function. Line 2 tests if the value in register $r_1$ is nil; if so, we return. Otherwise, we perform one unit of work, which involves shifting one cons cell from $r_1$ to $r_2$. Note lines 7 and 8, where we load the constant label 1 into $r_0$ and jump to it; this sequence is typical of “static” function calls. Since the code itself is loaded at label 1, this is a recursive call.

We give the specification in figure 5c. Note that the pre- and postconditions of the addition function are parameterized by the pair of numbers to be added. Recall that we allow termination measures to be partial functions; we use that power here because addt is only defined when the value in $r_1$ encodes some natural number.

The addition function is a simple self-recursive function, so we can verify it using the Vsimple rule. The proof obligation that is generated by Vsimple (after some minor simplifications) is shown in Figure 5d, and it is straightforward to use the H-rules of our logic to fulfill this verification obligation proof.

Example 4.2 [apply] While the code for the “apply” function is dead simple, the specification is rather subtle. The “apply” function makes essential use of function pointers and thus has a higher-order specification. The basic idea is that one packages a function label with some arguments using a cons cell in $r_0$. Apply

---

1 Even the type of $a$ which connects the pre- and postconditions can depend on the value of $b$.  

139
Dockins, Hobor

(a) Encoding naturals

<table>
<thead>
<tr>
<th>Number</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>2</td>
<td>(0, (0, 0))</td>
</tr>
<tr>
<td>3</td>
<td>(0, (0, (0, 0)))</td>
</tr>
</tbody>
</table>

(b) Code \(c_{\text{add}}\), loaded at label 1

\[
\text{addP}(n, m) \equiv \exists v_1 v_2, r_1 \downarrow v_1 \land r_2 \downarrow v_2 \land \text{listnat}(n, v_1) \land \text{listnat}(m, v_2)
\]

\[
\text{addQ}(n, m) \equiv \exists v_2. r_2 \downarrow v_2 \land \text{listnat}(n + m, v_2)
\]

\[
\text{addt}(\rho) \equiv \text{the unique } n \text{ s.t. } \exists v_1. \rho(r_1) = v_1 \land \text{listnat}(n, v_1)
\]

(c) Pre- and postcondition; termination measure

\[
\forall n_1, m_1, n.
\]

\[
(\text{funptr 1 addt } [\lambda n_2 m_2. \text{addP}(n_2, m_2) \land \llbracket\text{addt}\rrbracket < n] \left[\text{addQ}\right],
\]

\[
\text{addQ}(n_1, m_1) \vdash_n \{\text{addP}(n_1, m_1) \land \llbracket\text{addt}\rrbracket = n\} \langle\text{code from fig. 5b}\rangle \{\bot\}
\]

(d) Verification obligation for unary addition (using Vsimple)

Fig. 5. Example 1: unary addition.

unpacks the cons cell and calls the contained function using the enclosed arguments. We toss in an interesting higher-order \texttt{assert} just before the call for fun.

In order to give a reasonable specification for this function and other higher-order operations, we identify a calling convention. We call functions that adhere to our calling convention “standard”. Register \(r_0\) is used for passing function arguments and results. Registers \(r_1\)–\(r_4\) are callee-saves registers (whose values must be preserved over the call) and all other registers are caller-saves. In addition, we require the precondition, postcondition, and termination measure, for standard functions, to be defined only on the argument/return value (the value in \(r_0\)). We say a function satisfies \texttt{stdfun}(\(t, P, Q\)) (where \(t, P\) and \(Q\) are defined over a single value rather than an entire store) if \(\ell\) is a standard function in the sense just defined.
The time bound any static assumptions about the functions it will be passed.

\[ csregs(v_1, v_2, v_3, v_4) \equiv (r_1 \downarrow v_1) \land (r_2 \downarrow v_2) \land (r_3 \downarrow v_3) \land (r_4 \downarrow v_4) \]

\[ \text{stdfun}(\ell, t_\ell, P_\ell, Q_\ell) \equiv \text{funptr } \ell (\lambda r. t_\ell(r_0)) \]

\[
\begin{align*}
[\lambda (v_1, v_2, v_3, v_4). (r_0 \downarrow v_0) \land P_\ell(a)(v_0) \land csregs(v_1, v_2, v_3, v_4)] \\
[\lambda (v_1, v_2, v_3, v_4). (r_0 \downarrow v_0) \land Q_\ell(a)(v_0) \land csregs(v_1, v_2, v_3, v_4)]
\end{align*}
\]

\[ \text{applyP}(t, P, Q)(a)(v) \equiv \exists v_2. v = (\ell, v_2) \land \text{stdfun}(\ell, t, P, Q) \land P(a)(v_2) \]

\[ \text{applyQ}(Q)(a)(v) \equiv Q(a)(v) \]

\[ \text{applyt}(t)(v) \equiv t(v) + 1 \]

(a) “standard” functions; precondition, postcondition, termination measure for apply

\[
R \equiv \exists v'_0. r_0 \downarrow v'_0 \land Q(a)(v'_0) \land csregs(v_1, v_2, v_3, v_4)
\]

\[ \vdash 0 \{ r_0 \downarrow (\ell, v) \land P(a)(v) \land t(v) + 1 = n \land \text{stdfun}(\ell, t, P, Q) \land csregs(v_1, v_2, v_3, v_4) \} \]

\[ r_5 := r_0.0 ; \]

\[ \vdash 0 \{ r_0 \downarrow (\ell, v) \land r_5 \downarrow \ell \land P(a)(v) \land t(v) + 1 = n \land \text{stdfun}(\ell, t, P, Q) \} \]

\[ r_0 := r_0.1 ; \]

\[ \vdash 0 \{ (r_0 \downarrow v) \land (r_5 \downarrow \ell) \land P(a)(v) \land (t(v) + 1 = n) \land \text{stdfun}(\ell, t, P, Q) \} \]

\[ \text{assert} (\exists \ell', t', P', Q'. (r_5 \downarrow \ell') \land \text{stdfun}(\ell', t', P', Q')) ; \]

\[ \vdash 0 \{ (r_0 \downarrow v) \land (r_5 \downarrow \ell) \land P(a)(v) \land (t(v) + 1 = n) \land \text{stdfun}(\ell, t, P, Q) \} \]

\[ \text{call } r_5 ; \]

\[ \vdash n \{ r_0 \downarrow v' \land Q(a)(v') \} \]

\[ \text{return} ; \]

\[ \vdash n \{ \bot \} \]

(b) code capp for apply with abbreviated Hoare triples, loaded at label 1

\[
\begin{align*}
\forall (t, P, Q) (v_1, v_2, v_3, v_4, a) n. & \quad \exists t_\ell. R \vdash n \{ P_{\text{apply}} \} \langle \text{code from fig. 6b} \rangle \{ \bot \} \\
P_{\text{apply}} \equiv & \quad \exists \ell, t_\ell. r_0 \downarrow (\ell, v_0) \land \text{stdfun}(\ell, t, P, Q) \land P(a)(v_0) \\
& \quad \land csregs(v_1, v_2, v_3, v_4) \land t(v_0) + 1 = n
\end{align*}
\]

(c) Verification obligation for apply (using Vfull, after simplification)

Fig. 6. Example 2: apply

In the specification for apply (Figure 6) \( t, P \) and \( Q \) are the parameters of the specification; they describe the function that will be called. We need the Vfull rule to verify the apply function, with \( b \) ranging over tuples \( (t, P, Q) \). This way we can specify and prove correct the apply function in complete isolation, without requiring any static assumptions about the functions it will be passed. The time bound \( n \)
in the verification comes from the bound on the input function. In some later verification, we can instantiate the specification with any function specification already verified. In particular, apply can be applied to itself! This would not be a recursive call, in the traditional sense, but rather a dynamic higher-order call.

Termination remains assured due to the way the specifications get “stacked” on top of each other. This stacking of function specifications creates a tree-like structure wherein the leaves must be first-order functions (whose specifications do not depend on the specifications of other functions). The whole thing hangs together because there is no way to create a cycle in the tree of function specifications, and thus no way to introduce new, potentially nonterminating, recursion patterns. See the formal development for an example of such “stacked” function applications.

The development also contains an implementation of and verification for the recursive higher-order function map, whose termination argument is the sum of the input function’s termination argument applied to each element of the list plus the length of the list (for the recursive calls of map itself); for more details see [7].

5 Resolving the Circularity in predicates

In this section we resolve the circularity in predicate from Figure 2a. In §6 we build a model for the program logic itself and prove that programs verified in our logic terminate within the correct time bound.

Using Indirection Theory to Stratify Through Syntax. The pseudomodel of predicates in Figure 2a fits into the pattern \( K \approx F((K \times O) \rightarrow T) \). In this pattern, \( F \) is a covariant functor, \( O \) is some kind of “flat data”, and \( K \) is an object one wishes to model. A cardinality argument shows that there are no solutions in set theory, so we instead build an approximate model using indirection theory [10]. In our case, \( F \) is the parameterized program \( \phi \) from Figure 1 and \( O \) is just store. Indirection theory “ties the knot” and defines \( K \) such that:

\[
\begin{align*}
\text{sq} \_ \text{program} & \quad \Psi \quad \equiv \quad K \equiv \ [2, \text{knot}_{\text{hered}}.v] \\
\text{state} & \quad \sigma \quad \equiv \quad \text{sq} \_ \text{program} \times \text{store} \\
\text{predicate} & \quad P \quad \equiv \quad \{ P : \text{state} \rightarrow T \mid \text{hereditary}(P) \}
\end{align*}
\]

The construction of the knot \( K \) is similar to the one given in [10, §8] but we have enhanced it so that all predicates inside the knot are hereditary, a technical property detailed later. A squashed program \( \text{sq} \_ \text{program} \) is simply a knot; a state is a pair of a \( \text{sq} \_ \text{program} \) and a store. A predicate is a hereditary function from states to truth values \( T \). We write \( \sigma \models P \) instead of \( P(\sigma) \) when we wish to emphasize that we are thinking of \( P \) as an assertion as opposed to a function. The squashed and unsquashed programs are related by two functions \( \text{squash} : (\mathbb{N} \times \text{program}) \rightarrow \text{sq} \_ \text{program} \) and \( \text{unsquash} : \text{sq} \_ \text{program} \rightarrow (\mathbb{N} \times \text{program}) \). The power of indirection theory is that two simple axioms relate \( \text{squash} \) and \( \text{unsquash} \):
Dockins, Hobor

\[
squash(\text{unsquash}(\Psi)) = \Psi \\
\text{unsquash}(\text{squash}(n, \Psi)) = (n, \text{prog}_n(\Psi))
\]

That is, \( \text{squash} \circ \text{unsquash} \) is the identity function, and \( \text{unsquash} \circ \text{squash} \) is a kind of approximation function. The \( \text{prog}_n(\Psi) \) function transforms \( \Psi \) by locating all of the \( \text{assert}(P) \) statements and replacing them with \( \text{assert}(\text{approx}_n(P)) \). The core of the approximation is handled by the \( \text{approx}_n(P) \) function:

\[
|\tilde{\Psi}| \equiv (\text{unsquash}(\tilde{\Psi})).1
\]

\[
\text{approx}_n(P) \equiv \lambda(\tilde{\Psi}, \rho). \begin{cases} P(\tilde{\Psi}, \rho) & |\tilde{\Psi}| < n \\
\bot & |\tilde{\Psi}| \geq n \end{cases}
\]

First we define the level of a squashed program \( \tilde{\Psi} \), written \(|\tilde{\Psi}|\), as the first projection of \( \tilde{\Psi} \)'s unsquashing. When a predicate is approximated to level \( n \)—i.e., \( P = \text{approx}_n(P) \). What happens if we apply \( P \) to a state containing \( \tilde{\Psi} \) itself? A review of (2) proves that the result must be \( \bot \). A predicate cannot say anything meaningful about the squashed program whence it came. Instead, we will do the next best thing: make \( \tilde{\Psi} \) a little simpler. We say that \( \tilde{\Psi} \) (or \( \sigma \)) is approximated to \( \tilde{\Psi}' \) (or \( \sigma' \)), written \( \tilde{\Psi} \sim \tilde{\Psi}' \), when:

\[
\tilde{\Psi} \sim \tilde{\Psi}' \equiv \text{let } (n, \Psi) = \text{unsquash}(\tilde{\Psi}) \text{ in } ((n > 1) \land (\tilde{\Psi}' = \text{squash}(n - 1, \Psi)))
\]

\[
(\tilde{\Psi}, \rho) \sim (\tilde{\Psi}', \rho') \equiv (\rho = \rho') \land (\tilde{\Psi} \sim \tilde{\Psi}')
\]

That is, we unsquash \( \tilde{\Psi} \) and then re-squash it to one level lower. Of course, we can only do this when we are not at level 0 to begin with! Since \(|\tilde{\Psi}'| = n - 1 < n \), \( P \) will be able to judge states containing \( \tilde{\Psi}' \). Every time we pull a predicate out of a squashed program \( \tilde{\Psi} \), we will approximate \( \tilde{\Psi} \) to \( \tilde{\Psi}' \) before we use \( P \).

Repeated approximation leads to a second microcost. Suppose \( \tilde{\Psi} \models P \) and \( \tilde{\Psi} \sim \tilde{\Psi}' \). We say \( P \) is hereditary—stable (or monotonic) as \( \tilde{\Psi} \) is approximated—so that \( \tilde{\Psi}' \models P \); that is, hereditary(\( P \)) \equiv \forall \tilde{\Psi}. (\tilde{\Psi} \models P) \rightarrow (\tilde{\Psi} \sim^* \tilde{\Psi}') \rightarrow (\tilde{\Psi}' \models P) \),

where we write \( \sim^* \) and \( \sim^+ \) to mean the reflexive and irreflexive transitive closures, respectively, of \( \sim \). Unfortunately, not all functions from state to \( \mathcal{T} \) are hereditary,
such as $P_{\text{bad}}(\Psi, \rho) \equiv |\Psi| > 5$. The $P_{\text{bad}}$ function will be true only while the level of the program is greater than 5; due to approximation, this function will eventually produce only the constant $\bot$. We only consider predicates that are hereditary, and every predicate defined in this paper (except for $P_{\text{bad}}$) has been proved so.

A central question is how these kinds of microcosts become macrocosts—that is, what are the fundamental limitations of step indexing techniques? For some time, it was thought that step-indexed models could not produce the kinds of existential witnesses needed for termination proofs; however, the present work proves otherwise. The practical limitations of step-indexed models remain unknown.

**Models for predicates.** We define the logical connectives for predicates ($\top, \bot, \land, \lor, \Rightarrow, \neg, \forall, \exists, \mu$), and entailment ($P \vdash Q$) by a standard intuitionistic lift over the $\leadsto^*$ relation as in [9]. We define the modality of approximation ($\triangleright P$), used in the metatheory but not by the end-user, as the boxy operator over $\leadsto^+$:

$$\begin{align*}
(\Psi, \rho) \models \triangleright P & \iff \forall \Psi'. (\Psi \leadsto^+ \Psi') \rightarrow \Psi' \models P
\end{align*}$$

The model for the terminating function pointer assertion $\text{funptr}$ is complex and is developed in §6; the other domain-specific predicates listed in Figure 2b are simply:

$$\begin{align*}
(\Psi, \rho) & \models x \downarrow v \quad \equiv \quad \rho(x) = v \\
(\Psi, \rho) & \models [x \leftarrow v]P \quad \equiv \quad (\Psi, [x \leftarrow v]\rho) \models P \\
(\Psi, \rho) & \models \text{closed } P \quad \equiv \quad \forall \rho'. (\Psi, \rho') \models P \\
(\Psi, \rho) & \models \langle t \rangle < n \quad \equiv \quad t(\rho) < n \quad \text{(etc. e.g., for } \langle t \rangle = n)\end{align*}$$

For further discussion on the models for predicates see [7].

## 6 A Step-indexed Model for Total Correctness

Soundness for our logic means that when a function in a verified program is run in a state satisfying its precondition, it will halt in a state satisfying its postcondition. Our soundness proof follows Appel and Blazy: build a semantic model for assertions; define the meaning of judgments; prove the inference rules of the logic as lemmas; and show that the judgment semantics implies the desired theorem.

**Operational semantics.** Most of the operational semantics of our language, involving simple data or control-flow, is straightforward. Here, we only highlight the more interesting portions of the operational semantics; for details see [7].

We define a small-step relation $(\Psi, \rho, s) \rightarrow (\Psi', \rho', s')$, in which $\rho$ and $s$ stand for local variables and stacks and $\Psi$ is the squashed program; it is modified as the
configHalts \_n \Psi, \rho, s) \equiv \exists \Psi', \rho'. (|\Psi| - |\Psi'| \leq n) \land (\Psi, \rho, s) \mapsto^* (\Psi', \rho', \text{nil})

(\Psi, \rho) \models \text{halts}_n s \equiv |\Psi| \geq n \rightarrow \text{configHalts}_n(\Psi, \rho, s)

guards\_n P s \equiv P \Rightarrow \text{halts}_n s

\Psi \models \text{funptr} \ell \text{ t} [P] [Q] \equiv \exists c. \text{let } (n, \Psi) = \text{unsquash}(\Psi) \text{ in } \Psi(\ell) = c \land 
\forall s, \Psi', n', a. (\Psi \sim^+ \Psi') \rightarrow 
(\forall \rho. (\Psi', \rho) \models \text{guards}_n Q(a) s) \rightarrow 
(\forall \rho n. t(\rho) = n \rightarrow (\Psi', \rho) \models \text{guards}_{n+n'} P ((c; \text{assert } \bot) :: s))

\Gamma, R \vdash_n \{P\} c \{Q\} \equiv \forall \Psi, n', k, s. \Psi \models \Gamma \rightarrow 
(\forall \rho. (\Psi, \rho) \models \text{guards}_n R s) \rightarrow 
(\forall \rho. (\Psi, \rho) \models \text{guards}_n Q (k :: s)) \rightarrow 
(\forall \rho. (\Psi, \rho) \models \text{guards}_{n+n'} P ((c; k) :: s))

\Psi' \models \text{approxedof}(\Psi) \equiv \Psi \sim^* \Psi'

\Psi : \Gamma \equiv \forall n. (\text{approxedof}(\text{squash}(n, \Psi)) \land \Gamma \models \Gamma)

Fig. 8. Terminating function pointers; Hoare tuples; whole-program verification

program runs in a very controlled way. Here is the rule for loading a label:

$$
(\Psi, \rho, (x := \ell; c) :: s) \mapsto (\Psi', [x \leftarrow \ell | \rho, c :: s]) \text{ Slabell}
$$

This rule, like most of the instructions in HAL, passes the program \( \Psi \) through without any change. Figure 7 lists the two rules of particular interest. The first is the Sassert rule, which shows how semantic assertions are checked as the program runs. The second is Scall, which shows what happens at function calls. This is the only rule wherein the program is modified: each assertion in the program text is approximated one level down. We must do this approximation so that assertions in the text of the function body will be able to judge the program. If we did not approximate the program at this point, any assertions in the function body would fail, foiling our desired soundness result. Thus, the level of the program \( \Psi \) is an upper bound on the number of calls the program can make before getting stuck.

Judgment Definitions. Appel and Blazy build their semantic Hoare triple using the more basic notion of guarding. They say that a predicate “guards” a program stack if, whenever a memory state satisfies the predicate, that stack is safe to run (i.e., will not go wrong). We follow a similar pattern, but use a guards predicate which enforces termination rather than safety. We say that a predicate \( P \) guards a stack \( s \) at level \( n \) if, whenever the memory state satisfies \( P \) and provided that the program level is at least \( n \), running the stack will eventually terminate (cf.
Figure 8). Notice that there is a clever trick being played here with the definition of $\text{halts}_n$. Halting is not normally a predicate which can be hereditary. As one ages a program, it is able to run for fewer steps and thus might not terminate before it exhausts its level. We work around this issue by saying that a program must only terminate if it has at least level $n$. As one ages a program, it will eventually cause $\text{halts}_n$ to be true vacuously (when its level falls below $n$).

We use guards in the terminating function pointer assertion. Here, $\ell$ is a program label, $t$ is a measure, and $P$ and $Q$ are functions from some type $A$ to assertions. This definition is in a continuation-oriented style. Whenever we have a stack $s$ which terminates in $n$ steps when $Q(x)$ is satisfied, then we know that running the function body of $\ell$ will terminate in $n + n'$ steps whenever $P(x)$ is satisfied, and where $n'$ is determined by the measure. Thus, $\text{funptr}$ captures the specification of a terminating function. Note the premise $\tilde{\Psi}\leadsto^+\tilde{\Psi}'$; this is one of the microcosts discussed in §5. $\tilde{\Psi}'$ must be strictly more approximate than $\tilde{\Psi}$ because stepping over a call instruction ages the program. By design, the $\text{funptr}$ predicate and the Hoare judgment are similar: assume the postcondition(s) guard the program continuation point(s) and demonstrate that the precondition guards the extended continuation.

The final definition is program verification $\Psi : \Gamma$. That is, we can prove $\Gamma$ provided that we assume the program under consideration is some squashed version of $\Psi$ and $\triangleright\Gamma$ (i.e., approximately $\Gamma$). The assumption $\triangleright\Gamma$ plays the role of an induction hypothesis and is what allows us to verify recursive functions. The $\text{approx\:eof}(\tilde{\Psi})$ predicate means that the current program is approximated from $\tilde{\Psi}$.

H- and V-rules. Now that we have finished our semantic definitions, we are prepared to prove the rules of the Hoare logic as lemmas. The proofs are straightforward for all of the rules aside from $\text{Hcall}$, which itself is not arduous [7].

The real magic happens in the proof of the function verification rule, $\text{Vfull}$. $\text{Vfull}$ converts Hoare derivations for function bodies into the corresponding $\text{funptr}$ assertions on programs containing those function bodies. $\Phi$ is a list containing the precondition, postcondition and termination measure for a group of mutually recursive functions; for each function in $\Phi$, one proves a particular Hoare derivation. $\Gamma$ contains the assumptions one is allowed to make and it includes functions already verified and those from $\Phi$, which allows recursive calls. However, the preconditions in $\Phi$ are altered to add a conjunct which strengthens the preconditions by requiring the termination measure to decrease. The return postcondition is the postcondition of the function. The precondition is the ordinary function precondition together with the assumption that the termination measure for the initial state is $n$; this is what connects the strengthened preconditions of the recursive assumptions with the initial state. The linear postcondition is $\bot$; this requires the function body to explicitly return. Finally, the Hoare derivation must bound the number of function calls by $n$; this connects the termination measures of the function specifications to their semantic meanings. By providing such a Hoare derivation for each function in $\Phi$, one can conclude that each function referenced in $\Phi$ respects its contract, and the corresponding $\text{funptr}$ facts can be conjoined with $\Gamma$ in the conclusion of the rule. The proof is by induction on the value of the termination measure; see [7].
**Total correctness.** The final soundness proof connects our definitions to a more traditional notion of total correctness. Suppose $\Psi : \Gamma$, and $\Gamma \vdash \text{funptr } \ell t \ [P] \ [Q]$. Then for all stores $\rho$ such that $t(\rho) = n$, and $(\text{squash}(n, \Psi), \rho)$ satisfies $P(a)$ (for some $a$), executing the function body $\Psi(\ell)$ will terminate in a state satisfying $Q(a)$.

Our core semantic ideas (§6) are compact, requiring only 1,315 lines of Coq.

7 Limitations and Related Work

**Limitations.** Our program logic is somehow simultaneously too weak and too strong. It is too weak in that the upper bound need not be tight, and we make no claims on the lower bound. Our logic is too strong in that the burden of constructing an explicit termination measure may be onerous for someone only concerned with termination. It would be better if one could provide a well-founded relation for each function, hiding the explicit bounds and termination measures under existentials.

We can relate the precondition to the upper bound so that we can verify, *e.g.*, that a program runs in polynomial bound (*e.g.*, examine the termination measure $\text{addt}$ from §4). However, the mechanism to do so is cumbersome; it might be better to allow the user to state termination arguments in the ordinals.

**Applications of step-indexing and its alternatives.** Step-indexing has been used to prove type safety [4], soundness of program logics [11], and program equivalence [3]. Indirection theory [10] provides clean axioms for step-indexed models. Domain theory is the classic tool for building semantic models. Birkedal *et al.* constructs indirection theory in ultrametric spaces [6].

**Predicates in syntax.** Semantic assertions are often used in program analysis settings such as BoogiePL [5]. Semantic assertions are one example of a larger class of bookkeeping instructions that embed formulas into program syntax, such as the $\text{makelock}$ instruction used in concurrent C minor [11].

**Program logics with function pointers.** Schwinghamer *et al.*’s recent work on “nested” Hoare triples [15] combines features of separation logic with the ability to reason about “stored code,” which is similar to function pointers. It is a logic of partial correctness. The work of Honda *et al.* seems nearest to our own in terms of logical power [13]. They provide a logic of total correctness for call-by-value PCF. The soundness proof goes by a reduction to the $\pi$-calculus equipped with a process logic in the rely/guarantee style [12]. Honda *et al.* do not consider embedded semantic assertions or explicit time bounds, but do consider the issue of completeness [8]. Aspinall *et al.* have developed a sound and complete program logic for Grail, a Java subset, which reasons about both correctness and resources [1]. Their system includes a form of virtual method invocation, but it is not clear if their formalism allows higher-order behaviors.

8 Conclusion

We have presented a simple language with embedded semantic assertions and function pointers, together with a logic of the total correctness of time resource bounds. Our logic is able to reason about terminating function pointers in a very general
way, including polymorphic mutually-recursive function groups. We have proved our logic sound with respect to an operational semantics using step-indexing, thus demonstrating that step-indexed models are useful for modeling resource logics.

Acknowledgement

We thank Andrew W. Appel for his helpful comments; and our support from NSF CNS-0910448, AFOSR FA9550-09-1-0138, and LKY PDF T1 251RES0902.

References


Abstract

We develop a cut-free (and hence analytic) nested sequent calculus for a modal logic of actions and propositions. The actions act on propositions via a dynamic modality (the *weakest precondition* of program logics), whose left adjoint we refer to as ‘update’ (the *strongest postcondition*). Both logics are positive and have adjoint pairs of epistemic modalities: the left adjoints express agents’ uncertainties and the right adjoints express their beliefs. The rules for ‘update’ encode learning as a result of discarding uncertainty. We prove admissibility of Cut, and hence the completeness of the logic w.r.t. an algebraic semantics. We interpret the logic on epistemic scenarios with honest and dishonest communication actions, add assumption rules to encode them and prove that the extended calculus still has the admissibility results. We apply the calculus to encode and solve the classic epistemic puzzle of *muddy children* and a modern version of it with dishonest agents.

Keywords: Proof Theory, Cut-Admissibility, Algebra, Adjoint Modalities, Epistemic Scenarios

1 Introduction

Temporal and epistemic logics can express properties of programs [10] and distributed system protocols [7]. The modalities of epistemic logics encode attitudes such as knowledge and belief, but dynamic changes to these attitudes, e.g. after announcements, have traditionally been formalised only in a semantic fashion. Dynamic program logics are, on the other hand, developed for the specific purpose of syntactic reasoning about changes of properties. Adding epistemic modalities...

---

1 Email: rd@st-andrews.ac.uk
2 M. Sadrzadeh acknowledges support by EPSRC grant EP/F042728/1.
3 Email: mehrs@cs.ox.ac.uk, truffaut.julien@gmail.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
to dynamic logics led to logics [3,4] allowing reasoning about belief updates, both syntactically and semantically. But, lacking cut-admissibility, the calculus proposed in [3] is not a basis for automatic proof search. Here we develop a cut-free sequent calculus as basis for a proof search procedure for one such logic. The calculus is nested, in the sense of calculi studied in, e.g., [5,8,9,11,12,16,15,18].

The logic from [3] comprises a linear logic of actions $Q$ and a logic of propositions $M$. The former act on the latter via modalities, similar to the dynamic modality of PDL, known there as the weakest precondition, and its right adjoint, the strongest postcondition. We develop an algebraic semantics for these logics in terms of a residuated lattice-monoid of actions and its right lattice of propositions; their completions are referred to as “quantale-module pairs” and their application to program semantics has been studied in [1]. We endow both $M$ and $Q$ with families of epistemic adjoint operators. We present sequent calculi for the logics, such that the logic of actions admits a cut rule and that the logic of propositions admits two cut rules (and weakening and contraction rules). The calculi are thus complete w.r.t. the algebraic semantics. Finally, after [3], we interpret the actions $Q$ as announcements (public or private, honest or dishonest), the operations of $Q$ on $M$ as the effects of the actions on propositions, the epistemic modalities in terms of agents’ uncertainties and beliefs, and distributivity properties involving the dynamic modalities in terms of properties of belief update procedures. We add rules to allow encoding of epistemic scenarios and show that the admissibility results are unaffected. We apply this to encode the classic epistemic scenario of muddy children and new versions of it with dishonest agents. The calculus (with some optimisations) and some example scenarios are implemented in [19].

The sequent calculus of actions and adjoint modalities enriches the calculus of [12] with the two classical operators of $\lor$ and $\land$. The rules for the logic of propositions include those of [17], together with new rules for the dynamic modalities, their distributivity interactions with epistemic modalities, and rules for action connectives in propositional contexts. The admissibility theorems are novel contributions. The algebraic semantics is a finite version of [3]; our formalisation allows for a completeness proof without lattice completions. Proving admissibility of the cut rules involves hundreds of cases., some already considered in [17]. Our solution of the muddy children puzzle is new; it does not rely on negation or the fix point operator of common knowledge, e.g. that in [10]. The sequent proof offers a unified method of solving such puzzles and is a major improvement on the inductive algebraic proof in [3].

Detailed proofs are available in a long version available from the authors.

2 Syntax, Algebra and Semantics for Actions

We give here the syntax and the intended algebraic semantics of our action logic; in section 3 we present a cut-free sequent calculus, and in section 6 a soundness and completeness theorem. Section 4 will add the syntax, an algebraic semantics and a sequent calculus for propositions, including those formed by the interaction
of actions and propositions in various ways.

2.1 Syntax and Algebra of Actions

The set $Q$ of actions $q$ of the action logic is generated over a set $A$ of agents $A$ and a set $B$ of basic actions $\sigma$ by the following grammar:

$$q ::= \bot \mid \top \mid 1 \mid \sigma \mid q \land q \mid q \lor q \mid q \cdot q \mid \Box_A q \mid \lozenge_A q$$

The binary connectives $\land$ and $\lor$ will be interpreted as lattice operations of meet and join, with $\top$ and $\bot$ as their units; $\cdot$ is a monoid multiplication with 1 as its unit, the modalities $\Box_A$ and $\lozenge_A$ will be adjoint endo-operators on the lattice monoid. We make this more precise as follows:

**Definition 2.1** Let $A$ be a set, with elements called agents. A lattice monoid with adjoint modalities (an LMAM) over $A$ is both a bounded lattice $(Q, \lor, \land, \top, \bot)$ and a unital monoid $(Q, 1, \cdot, \leq)$, where $\cdot$ preserves joins, with two $A$-indexed families $\{\lozenge_A\}_{A \in A} : Q \to Q$ and $\{\Box_A\}_{A \in A} : Q \to Q$ of order-preserving maps, each $\lozenge_A$ being left adjoint to $\Box_A$. Thus, in addition to the lattice axioms, the following hold, for all $q, q', q'' \in Q$ and all $A \in A$:

1. $q \cdot (q' \lor q'') = (q \cdot q') \lor (q \cdot q'')$ and $(q' \lor q'') \cdot q = (q' \cdot q) \lor (q'' \cdot q)$
2. $q \cdot 1 = q$ and $1 \cdot q = q$
3. $q \leq q'$ implies $\lozenge_A q \leq \lozenge_A q'$
4. $q \leq q'$ implies $\Box_A q \leq \Box_A q'$
5. $\lozenge_A q \leq q' \iff q \leq \Box_A q'$

**Proposition 2.2** In any LMAM $Q$ over $A$, the following hold:

<table>
<thead>
<tr>
<th>6</th>
<th>$\lozenge_A (q \lor q') = \lozenge_A q \lor \lozenge_A q'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$\lozenge_A (q \land q') \leq \lozenge_A q \land \lozenge_A q'$</td>
</tr>
<tr>
<td>10</td>
<td>$\lozenge_A \bot = \bot$</td>
</tr>
<tr>
<td>12</td>
<td>$q \cdot (q' \lor q'') \leq (q \cdot q') \land (q \cdot q'')$</td>
</tr>
<tr>
<td>14</td>
<td>$\lozenge_A \Box_A q \leq q$</td>
</tr>
</tbody>
</table>

**Definition 2.3** An LMAM $Q$ over $A$ is multiplicative iff we have the following:

1. $\lozenge_A (q \cdot q') \leq (\lozenge_A q) \cdot (\lozenge_A q')$ (16)
2. $\lozenge_A 1 \leq 1$ (17)

**Proposition 2.4** In any multiplicative LMAM $Q$ over $A$, the following hold:

1. $(\Box_A q) \cdot (\Box_A q') \leq \Box_A (q \cdot q')$ (18)
2. $1 \leq \Box_A 1$ (19)

The $qs$ are for interpretation of epistemic programs, such as announcements, that may change belief, and $\leq$ is the information order; hence $\lor$ is non-deterministic choice and $\bot$ is failure, $\cdot$ is sequential composition and 1 is skip (the null action),
\( \Diamond_A q \) is the “uncertainty” of agent \( A \) about action \( q \) and \( \Box_A q \) is the agent’s belief about it. As argued in [3], these epistemic operators allow encoding of various types of announcements such as public and private. Inequality (16) encodes a form of belief update, that the uncertainty about a composite action is derived from the composition of the uncertainties of the component actions.

### 2.2 Semantics of Actions

Let \( Q \) be a multiplicative LMAM over \( \mathcal{A} \). An interpretation in \( Q \) of the action logic (over \( \mathcal{A} \), and with set \( \mathcal{B} \) of basic actions) is a map \( [\cdot ] : \mathcal{B} \to Q \). The interpretations of compound actions are obtained by induction: 
\[
[q_1 \lor q_2] = [q_1] \lor [q_2], [q_1 \land q_2] = [q_1] \land [q_2], [q_1 \cdot q_2] = [q_1] \cdot [q_2], [\Diamond_A q] = \Diamond_A [q], [\Box_A q] = \Box_A [q], [\top] = \top, [\bot] = \bot
\]
and \([1] = 1\).

Note that we use \( Q \) both for the set of actions of the logic and for the LMAM interpreting them; the context will always make clear which is intended.

### 3 Sequent Calculus for Actions

#### 3.1 Preliminaries

We have action items \( T \) and action contexts \( \Theta \) generated by the following syntax:
\[
T ::= q \mid \Theta^A \quad \Theta ::= T \text{ list}
\]
where \( \Theta^A \) will be interpreted as \( \Diamond_A (\circ \Theta) \), for \( \circ \Theta \) the sequential composition of the interpretations of elements in the list \( \Theta \). The use of lists rather than sets or multi-sets reflects the non-commutativity (and non-idempotence) of the composition of actions.

Thus, action contexts (abbreviated to \( a\)-contexts) are finite lists of action items, where action items (abbreviated to \( a\)-items) are either actions or agent-annotated action contexts. The term context is not in the sense of “having a gap waiting to be filled” but as a short name for the antecedent of a sequent, i.e. the context in which the succedent action is performed (or, when we discuss propositions, asserted). If one of the \( a\)-items inside an \( a\)-context is replaced by a “hole” \([\cdot]\), we have an \( a\)-context with an \( a\)-hole. More precisely, we have the notions of an \( a\)-context-with-an-a-hole \( \Sigma \) and an \( a\)-item-with-an-a-hole \( R \), defined using mutual recursion as follows:
\[
\Sigma ::= \Theta, R, \Theta', R
\]
where \( R^A \) is defined recursively (together with the application of an \( a\)-item-with-an-a-hole \( R \) to an \( a\)-context, to form an \( a\)-context) as follows:
\[
\Sigma ::= \Theta, R, \Theta'
\]
Given an \( a\)-context-with-an-a-hole \( \Sigma \) and an \( a\)-context \( \Theta \), the result \( \Sigma[\Theta] \) of applying the first to the second, i.e. replacing the hole \([\cdot]\) in \( \Sigma \) by \( \Theta \), is an \( a\)-context, defined recursively (together with the application of an \( a\)-item-with-an-a-hole \( R \) to an \( a\)-context, to form an \( a\)-context) as follows:
\[
(\Theta', R', \Theta'')[\Theta] = \Theta', R'[\Theta], \Theta'' \\
([\cdot])[\Theta] = \Theta \\
(\Sigma^A)[\Theta] = (\Sigma[\Theta]^A)
\]
This notion of “context with a hole” allows, in the expression of the inference rules, a precise notation for replacement deep inside contexts. For example, with $\Sigma = c, d, (e, [])^1, f$ and $\Theta = a, b$, we obtain $\Sigma[\Theta] = c, d, (e, a, b)^1, f$.

Given a-contexts-with-an-a-hole $\Sigma'$, $\Sigma$, and an a-item-with-an-a-hole $R$, the combinations $\Sigma' \bullet \Sigma$ and $R \bullet \Sigma$ are defined to be a-contexts with a-holes, as follows, by mutual recursion on the structures of $\Sigma'$ and $R$:

$$(\Theta, R', \Theta') \bullet \Sigma = \Theta, (R' \bullet \Sigma), \Theta' \quad ([\ ] \bullet \Sigma = \Sigma \quad (\Sigma''A) \bullet \Sigma = (\Sigma'' \bullet \Sigma)^A$$

and satisfy $(\Sigma' \bullet \Sigma)[\Theta] = \Sigma'[\Sigma[\Theta]]$ and $(R \bullet \Sigma)[\Theta] = R[\Sigma[\Theta]]$.

### 3.2 Sequents and their Interpretations

[Action] sequents consist of an a-context $\Theta$ (on the left), a turnstile and an action $q$ (on the right). We omit the list constructors, e.g. we write $1, \sigma, \bot \vdash \sigma'$ rather than $\langle 1, \sigma, \bot \rangle \vdash \sigma'$; the empty list $\langle \rangle$ is often omitted.

The meanings of a-items and of a-contexts, given an interpretation, are obtained by mutual induction on their structure: $[[\Theta^A]] = \bullet_A[[\Theta]]$, $[[I_1, \cdots, I_n]] = [[I_1]] \bullet \cdots \bullet [[I_n]]$ and $[[\langle \rangle]] = 1$

A sequent $\Theta \vdash q$ is true in an interpretation $[[\cdot]]$ in $Q$ iff $[[\Theta]] \leq [[q]]$; it is true in $Q$ iff true in all interpretations in $Q$; it is valid iff true in every multiplicative LMAM. When checking validity of the rules, we use the easy fact that if $\Theta \leq \Theta'$ then $\Sigma[\Theta] \leq \Sigma[\Theta']$.

### 3.3 Sequent Calculus

We have the following initial sequents (in which $\sigma$ is restricted to being a basic action, i.e. is from $B$):

$$\frac{}{1 \vdash \top} R \quad \frac{\sigma \vdash \sigma}{\vdash \sigma \cdot \sigma} Id \quad \frac{\Sigma[\bot] \vdash q \bot}{\vdash \bot} L \quad \frac{\Theta \vdash \bot}{\vdash \bot} R$$

The rules for the lattice operations, composition and the modalities are:
We also have structural rules, i.e. rules not involving the algebraic operators:

\[
\frac{\Sigma[\Theta^A, \Theta'^A] \vdash q}{\Sigma[(\Theta, \Theta')^A] \vdash q} \quad \text{Dist} \quad \frac{\Sigma[[\Theta^A]] \vdash q}{\Sigma[[\Theta^A]] \vdash q} \quad \text{Unit}
\]

The two indicated occurrences of $\sigma$ in the $\text{Id}$ rule are principal. Each right rule has its conclusion’s succedent as its principal action (but we will always just call it a principal formula); in addition, the $\diamond A$ rule has $\Theta^A$ as a principal item. Each left rule has a principal item: these are as usual.

The size of an action $q$ is just the (weighted) number of operator occurrences, counting each operator $\diamond A$ and $\Box A$ as having weight 2; the size of an a-item $\Theta^A$ is the size of $\Theta$ plus 1, and the size of an a-context $\Theta$ is the sum of the sizes of its items. The size of a sequent $\Theta \vdash q$ is just the sum of the sizes of $\Theta$ and $q$. Note that each premiss of a rule instance has lower size than the conclusion, except for the rule Dist, whose presence leads to non-termination of a naive implementation of the calculus.

**Lemma 3.1** For every action $q$, the sequent $q \vdash q$ is derivable.

Following this, in proofs and examples below, we shall allow the use of $\text{Id}$ in the form $q \vdash q$ even where the action $q$ is not a basic action.

**Lemma 3.2** The $1L$, $\lor L$, $\bullet L$, $\diamond A L$, $\land R$ and $\Box A R$ rules are invertible.

**Lemma 3.3** The rules $\top L^-$ and $\bot R^-$ are admissible:

\[
\frac{\Sigma[\top] \vdash q}{\Sigma[\top] \vdash q} \quad \frac{\Theta \vdash \bot}{\Sigma[\Theta] \vdash q} \quad \frac{\bot R^-}{\Sigma[\bot] \vdash q} \quad \frac{\top L^-}{\Sigma[\top] \vdash q}
\]

154
As an example of a derivation, we show that a sequence of $\Diamond_A$ operations preserves composition and conjunction in one direction:

$$
\frac{\Theta \vdash q}{\Sigma'[q] \vdash q'} \text{Cut}
$$

**Proof.** Strong induction on the rank of the cut, where the rank is given by the pair (size of cut formula $q$, sum of heights of derivations of premisses). Full details are in the appendix of the long version. \qed

## 4 Syntax and Algebra for Propositions

### 4.1 Syntax of propositions

Given sets $A$ of agents $A$ and $B$ of basic actions $\sigma$, we have as above an action logic with a set $Q$ of actions $q$. Now let $At$ be a set of (propositional) atoms $p$; the set $M$ of propositions $m$ is generated by the following grammar:

$$
m ::= \bot \mid \top \mid p \mid m \land m \mid m \lor m \mid \Box_A m \mid \Diamond_A m \mid m \cdot q \mid [q]m
$$

The last two binary connectives are mixed action-proposition connectives: the operator $[q]_\cdot$ is the *dynamic modality* operator, with $[q]m$ read as “after action $q$, proposition $m$ holds”; $\cdot q$ is its left adjoint, called *update*, with $m \cdot q$ read as “proposition $m$ is updated by action $q$”. The difference is like that in ordinary logic between $a \rightarrow b$ and $b \land a$. $\Diamond_A$ is the left adjoint of $\Box_A$: the former expresses the uncertainty of an agent about a proposition, and the latter expresses his belief; $\Box_A m$ is read as ‘$A$ believes that $m$’.
4.2 Algebraic semantics for propositions

**Definition 4.1** Let \( A \) be a set, with elements called agents. A **DLAM** over \( A \) is a bounded distributive lattice \((L, \wedge, \vee, \top, \bot)\) with two \( A \)-indexed families \( \{\bullet_A\}_{A \in A}: L \to L \) and \( \{\Box_A\}_{A \in A}: L \to L \) of order-preserving maps, with each \( \bullet_A \) left adjoint to \( \Box_A \), i.e. \( \bullet_A(l) \leq l' \) iff \( l \leq \Box_A(l') \).

For a detailed list of derived properties of a **DLAM**, see [17].

**Definition 4.2** A multiplicative **LMAM** \( Q \) acts on a **DLAM** \( M \) (over the same sets of agents) whenever we have two order-preserving maps \( \cdot \cdot \cdot : M \times Q \to M \) and \( [\_]: Q \times M \to M \), with each \( \cdot \cdot \cdot \cdot \cdot q \) left adjoint to \( [\_q] \). Thus, in addition to the lattice axioms, the following hold, for all \( q, q' \in Q \) and all \( m, m' \in M \):

\[
\begin{align*}
q &\leq q' \implies m \cdot q \leq m \cdot q' \\
m &\leq m' \implies m \cdot q \leq m' \cdot q \\
m \cdot q &\leq m' \iff m \leq [q]m' \\
m \cdot (q \cdot q') &= (m \cdot q) \cdot q' \\
m \cdot 1 &= m \\
\bullet_A(m \cdot q) &\leq \bullet_A(m) \cdot \bullet_A(q) 
\end{align*}
\]

Equalities (23) and (24) are axioms for a Quantale-Module [1]; inequality (25) encodes a form of learning: the uncertainty of an "updated" proposition \( m \cdot q \) is learnt from the update of the uncertainty of \( m \) by the uncertainty of \( q \).

**Proposition 4.3** Whenever a multiplicative **LMAM** \( Q \) acts on a **DLAM** \( M \), the following hold, for all \( q, q' \in Q \) and \( m, m' \in M \):

\[
\begin{array}{cccc}
26 & (m \vee m') \cdot q = (m \cdot q) \vee (m' \cdot q) & 27 & (m \wedge m') \cdot q \leq (m \cdot q) \wedge (m' \cdot q) \\
28 & [q] (m \wedge m') = [q] m \wedge [q] m' & 29 & [q] m \vee [q] m' \leq [q] (m \vee m') \\
30 & \bot \cdot q = \bot & 31 & [q] \top = \top \\
32 & ([q] m) \cdot q \leq m & 33 & m \leq [q] (m \cdot q) \\
34 & [q \cdot q'] m = [q] [q'] m & 35 & [1] m = m
\end{array}
\]

Let \( Q \) be a multiplicative **LMAM** acting on a **DLAM** \( M \) and \([\_]\) an interpretation of the set of actions of the action logic (over a set \( B \) of basic actions) in \( Q \), as defined and extended in subsection 2.2. An **interpretation** of the set of propositions of the propositional logic (over a set \( At \) of atoms) in \( M \) is given by a map \([\_]: At \to M\); extension of the two interpretations to an interpretation of compound propositions is obtained by structural induction:

\[
\begin{align*}
[m_1 \vee m_2] &= [m_1] \vee [m_2], & [m_1 \wedge m_2] &= [m_1] \wedge [m_2], \\
\bullet_A(m) &= \bullet_A([m]), & \Box_A(m) &= \Box_A([m]), \\
\top &= \top, & \bot &= \bot, \\
[m \cdot q] &= [m] \cdot [q], & \boxed{[q] m} &= \boxed{[q]} m.
\end{align*}
\]
5 Sequent Calculus for Propositions

5.1 Preliminaries

As in the action logic, we have propositional contexts $\Gamma$ and propositional items $I$ (abbreviated to p-contexts and p-items), generated by the following grammar:

$$\Gamma ::= I \text{ multiset} \quad I ::= m \mid \Gamma^A \mid \Gamma^\Theta.$$  

The union of two multi-sets is indicated by a comma, as in $\Gamma, \Gamma'$ or (treating an item $I$ as a one element multiset) as in $\Gamma, I$. A p-item $I$ can be either a proposition or an agent-annotated p-context, as in [17]; but it can also be a p-context $\Gamma$ annotated by an a-context $\Theta$.

To express the rules correctly, we need, as in Section 3, some notion of p-context (or item) with a hole. There are now two kinds of hole, one for propositions and one for actions, both represented by $[\ ]$; we use the notations $\Delta$ for a p-context-with-a-p-hole, $J$ for a p-item-with-a-p-hole, $\Lambda$ for a p-context-with-an-a-hole and $K$ for a p-item-with-an-a-hole, defined, using mutual recursion, as follows:

$$\Delta ::= \Gamma, J \quad J ::= [\ ] \mid \Delta^A \mid \Delta^\Theta \quad \Lambda ::= \Gamma, K \quad K ::= \Gamma^\Sigma \mid \Lambda^\Theta$$

in which we recall from Section 3 that $\Sigma$ indicates an a-context-with-an-a-hole. Various applications of something with an appropriate hole to p-contexts and p-items and combinations of these can be defined in a way similar to that presented in the action calculus.

5.2 Sequents for Propositions and their Interpretations

Our second sequent calculus (see Sub-section 5.3) allows reasoning about relationships between propositions. Sequents are now of the form $\Gamma \vdash m$. Given the meanings of action items and contexts in a multiplicative $\text{LMAM}_Q$ as defined above, and with the obvious notion of an interpretation of atomic propositions in a $\text{DLAM}$ acted on by $Q$, extended in the obvious way to compound propositions, the meanings of p-items and of p-contexts are obtained by mutual induction on their structure:

$$[\Gamma^A] = \blacklozenge_A([\Gamma]), [\Gamma^\Theta] = [\Gamma] \cdot [\Theta], [I_1, \ldots, I_n] = [I_1] \wedge \cdots \wedge [I_n] \text{ and } [\emptyset] = \top. \quad \text{A sequent } \Gamma \vdash m \text{ is true in an interpretation } [-] \text{ in } M \text{ iff } [\Gamma] \leq [m]; \text{ it is true in } M \text{ iff true in all interpretations in } M, \text{ and it is valid true in every } \text{DLAM}. \text{ That the initial sequents of the following calculus are valid and the rules preserve truth are proved using ideas familiar from the action calculus: if } [\Gamma] \leq [\Gamma'] \text{ then } [\Delta[\Gamma]] \leq [\Delta[\Gamma']] \text{ and if } [\Theta] \leq [\Theta'] \text{ then } [\Lambda(\Theta)] \leq [\Lambda(\Theta')].$$

5.3 Sequent Calculus

We have the following initial sequents (in which $p$ is restricted to being an atom, i.e. in $At$):
The rules for the lattice operations and the modal operators are:

\[
\begin{align*}
\Delta[p_1, p_2] &\vdash m & \Delta[p_1 \land p_2] &\vdash m & \Delta[p_1 \lor p_2] &\vdash m & \Theta[p]\Delta[p_1] &\vdash m & \Theta[p]\Delta[p_2] &\vdash m & \Theta[p]\Delta[\Box p] &\vdash m & \Theta[p]\Delta[\Diamond p] &\vdash m
\end{align*}
\]

The rules for the dynamic operations are:

\[
\begin{align*}
\Theta \vdash q &\quad \Delta[p \cdot q] &\vdash m' &\quad \Delta[p \cdot q] &\vdash m' &\quad \Delta[q \cdot m] &\vdash m' &\quad \Delta[q \cdot m] &\vdash m' &\quad \Delta[q \cdot m] &\vdash m'
\end{align*}
\]

The structural rules are, with \( \Gamma' \) in the first two to ensure admissibility of \( W_k \):

\[
\begin{align*}
\Delta[\Theta \Delta[p]] &\vdash m &\quad \Theta \Delta[\Theta \Delta[p]] &\vdash m &\quad \Theta \Delta[\Theta \Delta[p]] &\vdash m &\quad \Theta \Delta[\Theta \Delta[p]] &\vdash m &\quad \Theta \Delta[\Theta \Delta[p]] &\vdash m
\end{align*}
\]

As in the action logic, the two indicated occurrences of \( p \) in the \( Id \) rule are principal and each right rule has its conclusion’s succedent as its principal formula. But in addition, the \( \Diamond_A R \) rule has \( \Gamma_A \) as its principal item and \( \Gamma' \) (which is there to ensure admissibility of \( W_k \)) as its parameter. Similarly, the \( \cdot R \) rule has \( \Gamma^\Theta \) as its principal item and \( \Gamma' \) as its parameter. [More detailed definitions of such matters are in the long version.]

We also include all the four kinds of initial sequent and all the fifteen rules of the action logic, and all the variants of the \( L \) rules (including \( \perp L, \text{Dist} \) and \( \text{Unit} \)) of the action logic obtained by replacing any \( \Sigma \) by \( \Lambda \) and the succedent action \( q \) by a proposition \( m \). These rules are \( \perp L, 1L, \land L, \lor L, \cdot L, \Diamond_L, \Box L, \text{Dist} \) and \( \text{Unit} \); we thus obtain propositional variants.
The next four lemmas are provable by routine methods:

**Lemma 5.1** All sequents $\Gamma, m \vdash m$ are derivable.

**Lemma 5.2** The following Weakening and Contraction rules are admissible:

\[
\frac{\Delta \vdash m}{\Delta, \Gamma \vdash m} \quad \text{Wk} \\
\frac{\Delta, \Gamma \vdash m}{\Delta \vdash m} \quad \text{Contr}
\]

**Lemma 5.3** The rules $\land L$, $\lor L$, $\lozenge A L$, $\Box A L$, $\cdot L$, $\land R$, $\lor R$, and $DyR$ are invertible.

**Lemma 5.4** The rules $\top L^-$ and $\bot R^-$ are admissible:

\[
\frac{\Gamma \vdash \bot}{\Delta[\Gamma] \vdash \bot} \quad \text{R-} \\
\frac{\Theta \vdash \bot}{\Delta[\Theta] \vdash \bot} \quad \text{R-} \\
\frac{\Delta[\Gamma] \vdash \bot}{\Delta[\Theta] \vdash \bot} \quad \text{L-} \\
\frac{\Delta[\Theta] \vdash \bot}{\Delta[\Gamma] \vdash \bot} \quad \text{L-}
\]

### 5.4 Admissibility of DyCut and PrCut

**Theorem 5.5** The following DyCut and PrCut rules are admissible:

\[
\frac{\Theta \vdash q \quad \Delta[q] \vdash m}{\Delta[\Theta] \vdash m} \quad \text{DyCut} \\
\frac{\Gamma \vdash m \quad \Delta[m] \vdash m'}{\Delta[\Gamma] \vdash m'} \quad \text{PrCut}
\]

**Proof.** Strong induction on the rank of the cut, where the rank is given by the pair: (size of cut formula $m$, sum of heights of derivations of premisses). Full details are in the long version. \(\Box\)

### 6 Main Result

**Theorem 6.1 (Soundness and Completeness)** Any derivable sequent is valid; any valid sequent is derivable.

**Proof.** First, we show that the initial sequents of the sequent calculus of actions are valid and that the rules are truth-preserving. For the converse, we follow the Lindenbaum-Tarski proof method. As usual, cut-admissibility is essential for showing transitivity of $\vdash$. Similar methods are followed for the calculus of propositions. Full details are in the long version. \(\Box\)
7 Interpretation and Assumption Rules

The interpretations of propositional and action connectives are as usual [1]: ∧, ∨ for conjunction and disjunction on propositions and non-deterministic choice and parallel composition for actions, • (⊗ of linear logic) is the sequential composition of actions, 1 is the skip action. The dynamic connectives [q]m and m·q are the weakest precondition and strongest postcondition, with interpretations as explained above in Subsection 4.1. As for the epistemic modalities, ♦_A expresses the appearance to agent A of, or uncertainty of agent A about, a proposition (or an action), and □_A expresses the belief of agent A about a proposition (or an action). The uncertainties are all the propositions (or actions) that appear to A as true (or as happening in case of actions) when the proposition m is indeed true (or the action q is indeed happening).

In a relational semantics, these assumptions are given by the accessibility relation, e.g. that a state (or action) can access certain other states (or actions). In the case of actions, these enable us to express honest and dishonest public and private announcements. The Dist and DyDist rules allow for the uncertainty of a proposition updated by a composition of actions to be decomposed to the uncertainties of the actions and the uncertainties of the proposition. A belief after a sequential composition of actions can thus be derived from the belief before it.

Each epistemic scenario has assumptions about the basic actions and atomic propositions involved in the scenario. These are represented algebraically in [3] and we present them proof theoretically here. First, not every action is applicable to every proposition; we can represent the assumption, that a basic action σ is inapplicable when a proposition k holds, by the rule Ker_(σ,k). Second, the facts of the world (i.e. atomic propositions p) are stable under basic actions such as announcements; so if a propositional atom p is true before a basic action σ, it will stay true after it; this is represented by the rule Stab_(p,σ). Finally, each agent A has some uncertainty about each atomic proposition p (or basic action σ). So we have one or more assumptions of the form “appearance to agent A of atomic proposition p is proposition n” and “appearance to agent A of basic action σ is the action w”, represented by PrApp_(A,p,n) and App_(A,σ,w) respectively:

\[
\begin{align*}
\Gamma \vdash k & \quad \Delta[\Gamma] \vdash m \quad \text{Ker}_{(\sigma,k)} \\
\Delta[\Gamma, p] \vdash n & \quad \Delta[\Gamma] \vdash m \quad \text{PrApp}_{(A,p,n)} \\
\Delta[\Gamma] \vdash m & \quad \Delta[\Gamma, p]^A \vdash \sigma \quad \text{Stab}_{(p,\sigma)} \\
\Delta[\Gamma] \vdash m & \quad \Delta[\Gamma^A] \vdash \sigma \quad \text{App}_{(A,\sigma,w)}
\end{align*}
\]

**Lemma 7.1** The calculus with the assumption rules admits Wk and Contr.

**Theorem 7.2** The calculus with the assumption rules admits DyCut and PrCut.

**Proof.** The Ker rules have σ basic; the Stab rule has p atomic and σ basic; similarly the PrApp and App rules have p atomic and (resp.) σ basic. So, no new principal cuts can be formed. The admissibility of the various forms of Cut is then just an
adaptation of the previous proof, with more permutations but no new principal cuts. A similar method is used in [14].

\section{Application}

A well-known epistemic scenario is that of the muddy children, as follows: \( n \) children are playing in the mud and \( k \) of them have muddy foreheads. Each child can see the other children's foreheads, but cannot see his own. Father announces to them “At least one of you has a muddy forehead.” and then asks “Do you know it is you who has a muddy forehead?” After \( k - 1 \) rounds of honest “no” answers by all the children, the muddy ones infer that they are muddy. We can also reason about variants with dishonesty. For instance, if the father falsely announces that at least one child is muddy, then all the children will wrongly believe that they are muddy. Or the muddy children are liars and after \( k - 1 \) honest answers they lie that they do not know that they are muddy; the clean children then wrongly infer that they are muddy.

For a formalization of the honest version, assume the children are enumerated and the first \( k \) are muddy. Consider the propositional atoms \( D_i \) for “child \( i \) is muddy” and \( C_i \) for “child \( i \) is clean”. Since the actions of this scenario are all epistemic, these propositional atoms are stable under all basic actions. Child \( i \) can see the other children’s foreheads: we capture this by \( PrApp(i,D_j,D_j) \) and \( PrApp(i,C_j,C_j) \) for \( i \neq j \). But he cannot see his own, so he is uncertain about himself being muddy or not, captured by \( PrApp(i,D_i,D_i \lor C_i) \) and \( PrApp(i,C_i,D_i \lor C_i) \). We denote father’s initial announcement by basic action \( \sigma \) and children’s “no” replies by basic action \( \sigma' \). These actions are honest public announcements, so their appearance to any child \( i \) is identity, captured by \( App(i,\sigma,\sigma) \) and \( App(i,\sigma',\sigma') \). Father’s initial announcement cannot happen when there is no muddy child, and children’s “no” replies cannot happen if any of them knows that he is muddy: these are captured by \( Ker(\sigma,\land_{i=1}^n C_i) \) and \( Ker(\sigma',\lor_{i=1}^n \neg D_i) \), respectively.

Fig. 1 presents the proof tree for \( n = 3, k = 2 \) for the proposition that, after father’s announcement and one round of “no” answers, a muddy child \( j = 1 \) knows that he is muddy. Consider the first dishonest variant of the puzzle; we denote father’s lying action by the basic action \( \overline{\sigma} \), which appears (in the honest version of the puzzle) to a child \( i \) as his honest announcement \( \sigma \); this is captured by the assumption rule \( App(i,\overline{\sigma},\sigma) \). The action \( \overline{\sigma} \) cannot happen when a child is muddy; this is captured by the assumption rule \( Ker(\overline{\sigma},\lor_i D_i) \). Figure 2 presents the proof of the proposition that, when no child is muddy and after father’s lie that at least one child is muddy, child \( i = 2 \) believes that he is muddy. Assumption rule instances and the proof tree for the second dishonest variant are presented in the appendix of the long version.

Our calculus allows a proof procedure close to human reasoning. In Figure 1, the lowest four steps \( \land L, DyR, \bullet L \) and \( \square_1 R \) rewrite the original statement into a normal form. Then \( DyDist \) and \( Dist \) apply the belief update procedure and decompose the uncertainty of child 1 about the updated proposition to the update of his propositional uncertainty by his action uncertainty. The forking rule \( \lor L \)
Fig. 1. Proof tree for the honest muddy children puzzle.

Fig. 2. Proof tree for the lying-father variant of the muddy children puzzle.
9 Decidability and Complexity

Our sequent calculus has a weak form of the sub-formula property (of a kind studied in [6]) adequate to allow the calculi to be called “analytic”; proof search is therefore effective (even if not efficient) and decidability is routine (since there are none of the issues from first-order logic of choosing terms).

A proof search procedure has been implemented in [19] using a loop-checking technique and variations of the Bounded Depth-First tree expansion strategy. These all have a polynomial space complexity \((n \log n)\), but the worst case time complexity of our optimal terminating strategy is \(O(b^d)\) for \(b\) the branching factor and \(d\) the depth of the shortest proof.

10 Summary and future work

Our nested sequent calculus, for a positive logic of propositions and actions with adjoint epistemic and dynamic modalities, admits cut, and is sound and complete with regard to an algebraic semantics inspired by [1,3]. With added assumption rules) the calculus is applicable to reasoning about dynamic epistemic scenarios. Existing proof theoretic treatments of dynamic epistemic logics, e.g. [13,2] only consider specific actions of public announcements. The only other calculus [3] that allows for arbitrary announcements (public/private, honest/dishonest) lacks cut-admissibility. Our work shows that dynamic epistemic logics can be represented in terms of a cut-free sequent calculus, albeit one with rules admitting substantial non-determinism in root-first proof search. The implementation [19] of this sequent calculus provides an automated tool for encoding scenarios involving both epistemic operators and actions. Refinements of the calculus, addressing issues such as termination and backtracking, are yet to be developed.

References


Game Semantics in the Nominal Model

Murdoch Gabbay
www.gabbay.org.uk

Dan Ghica
www.cs.bham.ac.uk/~drg

Abstract
We present a model of games based on nominal sequences, which generalise sequences with atoms and a new notion of coabstraction. This gives a new, precise, and compositional mathematical treatment of justification pointers in game semantics.

Keywords: Game semantics, nominal sets, nominal abstraction and coabstraction, equivariance

1 Introduction

Game semantics is a successful collection of techniques for giving denotations to logic and computation. It came to particular prominence by solving the open problem of full abstraction for PCF [2,19] and is widely used from philosophy and logic, to model checking and synthesis of digital circuits [22,13].

The game metaphor is a dialogue between Proponent and Opponent. This is intensional: a play of a game records interactions between a term (the Proponent) and its context (the Opponent), and how they are scheduled.

One way to model a play is as a labelled acyclic graph called a pointer sequence. Each node in the graph is a Proponent or Opponent move and edges in the graph represent the justification for that move. Thus, a pointer sequence records what moves were made and in what order, and also why.

We propose a model of games based on nominal sets, inspired by pointer sequences—indeed it is virtually isomorphic to them—with the difference that we model edges using atoms from nominal techniques. Why this is useful will become clear in a moment.

Atoms are just a countably infinite set of distinct symbols: a, b, c, . . . . A diagram shows how these can model pointer sequences. The pointer sequence on the left

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.com/locate/entcs
corresponds to the nominal sequence on the right:

\[q, q', a, a', a''\] corresponds to \[qc[a] q'a[b] aa a'b a''b\]

Questions and answers are written \(q\) and \(a\), and atoms are used as pointers. The symbols \([a]\) and \([b]\) can be thought of as naming the questions \(q\) and \(q'\) and are binders into the ‘future’. So we see above that \(q\) justifies two moves: \(q'\) and \(a\). Pointers (arrows, in the diagram above) are rendered as a pair of atoms. The tip of the arrow is represented by a coabstraction \([b]\) which must be unique (this is formalised by the condition \(b \notin atoms(e)\) in Definition 2.8). The tail of the arrow, which need not be unique, is an occurrence of the name. This deals straightforwardly with dangling pointers, which are viewed just as free names; in the sequence above \(c\) is free. Nominal sequences have the following good properties:

(i) A subsequence of a nominal sequence is a nominal sequence. A subgraph of a pointer sequence is not a pointer sequence, because it might have ‘dangling pointers’. In that sense, nominal sequences generalise pointer sequences and help talk easily about ‘open sequences’ (easy handling of open elements is a typical benefit of nominal techniques).

(ii) A concatenation of two nominal sequences is a nominal sequence; names link up and there are no reindexing isomorphisms. It is not so clear how to concatenate pointer sequences.

(iii) Nominal sequences are an inductive datatype and can be manipulated with standard tools (to fully benefit would require a mechanised nominal system [26] but we shall see our sequences simplify paper-and-pencil proofs too).

There is an important, specifically nominal advantage to using names in particular: it enables a particularly efficient management of renaming pointers to avoid ‘accidental clash’. It is important and useful that we use names to name moves and not e.g. numbers, because names are by definition symmetric; not only can we use permutations to \(\alpha\)-convert, but taking names and their permutative symmetry as primitive saves effort since permutative symmetries propagate necessarily to the things we will build using them, such as plays and strategies. This style of name management is characteristic of nominal techniques and we shall see that it is effective here.

We try to formalise [19,23] and [14], at low overhead. The decoration of sequences by atoms is no more of an overhead than decoration by pointers, and equivariance is a very efficient way to manage renamings, so the overhead is low and the advantages in precision and conciseness appear to be significant.

We cannot replicate all definitions and proofs from these two large papers in this conference paper but we hope that it will be entirely obvious to the reader how this could be done. We do not claim to make the work above trivial. However, we do claim that using our formulation, game semantics can be carried out more quickly, more accurately, and more transparently.

This is important for more than good practice: we speculate that by our formula-
tion, implementation and mechanisation of game semantics proofs are significantly easier. The reader can compare the definitions in this paper with those from [19] or [14] and judge which would be easier to work with, in a prover like Isabelle. Furthermore, game semantics can provide theoretical foundations for program verification and for hardware synthesis, where the pen-and-paper style of much previous work must be augmented by machine-checked proofs, because of scale, or for safety, or both. Here, the compositionality, computational, and symmetry properties enumerated and discussed above really count. Finally, game semantics can reconcile the compositionality of denotational semantics with the effectiveness of operational semantics via communicating abstract machines [16]; here, the conventional representation of pointers is arguably actively counterintuitive, whereas the use of names as tags for messages carries immediate computational intuitions.

2 Nominal game theory

2.1 Nominal sequences

Definition 2.1 Fix disjoint countably infinite set of atoms $A$, and constants. $a, b, c$ will range over distinct atoms (the permutative convention). $f, g, h$ will range over constants, not necessarily distinct.

Define (nominal) sequences by $e ::= \varepsilon \mid ea \mid ef \mid e[a]$.

Remark 2.2 Call $\varepsilon$ the empty list and write $ee'$ for list concatenation. Call $[a]$ a coabstraction. The reminds us of the atoms-abstraction of nominal techniques [12]—but in $e[a]$, $a$ is bound ‘in the future’ in whatever $e'$ we might concatenate after $e[a]$. We contrast the intended denotations of abstraction and coabstraction in the Conclusion.

Definition 2.3 Define coabstracted and free atoms $ca(e)$ and $fa(e)$ by:

$\begin{align*}
ca(\varepsilon) &= \emptyset \\
ca(ea) &= ca(e) \\
ca(ef) &= ca(e) \\
ca(e[a]) &= ca(e) \cup \{a\} \\
fa(\varepsilon) &= \emptyset \\
fa(ea) &= fa(e) \cup \{a\} \setminus ca(e) \\
fa(ef) &= fa(e) \\
fa(e[a]) &= fa(e)
\end{align*}$

Define the atoms in an expression $atoms(e)$ by $atoms(e) = fa(e) \cup ca(e)$.

Lemma 2.4 $ca(e \ e') = ca(e) \cup ca(e')$ and $fa(e \ e') = fa(e) \cup (fa(e') \setminus ca(e))$.

Definition 2.5 A renaming $\rho$ is a function from atoms to atoms such that $dom(\rho) = \{a \mid \rho(a) \neq a\}$ is finite. Write $id$ for the identity renaming such that $id(a) = a$ and $\rho' \circ \rho$ for composition such that $(\rho' \circ \rho)(a) = \rho'(\rho(a))$.

Call bijective $\rho$ permutations. Following [12] let $\pi$ range over permutations (the application of renamings in a nominal context goes back to [11]).

Definition 2.6 Define a renaming action $\rho \cdot e$ on sequences by:

$\begin{align*}
\rho \cdot \varepsilon &= \varepsilon \\
\rho \cdot (ea) &= (\rho \cdot e)\rho(a) \\
\rho \cdot (ef) &= (\rho \cdot e)f \\
\rho \cdot (e[a]) &= (\rho \cdot e)[\rho(a)]
\end{align*}$
2.2 Nominal game semantics

A game is an arena (Definition 2.7) along with some set of legal plays which are lists of moves by a proponent or opponent—precisely what classes of plays are legal, determines the type of game they play.

Definition 2.7 An arena is a tuple \( \mathcal{A} = (qst_\mathcal{A}, ans_\mathcal{A}, \lambda_\mathcal{A}, \vdash_\mathcal{A}, ini_\mathcal{A}) \) of:

- Disjoint sets of questions \( q \in qst_\mathcal{A} \) and answers \( a \in ans_\mathcal{A} \).
  
  Write \( m \in mvs_\mathcal{A} = qst_\mathcal{A} \cup ans_\mathcal{A} \) for short and call this the set of moves.

- A polarity function \( \lambda_\mathcal{A} : mvs_\mathcal{A} \to \{O, P\} \). Write \( O^* = P \) and \( P^* = O \).

- An enabling relation \( \vdash_\mathcal{A} \subseteq mvs_\mathcal{A} \times mvs_\mathcal{A} \) where \( m \vdash_\mathcal{A} m' \) implies \( m \in qst_\mathcal{A} \) and \( \lambda_\mathcal{A}(m) = \lambda_\mathcal{A}(m')^* \).

- A set of initial questions \( ini_\mathcal{A} \subseteq qst_\mathcal{A} \) such that:
  - \( \lambda_\mathcal{A}(i) = O \) for every initial \( i \in ini_\mathcal{A} \).
  - If \( q \in qst_\mathcal{A} \) and \( i \in ini_\mathcal{A} \) then \( q \not\vdash_\mathcal{A} i \).

Definition 2.8 Define proto-plays \( e \) over an arena \( \mathcal{A} \) inductively by:

\[
e ::= \varepsilon \mid e \, m[a]\[b] \quad (b \not\in atoms(e))
\]

Recall that \( m \in qst_\mathcal{A} \cup ans_\mathcal{A} \). Write \( pply_\mathcal{A} \) for the set of all proto-plays of \( \mathcal{A} \).

Every proto-play is a sequence; not every sequence is a proto-play. It will always be clear what \( e \) ranges over.

Definition 2.9 Call \( m[a]\[b] \) a (named) move; \( m \) will range over named moves.

Remark 2.10 • A proto-play consists of a sequence of named moves, each of which consists of a move \( m \), a justifying name \( a \) and a coabstraction \( [b] \) which we call the name of \( m \). This \( b \) ‘names’ its move, so that a later named move’s justifying name can point to \( m \) by its name \( b \).

- The freshness condition \( b \not\in atoms(e) \) makes \( b \) name its move uniquely in the sequence. This is inefficient—we cannot reuse names even if somehow we know we could—but we optimise for mathematical convenience.

- In the games models of [19,23,14] only questions justify, so for the applications in this paper \( \vdash_\mathcal{A} \subseteq qst_\mathcal{A} \times mvs_\mathcal{A} \) and we can drop the coabstractions naming answers in protoplays (so: \( qa[b] \) but just \( aa \)). However, this complicates definitions and loses generality, so we leave in (dummy) coabstractions and take \( \vdash_\mathcal{A} \subseteq mvs_\mathcal{A} \times mvs_\mathcal{A} \). Answers justifying moves is used to construct ‘coproduct arenas’ in game semantics for call-by-value languages [3].

Definition 2.11 Suppose \( e \) and \( e' \) are sequences. Write \( e' \leq e \) when \( e' = e \) for some \( e'' \); call \( e' \) a prefix of \( e \). Write \( e' \subseteq e \) when \( e'' = e \) for some \( e''' \) and \( e'' \); call \( e' \) a segment of \( e \).

Definition 2.12 Define enabled(\( e \)) the moves enabled by \( e \in pply_\mathcal{A} \) by:

\[
\text{enabled}(\varepsilon) = \emptyset \quad \text{enabled}(e \, m[a]\[b]) = \text{enabled}(e) \cup \{m'b \mid m \vdash_\mathcal{A} m'\}
\]
Lemma 2.13 enabled(e) = ∪{mb | m′a[b] ⊆ e, m′ ⊢_{\mathcal{A}} m}.

Definition 2.14 Given e ∈ pply_{\mathcal{A}} define its underlying sequence |e| by:

|e| = ε  |ema[b]| = |e|m

Definition 2.15 Suppose \mathcal{A} is an arena and A ⊆ A.

(i) Call e ∈ pply_{\mathcal{A}} justified when e′ ma ≤ e and m ∉ ini_{\mathcal{A}} imply ma ⊆ enabled(e′).
(ii) Call e ∈ pply_{\mathcal{A}} well-opened when e′ia[b] ≤ e implies e′ = ε.
(iii) Call e ∈ pply_{\mathcal{A}} strictly scoped when aa[b]e′ ⊆ e implies a ∉ fa(e′), for every e′ ∈ pply_{\mathcal{A}}, a ∈ ans_{\mathcal{A}}, and atom a.
(iv) Call e ∈ pply_{\mathcal{A}} strictly nested when qa[b]e_2 q′b[c] e_3 ab ⊆ e implies a′c ⊆ e_3 for some answering move a′ ∈ ans_{\mathcal{A}}.  
(v) Call e ∈ pply_{\mathcal{A}} alternating when mm′ ⊆ |e| implies λ_{\mathcal{A}}(m) ≠ λ_{\mathcal{A}}(m′).

Definitions 2.16 and 2.17 follow [23, pp. 7-8]:

Definition 2.16 Given justified e ∈ pply_{\mathcal{A}} define the proponent view \( e^\uparrow \) and opponent view \( e^\downarrow \) by:

\[
\begin{align*}
\text{e}^\uparrow &= \varepsilon \\
\text{ema}[b] &= \text{e}^\uparrow \text{ma}[b] & \lambda_{\mathcal{A}}(m) &= \text{P} \\
\text{eia}[b] &= \text{ia}[b] & \lambda_{\mathcal{A}}(m) &= \text{O} \\
\text{eqa}[b]e^\uparrow\text{mb}[c] &= \text{e}^\uparrow\text{qa}[b]\text{mb}[c] & \lambda_{\mathcal{A}}(m) &= \text{O} \\
\end{align*}
\]

\[
\begin{align*}
\text{eqa}[b]e^\downarrow\text{mb}[c] &= \text{e}^\downarrow\text{qa}[b]\text{mb}[c] & \lambda_{\mathcal{A}}(m) &= \text{P} \\
\end{align*}
\]

Definition 2.17 A justified proto-play e ∈ pply_{\mathcal{A}} satisfies visibility when e′qa[b]e''q′b[c] ≤ e implies that if \( \lambda_{\mathcal{A}}(q) = \text{P} \) then qa[b] ⊆ e′qa[b]e''q′ and, if \( \lambda_{\mathcal{A}}(q) = \text{O} \) then qa[b] ⊆ e′qa[b]e''q′.

Remark 2.18 Intuitively, Definition 2.15 means:

- \( e \) is justified when every non-initial move responds to a preceding move.
- \( e \) is well-opened when the initial move is unique and first in the sequence.
- \( e \) is strictly scoped when a question can receive at most one answer. If we read games as processes, this means answering a question stops the process associated with that question.
- \( e \) is strictly nested when questions are answered in (reverse) order. This forbids starting a process b, then c from inside b, then stopping b before c.

The intuition of \( e \) alternating seems clear but it does not have directly to do with names and binding, so we will not consider it further. Visibility is subtle, typical of languages that are pure or have only ground-type state. We have shown above how to formalise it in our framework, but the associated proofs are inherently non-trivial and for reasons other than the handling of names and binding, so this too we will not consider further.

---

1 What is important here is the atom c.
2 \( \varepsilon.e.\) defines a total function on justified e; ia[b]e′ where \( \lambda_{\mathcal{A}}(i) = \text{P} \) is not justified.
3 In [14, p. 7] ‘strictly scoped’ is called fork and ‘strictly nested’ is called join.
Remark 2.19 We can now characterise the plays of HO-games (the games from [19]) and GM-games (those from [14]). Suppose $\mathfrak{A}$ is an arena and $e \in pply_\mathfrak{A}$ is a proto-play. Then:

- In HO-games, $e$ is a legal play when $fa(e) = \{a\}$ for some $a \in \mathfrak{A}$ and $e$ is justified, well opened, alternating, strictly nested and satisfies visibility (see [19, Def. 4.2, Def. 4.4]).
- In GM-games, $e$ is a legal play when $fa(e) = \{a\}$ as above and $e$ is justified, well-opened, strictly scoped, and strictly nested (see [14, Def. 1]).

The condition $fa(e) = \{a\}$ implies $e$ has one free atom $a$; one ‘dangling pointer’. With being well-opened, this ensures $a$ names the initial question.

How do we choose $a$ above? We do not. It is a non-evident design decision that proto-plays do not have $\alpha$-conversion on coabstracted atoms. This preserves compositionality: if $[a]a$ equals $[b]b$ then $[a]ab$ equals $[b]bb$, which is nonsense.\(^4\)

In our framework $\alpha$-conversion lives in strategies (sets of proto-plays), which are subject to an equivariance (symmetry) condition up to the choice of atoms in the proto-plays they contain. So $\alpha$-equivalence does not live in the elements, it lives in the sets of elements. More on this in Remark 5.3.

3 Operations on plays

3.1 Deletion of moves from a play

We often want to delete moves from pointer sequences, reflecting ‘hiding’ of irrelevant parts of a computation (see e.g. Definition 5.2). But pointers into and out of deleted moves need to be updated. Definition 3.1 and Proposition 3.4 make that formal for our nominal framework. The culminating result of this subsection is Theorem 3.11, which uses Proposition 3.4 amongst other constructions to show that properties of proto-plays are preserved by deletion.

Definition 3.1 Suppose $\mathcal{X} \subseteq mvs_\mathfrak{A}$ is some set of moves from an arena $\mathfrak{A}$, and $e \in pply_\mathfrak{A}$. Define deletion $e|\mathcal{X}$ inductively on $e$ as follows, where we take inductively $(f, \rho) = e|\mathcal{X}$ and $m' \not\in \mathcal{X}$ and $m \in \mathcal{X}$:

$$
\varepsilon|\mathcal{X} = (\varepsilon, id) \quad (e m'[a[b]], \mathcal{X} = (f m'[\rho(a)[b]], \rho) \quad (e ma[b])[\mathcal{X} = (f, \rho[b := \rho(a)])]
$$

We may write $e|\mathcal{X}$ for $e|mvs_\mathfrak{A}$ (deletion of the set of moves of $\mathfrak{A}$).

Remark 3.2 Intuitively $e|\mathcal{X}$ is ‘$e$ with the moves in $\mathcal{X}$ deleted’. Some reindexing has to take place when we do this: e.g. if $qa[b]$ is deleted then any pointers to $b$ are ‘reattached’ so that they point to wherever $a$ points to:

In the diagram above the shaded nodes (circles) are in $\mathcal{X}$ and are deleted.

\(^4\) It is possible to reconcile $\alpha$-conversion with proto-plays, by appending a ‘future permutation’, like so: $[a]e$. Then $[a]leq$ equals $[b](b_{(a)})b$, not $[b]ab$. This is not needed here.
Proof By induction on 
Definition 3.9
Call
Suppose
Lemma 3.10
Lemma 3.8
If
Lemma 3.7
Proof
fa
enabled
Lemma 3.5
Suppose
Lemma 3.6
of deletions can be dealt with similarly. forming an entire sub-tree in the arena, preserves legality properties. Other kinds tion 2.15. Legality is not preserved by arbitrary deletions, but deletion is usually used in a controlled way which ensures preservation. For instance deletion of moves forming an entire sub-tree in the arena, preserves legality properties. Other kinds of deletions can be dealt with similarly.

Lemma 3.6 Suppose \( \mathcal{X} \subseteq \text{mv}_\mathfrak{A} \) and \( e \in \text{pply}_\mathfrak{A} \). Write \( e \mid \mathcal{X} = (f, \rho) \). Then \( \text{fa}(f) \subseteq \text{fa}(e) \) and \( \text{ca}(f) \subseteq \rho \cdot \text{ca}(e) \).

Lemma 3.7 Suppose \( e \in \text{pply}_\mathfrak{A} \). If \( \text{ma} \in \text{enabled}(e) \) then \( a \in \text{atoms}(e) \).

Proof By a routine induction on the proto-play \( e \), using Definition 2.12.

Lemma 3.8 If \( \text{ma}[b] \subseteq e \in \text{pply}_\mathfrak{A} \) and \( m'b \in \text{enabled}(e) \) then \( m \vdash m' \).

Proof By induction on \( e \). We consider one case:
• The case \( e \cdot \text{ma}[b] \). Suppose \( m'b \in \text{enabled}(e \cdot \text{ma}[b]) \). By assumption in Definition 2.8 \( b \notin \text{atoms}(e) \) and by Lemma 3.7 \( m'b \notin \text{enabled}(e) \). Unpacking Definition 2.12 it follows that \( m \vdash m' \).

Definition 3.9 Call \( \mathcal{X} \subseteq \text{mv}_\mathfrak{A} \) closed under \( \vdash \mathfrak{A} \) when \( m \in \mathcal{X} \) and \( m \vdash m' \) implies \( m' \in \mathcal{X} \).

Lemma 3.10 Suppose \( \mathcal{X} \subseteq \text{mv}_\mathfrak{A} \) is closed under \( \vdash \mathfrak{A} \). Suppose \( e \mid \mathcal{X} = (e', \rho) \). Then if \( \text{ma} \in \text{enabled}(e) \) and \( \text{ma} \notin \mathcal{X} \) then \( \text{ma}(a) \in \text{enabled}(e') \).

\( ^5 \rho \cdot e \) is a nominal sequence but it might not be a proto-play because coabstracted atoms need not be distinct. Also \( e \cdot \mathcal{X} \) need not be legal because naive deletion does not update links. Proposition 3.4 shows that Definition 3.1 calculates \( \rho \) and \( \mathcal{X} \) such that if we do these two naive operations together, then we are all right.

\( ^6 \) This is the crux of the proof: because \( b \) is fresh, changing \( \rho \) to \( \rho[b:=\rho(a)] \) does not change whatever we have calculated so far.
Proof By Lemma 3.5 it suffices to show that if $ma \in \text{enabled}(e)$ then $m\rho(a) \in \text{enabled}((\rho\cdot e)\cdot X)$. We work by induction on $e$ and consider one case:

- **The case of $\text{ema}'[a]$.** Write $e|X = (e', \rho)$ and suppose $m'a \in \text{enabled}(\text{ema}'[a])$ and $m' \not\in X$. By Lemma 3.8 $m \models m'$. Since $m' \not\in X$ it follows by closure of $X$ under $\models X$ that $m \not\in X$. So $(\rho'(\text{ema}'[a]))-X = ((\rho\cdot e)-X)(m\rho(a')[\rho(a)])$. By Definition 2.12, $m'\rho(a) \in \text{enabled}(((\rho\cdot e)-X)(m\rho(a')[\rho(a)]))$.

**Theorem 3.11** Suppose $X \subseteq m\text{vs}_A$ and $e|X = (f, \rho)$.

(i) If $X \subseteq m\text{vs}_A$ is closed under $\models X$ then if $e$ is justified then so is $f$.
(ii) If $\text{ini}_X \cap X = \emptyset$ then if $e$ is well-opened then so is $f$.
(iii) If $e$ is strictly scoped then so is $f$.
(iv) If $X \subseteq m\text{vs}_A$ is closed under $\models X$ then if $e$ is strictly nested then so is $f$.

**Proof**

(i) Suppose $f'\cdot m\rho(b) \leq f$ where $e|X = (f, \rho)$ and $m \not\in \text{ini}_X$. Using Proposition 3.4 $f'\cdot m\rho(b) = ((\rho\cdot e)-X)m\rho(b)$ for some $e'mb \leq e$, and also $m \not\in X$. Since $e$ is justified, by Lemma 2.13 it must be that $q \models m$ for some $qa[b] \subseteq e$. Since $X$ is closed under $\models X$ we know $q \not\in X$. It follows by Proposition 3.4 that $q\rho(a)[\rho(b)] \subseteq f'$ and we are done.

(ii) By an easy argument using Proposition 3.4.

(iii) Suppose $ap(a)f' \subseteq f$. Then $aa'e \subseteq e$ for some $e' \in \text{ply}_X$. Since $e$ is strictly scoped we know that $a \not\in fa(e')$. By Lemma 3.6 also $a \not\in fa(f')$.

(iv) Much as the previous case.

### 3.2 Restriction to a hereditarily justified subplay

The structure of this subsection resembles that of Subsection 3.1. We have a more complex operation than deletion; extracting the **hereditarily justified** sub-pointer sequence. In our framework the definition is absolutely routine; we just take a subsequence. This is Definition 3.12; then Proposition 3.14 shows how to quickly calculate the relevant subsequence using names, and Theorem 3.18 expresses how properties are preserved.

**Definition 3.12** Suppose $e \in \text{ply}_X$ and $A \subseteq \mathcal{A}$. Define the **hereditarily justified** proto-play $e|A \subseteq \text{ply}_X$ as follows, where we take $(f, B) = e|A$ and $a \in B$ and $a' \not\in B$:

$$
\varepsilon|A = (\varepsilon, A) \quad (e ma[b])|A = (f ma[b], B \cup \{b\}) \quad (e ma'[b])|A = (f, B)
$$

**Definition 3.13** Suppose $e \in \text{ply}_X$ and $A \subseteq \mathcal{A}$. Define $e@A$ as follows, where $a \in A$ and $a' \not\in A$ (the resemblance with atoms-concretion from [12] is deliberate):

$$
\varepsilon@A = \varepsilon \quad (e ma[b])@A = (e@A) ma[b] \quad (e ma'[b])@A = e@A
$$

**Proposition 3.14** If $e|A = (f, B)$ then $e@B = f$.

**Corollary 3.15** Suppose $e|A = (f, B)$. Then:

- If $f'ma[b] \leq f$ then $e'$ exists such that $e'ma[b] \leq e$ and $(e'ma[b])@B = f'ma[b]$.

172
Suppose Lemma 4.3 composition of HO or GM strategies is well-defined. 

Evaluating composition of strategies, so preservation of legality is essential to show that plays—deletion and unravelling—preserve certain well-formedness properties which This is Lemmas 4.3 and 4.5. These state that two important operations on proto-plays—deletion and unravelling—preserve certain well-formedness properties which

Definition 4.1 (i) Suppose f is a function on a set X and g is a function on a disjoint Y. Write \([f, g]\) for the co-pairing function on \(X \cup Y\) such that \([f, g](x) = f(x)\) and \([f, g](y) = g(y)\) for \(x \in X\) and \(y \in Y\) respectively. 

(ii) Suppose g is a function to \(\{O, P\}\). Write \(g^*\) for the function mapping x to g(x)* (Definition 2.7).

Definition 4.2 Define product \(\mathfrak{A} \times \mathfrak{B}\) and arrow \(\mathfrak{A} \Rightarrow \mathfrak{B}\) of arenas by:

- \(\mathfrak{A} \times \mathfrak{B} = (qst_\mathfrak{A} + qst_\mathfrak{B}, ans_\mathfrak{A} + ans_\mathfrak{B}, [\lambda_\mathfrak{A}, \lambda_\mathfrak{B}], \vdash_\mathfrak{A} + \vdash_\mathfrak{B}, ini_\mathfrak{A} + ini_\mathfrak{B})\)
- \(\mathfrak{A} \Rightarrow \mathfrak{B} = (qst_\mathfrak{A} + qst_\mathfrak{B}, ans_\mathfrak{A} + ans_\mathfrak{B}, [\lambda_\mathfrak{A}, \lambda_\mathfrak{B}], \vdash_\mathfrak{A} + \vdash_\mathfrak{B}, ini_\mathfrak{A} + ini_\mathfrak{B})\)

- Above, the symbol + denotes disjoint sets union (for convenience assume sets of moves of distinct arenas are distinct), and

- \(ini_\mathfrak{B} \times ini_\mathfrak{A} = \{(i', i) \mid i' \in ini_\mathfrak{B}, i \in ini_\mathfrak{A}\}\). So \(\vdash_\mathfrak{A} + \vdash_\mathfrak{B} + ini_\mathfrak{B} \times ini_\mathfrak{A}\) is the disjoint union of the enabling relations of \(\mathfrak{A}\) and \(\mathfrak{B}\), disjoint union \(ini_\mathfrak{A} \times ini_\mathfrak{B}\)

We show how from proto-plays in \(\mathfrak{A} \Rightarrow \mathfrak{B}\) we recover proto-plays in \(\mathfrak{A}\) and \(\mathfrak{B}\). This is Lemmas 4.3 and 4.5. These state that two important operations on proto-plays—deletion and unravelling—preserve certain well-formedness properties which define the notion of HO and GM legal plays. These operations are key to formulating composition of strategies, so preservation of legality is essential to show that composition of HO or GM strategies is well-defined.

**Lemma 4.3** Suppose \(e \in pply_{\mathfrak{A} \Rightarrow \mathfrak{B}}\) and \(e \downarrow mvs_{\mathfrak{B}} = (f, \rho)\). Then \(f \in pply_{\mathfrak{B}}\). If \(e\) is justified, well-opened, strictly scoped, or strictly nested, then so is \(f\).

**Proof** For the first part, by Proposition 3.4 \(f\) contains only moves in mvs_{\mathfrak{B}}. The second part follows by Theorem 3.11 and we note that the enabling relation \(\vdash_{\mathfrak{A} \Rightarrow \mathfrak{B}}\) restricted to the moves mvs_{\mathfrak{B}}, is just \(\vdash_{\mathfrak{B}}\).

**Definition 4.4** Define the **unravelling** of \(e \in pply_{\mathfrak{A}}\) by unravel\((e) = \{e\downarrow\{a\} \mid a \in fa(e)\}\).
Unravelling is key to constructing exponential games [23, Sec 2.4]. Intuitively, in a play in \( \mathcal{A} \Rightarrow \mathcal{B} \) we can recover one play in \( \mathcal{B} \), by deleting the moves of \( \mathcal{A} \). Removing the moves in \( \mathcal{B} \) yields an interleaved set of plays of \( \mathcal{A} \). Unravelling separates these plays by following pointers, as illustrated:

![Unravelling Diagram]

It is easy to see that if \( e \) is justified then \( \text{unravel}(e) \) captures the idea of “the set of threads in \( e \)”, and if \( e \) is additionally well-opened then \( \text{unravel}(e) = \{e\} \).

**Lemma 4.5** If \( e \in \text{ply}_{\mathcal{A} \Rightarrow \mathcal{B}} \) then \( \text{unravel}(e|\mathcal{B}) \subseteq \text{ply}_{\mathcal{B}} \). If \( e \) is justified / well-opened / strictly-scoped / strictly-nested then so is every \( f \in \text{unravel}(e|\mathcal{B}) \).

**Proof** Directly from Lemma 4.3.

---

## 5 Strategies

### 5.1 Strategies and equivariance

**Definition 5.1** Call \( \sigma \subseteq \text{ply}_{\mathcal{A}} \) equivariant when \( e \in \sigma \) implies \( \pi \cdot e \in \sigma \) for every permutation \( \pi \). Write \( \sigma : \mathcal{A} \) when \( \sigma \) is an equivariant subset of \( \text{ply}_{\mathcal{A}} \) and call \( \sigma \) a strategy. (The notion of strategy is usually subject to further constraints; these are discussed below.)

Recall deletion \( e|\mathcal{A} \) from Definition 3.1. We follow [23, Sec. 2.2.3]:

**Definition 5.2** Suppose \( \mathcal{A}, \mathcal{B}, \) and \( \mathcal{C} \) are arenas on disjoint moves. Write \( \text{int}(\mathcal{A}, \mathcal{B}, \mathcal{C}) \) for the arena obtained taking relevant unions of moves, polarities, and enabling relations.\(^7\) Then for strategies \( \sigma : \mathcal{A} \Rightarrow \mathcal{B} \) and \( \tau : \mathcal{B} \Rightarrow \mathcal{C} \) define their interaction and composition by:

\[
\sigma \| \tau = \{e \in \text{int}(\mathcal{A}, \mathcal{B}, \mathcal{C}) \mid \pi_1(e|\mathcal{C}) \in \sigma \land \pi_1(e|\mathcal{B}) \in \tau\} \quad \sigma ; \tau = \{\pi_1(e|\mathcal{B}) \mid e \in \sigma \| \tau\}
\]

Here \( \pi_1 \) is first projection; a technical annoyance, since deletion returns a pair. This is the linear version of strategy composition; exponential games are constructed using the concept of unravelling introduced earlier (Definition 4.4).

**Remark 5.3** Equivariance is symmetry under permuting atoms. Names fulfil the function that links fulfil in e.g. [19,23,14]. Permutive symmetry of strategies amounts to saying ‘we can \( \alpha \)-rename’. So proto-plays do not have \( \alpha \)-equivalence in our framework but sets of proto-plays do (cf. [10]). Thus, Theorem 5.4 becomes a one-line argument by symmetry/equivariance. This avoids arguments about \( \alpha \)-renaming, reindexing, or relinking that would be needed if we used numbers or explicitly linked lists. So we have:

\(^7\) We never use the polarity or enabling relations; we just want to generate proto-plays over questions and answers taken from \( \mathcal{A}, \mathcal{B}, \) and \( \mathcal{C} \).
**Theorem 5.4** If $\sigma$ and $\tau$ are equivariant then so is $\sigma; \tau$.

**Proof** Using [7, Theorem 4.4], which is a fancy way of saying that Definition 5.2 is symmetric in permuting atoms, so if the inputs $\sigma$ and $\tau$ are symmetric then so is the output.

Just as for proto-plays, GM and HO strategies are subject to constraints. We sketch, sometimes in detail, how these can be expressed.

Two standard conditions are being prefix closed and opponent closed; see [19, Section 5] (where opponent closed is called contingent completeness) or [14, Definition 4]. These are straightforward to formalise:

**Definition 5.5** Call $\sigma \subseteq \text{pply}_A$ prefix closed and opponent-closed respectively when:

\[
\begin{align*}
ema[b] & \in \sigma \\
\ell & \in \sigma \\
\lambda_A(m) &= O \\
ema[b] & \in \text{pply}_A
\end{align*}
\]

5.2 The asynchrony preorder on proto-plays

In [14] the authors were interested in modelling asynchronous concurrency. Accordingly strategies must be saturated under certain move swappings [14, Subsection 2.5] (the idea goes back to [25]).

**Definition 5.6** Call a relation $\leq$ on sequences compatible when $e \leq e'$ implies $ef \leq e'f$ and $fe \leq f'e'$. Define $\preceq$ on $\text{pply}_A$ to be the least compatible preorder such that:

\[
\begin{align*}
(b /\notin fa(e), \lambda_A(m) &= O) & \quad (b /\notin fa(e), \lambda_A(m) = P) \\
ma[b] e & \leq e ma[b] \\
(bX) & \quad e ma[b] \leq ma[b] e \\
(bXm)
\end{align*}
\]

Call $\sigma \subseteq \text{pply}_A \preceq$-saturated when $e \in \sigma$, $e' \in \text{pply}_A$ and $e' \preceq e$ imply $e' \in \sigma$.

**Remark 5.7** It may be worth quoting the definition from [14] (text just before Definition 6) for comparison with Definition 5.6:

...we define a preorder $\preceq$ on $\text{play}_A$ for any arena $A$ as the least reflexive and transitive relation satisfying $s' \preceq s$ for all $s, s' \in \text{play}_A$

(i) $s'=s_0 \cdot o \cdot s_1 \cdot s_2$ and $s=s_0 \cdot s_1 \cdot o \cdot s_2$, or

(ii) $s'=s_0 \cdot s_1 \cdot p \cdot s_2$ and $s=s_0 \cdot p \cdot s_1 \cdot s_2$,

where $o$ is any $O$ move and $p$ is any $P$ move and the justification pointers in $s$ are “inherited” from $s'$... . . .

Nominal sequences help make these intuitions formal.

In [14, Lemma 7] a small step version $\preceq'$ is given and the equality $\preceq'=\preceq$ is claimed. With what we have so far, this is a routine inductive argument:
Definition 5.8 Give \( \{O, P\} \) a partial order such that \( O \leq O, O \leq P, \) and \( P \leq P \). Define a preorder \( \preceq' \) on closed sequences to be the least reflexive transitive relation such that:

\[
\frac{(\lambda_\varphi(m_1) \leq \lambda_\varphi(m_2))}{m_1a_1[b_1] m_2a_2[b_2] \preceq' m_2a_2[b_2] m_1a_1[b_1]} \quad (\text{smm}) \quad \frac{(\lambda_\varphi(m_1) \leq \lambda_\varphi(m_2))}{m_1a_1[b_1] m_2a_2[b_2] \preceq' m_2a_2[b_2] m_1a_1[b_1]} \quad (\text{smm'})
\]

Here is the asynchronous swapping rule (smm) interpreted for \( q_1, q_2 \):

\[
\begin{array}{c}
\text{a}_2 \quad \text{b}_1 \\
\text{a}_1 \quad \text{b}_2
\end{array}
\quad \leq \quad
\begin{array}{c}
\text{a}_2 \quad \text{b}_1 \\
\text{a}_1 \quad \text{b}_2
\end{array}
\]

Lemma 5.9 \( \preceq' \subseteq \preceq \).

Proof We show \( \preceq' \subseteq \preceq \) by induction on \( \preceq' \):

- Rule (smm). By (bmX) if \( \lambda_\varphi(m_1)=O \) and by (bXm) if \( \lambda_\varphi(m_1)=P=\lambda_\varphi(m_2) \).
- Rule (smm'). By (bmX) if \( \lambda_\varphi(m_1)=O \) and by (bXm) if \( \lambda_\varphi(m_1)=P=\lambda_\varphi(m_2) \).

In both cases the side-condition \( b_2 \notin \text{fa}(a_1 a_1) \) is valid. Next we show that \( \preceq \subseteq \preceq' \) by induction on \( \preceq \) and the length of \( e \):

- Rule (bmX). We use (smm) and (smm') to swap \( ma[b] \) with the leftmost move in \( e \). The condition \( b \notin \text{fa}(e) \) matches the distinctness condition \( b_2 \notin \{a, a_1, b_1\} \).
- Rule (bXm). We use (smm) and (smm') to swap \( ma[b] \) with the rightmost move in \( e \).

5.3 Innocence

An important notion in HO games is innocence [19, Definition 5.2], which characterises side-effect-free sequential computation. For us this is Definition 5.10 and with the tools we have built so far, it is quite compact:

Definition 5.10 Suppose \( m \) and \( m' \) are named moves (Definition 2.9). Given HO-legal plays \( emm', e'm \) in \( \mathfrak{A} \), where \( |emm'| \) is even length, \( ca(m') \cap \text{atoms}(e') = \emptyset \) and \( \text{r}emm\varphi = \text{r}e'm\varphi \), there is a unique renaming \( \rho = (c \mapsto c') \) with \( c \in \text{fa}(m') \) and \( c' \in ca(e') \) such that \( \text{r}emm\varphi \varphi = \text{r}e'm(\rho \cdot m') \varphi \). Call \( \sigma : \mathfrak{A} \) innocent when

\[
emmm', e' \in \sigma \wedge e'm \text{ HO-legal} \wedge \text{r}emm\varphi = \text{r}e'm\varphi \implies e'm(\rho \cdot m') \in \sigma.
\]

In [19, Definition 5.2] Hyland and Ong must write in English about the manipulation of pointers, and that this has to be done throughout their work (and this is typical of similar papers). We propose nominal techniques as a way to deal with this quickly and elegantly.

In HO games, moves can be repeated, which leads to a need to identify particular occurrences of moves in sequences. This goes away in our setting because every question or answer is uniquely identified by a name: the coabstractioned name that it introduces. So implicit in our framework is a separation of ‘move’ versus

\[\text{We use McCusker’s equivalent formulation [23, Subsection 2.2.4].}\]
‘occurrence’, removing a significant source of ambiguity. In this paper we have been able to implicitly identify an occurrence of a move in a proto-play with the named move in which the occurrence appears, since coabstracted atoms in proto-plays are distinct (Definition 2.8).

6 Conclusions

We have seen how pointer sequences can be modelled as nominal sequences. Pointers are split into a coabstraction \([b]\) corresponding to the head of the arrow, and a (free) atom \(b\) corresponding to its tail. Unlike pointers, a name carries its identity with it; \(b\) points to \([b]\) wherever we put it. Furthermore, unlike e.g. numbers, a name is permutatively symmetric, so reindexing / renaming can be expressed at a high level of abstraction. Because of this, nominal sequences are easy to break apart, compose, and reindex.

We have considered some non-trivial operations, like deletion and hereditarily justified subsequences; and some important definitions, like strategy composition, asynchronous reordering, and innocence. We have seen how these operations and definitions become straightforward and precise, if we choose the right machinery. This is attractive, but we also believe it will be almost a prerequisite for the kind of mechanised treatment of game semantics that is required for games to be applied in the second author’s research programme.

We have mentioned pointer sequences [19,14,23]. The Abramsky-Jagadeesan-Malacaria (AJM) games [2] rely on tags instead of pointers. These do not raise the problems of pointers and are fully formalised, but they are a more restricted formalism which was only used for PCF. For languages with effects the flexibility of pointers was required.

Another strategy is to become more abstract: so [5,17,24,21] revise the whole game semantic paradigm per se, in categorical terms. Some readers will instinctively believe that this categorical generalisation obsoletes any concrete realisation, but this is incorrect; there will always be a need for concrete models—especially if we want to implement or mechanise theorems. We seek convenient reformulations of the impressive collection of existing game models to make them more suitable for our intended applications. The work cited above is complementary, but also orthogonal.

Representations of pointer games [18] and games models of nominal languages [20] exist, including work by the second author with others on game semantics for nominal or nominal-related languages [15,1]. However, there has been no nominal representation of pointer sequences themselves. The closest the literature gets is in the Introduction to [24] where Melliès discusses representing pointers using integer indexes acted on by two group actions.

There is more to this paper than representing pointers. We use atoms in FM sets, which have structure that ZF sets do not. Functions, predicates, and subsets have symmetry (equivariance) properties and apartness (freshness) structure which make it relatively more convenient to handle distinctness conditions (like in Definition 2.8) or to deduce symmetry properties (as in Theorem 5.4), and so on (a very general
treatment is in [6, Section 5]).

In this paper, coabstraction is a syntactic token in sequences. We give a denotational intuition how this differs from nominal atoms-abstraction: suppose $X$ is a nominal set with an internal atoms-abstraction $[\lambda]X \to X$ written $[a]x$ (for definitions see [12,7]). Suppose $\mathcal{R} \subseteq X \times X$ is a relation on $X$. Then (briefly) $\mathcal{R}[\lambda]$ is the least relation such that if $x \mathcal{R} y$ and $a\#x, \mathcal{R}$ then $x \mathcal{R}[a]y$, and $\mathcal{R}[a]$ is the least relation such that if $x \mathcal{R} y$ and $a\#x, \mathcal{R}$ then $x \mathcal{R}[a]y$. This is coabstraction.

Nominal terms admit a similar generalisation; we would admit freshness $a\#X$ and cofreshness $a\%X$ conditions. More on this in a later paper.

We can read this paper as an exciting, if only partially articulated, commentary on semantics. The issue of dangling pointers and compositionality has not been properly addressed in the games literature and it remains to understand where the nominal model will take us. The nominal model of this paper exists in a larger context of nominal sets, substitution models, and some sophisticated logical and semantic theory [7,9], including abstract treatments of metavariables and renamings [8,11] and even e.g. trees with pointers [4]; the fruit of applying this theory, remains to be discovered.

Acknowledgement

The first author acknowledges the support of the Leverhulme trust.

References


A System-Level Semantics

Dan R. Ghica

University of Birmingham

Nikos Tzevelekos

Queen Mary, University of London

Abstract

Game semantics is a trace-like denotational semantics for programming languages where the notion of legal observable behaviour of a term is defined combinatorially, by means of rules of a game between the term (the Proponent) and its context (the Opponent). In general, the richer the computational features a language has the less constrained the rules of the semantic game. In this paper we consider the consequences of taking this relaxation of rules to the limit, by granting the Opponent omniscience, that is, permission to play any move without combinatorial restrictions. However, we impose an epistemic restriction by not granting Opponent omniscience, so that Proponent can have undisclosed secret moves. We introduce a basic C-like programming language and we define such a semantic model for it. We argue that the resulting semantics is an appealingly simple combination of operational and game semantics and we show how certain traces explain system-level attacks, i.e. plausible attacks that are realisable outside of the programming language itself. We also show how allowing Proponent to have secrets ensures that some desirable equivalences in the programming language are preserved.

1 Introduction

Game semantics came to prominence by solving the long-standing open problem of full abstraction for PCF [2,7] and it consolidated its status as a successful approach to modelling programming languages by being used in the definition of numerous other fully abstract programming language models. The approach of game semantics is to model computation as a formal interaction, called a game, between a term and its context. Thus, a semantic game features two players: a Proponent (P), representing the term, and an Opponent (O), representing the context. The interaction is formally described by sequences of game moves, called plays, and a term is modeled by a corresponding strategy, that is, the set of all its possible plays. To define a game semantics one needs to define what are the rules of the game and what are the abilities of the players.

For PCF games, the rules are particularly neat, corresponding to the so-called “principles of polite conversation”: moves are divided into questions and answers;

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
players must take turns; no question can be asked unless it is made possible (enabled) by an earlier relevant question; no answer can be given unless it is to the most recent unanswered question. The legality constraints for plays can be imposed as combinatorial conditions on sequences of moves.

Strategies also have combinatorial conditions which characterise the players rather than the game. They are uniformity conditions which stipulate that if in certain plays P makes a certain move, in other plays it will make an analogous move. The simplest condition is determinism, which stipulates that in any strategy if two plays are equal up to a certain P move, their subsequent P moves must also be the same. Relaxing some of the combinatorial constraints on plays and strategies elegantly leads from models of PCF to models of more expressive programming languages. For example, relaxing a condition called innocence leads to models of programming language with state [3], relaxing bracketing leads to models of programming languages with control [10], and in the absence of alternation we obtain languages for concurrency [6].

**Contribution.**

In this paper we consider the natural question of what happens if in a game semantics we remove combinatorial constraints from O’s behaviour. Unlike conventional game models, our construction is asymmetric: P behaves in a way determined by the programming language and its inherent limitations, whereas O can represent plausible behaviour which may not be, however, syntactically realizable neither in the language nor in some obvious extensions. We will see that such a model is, in a technical sense, well formed and that the notion of equivalence it induces is interesting and useful.

We study such a relaxed game model using an idealized type-free C-like language. The notion of available move is modeled using a notion of secret similar to that used in models of security protocols, formally represented using names. This leads to a notion of Opponent which is omnipotent but not omniscient: it can make any available move in any order, but some moves can be hidden from it. This is akin to the Dolev-Yao attacker model of security.

We show how inequivalences in this semantic model capture system-level attacks, i.e. behaviours of the ambient system which, although not realizable in the language itself, can be nevertheless plausibly enacted within the system. Despite the ambient system allowing surprising attacks, we note that many interesting equivalences still hold. This provides evidence that questions of semantic equivalence can be formulated outside the conventional framework of a syntactic context.

Technically, the model is expressed in an operationalised version of game semantics like Laird’s [12] and names are handled using nominal sets [5].
2 A system-level semantics

2.1 Syntax and operational semantics

We introduce a simple type-free C-like language which is just expressive enough to illustrate the basic concepts. A program is a list of modules, corresponding roughly to files in C. A module is a list of function or variable declarations. An exported variable or function name is globally visible, otherwise its scope is the module. In extended BNF-like notation we write:

\[
\begin{align*}
\text{Prog} &::= \text{Mod}^{*} \\
\text{Hdr} &::= \text{export } \overline{x}; \text{import } \overline{x}; \\
\text{Mod} &::= \text{Hdr} \text{ Dcl} \\
\text{Dcl} &::= \text{decl } x = n; \text{ Dcl} | \text{decl } \text{Func}; \text{ Dcl} | \epsilon
\end{align*}
\]

The header Hdr is a list of names exported and imported by the program, with \(x\) an identifier (or list of identifiers \(\overline{x}\)) taken from an infinite set \(\mathbb{N}\) of names, and \(n \in \mathbb{Z}\).

As in C, functions are available only in global scope and in uncurried form:

\[
\text{Func} ::= x(\overline{x})\{\text{local } \overline{x}; \text{ Stm return } \text{Exp;} \}
\]

A function has a name and a list of arguments. In the body of the function we have a list of local variable declarations followed by a list of statements terminated by a return statement. We define statements and expressions as follows (with \(n \in \mathbb{Z}\)).

\[
\begin{align*}
\text{Stm} &::= \epsilon | \text{if}(\text{Exp})\{\text{Stm}\}\text{else}\{\text{Stm}\}; \text{Stm} | \text{Exp}=\text{Exp}; \text{Stm} | \text{Exp}(\text{Exp}^{*}); \text{Stm} \\\n\text{Exp} &::= \text{Exp} \star \text{Exp} | \star \text{Exp} | \text{Exp}(\text{Exp}^{*}) | (\text{Exp}, \text{Exp}) | \text{new}(()) | n | x
\end{align*}
\]

Statements are branching, assignment and function call. For simplicity, iteration is not included as we allow recursive calls. Expressions are arithmetic and logical operators, variable dereferencing (\(\star\)), pairing, variable allocation and integer and variable constants. A function call can be either an expression or a statement. Because the language is type-free the distinction between statement and expression is arbitrary and only used for convenience.

If \(\text{decl } f(\overline{x})\{e\}\) is a declaration in module \(M\) we define \(f \circ M = e[\overline{x}],\) interpreted as “the definition of \(f\) in \(M\) is \(e\), with arguments \(\overline{x}\).”

A frame is given by the grammar below, with \(op \in \{=,\star,;\}, op' \in \{*,-\}\).

\[
t ::= \text{if}(\square)\{e\}\text{else}\{e\} | \square \text{op } e | v \text{op } \square | \text{op'} \square | \square e | v \square | (\square, e) | (v,\square)
\]

We denote the “hole” of the frame by \(\square\). We denote by \(\mathcal{F}s\) the set of lists of frames, the frame stacks. By \(v\) we denote values, defined below.

Our semantic setting is that of nominal sets [5] (see Appendix A), constructed over the multi-sorted set of names \(\mathcal{N} = \mathcal{N}_{\lambda} \uplus \mathcal{N}_{\phi} \uplus \mathcal{N}_{\kappa}\) where each of the three components is a countably infinite set of location names, function names and function continuation names respectively. That is, our objects can involve finitely many elements of \(\mathcal{N}\), and they come with a canonical notion of applying name permutations.
to them. For an object \( x \) and a permutation \( \pi \), the result of applying \( \pi \) to \( x \) is denoted by \( \pi \cdot x \). We range over names by \( a, b \), etc. Specifically for function names we may use \( f \), etc.; and for continuation names \( k \), etc. For each set of names \( \mathcal{X} \) we write \( \lambda(\mathcal{X}) \), \( \phi(\mathcal{X}) \) and \( \kappa(\mathcal{X}) \) for its restriction to location, function and continuation names respectively. For any object \( x \) involving names, we write \( \nu(x) \) for its support, i.e. the set of all the names occurring in it.

A store is defined as a pair of partial functions with finite domain: \( s \in \text{Sto} = (\mathcal{N}_\mathcal{A} \rightarrow \mathcal{F}_s) \times (\mathcal{N}_\mathcal{A} \rightarrow \mathcal{F}_s \times \mathcal{N}_\lambda) \). The first component of the store assigns integer values (data), other locations (pointers) or function names (pointers to functions) to locations. The second stores continuations, used by the system to resume a suspended function call.

We write \( \lambda(s) \), \( \kappa(s) \) for the two projections of a store \( s \). By abuse of notation, we may write \( s(a) \) instead of \( \lambda(s)(a) \) or \( \kappa(s)(a) \). Since names are sorted, this is unambiguous. The support \( \nu(s) \) of \( s \) is the set of names appearing in its domain or value set. For all stores \( s, s' \) and set of names \( \mathcal{X} \), we use the notations:

- \((\text{restrict-to})\) the sub-store of \( s \) defined on \( \mathcal{X} \):
  \[ s \upharpoonright \mathcal{X} = \{(a, y) \in s \mid a \in \mathcal{X}\}; \]

- \((\text{restrict-from})\) the sub-store of \( s \) that is not defined on \( \mathcal{X} \):
  \[ s \setminus \mathcal{X} = s \setminus (\text{dom}(s) \setminus \mathcal{X}); \]

- \((\text{update})\) change the values in \( s \):
  \[ s[a \mapsto x] = \{(a, x)\} \cup (s \setminus \{a\}); \]

  and, more generally:
  \[ s[s'] = s' \cup (s \setminus \text{dom}(s')); \]

- \((\text{extension})\) \( s \subseteq s' \) if \( \text{dom}(s) \subseteq \text{dom}(s') \);

- \((\text{closure})\) \( Cl(s, \mathcal{X}) \) is the least set of names containing \( \mathcal{X} \) and all names reachable from \( \mathcal{X} \) through \( s \) in a transitively closed manner, i.e. \( \mathcal{X} \subseteq Cl(s, \mathcal{X}) \) and if \( (a, y) \in s \) with \( a \in Cl(s, \mathcal{X}) \) then \( \nu(y) \in Cl(s, \mathcal{X}) \).

We define a value to be a name, an integer, or a tuple of values: \( v ::= () \mid a \mid n \mid (v, v) \). The value () is the unit for the tuple operation.\(^1\)

We give a semantics for the language using a frame-stack abstract machine. It is convenient to take identifiers to be names, as it gives a simple way to handle pointers to functions in a way much like that of the C language. The Program configurations of the abstract machine are of the form:

\[ (N \mid P \vdash s, t, e, k) \in \mathcal{N} \times \mathcal{N} \times \text{Sto} \times \mathcal{F}_s \times \mathcal{E}_p \times \mathcal{N}_\lambda \]

\( N \) is a set of used names; \( P \subseteq N \) is the set of public names; \( s \) is the program state; \( t \) is a list of frames called the frame stack; \( e \) is the expression being evaluated; and \( k \) is a continuation name, which for now will stay unchanged.

The transitions of the abstract machine are a relation on the set of configurations. They are defined by case analysis on the structure of \( e \) then \( t \) in a standard fashion, as in Fig. 1. Branching is as in C, identifying non-zero values with true and zero with false. Binary operators are evaluated left-to-right, also as in C. Arithmetic and logic operators (\( \ast \)) have the obvious evaluation. Dereferencing is given the usual evaluation, with a note that in order for the rule to apply it is implied that \( v \) is a location and \( s(v) \) is defined. Local-variable allocation extends the domain of \( s \)

\(^1\) Tupling is associative and for simplicity we identify tuples up to associativity and unit isomorphisms, so \((v, (v, v)) = ((v, v), v) = (v, v, v) \) and \((v, (v)) = v;\) etc.
Case $e = v$ is a value.

$$\langle N \mid P \vdash s, t \circ (\text{if } (\Box e) \text{ then } \{e_1\} \text{ else } \{e_2\}), v, k \rangle \rightarrow \langle N \mid P \vdash s, t, e_1, k \rangle,$$  
if $v \in \mathbb{Z} \setminus \{0\}$

$$\langle N \mid P \vdash s, t \circ (\text{if } (\Box e) \text{ then } \{e_1\} \text{ else } \{e_2\}), v, k \rangle \rightarrow \langle N \mid P \vdash s, t, e_2, k \rangle,$$  
if $v = 0$

$$\langle N \mid P \vdash s, t \circ (\text{if } (\Box e) \text{ then } \{e_1\} \text{ else } \{e_2\}), v, k \rangle \rightarrow \langle N \mid P \vdash s, t, (v \circ e), e, k \rangle$$
for $op \in \{=, \star, ;\}$

$$\langle N \mid P \vdash s, t \circ (v \star (e'), v', k) \rightarrow \langle N \mid P \vdash s, t, v'', k \rangle,$$  
and $v'' = v \star v'$

$$\langle N \mid P \vdash s, t \circ (a = \Box e), v, k \rangle \rightarrow \langle N \mid P \vdash s[a \rightarrow v], t, k \rangle$$

$$\langle N \mid P \vdash s, t \circ (s \circ (\Box e)), v, k \rangle \rightarrow \langle N \mid P \vdash s, t, s(v), k \rangle$$

$$\langle N \mid P \vdash s, t \circ (\Box e), \text{local } x, k \rangle \rightarrow \langle N \mid a \cup \{a \mid P \vdash s[a \rightarrow 0], t, e[a/x], k\}, a \not\in N$$

$$\langle N \mid P \vdash s, t \circ (\Box e), v, k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (v(\Box e)), e, k \rangle$$

$$\langle N \mid P \vdash s, t \circ ((v, e)), v, k \rangle \rightarrow \langle N \mid P \vdash s, t \circ ((v, \Box e)), e, k \rangle$$

$$\langle N \mid P \vdash s, t \circ (f(\Box e)), v', k \rangle \rightarrow \langle N \mid P \vdash s, t, e[v'/\Box], k \rangle,$$  
if $f \in M = e[\Box]$ (F)

Case $e$ is not a value.

$$\langle N \mid P \vdash s, t, \text{if } (e) \text{ then } \{e_1\} \text{ else } \{e_2\}, k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (\text{if } (\Box e) \text{ then } \{e_1\} \text{ else } \{e_2\}), e, k \rangle$$

$$\langle N \mid P \vdash s, t, \text{ if } e', k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (\Box e'), e, k \rangle,$$  
if $op \in \{=, \star, ;\}$

$$\langle N \mid P \vdash s, t, e, k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (\Box e), e, k \rangle$$

$$\langle N \mid P \vdash s, t, \text{return}(e), k \rangle \rightarrow \langle N \mid P \vdash s, t, e, k \rangle$$

$$\langle N \mid P \vdash s, t, \text{new}(e), k \rangle \rightarrow \langle N \cup \{a \mid P \vdash s[a \rightarrow 0], t, a, k\}, a \in \mathcal{A} \setminus N$$

$$\langle N \mid P \vdash s, t, e', k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (\Box (e')), e, k \rangle$$

$$\langle N \mid P \vdash s, t, (e', e'), k \rangle \rightarrow \langle N \mid P \vdash s, t \circ (\Box (e')), e, k \rangle$$

Fig. 1. Operational semantics

with a fresh secret name. Local variables are created fresh, locally for the scope of a function body. The new() operator allocates a secret and fresh location name, initialises it to zero and returns its location. The return statement is used as a syntactic marker for an end of function but it has no semantic role.

Structural rules, such as function application and tuples are as usual in call-by-value languages, i.e. left-to-right. Function call also has a standard evaluation. The body of the function replaces the function call and its formal arguments $\overline{x}$ are substituted by the tuple of arguments $\overline{v'}$ in point-wise fashion. Finally, non-canonical forms also have standard left-to-right evaluations.

2.2 System semantics

The conventional function-call rule (F) is only applicable if there is a function definition in the module. If the name used for the call is not the name of a known function then the normal operational semantics rules no longer apply. We now extend our semantics so that calls and returns of locally undefined functions become a mechanism for interaction between the program and the ambient system. We call the resulting semantics the System Level Semantics (SLS).

Given a module $M$ we will write as $[M]$ the transition system defining its SLS. Its states are $\mathcal{S}[M] = \mathit{Sys}[M] \cup \mathit{Prog}[M]$, where $\mathit{Prog}[M]$ is the set of abstract-machine configurations of the previous section and $\mathit{Sys}[M]$ is the set of system configurations, which are of the form: $\langle N \mid P \vdash s \rangle \in \mathcal{N} \times \mathcal{N} \times \mathit{Sto}$. The SLS is defined at the level of modules, that is programs with missing func-
tions, similarly to what is usually deemed a compilation unit in most programming languages. The transition relation $\delta[M]$ of the SLS operates on a set of labels $L \cup \{\epsilon\}$ and is of the following type.

$$\delta[M] \subseteq (\text{Prog}[M] \times \{\epsilon\} \times \text{Prog}[M]) \cup (\text{Prog}[M] \times L \times \text{Sys}[M])$$

$$\cup (\text{Sys}[M] \times L \times \text{Prog}[M])$$

$$L = \{(s, \text{call } f, v, k) \mid s \in \text{Sto}, \kappa(s) = \emptyset, f \in \mathcal{N}, k \in \mathcal{N}_\kappa, v \text{ a value}\}$$

$$\cup \{(s, \text{ret } v, k) \mid s \in \text{Sto}, \kappa(s) = \emptyset, k \in \mathcal{N}_\kappa, v \text{ a value}\}$$

Thus, at the system level, program and system configurations may call and return functions in alternation, in very much the same way that P and O make moves in game semantics. We write $X \xrightarrow{s} X'$ for $(X, (s, \alpha), X') \in \delta[M]$, and $X \rightarrow X'$ for $(X, \epsilon, X') \in \delta[M]$.

In transferring control between Program and System the continuation pointers ensure that upon return the right execution context can be recovered. We impose several hygiene conditions on how continuations can be used. We distinguish between P- and S-continuation names. The former are created by the Program and stored for subsequent use, when a function returns. The latter are created by the System and are not stored. The reason for this distinction is both technical and intuitive. Technically it will simplify proving that composition is well-defined. Mixing S and P continuations would not create any interesting behaviour: if P receives a continuation it does not know then the abstract machine of P cannot evaluate it, which can be interpreted as a crash. But S always has ample opportunities to crash the execution, so allowing it seems uninteresting. However, this is in some sense a design decision and an alternative semantics, with slightly different properties, can be allowed to mix S and P continuations in a promiscuous way.

The first new rule, called **Program-to-System call** is:

$$\langle N \mid P \vdash s, t \circ (f(\square)), v, k \rangle \xrightarrow{\text{call } f, v, k'} \langle N \cup \{k'\} \mid P' \cup \{k'\} \vdash s[k' \mapsto (t, k)]\rangle$$

if $f \not\in M$ not defined, $k' \notin N$, $P' = \text{Cl}(s, P \cup \nu(v))$

When a non-local function is called, control is transferred to the system. In game semantics this corresponds to a Proponent question, and is an observable action. Following it, all the names that can be transitively reached from public names in the store also become public, so it gives both control and information to the System. Its observability is marked by a label on the transition arrow, which includes: a tag call, indicating that a function is called, the name of the function ($f$), its arguments ($v$) and a fresh continuation ($k'$), which stores the code pointer; the transition also marks that part of the store which is observable because it uses publicly known names.

The counterpart rule is the **System-to-Program return**, corresponding to a return from a non-local function.
\[\langle\langle N \mid P \vdash s\rangle\rangle \xrightarrow{\text{ret}_{v,k'}} \langle\langle N \cup \nu(v, s') \mid P \cup \nu(v, s') \vdash s[s'], f, v, k\rangle\]

if \(s(k') = (f, k), \nu(v, s') \cap N \subseteq P, \lambda(\nu(v)) \subseteq \nu(s'), s \upharpoonright \lambda(P) \subseteq s'\)

This is akin to the game-semantic Opponent answer. Operationally it corresponds to S returning from a function. Note here that the only constraints on what S can do in this situation are epistemic, i.e. determined by what it knows:

(i) it can return with any value \(v\) so long as it only contains public names or fresh names (but not private ones);

(ii) it can update any public location with any value;

(iii) it can return to any (public) continuation \(k'\).

However, the part of the store which is private (i.e. with domain in \(N \setminus P\)) cannot be modified by S. So S has no restrictions over what it can do with known names and to known names, but it cannot guess private names. Therefore it cannot do anything with or to names it does not know. The restriction on the continuation are just hygienic, as explained earlier.

There are two converse transfer rules System-to-Program call and Program-to-System return, corresponding to the program returning and the system initiating a function call:

\[\langle\langle N \mid P \vdash s\rangle\rangle \xrightarrow{\text{call}_{f,v,k}} \langle\langle N \cup \{k\} \cup \nu(v, s') \mid P \cup \{k\} \cup \nu(v, s') \vdash s[s'], f, \Box, v, k\rangle\]

if \(f@M\) defined, \(k \notin \text{dom}(s), \nu(f, v, s') \cap N \subseteq P, \lambda(\nu(v)) \subseteq \nu(s'), s \upharpoonright \lambda(P) \subseteq s'\)

\[\langle\langle N \mid P \vdash s, \_, v, k\rangle\rangle \xrightarrow{\text{ret}_{v,k}} \langle\langle N \mid P' \vdash s\rangle\rangle \quad \text{where } P' = \text{Cl}(s, P \cup \nu(v))\]

In the case of the S-P call it is S which calls a publicly-named function from the module. As in the case of the return, the only constraint is that the function \(f\), arguments \(v\) and the state update \(s'\) only involve public or fresh names. The hygiene conditions on the continuations impose that no continuation names are stored, for reasons already explained. Finally, the P-S return represents the action of the program yielding a final result to the system following a function call. The names used in constructing the return value are disclosed and the public part of the store is observed. In analogy with game semantics the function return is a Proponent answer while the system call is an Opponent question.

The initial configuration of the SLS for module \(M\) is \(S^0_M = \langle\langle N_0 \mid P_0 \vdash s_0\rangle\rangle\). It contains a store \(s_0\) where all variables are initialised to the value specified in the declaration. The set \(N_0\) contains all the exported and imported names, all declared variables and functions. The set \(P_0\) contains all exported and imported names. When \(M\) is not clear from the context, we may write \(P^0_M\) for \(P_0\), etc.
3 Compositionality

The SLS of a module $M$ gives us an interpretation $\llbracket M \rrbracket$ which is modular and effective (i.e. it can be executed) so no consideration of the context is required in formulating properties of modules based on their SLS. Technically, we can reason about SLS using standard tools for transition systems such as trace equivalence, bisimulation or Hennessy-Milner logic.

We first show that the SLS is consistent by proving a compositionality property. SLS interpretations of modules can be composed semantically in a way that is consistent with syntactic composition. Syntactic composition for modules is concatenation with renaming of un-exported function and variable names to prevent clashes, which we will denote by using $-\cdot-$. In particular, we show that we can define a semantic SLS composition $\otimes$ so that, for an appropriate notion of isomorphism in the presence of $\tau$-transitions ($\cong_{\tau}$), the following holds.

For any modules $M, M'$: $\llbracket M \cdot M' \rrbracket \cong_{\tau} \llbracket M \rrbracket \otimes \llbracket M' \rrbracket$.

We call this the principle of functional composition.

Let $P$ range over program configurations, and $S$ over system configurations. Moreover, assume an extended set of continuation names $N' = N_k \uplus N_{aux}$, where $N_{aux}$ is a countably infinite set of fresh auxiliary names. We define semantic composition of modules inductively as in Fig. 2 (all rules have symmetric, omitted counterparts). We use an extra component $\Pi$ containing those names which have been communicated between either module and the outside system, and we use an auxiliary store $s$ containing values of locations only. The latter records the last known values of location names that are not private to a single module. Continuation names in each $\Pi$ are assigned Program/System polarities (we write $k \in \Pi_P$ / $k \in \Pi_S$), thus specifying whether a continuation name was introduced by either of the modules or from the outside system. Cross calls and returns are assigned $\tau$-labels and are marked by auxiliary continuation names. We also use the following notations for updates of $\Pi$ when an interaction with the outside system is made, where we write $Pr$ for the set of private names $\nu(S, S') \setminus \Pi$.

- $(\Pi, s')^P[v, k, s] = Cl(s'[s], \nu(v) \cup \Pi) \cup \{k\}$, and assign P polarity to $k$;
- $(\Pi, s')^S[v, k, s] = \Pi \cup \nu(v, s \setminus Pr) \cup \{k\}$, and assign S polarity to $k$.

The notations apply also to the case when no continuation name $k$ is included in the update (just disregard $k$). The semantic composition of modules $M$ and $M'$ is thus given by:

$$\llbracket M \rrbracket \otimes \llbracket M' \rrbracket = \llbracket M \rrbracket \otimes_{\Pi_0}^{s_0 \cup s'_0} \llbracket M' \rrbracket$$

where $s_0$ is the store assigning initial values to all initial public locations of $\llbracket M \rrbracket$, and similarly for $s'_0$, and $\Pi_0$ contains all exported and imported names.

The rules of Fig. 2 feature side-conditions on choice of continuation names, system stores and name privacy. The latter originate from nominal game se-

---

2 That is, auxiliary names are used precisely for cross calls and returns.

3 These stipulate (rules (vi)-(vii)) that the store produced in each outside system transition must: (a) be
mantics [1,11] and they guarantee that the names introduced (freshly) by $M$ and $M'$ do not overlap (rule (i)), and that the names introduced by the system in the composite module do not overlap with any of the names introduced by $M$ or $M'$ (rules (vi)-(vii)). They safeguard against incorrect name flow during composition.

Let us call the four participants in the composite SLS Program $A$, System $A$, Program $B$, System $B$. Whenever we use $X$, $Y$ as Program or System names they can be either $A$ or $B$, but different. Whenever we say Agent we mean Program or System. A state of the composite system is a pair (Agent $X$, Agent $Y$) noting that they cannot be both Programs. The composite transition rules reflect the following intuitions.

- Rule (i): If Program $X$ makes an internal (operational) transition System $Y$ is not affected.
- Rules (ii)-(iii): If Program $X$ makes a system transition to System $X$ and System $Y$ can match the transition going to Program $Y$ then the composite system makes an internal ($\tau$) transition. This is the most important rule and it is akin to game semantic composition via “synchronisation and hiding”. It signifies $M$ making a call (or return) to (from) a function present in $M'$.
- Rules (iv)-(v): If Program $X$ makes a system transition that cannot be matched by System $Y$ then it is a system transition in the composite system, a non-local call or return.

---

**Table 2. Rules for semantic composition**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>$P \xrightarrow{} P'$</td>
<td>Internal move $\nu(P') \subseteq \nu(P)$</td>
</tr>
<tr>
<td>(ii)</td>
<td>$P \xrightarrow{s} S \xrightarrow{v,k} S'$</td>
<td>Cross-call $k \in N_{aux} \setminus \nu(S)$</td>
</tr>
<tr>
<td>(iii)</td>
<td>$P \xrightarrow{s} S \xrightarrow{v,k} S'$</td>
<td>Cross-return $k \in N_{aux}$</td>
</tr>
<tr>
<td>(iv)</td>
<td>$P \xrightarrow{s} S \xrightarrow{v,k} S'$</td>
<td>Program call $\Pi' = (\Pi, s')^P[v,k]$ and $k \in N_{aux}$</td>
</tr>
<tr>
<td>(v)</td>
<td>$P \xrightarrow{s} S \xrightarrow{v,k} S'$</td>
<td>Program return $\Pi' = (\Pi, s')^P[v,k]$ and $k \in N_{aux}$</td>
</tr>
<tr>
<td>(vi)</td>
<td>$S \xrightarrow{s} S' \xrightarrow{v,k} P$</td>
<td>System call $k \in N_{aux} \setminus \nu(S'), \Pi' = (\Pi, s')^S[v,k,s]$, $\lambda(\Pi) \subseteq \text{dom}(s), s' \subseteq \nu(s)$ and $\nu(v, s, \text{Pr}) \cap \text{Pr} = \emptyset$</td>
</tr>
<tr>
<td>(vii)</td>
<td>$S \xrightarrow{s} S' \xrightarrow{v,k} P$</td>
<td>System return $k \in N_{aux} \setminus \nu(S'), \Pi' = (\Pi, s')^S[v,s]$, $\lambda(\Pi) \subseteq \text{dom}(s), s' \subseteq \nu(s)$ and $\nu(v, s, \text{Pr}) \cap \text{Pr} = \emptyset$</td>
</tr>
</tbody>
</table>

---

*defined on all public names (i.e. names in $H$), and (b) agree with the old one on all other names. The latter safeguards against the system breaking name privacy.*
- Rules (vi)-(vii): From a composite system configuration (both entities are in a system configuration) either Program X or Program Y can become active via a call or return from the system.

We can now formalise and prove functional composition (proof in Appendix B).

**Definition 3.1** Let \( \mathcal{G}_1, \mathcal{G}_2 \) be LTSs corresponding to a semantic composite SLS and an ordinary SLS respectively. A function \( R \) from states of \( \mathcal{G}_1 \) to configurations of \( \mathcal{G}_2 \) is called a \( \tau \)-isomorphism if it maps the initial state of \( \mathcal{G}_1 \) to the initial configuration of \( \mathcal{G}_2 \) and, moreover, for all states \( X \) of \( \mathcal{G}_1 \) and \( \ell \in \mathcal{L} \),

(i) if \( X \xrightarrow{\tau} X' \) then \( R(X) = R(X') \),
(ii) if \( X \rightarrow X' \) then \( R(X) \rightarrow R(X') \),
(iii) if \( R(X) \rightarrow Y \) and \( X \xrightarrow{\tau} \) then \( X \rightarrow X' \) with \( R(X') = Y \),
(iv) if \( X \xrightarrow{\ell} X' \) then \( R(X) \xrightarrow{\ell} R(X') \),
(v) if \( R(X) \xrightarrow{\ell} Y \) and \( X \xrightarrow{\tau} \) then \( X \xrightarrow{\ell} X' \) with \( R(X') = Y \).

We write \( \mathcal{G}_1 \simeq \tau \mathcal{G}_2 \) if there is a \( \tau \)-isomorphism \( R : \mathcal{G}_1 \rightarrow \mathcal{G}_2 \).

**Proposition 3.2** For all modules \( M, M' \), \( [M] \otimes [M'] \simeq [M \cdot M'] \).

### 4 Reasoning about SLS

The epistemically-constrained system-level semantics gives a security-flavoured semantics for the programming language which is reflected by its logical properties and by the notion of equivalence it gives rise to.

We will see that certain properties of traces in the SLS of a module correspond to “secrecy violations”, i.e. undesirable disclosures of names that are meant to stay secret. In such traces it is reasonable to refer to the System as an attacker and consider its actions an attack. We will see that although the attack cannot be realised within the given language it can be enacted in a realistic system by system-level actions.

We will also see that certain equivalences that are known to hold in conventional semantics still hold in a system-level model. This means that even in the presence of an omnipotent attacker, unconstrained by a prescribed set of language constructs, the epistemic restrictions can prevent certain observations, not only by the programming context but by any ambient computational system. This is a very powerful notion of equivalence which embodies tamper-resistance for a module.

Note that we chose these examples to illustrate the conceptual interest of the SLS-induced properties rather than as an illustration of the mathematical power of SLS-based reasoning techniques. For this reason, the examples are as simple and clear as possible.
4.1 A system-level attack: violating secrecy

This example is inspired by a flawed security protocol which is informally described as follows.

Consider a secret, a locally generated key and an item of data read from the environment. If the local key and the input data are equal then output the secret, otherwise output the local key.

In a conventional process-calculus syntax the protocol can be written as

$$\nu s \nu k. \text{in}(a). \text{if } k = a \text{ then } \text{out}(s) \text{ else } \text{out}(k).$$

It is true that the secret $s$ is not leaked because the local $k$ cannot be known as it is disclosed only at the very end. This can be proved using bisimulation-based techniques for anonymity. Let us consider an implementation of the protocol:

```cpp
export prot;
import read;
decl prot( ) {
  local s, k, x; s = new(); k = new(); x = read();
  if (*x == *k) then *s else *k}
```

We have local variables $s$ holding the “secret location” and $k$ holding the “private location”. We use the non-local, system-provided, function `read` to obtain a name from the system, which cannot be that stored at $s$ or $k$. A value is read into $x$ using untrusted system call `read()`. Can the secrecy of $s$ be violated by making the name stored into it public? Unlike in the process-calculus model, the answer is “yes”.

The initial configuration is $⟨⟨\text{prot, read | prot, read} \vdash \emptyset⟩⟩$. We denote the body of `prot` by $E$. The transition corresponding to the secret being leaked is shown in Fig. 3. The labelled transitions are the interactions between the program and the system and are interpreted as follows:

(i) system calls `prot()` giving continuation $k$
(ii) program calls `read()` giving fresh continuation $k'$
(iii) system returns (from `read`) using $k'$ and producing fresh name $a_2$
(iv) program returns (from `prot`) leaking local name $a_1$ stored in $k$
(v) system uses $k'$ to fake a second return from `read`, using the just-learned name $a_1$ as a return value
(vi) with $a_1$ the program now returns the secret $a_0$ stored in $s$ to the environment.

Values of $a_2$ are omitted as they do not affect the transitions.

The critical step is (v), where the system is using a continuation in a presumably illegal, or at least unexpected, way. This attack can be implemented in several ways:

- If the attacker has access to more expressive control commands such as `callcc` then the continuation can simply be replayed.
it embodies a principle of locality for state. equivalences we can show, but we will choose a simple but important one, because by proving that there are nontrivial equivalences which hold. There are many such of view. In this section we want to further emphasise this point. We can do that This already shows that our language is “well behaved” from a system-level point

Functional Compositionality gives an internal consistency check for the semantics. Above,

• The attacker’s ability to clone the configuration can lead to attacks which are continuation \( k’ \), i.e. the memory pointed at by the name \( k’ \). In order to execute the attack it is not required to have an understanding of the actual machine code or byte-code, as the continuation is treated as a black box. This means that the attack cannot be prevented by any techniques reliant on obfuscation of the instruction or address space, such as randomisation.

• If the attacker has access to fork-like concurrency primitives then it can exploit them for the attack because such primitives duplicate the thread of execution, creating copies of all memory segments. Note that the behaviour of the conventional Unix fork is richer than what we consider in our system model, but it can be readily accommodated in our framework by a configuration-cloning system transition:

\[
\langle\langle N | P \vdash s \rangle \rangle \rightarrow \langle\langle N + N | P + P \vdash s[N/inl(N)] \cup s[N/inr(N)] \rangle \rangle
\]

The attacker’s ability to clone the configuration can lead to attacks which are purely systemic, for example executing the program into a virtual machine, pausing execution, cloning the state of the machine, then playing the two copies against each other.

\[\langle\langle N | P \vdash 0 \rangle \rangle \rightarrow \langle\langle N_0, k | P_0, k \vdash 0, \ldots, E, k \rangle \rangle\]

\[\alpha \rightarrow^{*} \langle\langle N_1, k, a_0, a_1 | P_1 \vdash (a \mapsto a_0, k \mapsto a_1, x \mapsto 0), (\text{\textsc{if}}(x = \text{\textsc{in}}) \circ \text{\textsc{read}}), (\text{\textsc{if}}), (\text{\textsc{fork}}) \rangle \rangle\]

\[\alpha \rightarrow^{*} \langle\langle N_2 | P_1, a_2 \vdash (a \mapsto a_0, k \mapsto a_1, x \mapsto 0, k’ \mapsto (t, k)), t, a_2, k \rangle \rangle\]

\[\alpha \rightarrow^{*} \langle\langle N_2 | P_1, a_2 \vdash (a \mapsto a_0, k \mapsto a_1, x \mapsto a_2, k’ \mapsto (t, k)), a, k \rangle \rangle\]

\[\alpha \rightarrow^{*} \langle\langle N_2 | P_1, a_2, a_1 \vdash (a \mapsto a_0, k \mapsto a_1, x \mapsto a_2, k’ \mapsto (t, k)), a, k \rangle \rangle\]

\[\alpha \rightarrow^{*} \langle\langle N_2 | P_1, a_2, a_1, a_0 \vdash (a \mapsto a_0, k \mapsto a_1, x \mapsto a_2, k’ \mapsto (t, k)), a, k \rangle \rangle\]

Fig. 3. Secret \( a_0 \) leaks.

4.2 Equivalence

Functional Compositionality gives an internal consistency check for the semantics. This already shows that our language is “well behaved” from a system-level point of view. In this section we want to further emphasise this point. We can do that by proving that there are nontrivial equivalences which hold. There are many such equivalences we can show, but we will choose a simple but important one, because it embodies a principle of locality for state.

This deceptively simple example was first given in [13] and establishes the fact
that a local variable cannot be interfered with by a non-local function. This was an interesting example because it highlighted a significant shortcoming of global state models of imperative programming. Although not pointed out at the time, functor-category models of state developed roughly at the same time gave a mathematically clean solution for this equivalence, which followed directly from the type structure of the programming language [15].

We compare SLSs be examining their traces. Formally, the set of traces of module \( M \) is given by \( T(M) = \{ (\pi \cdot w) \in \mathcal{L}^* \mid S_0^M \xrightarrow{w} X, \forall a \in P_0^M, \pi(a) = a \} \)

**Definition 4.1** Let \( M_1, M_2 \) be modules with common public names. We say that \( M_1 \) and \( M_2 \) are trace equivalent, written \( M_1 \cong M_2 \), if \( T(M_1) = T(M_2) \).

The above extends to modules with \( P_0^{M_1} \neq P_0^{M_2} \) by explicitly filling in the missing public names on each side. We next introduce a handy notion of bisimilarity which precisely captures trace equivalence. For each configuration \( X \), let us write \( P(X) \) for the set of public names of \( X \).

**Definition 4.2** Let \( R \) be a relation between configurations. \( R \) is a simulation if, whenever \( (X_1, X_2) \in R \), we have \( P(X_1) = P(X_2) \) and also:

- \( X_1 \xrightarrow{L} X_1' \) implies \( (X_1', X_2) \in R \);
- \( X_1 \xrightarrow{\ell} X_2 \xrightarrow{X_2''} \) with \( (\pi \cdot X_2'') \xrightarrow{L} X_2' \) and \( (X_1', X_2') \in R \), for some name permutation \( \pi \) such that \( \pi(a) = a \) for all \( a \in P(X_1) \).

\( R \) is a bisimulation if it and its inverse are simulations. Modules \( M_1 \) and \( M_2 \) are bisimilar, written \( M_1 \sim M_2 \), if there is a bisimulation \( R \) such that \( (S_0^{M_1}, S_0^{M_2}) \in R \).

**Lemma 4.3** Bisimilarity coincides with trace equivalence.

**Proposition 4.4** Trace equivalence is a congruence for module composition \(- \cdot -\).

Intuitively, the reason is that in the first two programs \( f \)-local (module-local, respectively) variable \( x \) is never visible to non-local function \( g \), and will keep its initial value, which it 0. The bisimulation relation is straightforward as the three LTSs are equal modulo silent transitions and permutation of private names for \( x \).

Other equivalences, for example in the style of parametricity [14] also hold, with simple proofs of equivalence via bisimulation:

```plaintext
export inc, get;
edcl x;
edcl inc() {x=(*x+1)%3;}
edcl get() {return *x;}
```

4 Note: the set of traces is orbited through in order to factor out the choice of initial private names.
These two programs, or rather libraries, implement a modulo-3 counter as an abstract data structure, using private hidden state $x$. The environment can increment the counter (inc) or read its value (get) but nothing else. The first implementation counts up, and the second counts down.

5 Conclusion

In this paper we have developed a relaxed notion of game semantics in which the behaviour of the Opponent is defined by epistemic rather than combinatorial constraints. This has led us to two conclusions which we considered important.

First, we want to re-emphasise the fact that operational semantics can be extended in a relatively straightforward way from handling programs to handling terms, without relying on translation or interpretation. This is an idea already implicit in techniques such as trace semantics [8] or environmental bisimulation [9]. In the process, operational semantics becomes compositional. An LTS denotational interpretation of terms emerges automatically, without losing its effective presentation. Unlike previous work, however, we do not treat this extension of the operational semantics as a means to an end, e.g. studying contextual equivalence, but we treat it as important in its own right.

Secondly, and most importantly, we want to show that a meaningful and useful notion of context for the execution of terms can be constructed outside the syntax of the language. This has several advantages. The first one is modularity, as we can define the language and the environment in which its terms operate independently; the principle of functional composition is the consistency check that we need to satisfy for the two to be able to work together. The second one is realism, as real-life software is syntactically heterogeneous; through mechanism such as separate compilation and foreign-function interface it may have components written in different languages. Such programs cannot be characterised by the usual notion of syntactic context. The third one is simplicity, as we show how it is possible to formulate restrictions on the environment in a way which is not computational but epistemic, resembling the established Dolev-Yao characterisation of context in security. We believe this has the potential to offer a semantic foundation for the study of security properties of programs (such as information flow or tamper-proof compilation) in a way which is less dependent on syntax and more modular.

Relevant related work with similar aims but different philosophy has been carried out in compositional compiler correctness [4]. Whereas our point of view is mainly analytic, being interested in characterising arbitrary (if not unrestricted) environments and examine operationally the behaviour of open terms in such environments, compositional compiler correctness is a primarily a synthetic concern, aiming at defining constraints on machine code which allow safe composition between code generated via compilation with code generated in arbitrary ways. We see these two approaches as two sides of the same problem and we believe a better understanding of the relation between them should be studied.
A Nominal Sets

It is handy to introduce here some basic notions from the theory of nominal sets [5]. We call nominal structure any structure which may contain names, i.e., elements of \( \mathcal{N} \), and we denote by \( \text{Perm} \) the set of finite permutations on \( \mathcal{N} \) which are sort-preserving (i.e., if \( a \in \mathcal{N}_\lambda \) then \( \pi(a) \in \mathcal{N}_\lambda \), etc.). We range over permutations by \( \pi \) and variants. Finiteness means that each set \( \{ a \in \mathcal{N} \mid \pi(a) \neq a \} \) is finite. For example, \( \text{id} = \{ (a, a) \mid a \in \mathcal{N} \} \) is the identity permutation. On the other hand, \( \{(a, b), (b, a)\} \cup \{(c, c) \mid c \neq a, b\} \) is the permutation which swaps \( a \) and \( b \) and fixes all other names, for all \( a, b \) of the same sort.

For each set \( X \) of nominal structures of interest, we define a function \( \cdot : \text{Perm} \times X \to X \) such that \( \pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x \) and \( \text{id} \cdot x = x \), for all \( x \in X \) and \( \pi, \pi' \in \text{Perm} \). \( X \) is called a nominal set if all its elements involve finitely many names, that is, for all \( x \in X \) there is a finite set \( S \subseteq \mathcal{N} \) such that \( \pi \cdot x = x \) whenever \( \forall a \in S, \pi(a) = a \). The minimal such set \( S \) is called the support of \( x \) and denoted by \( \nu(x) \). For example, \( \mathcal{N} \) is a nominal set with action \( \pi \cdot a = \pi(a) \), and so is \( \mathcal{P}_\text{fin}(\mathcal{N}) \) with action \( \pi \cdot S = \{ \pi(a) \mid a \in S \} \).

Also, any set of non-nominal structures is a nominal set with trivial action \( \pi \cdot x = x \). More interestingly, if \( X, Y \) are nominal sets then so is \( X \times Y \) with

\[ (\pi, \pi') \cdot (x, y) = (\pi x, \pi' y) \]
action \( \pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y) \). This extends to arbitrary products and to strings. Finally, if \( X, Y \) are nominal sets then so is the set \( X \rightarrow_f Y \) with action \( \pi \cdot f = \{(\pi \cdot x, \pi \cdot y) \mid (x, y) \in f\} \).

### B Functional composition

We start with a lemma which stems from the definitions. We write \( \kappa'(\_\_\_\_) \) for the projection on \( \mathcal{N}'_\kappa = \mathcal{N}_\kappa \oplus \mathcal{N}_\text{aux} \), and \( \kappa(\_\_\_\_) \) for the projection on \( \mathcal{N}_\kappa \).

**Lemma B.1** Let \( X_1 \otimes^s_H X_2 \) be a state in the transition graph of \([M] \otimes [M']\) that is reachable from the initial state. Then, if each \( X_i \) includes the triple \( (N_i, P_i, s_i) \), the following conditions hold.

- \( (N_1 \setminus P_1) \cap N_2 = N_1 \cap (N_2 \setminus P_2) = \emptyset \), \( P_1 \setminus \Pi = P_2 \setminus \Pi \), \( \Pi \subseteq \nu(s) \cup \kappa(P_1 \cup P_2) \subseteq P_1 \cup P_2 \) and \( \text{dom}(s) = \lambda(P_1 \cup P_2) \).
- \( \text{dom}(\kappa'(s_1)) \cap \text{dom}(\kappa'(s_2)) = \emptyset \), \( (\text{dom}(\kappa'(s_1)) \cup \text{dom}(\kappa'(s_2))) \cap \Pi_S = \emptyset \) and \( \kappa'(P_1 \cap P_2) \setminus \Pi_S = \kappa'(P_1 \cup P_2) \setminus \Pi \subseteq \mathcal{N}_\text{aux} \).
- If both \( X_1, X_2 \) are system configurations and \( X_1 \otimes^s_H X_2 \) is preceded by a state of the form \( \mathcal{P} \otimes^s_H S \) then \( s \upharpoonright P_1 \subseteq s_1 \) and \( s \upharpoonright (P_2 \setminus P_1) \subseteq s_2 \), and dually if preceded by \( S \otimes^s_H \mathcal{P} \). Thus, in both cases, \( s \upharpoonright (P_1 \cap (P_1 \cap P_2)) \subseteq s_i \) for \( i = 1, 2 \).
- Not both \( X_1, X_2 \) are program configurations. If \( X_i \) is a program configuration then \( s \upharpoonright (P_{3-i} \setminus P_i) \subseteq s_{3-i} \).

Semantic composition introduces a notion of private names: internal continuation names passed around between the two modules in order to synchronise their mutual function calls. As the previous lemma shows, these names remain private throughout the computation. Therefore, in checking bisimilarity for such reduction systems, special care has to be taken so that these private names cannot be captured by external system transitions. This is achieved by selecting (only) these names from the auxiliary set \( \mathcal{N}_\text{aux} \).

We define the following translation \( R \) from reachable states of \([M] \otimes [M']\) to configurations of \([M \cdot M']\).

\[
\begin{align*}
\langle N_1 \mid P_1 \vdash s_1 \rangle \otimes^s_H \langle N_2 \mid P_2 \vdash s_2 \rangle & \mapsto \langle (N_1 \cup N_2) \setminus \mathcal{N}_\text{aux} \mid \Pi \vdash (\hat{s}_1[s'] \cup \hat{s}_2[s']) \setminus \mathcal{N}_\text{aux} \rangle \\
\langle N_1 \mid P_1 \vdash s_1 \rangle \otimes^s_H \langle N_2 \mid P_2 \vdash s_2, t, v, k \rangle & \mapsto \langle (N_1 \cup N_2) \setminus \mathcal{N}_\text{aux} \mid \Pi \vdash \hat{s}_1[\hat{s}_2] \setminus \mathcal{N}_\text{aux}, t', v, k' \rangle \\
\langle N_1 \mid P_1 \vdash s_1, t, v, k \rangle \otimes^s_H \langle N_2 \mid P_2 \vdash s_2 \rangle & \mapsto \langle (N_1 \cup N_2) \setminus \mathcal{N}_\text{aux} \mid \Pi \vdash \hat{s}_2[\hat{s}_1] \setminus \mathcal{N}_\text{aux}, t', v, k' \rangle
\end{align*}
\]

where \( s' = s \upharpoonright (P_1 \cap P_2) \), \( \hat{s}_i = s_i[k \mapsto (s_1, s_2)\downarrow_i(s_i(n))] \) for all \( k \in \text{dom}(\kappa(s_i)) \), and \( (t', k') = (s_1, s_2)\downarrow(t, k) \). The function \( (s_1, s_2)\downarrow \) fetches the full external frame stack and the external continuation searching back from \((t, k)\), that is:

\[
(s_1, s_2)\downarrow(t, k) = \begin{cases} (t, k) & \text{if } k \notin \mathcal{N}_\text{aux} \\ (s_1, s_2)\downarrow(t \circ t', k') & \text{if } k \in \mathcal{N}_\text{aux} \text{ and } s_i(k) = (t', k') \end{cases}
\]

Thus, the translation merges names from the component configurations and deletes
the names in $\mathcal{N}_{\text{aux}}$: these private names do not appear in $[M \cdot M']$, as there the corresponding function calls happen without using the call-return mechanism. Note that $[M \cdot M']$ is defined over the original $\mathcal{N}$, so it cannot capture any $k \in \mathcal{N}_{\text{aux}}$. The translation also sets $\Pi$ as the set of public names. Moreover, the total store is computed as follows. In system configurations we just take the union of the component stores and update them with the values of $s$, which contains the current values of all common public names. In program configurations we use the fact that the P-component contains more recent values than those of the S-component.

**Proposition B.2** For $R$ defined as above and $X_1 \otimes_{II} X_2$ a reachable configuration,

1. if $X_1 \otimes_{II} X_2 \xrightarrow{s} X'_1 \otimes_{II} X'_2$ then $R(X_1 \otimes_{II} X_2) = R(X'_1 \otimes_{II} X'_2)$,
2. if $X_1 \otimes_{II} X_2 \xrightarrow{s} X'_1 \otimes_{II} X'_2$ then $R(X_1 \otimes_{II} X_2) = R(X'_1 \otimes_{II} X'_2)$,
3. if $R(X_1 \otimes_{II} X_2) \xrightarrow{a} X$ and $Y \xrightarrow{a} X_1 \otimes_{II} X_2$ then $X_1 \otimes_{II} X_2 \xrightarrow{a} X'_1 \otimes_{II} X'_2$ with $Y = R(X'_1 \otimes_{II} X'_2)$,
4. if $X_1 \otimes_{II} X_2 \xrightarrow{s'} X'_1 \otimes_{II} X'_2$ then $R(X_1 \otimes_{II} X_2) \xrightarrow{s'} R(X'_1 \otimes_{II} X'_2)$,
5. if $R(X_1 \otimes_{II} X_2) \xrightarrow{s'} X$ and $Y \xrightarrow{s'} X_1 \otimes_{II} X_2$ then $X_1 \otimes_{II} X_2 \xrightarrow{s'} X'_1 \otimes_{II} X'_2$ with $Y = R(X'_1 \otimes_{II} X'_2)$.

**Proof.** For 1, let $X_1 = \langle N_1 \mid P_1 \vdash s_1, t \circ f(\square), v, k \rangle$, $X_2 = \langle N_2 \mid P_2 \vdash s_2 \rangle$ and the $\tau$-transition being due to an internal transition with label $(s_1, \text{call } f, v, k')$. Thus, $X'_1 = \langle N'_1 \mid P'_1 \vdash s'_1 \rangle$, $X'_2 = \langle N'_2 \mid P'_2 \vdash s'_2, f(\square), v, k' \rangle$, and so $R(X_1 \otimes_{II} X_2) = (N_0 \mid \Pi \vdash s_0, t, v, k_0)$ and $R(X'_1 \otimes_{II} X'_2) = (N'_0 \mid \Pi \vdash s'_0, t', v, k'_0)$. Note that $k' \in \mathcal{N}_{\text{aux}}$. Moreover, $s'_1 = s_1[k' \mapsto (t_1, k)]$ and $s'_2 = s \cup (s_2 \setminus P_2)$, so $(t'_0, k'_0) = (s'_1, s'_2)(f(\square), k') = (s_1, s_2)(t_0, k)$. Moreover, $N_0 = (N_1 \cup N_2) \setminus \mathcal{N}_{\text{aux}}$ and $N'_0 = (N'_1 \cup N'_2) \setminus \mathcal{N}_{\text{aux}} = (N_1 \cup \{k'\} \cup N_2) \setminus \mathcal{N}_{\text{aux}}$. As $\nu(v, s_i) \subseteq N_1$ and $k' \in \mathcal{N}_{\text{aux}}$, we get $N_0 = N'_0$. Finally, $s_0 = \hat{s}_2[s_1] \setminus \mathcal{N}_{\text{aux}}$ and $s'_0 = \hat{s}_2[s'_1] \setminus \mathcal{N}_{\text{aux}}$. Thus, $s'_0 = s'_1 \setminus \mathcal{N}_{\text{aux}} = \hat{s}_1[s' \cup (s_2 \setminus \lambda(P_2))] \setminus \mathcal{N}_{\text{aux}}$. Moreover, $s' = s_1 \setminus \lambda(P'_1)$ so $s'_0 = s_1 \setminus \lambda(P'_2) \setminus \mathcal{N}_{\text{aux}}$. But now note that $\text{dom}(s_2) \setminus \lambda(P_2) \cap \text{dom}(s_1) = \emptyset$: by the previous lemma, $\text{dom}(s_1)$ and $\text{dom}(s_2)$ share no continuation names, and if $a$ is a location name in $\text{dom}(s_2) \setminus P_2$ then $a \notin N_1$. Thus, $s_0 = s'_0$. Similarly if the $\tau$-transition is due to an internal return.

Item 2 is straightforward. For 3, the only interesting issue is establishing that if $X_1 \otimes_{II} X_2$ is in such a form that a $\tau$-transition needs to take place then the latter is possible. This follows directly from the definitions and the conditions of the previous lemma. In the following cases we consider call transitions; cases with return transitions are treated in a similar manner.

For 4, let $X_1 = \langle N_1 \mid P_1 \vdash s_1 \rangle$, $X_2 = \langle N_2 \mid P_2 \vdash s_2 \rangle$, $\alpha = (s', \text{call } f, v, k)$ and suppose the transition is due to $X_1$ reducing to $X'_1 = \langle N'_1 \mid P'_1 \vdash s'_1, f(\square), v, k \rangle$ with label $(s_i, \text{call } f, v, k)$. We have $\Pi' = \Pi \cup \nu(v, k_i \setminus P_r)$, $P_r = (N_1 \cup N_2) \setminus \Pi$, $s' = s_i \setminus \Pi'$ and $\nu(v, s_i \setminus P_r) \cap P_r = \emptyset$. Let $R(X_1 \otimes_{II} X_2) = \langle N_0 \mid \Pi \vdash s_0 \rangle$. As $k \notin \text{dom}(s_1)$ and $k \notin \nu(X_2) \setminus \Pi_S$, by previous lemma we obtain $k \notin \text{dom}(s_0)$, so the latter reduces to $\langle N'_0 \mid P \vdash s'_0, f(\square), v, k \rangle$ with transition $(s''', \text{call } f, v, k)$, for any appropriate $s'''$. In fact, if $\nu(v, s') \cap N_0 \subseteq \Pi$ then we can choose $s''' = s'$. Indeed,
\((\nu(v, s') \cap N_0) \setminus \Pi \subseteq \nu(v, s') \cap (N_0 \setminus \Pi) \subseteq \nu(v, s') \cap Pr = \nu(v, s_i \upharpoonright \Pi') \cap Pr = \nu(v, s_i \setminus Pr) \cap Pr = \emptyset\). Let \(R(X'_1 \otimes^{
u,f}_I X_2) = \langle N''_0 | \Pi' \vdash s''_0 \rangle\). We can see that \(N'_0 = N''_0\). Also, \(P = \Pi \cup \{k\} \cup \nu(v, s')\) while \(\Pi' = \Pi \cup \nu(v, k, s_i \setminus Pr) = \Pi \cup \nu(v, k, s')\).

Moreover, \(s'_0 = s' \cup (s_0 \setminus \lambda(\Pi)) = s' \cup (\tilde{s}_1[s_{12}] \cup \tilde{s}_2[s_{12}] \setminus (\mathcal{N}_{aux} \cup \lambda(\Pi)))\) with \(s_{12} = s \upharpoonright (P_1 \cap P_2)\), and \(s''_0 = \tilde{s}_2[s'_1] \setminus \mathcal{N}_{aux} = \tilde{s}_2[s_i \setminus \lambda(\Pi)] \setminus \mathcal{N}_{aux}\). Moreover, \(s'_0\) and \(s''_0\) agree on the domain of \(s'\) and on continuation names. Also, if location name \(a \in N'_0 \setminus N_0\) then \(a \in \nu(v, s')\) and thus \(a \in \text{dom}(s')\). Thus, we need to show that \(s'_0, s''_0\) agree on location names \(a\) from \(N_0 \setminus \Pi\). If \(a \in N_1 \setminus P_1\) then \(s'_0(a) = s_1(a) = s''_0(a)\), and similarly if in \(N_2 \setminus P_2\) using the fact that \((N_2 \setminus P_2) \cap N_1 = \emptyset\). Finally, if \(a \in P_1 \setminus P_2 \setminus \Pi\) then \(s'_0(a) = s(a) = s_1(a) = s''_0(a)\), by restrictions on \(s_i\).

Now let \(X_1 = \langle N_1 \mid P_1 \vdash s_1, to f(\square), v, k \rangle, X_2 = \langle N_2 \mid P_2 \vdash s_2 \rangle, \alpha = \text{call } f, v, k'\) and suppose the transition is due to \(X_1\) reducing to \(X'_1 = \langle N'_0 \mid \Pi \vdash s'_1 \rangle\) with label \((s_i, \text{call } f, v, k')\). We have \((\Pi', s'') = (\Pi, s)[v, s_1]\) and \(s' = s'' \setminus \Pi'\). We can assume, by definition, that \((s_1, s_2)[k'](t \circ f(\square), k) = (t_0 \circ f(\square), k_0)\), so \(R(X_1 \otimes^s_I X_2) = \langle N'_0 \mid \Pi \vdash s_0, t_0 \circ f(\square), v, k_0 \rangle\). As \(f\) is not defined in either of the modules and \(k'\) is completely fresh, the latter reduces to \(\langle N'_0 \mid P \vdash s_0 \rangle\) with transition \((s'', \text{call } f, v, k')\). Let \(R(X'_1 \otimes^{
u,f}_I X_2) = \langle N''_0 \mid \Pi' \vdash s''_0 \rangle\). It is easy to see that \(N'_0 = N''_0\). Moreover, \(s'_0 = s_0[k' \mapsto (t_0, k_0)]\) and \(s''_0 = (s'_1[s'_{12}] \cup s_2[s'_{12}]) \setminus \mathcal{N}_{aux}\) where \(s'_{12} = s'' \upharpoonright (P'_1 \cap P_2)\). Note that \(s'_0(k') = s'_1(k') = (s'_1, s_2)[t, k] = (t_0, k_0)\). Also, \(s'_0\) and \(s''_0\) agree on all other continuation names. Thus, in order to establish that \(s'_0 = s''_0\), it suffices to show that \(s_2[s_1]\) and \(s'_1[s'_{12}] \cup s_2[s'_{12}]\) agree on locations. From the previous lemma, \(s''_0\) agrees with \(s_1\) on locations in \(P'_1\) and with \(s_2\) on locations in \(P_2 \setminus P'_1\), and so \(s''_0 \subseteq s_2[s_1]\). Thus, \(\lambda(s'_1[s'_{12}] \cup s_2[s'_{12}]) = \lambda(s_1 \cup (s_2 \setminus P'_1)) = \lambda(s_2[s_1])\).

For public names, we have \(P = \text{Cl}(s_0, \Pi \cup \nu(v) \cup \{k'\}) = \text{Cl}(s'_0, \Pi \cup \nu(v, k'))\) while \(\Pi' = \text{Cl}(s'', \Pi \cup \nu(v, k'))\). As \(\kappa(P) = \kappa(\Pi)\setminus \{k'\}\), we can focus on location names. We have \(s''_0 \subseteq s'_0\) and, moreover, \(\text{dom}(s'') \supseteq (P'_1 \cup P_2) \supseteq (\Pi \cup \nu(v, k'))\), thus \(P = \Pi'\). Finally, \(s' = s''\) follows from the fact that these are restrictions of the final stores to the final sets of public location names.

For 5, let \(X_1 = \langle N_1 \mid P_1 \vdash s_1 \rangle, X_2 = \langle N_2 \mid P_2 \vdash s_2 \rangle, R(X_1 \otimes^f_I X_2) = \langle N_0 \mid \Pi \vdash s_0 \rangle\) and \(\alpha = (s', \text{call } f, v, k)\). We have that \(f\) is defined in \(M \cdot M'\) so WLOG assume that it is defined in \(M\). Then, \(X_1\) reduces to \(X'_1 = \langle N'_1 \mid P'_1 \vdash s'_1, to f(\square), v, k \rangle\) with \((s_i, \text{call } f, v, k), s_i = s' \setminus (s \setminus \Pi)\), if the relevant conditions for S-P calls are satisfied.

If \(k \in \text{dom}(s_1)\) then, by lemma, \(k \notin \Pi_S\). By assumption, \(k \in \Pi\) so \(k \notin \kappa(P_1 \cup P_2) \setminus \Pi\) and thus, by lemma, \(k \notin \kappa(P_1 \cap P_2) \setminus \Pi_S\), so \(k \notin P_2\). But the latter would imply \(k \in \text{dom}(s_1)\), which is disallowed by definition. Thus, \(k \notin \text{dom}(s_1)\).

Moreover, if \(a \in \nu(v, s_i) \cap (N_1 \setminus P_1) = \nu(v, s') \cap (N_1 \setminus P_1)\) then \(a \in \nu(v, s') \cap (N_0 \setminus P_1)\) and \(a \notin P_2\), so \(a \in \nu(v, s') \cap (N_0 \setminus (P_1 \cup P_2)) \subseteq \nu(v, s') \cap (N_0 \setminus \Pi)\), thus contradicting the conditions for the transition \(\alpha\). We still need to check that \(s_1 \upharpoonright \lambda(\Pi) \subseteq s_1 = s' \setminus (s \setminus \Pi)\). Given that \(s_0 \upharpoonright \lambda(\Pi) = \lambda(s_1[s_{12}] \cup s_2[s_{12}]) \cap \lambda(\Pi) \subseteq s'\), the condition follows from the previous lemma. We therefore obtain a transition from \(X_1 \otimes^f_I X_2\) to \(X'_1 \otimes^f_I X_2\); the relevant side-conditions are shown to be satisfied similarly as above. Finally, working as in 4, we obtain \(R(X'_1 \otimes^f_I X_2) = \emptyset\) and \(s' = s''\).

Now let \(X_1 = \langle N_1 \mid P_1 \vdash s_1, to f(\square), v, k \rangle, X_2 = \langle N_2 \mid P_2 \vdash s_2 \rangle, R(X_1 \otimes^f_I X_2) = \emptyset\), \(\alpha = (s', \text{call } f, v, k)\), and \(\beta = (s'', \text{call } f, v, k)\).
We first define a notion of accepted traces for arbitrary configurations by setting
\[ T(X) = \{ (\pi \cdot w) \in L^* \mid X \xrightarrow{w} X', \forall a \in P(X). \pi(a) = a \}, \]
where \( P(X) \) is the set of public names of \( X \). Note that \( w \in T(X) \) implies \( \pi \cdot w \in T(X) \) for any \( \pi \) that fixes all names in \( P(X) \) and, moreover, since \( X \xrightarrow{\ell/\epsilon} X' \implies (\pi \cdot X) \xrightarrow{\pi\ell/\epsilon} (\pi \cdot X') \), that
\[ T(X) = \{ w \in L^* \mid (\pi \cdot X) \xrightarrow{w} X', \forall a \in P(X). \pi(a) = a \}. \]

**Lemma C.1** Let \( M_1, M_2 \) be modules with common initial public names. Then, \( M_1 \sim M_2 \iff M_1 \cong M_2. \)

**Proof.** (\( \Rightarrow \)) Let \( \mathcal{R} \) be a bisimulation witnessing \( M_1 \sim M_2 \). For each transition sequence
\[ S^0_{M_1} \rightarrow X_1 \xrightarrow{\ell_1} X'_1 \rightarrow X_2 \xrightarrow{\ell_2} X'_2 \rightarrow \cdots \rightarrow X_n \xrightarrow{\ell_n} X'_n, \]
\( \mathcal{R} \) yields a diagram as below,

![Diagram](image_url)

and hence filling in the gaps we obtain:
\[ \pi^n \cdot S^0_{M_2} \rightarrow \pi^n \cdot Y_1 \xrightarrow{\pi^n \cdot \ell_1} \pi^n \cdot Y'_1 \rightarrow \pi^n \cdot Y_2 \xrightarrow{\pi^n \cdot \ell_2} \pi^n \cdot Y'_2 \rightarrow \cdots \rightarrow \pi^n \cdot Y_n \xrightarrow{\pi^n \cdot \ell_n} \pi^n \cdot Y'_n, \]
where \( \pi^n_i = \pi_n \circ \cdots \circ \pi_i \). By definition, each \( \pi_i \) fixes all names in \( P(X_i) \) so, in particular, \( \pi_i \cdot \ell_j = \ell_j \) for all \( j < i \), thus \( \pi^n_i \cdot S^0_{M_2} \xrightarrow{\ell_1 \ldots \ell_n} Y'_n \).

Hence, \( w \in T(M_1) \) implies \( w \in T(M_2) \). The other inclusion is shown similarly.

Conversely, suppose \( T(M_1) = T(M_2) \) and let us define the relation:
\[ \mathcal{R} = \{(X_1, X_2) \mid S^0_{M_i} \xrightarrow{w_i} X_i, P(X_1) = P(X_2), T(X_1) = T(X_2)\} \]

We claim that \( \mathcal{R} \) is a bisimulation. By symmetry, it suffices to show it is a simulation. Observe that all related configurations have common public names and, moreover, that \( \mathcal{R} \) is closed under \( \epsilon \)-transitions (by determinacy of internal transi-
tions up to choice of fresh names). Now let \((X_1, X_2) \in \mathcal{R}\) with \(X_1 \xrightarrow{\ell} X'_1\). Since \(\ell \in T(X_1) = T(X_2)\), there is a configuration \(X''_2\) and a permutation \(\pi\) fixing all elements of \(P = P(X_1) = P(X_2) = P(X''_2)\) such that \(\pi \cdot X_2 \xrightarrow{\ell} X_2\). Thus, \(X_2 \xrightarrow{\pi^{-1} \cdot X''_2} \pi \cdot X''_2\) and \(\pi \cdot (\pi^{-1} \cdot X''_2) \xrightarrow{\ell} X'_2\). Also, \(P' = P(X'_1) = P(X'_2) = P \cup \nu(\ell)\).

Take now any \(X'_1 \xrightarrow{\nu} X''_1\). We have \(\ell \nu \in T(X_1) = T(X_2)\), so, for some \(\pi''\) fixing all names in \(P, \pi'' \cdot X_2 \xrightarrow{\ell} \tilde{X}_2 \xrightarrow{w} \tilde{X}_2\). In particular, \(X_2 \xrightarrow{\pi''^{-1} \cdot \tilde{X}_2'}\) and hence, since internal transitions are up to choice of fresh (private) names, \(X''_2 = \pi \cdot \tilde{X}_2\) for some \(\pi\) fixing all names in \(P\). We thus obtain:

\[
X''_2 \xrightarrow{\ell} X'_2, \quad X'_2 \xrightarrow{\pi \cdot \ell} \tilde{X}_2. \tag{*}
\]

Suppose \(X''_2\) is a \(P\) configuration. Then, by determinacy, \(\ell = \tilde{\pi} \cdot \ell\) and \(X'_2 = \tilde{\pi} \cdot \tilde{X}_2\).

Recall that \(\tilde{\pi}\) fixes all names in \(P\). Moreover, the closure conditions on \(P\)-to-\(S\) transitions stipulate that all names in \(P' \setminus P\) are reachable from \(P \cup \nu(v)\) through the store \(s\), where \(v, s\) the value and store components of \(\ell\) respectively. This implies that \(\tilde{\pi}\) fixes all names in \(P'\). Hence, from \(\tilde{X}_2 \xrightarrow{w} \tilde{X}_2\) we obtain \(w \in T(X'_2)\).

On the other hand, if \(X''_2\) is an \(S\) configuration then let \(a_1, \ldots, a_N\) be an enumeration of \(\nu(X''_2)\). We define permutations \(\pi_0, \pi_1, \ldots, \pi_N\) by:

\[
\pi_0 = \text{id}, \quad \pi_{i+1} = (a_{i+1} \cdot (\pi_i \circ \tilde{\pi})(a_{i+1})) \circ \pi_i.
\]

We claim that, for each \(0 \leq i \leq N\) and \(1 \leq j \leq i\), we have

\[
\pi_i \cdot \tilde{\pi} \cdot a_j = a_j, \quad \forall a \in \nu(\ell) \setminus P, \pi_i \cdot \tilde{\pi} \cdot a = \tilde{\pi} \cdot a, \quad \forall a \in P, \pi_i \cdot a = a.
\]

We do induction on \(i\); the case of \(i = 0\) is clear. For the inductive step, if \(\pi_i \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}\) then \(\pi_{i+1} = \pi_i\), and \(\pi_{i+1} \cdot \tilde{\pi} \cdot a_{i+1} = \pi_{i+1} \cdot \tilde{\pi} \cdot a_i = a_{i+1}\). Moreover, by IH, \(\pi_{i+1} \cdot \tilde{\pi} \cdot a_j = a_j\) for all \(1 \leq j \leq i\), and \(\pi_{i+1} \cdot \tilde{\pi} \cdot a = \tilde{\pi} \cdot a\) for all \(a \in \nu(\ell) \setminus P\), and \(\pi_{i+1} \cdot a = a\) for all \(a \in P\). If \(\pi_i \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}\) then, by construction, \(\pi_{i+1} \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}\). Moreover, for each \(1 \leq j \leq i\), by IH, \(\pi_{i+1} \cdot \tilde{\pi} \cdot a_j = (a_{i+1} \cdot a_{i+1}) \cdot a_j\), and the latter equals \(a_j\) since \(a_{i+1} \neq a_j\) implies \(a'_{i+1} \neq \pi_i \cdot \tilde{\pi} \cdot a_j = a_j\). For any \(a \in \nu(\ell) \setminus P\), \(\pi_{i+1} \cdot \tilde{\pi} \cdot a = (a_{i+1} \cdot a_{i+1}) \cdot \tilde{\pi} \cdot a = (a_{i+1} \cdot a_{i+1}) \cdot \tilde{\pi} \cdot a\), by IH.

Now, \(a \neq a_{i+1}\) since \(\nu(\ell) \cap \nu(X''_2) \subseteq P\), hence \(\tilde{\pi} \cdot a = \pi_i \cdot \tilde{\pi} \cdot a \neq a'_{i+1}\). Moreover, \(\nu(\tilde{\pi} \cdot \ell) \cap \nu(X''_2) \subseteq P\) implies \(\tilde{\pi} \cdot a \neq a_{i+1}\), so \(\pi_{i+1} \cdot a = \tilde{\pi} \cdot a\). Finally, for any \(a \in P\) we have \(\pi_{i+1} \cdot a = (a_{i+1} \cdot a_{i+1}) \cdot \pi_i \cdot a = (a_{i+1} \cdot a_{i+1}) \cdot a\). If \(a = a_{i+1}\) then \((a_{i+1} \cdot a_{i+1}) \cdot a = a_{i+1} \cdot a = \pi_i \cdot a = a\), by IH and the fact that \(\tilde{\pi}\) fixes all \(a \in P\). If \(a = a'_{i+1}\) then \((a_{i+1} \cdot a_{i+1}) \cdot a = a_{i+1} = (\pi_i \circ \tilde{\pi})^{-1} \cdot a = a\).

Setting \(\tilde{\pi} = \pi_N \circ \tilde{\pi}\), for each \(1 \leq j \leq N\) we thus have \(\tilde{\pi} \cdot a_j = a_j\) and \(\tilde{\pi} \cdot \ell = \tilde{\pi} \cdot \ell\).

So, \(\tilde{\pi} \cdot X''_2 = X''_2\) and therefore \(X''_2 \xrightarrow{\pi \ell} \tilde{\pi} \cdot X'_2\), that is, \(X''_2 \xrightarrow{\pi \ell} \tilde{\pi} \cdot X'_2\). Hence, by (\(\ast\)) and determinacy, \(\tilde{\pi} \cdot X'_2 = \tilde{\pi} \cdot \tilde{X}_2\), so \(X'_2 = \tilde{\pi}^{-1} \cdot \tilde{\pi} \cdot X'_2\). But observe that \(\tilde{\pi}^{-1} \circ \tilde{\pi}\) fixes all names in \(P' = P \cup \nu(\ell)\) and therefore \(w \in T(X'_2)\).

The above shows that \(T(X'_1) \subseteq T(X'_2)\) and similarly we show \(T(X'_2) \subseteq T(X'_1)\).

Hence, \((X'_1, X'_2) \mathcal{R} \text{ and } \mathcal{R} \text{ is a simulation.}\)

\[\square\]

\textbf{Proposition C.2} \textit{Trace equivalence is a congruence for module composition}. \textemdash.
**Proof.** Let $M_1, M_2$ be modules with common public names, and $M$ a third module. By the previous lemma, it suffices to show that $M_1 \sim M_2$ implies $(M_1 \cdot M) \sim (M_2 \cdot M)$. So let us assume $M_1 \sim M_2$ with $\mathcal{R}$ a witnessing bisimulation and let $R_i : [M_i] \otimes [M] \to [M_i \cdot M]$ be a $\tau$-isomorphism, for $i = 1, 2$. We define the following relation between configurations of $[M_1 \cdot M]$ and $[M_2 \cdot M]$.

$$\mathcal{R}' = \{ (R_1(X_1 \otimes_H Y), R_2(X_2 \otimes_H Y)) \mid X_i \otimes_H Y \text{ reachable}, (X_1, X_2) \in \mathcal{R} \}$$

Using Proposition B.2 we can show that $\mathcal{R}'$ is a bisimulation. \qed
A Representation Theorem for Unique Decomposition Categories

Naohiko Hoshino

Research Institute for Mathematical Sciences
Kyoto University
Kyoto, Japan

Abstract

Haghverdi introduced the notion of unique decomposition categories as a foundation for categorical study of Girard’s Geometry of Interaction (GoI). The execution formula in GoI provides a semantics of cut-elimination process, and we can capture the execution formula in every unique decomposition category: each hom-set of a unique decomposition category comes equipped with a partially defined countable summation, which captures the countable summation that appears in the execution formula. The fundamental property of unique decomposition categories is that if the execution formula in a unique decomposition category is always defined, then the unique decomposition category has a trace operator that is given by the execution formula. In this paper, we introduce a subclass of unique decomposition categories, which we call strong unique decomposition categories, and we prove the fundamental property for strong unique decomposition categories as a corollary of a representation theorem for strong unique decomposition categories: we show that for every strong unique decomposition category \( C \), there is a faithful strong symmetric monoidal functor from \( C \) to a category with countable biproducts, and the countable biproducts characterize the structure of the strong unique decomposition category via the faithful functor.

Keywords: Geometry of interaction, unique decomposition category, traced monoidal category, representation theorem

1 Introduction

Girard introduced Geometry of Interaction (GoI) [3], which aims to capture semantics of cut-elimination process rather than invariant under cut-elimination like usual denotational semantics. GoI interprets proofs as square matrices, and if a proof reduces to another proof via cut-elimination, then the execution formula

\[
\text{Ex} \left( \begin{array}{cc} A & B \\ C & D \end{array} \right) := A + \sum_{n=0}^{\infty} BD^n C
\]

provides an invariant under the cut-elimination.

Email: naophiko@kurims.kyoto-u.ac.jp

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
Work by Hyland, Abramsky, Haghverdi and Scott \cite{4,1} showed that traced symmetric monoidal categories \cite{11} play important roles in modeling the execution formula. Especially, in \cite{4,5}, Haghverdi and Scott got much closer to the original execution formula by using unique decomposition categories. The notion of unique decomposition categories introduced by Haghverdi is a generalization of partially additive categories \cite{15}. The main point of unique decomposition categories is that in a unique decomposition category, we can uniquely decompose a morphism $f : X \otimes Z \to Y \otimes Z$ into four components

$$\begin{pmatrix}
  f_{XY} : X \to Y \\
  f_{ZY} : Z \to Y \\
  f_{XZ} : X \to Z \\
  f_{ZZ} : Z \to Z
\end{pmatrix},$$

and each hom-set comes equipped with a partially defined countable summation. For example, we can partially define the standard trace formula \cite{5}:

$$f_{XY} + \sum_{n=0}^{\infty} \left( f_{ZY} \circ f_{XZ} \circ f_{ZZ} \right) : X \to Y.$$

The following fundamental property of unique decomposition categories connects the standard trace formula with categorical trace operators.

**Proposition 1.1** (\cite{4,5}) If the standard trace formula is defined for any morphism of the form $f : X \otimes Z \to Y \otimes Z$, then the standard trace formula provides a trace operator of the unique decomposition category.

In the proof of the proposition, there are certain implicit assumptions aside from the definition of unique decomposition categories (see Appendix B in \cite{8}), and a sufficient condition would be to require quasi projections and quasi injections, which is a part of data of unique decomposition categories, to be “natural” and “compatible with monoidal structural isomorphisms”. The main motivation of this paper is to explicitly describe a subclass of unique decomposition categories that enjoys the fundamental property. Our idea is to find a subclass of unique decomposition categories that provides “good” embedding of unique decomposition categories in the subclass into categories with countable biproducts. We consider categories with countable biproducts because countable biproducts always provide a trace operator given by the execution formula (see Section 5). Although we found a subclass of unique decomposition categories, namely strong unique decomposition categories, in this paper by trial and error, organization of this paper is top-down:

(i) In Section 2, we recall Kleene equality, biproducts and categorical traces.

(ii) In Section 3, we recall the definition of $\Sigma$-monoids and embed each $\Sigma$-monoid into a total $\Sigma$-monoid.

(iii) In Section 4, we introduce strong unique decomposition categories, and we embed a strong unique decomposition category into a total strong unique decomposition category via the embedding in (ii). We give examples of strong unique decomposition categories.
(iv) In Section 5, we embed a total strong unique decomposition category into a category with countable biproducts by matrix construction [13]. Then, we give a representation theorem for strong unique decomposition categories (Theorem 5.3). The fundamental property for strong unique decomposition categories is a corollary of the representation theorem. Consequences of the representation theorem are:

- A proof of Proposition 1.1 in which we do not need to be careful with partiality of summations on hom-sets of strong unique decomposition categories.
- We show that all strong unique decomposition categories are \textit{partially traced}.

Related work

The paper by Malherbe, Scott and Selinger [14] is closely related to our work. They gave an embedding of partially traced symmetric monoidal categories introduced in [6] into traced symmetric monoidal categories. Since our result tells us that every strong unique decomposition category is partially traced (Corollary 5.4), we can embed a strong unique decomposition category into a traced symmetric monoidal category by their result. On the other hand, our result also provides an embedding of a strong unique decomposition category into a traced symmetric monoidal category since a category with countable biproducts is traced (Theorem 3 in [16]). As we concentrate only on strong unique decomposition categories, our embedding tells us further information on strong unique decomposition categories: an explicit description of their trace operators, for example. However, there are some other partially traced symmetric monoidal categories that are not strong unique decomposition categories. At this point, we do not know clear comparison between our work and their work.

2 Preliminary

2.1 Kleene equality

For expressions $e$ and $e'$ that possibly include partial operations, we write $e \preceq e'$ if $e$ is defined, then $e'$ is defined, and they denote the same value. We use $\simeq$ for the \textit{Kleene equality}: we write $e \simeq e'$ when we have $e \preceq e'$ and $e' \preceq e$. For example, the following Kleene equality holds for all real numbers $x$ and $y$.

$$
\frac{x \cdot 3}{x^2} \cdot \frac{1}{y^2} \cdot y \simeq \frac{3}{x \cdot y}.
$$

2.2 Biproducts

\textbf{Definition 2.1} Let $C$ be a category. For a set $I$, an \textit{I-ary biproduct} of a family \{$X_i \in C\}_{i \in I}$ consists of an object $\bigoplus_{i \in I} X_i$ and a family of $C$-morphisms \{$\pi_i : \bigoplus_{i \in I} X_i \Rightarrow X_i : \kappa_i\}_{i \in I}$ such that

\begin{itemize}
  \item $\pi_i \circ \kappa_i = \text{id}_{X_i}$ for every $i \in I$.
\end{itemize}
Hoshino

- $\bigoplus_{i \in I} X_i$ with $\{\pi_i\}_{i \in I}$ forms a product of $\{X_i\}_{i \in I}$.
- $\bigoplus_{i \in I} X_i$ with $\{\kappa_i\}_{i \in I}$ forms a coproduct of $\{X_i\}_{i \in I}$.
- For each $f_i : X_i \to Y_i$, the tupling $(f_i \circ \pi_i)_{i \in I} : \bigoplus_{i \in I} X_i \to \bigoplus_{i \in I} Y_i$ coincides with the cotupling $[\kappa_i \circ f_i]_{i \in I} : \bigoplus_{i \in I} X_i \to \bigoplus_{i \in I} Y_i$.

A zero object is a $\emptyset$-ary biproduct, and a binary biproduct of $X_0$ and $X_1$ is a $\{0, 1\}$-ary biproduct of $\{X_i\}_{i \in \{0,1\}}$, for which we write $X_0 \oplus X_1$.

We use countable to mean at most countable. We say that $C$ has countable (finite) biproducts when for every countable (finite) set $I$ and every $I$-indexed family of $C$-objects, there exists an $I$-ary biproduct of the family.

**Definition 2.2** Let $F : C \to D$ be a functor between categories with finite biproducts. We say that $F$ preserves finite biproducts when for any objects $X_0, X_1 \in C$, the canonical morphisms $[F\kappa_0, F\kappa_1] : FX_0 \oplus FX_1 \cong F(X_0 \oplus X_1) : (F\pi_0, F\pi_1)$ and $F0 \cong 0$ form isomorphisms.

The definition of biproducts is from [9]. Definition 2.1 depends on neither abelian-group enrichment as in [13] nor existence of zero morphisms defined through a zero object as in [10]. The above definition of finite biproducts is equivalent to the definition of finite biproducts in [10].

### 2.3 Partial trace operators

Let $C$ be a symmetric monoidal category (for the definition, see [13]). We recall the definition of partial trace operators in [6] that is a generalization of trace operators introduced in [11].

**Definition 2.3** A partial trace operator of $C$ is a family of partial maps

$$\{\text{tr}^Z_{X,Y} : C(X \otimes Z, Y \otimes Z) \to C(X, Y)\}_{X,Y,Z \in C}$$

subject to the following conditions:

- **(Naturality)** For $g : X' \to X$, $h : Y \to Y'$ and $f : X \otimes Z \to Y \otimes Z$,

  $$h \circ \text{tr}^Z_{X,Y}(f) \circ g \preceq \text{tr}^Z_{X',Y'}(((h \otimes \text{id}_Z) \circ f) \circ (g \otimes \text{id}_Z)).$$

- **(Dinaturality)** For $f : X \otimes Z \to Y \otimes Z'$ and $g : Z' \to Z$,

  $$\text{tr}^Z_{X,Y}(\text{id}_Y \otimes g) \circ f) \simeq \text{tr}^{Z'}_{X,Y}(f \circ (\text{id}_X \otimes g)).$$

- **(Vanishing I)** For $f : X \otimes I \to Y \otimes I$,

  $$\text{tr}^1_{X,Y}(f) \simeq f.$$

- **(Vanishing II)** For $f : X \otimes Z \otimes W \to Y \otimes Z \otimes W$,

  $$\text{tr}^W_{X \otimes Z, Y \otimes Z}(f) \text{ is defined} \implies \text{tr}^{Z \otimes W}_{X,Y}(f) \simeq \text{tr}^Z_{X,Y}(\text{tr}^W_{X \otimes Z, Y \otimes Z}(f)).$$

204
• (Superposing) For \( f : X \otimes Z \to Y \otimes Z \),
\[
\text{id}_W \otimes \tr_{X,Y}^Z(f) \leq \tr_{W \otimes X, W \otimes Y}^Z(\text{id}_W \otimes f).
\]

• (Yanking)
\[
\tr_{X,X}^X(\sigma_{X,X}) \simeq \text{id}_X.
\]

Here we omit several coherence isomorphisms. Although our superposing rule is weaker than the original superposing rule in [6], we can derive the original superposing rule from the above axioms. A \textit{trace operator} is a partial trace operator consisting of total maps. We say that a partial trace operator is \textit{uniform} when for any \( f : X \otimes Z \to Y \otimes Z, g : X \otimes Z' \to Y \otimes Z' \) and \( h : Z \to Z' \), if \((\text{id}_Y \otimes h) \circ f = g \circ (\text{id}_X \otimes h)\), then \( \tr_{X,Y}^Z(f) \simeq \tr_{X,Y}^Z(g) \).

\section{\( \Sigma \)-monoids}

We recall the definition of \( \Sigma \)-monoids from [4]. For a set \( X \), a \textit{countable family} on \( X \) is a map \( x : I \to X \) for a countable set \( I \). We denote such a family \( x \) by \( \{x_i\}_{i \in I} \). A \textit{countable partition} of a set \( I \) is a countable family \( \{I_j\}_{j \in J} \) consisting of pairwise disjoint subsets of \( I \) such that \( \bigcup_{j \in J} I_j = I \). We define \( X^* \) to be the set of countable families on \( X \) whose indexing sets are subsets of the set of natural numbers \( \mathbb{N} = \{0, 1, 2, \cdots\} \). This restriction on indexing sets is to define \( \Sigma \) in the following definition to be a set theoretic partial map.

\textbf{Definition 3.1} A \textit{\( \Sigma \)-monoid} is a set \( X \) with a partial map \( \Sigma : X^* \to X \) subject to the following axioms:

• If \( I \) is a singleton \( \{n\} \), then \( \Sigma\{x_i\}_{i \in I} \simeq x_n \).

• If \( \{I_j\}_{j \in J} \) is a countable partition of a countable subset \( I \subset \mathbb{N} \), then for every countable family \( \{x_i\}_{i \in I} \) on \( X \), we have \( \Sigma\{x_i\}_{i \in I} \simeq \Sigma \{\Sigma\{x_i\}_{i \in I_j}\}_{j \in J} \).

A countable family \( \{x_i\}_{i \in I} \) is \textit{summable} when \( \Sigma\{x_i\}_{i \in I} \) is defined. We say that a \( \Sigma \)-monoid \((X, \Sigma)\) is \textit{total} when the operator \( \Sigma \) is a total map.

In the following, we simply say that \( X \) is a \( \Sigma \)-monoid without mentioning its sum operator, and we write \( \sum_{i \in I} x_i \) for \( \Sigma\{x_i\}_{i \in I} \). We informally write \( x_0 + x_1 + \cdots \) for \( \sum_{i \in \mathbb{N}} x_i \) and \( x_0 + x_1 + \cdots + x_n \) for \( \sum_{i \in \{0, 1, \cdots, n\}} x_i \). By the definition of \( \Sigma \)-monoids, every subfamily of a summable countable family is summable. Especially, the empty family \( \emptyset \) is summable. The \textit{zero element} \( \emptyset := \sum \emptyset \) behaves as a unit of the summation: \( \sum_{i \in I} x_i \simeq \sum_{j \in \{i \in I \mid x_i \neq 0\}} x_j \). We note that \( \Sigma\{x_i\}_{i \in I} \simeq \Sigma\{y_j\}_{j \in J} \) when there is a bijection \( \theta : I \to J \) such that \( x_i = y_{\theta(i)} \) for every \( i \in I \). For a proof, see [7].

For every countable set \( S \), we can define \( S \)-indexed summation \( \sum_{s \in S} x_s \) by choosing a bijection \( \theta : I \to S \) for some subset \( I \subset \mathbb{N} \): we define \( \sum_{s \in S} x_s \) to be \( \sum_{i \in I} x_{\theta(i)} \). The definition does not depend on our choice of \( I \) and the bijection \( \theta : I \to S \) since the summation is independent of renaming of indexing sets. Hence, the definition is well-defined. In the following, we implicitly extend summations in
this way.

**Example 3.2** Let $M$ be a commutative monoid that does not have non-trivial subgroup. $M$ forms a Σ-monoid by the following summation:

\[
\sum_{i \in I} x_i := \begin{cases} 
\sum_{i \in I'} x_i & (I' := \{i \in I \mid x_i \neq 0\} \text{ is finite}) \\
\text{undefined} & \text{(otherwise)}
\end{cases}
\]

Examples are the set of natural numbers and the set of non-negative reals associated with the addition. Another example is $M/N$ where $M$ is a commutative monoid, and $N$ is the submonoid of $M$ consisting of invertible elements in $M$. Generally, if an element of a Σ-monoid is invertible, then it is equal to the zero element:

\[x = x + 0 + 0 + \cdots = x + (-x) + x + (-x) + x + \cdots = 0.\]

**Example 3.3** A bounded complete poset $D$ forms a Σ-monoid:

\[
\sum_{i \in I} x_i := \begin{cases} 
\bigvee_{i \in I} x_i & (\{x_i \in D \mid i \in I\} \text{ is bounded}) \\
\text{undefined} & \text{(otherwise)}
\end{cases}
\]

3.1 The category of Σ-monoids

We define a category $\mathbf{M}$ of Σ-monoids: objects are Σ-monoids, and a morphism $f : X \to Y$ is a map $f : X \to Y$ such that for each summable countable family $\{x_i\}_{i \in I}$ on $X$, the summation $\sum_{i \in I} fx_i$ is defined to be $f(\sum_{i \in I} x_i)$. In this section, we show that $\mathbf{M}$ is a symmetric monoidal closed category. Due to lack of space, proofs of propositions in this section are in [8].

**Definition 3.4** For a positive natural number $n$ and Σ-monoids $X_1, \cdots, X_n$ and $Y$, we say that a map $f : X_1 \times \cdots \times X_n \to Y$ is $n$-linear when

\[
f(x_1, \cdots, x_{k-1}, -, x_{k+1}, \cdots, x_n) : X_k \to Y
\]

is an $\mathbf{M}$-morphism for all $k = 1, 2, \cdots, n$ and $x_1 \in X_1$, $\cdots$, $x_n \in X_n$. We write $\mathbf{M}(X_1, \cdots, X_n; Y)$ for the set of $n$-linear morphisms of the form $f : X_1 \times \cdots \times X_n \to Y$.

**Proposition 3.5** A functor $\mathbf{M}(X, Y; -) : \mathbf{M} \to \mathbf{Set}$ is representable, i.e., there is an object $X \otimes Y$ such that $\mathbf{M}(X, Y; -) \cong \mathbf{M}(X \otimes Y, -)$.

We define $I$ to be a Σ-monoid $\{0, 1\}$ associated with a summation

\[
\sum_{i \in I} x_i := \begin{cases} 
0 & (\{i \in I \mid x_i = 1\} \text{ is empty}) \\
1 & (\{i \in I \mid x_i = 1\} \text{ is a singleton}) \\
\text{undefined} & \text{(otherwise)}
\end{cases}
\]
For \(\Sigma\)-monoids \(X\) and \(Y\), we define a \(\Sigma\)-monoid \([X,Y] := M(X,Y)\) by

\[
\sum_{i \in I} f_i := \begin{cases} 
\lambda x. \sum_{i \in I} f_i x & (\text{\(\sum_{i \in I} f_i x\) is defined for all } x \in X) \\
\text{undefined} & (\text{otherwise}).
\end{cases}
\]

(1)

**Proposition 3.6** \((M, I, \otimes, [-, -])\) is a symmetric monoidal closed category.

### 3.2 A reflective full subcategory \(M_t\)

We define \(M_t\) to be the full subcategory of \(M\) consisting of total \(\Sigma\)-monoids.

**Lemma 3.7** The inclusion functor \(U : M_t \to M\) has a left adjoint functor.

**Proof.** For \(X \in M\), let \(S\) be the set of total \(\Sigma\)-monoids whose underlying sets are quotients of \(X^*\). We show that \(S\) satisfies the solution set condition: for each morphism \(f : X \to Y\) whose codomain \(Y\) is in \(M_t\), there exists a morphism \(s : X \to A\) and a morphism \(h : A \to Y\) for some \(A \in S\) such that \(f = h \circ s\). We define a map \(p : X^* \to Y\) by \(p\{x_i\}_{i \in I} := \sum_{i \in I} f x_i\). Let \(A\) be the quotient of \(X^*\) by an equivalence relation on \(X^*\) given by \(\{x_i\}_{i \in I} \approx \{x'_j\}_{j \in J} \iff p\{x_i\}_{i \in I} = p\{x'_j\}_{j \in J}\). Since the image of \(p\) is closed under the summation of \(Y\), the total \(\Sigma\)-monoid structure of \(Y\) induces a total \(\Sigma\)-monoid structure of \(A\), and we obtain a monomorphism \(h : A \to Y\). Since the image of \(f\) is in the image of \(h\), there exists a morphism \(s : X \to A\) such that \(f = h \circ s\). Hence, \(S\) satisfies the solution set condition. Since \(M_t\) is small complete \([8]\), and \(U\) preserves all limits, \(U\) has a left adjoint functor by the adjoint functor theorem \([13]\). \(\square\)

For a category \(C\), a **reflective full subcategory of \(C\)** is a full subcategory of \(C\) such that the inclusion functor has a left adjoint functor. For a symmetric monoidal closed category \((C, I, \otimes, [-, -])\) and its full subcategory \(B\), we say that \(B\) is an **exponential ideal of \(C\)** when for any \(X \in C\) and \(Y \in B\), the exponential \([X,Y]\) is a \(B\)-object.

**Theorem 3.8** \([2]\) Let \(B\) be a reflective full subcategory of a symmetric monoidal closed category \(C\). If \(B\) is an exponential ideal of \(C\), then \(B\) has a symmetric monoidal closed structure, and the adjunction is symmetric monoidal.

By the definition (1) of the exponential \([-,-]\) of \(M\), it is easy to check that \(M_t\) is an exponential ideal of \(M\).

**Corollary 3.9** \(M_t\) is a symmetric monoidal closed category, and the adjunction between \(M\) and \(M_t\) is symmetric monoidal with respect to the structures.

Let \(T\) be the symmetric monoidal monad on \(M\) induced by the symmetric monoidal adjunction. We show several properties of the unit \(\eta_X : X \to TX\).

**Definition 3.10** We say that an \(M\)-morphism \(f : X \to Y\) **reflects summability** when for every countable family \(\{x_i\}_{i \in I}\) on \(X\) if \(\sum_{i \in I} f x_i\) is summable and is in the image of \(f\), then \(\{x_i\}_{i \in I}\) is summable.
Lemma 3.11 The unit $\eta_X : X \to TX$ is monic and reflects summability.

Proof. We define a total $\Sigma$-monoid $X'$ by $X' := X + \{\bot\}$ with a summation

$$\sum_{i \in I} y_i := \begin{cases} \text{inl} \left( \sum_{i \in I} x_i \right) & \text{(for each } i \in I, \text{ } y_i \text{ is of the form } \text{inl}(x_i), \text{ and } \{x_i\}_{i \in I} \text{ is summable} \\ \text{inr}(\bot) & \text{(otherwise)} \end{cases}$$

where inl($-$) is the left injection, and inr($-$) is the right injection. We define an $M$-morphism $h : X \to X'$ by $hx := \text{inl}(x)$. Since an $M$-morphism is monic if and only if its underlying map is injective, $h$ is monic. Let $k : TX \to X'$ be the unique morphism such that $h = k \circ \eta_X$. Since $h : X \to X'$ is monic, the unit $\eta_X$ is also monic. For a countable family $\{x_i\}_{i \in I}$ on $X$, if $\sum_{i \in I} \eta_X x_i$ is in the image of $\eta_X$, then we have

$$\sum_{i \in I} hx_i = \sum_{i \in I} k \eta_X x_i = k \left( \sum_{i \in I} \eta_X x_i \right) \in \text{image}(k \circ \eta_X) = \text{image}(h),$$

which means that $\{x_i\}_{i \in I}$ is summable. Hence, $\eta_X$ reflects summability. \hfill \Box

Although our construction of $T$ is abstract, for some $\Sigma$-monoids $X$, we can concretely describe $TX$ via the universality of $T$.

Example 3.12 For countable sets $A$ and $B$, let $\text{Pfn}(A, B)$ be the set of partial maps from $A$ to $B$. The set $\text{Pfn}(A, B)$ forms a $\Sigma$-monoid by the union of graph relations:

$$\sum_{i \in I} f_i := \begin{cases} \bigcup_{i \in I} f_i & \text{(} \bigcup_{i \in I} f_i \text{ represents a partial map)} \\ \text{undefined} & \text{(otherwise)} \end{cases}$$

Let $\text{Rel}(A, B)$ be the set of relations between $A$ and $B$, which forms a total $\Sigma$-monoid by the union of graphs. There is an obvious inclusion $h : \text{Pfn}(A, B) \to \text{Rel}(A, B)$ between $\Sigma$-monoids. For a total $\Sigma$-monoid $X$ and an $M$-morphism $f : \text{Pfn}(A, B) \to X$, there is an $M$-morphism $g : \text{Rel}(A, B) \to X$ given by $g(R) := \sum_{(a,b) \in R} f(\delta_{a,b})$ where $\delta_{a,b} := \{(a, b)\}$. Since every partial map in $\text{Pfn}(A, B)$ is equal to a sum of partial maps of the form $\delta_{a,b}$, we obtain $g \circ h = f$. Such $g$ is unique since $g$ must satisfy the following equation.

$$g(R) = g \left( \sum_{(a,b) \in R} \delta_{a,b} \right) = \sum_{(a,b) \in R} g(\delta_{a,b}) = \sum_{(a,b) \in R} gh(\delta_{a,b}) = \sum_{(a,b) \in R} f(\delta_{a,b}).$$

By the universality of $T$, we see that $T \text{Pfn}(A, B)$ is isomorphic to $\text{Rel}(A, B)$.

Example 3.13 For a countable set $A$, we define sets $A^*$ and $A^\ast$ by

$$A^* := \{x : A \to N \mid \text{dom}(x) \text{ is finite} \}, \quad A^\ast := \text{Set}(A, \mathbb{N} \cup \{\infty\})$$
where \( \text{dom}(x) := \{ a \in A \mid x(a) \neq 0 \} \). The sets \( A^* \) and \( A^\ast \) are \( \Sigma \)-monoids with the pointwise summations. The \( \Sigma \)-monoid \( A^\ast \) is total. As in Example 3.12, we can show that \( TA^\ast \) is isomorphic to \( A^\ast \).

4 Unique decomposition categories

4.1 M-categories

With respect to the symmetric monoidal structure of \( M \), we consider \( M \)-enrichment [12]. By Proposition 3.5, we can say that an \( M \)-enriched category (\( M \)-category) \( C \) is a category with a \( \Sigma \)-monoid structure on each hom-set \( C(X,Y) \) such that for any summable countable families \( \{ f_i : X \to Y \}_{i \in I} \) and \( \{ g_j : Y \to Z \}_{j \in J} \), the summation \( \sum_{(i,j) \in I \times J} g_j \circ f_i \) is defined to be \( (\sum_{j \in J} g_j) \circ (\sum_{i \in I} f_i) \), i.e., the composition distributes over the summations if they exist. We write \( 0_{X,Y} : X \to Y \) for the zero element in the \( \Sigma \)-monoid \( C(X,Y) \) and call \( 0_{X,Y} \) a zero morphism. By the definition of \( M \)-categories, the composition of a morphism with a zero morphism is a zero morphism.

For \( M \)-categories \( C \) and \( D \), an \( M \)-enriched functor (\( M \)-functor) \( F : C \to D \) is a functor from \( C \) to \( D \) such that for any \( X,Y \in C \), the map \( F : C(X,Y) \to D(FX,FY) \) is an \( M \)-morphism. We say that \( F : C \to D \) reflects summability when \( F : C(X,Y) \to D(FX,FY) \) reflects summability for all \( X \) and \( Y \) in \( C \).

By symmetric monoidal \( M \)-category, we mean an \( M \)-category with a symmetric monoidal structure on its underlying category. We do not assume that the symmetric monoidal structure is compatible with the \( M \)-enrichment. For symmetric monoidal \( M \)-category \( C \) and \( D \), a symmetric monoidal \( M \)-functor from \( C \) to \( D \) is an \( M \)-functor from \( C \) to \( D \) that is symmetric monoidal.

4.2 Strong unique decomposition categories

We recall the definition of unique decomposition categories in [4], and we give a subclass of unique decomposition categories.

Definition 4.1 A unique decomposition category is a symmetric monoidal \( M \)-category such that for all \( i \in I \), there are morphisms called quasi projections \( \rho_i : \bigotimes_{i \in I} X_i \to X_i \) and quasi injections \( \iota_i : X_i \to \bigotimes_{i \in I} X_i \) subject to the following axioms:

\[
\rho_i \circ \iota_j = \begin{cases} id_{X_i} & (i = j) \\ 0_{X_j,X_i} & \text{(otherwise)} \end{cases}, \quad \sum_{i \in I} \iota_i \circ \rho_i \simeq id_{\bigotimes_{i \in I} X_i}.
\]

Definition 4.2 A strong unique decomposition category \( C \) is a symmetric monoidal \( M \)-category \( C \) such that

- The identity on the unit \( I \) is equal to \( 0_{1,1} \).
- \( id_X \otimes 0_{Y,Z} + 0_{X,Z} \otimes id_Y \) is defined to be \( id_{X \otimes Z} \).

We say that \( C \) is total when every hom-object is a total \( \Sigma \)-monoid.
The class of strong unique decomposition categories forms a subclass of unique decomposition categories: a strong unique decomposition category has binary quasi projections and binary quasi injections given as follows:

\[
\rho_{X,Y} := X \otimes Y \xrightarrow{\text{id}_X \otimes 0_{Y,I}} X \otimes I \xrightarrow{\sim} X \\
\rho'_{X,Y} := X \otimes Y \xrightarrow{0_{X,I} \otimes \text{id}_Y} I \otimes Y \xrightarrow{\sim} Y \\
\iota_{X,Y} := X \xrightarrow{\sim} X \otimes I \xrightarrow{\text{id}_X \otimes 0_{Y,I}} X \otimes Y \\
\iota'_{X,Y} := Y \xrightarrow{\sim} Y \otimes I \xrightarrow{0_{I,X} \otimes \text{id}_Y} X \otimes Y.
\]

We can similarly define quasi projections and quasi injections for general cases. It is easy to check that a strong unique decomposition category with the above morphisms forms a unique decomposition category.

**Remark 4.3** As the main point of unique decomposition categories is their unique decomposition of morphisms into matrices of morphisms via quasi projections and quasi injections (Proposition 4.0.6 in [4]), it would be better to employ quasi projections and quasi injections as primal data for strong unique decomposition categories. We choose the above definition of strong unique decomposition categories because of its compactness. At this point, we do not know “equivalent” definition that employs quasi projections and quasi injections as primal data, which would consist of a series of equalities that require quasi projections and quasi injections to be natural and compatible with monoidal structural isomorphisms. In fact, the above quasi projections and quasi injections satisfy naturality and compatibility with monoidal structural isomorphisms; see Proposition 4.8 for the case of total unique decomposition categories.

**Example 4.4** All the examples of unique decomposition categories in [5] are strong unique decomposition categories. For example, sets and partial injections, sets and partial maps, sets and relations are strong unique decomposition categories.

**Example 4.5** The opposite category of a strong unique decomposition category is a strong unique decomposition category.

**Example 4.6** A category \( \mathcal{C} \) with countable biproducts is a total strong unique decomposition category, c.f. [4]. For a countable family \( \{f_i\}_{i \in I} \) on \( \mathcal{C}(X,Y) \), we define its summation by

\[
\sum_{i \in I} f_i := X \xrightarrow{\delta_X} \bigoplus_{i \in I} X \xrightarrow{\bigoplus_{i \in I} f_i} \bigoplus_{i \in I} Y \xrightarrow{\gamma_X} Y
\]

where \( \delta_X \) and \( \gamma_X \) are the diagonal morphisms. Since the composition distributes over the summation, we obtain an \( \mathcal{M} \)-enrichment of \( \mathcal{C} \). We take the finite biproducts as a symmetric monoidal structure of \( \mathcal{C} \). By these data, \( \mathcal{C} \) forms a strong unique decomposition category. Concrete examples are: sets and relations, sup-complete lattices and continuous maps, and \( \mathcal{M}_t \).

**Example 4.7** Let \( F : \mathcal{C} \to \mathcal{D} \) be a faithful functor from a symmetric monoidal category \( \mathcal{C} \) to a category \( \mathcal{D} \) with countable biproducts. We say that \( F : \mathcal{C} \to \mathcal{D} \) is *downward-closed* when for every countable family \( \{f_i : X \to Y\}_{i \in I} \) on \( \mathcal{C} \)-morphisms,
if the summation $\sum_{i \in I} Ff_i : FX \to FY$ is in the image of $F$, then for every subset $J \subset I$, the summation $\sum_{i \in J} Ff_i : FX \to FY$ is also in the image of $F$. If the faithful functor $F : \mathcal{C} \to \mathcal{D}$ is downward-closed, then $\mathcal{C}$ forms a strong unique decomposition category: for a countable family $\{f_i\}_{i \in I}$ on $\mathcal{C}(X,Y)$, we define $\sum_{i \in I} f_i$ to be $f$ when $\sum_{i \in J} Ff_i$ is equal to $Ff$; when $\sum_{i \in I} Ff_i$ is not in the image of $F$, we do not define $\sum_{i \in I} f_i$.

**Proposition 4.8** If a strong unique decomposition category is total, then it has finite biproducts: the unit is a zero object, and $X \otimes Y$ with morphisms $(\rho_{X,Y}, \rho'_{X,Y}, \eta_{X,Y}, \eta'_{X,Y})$ forms a biproduct of $X$ and $Y$. Furthermore, the symmetric monoidal structure coincides with the symmetric monoidal structure derived from the finite biproducts.

**Proof.** In every strong unique decomposition category, the unit is a zero object since the identity on the unit is a zero morphism. When the strong unique decomposition category is total, $(X \otimes Y, \rho_{X,Y}, \rho'_{X,Y})$ forms a product of $X$ and $Y$, and $(X \otimes Y, \eta_{X,Y}, \eta'_{X,Y})$ forms a coproduct of $X$ and $Y$. For $f : X \to Y$ and $g : Z \to W$, the tupling $\{f \circ \rho_{X,Z}, g \circ \rho'_{X,Z}\}$ is $\eta_Y \circ f \circ \rho_{X,Z} + \eta'_Y \circ g \circ \rho'_{X,Z}$, which is equal to the cotupling $\{\eta_W \circ f, \eta'_W \circ g\}$. Hence, $(X \otimes Y, \rho_{X,Y}, \rho'_{X,Y}, \eta_{X,Y}, \eta'_{X,Y})$ forms a biproduct of $X$ and $Y$. By the universality of biproducts, we can check that coherence isomorphisms of the symmetric monoidal structure of the strong unique decomposition category coincide with the symmetric monoidal structure derived from the biproducts. \hfill \square

## 5 A representation theorem

For a strong unique decomposition category $\mathcal{C}$, since $T$ is a symmetric monoidal functor (Corollary 3.9), we can define a new $\mathcal{M}$-category $T\mathcal{C}$ by the action of $T$: objects are objects of $\mathcal{C}$, and $T\mathcal{C}(X,Y) := T(\mathcal{C}(X,Y))$. Furthermore, the unit $\eta_X : X \to TX$ induces an $\mathcal{M}$-functor $H : \mathcal{C} \to T\mathcal{C}$ given by $HX := X$ and $Hf := \eta_{\mathcal{C}(X,Y)}(f)$ for $f : X \to Y$.

**Proposition 5.1** $T\mathcal{C}$ is a total strong unique decomposition category, and $H$ is a faithful strong symmetric monoidal $\mathcal{M}$-functor that reflects summability.

**Proof.** We give a symmetric monoidal structure on the underlying category. For objects, we employ the symmetric monoidal structure of $\mathcal{C}$. For $f : X \to Y$ and $g : Z \to W$ in $T\mathcal{C}$, we define $f \otimes g : X \otimes Z \to Y \otimes W$ to be

$$H\eta_{Y,W} \circ f \circ H\rho_{X,Z} + H\eta'_{Y,W} \circ g \circ H\rho'_{X,Z}.$$ 

Functoriality of $\otimes$ follows from $\mathcal{M}$-enrichment of $H$. For example,

$$\text{id}_X \otimes \text{id}_Y = H(\eta_{X,Y} \circ \rho_{X,Y} + \eta'_X \circ \rho'_{X,Y}) = H(\text{id}_{X \otimes Y}) = \text{id}_{X \otimes Y}.$$ 

We can similarly check that $\otimes$ is compatible with the composition of $\mathcal{C}$. By $\mathcal{M}$-enrichment of $H$ again, we can check that $\otimes$ with $H\lambda_X, H\rho_X, H\alpha_{X,Y,Z}$ and $H\sigma_{X,Y}$ provide a symmetric monoidal structure on $T\mathcal{C}$ where $\lambda_X : X \otimes I \to X$, $\rho_X :$
I ⊗ X → X, α_{X,Y,Z} : X ⊗ (Y ⊗ Z) → (X ⊗ Y) ⊗ Z and σ_{X,Y} : X ⊗ Y → Y ⊗ X are the coherence isomorphisms of C. The identity on the unit is the zero morphism. In fact, Hid_1 = H0_{1,1} = 0_{1,1}. We also have

\[ \text{id}_X \otimes 0_{Y,Z} + 0_{X,Y} \otimes \text{id}_Z = H_{i,X,Y} \circ H_{\rho_{X,Y}} + H_{\rho'_{X,Y}} \circ H_{i,Y,Z} = \text{id}_X \otimes \text{id}_Z = \text{id}_{X \otimes Y} \]

in T_sC. Therefore, we see that T_sC is a strong unique decomposition category. Since T constructs total Σ-monomoids, T_sC is total. By the definition of symmetric monoidal structure of T_sC, we see that H is strong symmetric monoidal. The \( M \)-functor H is faithful and reflects summability by Lemma 3.11.

Since H : C → T_sC is faithful and reflects summability, H completely characterizes the summation of C-morphisms:

\[ \sum_{i \in I} f_i \text{ is defined to be } f \iff Hf = \sum_{i \in I} Hf_i \text{ in } T_sC(X,Y). \]

We go a bit farther so as to give an embedding into a category that is more familiar to us than total strong unique decomposition categories. For a total strong unique decomposition category \( \mathcal{A} \), we define a category \( \mathcal{B}(\mathcal{A}) \) by:

- An object is a countable family on the set of \( \mathcal{A} \)-objects.
- A morphism \( f : \{X_i\}_{i \in I} \rightarrow \{Y_j\}_{j \in J} \) is a family \( \{f_{ij} : X_i \rightarrow Y_j\}_{(i,j) \in I \times J} \).
- The identity \( \text{id}_{\{X_i\}_{i \in I}} \) on \( \{X_i\}_{i \in I} \) and the composition \( g \circ f \) are given by

\[
(id_{\{X_i\}_{i \in I}})_{i,i'} := \begin{cases} 
\text{id}_{X_i} & \text{if } i = i' \\
0_{X_i,X_{i'}} & \text{if } i \neq i'
\end{cases}, \quad (g \circ f)_{i,k} := \sum_{j \in J} g_{j,k} \circ f_{ij}.
\]

\( \mathcal{B}(\mathcal{A}) \) has countable biproducts: a biproduct \( \bigoplus_{i \in I} \{X_{ij}\}_{j \in J_i} \) of a countable family \( \{\{X_{ij}\}_{j \in J_i}\}_{i \in I} \) is \( \{X_{ij}\}_{(i,j) \in I \times J} \) whose \( i \)-th projection and \( i \)-th injection \( \pi_i : \bigoplus_{i \in I} \{X_{ij}\}_{j \in J_i} \rightarrow \{X_{ij}\}_{j \in J_i} : \kappa_i \) for \( i \in I \) are given as follows:

\[
\pi_i((i',j'),j) := \begin{cases} 
\text{id}_{X_{ij}} & \text{if } (i,j) = (i',j') \\
0_{X_{i',j},X_{ij}} & \text{otherwise},
\end{cases} \quad \kappa_i(j,(i',j')) := \begin{cases} 
\text{id}_{X_{ij}} & \text{if } (i,j) = (i',j') \\
0_{X_{ij},X_{i',j'}} & \text{otherwise}.
\end{cases}
\]

The induced summation of a countable family \( \{f_k : \{X_i\}_{i \in I} \rightarrow \{Y_j\}_{j \in J}\}_{k \in K} \) is pointwise: the \((i,j)\)-th entry of \( \sum_{k \in K} f_k \) is \( \sum_{k \in K}(f_k)_{ij} \). By Example 4.6, \( \mathcal{B}(\mathcal{A}) \) is a total strong unique decomposition category. A similar construction appears in [13] called matrix construction.

We define a fully faithful functor \( K : \mathcal{A} \rightarrow \mathcal{B}(\mathcal{A}) \) by \( KX := \{X\} \) and \( Kf := \{f\} \) where we simply write \( \{x\} \) for a family indexed by a singleton \( \{\bullet\} \) such that \( \{x\} \bullet = x \).

**Lemma 5.2** \( K \) is a fully faithful strong symmetric monoidal \( M \)-functor.
Proof. Since summations on hom-sets of \( \mathcal{B}(A) \) are pointwise, the functor \( K \) preserves summations, i.e., \( K \) is an \( M \)-functor. \( K \) is fully faithful by the definition. It remains to see that \( K \) is strong symmetric monoidal. Since the symmetric monoidal structure of \( A \) is given by the finite biproducts (Proposition 5.1 and Proposition 4.8), we show that \( K \) preserves finite biproducts. There are canonical morphisms \( \varphi := (K \rho_{X,Y}, K \rho'_{X,Y}) : K(X \otimes Y) \to KX \oplus KY \) and \( \psi := [K \iota_{X,Y}, K \iota'_{X,Y}] : KX \oplus KY \to K(X \otimes Y) \). By the universality of biproducts and \( M \)-enrichment of \( K \), we see that \( \varphi \circ \psi = id_{KX \oplus KY} \) and \( \psi \circ \varphi = K(\iota_{X,Y} \circ \rho_{X,Y}) + K(\iota'_{X,Y} \circ \rho'_{X,Y}) = id_{K(X \otimes Y)} \). It is easy to check that \( KI \) is a zero object of \( \mathcal{B}(A) \).

Now, we obtain a representation theorem for strong unique decomposition categories by composing two embeddings \( K \) and \( H \).

Theorem 5.3 For every strong unique decomposition category \( \mathcal{C} \), there is a category \( \mathcal{D} \) with countable biproducts and a faithful strong symmetric monoidal \( M \)-functor \( F : \mathcal{C} \to \mathcal{D} \) that is downward-closed and reflects summability.

Proof. By Proposition 5.1 and Lemma 5.2, for every strong unique decomposition category \( \mathcal{C} \), the category \( \mathcal{B}(T, \mathcal{C}) \) has countable biproducts, and we have a faithful strong symmetric monoidal \( M \)-functor \( KH : \mathcal{C} \to \mathcal{B}(T, \mathcal{C}) \) that reflects summability. Downward-closedness of \( KH \) follows from the axioms of \( \Sigma \)-monoids and that \( KH \) reflects summability.

The faithful functor \( KH \) characterizes the \( \Sigma \)-monoid structure on \( \mathcal{C}(X, Y) \):

\[
\sum_{i \in I} f_i \text{ is defined to be } f \iff KHf = \sum_{i \in I} KHf_i.
\]

So as to prove the fundamental property of strong unique decomposition categories, we construct a trace operator following the argument in [16]. Let \( \mathcal{D} \) be a category with countable biproducts. For \( f : X \oplus Z \to Y \oplus Z \) in \( \mathcal{D} \), we define \( f_{XY} : X \to Y \), \( f_{XZ} : X \to Z \), \( f_{ZY} : Z \to Y \) and \( f_{ZZ} : Z \to Z \) by:

\[
f_{XY} := \pi_0 \circ f \circ \kappa_0, \quad f_{XZ} := \pi_1 \circ f \circ \kappa_0, \quad f_{ZY} := \pi_0 \circ f \circ \kappa_1, \quad f_{ZZ} := \pi_1 \circ f \circ \kappa_1.
\]

By Theorem 3 in [16] and the argument in the paper, \( \mathcal{D} \) has a uniform trace operator given by

\[
\text{tr}_{X,Y}^Z(f) := X \xrightarrow{(X, \infty)} X \oplus \bigoplus_{i \in \mathbb{N}} X \xrightarrow{X \oplus u_I} X \oplus Z \xrightarrow{f} Y \oplus Z \xrightarrow{\pi_0} Y
\]

where \( \infty : X \to \bigoplus_{i \in \mathbb{N}} X \) is the diagonal morphism, and \( u_I : \bigoplus_{i \in \mathbb{N}} X \to Z \) is the unique morphism such that \( u_I \circ \kappa_i = f_{ZZ}^I \circ f_{XZ} \) for each \( i \in \mathbb{N} \). By simple calculation, we see that the obtained trace operator is equal to the standard trace formula: \( \text{tr}_{X,Y}^Z(f) = f_{XY} + \sum_{i \in \mathbb{N}} f_{ZY} \circ f_{ZZ}^I \circ f_{XZ} \).

Corollary 5.4 Every strong unique decomposition category \( \mathcal{C} \) has a uniform partial trace operator. If the summation \( \text{Ex}_{X,Y}^Z(f) := f_{XY} + \sum_{i \in \mathbb{N}} f_{ZY} \circ f_{ZZ}^I \circ f_{XZ} \) is defined
for all $X, Y, Z \in \mathcal{C}$ and $f : X \otimes Z \to Y \otimes Z$, then $E_X$ is a uniform trace operator of $\mathcal{C}$.

**Proof.** By the above argument, $\mathcal{B}(T, \mathcal{C})$ has a uniform trace operator given by the standard trace formula. Since $KH : \mathcal{C} \to \mathcal{B}(T, \mathcal{C})$ is strong monoidal and reflects summability, $E_X$ provides a uniform partial trace operator of $\mathcal{C}$. If $E_X^Z(f)$ is defined for all $X, Y, Z \in \mathcal{C}$ and $f : X \otimes Z \to Y \otimes Z$, then by the definition of trace operators, $E_X$ is a trace operator of $\mathcal{C}$.  

**Acknowledgement**

The author would like to thank Masahito Hasegawa and Shin-ya Katsumata for comments and advice. The author also thank Esfandiar Haghverdi and Philip Scott for helpful comments and advice on the definition of unique decomposition categories.

**References**


Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming

Wolfgang Jeltsch

TTÜ Kü Btnetska Instituut
Tallinn, Estonia

Abstract
Linear-time temporal logic (LTL) and functional reactive programming (FRP) are related via a Curry–Howard correspondence. Based on this observation, we develop a common categorical semantics for a subset of LTL and its corresponding flavor of FRP. We devise a class of categorical models, called fan categories, that explicitly reflect the notion of time-dependent trueness of temporal propositions and a corresponding notion of time-dependent type inhabitation in FRP. Afterwards, we define the more abstract concept of temporal category by extending categorical models of intuitionistic S4. We show that fan categories are a special form of temporal categories.

Keywords: temporal logic, modal logic, functional reactive programming, categorical semantics

1 Introduction
It was shown recently that there is a Curry–Howard correspondence between linear-time temporal logic (LTL) and functional reactive programming (FRP) [6,7,5]. This suggests that LTL and FRP can be given a common semantics. Category theory has been proven useful for modeling logics and programming calculi. So our goal is to define a class of categorical structures that can serve as models for LTL and for a corresponding FRP dialect. This paper describes our first results in this direction. We present the following contributions:

• In Section 2, we develop a class of categorical models for an intuitionistic temporal logic with a “globally” and a “finally” modality. We call these categorical models

---

1 This work was partly supported by the ERDF through the Estonian Center of Excellence in Computer Science (EXCS).
2 Email: wolfgang@cs.ione.ee

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
fan categories. Fan categories directly reflect the fact that trueness of temporal formulas depends on the time.

- In Section 3, we demonstrate that fan categories are also models of FRP. We show that the time-dependent notion of trueness in temporal logic is related to time-dependent type inhabitance in FRP. We use our categorical semantics to explain the correspondence between the temporal modalities and the type constructors for behaviors and events, which are the key concepts of FRP.

- In Section 4, we define the notion of intuitionistic S4 category based on earlier work by Kobayashi [8] and Bierman and de Paiva [3]. We prove that fan categories are a special form of S4 categories.

- In Section 5, we introduce variants of the temporal modalities that refer only to the future. To reflect this in the semantics, we define ideal intuitionistic S4 categories. We prove a relationship between fan categories and ideal intuitionistic S4 categories that is analog to the result from Section 4.

- In Section 6, we extend ideal intuitionistic S4 categories with additional structure that captures the notion of linear time. We call the resulting structures temporal categories and prove that temporal categories cover fan categories as a special case.

We discuss related work in Section 7 and give conclusions and an outlook on further work in Section 8.

Throughout this paper, we will use certain notation when working with categorical products and coproducts. Let us define this notation, before starting with the payload of this paper.

**Definition 1.1 (Operations on products and coproducts)** For working with products and coproducts, we introduce notation as follows:

- Let $I$ be an index set, \( \{A_i\}_{i \in I} \) be a family of objects for which a product exists, and \( \{f_i\}_{i \in I} \) be a family of morphisms \( f_i : B \to A_i \). Then

\[
\langle \{f_i\}_{i \in I} \rangle : B \to \prod_{i \in I} A_i
\]

denotes the generalization of the binary product operation \( \langle \cdot, \cdot \rangle \) applied to the \( f_i \). Furthermore for any \( i \in I \), \( \pi_i \) denotes the projection that corresponds to \( i \).

- Expressions of the form \( \{\{f_i\}_{i \in I}\} \) and \( \iota_i \) for appropriate \( \{f_i\} \) and \( i \) denote the dual operations on coproducts.

- If we work in a bicartesian closed category (BCCC), then

\[
\sigma : B \times \prod_{i \in I} A_i \to \prod_{i \in I} (B \times A_i)
\]

denotes the natural transformation whose existence follows from the fact that BCCCs are distributive with respect to all coproducts. In the case of a binary coproduct, we add indices to \( \sigma \) that denote the objects involved, so that we have...
the following definition:

\[ \sigma_{A,B,C} : A \times (B + C) \to A \times B + A \times C \]
\[ \sigma_{A,B,C} := [\text{id}_A \times \iota_1, \text{id}_A \times \iota_2]^{-1} \]

2 Temporal Logic and Fan Categories

We consider a temporal logic with a linear notion of time and with \( \square \) ("globally") and \( \Diamond \) ("finally") as its only temporal operators. Since we want to have a Curry–Howard correspondence with FRP, our logic is intuitionistic instead of classical. Let \( P \) denote a set of atomic propositions. Then the syntax of formulas is given by the following BNF rule:

\[ F ::= P \mid \top \mid \bot \mid F \land F \mid F \lor F \mid F \to F \mid \square F \mid \Diamond F \]

In temporal logic, it depends on the time whether a formula is true or not. So intuitively, we can identify a formula \( \varphi \) of intuitionistic temporal logic with a function from times to formulas of intuitionistic propositional logic. We devise a class of categorical models that reflect this intuition. We call these models fan categories.

The standard categorical models of intuitionistic propositional logic are bicartesian closed categories (BCCCs). So we say that a fan category must be a product category \( C^T \) where \( C \) is a BCCC, and \( T \) is a set of times. An object of such a category is a function from \( T \) to \( \text{Obj} \( C \)), and a morphism \( f : A \to B \) is a function that maps each time \( t \) to a morphism \( f(t) : A(t) \to B(t) \). The latter means that for any temporal formulas \( \varphi \) and \( \psi \), a proof of \( \varphi \vdash \psi \) shows that \( \varphi(t) \vdash \psi(t) \) holds for all times \( t \).

The bicartesian closed structure of \( C \) gives rise to a bicartesian closed structure of \( C^T \), where operations of the latter are just pointwise applications of the respective operations of \( C \). For example, the product-related operations of \( C^T \) are defined as follows:

\[ (A \times B)(t) := A(t) \times B(t) \]
\[ \langle f, g \rangle(t) := \langle f(t), g(t) \rangle \]
\[ \pi_1(t) := \pi_1 \]
\[ \pi_2(t) := \pi_2 \]

Clearly, the bicartesian closed structure of \( C^T \) reflects the usual meanings of finite conjunctions, finite disjunctions, and implications in temporal logic.

For modeling the temporal modalities \( \square \) and \( \Diamond \), we equip our set \( T \) of times with a total order \( \leq \), from which we derive orders \( < \), \( \geq \), and \( > \) in the usual way. The intuition is that \( t < t' \) holds if at time \( t \), \( t' \) lies in the future. A formula \( \Box \varphi \) states that \( \varphi \) holds now and at every future time, while \( \Diamond \varphi \) states that \( \varphi \) holds now or at some future time. So a proposition \( (\Box \varphi)(t) \) corresponds to a (possibly infinite) conjunction of all \( \varphi(t') \) with \( t' \geq t \), while a proposition \( (\Diamond \varphi)(t) \) corresponds to a disjunction of all such \( \varphi(t') \).

Therefore, we model the modalities \( \Box \) and \( \Diamond \) by two functions \( \Box \) and \( \Diamond \) that
turn objects into objects such that for any object \( A \), the following holds:

\[
(\Box A)(t) = \prod_{t' \geq t} A(t') \quad (\Diamond A)(t) = \coprod_{t' \geq t} A(t')
\]

For this to work, we have to require that every family of objects of \( C \) that is indexed by a set \( \{ t' \mid t' \geq t \} \) admits a product and a coproduct. We actually demand a slightly stronger property, using index sets of the form \( \{ t' \mid t' > t \} \), because we will need this stronger property in Section 5. We are now ready to give the definition of a fan category.

**Definition 2.1 (Fan category)** Let \((T, \leq)\) be a totally ordered set, and let \( C \) be a BCCC where every family of objects indexed by a set \( \{ t' \mid t' > t \} \) has a product and a coproduct. The product category \( C^T \) is then called a fan category.

The object mappings \( \Box \) and \( \Diamond \) can be turned into functors by defining the lifting of morphisms in the natural way.

**Definition 2.2 (Temporal functors of a fan category)** For each fan category \( C^T \), the temporal functors \( \Box \) and \( \Diamond \) are defined such that for every morphism \( f \) and every \( t \in T \), the following equations hold:

\[
(\Box f)(t) = \prod_{t' \geq t} f(t') \quad (\Diamond f)(t) = \coprod_{t' \geq t} f(t')
\]

### 3 Connection to Functional Reactive Programming

The temporal logic we have defined in Section 2 corresponds to a type system for FRP \([6,7,5]\). Thereby, an FRP type corresponds to a temporal formula. Since such a formula can be seen as a function from times to formulas of intuitionistic propositional logic, an FRP type can be seen as a function from times to types of a simply typed \( \lambda \)-calculus with finite products and sums. So it depends on the time what values an FRP type inhabits.

Since fan categories are models of temporal logic, they are also models of FRP. If \( A \) is an object of a fan category that models an FRP type \( \tau \), and \( t \) is a time, \( A(t) \) is the meaning of \( \tau(t) \), that is, the simple type that corresponds to \( \tau \) at \( t \). If \( A \) and \( B \) model FRP types \( \tau_1 \) and \( \tau_2 \), a morphism from \( A \) to \( B \) models a family of functions from \( \tau_1 \) to \( \tau_2 \), one for each time.

The key constructs of FRP are behaviors and events, which are used to describe temporal phenomena. A behavior is a time-varying value, while an event is an occurrence time with an attached value. The temporal modality \( \Box \) corresponds to a type constructor \( \Box \) for behaviors, while the modality \( \Diamond \) corresponds to a type constructor \( \Diamond \) for events. This can be seen by looking at the endofunctors \( \Box \) and \( \Diamond \), which model the temporal modalities and hence also the type constructors that correspond to them. Remember that for any object \( A \), we defined \( \Box A \) and \( \Diamond A \) as
follows:

\[(\Box A)(t) := \prod_{t' \geq t} A(t')\] \[(\Diamond A)(t) := \bigvee_{t' \geq t} A(t')\]

So an FRP type \((\Box \tau)(t)\) corresponds to a (possibly infinite) product of all types \(\tau(t')\) with \(t' \geq t\). This means that an inhabitant of \((\Box \tau)(t)\) assigns a value of type \(\tau(t')\) to every time \(t' \geq t\) and thus characterizes a time-varying value of type \(\tau\). Likewise, a type \((\Diamond \tau)(t)\) corresponds to a sum of all types \(\tau(t')\) with \(t' \geq t\). So an inhabitant of \((\Diamond \tau)(t)\) is a pair of a time \(t' \geq t\) and a value of type \(\tau(t')\) and thus characterizes an occurrence time with an attached value of type \(\tau\).

4 Connection to Models of Intuitionistic S4

The classical modal logic S4 corresponds to the class of Kripke frames whose accessibility relation is a preorder. Classical temporal logics with a linear notion of time use totally ordered sets of times as Kripke frames. So a Kripke model for such a logic is also a Kripke model for S4. It is reasonable to assume that a similar connection exists in the case of intuitionistic logics and categorical models. In this section, we show that this is in fact the case.

Categorical models for intuitionistic S4 variants are studied by Kobayashi [8] as well as by Bierman and de Paiva [3]. We define the notion of intuitionistic S4 category based on their work and show that fan categories give rise to intuitionistic S4 categories.

Definition 4.1 (Cartesian comonad) Let \(\mathcal{C}\) be a category with finite products. A tuple \((U, \varepsilon, \delta, m, n)\) is a cartesian comonad on \(\mathcal{C}\) if \((U, \varepsilon, \delta)\) is a comonad on \(\mathcal{C}\), and \((U, m, n)\) is a cartesian endofunctor on \(\mathcal{C}\), that is, a strong monoidal functor from the monoidal category \((\mathcal{C}, \times, 1)\) to itself.

Definition 4.2 (\(U\)-strong monad) Let \(\mathcal{C}\) be a category with finite products and \(U = (U, \varepsilon, \delta, m, n)\) be a cartesian comonad on \(\mathcal{C}\). A tuple \((T, \eta, \mu, s)\) is a \(U\)-strong monad if \((T, \eta, \mu)\) is a monad on \(\mathcal{C}\), \(s\) is a natural transformation with \(s_{A,B} : UA \times TB \to T(UA \times B)\), and the diagrams in Figures 1 and 2 commute. The transformation \(s\) is called tensorial strength.

Definition 4.3 (Intuitionistic S4 category) An intuitionistic S4 category is a tuple \((\mathcal{C}, \Box, \varepsilon, \delta, m, n, \Diamond, \eta, \mu, s)\) where \(\mathcal{C}\) is a BCCC, \(U = (\Box, \varepsilon, \delta, m, n)\) is a cartesian comonad on \(\mathcal{C}\), and \((\Diamond, \eta, \mu, s)\) is a \(U\)-strong monad.

Kobayashi [8] defines the notion of CS4 structure, which is very similar to our notion of intuitionistic S4 category. The difference is that a CS4 structure may only have weak coproducts instead of proper coproducts, and that the functor \(\Diamond\) must preserve weak initial objects. Kobayashi probably needs this, because his logic has \(\Diamond \bot \to \bot\) as a theorem. In FRP terms, this would mean that there is a function of type \(\Diamond 0 \to 0\), that is, a function that can yield a non-existing value of type 0 now, although such a value is only promised to be available at some time that may
not have been reached yet. Clearly, such a function cannot exist. Since we want to maintain a Curry–Howard correspondence between temporal logic and FRP, we reject $\perp \rightarrow \perp$.

Bierman and de Paiva [3] define categorical models for the intuitionistic modal logic IS4. In contrast to us, they do not require the monoidal endofunctor $(\Box, m, n)$ to be strong. However, they enforce certain coherence conditions between the monoidal functor structure and the comonad structure, which hold automatically for a strong monoidal functor. In Section 10 of their paper, they discuss some possible extra conditions related to the maps $!_{\Box A} : \Box A \rightarrow 1$ and $\Delta_{\Box A} : \Box A \rightarrow \Box A \times \Box A$. These conditions also hold automatically if $(\Box, m, n)$ is strong, as is the case in our intuitionistic S4 categories. Furthermore, their coherence conditions for tensorial strength differ from what we have depicted in Figures 1 and 2.

We will now state and prove the relationship between fan categories and intuitionistic S4 categories that we mentioned at the beginning of this section.
\[ \varepsilon(t) : \prod_{t' \geq t} A(t') \to A(t) \quad \delta(t) : \prod_{t' \geq t} A(t') \to \prod_{t' \geq t} \prod_{t'' \geq t'} A(t'') \]
\[ \varepsilon(t) := \pi_t \quad \delta(t) := \langle \{ \{ \pi_{t''} \}_{t'' \geq t} \} \rangle \]

Fig. 3. Comonad structure of a fan category

\[ m(t) : \prod_{t' \geq t} A(t') \times \prod_{t' \geq t} B(t') \to \prod_{t' \geq t} (A(t') \times B(t')) \quad n(t) : 1 \to \prod_{t' \geq t} 1 \]
\[ m(t) := \langle \{ \pi_t \times \pi_t \}_{t' \geq t} \rangle \quad n(t) := \langle \{ \text{id}_1 \}_{t' \geq t} \rangle \]

Fig. 4. Cartesian endofunctor structure of a fan category

\[ \eta(t) : A(t) \to \prod_{t' \geq t} A(t') \quad \mu(t) : \prod_{t' \geq t} \prod_{t'' \geq t'} A(t'') \to \prod_{t' \geq t} A(t') \]
\[ \eta(t) := t_t \quad \mu(t) := \langle \{ [\{ t_{t''} \}_{t'' \geq t'}] \}_{t' \geq t} \rangle \]

Fig. 5. Monad structure of a fan category

\[ s(t) : \prod_{t' \geq t} A(t') \times \prod_{t' \geq t} B(t') \to \prod_{t' \geq t} \left( \prod_{t'' \geq t'} A(t'') \times B(t') \right) \]
\[ s(t) := \left( \prod_{t' \geq t} \langle \{ \pi_{t''} \}_{t'' \geq t'} \times \text{id}_{B(t')} \rangle \right) \sigma \]

Fig. 6. Tensorial strength of a fan category

**Theorem 4.4** If \( C^T \) is a fan category, and \( \Box \) and \( \Diamond \) are its temporal functors, then there are natural transformations \( \varepsilon, \delta, m, n, \eta, \mu, s \) such that \( (C, \Box, \varepsilon, \delta, m, n, \Diamond, \eta, \mu, s) \) is an intuitionistic S4 category.

**Proof.** We construct the abovementioned natural transformations as shown in Figures 3 through 6. Proving that these transformations fulfill the necessary conditions is straightforward and therefore left out here. \( \square \)

5 **Future Only**

A proposition \( \Box \varphi \) of our temporal logic forces \( \varphi \) to hold also at the current time. Likewise, a proposition \( \Diamond \varphi \) allows \( \varphi \) to hold at the current time instead of in the future. However, there are cases where modalities that only refer to the future are desired. In LTL, where we have a discrete notion of time and a “next” modality \( \Diamond \), we can define future-only variants of \( \Box \) and \( \Diamond \) as follows:

\[ \Box' \varphi := \Diamond \Box \varphi \quad \Diamond' \varphi := \Box \Diamond \varphi \]

In our logic, where time is not necessarily discrete, it is not possible to derive \( \Box' \) and \( \Diamond' \) from \( \Box \) and \( \Diamond \). So it is worthwhile to introduce \( \Box' \) and \( \Diamond' \) as the
fundamental modalities and define $\Box$ and $\diamond$ in terms of them as follows:

$$
\Box \varphi := \varphi \land \Box' \varphi \\
\diamond \varphi := \varphi \lor \diamond' \varphi
$$

This increases the expressiveness of our logic. Expressiveness of FRP can be increased in an analog way. In the next section, we will use the additional expressiveness that $\diamond'$ gives us.

We define a variant of intuitionistic S4 categories that also models the future-only modalities. We introduce two new endofunctors $\Box'$ and $\diamond'$, and derive $\Box$ and $\diamond$ from them as follows:

$$
\Box A := A \times \Box' A \\
\diamond A := A + \diamond' A
$$

According to Definition 4.3, we need a comonad structure for $\Box$ and a monad structure for $\diamond$. The natural way to get these is to add an ideal comonad structure for $\Box'$ and an ideal monad structure for $\diamond'$.

**Definition 5.1 (Ideal comonad)** A pair $(U', \delta')$ is an ideal comonad on a category $C$ with binary products if $U'$ is an endofunctor on $C$, $\delta'$ is a natural transformation from $U'$ to $U'(\text{Id} \times U')$, and $(\text{Id} \times U', \pi_1, (\text{id}, \delta' \pi_2))$ is a comonad.

**Definition 5.2 (Ideal monad)** A pair $(T', \mu')$ is an ideal monad on a category $C$ with binary coproducts if $T'$ is an endofunctor on $C$, $\mu'$ is a natural transformation from $T'(\text{Id} + T')$ to $T'$, and $(\text{Id} + T', \iota_1, [\text{id}, \iota_2 \mu'])$ is a monad.

From an ideal comonad $(\Box', \delta')$ and an ideal monad $(\diamond', \mu')$, we can derive the comonad $(\Box, \varepsilon, \delta)$ and the monad $(\diamond, \eta, \mu)$ that we need for an intuitionistic S4 category. We also want to derive the natural transformations $m, n,$ and $s$ from more basic transformations that work with $\Box'$ and $\diamond'$. For this, we introduce the two new concepts of ideal cartesian comonad (Definition 5.3) and $U'$-strong ideal monad (Definition 5.5).

**Definition 5.3 (Ideal cartesian comonad)** Let $C$ be a category with finite products. A tuple $(U', \delta', m', n')$ is an ideal cartesian comonad on $C$ if $(U', \delta')$ is an ideal comonad on $C$, and $(U', m', n')$ is a cartesian endofunctor on $C$.

**Lemma 5.4** If $(U', \delta', m', n')$ is an ideal cartesian comonad on a category $C$ with finite products, then

$$(\text{Id} \times U', \pi_1, (\text{id}, \delta' \pi_2), (\pi_1 \times \pi_1, m'(\pi_2 \times \pi_2)), (\text{id}_1, n'))$$

is a cartesian comonad on $C$.

**Proof.** $(\text{Id} \times U', \pi_1, (\text{id}, \delta' \pi_2))$ is a comonad on $C$ according to Definition 5.1. Checking that $(\text{Id} \times U', (\pi_1 \times \pi_1, m'(\pi_2 \times \pi_2)), (\text{id}_1, n'))$ is a cartesian endofunctor is straightforward.

**Definition 5.5 ($U'$-strong ideal monad)** Let $C$ be a distributive category, $U' = (U', \delta', m', n')$ be an ideal cartesian comonad on $C$ and $\mathcal{U} = (U, \varepsilon, \delta, m, n)$ be the
cartesian comonad induced by \( \mathcal{U}' \) according to Lemma 5.4. A tuple \((T', \mu', s')\) is a \( \mathcal{U}'\)-strong ideal monad if the following conditions hold:

- \((T', \mu')\) is an ideal monad on \( \mathcal{C} \).
- \(s'\) is a natural transformation with \( s'_{A,B} : U' A \times T' B \to T'(U A \times B) \).
- If \((T, \eta, \mu)\) is the monad induced by \((T', \mu')\) according to Definition 5.2, and the natural transformation \( s \) is defined by

\[
s_{A,B} : U A \times T B \to T(U A \times B) \\
s_{A,B} := (\text{id}_{U A \times B} + s'_{A,B}(\pi_2 \times \text{id}_{T' B})) \sigma_{U A, T B} ,
\]

then \((T, \eta, \mu, s)\) is an \( \mathcal{U}\)-strong monad.

**Definition 5.6 (Ideal intuitionistic S4 category)** An ideal intuitionistic S4 category is a tuple \((\mathcal{C}, \Box', \delta', m', n', \Diamond', \mu', s')\) where \( \mathcal{C} \) is a BCCC, \( \mathcal{U}' = (\Box', \delta', m', n') \) is an ideal cartesian comonad on \( \mathcal{C} \), and \((\Diamond', \mu', s')\) is a \( \mathcal{U}'\)-strong ideal monad.

From Lemma 5.4 and Definition 5.5, it is immediately clear that every ideal intuitionistic S4 category gives rise to an intuitionistic S4 category. Another important property is that fan categories give rise to ideal intuitionistic S4 categories. We define the ideal temporal functors of a fan category analogously to Definition 2.2 and obtain a fact similar to the one of Theorem 4.4.

**Definition 5.7 (Ideal Temporal Functors of a Fan Category)** Let \( \mathcal{C}^T \) be a fan category. The ideal temporal functors \( \Box' \) and \( \Diamond' \) of \( \mathcal{C}^T \) are defined such that for every morphism \( f \) and every \( t \in T \), the following equations hold:

\[
(\Box' f)(t) = \prod_{t' > t} f(t') \\
(\Diamond' f)(t) = \prod_{t' > t} f(t')
\]

**Theorem 5.8** If \( \mathcal{C}^T \) is a fan category, and \( \Box' \) and \( \Diamond' \) are its ideal temporal functors, then there are natural transformations \( \delta', m', n', \mu' \), and \( s' \) such that \((\mathcal{C}, \Box', \delta', m', n', \Diamond', \mu', s')\) is an ideal intuitionistic S4 category.

**Proof.** We define the natural transformations \( \delta', m', n', \mu' \), and \( s' \) by taking the definitions of \( \delta, m, n, \mu \), and \( s \) from the proof of Theorem 4.4 and replacing \( \geq \) by \( > \) wherever we now deal with \( \Box' \) and \( \Diamond' \) instead of \( \Box \) and \( \Diamond \). In the following, we show that these definitions lead in fact to an ideal intuitionistic S4 category.

We derive functors \( \Box \) and \( \Diamond \) and natural transformations \( \varepsilon, \delta, m, n, \eta, \mu, \) and \( s \) from \( \Box', \Diamond', \delta', m', n', \mu' \), and \( s' \) according to Lemma 5.4 and Definitions 5.2 and 5.5. The functors \( \Box \) and \( \Diamond \) are isomorphic to the temporal functors from Definition 2.2, and the natural transformations are the ones defined in the proof of Theorem 4.4 up to isomorphism. So they form an intuitionistic S4 category. This means that \( \mathcal{U} = (\Box, \varepsilon, \delta, m, n) \) is a cartesian comonad, and \((\Diamond, \eta, \mu, s)\) is a \( \mathcal{U}\)-strong monad. As a result, \( \mathcal{U}' = (\Box', \delta', m', n') \) is an ideal cartesian comonad, and \((\Diamond', \mu', s')\) is a \( \mathcal{U}'\)-strong ideal monad. This proves the claim. \( \square \)
6 Linear Time

Fan categories are rather concrete. In this section, we develop a much more abstract notion of categorical model for temporal logic and FRP. Ideal intuitionistic S4 categories are a good starting point for this undertaking. Their problem is that they do not capture the notion of linear time. This is analogous to Kripke models of classical logics, where S4 permits arbitrary preorders as Kripke frames, while linear-time temporal logics only permit total orders. In this section, we enrich ideal intuitionistic S4 categories with further structure that reflects the linearity of time. We call the resulting constructs temporal categories.

In order to see how we can encode linearity of time in our categorical models, let us first look at FRP. Linearity of time is ensured if there is a function $\text{race}$ of type

$$\Diamond \tau_1 \times \Diamond \tau_2 \rightarrow \Diamond (\tau_1 \times \tau_2 + \tau_1 \times \Diamond' \tau_2 + \Diamond' \tau_1 \times \tau_2).$$

In the following, we will explain why this is the case.

Let $e_1$ and $e_2$ be events of types $\Diamond \tau_1$ and $\Diamond \tau_2$, and let $t_1$, $t_2$, and $t$ be the times at which $e_1$, $e_2$, and $\text{race}(e_1, e_2)$ fire. If $\text{race}(e_1, e_2)$ contains a value of type $\tau_1 \times \tau_2$, the components of this pair must come from $e_1$ and $e_2$, because the values that $e_1$ and $e_2$ carry are the only values of types $\tau_1$ and $\tau_2$ that are available to $\text{race}$. Since our FRP dialect generally does not allow us to shift values to different times, we have $t = t_1 = t_2$. If $\text{race}(e_1, e_2)$ contains a value of type $\tau_1 \times \Diamond' \tau_2$ or $\Diamond' \tau_1 \times \tau_2$, we get a remainder event of a type $\Diamond' \tau_i$, which fires after $t$. Since it contains a value of type $\tau_i$, it fires at $t_i$. So the second and the third alternative correspond to the conditions $t = t_1 < t_2$ and $t = t_2 < t_1$, respectively. All in all, we now that one of the three alternatives $t_1 = t_2$, $t_1 < t_2$, and $t_1 > t_2$ holds, which ensures that time is linear. We furthermore know that $\text{race}(e_1, e_2)$ fires at time $\min(t_1, t_2)$.

Let us now turn to category theory again. Say we have an ideal intuitionistic S4 category $(\mathcal{C}, \sqcup', \delta', m', n', \Diamond', \mu', s')$, which induces an intuitionistic S4 category $(\mathcal{C}, \Box, \varepsilon, \delta, m, n, \Diamond, \eta, \mu, s)$. We define a binary operation $\odot$ on objects as follows:

$$A \odot B := A \times B + A \times \Diamond' B + \Diamond' A \times B$$

To give a meaning to $\text{race}$, we require that for any morphisms $f : C \rightarrow \Diamond A$ and $g : C \rightarrow \Diamond B$, there is a morphism $\langle f, g \rangle : C \rightarrow \Diamond (A \odot B)$. We realize this by requiring that for any objects $A$ and $B$, $A \odot B$ is a product of $A$ and $B$ in the Kleisli category of the monad $(\Diamond, \eta, \mu)$.

For a proper product structure, we also need projections, which we call $\pi_1$ and $\pi_2$ in order to not confuse them with the projections $\pi_1$ and $\pi_2$ of the original category $\mathcal{C}$. The projections $\pi_i$ have the types $C_1 \odot C_2 \rightarrow C_i$ in the Kleisli category. So they have the types $C_1 \odot C_2 \rightarrow \Diamond C_i$ in the original category, which are the same as

$$C_1 \times C_2 + C_1 \times \Diamond' C_2 + \Diamond' C_1 \times C_2 \rightarrow C_i + \Diamond C_i.$$

The straightforward definition of the $\pi_i$ is $\pi_i := [t_1 \pi_i, t_2 \pi_i, t_1 \pi_i]$. 

224
For $\odot, \langle\cdot,\cdot\rangle, \varpi_1$, and $\varpi_2$ to form a product, the following equations must hold in the Kleisli category for all suitable $h_1, h_2, h$, and $i$:

$$\varpi_i(h_1, h_2) = h_i \quad \langle\varpi_1 h, \varpi_2 h\rangle = h$$

This means that the following equations must hold in the original category $C$:

$$\mu(\odot \varpi_i)(h_1, h_2) = h_i \quad \langle\mu(\odot \varpi_1)h, \mu(\odot \varpi_2)h\rangle = h$$

Looking at FRP, the first equation tells us that we can recover the events $e_i$ from a value $\text{race}(e_1, e_2)$ using functions $\text{recover}_i$ that correspond to the transformations $\mu(\odot \varpi_i)$. The second equation states that every value $e$ from the codomain of $\text{race}$ can be constructed by applying $\text{race}$ to $(\text{recover}_1 e, \text{recover}_2 e)$.

We now define temporal categories by extending ideal intuitionistic S4 categories as described above.

**Definition 6.1 (Temporal category)** Say $M = (C, \Box', \delta', m', n', \odot', \mu', s')$ is an ideal intuitionistic S4 category, and $(C, \Box, \varepsilon, \delta, m, n, \odot, \eta, \mu, s)$ is the intuitionistic S4 category induced by it. For all objects $A$ and $B$, let $A \odot B$ be defined by

$$A \odot B := A \times B + A \times \odot' B + \odot' A \times B,$$

and for all objects $C_1$ and $C_2$ and all $i \in \{1, 2\}$, let $\varpi_i$ be defined as follows:

$$\varpi_i : C_1 \odot C_2 \rightarrow \odot C_i$$

$$\varpi_i := [t_1 \pi_i, t_i \pi_i, t_{1-i} \pi_i]$$

$M$ is a temporal category if each $A \odot B$ is a product of $A$ and $B$ with projections $\varpi_1$ and $\varpi_2$ in the Kleisli category of $(\odot, \eta, \mu)$.

**Definition 4.2** enforces certain relationships between a strength transformation $s$ and other natural transformations, which are depicted in Figures 1 and 2. The reader might wonder why we did not specify similar relationships between $\langle\cdot,\cdot\rangle$ and the transformations $\mu'$ and $s'$ in **Definition 6.1**. We did not do so, since we strongly conjecture that any such coherence conditions that are sensible already follow from **Definition 6.1** as it is. Making this claim precise and proving it is a possible goal for the future.

Our final result is that temporal categories are indeed a generalization of fan categories.

**Theorem 6.2** If $C_T$ is a fan category, and $\Box'$ and $\odot'$ are its ideal temporal functors, then there are natural transformations $\delta', m', n', \mu'$, and $s'$ such that $(C, \Box', \delta', m', n', \odot', \mu', s')$ is a temporal category.

**Proof.** We construct the abovementioned natural transformations like we did in the proof of Theorem 5.8. So we know that $(C, \Box', \delta', m', n', \odot', \mu', s')$ is an ideal intuitionistic S4 category.

225
We now define the operation $\langle\langle \cdot, \cdot \rangle\rangle$. We first introduce a helper morphism $\theta_t$ for every time $t$:

$$\theta_t : \coprod_{t_1 \geq t} A(t_1) \times \coprod_{t_2 \geq t} B(t_2) \to \coprod_{t_1 \geq t} \coprod_{t_2 \geq t} (A(t_1) \times B(t_2))$$

$$\theta_t := \left( \prod_{t_1 \geq t} \sigma(\pi_2, \pi_1) \right) \sigma(\pi_2, \pi_1)$$

We furthermore define transformations $\kappa_{t,t_1,t_2}$ for times $t$, $t_1$, and $t_2$ with $t \leq t_1$ and $t \leq t_2$:

$$\kappa_{t,t_1,t_2} : A(t_1) \times B(t_2) \to \coprod_{t' \geq t} (A \odot B)(t')$$

$$\kappa_{t,t_1,t_2} := \begin{cases} 
  \iota_{t_1} \iota_{t_2} & \text{if } t_1 = t_2 \\
  \iota_{t_1} \iota_{t_2} (\iota_{t_1} \times \iota_{t_2}) & \text{if } t_1 < t_2 \\
  \iota_{t_2} \iota_{t_3} (\iota_{t_1} \times \iota_{t_2}) & \text{if } t_1 > t_2 
\end{cases}$$

Finally, we define $\langle\langle f, g \rangle\rangle$ for any morphisms $f : C \to \Diamond A$ and $g : C \to \Diamond B$ as follows:

$$\langle\langle f, g \rangle\rangle(t) : C(t) \to \coprod_{t' \geq t} (A \odot B)(t')$$

$$\langle\langle f, g \rangle\rangle(t) := [\{\{\kappa_{t,t_1,t_2}\}_{t_2 \geq t}\}_{t_1 \geq t} \theta_t(f(t), g(t))$$

We leave it as an exercise to the reader to show that in the Kleisli category, $\varpi_1 \langle\langle h_1, h_2 \rangle\rangle = h_1$ and $\langle\langle \varpi_1 h, \varpi_2 \rangle\rangle = h$ hold for all suitable $h_1$, $h_2$, and $h$. This then completes the proof. \qed

### 7 Related Work

Jeffrey [5] presents an implementation of FRP in the dependently typed programming language Agda. Based on this, he develops a category $\textbf{RSet}$, which expresses the notion of time-dependent type inhabitation and is thus strongly related to our fan categories. Jeffrey also uses FRP analogs of advanced temporal operators to develop variants of $\textbf{RSet}$ that enforce causality of FRP operations.

In Section 4, we discussed the relationships between our intuitionistic S4 categories and the categorical models by Kobayashi [8] and by Bierman and de Paiva [3]. Alechina et. al. [1] show how the latter are related to algebraic models and Kripke models.\footnote{Note that there is a slight confusion in terminology. Kobayashi calls his S4 variant CS4, Bierman and de Paiva call theirs IS4, but Alechina et. al. use the name CS4 for the logic of Bierman and de Paiva.}

Bellin et. al. [2] study an intuitionistic version IK of the basic modal logic K. They also define categorical models of IK. These models lack the comonadic and monadic structure that intuitionistic S4 categories possess, and use a tensorial...
strength transformation $t$ with

$$t_{A,B} : \Box A \times \Diamond B \to \Diamond (A \times B)$$

instead of the strength transformation $s$ with

$$s_{A,B} : \Box A \times \Diamond B \to \Diamond (\Box A \times B) .$$

In intuitionistic S4 categories, we can derive $t$ from $s$ by $t := (\Diamond (\varepsilon_A \times \text{id}_B))s$. As a result, every model of IK is also an intuitionistic S4 category. The lack of structure in IK models corresponds to a lack of axioms in logic. Classically, this lack of axioms corresponds to the fact that K corresponds to the class of all Kripke frames, while S4 corresponds to the class of Kripke frames where the accessibility relation is a preorder.\footnote{Note that the axiom $\Box \varphi \rightarrow \varphi$ ensures reflexivity, and the axiom $\Diamond \varphi \rightarrow \Box \Diamond \varphi$ ensures transitivity. Analogously, $\varphi \rightarrow \Diamond \varphi$ ensures reflexivity, and $\Diamond \Box \varphi \rightarrow \Diamond \varphi$ ensures transitivity.}

Krishnaswami and Benton [9] give an FRP semantics based on the category of 1-bounded ultrametric spaces, and Birkedal et al [4] study the related category of presheaves over the natural numbers. Both approaches use a discrete notion of time, while our work is compatible with any totally ordered set of times. Studying the connections between our developments and the ones of Krishnaswami and Benton as well as the ones of Birkedal et al. remains a task for the future.

8 Conclusions and Further Work

We have defined fan categories, which are categorical models of a subset of an intuitionistic LTL variant and a corresponding flavor of FRP. Fan categories directly express the notion of time-dependent trueness in LTL and the related notion of time-dependent type inhabitance of our FRP dialect. We have furthermore defined the more abstract notion of temporal category based on categorical models of intuitionistic S4 and shown that fan categories are a specialization of temporal categories.

In a future publication, we want to extend temporal categories such that they also cover other modalities of LTL and their FRP counterparts. Furthermore, we want to study how recursion can be integrated into categorical models of FRP. We also want to use concepts from temporal categories in the interface design and possibly the implementation of FRP systems. Furthermore, we are interested in combining temporal logic with other kinds of logic and studying the corresponding programming paradigms. Another task is to find out about relationships to other categorical FRP semantics, as discussed in Section 7.

Acknowledgement

I thank Tarmo Uustalu for the various discussions about the topics of this work. Furthermore, I thank Alan Jeffrey and the anonymous reviewers for giving me helpful
feedback on earlier versions of this paper.

References


Abstract

Polynomial functors (over $\text{Set}$ or other locally cartesian closed categories) are useful in the theory of data types, where they are often called containers. They are also useful in algebra, combinatorics, topology, and higher category theory, and in this broader perspective the polynomial aspect is often prominent and justifies the terminology. For example, Tambara’s theorem states that the category of finite polynomial functors is the Lawvere theory for commutative semirings [44], [18].

In this talk I will explain how an upgrade of the theory from sets to groupoids (or other locally cartesian closed 2-categories) is useful to deal with data types with symmetries, and provides a common generalisation of and a clean unifying framework for quotient containers (in the sense of Abbott et al.), species and analytic functors (Joyal 1985), as well as the stuff types of Baez and Dolan. The multi-variate setting also includes relations and spans, multispan, and stuff operators. An attractive feature of this theory is that with the correct homotopical approach — homotopy slices, homotopy pullbacks, homotopy colimits, etc. — the groupoid case looks exactly like the set case.

After some standard examples, I will illustrate the notion of data-types-with-symmetries with examples from perturbative quantum field theory, where the symmetries of complicated tree structures of graphs play a crucial role, and can be handled elegantly using polynomial functors over groupoids. (These examples, although beyond species, are purely combinatorial and can be appreciated without background in quantum field theory.)

Locally cartesian closed 2-categories provide semantics for a 2-truncated version of Martin-Löf intensional type theory. For a fullledged type theory, locally cartesian closed $\infty$-categories seem to be needed. The theory of these is being developed by David Gepner and the author as a setting for homotopical species, and several of the results exposed in this talk are just truncations of $\infty$-results obtained in joint work with Gepner. Details will appear elsewhere.

Keywords: Polynomial functors, groupoids, data types, symmetries, species, trees.

1 Polynomial functors over $\text{Set}$ and data types

1.1 Polynomial functors in one variable. In its simplest form, a polynomial
functor is an endofunctor of \textbf{Set} of the form

\[ X \mapsto \sum_{b \in B} X^{E_b}. \tag{1} \]

Here the sum sign is disjoint union of sets, \( X^{E_b} \) denotes the hom set \( \text{Hom}(E_b, X) \), and \((E_b \mid b \in B)\) is a \( B \)-indexed family of sets, encoded conveniently as a single map of sets \( E \to B \).

Viewed as a data type constructor, \( E \to B \) is often called a container \([1,2,3,4,5,7]\); then \( B \) is regarded as a set of shapes, and the fibre \( E_b \) is the set of positions in the shape corresponding to \( b \). The data to be inserted into these positions can be of any type \( X \): the polynomial functor receives a type \( X \) (a set) and returns the new more elaborate type \( \sum X^{E_b} \). Polymorphic functions correspond to natural transformations of polynomial functors, and these can be handled in terms of the representing sets \( E \to B \) alone, cf. [1], [18], and 2.6 below. A fundamental example is the list functor, \( X \mapsto \sum_{n \in \mathbb{N}} X^n \), which to a set \( X \) associates the set of lists of elements in \( X \). Here \( n \in \mathbb{N} \) is the shape, and \( n \) denotes the \( n \)-element set \( \{0, 1, \ldots, n - 1\} \) of positions in a length-\( n \) list.

There is another important use of polynomial functors in type theory: one then regards \( E \to B \) as a signature generating an algebra, namely the initial algebra for the polynomial functor. Initial algebras for polynomial functors are inductive data types, corresponding to \( W \)-types in (extensional) Martin-Löf type theory \([41], [39]\). Similarly, terminal coalgebras are coinductive data types (sometimes called \( M \)-types), often interpreted as programs or systems (see for example \([42], [22]\)).

1.2 \textbf{Species and analytic functors.} A functor is \textit{finitary} when it preserves \( \omega \)-filtered colimits. For a polynomial functor this is equivalent to \( E \to B \) having finite fibres. Let \( \mathbb{B}_\omega \) denote the groupoid of finite sets and bijections. A \textit{species} \([25]\) is a functor \( F : \mathbb{B}_\omega \to \textbf{Set} \), written \( S \mapsto F[S] \); the set \( F[S] \) is to be thought of as the set of \( F \)-structures that can be put on the set \( S \). The \textit{extension} of \( F \) is the endofunctor

\[ \textbf{Set} \to \textbf{Set} \]

\[ X \mapsto \sum_{n \in \mathbb{N}} \frac{F[n] \times X^n}{\text{Aut}(n)} \]

which is the left Kan extension of \( F \) along the (non-full) inclusion \( \mathbb{B}_\omega \subset \textbf{Set} \).

A functor of this form is called \textit{analytic} \([26]\). Joyal established an equivalence of categories between species and analytic functors, and characterised analytic functors as the finitary functors preserving cofiltered limits and weak pullbacks \([26]\), see also \([23]\) and \([6]\). Finitary polynomial functors are precisely the analytic functors which preserve pullbacks strictly. In terms of species they correspond to those for which the symmetric group actions are free.

Monoids in species (under the operation of substitution, which corresponds to composition of analytic functors) are precisely operads. Many important polynomial functors have the structure of monad. For example, the list functor has a natural monad structure by concatenation of lists. Polynomial monads equipped
with a cartesian monad map to the list monad are the same thing as non-symmetric operads [36]. More generally, finitary polynomial monads correspond to projective operads [31] (i.e. such that every epi to it splits).

1.3 Polynomial functors in many variables. Following [18], a polynomial is a diagram of sets

\[ I \leftarrow E \xrightarrow{p} B \xrightarrow{r} J, \]

and the associated polynomial functor (or the extension of the polynomial) is given by the composite

\[ \text{Set}_I \xrightarrow{\Delta_s} \text{Set}_E \xrightarrow{\Pi_p} \text{Set}_B \xrightarrow{\Sigma_t} \text{Set}_J, \]

where \( \Delta_s \) is pullback along \( s \), \( \Pi_p \) is the right adjoint to pullback (called dependent product), and \( \Sigma_t \) is left adjoint to pullback (called dependent sum). For a map \( f : B \rightarrow A \) we have the three explicit formulae

\[
\begin{align*}
\Delta_f(X_a \mid a \in A) &= (X_{f(b)} \mid b \in B) \\
\Sigma_f(Y_b \mid b \in B) &= (\sum_{b \in B_a} Y_b \mid a \in A) \\
\Pi_f(Y_b \mid b \in B) &= (\prod_{b \in B_a} Y_b \mid a \in A),
\end{align*}
\]

giving altogether the following formula for (4)

\[ (X_i \mid i \in I) \mapsto (\sum_{b \in B_j} \prod_{e \in E_b} X_{s(e)} \mid j \in J), \]

which specialises to (1) when \( I = J = 1 \).

The multi-variate polynomial functors correspond to indexed containers [7], and their initial algebras are sometimes called general tree types [40, Ch. 16].

From the abstract description in terms of adjoints, it follows that the notion of polynomial functor (and most of the theory) makes sense in any locally cartesian closed category, and polynomial functors are the most natural class of functors between slices of such categories. They have been characterised intrinsically [30] as the local fibred right adjoints.

1.4 Incorporating symmetries. A container is a rigid data structure: it does not allow for data to be permuted in any way among the positions of a given shape. In many situations it is desirable to allow for permutation, so that certain positions within a shape become indistinguishable. In quantum physics, the principle of indistinguishable particles imposes such symmetry at a fundamental level. A fundamental example is the multiset data type, whose extension is the functor

\[ X \mapsto \sum_{n \in \mathbb{N}} \frac{X^n}{\text{Aut}(n)}, \]

which is analytic but not polynomial.
In order to account for such data types with symmetries, Abbott et al. [5] (see also Gylterud [21]) have extended the container formalism by adding the symmetries ‘by hand’: for each shape (element $b$ in $B$) there is now associated a group of symmetries of the fibre $E_b$, and data inserted into the corresponding positions is quotiented out by this group action. It is not difficult to see (cf. also [6]) that in the finitary case, this is precisely the notion of species and analytic functors.

In fact it has been in the air for some time (see for example [14], and more recently [12], [46]) that species should be a good framework for data type theory. It is the contention of the present contribution that polynomial functors over groupoids provide a clean unifying framework: in the setting of groupoids, the essential distinction between ‘analytic’ and ‘polynomial’ evaporates (3.7), and the functors can be represented by diagrams with combinatorial content (3) just as polynomials over sets, as we proceed to explain.

¿From the viewpoint of species, there are other reasons for this upgrade anyway. In fact, it was soon realised by combinatorists that the 1985 notion of analytic functors is not optimal for enumerative purposes: taking cardinality simply does not yield the exponential generating functions central to enumerative combinatorics! (It does so if the analytic functor is polynomial.) In fact, the Species Book [11] does not mention analytic functors at all.

The issue was sorted out by Baez and Dolan [9]: the problem is that dividing out by the group action in (2) is a bad quotient from the viewpoint of homotopy theory, and does not behave well with respect to cardinality. To get the correct cardinalities, it is necessary to use homotopy quotients, and the result is then no longer a set but a groupoid, and the cardinality has to be homotopy cardinality. So it is necessary to work from the beginning with groupoids instead of sets. Baez and Dolan introduced species in groupoids (3.6), dubbing them stuff types, showed that homotopy cardinality gives the correct generating functions, and illustrated the usefulness of the broader generality by showing how the types needed for a combinatorial description of the quantum harmonic oscillator are stuff types, not classical species [9].

Joint work with David Gepner closes the circle by observing that over groupoids, species/analytic functors are the same thing as discrete finitary polynomial functors (3.7); hence the neat formalism of polynomials provides a natural unifying framework for (quotient) containers and species.

2 Polynomial functors over groupoids

A groupoid is a category in which all arrows are invertible. A useful intuition for the present purposes is that groupoids are ‘sets fattened with symmetries’. ¿From the correct homotopical viewpoint groupoids behave very much like sets. We are interested in groupoids up to equivalence, and for this reason many familiar 1-categorical notions, such as pullback and fibre, are not appropriate, as they are not invariant under equivalence. The good notions are the corresponding homotopy notions, which we briefly recall. They can all be deduced from the beautiful simplicial machinery
developed by Joyal [27,28] to generalise the theory of categories to quasi-categories (called $\infty$-categories by Lurie [37]). Since the 2-category $\text{Grpd}$ of groupoids has only invertible 2-cells, it is an example of a quasi-category. From now on when we say 2-category we shall mean ‘2-category with only invertible 2-cells’.

2.1 Slices. If $I$ is a groupoid, the homotopy slice $\text{Grpd}/I$ is the 2-category of projective cones with base $I$ (cf. [27]): its objects are maps $X \to I$; its arrows are triangles with a 2-cell

$$X \xrightarrow{\alpha} Y \xrightarrow{\beta} I$$

and 2-arrows are diagrams

$$X \xrightarrow{\alpha} Y \xrightarrow{\beta} I$$

commuting with the structure triangles. More generally, if $d : T \to \text{Grpd}$ is any diagram, there is a 2-category $\text{Grpd}_{/d}$ of projective cones with base $d$.

A homotopy terminal object in a 2-category $\mathcal{C}$ is an object $t$ such that for any other object $x$, the groupoid $\mathcal{C}(x,t)$ is contractible, i.e. equivalent to a point. More general homotopy limits are defined in the usual way using homotopy slices: the homotopy limit of a functor $d : T \to \text{Grpd}$ is by definition a homotopy terminal object in the homotopy slice $\text{Grpd}_{/d}$. Homotopy limits are unique up to equivalence.

2.2 Pullbacks and fibres. Given a diagram of groupoids $X,Y,S$ indicated by the solid arrows,

$$X \times_S Y \xrightarrow{f} Y \xrightarrow{g} S$$

the homotopy pullback is the homotopy limit, i.e. given as a homotopy terminal object in a certain slice 2-category of projective cones over the solid diagrams of the shape in question, and as such it is determined uniquely up to equivalence. A specific model is the groupoid $X \times_S Y$ whose objects are triples $(x,y,\phi)$ with $x \in X$, $y \in Y$ and $\phi : fx \to gy$ an arrow of $S$, and whose arrows are pairs $(\alpha,\beta) : (x,y,\phi) \to (x',y',\phi')$ consisting of $\alpha : x \to x'$ an arrow in $X$ and $\beta : y \to y'$ an arrow in $Y$ such that the following diagram commutes in $S$

$$\begin{array}{ccc}
fx & \xrightarrow{\phi} & gy \\
\downarrow{f(\alpha)} & & \downarrow{g(\beta)} \\
fx' & \xrightarrow{\phi'} & g'y'.
\end{array}$$

(One should note that if $f$ or $g$ is a fibration then the naïve set-theoretic pullback is equivalent to the homotopy pullback.)

The homotopy fibre $E_b$ of a morphism $p : E \to B$ over an object $b$ in $B$ is the homotopy pullback of $p$ along the inclusion map $1 \xrightarrow{b} B$:
(Note that the homotopy fibre $E_b$ is not in general a subgroupoid of $E$, although the map $E_b \to E$ is always faithful. But again, if $p$ is a fibration then the set-theoretic fibre is equivalent to the homotopy fibre.)

### 2.3 Homotopy quotients

Whenever a group $G$ acts on a set or a groupoid $X$, the **homotopy quotient** $X/G$ is the groupoid obtained by gluing in a path (i.e. an arrow) between $x$ and $y$ for each $g \in G$ such that $gx = y$. More formally it is the total space of the Grothendieck construction of the presheaf $G \to \text{Grpd}$ that the action constitutes; it is a special case of a homotopy colimit. (The notation $X//G$ is often used [9].) If $G$ acts on the point groupoid $1$, then $1/G$ is the groupoid with one object and vertex group $G$.

If $p : X \to B$ is a morphism of groupoids, for $b \in B$ the `inclusion' of the homotopy fibre $X_b \to X$ is faithful but not full in general. But $\text{Aut}(b)$ acts on $X_b$ canonically, and the homotopy quotient $X_b/\text{Aut}(b)$ provides exactly the missing arrows, so as to make the natural map $X_b/\text{Aut}(b) \to X$ fully faithful. Since every object $x \in X$ must map to some connected component of $B$, we find the equivalence

$$X \simeq \sum_{b \in \pi_0 B} X_b/\text{Aut}(b) =: \int^{b \in B} X_b,$$

expressing $X$ as the homotopy sum of the fibres, or equivalently as a family of groupoids (indexed by $\pi_0(B)$ and with a group action in each). Given morphisms of groupoids $Y \xrightarrow{p} B \xrightarrow{f} A$, we have the following `Fubini formula':

$$\int^{b \in B} Y_b \simeq \int^{a \in A} \int^{b \in B_a} Y_b$$

which is actually the formula for the `dependent-sum' functor $\Sigma_f : \text{Grpd}_B \to \text{Grpd}_A$ given by postcomposition. In family notation the formula reads

$$\Sigma_f(Y_b \mid b \in B) = (\int^{b \in B_a} Y_b \mid a \in A),$$

just as Formula (6) in the **Set** case.

Pullback along $f : B \to A$, denoted $\Delta f$, is right adjoint to $\Sigma f$. This means of course homotopy adjoint, and amounts to a natural equivalence of mapping groupoids $\text{Grpd}_A(\Sigma f Y, X) \simeq \text{Grpd}_B(Y, \Delta f X)$. The proof relies on the universal property of the pullback. One may note the following formula for pullback, in family notation:

$$\Delta f(X_a \mid a \in A) = (X_f(b) \mid b \in B),$$

again completely analogous to the **Set** case (Formula (5)).
The 2-category of groupoids is locally cartesian closed. This means that the pullback functor in turn has a right adjoint \( \Pi_f : \text{Grpd}_{/B} \to \text{Grpd}_{/A} \). The general formula is an end formula; for \( Y \to B \), the fibre of \( \Pi_f Y \) over \( a \in A \) can be described explicitly as the mapping groupoid

\[
(\Pi_f Y)_a = \text{Grpd}_{/B}(B_a, Y).
\]

(A more explicit formula will be derived in the discrete case below.)

2.4 Polynomial functors. A polynomial is a diagram of groupoids

\[
I \xleftarrow{s} E \xrightarrow{p} B \xrightarrow{t} J.
\]

The associated polynomial functor (or the extension of the polynomial) is given as the composite

\[
\text{Grpd}_{/I} \xrightarrow{\Delta_s} \text{Grpd}_{/E} \xrightarrow{\Pi_p} \text{Grpd}_{/B} \xrightarrow{\Sigma_t} \text{Grpd}_{/J}.
\]

2.5 Beck–Chevalley, distributivity, and composition. Given a homotopy pullback square

\[
\begin{array}{ccc}
\alpha & \xrightarrow{\psi} & \beta \\
\downarrow & & \downarrow \\
\psi & \xrightarrow{\delta} & \delta
\end{array}
\]

there are natural equivalences of functors

\[
\Sigma_\alpha \circ \Delta_\psi \cong \Delta_\delta \circ \Sigma_\beta \quad \text{and} \quad \Delta_\beta \circ \Pi_\delta \cong \Pi_\psi \circ \Delta_\alpha.
\]

usually called the Beck–Chevalley conditions. A more subtle feature of the theory is distributivity, which in this setting is an equivalence saying how to distribute dependent products over dependent sums (and which can be interpreted as a type-theoretic form of the axiom of choice \([38]\)). We shall not need the details here. See \([18]\) for the classical case, and Weber \([45]\) for a deeper treatment. The Beck–Chevalley conditions and distributivity yield a formula for composing polynomial functors \([18]\).

2.6 Natural transformations. Just as in the classical case \([18]\), homotopy cartesian natural transformations \( P' \Rightarrow P \) of polynomial functors (in one variable) correspond precisely to homotopy cartesian diagrams

\[
\begin{array}{ccc}
E' & \xrightarrow{d} & B' \\
\downarrow & & \downarrow \\
E & \xrightarrow{d} & B.
\end{array}
\]
This is an easy consequence of Beck–Chevalley. Showing more generally that arbitrary natural transformations are given essentially uniquely by diagrams

\[
\begin{array}{ccc}
E' & \\ & \\ & \\
\downarrow & & \downarrow \\
B' & \\ & \\
\downarrow & & \downarrow \\
E & \rightarrow & B
\end{array}
\]

is a bit more involved and depends on a homotopy version of the Yoneda lemma. (At the time of this writing, this result is not as precise as in the 1-dimensional case of [18].)

2.7 Spans and stuff operators. Spans of groupoids are the special case of groupoid polynomials where the middle map is the identity (or an equivalence). These constitute a categorification of matrix algebra, and were called stuff operators by Baez and Dolan [9]; they have been used to give groupoid models for certain aspects of Hecke algebras and Hall algebras [10].

3 Exactness; combinatorial polynomial functors

The following results from [20] are actually proved in the setting of ∞-groupoids, but the proofs work also for 1-groupoids. We now suppress the clumsy ‘homotopy’ everywhere, although of course all limits and colimits mentioned refer to the homotopy notions.

**Theorem 3.1** (Gepner-Kock [20].) A functor \( \text{Grpd}_{/I} \rightarrow \text{Grpd}_{/J} \) is polynomial if and only if it is accessible and preserves conical limits.

By conical limit we mean limit over a diagram with a terminal vertex. Recall that a functor is accessible [37, Ch. 5] when it preserves \( \kappa \)-filtered colimits for some regular cardinal \( \kappa \). The regular cardinal here is explicitly characterised:

**Proposition 3.2** ([20]) A polynomial functor given by \( I \leftarrow E \xrightarrow{p} E \rightarrow J \) preserves \( \kappa \)-filtered colimits if and only if \( p \) has \( \kappa \)-compact fibres.

An important case is \( \kappa = \omega \). A groupoid is \( \omega \)-compact when it has finitely many components (i.e. \( \pi_0(X) \) is a finite set) and all vertex groups are finitely presented.

3.3 Discreteness. For many data types occurring in practice (including species and all the examples below), although they may have symmetries, the positions in each shape form a discrete groupoid, i.e. a groupoid equivalent to a set. In the polynomial formalism this amounts to the middle map \( p : E \rightarrow B \) having discrete fibres. In this case, the dependent product formula simplifies to

\[
(\Pi_p Y)_b = \prod_{e \in \pi_0(E_b)} Y_e,
\]

in analogy with (7), and hence all the formulae look exactly like the \( \text{Set} \) case.
The corresponding exactness condition is preservation of sifted colimits. A \( \kappa \)-
sifted colimit is a colimit over a diagram \( D \) whose diagonal \( D \to D^2 \) is cofinal for
every set \( S \) of cardinality \( < \kappa \) [37, Ch. 5].

**Proposition 3.4** ([20]) A polynomial functor given by \( I \leftarrow E \overset{p}{\to} E \to J \) preserves
\( \kappa \)-sifted colimits if and only if \( p \) has \( \kappa \)-compact discrete fibres.

### 3.5 Combinatorial polynomial functors.

We call a polynomial functor \( I \leftarrow E \overset{p}{\to} B \to J \) combinatorial if the fibres of \( p \) are equivalent to finite sets (i.e. are
\( \omega \)-compact discrete).

### 3.6 Species in groupoids (stuff types).

A Baez-Dolan stuff type [9] is a map of
groupoids \( F \to \mathbb{B}_\omega \). We prefer the name species in groupoids. (A classical species
is when the map has discrete fibres, or equivalently is faithful.) Its extension is the
left homotopy Kan extension of \( n \mapsto F_n \) along \( \mathbb{B}_\omega \subset \text{Grpd} \):

\[
X \mapsto \sum_{n \in \pi_0(\mathbb{B}_\omega)} \frac{F_n \times X^n}{\text{Aut}(n)}.
\]

(That’s a homotopy quotient of course.)

This functor is polynomial [20]: the representing groupoid map is the top row
in the pullback

\[
\begin{array}{ccc}
E & \rightarrow & F \\
\downarrow & & \downarrow \\
\mathbb{B}'_\omega & \rightarrow & \mathbb{B}_\omega.
\end{array}
\]

This map has finite discrete fibres since \( \mathbb{B}'_\omega \to \mathbb{B}_\omega \) has. (Here \( \mathbb{B}'_\omega \) is the groupoid
of finite pointed sets.) Conversely, given a groupoid polynomial \( E \to F \) with finite
discrete fibres, the ‘classifying map’ \( F \to \mathbb{B}_\omega \) (obtained since \( \mathbb{B}'_\omega \to \mathbb{B}_\omega \) classifies
finite discrete fibrations) yields a species in groupoids. One can check that the
extension of the polynomial agrees with the extension of the species. In conclusion:

**Proposition 3.7** ([20]) Combinatorial polynomial functors \( \text{Grpd} \to \text{Grpd} \) are
the same thing as analytic functors (in the sense of Baez-Dolan).

Combining these results we get a ‘Joyal theorem’:

**Corollary 3.8** ([20]) A functor \( \text{Grpd} \to \text{Grpd} \) is analytic (in the sense of Baez-
Dolan) if and only if it preserves \( \omega \)-sifted colimits and conical limits.

### 3.9 Generalised species.

The relationship between polynomial functors and the
generalised species of [15] has been sketched by Gambino and the author (unpublished). A generalised species depends on two categories \( I \) and \( J \), and has as extension a generalised analytic functor \( \text{PrSh}(I) \to \text{PrSh}(J) \); this generalises the
1985 notion but not the Baez-Dolan notion. If \( I \) and \( J \) are groupoids, these generalised analytic functors correspond to the ‘classical’ extension of combinatorial
polynomials over groupoids, i.e. involving \( \pi_0 \) on quotients.

### 3.10 Examples.

Groupoid polynomials encode data types in groupoids. For example, \( \mathbb{B}'_\omega \to \mathbb{B}_\omega \) encodes the multiset data type: the groupoid \( \mathbb{B}_\omega \) of finite sets and
bijections is the groupoid of shapes — the shape of a multiset is really the set
indexing its elements, not just its size. There are \( \mathbb{N} \)-many isoclasses; the isomor-
Kock

Isomorphisms should be interpreted as propositional equality. The fibre over \( S \in \mathbb{B}_\omega \) is the discrete groupoid of positions in \( S \), i.e. a uniform prescription of positions in multisets indexed by \( S \). Indeed, since \( \mathbb{B}_\omega' \to \mathbb{B}_\omega \) is a fibration, the fibre is canonically identified with the set \( S \) itself — note its natural \( \text{Aut}(S) \)-action. The discreteness of the fibre means that propositional equality between positions can be regarded as definitional equality. The extension of this quotient container is naturally an endofunctor \( \text{Grpd} \to \text{Grpd} \). But one obtains an endofunctor \( \text{Set} \to \text{Set} \) (in this case precisely (8)) by precomposing with the natural inclusion \( \text{Set} \to \text{Grpd} \) and postcomposing with \( \pi_0 : \text{Grpd} \to \text{Set} \). The first is harmless. The second corresponds to collapsing all isomorphisms to identity, i.e. interpreting propositional equality as definitional equality. If the argument is a set, the only collapse is the passage from homotopy quotient to naïve quotient (of actions on sets).

The data type of cyclic lists is groupoid polynomial, represented by \( \mathbb{C}_\omega' \to \mathbb{C}_\omega \), where \( \mathbb{C}_\omega \) is the groupoid of finite cyclically ordered sets, and \( \mathbb{C}_\omega' \) is the groupoid of pointed cyclically ordered finite sets. From 1.1, the list data type is represented by \( \mathbb{N}' \to \mathbb{N} \), interpreted as the groupoids of linearly ordered finite sets and pointed ditto. The diagram of groupoids now represents the cartesian natural transformations, or polymorphic functions, from lists to cyclic lists to multisets.

### 4 Trees

W-types in Martin-Löf type theory correspond to initial algebras of polynomial functors (cf. [17] and [8] for the extensional case, and [39] for the fully intensional case). The initial algebra for \( 1 + P \) can also be described as the set of operations for the free monad on \( P \), which in turn is the set of \( P \)-trees. \( P \)-trees (for \( P \) a polynomial functor over \( \text{Set} \) or any lccc) are always rigid, i.e. have no symmetries. Abstract trees, on the other hand, admit symmetries, so they are not \( P \)-trees for any \( \text{Set} \)-polynomial functor \( P \), and they are neither W-types nor containers in the classical sense. Instead, according to [31], abstract trees are themselves polynomial functors. It is convenient to take the following characterisation of trees as a definition:

4.1 Trees. ([31]) A (finite) tree is a diagram of finite sets

\[
\begin{array}{ccc}
A & \leftarrow & M \\
\downarrow & & \downarrow \\
N & \to & A
\end{array}
\]

satisfying the following three conditions:

(1) \( t \) is injective
(2) \( s \) is injective with singleton complement (called root and denoted 1).

With \( A = 1 + M \), define the walk-to-the-root function \( \sigma : A \to A \) by \( 1 \mapsto 1 \) and \( e \mapsto t(p(e)) \) for \( e \in M \).

(3) \( \forall x \in A : \exists k \in \mathbb{N} : \sigma^k(x) = 1 \).

The elements of \( A \) are called edges. The elements of \( N \) are called nodes. For \( b \in N \), the edge \( t(b) \) is called the output edge of the node. That \( t \) is injective is just to say that each edge is the output edge of at most one node. For \( b \in N \), the elements of the fibre \( M_b \) are called input edges of \( b \). Hence the whole set \( M = \sum_{b \in N} M_b \) can be thought of as the set of nodes-with-a-marked-input-edge, i.e. pairs \((b, e)\) where \( b \) is a node and \( e \) is an input edge of \( b \). The map \( s \) returns the marked edge. Condition (2) says that every edge is the input edge of a unique node, except the root edge. Condition (3) says that if you walk towards the root, in a finite number of steps you arrive there. The edges not in the image of \( t \) are called leaves.

4.2 Decorated trees: \( P \)-trees ([31]; see also [32,33,34]) An efficient way of encoding and manipulating decorations of trees is in terms of polynomial endofunctors.

Let \( P \) be a polynomial endofunctor given by \( I \xleftarrow{d} E \xrightarrow{p} B \xleftarrow{c} I \). A \( P \)-tree is a diagram

\[
\begin{array}{c}
A \\
\downarrow \downarrow
\end{array}
\xleftarrow{M} \xrightarrow{I} \begin{array}{c}
M \\
\downarrow \downarrow
\end{array} \xrightarrow{N} \begin{array}{c}
E \\
\downarrow \downarrow
\end{array} \xrightarrow{B} \begin{array}{c}
I \\
\downarrow \downarrow
\end{array}
\]

where the top row is a tree. The squares are commutative up to isomorphism, and it is important that the 2-cells be specified as part of the structure. Unfolding the definition, we see that a \( P \)-tree is a tree whose edges are decorated in \( I \), whose nodes are decorated in \( B \), and with the additional structure of an equivalence \( M_n \simeq E_b \) for each node \( n \in N \) with decoration \( b \in B \) (this is essentially just a bijection, since the fibres are discrete), an iso in \( I \) between the decoration of an edge \( m \in M_n \) and the corresponding \( d(e) \), and finally an iso in \( I \) between the decoration of the the output edge of \( n \) and \( c(b) \).

4.3 Examples of \( P \)-trees. Natural numbers are \( P \)-trees for the identity monad \( P(X) = X \), and are also the set of operations of the list monad. Planar finite trees are \( P \)-trees for \( P \) the list monad, and are also the set of operations of the free-non-symmetric-operad monad [36]. These two examples are the first entries of a canonical sequence of inductive data types underlying several approaches to higher category theory, the opetopes: opetopes in dimension \( n \) are \( P \)-trees for \( P \) a \textbf{Set}-polynomial functor whose operations are \((n-1)\)-opetopes [34]; hence opetopes are higher-dimensional trees.

Abstract finite trees are \( P \)-trees for the multiset functor \( 1 \xleftarrow{B'} \xrightarrow{B} 1 \), but cannot be realised as \( P \)-trees for any \textbf{Set}-polynomial \( P \).

4.4 Trees of Feynman graphs. In the so-called BPHZ renormalisation of perturbative quantum field theories, one is concerned with nestings of 1-particle irreducible (1PI) Feynman graphs, i.e. graphs [29] for which no single edge removal disconnects. Kreimer [35] discovered that the BPHZ procedure is encoded in a Hopf algebra of (non-planar) rooted trees, expressing the nesting of graphs.
In the picture the combinatorial tree in the middle expresses the nesting of 1PI subgraphs on the left; such trees are sufficient in Kreimer’s Hopf-algebra approach to BPHZ, but do not capture the symmetries of the graph. To this end, further decoration is needed in the tree, as partially indicated on the right. First of all, each node in the tree should be decorated by the 1PI graph it corresponds to in the nesting, and second, the tree should have leaves (input slots) corresponding to the vertices of the graph. The decorated tree should be regarded as a recipe for reconstructing the graph by inserting the decorating graphs into the vertices of the graphs of parent nodes. The numbers on the edges indicate the type constraint of each substitution: the outer interface of a graph must match the local interface of the vertex it is substituted into. But the type constraints on the tree decoration are not enough to reconstruct the graph, because for example the small graph decorating the left-hand node could be substituted into various different vertices of the graph.

The solution found in [33] is to consider $P$-trees, for $P$ the polynomial endo-functor given by $I \xleftarrow{s} E \xrightarrow{p} B \xrightarrow{t} I$, where $I$ is the groupoid of interaction labels for the theory (in this case the one-vertex graphs) and $B$ is the groupoid of connected 1PI graphs of the theory, and $E$ is the groupoid of such 1PI graphs with a marked vertex. The map $s$ returns the one-vertex subgraph at the mark, $p$ forgets the mark, and $t$ returns the outer interface of the graph, i.e. the graph obtained by contracting everything to a point, but keeping the external lines. A $P$-tree is hence a diagram like (10) with specified 2-cells. These 2-cells carry much of the structure: for example the 2-cell on the right says that the 1PI graph decorating a given node must have the same outline as the decoration of the outgoing edge of the node — or more precisely, and more realistically: an isomorphism is specified (it’s a bijection between external lines of one-vertex graphs). Similarly, the left-hand 2-cell specifies for each node-with-a-marked-incoming-edge $x' \in M$, an isomorphism between the one-vertex graph decorating that edge and the marked vertex of the graph decorating the marked node $x'$. Hence the structure of a $P$-tree is a complete recipe not only for which graphs should be substituted into which vertices, but also how specific bijections prescribe which external lines should be identified with which lines in the receiving graph. In fact, there is an equivalence of groupoids between nested graphs and $P$-trees [33]. This is exploited in [16] to establish algebraic identities concerning graphs by interpreting them as homotopy cardinalities of equivalences of groupoids of decorated trees.

Notice that the polynomial functor $P$ is combinatorial, since each graph has a discrete finite set of vertices. It is not a species in the classical sense though: the classifying map $B \rightarrow B_\omega$ sends a graph to its set of vertices, and since a graph
may have nontrivial automorphisms that fix every vertex, this map does not have discrete fibres.

5 Outlook

A 2-category is called **locally cartesian closed** when for every arrow \( f : B \to A \), we have the string of adjoint functors \( \Sigma f \dashv \Delta f \dashv \Pi f \). This structure formally implies the Beck-Chevalley equivalences and distributivity, which are the minimal requirements for a reasonable theory of polynomial functors. The theory of strength can be copied almost verbatim from [18], and it seems that the representation theorem of [18] also carries over.

While locally cartesian closed categories provides semantics for an extensional version of Martin-Löf type theory [43], [13], and locally cartesian closed 2-categories capture some 2-truncated version ([24], [19]), recent insight of Homotopy Type Theory strongly suggests that in the long run, the case of \( \infty \)-groupoids and other locally cartesian closed \( \infty \)-categories will be the real meat for type theory. Large parts of the \( \infty \)-theory of polynomial functors has already been worked out in joint work with David Gepner, and will appear elsewhere [20], and the general theory of locally cartesian closed \( \infty \)-categories is also under development. Nevertheless the groupoid case is interesting in its own right, since it already covers important applications: in particular for many purposes of combinatorial nature, 1-groupoids are all that is needed in order to handle symmetry issues. Time will tell whether for purposes of program semantics the groupoid level is enough too — otherwise it is a good stepping stone into the \( \infty \)-world.

References


Extracting a DPLL Algorithm

Andrew Lawrence 1,2 Ulrich Berger 3 Monika Seisenberger 4

Department of Computer Science
Swansea University
Swansea, UK

Abstract

We formalize a completeness proof for the DPLL proof system and extract a DPLL solver from it. When applied to a propositional formula in conjunctive normal form the program produces either a satisfying assignment or a derivation which shows that it is unsatisfiable. We use non-computational quantifiers to remove redundant computational content from the program and improve its performance. The formalization is carried out in the Minlog system.

Keywords: DPLL, Program Extraction, Interactive Theorem Proving, SAT.

1 Introduction

In order for verification tools to be used in an industrial context they have to be trusted to a high degree and in many cases are required to be certified. We present a new application of program extraction to develop correct certifiable decision procedures. SAT-solvers are one such decision procedure which are common in verification tools and attempt to solve the Boolean satisfiability problem. The majority of SAT-solvers used in an industrial context are based on the DPLL proof system. We have formalized a correctness and completeness proof of the DPLL proof system in the interactive theorem prover Minlog [2,4]. Using the program extraction facilities of Minlog we have been able to obtain a formally verified SAT-solving algorithm. When run on a CNF formula this algorithm produces a model satisfying the formula or a DPLL derivation showing its unsatisfiability. Computational redundancy was then removed from the algorithm by labelling certain universal quantifiers in the proof as non-computational, an optimisation available in the Minlog system.

1 I would like to thank Invensys Rail UK for their support.
2 Email: csal@swansea.ac.uk
3 Email: u.berger@swansea.ac.uk
4 Email: m.seisenberger@swansea.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
The performance of the resulting program was tested with a number of pigeon hole formulæ.

Program extraction aims at producing formally verified programs from a constructive proof. An example of early work with program extraction is that done in the Nuprl system [10]. Examples of program extraction in Minlog are [5,3]. Other mature interactive theorem provers that support program extraction are Coq [6], which is based on the calculus of inductive constructions, and Isabelle [22,21], a generic theorem prover with extensions for many logics. More recently, other interactive theorem provers based on dependent types [PM80], such as Agda [9] and Epigram [19], have emerged which realise the Curry-Howard correspondence [13,12,14] and therefore can also be viewed as supporting program extraction.

Several attempts have been made to integrate an automatic theorem prover into Coq. Most of this work has made use of Coq’s program extraction facilities to extract programs from proofs of decision procedures. A SAT-solver based on the DPLL algorithm has been formalized and its soundness and completeness are verified in Coq [16] and code has been extracted from the proof. Finally, the extracted system has instantiated on the propositional fragment of Coq’s logic creating a user friendly proof tactic. Binary decision diagrams have been formalized in Coq [26]. Then their correctness has been proven and certified BDD algorithms have been extracted in Caml. The main reason for their formalization was to integrate symbolic model checking in Coq.

Significant work has also been performed in Isabelle with several decision procedures having been verified and integrated into the system. The DPLL algorithm has been formalized in [17]. The automatic theorem prover Metis [23] was formally verified inside Isabelle and is now used to reconstruct proofs from faster external procedures such as the ones used in Sledgehammer [8].

The approaches to formalize a DPLL SAT-solver in both Coq and Isabelle involve explicitly stating the algorithm to be verified. In contrast, we prove a theorem that just states that each formula in CNF is either unsatisfiable or has a model, and synthesise the program from the proof. In the long run we would like to integrate automatic verification techniques into Minlog. Extracting a SAT-solver in Minlog is one step towards our end goal.

2 Preliminaries

We begin with some basic definitions following closely [16,17].

Definition 2.1

(i) A literal $l$ is either a positive variable $+v$ or a negative variable $-v$, i.e. a variable $v$ with a label $+$ or $-$ attached.

(ii) We define a bar operation which computes the opposite value of a literal as follows: $\bar{+}v = -v$, $\bar{-}v = +v$.

(iii) A clause $C$ is a finite set of literals $\{l_1, \ldots, l_k\}$.  

245
(iv) A formula is in conjunctive normal form (CNF) if it is a finite conjunction of clauses. By a formula $\Delta$ we will always mean a formula in CNF, and we will identify it with a finite set of clauses $\{C_1, \ldots, C_k\}$ in the obvious way.

(v) A valuation $\Gamma$ is a set of literals $\{l_1, \ldots, l_k\}$.

(vi) A valuation $\Gamma$ is consistent (consistent $\Gamma$) if $\forall l \in \Gamma \rightarrow l \notin \Gamma$).

(vii) A model is a total function $\text{Mod}$ which maps literals to booleans and satisfies the property $\forall (\text{Mod} l \leftrightarrow \neg \text{Mod} \bar{l})$.

We shall use the following two abbreviations:

- For a given valuation $\Gamma$, $\forall l \in \Gamma \text{Mod} l$ is abbreviated as $\text{Mod} |\Gamma| = \Gamma$.
- For a given formula $\Delta$, $\forall C \in \Delta \exists l \in C \text{Mod} l$ is abbreviated as $\text{Mod} |\Delta| = \Delta$.

We call a valuation $\Gamma$ and a formula $\Delta$ compatible (compatible$(\Gamma, \Delta)$) if the exists a model satisfying both, i.e. $\exists \text{Mod} (\text{Mod} \models \Gamma \land \text{Mod} \models \Delta)$; otherwise $\Gamma$ and $\Delta$ are called incompatible (incompatible$(\Gamma, \Delta)$).

**Definition 2.2** A sequent $\Gamma \vdash \Delta$ is a pair consisting of a valuation and a formula.

The intended meaning of a sequent $\Gamma \vdash \Delta$ is that $\Gamma$ and $\Delta$ are incompatible. As a special case, when $\Gamma$ is empty, $\vdash \Delta$ means that $\Delta$ is unsatisfiable.

**Definition 2.3** [DPLL Proof System] The DPLL proof system consists of 5 rules:

\[
\begin{align*}
\frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, l} & \quad \text{(Unit)} \quad \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, l \lor C} & \quad \text{(Red)} \quad \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \lor C} & \quad \text{(Elim)} \\
\frac{\Gamma \vdash \Delta, \emptyset}{\Gamma \vdash \Delta} & \quad \text{(Conflict)} \quad \frac{\Gamma, l \vdash \Delta}{\Gamma, \bar{l} \vdash \Delta} & \quad \text{(Split)}
\end{align*}
\]

Notation: We use a comma to denote adding an element to a set as seen in the above proof system with $\Gamma, l := \Gamma \cup \{l\}$. In the case of $\Gamma, l$ it is understood that $l \notin \Gamma$.

3 Soundness and Completeness

In this section we sketch the formal proof of soundness and completeness of the DPLL proof system. We will be very brief with the Soundness Theorem since its proof doesn’t carry computational content and a similar proof is carried out in [16,17]. On the other hand, we will describe the proof of the Completeness Theorem in some detail since we extract our SAT solver from it.

We first reformulate the DPLL proof system as an inductive definition that can be immediate formalized in the Minlog system. The definition has a clause for each rule. We notationally identify a sequent $\Gamma \vdash \Delta$ with the statement "$\Gamma \vdash \Delta$ is derivable".

**Definition 3.1** The set of derivable sequents $\Gamma \vdash \Delta$ is defined inductively by the following clauses:
Conflict
\[ \forall \Gamma, \Delta (\emptyset \in \Delta \rightarrow \Gamma \vdash \Delta) \]

Unit
\[ \forall \Gamma, \Delta, l ([l] \in \Delta \rightarrow \Gamma, l \vdash \Delta \setminus \{l\} \rightarrow \Gamma \vdash \Delta) \]

Elim
\[ \forall \Gamma, \Delta, l, C (l \in \Gamma \rightarrow l \in C \rightarrow C \in \Delta \rightarrow \Gamma \vdash \Delta \setminus \{C\} \rightarrow \Gamma \vdash \Delta) \]

Red
\[ \forall \Gamma, \Delta, l, C (C \in \Gamma \rightarrow \bar{l} \in C \rightarrow C \in \Delta \rightarrow \Gamma \vdash (\Delta \setminus C), (C \setminus \{\bar{l}\}) \rightarrow \Gamma \vdash \Delta) \]

Split
\[ \forall \Gamma, \Delta, l ((\Gamma, l \vdash \Delta) \rightarrow (\Gamma, \bar{l} \vdash \Delta) \rightarrow \Gamma \vdash \Delta) \]

Theorem 3.2 (Soundness) \textit{If} \[ \Delta \vdash \Gamma \text{ is derivable, then} \quad \Delta \text{ and } \Gamma \text{ are incompatible.} \]

The proof proceeds by structural induction on the given derivation of the sequent \[ \Gamma \vdash \Delta \]. Since the proof is similar to the one given in [16] and doesn’t contain interesting computational content, we omit further details.

We now turn our attention to the Completeness Theorem for the DPLL proof system. The expected statement of completeness is:
\[ \forall \Gamma, \Delta (\text{incompatible}(\Gamma, \Delta) \rightarrow \Gamma \vdash \Delta) \]

A constructive proof of this statement would yield a program that computes a DPLL proof for incompatible \( \Gamma, \Delta \). We reformulate the statement by replacing the implication ‘\( \text{incompatible}(\Gamma, \Delta) \rightarrow \Gamma \vdash \Delta \)’ with the classically equivalent but constructively stronger disjunction ‘\( \text{compatible}(\Gamma, \Delta) \lor \Gamma \vdash \Delta \)’. In this way, we obtain an enhanced program that still computes a DPLL proof for incompatible \( \Gamma, \Delta \), but in addition produces a model if \( \Gamma \) and \( \Delta \) are compatible.

Theorem 3.3 (Completeness of DPLL)
\[ \forall \Gamma, \Delta (\text{compatible}(\Gamma, \Delta) \lor \Gamma \vdash \Delta) \]

Proof. We aim to perform the proof in such a way that an efficient program is extracted. Therefore, we adopt the following strategy:

(i) Since performing a split is the only computational expensive operation – it is the only rule forcing the proof search to branch – we only apply it when it is absolutely necessary.

(ii) We perform an optimisation on the proof level by partitioning the clauses into ‘clean’ and ‘unclean’ clauses, where a clause is called clean if we cannot apply \textit{Elim}, \textit{Reduce} or \textit{Unit} to that clause. This increases the efficiency of the algorithm by reducing the number of comparisons needed.
To this end we prove the statement

$$\forall \Gamma, \Delta, \Theta (\emptyset \notin \Theta \land \text{consistent } \Gamma \land \text{Var}(\Gamma) \cap \text{Var}(\Theta) = \emptyset \rightarrow \text{success}(\Gamma, \Delta, \Theta))$$

where

$$\text{Var}(\Gamma) := \{v \mid +v \in \Gamma \lor -v \in \Gamma\}$$

$$\text{success}(\Gamma, \Delta, \Theta) := \text{compatible}(\Gamma, \Delta \cup \Theta) \lor \Gamma \vdash \Delta \cup \Theta$$

The proof is by induction on the following measure:

$$\mu(\Delta; \Theta; \Gamma) := |(\Delta \cup \Theta) \setminus \text{Var}(\Gamma)| + |\cup \Delta| + |\cup \Theta|$$

where

$$|X| := \text{the cardinality of a set } X$$

$$\cup X := \{l \mid \exists C \in \Delta \land l \in C\}$$

$$\Delta \setminus \setminus V := \{l \mid \exists l \in \cup \Delta \land \text{Var}(l) \notin V\}$$

There will also be a side induction on $$|\Delta|$$ (the number of clauses in $$\Delta$$).

**Case 1** $$\Delta = \emptyset$$

**Case 1.1** $$\Theta = \emptyset$$

In this case one has to show $$\forall \Gamma (\text{consistent } \Gamma \land \text{Var}(\Gamma) \cap \text{Var}(\emptyset) = \emptyset \rightarrow (\Gamma \vdash \emptyset) \lor \exists \text{Mod}(\text{Mod} \models \Gamma \land \text{Mod} \models \emptyset))$$. We prove $$\exists \text{Mod}(\text{Mod} \models \Gamma \land \text{Mod} \models \emptyset)$$ by introducing a model $$M(l) = True \leftrightarrow l \in \Gamma$$.

**Case 1.2** $$\Theta \neq \emptyset$$

Using the assumptions we show $$\Gamma \vdash \Theta \lor \exists \text{Mod}(\text{Mod} \models \Gamma \land \text{Mod} \models \emptyset)$$. To this end, we select a literal with which to apply our *Split* rule. There are many possible ways to do this, for example one could pick the most commonly occurring literal. We will simply pick the first literal to occur in $$\Theta$$.

Let $$l$$ be the first literal to occur in $$\Theta$$, then the following holds $$\mu(\Theta; \emptyset; (l, \Gamma)) < \mu(\emptyset; \Theta; \Gamma)$$ and $$\mu(\Theta; \emptyset; (\bar{l}, \Gamma)) < \mu(\emptyset; \Theta; \Gamma)$$. We instantiate the induction hypothesis twice, firstly with $$(l, \Gamma)$$, $$\Theta$$ and $$\emptyset$$ and secondly with $$(\bar{l}, \Gamma)$$, $$\Theta$$ and $$\emptyset$$.

We perform a case distinction on $$(\Gamma, l \vdash \Theta) \lor \exists \text{Mod}(\text{Mod} \models \Gamma \land \text{Mod} \models \emptyset)$$ and $$(\Gamma, \bar{l} \vdash \Theta) \lor \exists \text{Mod}(\text{Mod} \models \Gamma \land \text{Mod} \models \emptyset)$$ resulting in 4 cases. In the case that $$\Gamma, l \vdash \Theta$$ and $$\Gamma, \bar{l} \vdash \Theta$$ hold the *Split* rule is applied and we obtain $$\Gamma \vdash \Theta$$. In all of the other cases we use the model which we obtain from the induction hypothesis. In the case where there exist two models we use the model which contains the non barred literal.

It still remains to prove consistent $$\Gamma, l$$ and consistent $$\Gamma, \bar{l}$$ which follows from consistent $$\Gamma$$ and $$\text{Var}(\Gamma) \cap \text{Var}(\emptyset) = \emptyset$$.

**Case 2** $$\Delta = \Delta', C$$

This is the case in which $$\Delta$$ is non empty and contains at least one clause namely $$C$$. We will now perform a case distinction on whether the valuation $$\Gamma$$ has any literals
in common in with $C$.

**Case 2.1** $\Gamma \cap C = \emptyset$

In the case we perform a further case distinction on the size of the clause $C$.

**Case 2.1.1** $C = \emptyset$

The proof of this case is fairly trivial. It suffices to show $\Gamma \vdash (\Delta', \emptyset) \cup \Theta$ which follows from the Conflict rule.

**Case 2.1.2** $C = \{l\}$

We will now show $\Gamma \vdash (\Delta', \{l\}) \cup \Theta$ or $\exists \text{Mod} (\text{Mod} | = \Gamma \land \text{Mod} | = (\Delta', \{l\}) \cup \Theta)$ by performing a case distinction on $\bar{l} \in \Gamma$.

In the case in which $\bar{l} \in \Gamma$ we apply the Reduce rule producing an empty clause, followed by the Conflict rule.

In the case that $\bar{l} \notin \Gamma$ holds we instantiate the induction hypothesis with $(\Gamma, l)$, $\Delta' \cup \Theta$ and $\emptyset$ and perform a case distinction on $(\Gamma, l \vdash \Delta' \cup \Theta) \lor \exists \text{Mod} (\text{Mod} | = \Gamma \land \text{Mod} | = \Delta' \cup \Theta)$.

It remains to show consistent $(\Gamma, l)$. This is done using consistent $(\Gamma)$, $\Gamma \cap \{l\} = \emptyset$ and $\bar{l} \notin \Gamma$.

**Case 2.1.3** $|C| \geq 2$

Using Lemma 3.4 now perform a case distinction on $\exists l (l \in C \land \bar{l} \in \Gamma) \lor \neg \exists l (l \in C \land \bar{l} \in \Gamma)$.

In the case that $\exists l (l \in C \land \bar{l} \in \Gamma)$ we obtain a witness $l'$ s.t. $l' \in C \land \bar{l} \in \Gamma$. We then instantiate the induction hypothesis and perform a case distinction on $(\Gamma \vdash \Delta', (C \setminus l') \cup \Theta) \lor \exists \text{Mod} (\text{Mod} | = \Gamma \land \text{Mod} | = \Delta' \cup (C \setminus l') \cup \Theta)$. In the case that $\Gamma \vdash \Delta', (C \setminus l') \cup \Theta$ holds, we apply the Reduce rule. In the other case we use the model from the induction hypothesis and show $\text{Mod} | = \Gamma \land \text{Mod} | = \Delta \cup \Theta$.

In the case that $\neg \exists l (l \in C \land \bar{l} \in \Gamma)$ we may move $C$ from $\Delta$ to $\Theta$: The side induction hypothesis is applied with $\Gamma$, $\Delta'$ and $(\Theta, C)$ resulting in two cases $\Gamma \vdash \Delta' \cup (\Theta, C) \lor \exists \text{Mod} (\text{Mod} | = \Gamma \land \text{Mod} | = \Delta' \cup (\Theta, C))$. Both cases are solved in a similar fashion. We can rewrite $(\Delta', C) \cup \Theta$ as $\Delta' \cup (\Theta, C)$. To complete the proof we must show $\text{Var}(\Gamma) \cap \text{Var}(\Theta, C) = \emptyset$ by $\text{Var}(\Gamma) \cap \text{Var}(\Theta) = \emptyset$, $\Gamma \cap C = \emptyset$ and $\neg \exists l (l \in C \land \bar{l} \in \Gamma)$.

**Case 2.2** $\Gamma \cap C \neq \emptyset$

The valuation $\Gamma$ and the clause $C$ have literals in common which we shall remove using the Elim rule: We apply the induction hypothesis instantiated with $\Gamma$, $(\Delta', (C \setminus l))$ and $\Theta$ resulting in two cases. In the first case, in which $\Gamma \vdash (\Delta', (C \setminus l)) \cup \Theta$ holds, we apply Lemma 3.5.

In the second case, in which $\exists \text{Mod} (\text{Mod} | = \Gamma \land \text{Mod} | = \Delta', C \setminus l \cup \Theta)$ holds, we
use this model and Lemma 3.5.

The following lemmata were used in the proof. Classically, these lemmas are trivial, but not so constructively, since their proofs contain a decision respective search procedure.

**Lemma 3.4** For all valuations $\Gamma$ and clauses $C$ the following holds:

$$\exists l (l \in C \land \bar{l} \in \Gamma) \lor \neg \exists l (l \in C \land \bar{l} \in \Gamma)$$

*Proof by induction on $C$ and on $\Gamma$.*

**Lemma 3.5** Given a clause $C$ and a valuation $\Gamma$

$$\Gamma \cap C \neq \emptyset \rightarrow \exists l (l \in C \land l \in \Gamma)$$

*Proof by induction on $C$ and $\Gamma$.*

### 4 Program Extraction

Program extraction in Minlog is based on modified realizability [15]. We highlight a few aspects that are important to understand the optimizations we achieved. For a complete and precise description of program extraction we refer to [25].

A formula is said to have *computational content* if it has at least one occurrence of $\exists$ or an $\lor$ at a strictly positive position. To every such formula $A$ one assigns a type $\tau(A)$ of 'potential realizers'. If the formula has no computational content, one sets $\tau(A) = \epsilon$. Then, from a proof of a formula $A$ with computational content one can extract a program $M$ of type $\tau(A)$ that realizes $A$ (written $M \ r A$), that is, $M$ solves the computational problem expressed by $A$.

In order to fine-tune the computational content, in particular to remove redundant content, Minlog offers, besides the usual quantifiers $\forall$ and $\exists$, the *non-computational (nc)* quantifiers $\forall_{nc}$ and $\exists_{nc}$. The non-computational quantifiers have the same logical meaning as the usual quantifiers, but indicate that the extracted program does not operate on the quantified variable, only on its realizer. The definitions of the type and the realizability relations for the ordinary quantifiers are:

$$\tau(\forall x^\rho A) = \rho \rightarrow \tau(A)$$

$$\tau(\exists x^\rho A) = \rho \times \tau(A)$$

$$f \ r \forall x^\rho A = \forall x^\rho (f(x) \ r A)$$

$$(a, y) \ r \exists x^\rho A = a \ r A[y/x]$$

Here, the notation $x^\rho$ means $x$ has type $\rho$.

For the nc-quantifiers the realizers do not depend on the quantified variables:
The program extraction procedure respects the different kind of quantifiers by omitting in the nc case any information corresponding to the quantified variable. The proof rules for the nc-quantifiers are subject to stricter variable conditions ensuring that the omitted information is indeed not needed in the extracted program. Minlog is able to automatically detect the maximal set of occurrences of quantifiers in a proof that can be made non-computational without compromising the correctness of the proof or changing the proven formula [24].

5 The Extracted Program

The extracted program has the same structure as the proof. It takes a formula \( \Delta \) in CNF as input and produces either a model of \( \Delta \) or a derivation of unsatisfiability. The Minlog system automatically generates algebraic data-types for realizers of inductively defined predicates. For instance, the inductive definition of derivable is extracted into a data structure representing derivations in the DPLL proof system and induction proofs are translated into structurally recursive procedures. The control structure of the program closely follows the inductions and case distinctions we performed in the proof. The measure induction at the start of the proof turns into guarded recursion using the same measure in the program. The separate lemmas which we have used in the proof are called like procedures. The main benefit of the extraction process is that everything is generated automatically and hence no errors occur during coding. Furthermore, a formal correctness proof for the extracted program is generated as well.

The main body of the program as extracted by Minlog is as follows:

\[
\begin{align*}
\text{cgRec} & \quad ([\text{cs3}] \text{(Rec list cla=>list cla=>valu=>}) \\
& \quad \quad \quad \quad \text{(list cla=>list cla=>valu=>algsuccess)=>algsuccess}) \\
& \quad \quad \quad \quad \text{cs3 cbase cstep})
\end{align*}
\]

The first thing we see in the extracted program is a constant \texttt{cgRec} representing the guarded recursion. The latter corresponds to the following definition:

\[
(G\text{RecGuard} \rho_1\rho_2\rho_3\tau)\mu z_1 z_2 z_3 G t = \\
f z_1 z_2 z_3 t \text{ where} \\
f : \rho_1 => \rho_2 => \rho_3 => \text{Bool} => \tau \\
f x_1 x_2 x_3 \text{True} = \\
G x_1 x_2 x_3 ([y_1, y_2, y_3], \\
f y_1 y_2 y_3 (\mu y_1 y_2 y_3 < \mu x_1 x_2 x_3))
\]
Similarly, it is possible to rewrite the operation \(\text{Rec list} \rho \Rightarrow \tau\) using a function \(f : \text{list} \rho \Rightarrow \tau\) which gives a more intuitive view of its behaviour.

\[
(\text{Rec list} \rho \Rightarrow \tau) \ ys \ z \ G = f \ ys \quad \text{where}
\]
\[
f : \text{list} \rho \Rightarrow \tau
\]
\[
f \ Nil = z
\]
\[
f (x :: xs) = G x \ xs (f \ xs)
\]

6 Execution of the Extracted Program

We have extracted two programs, one from the proof above (\(\forall\) solver) and one from the proof involving nc-quantifiers (\(\forall_{nc}\) solver). The nc-quantifiers were inserted at strategic places in the proof above such as in inductive definitions and in the lemmas. (We did this by hand as also some definitions were involved.) In the following we will see how both \(\forall\) and \(\forall_{nc}\) solvers behave when they are applied to a number SAT problems. The extracted decision procedure was run on several instances of the pigeon hole principle [11]. The pigeon hole principle states that there is no injective function that maps \(\{1, 2 \ldots, n\}\) to \(\{1, 2 \ldots, n - 1\}\). Intuitively one can think of this as stating “it is not possible to put \(n\) pigeons into \(n - 1\) holes and only have one pigeon in each hole”.

Definition 6.1 [Pigeon Hole Formula]

\[\text{PHP}(n, m) := \{l_{i,1} \lor \ldots \lor l_{i,m} | 1 \leq i \leq n\} \cup \{-l_{i,k} \lor -l_{j,k} | 1 \leq i < j \leq n, 1 \leq k \leq m\}\]

Here \(l_{i,k}\) represents the statement “pigeon \(i\) sits in hole \(k\)”. The whole formula \(\text{PHP}(n, m)\) states that \(n\) pigeons sit in \(m\) holes such that no two pigeons sit in the same hole. Hence, \(\text{PHP}(n, m)\) is satisfiable iff \(n \leq m\). For example, if we run our DPLL solver with the formula \(\text{PHP}(2, 1) = l_{11} \land l_{21} \land (\neg l_{11} \lor \neg l_{21})\), the following derivation is produced:

\[
\frac{l_{11}, l_{21} \vdash \emptyset}{\text{Conflict}}
\]
\[
\frac{l_{11}, l_{21} \vdash \{-l_{21}\}}{\text{Reduce}, l_{21}, \{-l_{21}\}}
\]
\[
\frac{l_{11}, l_{21} \vdash \{-l_{11}, \neg l_{21}\}}{\text{Reduce}, l_{11}, \{-l_{11}, \neg l_{21}\}}
\]
\[
\frac{l_{11} \vdash \{l_{21}\}, \{-l_{11}, l_{21}\}}{\text{Unit}, l_{21}}
\]
\[
\frac{l_{11} \vdash \{l_{21}\}, \{-l_{11}, \neg l_{21}\}}{\text{Unit}, l_{11}}
\]

Running the DPLL solver on a satisfiable formula results in a function which maps literals to booleans. For example running the solver with \(\text{PHP}(2, 2)\) results in the function \(\text{mod} : \text{literals} \rightarrow \mathbb{B}\) where \(\text{mod}(x) = T\) iff \(x \in \{l_{12}, \neg l_{11}, l_{21}, \neg l_{22}\}\).

\(\text{Inhab}\) is a dummy variable indicating a use of the axiom scheme efq: \(\bot \rightarrow A\).
6.1 Comparison of Program Performance with and without nc-Quantifiers

We have performed two comparisons between our extracted solvers with and without the $\forall_{nc}$ quantifiers. The first comparison is on unsatisfiable pigeon hole formulae $PHP(n+1,n)$ which will demonstrate the relative efficiency of the solvers in constructing a derivation. The second comparison is on satisfiable formulae $PHP(n,n)$. The programs have been run on the term level in the Minlog system using the built in term rewriting system. Therefore, the running times can not be expected to be competitive, but it is interesting to observe the relative difference between the two solvers.

<table>
<thead>
<tr>
<th>Solver</th>
<th>$PHP(2,1)$</th>
<th>$PHP(3,2)$</th>
<th>$PHP(4,3)$</th>
<th>$PHP(5,4)$</th>
<th>$PHP(6,5)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall$</td>
<td>$&lt; 1$ Sec</td>
<td>1.17</td>
<td>33.62</td>
<td>13:54</td>
<td>5:35:41</td>
</tr>
<tr>
<td>$\forall_{nc}$</td>
<td>$&lt; 1$ Sec</td>
<td>$&lt; 1$ Sec</td>
<td>11.61</td>
<td>2:41</td>
<td>37:25.27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solver</th>
<th>$PHP(2,2)$</th>
<th>$PHP(3,3)$</th>
<th>$PHP(4,4)$</th>
<th>$PHP(5,5)$</th>
<th>$PHP(6,6)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall$</td>
<td>$&lt; 1$ Sec</td>
<td>$&lt; 1$ Sec</td>
<td>5.45</td>
<td>26.09</td>
<td>1:34.11</td>
</tr>
<tr>
<td>$\forall_{nc}$</td>
<td>$&lt; 1$ Sec</td>
<td>$&lt; 1$ Sec</td>
<td>5.25</td>
<td>25.03</td>
<td>1:24.88</td>
</tr>
</tbody>
</table>

These results demonstrate that the solver extracted from the proof containing the $\forall_{nc}$ quantifiers is significantly faster on unsatisfiable formulae than the solver extracted from the original proof. This is due to the number of non-computational quantifiers added to the definition of derivable. When applied to the pigeon hole formula $PHP(6,5)$ the $\forall$ solver takes 5 and a half hours where as the $\forall_{nc}$ solvers takes only 37 minutes.

7 Conclusion

In this paper we presented a new application of program extraction to decision procedures. The DPLL proof system was formalized and then a constructive proof of completeness was performed from which we extracted a program. The extracted program attempts to show the (un)satisfiability of a propositional formula in conjunctive normal form (CNF). If the CNF formula is satisfiable it produces a model of the formula; otherwise it produces a derivation showing the unsatisfiability of the formula. We strategically inserted $\forall_{nc}$ quantifiers into the proof to reduce the complexity of the extracted program and increase its performance. The performance of the original solver was then compared with this improved solver using pigeon hole formulae.

Overall, the case study shows that the approach of developing verified programs via extraction from proofs is scalable to non-trival applications. Furthermore, it demonstrates how to include efficiency considerations into this approach (for instance, we have avoided repeated unnecessary look-ups of clauses by the split of
clause sets in two sets $\Delta$ and $\Theta$). This counters the often heard argument that with program extraction one 'looses the grip' on the program and its efficiency. The big challenge, however, will be to extract from proofs also qualitative information about quantitative aspects of the extracted programs (e.g. computational complexity), for example, by combining approaches to implicit complexity with program extraction (see e.g. [1]).

**Future Work**

In order to apply the solver practically we would have to translate the Minlog term into a functional programming language such as Scheme or Haskell. Currently a partial translation mechanism from Minlog into Scheme is available which we would like to extend to a full translation mechanism. Having our DPLL solver as a Haskell program would allow us to observe how lazy evaluation affects performance.

We further want to prove the equivalence of the DPLL proof system and the tree resolution proof system. This would then allow us to extract a resolution solver based on the DPLL algorithm. Showing the equivalence between these two systems has not been formalized in an interactive theorem prover.

Extracting efficient data structures for our DPLL solver would greatly increase the efficiency of the solver and would provide another interesting example of program extraction. Since Haskell is based on lazy data structures such as tries, we would like to know whether a solver in Haskell would make use of these structures and gain something in efficiency. The solver would also benefit from a heuristic to select a splitting literal, currently it just selects the first literal in the formula $\Theta$.

Our current solver could be further improved by adding optimisation techniques such as clause learning and conflict analysis [7,20,18]. This would require a modification of the current completeness theorem so that the derivation is performed with respect to an added implication graph $G$ i.e. $\Gamma \vdash_G \Delta$.

**References**


**Appendix: The Extracted Program**

The appendix lists the full extracted program as it was produced by Minlog.

**Main Program**

```plaintext
Main Program =
cgRec (cs3)(list cla=>list cla=>valu=> (list cla=>list cla=>valu=>algsuccess=>algsuccess) cs3 cbase cstep)

bgRec =
[(list cla=>valu=>(list cla=>list cla=>valu=>algsuccess=>algsuccess)_0,cs1,cs2,val3)
 (list cla=>valu=>(list cla=>list cla=>valu=>algsuccess=>algsuccess)_0 cs1)
```

255
Lawrence, Berger and Seisenberger

```
cs2 val3
( [cs4,cs5,val6] (GRecGuard list cla list cla valu algsuccess) 
  [cs7,cs8,val9]
  Lh(toLit cs7)+Lh(toLit cs8)+Lh(setminus(varSetClaList(cs7++cs8))(var val9))
) cs4 cs5 val6

(list cla=>>list cla=>>valu=>>algsuccess) _0
(Lh(toLit cs4)+Lh(toLit cs5)+Lh(setminus(varSetClaList(cs4++cs5))(var val6))<Lh(toLit cs1)+Lh(toLit cs2)+Lh(setminus(varSetClaList(cs1++cs2))(var val3))
)

cbase = [cs0, val1, (list cla=>>list cla=>>valu=>>algsuccess) _2]
  [if cs0
    (cModelCase val1)
    
    [cs3, cs4] cSplitCase cs0 val1 c3 cs4 (list cla=>>list cla=>>valu=>>algsuccess) _2
  ]

cstep = [c0, cs1, (list cla=>>valu=>>list cla=>>valu=>>algsuccess) _2, cs3, val4, (list cla=>>list cla=>>valu=>>algsuccess) _5]
  [if (vcIntersection val4 c0=(Nil lit))
    [if c0
      [ls6]
      [if ls6
        (cConflictCase cs1 cs3 val4)
        
        [17, ls8]
        [if ls8
          [if (memlv(opposite l7) val4)
            (cElimConflictCase cs1 cs3 val4)
            (cUnitCase l7 c0 ls6 ls8 cs1 cs3 val4 (list cla=>>list cla=>>valu=>>algsuccess) _5)
            
            [19, ls10]
            [if (cindmemlem(17::19::ls10) val4)
              (cReduceCase cs1 c0 cs3 val4 ls6 17 ls8 19 ls10 (list cla=>>list cla=>>valu=>>algsuccess) _5)
              (cCleanCase cs1 c0 cs3 val4 ls6 17 ls8 19 ls10 (list cla=>>list cla=>>valu=>>algsuccess) _5)
            ]
            ]
          ]
        ]
      ]
    ]
  ]
  (cElimCase c0 cs1 cs3 val4 (list cla=>>list cla=>>valu=>>algsuccess) _5)
]

cConflictCase = [cs0, cs1, val2] csuccessZero cderivableZero

cElimConflictCase = [10, c1, cs2, val3] csuccessZero (cderivableThree(CC 10::) (opposite 10) cderivableZero

UnitCase = [10, c1, ls2, ls3, cs4, cs5, val6, (list cla=>>list cla=>>valu=>>algsuccess) _7]
  [if ((list cla=>>list cla=>>valu=>>algsuccess) _7 (remccl(CC 10:: cs4++remccl(CC 10::cs5) (Nil cla) (conclv 10 val6)))
    (cderivableEight) csuccessZero (cderivableTwo 10 algderivable8)) csuccessOne
  ]
cReduceCase = [cs0, c1, cs2, val3, ls4, 15, ls6, 17, ls8, (list cla=>>list cla=>>valu=>>algsuccess) _9, 110]
  [if ( (list cla=>>list cla=>>valu=>>algsuccess) _9

256
```
(\textbf{CleanCase} = [\text{cs0}, \text{c1, cs2}, \text{val}3, 15, 15, 16, 16, 15, 15, \text{ls}8, \text{Cont} \text{Cla} => \text{Val} \Rightarrow (\text{List} \text{Cla} => \text{Val} \Rightarrow \text{AlgSuccess}) \Rightarrow \text{AlgSuccess}])

(\textbf{ElimCase} = [\text{c0}, \text{c1, cs2}, \text{val}3, 15, 15, 15, 15, \text{ls}8, \text{Cont} \text{Cla} => \text{Val} \Rightarrow (\text{List} \text{Cla} => \text{Val} \Rightarrow \text{AlgSuccess}) \Rightarrow \text{AlgSuccess})]

(\textbf{SplitCase} = [\text{cs0}, \text{val}1, \text{c2}, \text{cs3}, 15, 15, 15, 15, \text{ls}8, \text{Cont} \text{Cla} => \text{Val} \Rightarrow (\text{List} \text{Cla} => \text{Val} \Rightarrow \text{AlgSuccess}) \Rightarrow \text{AlgSuccess})]

(\textbf{ModelCase} = [\text{val}0, \text{Val} \Rightarrow \text{AlgSuccess}])
Lawrence, Berger and Seisenberger

[10, 11, 12, ls3, cs4, cs5, algderivable6]
(Rec algderivable=>algderivable)
algderivable6
derivableZero
([c7, 18, algderivable6] derivableOne c7 18)
([l7, algderivable8] derivableTwo l7)
([c7, 18, algderivable9] derivableThree c7 18)
([l7, algderivable8, algderivable9] derivableFour l7)
derivableZero =
[ls0]
(Rec list lit=>valu=>algindmem)
ls0
([val2] cindmemOne)
{
  [l2, ls3, (valu=>algindmem)_4, val5]
  [if ((valu=>algindmem)_4 val5)
      [l6]
      [if (memlv(opposite l2) val5)
          (cindmemZero l2)
          (cindmemZero l6)
      ]
  ]
  [if (memlv(opposite l2) val5)
      (cindmemZero 12)
      (cindmemZero 16)
      ]
}
cindmemOne
}
A Simply Typed $\lambda$-Calculus of Forward Automatic Differentiation

Oleksandr Manzyuk$^{1,2}$

Department of Computer Science
National University of Ireland Maynooth
Maynooth, Ireland

Abstract

We present an extension of the simply typed $\lambda$-calculus with pushforward operators. This extension is motivated by the desire to incorporate forward automatic differentiation, which is an important technique in numeric computing, into functional programming. Our calculus is similar to Ehrhard and Regnier’s differential $\lambda$-calculus, but is based on the differential geometric idea of pushforward rather than derivative. We prove that, like the differential $\lambda$-calculus, our calculus can be soundly interpreted in differential $\lambda$-categories.

Keywords: tangent bundle, differential $\lambda$-calculus, differential category, categorical semantics.

1 Introduction

Automatic differentiation (AD) is a powerful technique for computing derivatives of functions given by programs in programming languages [6]. AD is superior to divided differences because AD-generated derivative values are free of approximation errors, and superior to symbolic differentiation because it can handle code of very high complexity and because it gives strong computational complexity guarantees. There exist many AD systems$^3$ (in the form of libraries and code pre-processors). A majority of these AD systems are built on top of imperative programming languages (Fortran, C/C++), whereas the idea of AD is most naturally embodied in a functional programming language. Indeed, the differentiation operator is almost a paradigmatic example of a higher-order function. However, despite a huge body of

---

$^1$ This work was supported, in part, by Science Foundation Ireland Principal Investigator grant 09/IN.1/I2637.

$^2$ Email: manzyuk@gmail.com

$^3$ http://www.autodiff.org/?module=Tools

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.com/locate/entcs
research and proliferation of AD implementations, a clear semantics of AD in the presence of first-class functions is lacking, which inhibits the incorporation of AD into functional programming. Siskind and Pearlmutter [12] discuss the problems one faces trying to extend a functional programming language with AD operators. In particular, they emphasize the subtleties of AD of higher-order functions. They describe [13] a novel AD system, StALIN∇, and claim that it correctly handles higher-order functions. Although StALIN∇ does produce the correct answers for examples where the other systems are known to fail, no correctness results are proven, which is hardly satisfactory.

In order to address these issues and to lay down a theoretical foundation for a functional programming language with support for AD, we suggest extending the λ-calculus with AD operators. In this paper, we make some first steps towards this goal. There are several variations of AD: forward, reverse, and mixtures thereof. We present a simply typed λ-calculus of forward AD, leaving the more complex reverse AD for future work.

The idea of extending the λ-calculus with differential operators is not novel. Drawing motivation from linear logic, Ehrhard and Regnier introduced the differential λ-calculus [4]. Despite its origin in the denotational semantics of linear logic, the differential λ-calculus is also an attractive foundation on which to build a functional programming language with built-in support for differentiation. Unlike, for example, symbolic differentiation, the differential λ-calculus can handle not only mathematical expressions, but arbitrary λ-terms. Most notably, it can take derivatives through and of higher-order functions. However, like symbolic differentiation, the differential λ-calculus, implemented naively, yields a grossly inefficient way to compute derivatives, suffering from the loss of sharing.

We propose a variation of the differential λ-calculus, the perturbative λ-calculus, which, we conjecture, does not necessitate this loss of efficiency. Like forward AD, the perturbative λ-calculus is based on the differential geometric idea of pushforward rather than that of derivative. Like the differential λ-calculus, the perturbative λ-calculus can be interpreted in an arbitrary differential λ-category of Bucciarelli et al. [3]. We prove that the interpretation is sound. We believe that the proofs of confluence and strong normalization for the differential λ-calculus given in [4] can be adapted to the perturbative λ-calculus. However, a formal treatment of these questions is left for future research.

2 Forward AD

To motivate the definition of the perturbative λ-calculus, we briefly outline the ideas behind forward AD. They are most naturally explained in differential geometric terms. Let TX denote the tangent bundle of a smooth manifold X. For example, the tangent bundle T R^n of the Euclidean space R^n can be identified with the cartesian product R^n × R^n. An element (x', x) of the tangent bundle T R^n is viewed

---

4 The publication database of the website http://www.autodiff.org contains 1277 entries at the moment of writing.
as a primal $x \in \mathbb{R}^n$ paired with a tangent (or perturbation) $x' \in \mathbb{R}^n$. A smooth map between smooth manifolds $f : X \to Y$ gives rise to a smooth map $Tf : TX \to TY$, called the pushforward of $f$. For example, the pushforward $Tf : T\mathbb{R}^m = \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^n \times \mathbb{R}^n = T\mathbb{R}^n$ of a smooth map $f : \mathbb{R}^m \to \mathbb{R}^n$ is given by $Tf(x', x) = (J_f(x) \cdot x', f(x))$, where $J_f(x)$ is the Jacobian of $f$ at the point $x$. The correspondences $X \mapsto TX$, $f \mapsto Tf$ constitute a functor from the category of smooth manifolds to itself. Preservation of composition is a consequence of the chain rule. Furthermore, $T$ preserves products. Informally, this means that in order to compute the pushforward of a compound function it suffices to know the pushforwards of its constituents. The implementations of forward AD take advantage of this, typically in one of the following ways:

- By overloading the primitives, so that they can accept both numbers and tangent bundle pairs as inputs. Each overloaded primitive denotes two functions: the function computed originally by that primitive and its pushforward. Then any user-defined procedure built out of the overloaded primitives also denotes (and can be used to compute by supplying arguments of appropriate types) two functions: a function $f : \mathbb{R}^m \to \mathbb{R}^n$ and its pushforward $Tf : T\mathbb{R}^m \to T\mathbb{R}^n$.

- By generating (either internally by the compiler or externally by a pre-processor) the source code of a procedure computing the pushforward $Tf : T\mathbb{R}^m \to T\mathbb{R}^n$ of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ from the source code of a procedure computing the function $f$. This transformational approach can be seen as an enhancement of symbolic differentiation that recovers sharing by simultaneously manipulating the primal and tangent values.

In either way, defining in a program a procedure computing a function $f$ automatically gives access to a procedure computing the pushforward of $f$.

Let us illustrate the method with an example covering ordinary arithmetic. Because the tangent bundle functor $T$ preserves products, it follows that the pushforwards $T(+), T(\cdot) : T\mathbb{R} \times T\mathbb{R} \simeq T(\mathbb{R} \times \mathbb{R}) \to T\mathbb{R}$ of the operations $+, \cdot : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ equip $T\mathbb{R}$ with the structure of a ring. The tangent bundle $T\mathbb{R}$ is isomorphic as a ring to the ring $\mathbb{R}[\varepsilon]/(\varepsilon^2)$ of dual numbers via $(x', x) \mapsto x + x'\varepsilon$, and below we identify $T\mathbb{R}$ with $\mathbb{R}[\varepsilon]/(\varepsilon^2)$. The operations $+$ and $\cdot$ in $T\mathbb{R}$ correspond exactly to the pushforwards of the operations $+$ and $\cdot$ in $\mathbb{R}$. Hence, we can overload $+$ and $\cdot$ to mean both. Computing the pushforward of a function built out of $+$ and $\cdot$ amounts to interpreting the body of the function over the dual numbers.

For example, the derivative of $\lambda x. x^2 + 1$ at the point 3 is obtained by computing the pushforward of $\lambda x. x^2 + 1$ at the point $3 + 1\varepsilon = 3 + \varepsilon$ and taking the perturbation part of the result. The former amounts to evaluating the expression $x^2 + 1$ at the point $3 + \varepsilon$ interpreting $+$ and $\times$ as addition and multiplication of dual numbers, respectively:

$$T(\lambda x. x^2 + 1)(3 + \varepsilon) = (\lambda x. x^2 + 1)(3 + \varepsilon) = (3 + \varepsilon) \cdot (3 + \varepsilon) + 1 = 10 + 6\varepsilon.$$ 

In other words, the derivative of a function $f$ at a point $x$ can be computed as $Df x = E(f(x + \varepsilon))$, where $E$ is given by $E(x + x\varepsilon) = x'$. 

261
The story becomes more complicated in the presence of first-class functions because the spaces of procedure inputs and outputs need no longer be Euclidean and may be function spaces instead. Furthermore, as pointed out by Siskind and Pearlmutter [12], implementations must be careful not to confuse the perturbations that arise when pushing the same function forward multiple times.

We illustrate this problem of perturbation confusion with an example involving nested invocations of the derivative operator $D$, for example $D (\lambda x. x \cdot D (\lambda y. x) 2) 1$. Because the inner derivative is equal to 0, the value of the whole expression should also be 0. However, applying the formula for $D$ naively, we obtain:

$$
D (\lambda x. x \cdot D (\lambda y. x) 2) 1 = E \left((\lambda x. x \cdot D (\lambda y. x) 2)(1 + \varepsilon)\right)
= E \left((1 + \varepsilon) \cdot D (\lambda y. (1 + \varepsilon)) 2\right)
= E \left((1 + \varepsilon) \cdot E ((\lambda y. (1 + \varepsilon))(2 + \varepsilon))\right)
= E \left((1 + \varepsilon) \cdot E (1 + \varepsilon)\right)
= E \left((1 + \varepsilon) \cdot 1\right) = E (1 + \varepsilon) = 1 \neq 0.
$$

As explained in [12], the root of this error is our failure to distinguish between the perturbations introduced by the inner and outer invocations of $D$. There are ways to solve this problem, for instance by tagging perturbations with a fresh $\varepsilon$ every time $D$ is invoked and incurring the bookkeeping overhead of keeping track of which $\varepsilon$ is associated with which invocation of $D$. Nonetheless, we hope the example serves to illustrate the value and nontriviality of a clear semantics for a $\lambda$-calculus with AD.

What do we mean when we say that symbolic differentiation suffers from the loss of sharing? And how does forward AD fix this problem? Let us illustrate with an example. Consider the problem of computing the derivative of a product of $n$ functions: $f(x) = f_1(x) \cdot f_2(x) \cdots \cdot f_n(x)$. Applying the product rule, we arrive at the expression for the derivative, which has size quadratic in $n$:

$$
f'(x) = f'_1(x) \cdot f_2(x) \cdots \cdot f_n(x) + f_1(x) \cdot f'_2(x) \cdots \cdot f_n(x) + \cdots + f_1(x) \cdot f_2(x) \cdots \cdot f'_n(x).
$$

Evaluating it naively would result in evaluating each $f_i(x)$ $n - 1$ times. If our cost model is that evaluating $f_i(x)$ or $f'_i(x)$ each cost 1, and the arithmetic operations are free, then $f(x)$ has a cost of $n$, whereas $f'(x)$ has a cost of $n^2$. Contrast this with forward AD: the pushforward $Tf(x + x'\varepsilon)$ is the product (in the sense of dual numbers) of the pushforwards $Tf_1(x + x'\varepsilon)$, $Tf_2(x + x'\varepsilon)$, ..., $Tf_n(x + x'\varepsilon)$. Evaluating each $Tf_i(x + x'\varepsilon)$ amounts to evaluating $f_i(x)$ and $f'_i(x)$, and hence has a cost of 2. Therefore, the cost of $Tf(x + \varepsilon)$ is $2n$. In general, forward AD guarantees that evaluating $f'$ takes no more than a constant factor times as many operations as evaluating $f$.

3 Tangent Bundles in Differential $\lambda$-Categories

The definition of the interpretation of the perturbative $\lambda$-calculus as well as the proof of its soundness rely on the theory of tangent bundles in differential $\lambda$-categories.
developed in [9]. To make this paper self-contained, we summarize the results of that theory here. The reader is referred to [9] for more details.

Let \( C \) be a cartesian category and \( X, Y, Z \) objects of \( C \). We denote by \( X \times Y \) the product of \( X \) and \( Y \) and by \( \pi_1 : X \times Y \to X, \pi_2 : X \times Y \to Y \) the projections. The terminal object is denoted by \( 1 \), and for any object \( X \), we denote by \( !_X \) the unique morphism from \( X \) to \( 1 \). For a pair of morphisms \( f : Z \to X \) and \( g : Z \to Y \), denote by \( \langle f, g \rangle : Z \to X \times Y \) the pairing of \( f \) and \( g \), i.e., the unique morphism such that \( \pi_1 \circ \langle f, g \rangle = f \) and \( \pi_2 \circ \langle f, g \rangle = g \).

A cartesian category \( C \) is called closed if for any pair of objects \( X \) and \( Y \) of \( C \) there exists an object \( X \Rightarrow Y \), called the exponential object, and a morphism \( \text{ev}_{X,Y} : (X \Rightarrow Y) \times X \to Y \), called the evaluation morphism, satisfying the following universal property: the map \( \Lambda : C(Z, X \Rightarrow Y) \to C(Z \times X, Y) \) given by \( \Lambda(g) = \text{ev}_{X,Y} \circ (g \times \text{id}_X) \) is bijective. Let \( \Lambda : C(Z \times X, Y) \to C(Z, X \Rightarrow Y) \) denote the inverse of \( \Lambda \). In other words, for a morphism \( f : Z \times X \to Y \), \( \Lambda(f) : Z \to X \Rightarrow Y \) is the unique morphism such that \( \text{ev}_{X,Y} \circ (\Lambda(f) \times \text{id}_X) = f \). The morphism \( \Lambda(f) \) is called the currying of \( f \). We shall make use of the equations

\[
\Lambda(f) \circ g = \Lambda(f \circ (g \times \text{id})), \tag{1}
\]
\[
\text{ev} \circ (\Lambda(f), g) = f \circ \langle \text{id}, g \rangle, \tag{2}
\]

which follow immediately from the definition of \( \Lambda \).

The notion of cartesian differential category was introduced by Blute et al. [2] as an axiomatization of differentiable maps as well as a unifying framework in which to study different notions reminiscent of the differential calculus.

**Definition 3.1 (\cite[Definition 1.1.1]{2})** A category \( C \) is left-additive if each hom-set is equipped with the structure of a commutative monoid \((C(X,Y), +, 0)\) such that \((g + h) \circ f = (g \circ f) + (h \circ f)\) and \(0 \circ f = 0\). A morphism \( f \) in \( C \) is additive if it satisfies \( f \circ (g + h) = (f \circ g) + (f \circ h) \) and \( f \circ 0 = 0 \).

**Definition 3.2 (\cite[Definition 1.2.1]{2})** A category is cartesian left-additive if it is a left-additive category with products such that all projections are additive, and all pairings of additive morphisms are additive.

**Remark 3.3** Let \( C \) be a cartesian left-additive category. Then the pairing map \( \langle - , - \rangle : C(Z, X) \times C(Z, Y) \to C(Z, X \times Y) \) is additive: \( \langle f + g, h + k \rangle = \langle f, h \rangle + \langle g, k \rangle \) and \( \langle 0, 0 \rangle = 0 \). Furthermore, for each pair of objects \( X \) and \( Y \) there are morphisms \( \iota_1 \overset{\text{def}}{=} \langle \text{id}_X, 0 \rangle : X \to X \times Y \) and \( \iota_2 \overset{\text{def}}{=} \langle 0, \text{id}_Y \rangle : Y \to X \times Y \), which satisfy the equations \( \pi_1 \circ \iota_1 = \text{id}_X, \pi_2 \circ \iota_2 = \text{id}_Y, \pi_k \circ \iota_l = 0 \) if \( k \neq l \), and \( \iota_1 \circ \pi_1 + \iota_2 \circ \pi_2 = \text{id}_{X \times Y} \). Note, however, that in contrast with additive categories, in a left-additive category these equations do not imply that the morphisms \( \iota_1 \) and \( \iota_2 \) equip \( X \times Y \) with the structure of a coproduct of \( X \) and \( Y \). This is so on the subcategory of additive morphisms, but not necessarily on the full category.

**Definition 3.4 (\cite[Section 1.4, Definition 2.1.1]{2}, \cite[Definition 4.2]{3})** A cartesian closed category is cartesian closed left-additive if it is a cartesian left-additive category such that each currying map \( \Lambda : C(Z \times X, Y) \to C(Z, X \Rightarrow Y) \)
is additive: $\Lambda(f + g) = \Lambda(f) + \Lambda(g)$ and $\Lambda(0) = 0$. A cartesian (closed) differential category is a cartesian (closed) left-additive category equipped with an operator $D : C(X, Y) \to C(X \times X, Y)$ satisfying the following axioms:

D1. $D(f + g) = D(f) + D(g)$ and $D(0) = 0$.
D2. $D(f) \circ \langle h + k, v \rangle = D(f) \circ \langle h, v \rangle + D(f) \circ \langle k, v \rangle$ and $D(f) \circ \langle 0, v \rangle = 0$.
D3. $D(\text{id}) = \pi_1, D(\pi_1) = \pi_1 \circ \pi_1, D(\pi_2) = \pi_2 \circ \pi_1$.
D4. $D(\langle f, g \rangle) = \langle D(f), D(g) \rangle$.
D5. $D(f \circ g) = D(f) \circ (D(g), g \circ \pi_2)$.
D6. $D(D(f)) \circ \langle \langle 0, h \rangle, \langle h, k \rangle \rangle = D(f) \circ \langle \langle 0, g \rangle, \langle h, k \rangle \rangle$.
D7. $D(D(f)) \circ \langle \langle 0, h \rangle, \langle h, k \rangle \rangle = D(D(f)) \circ \langle \langle 0, g \rangle, \langle h, k \rangle \rangle$.

The paradigmatic example of a cartesian differential category is the category of smooth maps, whose objects are natural numbers and morphisms $m \to n$ are smooth maps $\mathbb{R}^m \to \mathbb{R}^n$. The operator $D$ takes an $f : \mathbb{R}^m \to \mathbb{R}^n$ and produces a $D(f) : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^n$ given by $D(f)(x', x) = J_f(x) \cdot x'$, where $J_f(x)$ is the Jacobian of $f$ at the point $x$.

Some intuition for the axioms: D1 says $D$ is additive; D2 that $D(f)$ is additive in its first coordinate; D3 and D4 assert that $D$ is compatible with the product structure, and D5 is the chain rule. We refer the reader to [2, Lemma 2.2.2] for the proof that D6 is essentially requiring that $D(f)$ be linear (in the sense defined below) in its first variable. D7 is essentially independence of order of partial differentiation.

Following [2], we say that a morphism $f$ is linear if $D(f) = f \circ \pi_1$. By [2, Lemma 2.2.2], the class of linear morphisms is closed under sum, composition, pairing, and product, and contains all identities, projections, and zero morphisms. Also, axiom D2 implies that any linear morphism is additive.

The notion of cartesian differential category was partly motivated by a desire to model the differential $\lambda$-calculus of Erhard and Regnier [4] categorically. Blute et al. [2] proved that cartesian differential categories are sound and complete to model suitable term calculi. However, the properties of cartesian differential categories are too weak for modeling the full differential $\lambda$-calculus because the differential operator is not necessarily compatible with the cartesian closed structure. For this reason, Bucciarelli et al. [3] introduced the notion of differential $\lambda$-category.

**Definition 3.5 ([3, Definition 4.4])** A differential $\lambda$-category is a cartesian closed differential category such that $D(\Lambda(f)) = \Lambda(D(f) \circ \langle \pi_1 \times 0_X, \pi_2 \times \text{id}_X \rangle)$ holds, for each $f : Z \times X \to Y$.

This differential $\lambda$-category axiom is essentially requiring that the evaluation morphism $\text{ev}$ be linear in its first argument.

**Example 3.6** Blute et al. [1] proved that the category of convenient vector spaces and smooth maps is a cartesian differential category. We have shown in [9] that it is in fact a differential $\lambda$-category. We refer the reader to [3] for two other examples of differential $\lambda$-categories.
The differential operator $D$ allows us to replicate the construction of the tangent bundle of a smooth manifold from differential geometry in any cartesian differential category. Let $\mathcal{C}$ be a cartesian differential category. The tangent bundle functor $T : \mathcal{C} \to \mathcal{C}$ is defined by $TX = X \times X$ and $T(f) = \langle D(f), f \circ \pi_2 \rangle$.

**Lemma 3.7 ([9, Lemma 3.1.3])** If $f$ is linear, then $T(f) = f \times f$.

The tangent bundle functor $T$ is part of a monad. For each object $X$ of $\mathcal{C}$, the unit $\eta_X$ of the monad is the morphism $\langle 0, \text{id}_{X} \rangle : X \to X \times X = TX$, and the multiplication $\mu_X$ of the monad is the morphism $\langle \pi_2 \circ \pi_1 + \pi_1 \circ \pi_2, \pi_2 \circ \pi_2 \rangle : TTX = (X \times X) \times (X \times X) \to X \times X = TX$. The monad $(\mu, \eta, \mu)$ is strong [10, Definition 3.2].

The tensorial strength $t_{X,Y} : X \times TY \to T(X \times Y)$, also called the right tensorial strength, is given by $t_{X,Y} = \langle (0, \pi_1 \circ \pi_2), \pi_1, \pi_2 \circ \pi_2 \rangle = \langle 0 \times \pi_1, \text{id}_{X} \circ \pi_2 \rangle$. Using the symmetry $c_{A,B} = \langle \pi_2, \pi_1 \rangle : A \times B \to B \times A$ of $\mathcal{C}$, we may also define the left tensorial strength by $t'_{X,Y} : T(Y,X) \circ t_{Y,X} \circ c_{TX,Y} : TX \times Y \to T(X \times Y)$. More explicitly, $t'_{X,Y} = \langle \langle \pi_1 \circ \pi_1, 0 \rangle, \langle \pi_2 \circ \pi_1, \pi_2 \circ \pi_1 \rangle \rangle = \langle \pi_1 \times 0, \pi_2 \times \text{id}_{Y} \rangle$.

Because the functor $T$ is part of a strong monad, by [7, Theorem 2.1] $T$ becomes a monoidal functor $(T, \psi, \psi^0) : \mathcal{C} \to \mathcal{C}$ if we put $\psi_{XY} = \mu_{X \times Y} \circ T(t_{X,Y}) \circ T(t'_{X,Y}) : TX \times TY \to T(X \times Y)$ and by putting $\psi^0 = \eta_\mathbb{1} : \mathbb{1} \to T\mathbb{1}$. The definition of $\psi_{XY}$ is asymmetric, and indeed there is also a morphism $\tilde{\psi}_{XY} = \mu_{X \times Y} \circ T(t'_{X,Y}) \circ T(t_{X,Y}) : TX \times TY \to T(X \times Y)$ that also makes $T$ into a monoidal functor. The morphisms $\psi$ and $\tilde{\psi}$ can be computed explicitly. It is shown in [9, Lemmas 3.4.1, 3.4.2] that $\psi$ and $\tilde{\psi}$ are equal and coincide with the distributivity isomorphism $\sigma$. In particular, $(\mu, \eta, \mu)$ is a commutative monad [7, Definition 3.1].

**Lemma 3.8 ([9, Lemma 3.4.5])** Let $f : Z \to X$, $g : Z \to Y$ be morphisms in $\mathcal{C}$. Then $T(f,g) = \psi \circ \langle T(f), T(g) \rangle$.

From now on we assume that $\mathcal{C}$ is a differential $\lambda$-category.

**Proposition 3.9 ([9, Proposition 3.6.2])** Let $g : A \times B \to C$ be a morphism in $\mathcal{C}$. Let $h = T(g) \circ \tau : TA \times B \to TC$. Then $T(\Lambda(g)) = \langle \Lambda(\pi_1 \circ h), \Lambda(\pi_2 \circ h) \rangle : TA \to T(B \Rightarrow C)$.

The cartesian closed category $\mathcal{C}$ is a symmetric monoidal closed category, with the monoidal structure given by product, and hence also a closed category of Eilenberg and Kelly [5]. By [5, Proposition 4.3], the monoidal functor $(T, \psi, \psi^0) : \mathcal{C} \to \mathcal{C}$ gives rise to a closed functor $(\hat{T}, \hat{\psi}, \hat{\psi}^0) : \mathcal{C} \to \mathcal{C}$, where $\hat{\psi} = \psi_{XY} : X \Rightarrow Y \to (TX \Rightarrow TY)$ is given by $\hat{\psi} = \Lambda(T(\text{ev}) \circ \psi)$. We have shown in [9, Section 3.7] that

$$T(\text{ev}) \circ \hat{\psi} = \langle \text{ev} \circ (\pi_1 \times \pi_2) + D(\text{ev}) \circ \circ (\pi_2 \times \text{id}), \text{ev} \circ (\pi_2 \times \pi_2) \rangle : T(X \Rightarrow Y) \times TX \to TY. \tag{3}$$

The cartesian closed category $\mathcal{C}$ gives rise to a category $\mathcal{C}$ enriched in $\mathcal{C}$: the objects of $\mathcal{C}$ are the objects of $\mathcal{C}$, and $\mathcal{C}(X,Y) = X \Rightarrow Y$, for each pair of object $X$ and $Y$ of $\mathcal{C}$. By [8, Theorem 1.3], the functor $T : \mathcal{C} \to \mathcal{C}$ equipped with the
Manzyuk
tensorial strength $t$ gives rise to a $\mathbf{C}$-functor $T : \mathbf{C} \to \mathbf{C}$ such that $TX = TX$ and $T = T_{X,Y} : (X \Rightarrow Y) \to (TX \Rightarrow TY)$ is given by $T = \Lambda(T(\text{ev}) \circ t)$.

Theorem 3.10 ([9, Theorem 3.8.2]) $T$ is a linear morphism.

Proposition 3.11 ([9, Proposition 3.8.3]) Let $f : Z \times X \to Y$ be a morphism in $\mathbf{C}$. Then $T \circ \Lambda(f) = \Lambda(T(f) \circ t)$.

We shall also need the following definition and easy lemma.

Definition 3.12 ([3, Definition 4.6]) Let $sw = sw_{X,Y,Z}$ denote the morphism $sw = \langle \langle \pi_1 \circ \pi_1, \pi_2 \circ \pi_1 \rangle, \pi_2 \circ \pi_1 \rangle : (X \times Y) \times Z \to (X \times Z) \times Y$. Clearly, $sw$ is a linear morphism. Furthermore, $sw \circ \langle \langle f, g \rangle, h \rangle = \langle \langle f, h \rangle, g \rangle$.

Lemma 3.13 $T(sw) \circ t' \circ (t \times \text{id}) = t \circ sw : (X \times TZ) \times Y \to T((X \times Y) \times Z)$.

4 Perturbative $\lambda$-Calculus

In this section we describe the perturbative $\lambda$-calculus. Its syntax is very similar to the syntax of the differential $\lambda$-calculus of Ehrhard and Regnier [4] with two notable differences:

- Instead of introducing a syntactic form $D_s \cdot t$ denoting the derivative of $s$ in direction $t$, we extend the $\lambda$-calculus with a syntactic form $Ts$ denoting the pushforward of $s$.
- To syntactically enforce the fact that pairs are linear, instead of introducing a syntax for pairs, we introduce two syntactic forms: $\iota_k s$ (injection of $s$ into the $k$th factor of a product) and $\pi_k s$ (projection of $s$ onto the $k$th coordinate), $k = 1, 2$.

With the syntax for injections, pairs can be introduced as syntactic sugar and linearity is automatic.

More formally, the set $\Lambda^p$ of perturbative $\lambda$-terms and the set $\Lambda^s$ of simple $\lambda$-terms are defined by mutual induction as follows:

\[
\Lambda^p : \quad M, N ::= 0 \mid s \mid s + N,
\]

\[
\Lambda^s : \quad s, t ::= x \mid \lambda x. s \mid sN \mid Ts \mid \iota_k s \mid \pi_k s, \quad k = 1, 2,
\]

Thus, every perturbative $\lambda$-term is a formal sum of a finite (possibly empty) bag of simple $\lambda$-terms. The operation $+$ is extended to arbitrary perturbative $\lambda$-terms in the obvious way as the union of bags. The term $0$ denoting the empty bag is the neutral element of the sum. We consider perturbative $\lambda$-terms up to $\alpha$-conversion, associativity and commutativity of the sum. We write $M \equiv N$ if $M$ and $N$ are syntactically equal up to the aforementioned equivalences. The term $Ts$ is the pushforward of $s$. The set $\text{FV}(M)$ of free variables of $M$ is defined as usual. We introduce the following syntactic sugar:

\[
\lambda x. \left( \sum_{i=1}^n s_i \right) \overset{\text{def}}{=} \sum_{i=1}^n \lambda x. s_i, \quad \iota_k \left( \sum_{i=1}^n s_i \right) \overset{\text{def}}{=} \sum_{i=1}^n \iota_k s_i,
\]

\[\sum_{i=1}^n s_i\]
we may assume that usual caveat applies to the definition of \( M \), rules of the simply typed \( \lambda \)-calculus. Note also tangent bundle, which would not in general be definable as \( \sigma \) be \(-\)categories, for example, the category of convenient vector spaces and smooth maps. The tangent bundle \(-\)calculus. We assume that we are given some atomic types \( \sigma \). Let us define the type system that characterizes the simply typed perturbative \( \lambda \)-terms and \( \sigma \) are types, then so are \( M \), denoted by \( \Gamma \vdash x : \sigma \). Let \( \Gamma \vdash N : \sigma \). Fig. 1. Typing rules of the perturbative \( \lambda \)-calculus

\[
\begin{align*}
\Gamma(x) &= \sigma \\
\Gamma \vdash x : \sigma \\
\Gamma \vdash \lambda x. s : \sigma \rightarrow \tau \\
\Gamma \vdash s : \sigma \rightarrow \tau & \quad \Gamma \vdash N : \sigma \\
\Gamma \vdash sN : \tau \\
\Gamma \vdash 0 : \sigma \\
\Gamma \vdash s + N : \sigma \\
\Gamma \vdash \iota k s : \sigma_1 \times \sigma_2 \\
\Gamma \vdash \pi k s : \sigma_k \\
\Gamma \vdash s : \sigma \rightarrow \tau \\
\Gamma \vdash T s : T \sigma \rightarrow T \tau
\end{align*}
\]

Fig. 2. Definition of substitution

\[
\begin{align*}
\Gamma(\sum_{i=1}^{n} s_i) & \text{ def } \sum_{i=1}^{n} \Gamma S_i \\
\sum_{i=1}^{n} s_i \text{ def } \sum_{i=1}^{n} s_i N \\
\langle M_1, M_2 \rangle & \text{ def } \iota_1 M_1 + \iota_2 M_2
\end{align*}
\]

where the sums reduce to the term 0 if \( n = 0 \). Note that the terms in the left hand sides of the above equations are not valid terms in the language. They are abbreviations for the respective terms in the right hand sides of the equations. Note also that this way we syntactically capture the linearity of abstractions, pushforwards, injections, projections, pairs, and applications in the operator position.

Let us define the type system that characterizes the simply typed perturbative \( \lambda \)-calculus. We assume that we are given some atomic types \( \alpha, \beta, \ldots \), and if \( \sigma \) and \( \tau \) are types, then so are \( \sigma \times \tau \) and \( \sigma \rightarrow \tau \). We define\(^5\) a type function \( T \) by \( T \sigma = \sigma \times \sigma \). The typing rules are shown in Figure 1. They are the standard typing rules of the simply typed \( \lambda \)-calculus with products, extended by the typing rules for \( T \) and the sum.

Let \( M, P \) be perturbative \( \lambda \)-terms and \( x \) a variable. The substitution of \( P \) for \( x \) in \( M \), denoted by \( M[P/x] \), is defined by induction on \( M \) as shown in Figure 2. The usual caveat applies to the definition of \( (\lambda y. s)[P/x] \), namely that by \( \alpha \)-conversion we may assume that \( x \neq y \) and \( y \notin \text{FV}(P) \).

\(^5\) The definition of \( T \) is motivated by our desire to interpret the perturbative \( \lambda \)-calculus in differential \( \lambda \)-categories, for example, the category of convenient vector spaces and smooth maps. The tangent bundle of a convenient vector space \( E \) is isomorphic to \( E \times E \), which we reflect in the calculus by defining \( T \sigma \) to be \( \sigma \times \sigma \). It would be interesting to generalize \( T \) to allow types \( \sigma \) to be general smooth spaces and \( T \sigma \) the tangent bundle, which would not in general be definable as \( \sigma \times \sigma \). We leave this for future work.

267
Manzyuk

\[(\lambda x. s)N \rightarrow_{\beta} s[N/x] \quad \pi_k(\iota_k s) \rightarrow_{\times} \times s \quad \pi_k(\iota_l s) \rightarrow_{\times} \times 0 \quad T(\lambda x. s) \rightarrow T \lambda x. T_x s\]

Fig. 3. Reduction rules of the perturbative \(\lambda\)-calculus

\[T_x y = \begin{cases} 
  x & \text{if } x = y \\
  \iota_2 y & \text{otherwise}
\end{cases} \quad T_x 0 = 0\]

\[T_x (\lambda y. s) = \langle \lambda y. \pi_1(T_x s), \lambda y. \pi_2(T_x s) \rangle \quad T_x (s + N) = T_x s + T_x N\]
\[T_x (sN) = (T_x s) \circ (T_x N) \quad T_x (\iota_k s) = \langle \iota_k(\pi_1(T_x s)), \iota_k(\pi_2(T_x s)) \rangle\]
\[T_x (T s) = \langle T(\pi_1(T_x s)), T(\pi_2(T_x s)) \rangle \quad T_x (\pi_k s) = \langle \pi_k(\pi_1(T_x s)), \pi_k(\pi_2(T_x s)) \rangle\]

\[M \circ N \overset{\text{def}}{=} \langle (\pi_1 M)(\pi_2 N) + \pi_1((T M(s N)), (\pi_2 M)(\pi_2 N))\]

Fig. 4. Definition of \(T_x\)

We impose the reduction rules listed in Figure 3: the usual \(\beta\)-reduction \((\lambda x. s)N \rightarrow_{\beta} s[N/x]\) and projection-injection rules \(\pi_k(\iota_k s) \rightarrow_{\times} s\) and \(\pi_k(\iota_l s) \rightarrow_{\times} 0\), where \(k, l \in \{1, 2\}, k \neq l\). There is also a reduction rule for \(T\), which is similar in shape to the reduction rule for \(D\) in the differential \(\lambda\)-calculus, but is suggested by the general categorical properties of tangent bundles in differential \(\lambda\)-categories [9]. More specifically, the reduction rule for \(T\) is \(T(\lambda x. s) \rightarrow_T \lambda x. T_x s\), where \(T_x M\) is defined by induction on \(M\) according to the equations shown in Figure 4. The definition of \(T_x (\lambda y. s)\) is subject to the standard side condition, namely that by \(\alpha\)-conversion we may assume that \(x\) is different from \(y\). To simplify the notation, we have introduced some syntactic sugar: the operation \(\circ\), which is also defined in Figure 4. The typing rule for \(\circ\) can be derived from the typing rules for other syntactic forms and reads as follows:

\[\Gamma \vdash M : \mathcal{T}(\sigma \rightarrow \tau) \quad \Gamma \vdash N : \mathcal{T} \sigma\]

Let us provide some intuition for these equations. We define \(T_x x = x\) because the pushforward of the identity function is the identity function; similarly, we set \(T_x y = \iota_2 y\) if \(x \neq y\) because the pushforward of the constant function is a constant function returning the lifted value. The definitions of \(T_x\) over abstractions and applications are suggested by Proposition 3.9 and equation (3), respectively. The definitions of \(T_x\) over \(\mathcal{T}\), \(\pi_k\), and \(\iota_k\) have the given shape because \(\mathcal{T}\), \(\pi_k\), and \(\iota_k\) are linear morphisms (\(\mathcal{T}\) is linear by Theorem 3.10). Finally, the definition of \(T_x\) over sums follows from the additivity of the tangent bundle functor.

Denote by \(\rightarrow\) the contextual closure of \(\rightarrow_{\beta} \cup \rightarrow_{\times} \cup \rightarrow_T\).

**Lemma 4.1** The type system of the simply typed perturbative \(\lambda\)-calculus satisfies the following properties.

(a) If \(\Gamma; x : \sigma \vdash M : \tau\) and \(\Gamma \vdash N : \sigma\), then \(\Gamma \vdash M[N/x] : \tau\).
\[
\begin{align*}
\frac{\Gamma \vdash (\lambda x. s)N : \tau}{\Gamma \vdash (\lambda x. s)N = s[N/x] : \tau} & \quad \text{(}} \beta \text{)} \quad \frac{\Gamma \vdash \lambda x. s = \lambda x. t : \sigma \to \tau}{\Gamma \vdash \lambda x. s} \quad \text{and} \quad \Gamma \vdash \tau \to \tau \to \tau \\
\frac{\Gamma; x : \sigma \vdash s = t : \tau}{\Gamma \vdash \lambda x. s = \lambda x. t : \sigma \to \tau} & \quad \text{(}} \xi \text{)} \quad \frac{\Gamma \vdash s_1 = s_2 : \sigma \quad \Gamma \vdash N_1 = N_2 : \sigma}{\Gamma \vdash s_1N_1 = s_2N_2 : \tau} \quad \text{(} \text{Ap} \text{)} \quad \frac{\Gamma \vdash s_1 = s_2 : \sigma}{\Gamma \vdash s_1 + N_1 = s_2 + N_2 : \sigma} \quad \text{(}} \text{Sum} \text{)} \\
\frac{\text{\Gamma} \vdash s : \sigma \quad \text{\Gamma} \vdash k \text{~is~closed}}{	ext{\Gamma} \vdash \pi_k s = \pi_k t : \sigma_k} & \quad \text{(}} \pi \text{)} \quad \frac{\text{\Gamma} \vdash \text{\tau} s : \sigma \quad \text{\Gamma} \vdash \text{\tau} s \text{~is~closed}}{	ext{\Gamma} \vdash \pi_k s = \pi_k t : \sigma_k} \quad \text{(}} \pi_l \text{)} \quad \frac{\text{\Gamma} \vdash s : \sigma}{\text{\Gamma} \vdash \pi_k (\text{\tau} s) = 0 : \sigma} \\
\end{align*}
\]

Fig. 5. Perturbative λ-theory rules

(b) If \( \Gamma; x : \sigma \vdash M : \tau \), then \( \Gamma; x : \top \sigma \vdash \top x M : \top \tau \).
(c) If \( \Gamma \vdash \pi_k (\pi_k N) : \sigma \), then \( \Gamma \vdash N : \sigma \).
(d) If \( \Gamma \vdash N : \sigma \) and \( N \vdash N' \), then \( \Gamma \vdash N' : \sigma \).

\textbf{Proof.} (a) and (b) follow by straightforward induction on the length of the proof of the corresponding typing judgment. (c) is obvious. (d) follows from (a), (b), and (c) because type derivations are contextual.

To facilitate the proof that applying the reduction rules preserves the meaning, which is what soundness really means, we introduce the notion of perturbative λ-theory. Let \( \mathcal{T} \) be a collection of judgments of the shape \( \Gamma \vdash M = N : \sigma \) such that \( \Gamma \vdash M : \sigma \) and \( \Gamma \vdash N : \sigma \). \( \mathcal{T} \) is called a perturbative λ-theory if it is closed under the rules shown in Figure 5 (where \( k, l \in \{1, 2\} \) and \( k \neq l \)) together with the obvious rules for reflexivity, transitivity, and symmetry. Equations (β), (T), (πι) are imposed by the reduction rules. The remaining rules say that the equality relation is contextual.

5 Categorical Semantics

In this section we show that, like the simply typed variant of the differential λ-calculus of Ehrhard and Regnier [4], the simply typed perturbative λ-calculus can be modeled by differential λ-categories of Bucciarelli et al. [3]. We have shown in [9] that an arbitrary differential λ-category is equipped with a canonical pushforward construction. The perturbative λ-calculus captures the pushforward construction as a syntactic operation.

Let \( \mathbf{C} \) be a differential λ-category. Let us define the interpretation of the simply typed perturbative λ-calculus from the previous section in the category \( \mathbf{C} \). The definition is similar to the interpretation of the simply typed differential λ-calculus in a differential λ-category defined in [3]. Types are interpreted as follows: \( |\alpha| = A \), for some object \( A \), \( |\sigma \times \tau| = |\sigma| \times |\tau| \), and \( |\sigma \to \tau| = |\sigma| \Rightarrow |\tau| \). Contexts are
interpreted as usual: $|\emptyset| = 1$ and $|\Gamma; x : \sigma| = |\Gamma| \times |\sigma|$. The interpretation of a judgment $\Gamma \vdash M : \sigma$ will be a morphism from $|\Gamma|$ to $|\sigma|$ denoted by $|M^\sigma|_\Gamma$ and defined inductively as follows:

- $|x^\sigma|_{\Gamma; x : \sigma} = \pi_2 : |\Gamma| \times |\sigma| \to |\sigma|$;
- $|y^\sigma|_{\Gamma; x : \sigma} = |y^\sigma|_\Gamma \circ \pi_1 : |\Gamma| \times |\sigma| \to |\tau|$ for $x \neq y$;
- $|(sN)^\sigma|_\Gamma = \mathrm{ev} \circ (|s^\sigma|_\Gamma \circ |N^\sigma|_\Gamma) : |\Gamma| \to |\tau|$;
- $|(\lambda x. s)^{\sigma \to \tau}|_\Gamma = \Lambda(|s^\sigma|_\Gamma : |\Gamma| \to |\sigma|) \Rightarrow |\tau|$;
- $|(T s)^{\sigma \to \tau}|_\Gamma = T_{|\sigma|,|\tau|} \circ |s^\sigma|_\Gamma : |\Gamma| \to T|\sigma| \Rightarrow T|\tau|$;
- $|0^\sigma|_\Gamma = 0 : |\Gamma| \to |\sigma|$;
- $|(s + N)^\sigma|_\Gamma = |s^\sigma|_\Gamma + |N^\sigma|_\Gamma : |\Gamma| \to |\sigma|$;
- $|(t_k s)^{\sigma_1 \times \sigma_2}|_\Gamma = t_k \circ |s^{\sigma_k}|_\Gamma : |\Gamma| \to |\sigma_1| \times |\sigma_2|$, $k = 1, 2$;
- $|(\pi_k s)^{\sigma}|_\Gamma = \pi_k \circ |s^{\sigma_1 \times \sigma_2}|_\Gamma : |\Gamma| \to |\sigma_k|$, $k = 1, 2$.

The only interesting case is that of $T s$, which is what we were really after; the other cases are standard. It follows from the definitions that $|(M, N)|_\Gamma = (|M|_\Gamma, |N|_\Gamma)$. We shall omit the superscript $\sigma$ in $|M^\sigma|_\Gamma$ when there is no risk of confusion.

Given a differential $\lambda$-category $C$, we define the theory of $C$ by

$$\text{Th}(C) = \{ \Gamma \vdash M = T : \sigma | \Gamma \vdash M : \sigma, \; \; \; \Gamma \vdash N : \sigma, \; \; \; |M^\sigma|_\Gamma = |N^\sigma|_\Gamma \}.$$  

We are going to prove that the interpretation $|\cdot|$ is sound for the simply typed perturbative $\lambda$-calculus, i.e., that $\text{Th}(C)$ is a perturbative $\lambda$-theory. We begin by proving some lemmas. Lemmas 5.1 and 5.2 are standard.

**Lemma 5.1**  $|M|_{\Gamma; x : \sigma; y : \tau} = |M|_{\Gamma; y : \tau; x : \sigma} \circ \text{sw}$.

**Lemma 5.2**  If $\Gamma \vdash M : \tau$ and $x \notin \text{FV}(M)$, then $|M^\tau|_{\Gamma; x : \sigma} = |M^\tau|_\Gamma \circ \pi_1$.

**Lemma 5.3**  Let $\Gamma \vdash M : T(\sigma \to \tau)$ and $\Gamma \vdash N : T \sigma$. Then

$$|(M \circ N)^T|_\Gamma = T(\text{ev}) \circ \psi \circ (|M^T(\sigma \to \tau)|_\Gamma, |N^T\sigma|_\Gamma).$$

**Proof.** Expanding $M \circ N$ and applying the definition of the interpretation, we find that $|M \circ N|_\Gamma$ is equal to

$$\langle \text{ev} \circ (\pi_1 \circ |M|_\Gamma, \pi_2 \circ |N|_\Gamma) + \pi_1 \circ \text{ev} \circ (T \circ \pi_2 \circ |M|_\Gamma, |N|_\Gamma),$$

$$\text{ev} \circ \pi_2 \circ |M|_\Gamma, \pi_2 \circ |N|_\Gamma) \rangle$$

$$= \langle \text{ev} \circ (\pi_1 \times \pi_2) + \pi_1 \circ \text{ev} \circ (T \circ \pi_2 \times \text{id}), \text{ev} \circ (\pi_2 \times \pi_2) \rangle \circ (|M|_\Gamma, |N|_\Gamma).$$

The summand $\pi_1 \circ \text{ev} \circ (T \circ \pi_2 \times \text{id})$ is equal to $\pi_1 \circ T(\text{ev}) \circ t \circ (\pi_2 \times \text{id}) = D(\text{ev}) \circ t \circ (\pi_2 \times \text{id})$ by the definitions of $T$ and $T$. We conclude that $|M \circ N|_\Gamma$ is equal to $\langle \text{ev} \circ (\pi_1 \times \pi_2) + D(\text{ev}) \circ t \circ (\pi_2 \times \text{id}), \text{ev} \circ (\pi_2 \times \pi_2) \rangle \circ (|M|_\Gamma, |N|_\Gamma)$, which coincides with $T(\text{ev}) \circ \psi \circ (|M|_\Gamma, |N|_\Gamma)$ by equation (3). \hfill $\Box$

**Lemma 5.4 (Substitution)**  Let $\Gamma; x : \sigma \vdash M : \tau$ and $\Gamma \vdash P : \sigma$. Then $|(M[P/x])^\tau|_\Gamma = |M^\tau|_{\Gamma; x : \sigma} \circ (\text{id}|_\Gamma, |P^\sigma|_\Gamma)$.
Proof. The proof is by induction on $M$. The only new cases are $M \equiv T \; s$, $M \equiv t_k \; s$, and $M \equiv \pi \; k \; s$. However, they are straightforward. For example, \(|(T \; s)(P/x)|_Γ = T \circ |s(P/x)|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Γ|_Gamma
is in turn equal to $T(\Lambda(\lambda y. s|_{\Gamma; x; \sigma})) \circ t = T(\lambda y. s|_{\Gamma; x; \sigma}) \circ t$ by Lemma 5.1.

- $M \equiv \mathsf{T}s$. By the definitions of $\mathsf{T}_x$ and $| \cdot |$, and by the induction hypothesis we have $|\mathsf{T}_x(Ts)|_{\Gamma; x; \tau} = |(\mathsf{T}(\pi_1(\mathsf{T}_x s)), \mathsf{T}(\pi_2(\mathsf{T}_x s)))|_{\Gamma; x; \tau} = (\mathsf{T} \circ \pi_1 \circ (\mathsf{T}_x s)|_{\Gamma; x; \tau}, \mathsf{T} \circ \pi_2 \circ (\mathsf{T}_x s)|_{\Gamma; x; \tau}) = (\mathsf{T} \times \mathsf{T}) \circ (\mathsf{T}_x s)|_{\Gamma; x; \tau} = (\mathsf{T} \times \mathsf{T}) \circ T(|s|_{\Gamma; x; \sigma}) \circ t$. The morphism $\mathsf{T}$ is linear by Theorem 3.10. Therefore $\mathsf{T} \times \mathsf{T} = \mathsf{T}(\mathsf{T})$ by Lemma 3.7. We conclude that $|\mathsf{T}_x(Ts)|_{\Gamma; x; \tau} = T(\mathsf{T}) \circ T(|s|_{\Gamma; x; \sigma}) \circ t = T(\mathsf{T} \circ |s|_{\Gamma; x; \sigma}) \circ t = T(|s|_{\Gamma; x; \sigma}) \circ t$ by the functoriality of $T$.

- The remaining cases ($M \equiv 0$, $M \equiv s + N$, $M \equiv \iota_k s$, and $M \equiv \pi_k s$) are straightforward.

The lemma is proven.

\textbf{Theorem 5.6} Let $\mathcal{C}$ be a differential $\lambda$-category. Then $\text{Th}(\mathcal{C})$ is a perturbative $\lambda$-theory.

\textbf{Proof.} We have to check that $\text{Th}(\mathcal{C})$ is closed under the rules from Figure 5.

(\beta) By the definition of $| \cdot |$, we have $|(\lambda x. s)N|_{\Gamma} = \text{ev} \circ \Lambda(|s|_{\Gamma; x; \sigma}), |N|_{\Gamma}$. On the other hand, by Lemma 5.4, we have $|s[N/x]|_{\Gamma} = |s|_{\Gamma; x; \sigma} \circ (\text{id}_{|N|}, |N|_{\Gamma})$, which is equal to $\text{ev} \circ \Lambda(|s|_{\Gamma; x; \sigma}), |N|_{\Gamma}$ by (2).

(\maths{T}) By the definition of $| \cdot |$, we have $|(\mathsf{T}(\lambda x. s)|_{\Gamma} = \mathsf{T} \circ \Lambda(|s|_{\Gamma; x; \sigma}).$ On the other hand, by Lemma 5.5, $|(\lambda x. ts)|_{\Gamma} = \Lambda(|(\mathsf{T} s)|_{\Gamma; x; \sigma}), |s|_{\Gamma} = \Lambda(T(|s|_{\Gamma; x; \sigma}) \circ t)$, which is equal to $\mathsf{T} \circ \Lambda(|s|_{\Gamma; x; \sigma})$ by Proposition 3.11.

(\pi) By the definition of $| \cdot |$, we have $|\pi_k (t s)|_{\Gamma} = \pi_k \circ \iota_t \circ |s|_{\Gamma}$, which is equal to $|s|_{\Gamma}$ if $k = l$ and to 0 otherwise.

The weakening rule (W) follows from Lemma 5.2. Symmetry, reflexivity, and transitivity are obvious from the definition of $\text{Th}(\mathcal{C})$. The remaining rules follow from the definition of the interpretation.

\section{Conclusions and Future Work}

We have presented a novel calculus, the perturbative $\lambda$-calculus, which shares some similarity with the differential $\lambda$-calculus of Ehrhard and Regnier [4], but is based on the differential geometric idea of pushforward rather than that of derivative. We believe that this feature makes the perturbative $\lambda$-calculus well suited for reasoning about forward AD. We have shown that the perturbative $\lambda$-calculus can be modeled by differential $\lambda$-categories. The following issues have been left open in this paper:

- Confluence and strong normalization of the perturbative $\lambda$-calculus. We believe that the proofs from [4] for the differential $\lambda$-calculus can be adapted for the perturbative $\lambda$-calculus.

- We conjecture that the perturbative $\lambda$-calculus does not suffer from the inefficiencies of the differential $\lambda$-calculus. Furthermore, we believe that the perturbative $\lambda$-calculus can provide the computational complexity guarantees of forward AD. Formal proofs of these statements require giving an operational semantics of the perturbative $\lambda$-calculus.
• The practical value of the perturbative $\lambda$-calculus is rather limited. To become a theoretical foundation of a practical programming language, the perturbative $\lambda$-calculus has to be extended with other programming language features (recursion, sum types, polymorphism etc.).

• We have considered only forward AD. The other modes of AD require other novel extensions to the $\lambda$-calculus.

We plan to address these issues in forthcoming papers.

Acknowledgement

I would like to thank Alexey Radul for fruitful discussions and for carefully reading the preliminary versions of this paper. I am grateful to anonymous referees for valuable suggestions that have improved the exposition.

References


273
A Graphical Foundation for Schedules

Guy McCusker\textsuperscript{1}, John Power\textsuperscript{2} and Cai Wingfield\textsuperscript{3,4}

Department of Computer Science
University of Bath
Bath BA2 7AY, United Kingdom

Abstract

In 2007, Harmer, Hyland and Melliès gave a formal mathematical foundation for game semantics using a notion they called a schedule. Their definition was combinatorial in nature, but researchers often draw pictures when describing schedules in practice. Moreover, a proof that the composition of schedules is associative involves cumbersome combinatorial detail, whereas in terms of pictures the proof is straightforward, reflecting the geometry of the plane. Here, we give a geometric formulation of schedule, prove that it is equivalent to Harmer et al.’s definition, and illustrate its value by giving a proof of associativity of composition.

Keywords: Game semantics, geometry, schedules, composites, associativity.

1 Introduction

Over recent decades, game semantics has become one of the standard forms of semantics for programming languages \cite{2,3,4,12,16}. In 2007, Harmer, Hyland and Melliès gave a formal mathematical foundation for game semantics \cite{9}. Their central construct was that of \textit{scheduling function} (or \textit{schedule}), a combinatorial device which describes an interleaving of plays; a position in the game $A \rightarrow B$ is given by a position in $A$, a position in $B$ and a schedule encoding a merge of those positions. They then defined a composite of schedules.

Formally, schedules are defined to be functions $e : \{1, \ldots, n\} \rightarrow \{0, 1\}$ with the conditions that $e(1) = 1$ and $e(2k + 1) = e(2k)$. Thus, a schedule $e$ is essentially a binary string of length $n$, where the domain of $e$ indexes the string left-to-right and where 1s and 0s come in pairs after the first 1 (see Section \ref{section:schedules}). For examples, 1001111001 and 1001100001 are schedules $\{1, \ldots, 10\} \rightarrow \{0, 1\}$.

\textsuperscript{1} Email: g.a.mccusker@bath.ac.uk
\textsuperscript{2} Email: a.j.power@bath.ac.uk
\textsuperscript{3} Email: c.a.j.wingfield@bath.ac.uk
\textsuperscript{4} Supported by a PhD grant from the EPSRC.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
Researchers typically describe schedules on the page or blackboard using a graphical representation [5,7,11,12]. For examples, Figures 1(I) and 1(II) are graphical representations of the above two schedules. Composites are also typically described graphically, in a manner implied by the description of schedules as pairs of order relations in [9]. Figures 1(III) and 1(IV) describe the composite of the two examples of schedules above. While many people draw precisely such diagrams as these, there is another common practice which is to omit the lines — i.e. a picture of a play in $A \rightarrow B$ will be drawn below a heading “$A \rightarrow B$” and have moves in $A$ written below the “$A$”, moves in $B$ written below the “$B$”, the sequential interleaving given by vertical position, but no actual lines drawn. Such pictorially laid-out plays are still really schedule diagrams and the graphical definition of schedules in this paper encompasses them; arguments involving their composition are essentially the same as those here. In a sense, it is the fact that lines could be drawn that means such pictures represent schedules.

This situation gives rise to several natural questions which we explore in this paper. First, in Section 3, we characterise those pictures that arise from schedules.
In Section 4, we formally prove that Harmer et al.’s combinatorial definition and our geometric definition agree. We also define a composite of schedules in geometric terms and show that it agrees with Harmer et al.’s. In Section 5 we prove associativity of composition, give left and right identities, and thereby exhibit the category $\text{Sched}$ of schedules.

Harmer et al. also assert that composition of schedules yields a category, but they do not include a proof in [9]. A proof in Harmer et al.’s terms is combinatorially cumbersome, whereas in geometric terms it follows directly from the associativity of juxtaposition in the plane.

Our graphical definition of schedule is set in the framework of Joyal and Street’s string diagrams for monoidal categories [14]. One might be tempted to use an algebraic definition of graph; however in order to properly consider the diagrams drawn by researchers, one would need to consider precisely what is meant by an embedding of an algebraic graph in the plane, entailing a similar discussion to that in Sections 3 and 4, and in [14]. It is also possible to characterise schedules using the free adjunction $\text{Adj}$ [19], cf. Melliès’ 2-categorical string diagrams for adjunctions [18]. We plan to extend this graphical approach to encompass pointer diagrams [5,8,12], and in this way reformulate all of Harmer et al.’s paper in geometric terms.

2 Combinatorial foundation for schedules

In this section we recall the combinatorial definition of schedules and of composition of schedules from [9].

**Definition 2.1** (as in Harmer et al. [9]) A $\rightarrow$-scheduling function is a function $e : \{1, \ldots, n\} \to \{0, 1\}$ satisfying $e(1) = 1$ and $e(2k + 1) = e(2k)$.

Schedules $e : \{1, \ldots, n\} \to \{0, 1\}$ are sequences of 0s and 1s. We write $|e|$ for the length $n$ of $e$. We also write $|e|_0$ for the number of 0s and $|e|_1$ for the number of 1s in the sequence; so $|e| = n = |e|_0 + |e|_1$.

$\rightarrow$-scheduling functions are also called schedules in [9], but we will take care not to confuse this with Definition 3.4 of schedule, to follow. When necessary for disambiguation, we will call the latter a “graphical schedule”. In Theorem 5.6 we will show that the definitions are equivalent.

**Example 2.2** The following are scheduling functions:

(i) $100111001$ and $1001100001$ are the examples we have already seen illustrated in Figures 1(I) and 1(II).

(ii) Any nonempty prefix (or restriction [9]) of a schedule is a schedule.

**Definition 2.3** (as in Harmer et al. [9]) We will use the notation $e : p \to q$ when $e$ is a schedule $e : \{1, \ldots, p + q\} \to \{0, 1\}$ with $|e|_0 = p$ and $|e|_1 = q$.

Let $e : p \to q$ be a such a schedule. Writing $[n]$ to denote the set $\{1, \ldots, n\}$, we will also write $[n]^+$ for the set of even elements of $[n]$ and $[n]^-$ for the set of odd elements of $[n]$. The schedule $e$ corresponds to a pair of order-preserving, collectively surjective embeddings $e_L : [p] \leftrightarrow [p + q]$ and $e_R : [q] \leftrightarrow [p + q]$, where $e_L$ is the
order-preserving surjection to $e^{-1}(0) \subset [p + q]$ and $e_R$ is likewise a surjection to $e^{-1}(1)$. These in turn correspond to order relations $e_L(x) < e_R(y)$ from $[p]^+\to [q]^+$, and $e_R(y) < e_L(x)$ from $[q]^-$ to $[p]^-$.

We may compose $e : p \to q$ with a schedule $f : q \to r$, to get a schedule $f.e : p \to r$, by taking the corresponding order relations, composing them as relations and then reconstructing the $\rightarrow$-scheduling function on $[p + r]$.

For instance, observe that the two schedules $e = 1001111001$ and $f = 1001100001$ from example 2.2(i) may be composed since $|e|_1 = |f|_0$. Their composite is $f.e = 10011001$. The graphical representation of this composition can be seen in Figures 1(III) and 1(IV).

**Definition 2.4** [9] A schedule $e : p \to p$ such that $e(2k + 1) \neq e(2k + 2)$ is called a copycat function.

A copycat function is of the form $10011001100\ldots$, and in this sense it is the “most alternating” schedule of its length. Any nonempty prefix of a copycat function is also a copycat function.

**Theorem 2.5** [9] Positive natural numbers and schedules $e : p \to q$ form a category, $\Upsilon$, with composition as in Definition 2.3, and with copycat scheduling functions as identities.

A proof of Theorem 2.5 does not appear explicitly in [9], though for associativity of composition, reference is made to the merges of sketches from [13]. The theorem is certainly true, but a proof of associativity seems combinatorially cumbersome.

### 3 Graphical foundation for schedules

There are several possible ways to formalise the schedule diagrams we have drawn. The framework we choose to work in is inspired by that of Joyal and Street’s treatment of the graphical calculus for monoidal categories [14]. We have chosen this framework as it resembles the pictures in the literature. It is possible to give an equivalent definition to this in terms of the algebraic definition of a graph, i.e., in terms of sets $V,E$ and functions $\text{dom, cod} : E \to V$. But by definition, to give an embedding of an edge, together with its domain and codomain, in the plane is exactly to give an injective continuous function from the unit interval $[0,1]$ into the plane. When the graph has more than one edge, one needs to add conditions that ensure that the images of the edges are disjoint except at endpoints. Ultimately, we see no way to express this in simpler terms than those given by Joyal and Street, even when we restrict ourselves to graphs generated by paths.

Proofs in this framework must work on the geometry of the plane graphs themselves [6]. The *compactness* in the definitions ensures that all diagrams and deformations may be finitely decomposed into elementary fragments, and larger constructions may be described in terms of these fragments and their arrangement.

**Definition 3.1** [14] A progressive graph, $\Gamma = (G, G_0)$, is given by:
• $G$, a Hausdorff space.
• $G_0 \subset G$, a finite subset such that $G \setminus G_0$ is a finite collection of edges $e_i$, each homeomorphic to the open interval $(0,1)$. $G_0$ is the set of (inner) nodes. We equip each edge with a direction and disallow directed cycles.

From a progressive graph $\Gamma = (G, G_0)$ we may form $\hat{\Gamma}$, the endpoint compactification of $\Gamma$. $\hat{\Gamma}$ is the compactification of $G$ achieved by affixing distinct endpoints to each edge which has fewer than two endpoints in $G_0$.

**Definition 3.2** [14] For a progressive graph $\Gamma = (G, G_0)$, a progressive embedding of $\Gamma$ in the plane is given by a continuous injection $\iota : \hat{\Gamma} \hookrightarrow \mathbb{R}^2$ such that:

• $\iota$ respects direction on edges: the “source” of an edge is “higher” than its “target” (with these words given a naive interpretation).
• The second projection $\pi_2 : \mathbb{R}^2 \to \mathbb{R}$ is injective on each edge.

We will call a progressive graph together with such an embedding a progressive plane graph.

**Example 3.3** Consider Figure 1(I). One way to characterise this as a progressive plane graph would be with $G = [1, 10] \subset \mathbb{R}$, $G_0 = \{1, 2, \ldots, 10\}$, and $\iota$ the obvious embedding on the page with $\iota(1)$ the node in the top right and $\iota(10)$ the node in the bottom right. (The direction on edges is not explicitly shown in this figure but may be recovered since sources are higher than targets.) Similarly, Figures 1(II), 1(III) and 1(IV) are progressive plane graphs.

In this paper, our primary interest is in the progressive plane graphs that are given by directed paths, since schedule diagrams are paths (see Figures 1(I), 1(II) and 1(IV)). We will rely on a number of elementary observations about paths. First, since our paths are directed, there is an implicit path order on both the nodes and the edges, which we shall denote on nodes by indices on the set of nodes $\{p_1, \ldots, p_n\}$, and similarly by indices on edges: $e_i : p_i \to p_{i+1}$ is the edge with source $p_i$ and target $p_{i+1}$.

Broadly speaking, composition of schedule diagrams involves the extraction of a path from a more complicated graph. One observation we will use in its definition is that graphs which are paths remain paths when we remove nodes (and glue adjoining edges) or add nodes (and split adjoining edges). In each case the relabelling of nodes by order is required (see Figure 2).

![Fig. 2. A node is removed, nodes and edges are relabelled; a node is added, nodes and edges are relabelled.](image)

**Definition 3.4** A schedule, $S_{m,n} = (U, V, \Sigma, \iota)$ consists of the following data:

• Positive natural numbers $m$ and $n$, identified with chosen totally ordered sets $U = \{u_1, \ldots, u_m\}$ and $V = \{v_1, \ldots, v_n\}$. (If we wish to emphasise size, we
may write these as $U_m$ and $V_n$, though these sizes can be recovered from the subscripts on $S_{m,n}$.

- A graph $\Sigma = (S, U + V)$ such that $S$ is a path and the implicit path-ordering of nodes $U + V = \{p_1, \ldots, p_{m+n}\}$ respects the ordering of both $U$ and $V$, and such that the following two conditions hold:

\[
p_1 = v_1 \tag{S1}
\]

for each $k$, either $\{p_{2k}, p_{2k+1}\} \subseteq U$ or $\{p_{2k}, p_{2k+1}\} \subseteq V \tag{S2}$

- Real numbers $u < v$ and chosen progressive embedding $\iota$ of $\Sigma$ in the vertical strip of plane $\{u, v\} \times \mathbb{R}$ such that, (using notation $L_x := \{x\} \times \mathbb{R}$)
  - $U$ embeds in the left-hand edge: $\iota(U) \subseteq L_u$
  - $V$ embeds in the right-hand edge: $\iota(V) \subseteq L_v$
  - Downwards ordering: $j < k \implies \pi_2(\iota(p_j)) > \pi_2(\iota(p_k))$
  - Only nodes touch edges: $\iota(\Sigma) \cap (\{u, v\} \times \mathbb{R}) = \iota(U + V)$. Note that this condition implies that $\Sigma \setminus \Sigma_0$ is strictly contained within $(u, v) \times \mathbb{R}$.

We may write $S_{m,n} : U \to V$ when a schedule $S_{m,n}$ has sets of inner nodes $U$ and $V$. Since the direction on the edges can always be recovered, we may safely omit the arrowheads when drawing schedules by hand.

For examples, Figure 1(I) shows a schedule $4 \to 6$, Figure 1(II) shows a schedule $6 \to 4$ and Figure 1(IV) shows a schedule $4 \to 4$.

We next need a notion of two schedules being “the same”. Joyal and Street’s framework provides a notion of deformation [14] which we will slightly adapt here:

**Definition 3.5** Let $S_{m,n} = (U, V, \Sigma, \iota)$ and $S'_{m,n} = (U', V', \Sigma', \iota')$ be schedules with embeddings $\iota : \hat{\Sigma} \hookrightarrow [u, v] \times \mathbb{R}$ and $\iota' : \hat{\Sigma}' \hookrightarrow [u', v'] \times \mathbb{R}$ respectively.

We say that $\Sigma$ is deformable into $\Sigma'$ if there is a continuous function $h : \hat{\Sigma} \times [0, 1] \to \mathbb{R}^2$ such that

- For each $t \in [0, 1]$, $h(-, t)$ is an embedding $\hat{\Sigma} \hookrightarrow [u_t, v_t] \times \mathbb{R}$ of $\Sigma$ as a schedule in the plane such that $h(U, t) \subseteq L_{u_t}$ and $h(V, t) \subseteq L_{v_t}$.
- $h(\hat{\Sigma}, 0) = \iota(\Sigma)$ is an embedding of $\Sigma$ as a schedule with $h(U, 0) \subseteq L_u$ and $h(V, 0) \subseteq L_v$ and $u_0 = u$ and $v_0 = v$.
- $h(\hat{\Sigma}, 1) = \iota'(\hat{\Sigma}')$ is an embedding of $\Sigma$ (also of $\Sigma'$) as a schedule such that $h(U, 1) \subseteq L_{u'}$ and $h(V, 1) \subseteq L_{v'}$ and $u_1 = u'$ and $v_1 = v'$.

Then we may also say that the schedule $S_{m,n}$ is a deformation of the schedule $S'_{m,n}$. We call $h$ the deformation and write “$S_{m,n} \sim S'_{m,n}$.”

Since the deformation implies that the sets of nodes and edges in each schedule are in bijection, we may automatically associate them to give a notion of node and edge for a deformation class.

For example, looking again at the schedule in Figure 1(IV), we may deform this by smoothly manipulating it in the plane, ensuring that the vertical order of nodes is not disturbed, and such that at each point in time it remains a schedule. Figure
Fig. 3. A “time-lapse” view of a deformation of the schedule in Figure 1(II). Arrowheads used to indicate directions have been omitted for clarity.

3 shows an example of this. One might use a deformation specifically like this in the “cleaning up” of composite schedules before reuse.

Since a plane graph Γ with its plane embedding ι is trivially deformable into the graph-in-the-plane ι(Γ) with the identity embedding, we often identify a graph with a chosen (or arbitrary) embedding where the distinction is unnecessary. Similarly, we will often take a deformation class representative to be a graph chosen as a subset of the plane with the identity embedding.

Example 3.6 For any schedule, the following are examples of deformations which we will use a number of times in this paper:

- A translation of that schedule in the plane.
- A horizontal or vertical scaling in the plane.
- A “piecewise” vertical scaling, achieved by dividing the plane by a finite number of horizontal lines and then applying a different scaling factor to each, as illustrated in Figure 4. This will allow us to place the nodes of a schedule wherever required without altering their order or left–right arrangement.

4 Composition of schedules

In order to examine a category of schedules in analogue to Υ, we need a concrete description of composition of schedules. Composition of two schedules will be performed by constructing a larger progressive graph in the plane from the two components and then extracting a path from it. Essentially, the strips in which each schedule is embedded will be positioned in the plane to meet at a single vertical line. We will begin to trace a path in the right-hand component schedule, but switch to
the other schedule whenever we meet it, and continue to swap back and forth whenever possible. In fact, this will give us the unique up-to-deformation path through all the nodes of both schedules, and such a path will itself be a schedule.

**Definition 4.1** Let \( S_{m,n} = (U, V, \Sigma, \iota) \) and \( S'_{n,r} = (V, W, \Sigma', \iota') \) be two schedules (which we will refer to as \( S \) and \( S' \) for brevity). We first observe that a pair of translations and of piecewise vertical scalings allow us to assume that \( \iota(V) = \iota'(V) \). We call a progressive plane graph formed in this way a (2-fold) composition diagram; it has nodes \( U + V + W \) and an edge for each edge in \( S \) and in \( S' \). (We will now not differentiate between vertex sets \( U, V, W \) and their chosen embeddings \( \iota(U), \iota(V) = \iota'(V), \iota'(W) \) where the context makes it clear what “\( \in \)” means.) Let us call any nodes not on the outside edges of a composition diagram internal and all other nodes external.

To form the composite of \( S \) and \( S' \), written \( S \parallel S' \), we will extract a path from the composition diagram. Eventually our composite will have only nodes \( U + W \). For now we consider nodes in \( V \) as well so that all edges have endpoints. Since \( S \) and \( S' \) are schedules and all edges are progressive, \( U + V + W \) may be unambiguously ordered top-to-bottom in the composition diagram, with order-adjacent nodes connected by at least one edge. Starting from the first edge in \( S' \) we trace a path comprised of edges in \( S \) and \( S' \). Upon reaching each external node, we take the unique outward edge from it. Upon reaching each internal node, we take the outward edge from it that lies in the other schedule from the inward edge we took. We stop when we reach a node with no outward edges. To complete the composite, we discard any edges we did not select and declassify all internal nodes, gluing together adjoining edges (as in Figure 2).

This gives us \( S \parallel S' \) as the data \( (U, W, P, \kappa) \) for a schedule, where \( P \) is the path formed of edges in this way and \( \kappa \) is the inclusion map of this path in the plane.

**Lemma 4.2** The path chosen in Definition 4.1 is the unique path (up to deformation) through all nodes of the composition diagram.

**Proof.** Let schedules \( S \) and \( S' \) be as in Definition 4.1. Consider the composition diagram. Since each component schedule is itself a path, the only nodes where we may have a choice of outward edges are the internal nodes — those shared
between $S$ and $S'$. At an internal node $x$ with more than one outward edge in the composition diagram, there are two possible cases of "local picture", examples of which are shown in Figures 6(I) and 6(II).

(I) As in Figure 6(I). Both outward edges from $x$ lead directly to another internal node. In this case, selecting either edge will yield the same result up to deformation.

(II) As in Figure 6(II). One edge leads directly to another internal node $x'$, and the other directly to an external node, $y$. Suppose $y$ is in $S$. We must take the edge to the external node $y$ (the "cross-schedule" edge). To see why this is necessary, suppose we take the edge to the next internal node, $x'$. Since $x'$ is an internal node, it is a node of $S$, and since $S$ is itself a path through all its nodes, it will eventually reach $x'$ from $x$. However, since the next node after $x$ in $S$ is $y$, $y$ is before $x'$ in the path order of $S$, and so $y$ is above $x'$.
Therefore, since all edges are progressive, if we take the edge directly to \( x' \) we will end up below \( y \) and so can never reach it. A similar argument applies if \( y \) is in \( S' \).

![Diagram showing local pictures around internal nodes in composition diagrams](image)

This gives us a unique path in the composition diagram through \( U + V + W \).

Based on Lemma 4.2, we could have defined the composite simply as the unique (up to deformation) path through every node in the composition diagram. In case (I), where we have two edges from an internal node to another internal node, the proof of the Lemma allows us to select either. However, if we decide always to select the outgoing edge on the opposite side to the incoming edge (so that we pass “through” the node), we have the property that we approach internal nodes from directions alternating right and left. This also constructs our composites in such a way that they resemble the string diagrams for adjunctions in [18].

For a full example of composition, we may compose our original example schedule from Figure 1(II) on the left with a schedule \( 4 \rightarrow 6 \). This is illustrated again by examples in Figures 5(I), 5(II) and 5(III).

**Proposition 4.3** Given schedules \( S : U \rightarrow V \) and \( S' : V \rightarrow W \), their composite \( S \parallel S' \) is a schedule \( U \rightarrow W \).

**Proof.** The data of a schedule and conditions on the embedding follow easily from Definition 4.1, as does (S1).

(S2) simply says that once the path of a schedule diagram reaches one of the two sides, it remains for an even number of nodes before swapping. During composition, all that happens is that some internal nodes are removed, which may result in consecutive sequences of nodes on the same side being concatenated. At the start of the path, in \( W \), this can only be a concatenation of an odd number with an even number, resulting in an odd number as required. Once the path reaches \( U \), concatenations will be of an even number with an even number, as required. The result therefore follows by induction on the length of the schedule.

**Remark 4.4** In the proof of Lemma 4.2, one might wonder why we can never have two outward edges from the same internal node, both to external nodes; or two inward edges from external nodes, both to the same internal node. Such hypothetical fragments of composition diagrams are shown in Figures 7(II) and 7(III), though in fact they can never occur.

While the reason for this may be derived from (S1) and (S2) by induction, there is also a “local” proof inspired by the colouring (or \( O/P \)-labelling) of nodes found...
in the literature [5,12]. Observe that an arbitrary schedule $S_{m,n} : U \rightarrow V$ with path order $U + V = \{p_1, \ldots, p_{m+n}\}$ may be coloured as follows:

- $v_1$ coloured white (drawn as ◦) and $u_1$ black (drawn as •).
- Nodes alternate white and black along the path order.
- Nodes in $U$ alternate black–white taken top-to-bottom, as do those in $V$.

In fact, it is the case that any progressive path with nodes on either side of a vertical strip of $\mathbb{R}^2$ which is coloured in this way is a schedule. The colouring scheme encodes the dynamics of a schedule, as an alternative to $(S1)$ and $(S2)$, locally and in terms of colours on the nodes rather than by the explicit odd–evenness of distance from the first node. By colouring, we attach to each node its parity in its schedule.

Figure 7(I) shows our original schedule from Figure 5(II) decorated in this way. Observe that $(S2)$ is satisfied if and only if this colour scheme is followed.

Note that edges are always directed ◦ → • if they move from one side to the other (this is the switching condition for $\rightarrow [1]$). Thus, if some $p_i$ is black and $p_i+1$ is white, then $\{p_i, p_{i+1}\} \subset U$ or $\subset V$. When composing schedules, the colours in the two copies of the internal nodes will be precisely reversed in each schedule. We can show this using ◦ and ● for the internal nodes of the composition diagram, such as the one in Figure 7(VI). Were we to have two cross-schedule edges from the same internal node, it is not the case that both of them could be ◦ → •, since the internal node is different colours in both component schedules; hence such a scenario is impossible. Similarly for two cross-schedule edges to the same internal node. Figures 7(IV) and 7(V) show the hypothetical fragments with a choice of colours, and the illegal edges marked with a ×. An analogous arguments using state diagrams exist elsewhere in the game semantics literature; for example, [1,7].
The category $Sched$

We now come to the key result, that of the associativity of composition. This, along with a definition of identities, will yield a description of the category of schedules.

**Proposition 5.1** Composition of schedules is associative.

**Proof.** It suffices to show that both possible three-fold compositions are equal. Suppose we are composing schedules $U \xrightarrow{S_{l,m}} V \xrightarrow{S'_{m,n}} W \xrightarrow{S''_{n,r}} X$ (which we will refer to as $S$, $S'$ and $S''$ for readability). We wish to show that $(S \parallel S') \parallel S''$ is deformable into $S \parallel (S' \parallel S'')$. Without loss of generality, we may position $S$, $S'$ and $S''$ so that the two copies of $V$ are identified and the two copies of $W$ are identified. This is the 3-fold composition diagram, an example of which can be seen in Figure 8. By Lemma 4.2, both composites $(S \parallel S') \parallel S''$ and $S \parallel (S' \parallel S'')$ are given by the unique path (up to deformation) in the 3-fold composition diagram which passes through each node $U + V + W + X$. Thus the difference in bracketing between $(S \parallel S') \parallel S''$ and $S \parallel (S' \parallel S'')$ corresponds to whether we remove unselected edges and inner nodes from $V$ or from $W$ first; both choices must yield the same path. In essence, associativity is due to the natural associativity of juxtaposition in the plane. 

We now proceed to examine the category of schedules. The objects of this category are natural numbers $m \in \mathbb{N}^+$, realised as finite indexed sets $U = \{u_1, \ldots, u_m\}$. A morphism $m \to n$ is a deformation-class of schedules $S_{m,n} : U \to V$.

**Definition 5.2** Copycat schedules are the “most alternating” schedules possible subject to the schedule axioms. For $n \in \mathbb{N}^+$, the schedule $I_{n,n}$ may be given by its path description on vertex set $P_{2n} = U_n' + U_n$.

$$p_{4k+1} = u_{2k+1}, \quad p_{4k+2} = u_{2k+1}', \quad p_{4k+3} = u_{2k+2}, \quad p_{4k+4} = u_{2k+2}'$$

Graphically, this can be seen in Figure 9. Alternatively, these copycat schedules may be characterised by saying that also $\{p_{2k+1}, p_{2k+2}\} \not\subset U_n$ and $\not\subset U_n'$. The following lemma is proved in Appendix A as Lemma A.1.

**Lemma 5.3** Copycat schedules $I_{n,n}$ are the identities of schedule composition.

**Theorem 5.4** Positive natural numbers, together with the graphical schedules form a category, called $Sched$, where composition is defined by Definition 4.1 and identities are copycat schedules.

We will demonstrate that $Sched$ is equivalent to $\Upsilon$ by exhibiting a functor $Sched \to \Upsilon$ giving the equivalence. Let $S_{m,n} : U \to V$ be a schedule in $[u, v] \times \mathbb{R}$; that is, an arrow of $Sched$. We construct a functor $C$ which acts on objects as the
Fig. 8. A three-way composition diagram with composite path highlighted. Note that, since we must always cross between schedules on reaching an internal node, there are no choices to be made.

\[ u_1 = p_1 \]
\[ p_2 = u_1' \]
\[ p_3 = u_2 \]
\[ u_2 = p_4 \]
\[ u_3 = p_5 \]
\[ p_6 = u_3' \]
\[ p_7 = u_4' \]
\[ u_4 = p_8 \]

Fig. 9. A prefix fragment of an identity schedule.

identity and which assigns to \( S_{m,n} \) a \( \rightarrow \)-schedule function \( e : [m + n] \rightarrow \{0, 1\} \) with

\[ e : i \mapsto \begin{cases} 
0 & \text{if } p_i \in L_u \\
1 & \text{if } p_i \in L_v
\end{cases} \]

In the combinatorial terms of Harmer et al. [9], a schedule \( e : m \rightarrow n \) corresponds
to injections $e_L : [m] \hookrightarrow [m + n]$ and $e_R : [n] \hookrightarrow [m + n]$, which in turn correspond to order relations $e_L(x) < e_R(y)$ from $[m]^+$ to $[n]^+$ and $e_R(y) < e_L(x)$ from $[n]^-$ to $[m]^-$.

Thinking in terms of diagrams, the decorations $+$ and $-$ correspond to the parity down each edge. Then the order relation $e_R(y) < e_L(x)$ is depicted by edges right-to-left in the diagram and the order relation from $e_L(x) < e_R(y)$ is depicted by edges left-to-right. The parity is indicated by the colours on nodes (though they are reversed on the left side). Composition of the order relation from two schedules is exactly what is performed during the composition on diagrams. Hence, we have the following proposition:

**Proposition 5.5** $C$ is a functor $\text{Sched} \rightarrow \Upsilon$.

**Theorem 5.6** $C : \text{Sched} \rightarrow \Upsilon$ is an equivalence of categories.

**Proof.** We exhibit an identity-on-objects functor $G : \Upsilon \rightarrow \text{Sched}$. $G$ assigns to a $\rightarrow\leftarrow$-scheduling function $e : [m + n] \rightarrow \{0, 1\}$ with $|e|_0 = m$ and $|e|_1 = n$, a schedule $S_{m,n} : U_m \rightarrow V_n$ in the following manner:

Nodes $p_1, \ldots, p_{m+n}$ are arranged in the vertical strip $[0, 1] \times \mathbb{R}$ with coordinates $p_i = (e(i), -i)$. Order-adjacent nodes $p_i, p_{i+1}$ are joined by a straight line if their first ordinates disagree (i.e., if $\pi_1 p_i \neq \pi_1 p_{i+1}$) and with a circular arc (of angle less than $\pi$) if their first ordinates agree (i.e., if $\pi_1 p_i = \pi_1 p_{i+1}$). This manner is similar to the explicit construction of identity schedules in Appendix A.

$CG = \text{id}$ by construction. To see that $GC \cong \text{id}$, we need to show that schedule is determined up-to-deformation by the vertical order and left–right arrangement of nodes. By an appropriate piecewise vertical scale, translation and horizontal scale, we may assume that nodes are arranged according to their path-order at integer heights (as would be the case in the image of $GC$). So, by looking at the simply connected rectangles $[0, 1] \times [i, i+1]$, we see that endpoint-preserving homotopies allow edges within these rectangles to be deformed into each other. \hfill $\Box$

6 Future work

Following these results it seems natural to generalise this approach to account for other ideas in [9]; to strategies, and to $\otimes$-scheduling functions. Capturing these similar notions in the diagrams used to represent them will present no challenge. It also seems appropriate to investigate the relationship with the the 2-categorical string diagrams for dialogue games in [18]. Further, we will examine the pointer functions and heaps — also ubiquitously diagrammatically represented — as their representation also seems well captured by their diagrams [9,10]. In future work we plan to give an account of all of Harmer et al.’s paper in these geometric terms.

Our arguments for key properties have been rendered far simpler through careful definition of the graphical objects under consideration. It may also be the case that these kinds of argument make it possible to define an associative composition for more relaxed notions of scheduling. Some refined notion of type, more sophisticated than just a number, may support a broader class of schedule.

287
As well as Joyal and Street’s framework providing a “realistic” foundation for schedule diagrams, it is also extensible into other classes of planar diagrams [15,17,20,21] and we hope choosing it will provide common ground for future work, perhaps contributing new categories of games and strategies.

References


A Appendix: Identity schedules

$I_{n,m}$ may be explicitly defined, for example as $I_{n,m} = (U_n', U_n, \Sigma_{n,m}, \text{id} : \Sigma \to [0,1] \times \mathbb{R})$ where:

- $U_n' = \{ u_i' \mid i \text{ odd } \Rightarrow u_i' = (0, -2i), \text{ i even } \Rightarrow u_i' = (0, 1 - 2i) \}$
- $U_n = \{ u_i \mid i \text{ odd } \Rightarrow u_i = (1, -2i), \text{ i even } \Rightarrow u_i = (1, 1 - 2i) \}$
- $\Sigma = (\Sigma, U_n' + U_n)$ where

$$\Sigma = \left[ \text{line segments } [u_{2k-1}, u_{2k-1}'] \right] \cup \left[ \text{line segments } [u_{2k}, u_{2k}]' \right] \cup \cdots \cup \left[ \text{circular arc, endpoints } \{u_{2k-1}', u_{2k}'\}, \text{ angle } \alpha < \pi \right] \cup \cdots \cup \left[ \text{circular arc, endpoints } \{u_{2k}, u_{2k+1}\}, \text{ angle } \alpha < \pi \right]$$

Of course, we consider any deformation of this to be an identity schedule.

(I) An edge of the schedule.

(II) A fragment of the identity schedule.

(III) A fragment of the deformation demonstrating that copycat schedules are identities of schedule composition.

Fig. A.1.

**Lemma A.1** Left and right composition with $I_{n,m}$ satisfies identity axioms.

**Proof.** First, for composition on the left, let $S_{m,n} : U \to V$ be a schedule and let $I_{m,m} : U' \to U$ be the identity schedule. We want to show that $I_{m,m} \parallel S_{m,n} \cong S_{m,n}$.

Since $S_{m,n}$ is a schedule, we have that $u_{2k+1}$ and $u_{2k+2}$ are joined by an edge, as in Figure A.1(I). Since $I_{m,m}$ is a copycat, we know that $u_{2k+1}$ and $u_{2k+2}$ are joined in $I_{m,m}$ by the identity schedule fragment in Figure A.1(II).

We know that in $I_{m,m} \parallel S_{m,n}$, the edge into $u_{2k+1}$ to be chosen will be the one from $S_{m,n}$, and the edge out of $u_{2k+1}$ will be the one from $I_{m,m}$, after which $u_{2k+1}$ will be “declassified” as a node. Similarly, the edge into $u_{2k+2}$ to be chosen will be the one from $I_{m,m}$ and the edge out will be the one from $S_{m,n}$ before $u_{2k+2}$ is declassified. Then the equality of the schedule fragment surrounding $u_{2k+1}$ and $u_{2k+2}$ (which will become the schedule fragment surrounding $u_{2k+1}'$ and $u_{2k+2}'$ in the composite) holds up to the evident deformation in Figure A.1(III).

Between points $u_{2k}$ and $u_{2k+1}$ all activity in the composite will be the activity in $S_{m,n}$, as $u_{2k}$ is approached from the right in the construction of $I_{m,m} \parallel S_{m,n}$ and so $u_{2k+1}$ must be approached from the left.

A similar argument also demonstrates that for schedule $S_{m,n} : U \to V$ and copycat schedule $I_{n,n} : V \to V'$, we have $S_{m,n} \cong S_{m,n} \parallel I_{n,n}$. $\square$

289
Network Conscious $\pi$-calculus:
A Concurrent Semantics

Ugo Montanari$^{1,2}$ Matteo Sammartino$^{1,3}$

Computer Science Department
University of Pisa
Pisa, Italy

Abstract

Traditional process calculi usually abstract away from network details, modeling only communication over shared channels. They, however, seem inadequate to describe new network architectures, such as Software Defined Networks [1], where programs are allowed to manipulate the infrastructure. In this paper we present a network conscious, proper extension of the $\pi$-calculus: we add connector names and the primitives to handle them, and we provide a concurrent semantics. The extension to connector names is natural and seamless, since they are handled in full analogy with ordinary names. Our observations are multisets of routing paths through which sent and received data are transported. However, restricted connector names do not appear in the observations, which thus can possibly be as abstract as in the $\pi$-calculus. Finally, we show that bisimilarity is a congruence, and this property holds also for the concurrent version of the $\pi$-calculus.

Keywords: $\pi$-calculus, network-awareness, concurrent semantics

1 Introduction

The trend in networking is going towards more “open” architectures, where the infrastructure can be manipulated in software. This trend started in the nineties, when OpenSig [3] and Active Networks [24] were presented, but neither gained wide acceptance due to security and performance problems. More recently, OpenFlow [16,1] or, more broadly, Software Defined Networking has become the leading approach, supported by Google, Facebook, Microsoft and others. Software defined networks (SDNs) are networks in which a programmable controller machine manages a group of switches, by instructing them to install or uninstall forwarding rules and report traffic statistics.

$^1$ Research supported by the EU Integrated Project 257414 ASCENS
$^2$ e-mail: ugo@di.unipi.it
$^3$ e-mail: sammarti@di.unipi.it

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
Traditional process calculi, such as π-calculus \cite{19,20}, CCS \cite{18} and others, seem inadequate to describe these kinds of networks, because they abstract away from network details. In fact, two processes are allowed to communicate only through shared channels and it is not possible to express explicitly the fact that there is some complex connector between them. To give better visibility to the network architecture, in recent years network-aware extensions of known calculi have been devised \cite{10,23}.

This paper focuses on the π-calculus, and aims at equipping it with a natural notion of network: nodes and connectors are computational resources, so it is reasonable to represent them as (structured) names. We call the resulting calculus \textit{Network Conscious π-calculus} (NCPi). We consider networks without hierarchies (e.g. administrative domains), where some parts may be private to a process and the public part is shared, as in CHARM \cite{6}. Networks can be used by many processes at the same time, but we impose some restrictions on how resources can be accessed. The calculus has the following features:

- We distinguish two types of names: \textit{sites}, which are the nodes of the network, and \textit{links}, named connectors between pairs of sites. Sites are just atoms, e.g. \textit{a}, links have the form \textit{l}_{ab}, meaning that there is a link named \textit{l} between \textit{a} and \textit{b}.
- The syntax can express the \textit{creation} of a link through the restriction operator, and the \textit{activation} of a transportation service over a link through a dedicated prefix. Separating these operations agrees with the π-calculus, where creating and using a channel as subject are two distinct operations. Moreover, since processes are not required to communicate on shared channels, an extended output primitive is introduced that specifies not only the emission site but also the destination one.
- We provide a concurrent semantics, where concurrent transmissions can be observed in the form of a multiset of routing paths. The associated behavioral equivalence is a \textit{congruence}.

We choose to have labelled connectors, instead of anonymous ones as in \cite{10} and \cite{23}, for two main reasons. First of all, they are intended to model transportation services with distinct features (cost, bandwidth...), which could be encoded in the label type, as we already do for the connectors’ source and target. In any case, NCPi allows one to recover anonymous connectors through the restriction operator. Second, this enables reusing most of the notions of the π-calculus (renaming, α-conversion, extrusion...), suitably extended.

The main result of this paper is that bisimilarity on our concurrent semantics is a congruence. This is a desirable property for a process calculus, because it allows for the compositional analysis of systems. The authors of \cite{23,10} treat bisimilarity and achieve compositionality as well, but they take a different approach than ours: they start from a reduction semantics, guess a suitable notion of barb, define barbed congruence by closing \textit{w.r.t.} all the contexts, and then characterize it as a bisimulation equivalence on a labelled version of the transition system. In general, this approach yields labelled transition systems with succinct observations, but may resort to non-standard notions of bisimilarity, where the closure under contexts is
“hardwired”. We show that we can gain the congruence property through a concurrent semantics, while keeping the notion of bisimilarity as standard as possible. We emphasize that interleaving semantics is far from being natural in this distributed setting. In fact, it is based on a mutual exclusion mechanism between remote actions which is simpler from a formal point of view, but not realistic for modeling concurrent systems.

Bisimilarity not being a congruence for the $\pi$-calculus depends on the interleaving nature of the semantics, and not on the language itself. In fact, we will show that, if we equip $\pi$-calculus with a concurrent semantics, the congruence property holds. This has already been shown in [14,15], but the semantics presented there allows observing the channel where a synchronization is performed, whereas our concurrent semantics is closer to the $\pi$-calculus, in the sense that we adopt a synchronization mechanism that hides such a channel.

The paper is organized as follows: in §2 we show a motivating example; in §3 we present the syntax of the language; in §4 we present the operational semantics and we show that bisimilarity is a congruence; in §5 we model a simple routing protocol. An extended version of this paper, including also an interleaving semantics, is available [21].

2 Motivating example

We consider the system shown in Fig. 1, made of a network manager $M$, using a reserved site $m$, and two processes $p$ and $q$, which access the network respectively through the sites $a$ and $b$. The manager is the only entity that can create new links and grant access to them. The process $p$ wants to send a message to $q$, but we assume that there are no links between $a$ and $b$ allowing $p$ and $q$ to communicate.

The processes act as follows: $M$ receives two sites at $m$, creates a new link between them and sends this link from $m$ to the first of the received sites. The process $p$ sends $a$ and $b$ from $a$ to $m$, waits for a link at $a$ and then evolves to the parallel composition of two components: the first component activates a transportation
service over the received link, which can be used by the other component; the second component sends $c$ from $a$ to $b$. The process $q$ simply waits for a datum at $b$. Finally, the process $L$, in order to simulate a persistent connection, repeatedly activates a transportation service over its argument: this is necessary, because the link prefix expresses a single activation of the service, as input/output prefixes in the $\pi$-calculus express a single usage of their subject channel.

We have that $p, L(l_{am})$ and $M$ can do the following transitions

$$\pi ma.\pi mb.a(l_{(xy)}).(L(l_{xy}) | \pi bc.p') \xrightarrow{\pi ma} \pi mb.a(l_{(xy)}).(L(l_{xy}) | \pi bc.p')$$

$$L(l_{am}) \xrightarrow{\text{ad}_{am}, m} L(l_{am})$$

$$m(x).m(y).(l_{xy})(\overline{\pi x}l_{xy}M) \xrightarrow{\text{mma} \cdot \cdot} m(y).(l_{ay})(\overline{\pi a}l_{ay}M)$$

where $\cdot \cdot \pi ma$ represents the beginning of transmission as a path of length zero, analogous to the $\pi$-calculus output action: the $\cdot \cdot$ on the left side indicates that the path can only extend rightward, i.e. subsequent hops will be listed after $\cdot \cdot$ from left to right in the form of a sequence of links; the string $\pi ma$ describes the path, telling (from left to right) the site where the datum is available, the destination site and the datum. Symmetrically, $\text{mma} \cdot \cdot$ means that $a$, which has destination $m$, is received at $m$ and then goes through a path of length zero; it is analogous to the $\pi$-calculus input action. In this case the destination and reception site coincide, as in the input prefix ($a(x)$ can be thought of as $aa(x)$), but the reception site will eventually become different as the path grows. The label $a; l_{am}; m$ represents the activation of a transportation service over $l_{am}$.

When these processes are put in parallel in $S$, their paths can be concatenated and a path representing a complete transmission on $l_{am}$ can be observed

$$S \xrightarrow{\cdot \cdot l_{am} \cdot \cdot} \pi mb.a(l_{(xy)}).(L(l_{xy}) | \pi bc.p') | m(y).(l_{ay})(\overline{\pi a}l_{ay}M) | q \mid L(l_{am}) \mid L(l'_{ma}) .$$

As in the $\pi$-calculus, the transmitted datum, namely $a$, is not observable. Then, a sequence of possible transitions after this one is:

$$\ldots \cdot \cdot l_{am} \cdot \cdot a(l_{(xy)}) .(L(l_{xy}) | \pi bc.p') | (l_{ab})(\overline{\pi a}l_{ab}M) | q \mid L(l_{am}) \mid L(l'_{ma})$$

(transmission of $b$)

$$\cdot \cdot l_{ma} \cdot \cdot (l_{ab})(L(l_{ab}) | \pi bc.p' | M) \mid q \mid L(l_{am}) \mid L(l'_{ma})$$

($l_{ab}$ scope extension, $l_{ab} \notin \text{fn}(p')$)

$$\cdot \cdot \cdot \cdot (l_{ab})(L(l_{ab}) | p' | M) \mid q'[c/x] \mid L(l_{am}) \mid L(l'_{ma})$$

(transmission of $c$)

Notice that the last transition hides the link used for transmission, namely $l_{ab}$, because it is restricted. We just observe $\cdot \cdot \cdot \cdot$, analogous to the $\pi$-calculus $\tau$.

The semantics also allows observing in parallel all the pieces of a path. For instance, we can observe $S$ doing $\cdot \cdot \pi ma \cdot a; l_{am}; m \mid \text{mma; \cdot \cdot}$, which represents a three-element multiset. These kinds of observations are exactly those making the behavioral equivalence compositional.
3 Syntax

We assume to have an enumerable set of site names $\mathcal{S}$ (or just sites) and an enumerable family of enumerable, disjoint sets of link names $\{\mathcal{L}_{a,b}\}_{a,b \in \mathcal{S}}$ (or just links). We let $\mathcal{L}_a = \bigcup_{b \in \mathcal{S}} \mathcal{L}_{a,b} \cup \mathcal{L}_{b,a}$ and $\mathcal{L} = \bigcup_{a,b \in \mathcal{S}} \mathcal{L}_{a,b}$, and we denote by $l_{ab}$ the element in $\mathcal{L}$ corresponding to $l \in \mathcal{L}_{a,b}$. Notice that, being $\mathcal{L}_{a,b}$ and $\mathcal{L}_{c,d}$ disjoint, for all $ab \neq cd$, we cannot have two links $l_{ab}$ and $l_{cd}$ unless $a = c$ and $b = d$.

Definition 3.1 NCPi processes are defined as follows, for $a, b \in \mathcal{S}, l_{ab} \in \mathcal{L}$:

$$
p ::= 0 \mid \pi.p \mid p + p \mid p \mid (r)p \mid A(r_1, r_2, \ldots, r_n) \\
r ::= a \mid l_{ab} \mid s ::= a \mid l_{(ab)} \mid \pi ::= \overline{ab}r \mid a(s) \mid l_{ab} \mid \tau \\
A(s_1, s_2, \ldots, s_n) \stackrel{def}{=} p \quad i \neq j \Rightarrow n(s_i) \cap n(s_j) = \emptyset
$$

Here we write $n(s)$ for the names in $s$, including $a$ and $b$ if $s$ is $l_{(ab)}$ (analogously for $n(l_{ab})$). We have the usual inert process, sum and parallel composition. For the recursive definition, we require that formal parameters do not have names in common, because otherwise we might have type dependencies between parameters, e.g. in $A(a, l_{(ab)})$ one of the second parameter’s endpoints depends on the first parameter. Prefixes can have the following forms:

- The output prefix $\overline{ab}r$: $\overline{ab}r.p$ can send the datum $r$ from $a$ addressed to $b$ and continue as $p$. Notice that, unlike $\pi$-calculus, the destination site can be different than the emission one.
- The input prefix $a(s)$: $a(s).p$ can receive at $a$ a datum to be bound to $s$ and continue as $p$. The intended meaning of $c(l_{ab}).p$ is an atomic, polyadic version of $c(a).c(b).c(l_{ab}).p$. Here a monadic link input prefix $c(l_{ab}).p$ is not allowed, since it would introduce a matching capability we prefer not to provide. Consequently, $a$ and $b$ are not free in $c(l_{ab}).p$.
- The $\tau$ prefix: $\tau.p$ can perform an internal action and continue as $p$.
- The link prefix $l_{ab}$: $l_{ab}.p$ can offer to the environment the service of transporting a datum from $a$ to $b$ through $l$ and then continue as $p$.

Finally, we have the restriction ($r$): $r$ is private in $(r)p$, i.e. it cannot be observed as free name in a communication. Notice that $a$ and $b$ are free in $(l_{ab})p$. Sequences of restrictions will be denoted by capital letters ($R$).

We define the set $\text{fn}(p)$ of free names of $p$ as:

$$
\begin{align*}
\text{fn}(0) &= \emptyset \\
\text{fn}(\overline{ab}r.p) &= \{a, b\} \cup n(r) \cup \text{fn}(p) \\
\text{fn}(a(s).p) &= \{b\} \cup (\text{fn}(p) \setminus (\{a\} \cup \mathcal{L}_a)) \\
\text{fn}(l_{ab}.p) &= \text{fn}(p) \setminus (\{a\} \cup \mathcal{L}_a) \\
\text{fn}(l_{(ab)}p) &= \{a, b\} \cup \text{fn}(p) \setminus \{l_{ab}\} \\
\text{fn}(A(r_1, \ldots, r_n)) &= n(r_1) \cup \cdots \cup n(r_n)
\end{align*}
$$

$$
\text{fn}(\tau.p) = \text{fn}(p) \\
\text{fn}(l_{ab}.p) = \{l_{ab}, a, b\} \cup \text{fn}(p) \\
\text{fn}(a(l_{bc}).p) = \{a\} \cup \text{fn}(p) \setminus \{\{b, c\} \cup \mathcal{L}_b \cup \mathcal{L}_c\} \\
\text{fn}(p + q) = \text{fn}(p | q) = \text{fn}(p) \cup \text{fn}(q) \\
\text{fn}(A(r_1, \ldots, r_n)) = n(r_1) \cup \cdots \cup n(r_n)
$$

294
where \( A(s_1, \ldots, s_n) \stackrel{\text{def}}{=} p \) implies \( \text{fn}(p) \subseteq \text{n}(s_1) \cup \cdots \cup \text{n}(s_n) \). Notice the definition of \( \text{fn}((a)p) \): if a link having \( a \) as one of its endpoints appears in \( p \), then it is considered bound. Similarly for \( b(a).p \) and \( a(l_{bc}).p \). This intuitively means that a global link cannot have private endpoints, analogously to what happens for free processes in a well-formed state of a CHARM [6]: their variables must belong to the global part.

The notion of renaming is defined as follows.

**Definition 3.2** A renaming \( \sigma \) is a pair of functions \( \langle \sigma_S : S \to S, \sigma_L : L \to L \rangle \) such that \( \sigma_L(l_{ab}) = l'_{a'b'} \) implies \( \sigma_S(a) = a' \) and \( \sigma_S(b) = b' \). We denote by \( r \sigma \) the result of applying the appropriate components of \( \sigma \) to \( r \).

The condition relating \( \sigma_S \) and \( \sigma_L \) ensures that \( \sigma \) acts as a graph homomorphism, i.e. each link is renamed by \( \sigma_L \) to a link whose endpoints are the image through \( \sigma_S \) of the original link’s endpoints. Some notation: we write \( [r_1/r_1, r_2/r_2, \ldots, r_n/r_n] \) to indicate the function mapping \( r_1 \) to \( r_1' \), \( r_2 \) to \( r_2' \), \( \ldots \), \( r_n \) to \( r_n' \), and we write \( [r'_{a'b'}/l_{ab}] \) as a shorthand for \( [a'/a, b'/b, l'_{a'b'}/l_{ab}] \). Notice that \([a'/a]\) does not uniquely characterize a renaming. In fact, while certainly \( \overline{abc}[a'/a] = \overline{a'b'c} \), \( a \notin \{b, c\} \), for \( l_{ab}[a'/a] \) we only know that it must belong to \( \mathcal{L}_{a'b'} \). Thus we should avoid applying such renaming to a link \( l_{ab} \), since the result would be undefined. A special case (see below) is when \( l_{ab} \) is bound.

Now we introduce well-formed NCi processes. Informally, a process is well-formed if each bound link it contains is bound explicitly, and not as a side-effect of binding a site, and if two links with the same label but different endpoints do not appear free in any of its sub-processes. For instance, \( (a(b).l_{bc}.)p \) and \( (l_{ab})l_{ab}.l_{cd}.p \) are not well-formed: the former because \( l_{bc} \) is implicitly bound by \( a(b) \), the latter because \( l \) labels two links between different sites.

**Definition 3.3** A NCi process \( p \) is well-formed if for every subterm \( q \):

(i) \( q = (a)p' \) implies \( \text{fn}(q) = \text{fn}(p') \setminus \{a\} \);

(ii) \( q = b(a).p' \) implies \( \text{fn}(q) = \{b\} \cup \text{fn}(p') \setminus \{a\} \);

(iii) \( q = c(l_{ab}).p' \) implies \( \text{fn}(q) = \{c\} \cup \text{fn}(p') \setminus \{a, b, l_{ab}\} \);

(iv) \( l_{ab}, l'_{cd} \in \text{fn}(q) \) and \( ab \neq cd \) implies \( l \neq l' \).

A first consequence of this definition is that we do not need to subtract \( \mathcal{L}_a \) or \( \mathcal{L}_b \cup \mathcal{L}_c \) when computing the free names of \( (a(b).p, (a)p \) or \( a(l_{bc}).p \), if these processes are well-formed.

Well-formedness also allows us to say how a generic substitution can act on processes as a proper renaming. This is needed in order to define \( \alpha \)-conversion, which in fact is given in Fig. 2 for well-formed processes only. \( \alpha \)-conversion for a restricted process is simply \( (a)p \equiv (a')p'[a'/a] \), with \( a' \notin \text{fn}((a)p) \), where \([a'/a]\) is never applied to a link \( l_{ab} \), since such link cannot be free in \( p \). If it is bound, i.e. if \( (l_{ab})p' \) is a subprocess of \( p \), then we simply have inductively \( ((l_{ab})p'[a'/a] \equiv (l'_{a'b'})p'[a'/a] \), for any \( l'_{a'b'} \) such that \( l'_{a'b'} \notin \text{fn}(p) \), for all \( a'', b'' \). Notice that, in order to maintain property (iv) of well-formedness, captures must be avoided not only in the presence of \( l'_{a'b'} \in \text{fn}(p) \), but also of links of the form \( l'_{a'b'} \in \text{fn}(p) \), for any \( a'', b'' \). A similar restriction also holds when \( \alpha \)-converting \( a(l_{bc}).p \). We remark that these processes
\[ \alpha \text{-equivalence:} \]
\[
(a)p \equiv (a')p[\alpha'/a] \quad b(a).p \equiv b(a').p[\alpha'/a] \quad a' \not\in \text{fn}((a)p)
\]
\[
(l_{ab})p \equiv (l'_{ab})p[l'_{ab}/l_{ab}] \quad \forall a', b' : l'_{ab}/b' \not\in \text{fn}((l_{ab})p)
\]
\[
a(l_{bc}).p \equiv a(l'_{bc'}).p[l'_{bc'}/l_{bc}].p \quad b', c' \not\in \text{fn}(a(l_{bc}).p) \land
\]
\[
\forall b'', c'' : l'_{bc''}/b'' \not\in \text{fn}(a(l_{bc}).p)
\]

**Unfolding law:** \( A(r_1, \ldots, r_n) \equiv massels[1/s_1, \ldots, r_n/s_n] \) if \( A(s_1, \ldots, s_n) \overset{\text{def}}{=} p \)

---

**Fig. 2. Structural congruence of well-formed processes.**

4 Concurrent semantics

Interleaving semantics can be considered inadequate for distributed system with partially asynchronous behavior, since it implicitly assumes the existence of a central arbiter who grants access to resources. This criticism is particularly relevant for our network-conscious calculus. Here we present a concurrent semantics where we can observe multisets of routing paths covered at the same time. Single paths are denoted by \( \alpha \), multisets of paths by \( \Lambda \) and are called *concurrent paths*.

**Definition 4.1** Paths and concurrent paths are defined as follows, for \( a, b \in S \), \( l_{ab} \in L \):

\[
\alpha ::= a; W; b \mid \bullet; W; \bullet \mid \bullet; W; abr \mid abr; W; \bullet \\
| ab(s); W; \bullet \mid n(s) \cap (n(W) \cup \{a, b\}) = \emptyset \\
r ::= a \mid l_{ab} \quad s ::= a \mid l_{(ab)} \quad W ::= l_{ab} \mid W; W \mid \epsilon \\
\Lambda ::= 1 \mid \alpha \mid \Lambda_1 | \Lambda_2 \mid (r) \Lambda
\]

Their structural congruence \( \equiv_{\Lambda} \) includes monoidality of \( ; \), with \( \epsilon \) as identity, and of \( | \), with \( 1 \) as identity, and the following scope extension axioms:

\[
(r)(r')\Lambda \equiv_{\Lambda} (r')(r)\Lambda \quad r \not\in n(r'), r' \not\in n(r) \\
\Lambda_1 | (r)\Lambda_2 \equiv_{\Lambda} (r)(\Lambda_1 | \Lambda_2) \quad r \not\in \text{fn}(\Lambda_1)
\]

A path \( \alpha \) can be of two general forms. It can be a *service path* \( a; W; b \), representing a transportation service from \( a \) to \( b \) that employs the resources listed in \( W \) and possibly other private, unobservable resources. Alternatively, it can be a sequence starting and/or ending with \( \bullet \), which represents an actual transmission over \( W \). More specifically, in this case \( \alpha \) can be:
Montanari, Sammartino

<table>
<thead>
<tr>
<th>path α</th>
<th>fn</th>
<th>bn</th>
<th>obj</th>
<th>obj_in</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>α;W; b</td>
<td>n(α)</td>
<td>⊙</td>
<td>⊙</td>
<td>⊙</td>
<td>{a,b}</td>
</tr>
<tr>
<td>•;W; •</td>
<td>n(α)</td>
<td>⊙</td>
<td>⊙</td>
<td>⊙</td>
<td>⊙</td>
</tr>
<tr>
<td>•;W;abr</td>
<td>n(α)</td>
<td>⊙</td>
<td>{b,r}</td>
<td>⊙</td>
<td>{a}</td>
</tr>
<tr>
<td>abr;W; •</td>
<td>n(α)</td>
<td>⊙</td>
<td>{b,r}</td>
<td>{r}</td>
<td>{a}</td>
</tr>
<tr>
<td>ab(s);W; •</td>
<td>n(α) \ n(s)</td>
<td>n(s)</td>
<td>{b}</td>
<td>⊙</td>
<td>{a}</td>
</tr>
</tbody>
</table>

Table 1
Free names, bound names, objects, input objects and interaction sites of a path α.

- an output path, if abr is on the right, representing the emission of r, whose destination is b, at a;
- an input path, if abr or ab(s) is on the left. In the former case, it is called free input path and means that r, whose destination is b, is received at a; in the latter case, it is called bound input path and s is a placeholder for the received name;
- a complete path, if • is on both sides, meaning that the transmission has already been completed.

Concurrent paths can have the following forms:
- the empty concurrent path 1 indicates that no activity is performed;
- the singleton concurrent path α is a concurrent path made of a single path;
- the union Λ₁ | Λ₂ means that the paths in Λ₁ and Λ₂ are being traversed at the same time;
- the extrusion restriction (r)Λ indicates that r is being extruded through one or more paths in Λ.

We will use \( W_\alpha \) to denote the sequence of links of α and |\( W_\alpha \)| to denote the set of links appearing in \( W_\alpha \). We call interaction sites of α, written is(α) and defined in Table 1, those sites where the interaction with another process may happen. These correspond to subjects of the π-calculus. Table 1 also defines the free names fn(), bound names bn(), objects obj(), input objects obj_in(). Their extensions to multisets is as expected. We have to be careful with the following cases:

\[
fn((a)\Lambda) = fn(\Lambda) \setminus \{a\} \cup L_a \\
bn((a)\Lambda) = bn(\Lambda) \cup \{a\} \cup (L_a \cap n(\Lambda))
\]

Notice that both the datum and the destination site are objects: this is analogous to actual routing, where a payload and its destination address travel together within a packet. We introduce some terminology for concurrent paths.

**Definition 4.2** Let Λ be a concurrent path. Then it is:
- well-formed if for every subterm Λ' of the form (a)Λ'' we have fn(Λ') = fn(Λ'') \ {a}, and for all \( l_{ab}, l'_{a'b'} \in fn(\Lambda) \) we have \( l \neq l' \) if \( ab \neq a'b' \);
- in canonical form if it has the form (R)Θ, where R is a sequence of restrictions and Θ does not contain extrusion restrictions (binders of the form ab(s) are still allowed in Θ);
- simple if, for all \( \alpha \in \Lambda \), each \( l_{ab} \in |W_\alpha| \) appears in \( W_\alpha \) once.

An example of non-well-formed concurrent path is (d)(•;l_{ab};bcd | a;l_{ad};d), be-
cause (d) implicitly binds $l_{ad}$, and $l_{ab}, l_{ad}$ have the same label but different endpoints. An example of non-simple concurrent path is $a; l_{ab}; l_{cd}; l_{ab}; b$, because there are two occurrences of $l_{ab}$. Simplicity is just one of the possible conditions. In general, one might want to express QoS requirements; this could be achieved through a type system that associates quantitative information to links.

Now we introduce the hiding operation, which we will use in the SOS rules to implement the effects that restricting a name of a process has on its paths.

**Definition 4.3** The hiding operation $/r$ acts on sequences of links as follows:

$$
eq r = \epsilon$$

$$(W; W')/r = (W/r); (W'/r)$$

$$l_{ab}/r = \begin{cases} \epsilon & r \in \{a, b, l_{ab}\} \\ l_{ab} & \text{otherwise} \end{cases}$$

Its extension to paths $\alpha/r$ is obtained by replacing $W_\alpha$ with $W_\alpha/r$ in $\alpha$. Its extension to concurrent paths $\Lambda/r$ applies the same operation to each $\alpha \in \Lambda$ if $r \notin \text{bn}(\Lambda)$, yields $\Lambda$ otherwise.

We can now present the **NCPi transition system**.

**Definition 4.4** The NCPi transition system is the smallest transition system generated by the rules in Fig. 3, where observations are up to $\equiv$ and transitions are closed under $\equiv$, i.e. if $p \Rightarrow q$, $p \equiv p'$ and $q \equiv q'$ then $p' \Rightarrow q'$.

The rules (fr-in) and (bnd-in) infer the reception of a global and a private name, respectively, while (out) infers the emission of a global name. These actions are represented as paths of length zero. As in the early $\pi$-calculus, a renaming must be applied to the continuation in the free input case; by well-formedness, such renaming can always be extended to act as a proper graph homomorphism. The reception of a global link should be treated carefully: the rule forbids it whenever another link with the same label, but different endpoints, already occurs free in the process, because the renaming would break well-formedness. The rule (link) is used to provide a transportation service to the environment, but we forbid services from a site to itself. The rule (internal) infers a transition labelled with the empty path $\bullet$, representing an internal action. The rule (res) infers a transition of $(r)p$ from the transitions of $p$, but it considers only those transitions such that $r$ is not an interaction site and is not sent or received. This side condition reflects that of the corresponding $\pi$-calculus rule, where $r$ must not be the subject or the object of the premise’s action, and its purpose is to avoid captures: e.g. if $(a)b(c).p$ is such that $c \in \text{fn}(p)$ and it is allowed to perform $bba; \bullet$, then $a$ would be captured in the continuation $(a)p[a/c]$. The rule (open) infers a scope extrusion, provided that the name to extrude is used as object in the premise’s concurrent path, but not as datum of an input or as interaction site. Notice that the rule allows one to “extrude” the destination site: the intuition is that we can use global resources to send or receive a datum to/from a local site, which becomes global if the communication is not complete. The rule (sum-l) is an obvious extension of the corresponding $\pi$-calculus rule. The rule (idle) infers a “no-op” transition, enabling the parallel
The concurrent path inferred by (COM), (SRV-IN), (SRV-OUT) and (SRV-SRV) must be simple.

Fig. 3. NCPI operational rules: (a) shows the SOS rules; (b) and (c) are the possible configurations for (COM). Any pair of configurations, one from (b) and one from (c), is valid (four possibilities).
composition of processes to behave in an interleaving style. The rule (par) makes
the union of two concurrent paths, but only if we do not lose well-formedness due
to inconsistent link labels and if the concurrent path of each of its premise has
bound names which are fresh w.r.t the other process and distinct from all the
names occurring in the other concurrent path. This last condition avoids inferring
transitions where the extruded name is free in the receiving process’s continuation
even if it has not been actually received, which might cause incorrect behaviors.
For instance, consider the processes
\[ p = (b)\text{aab}.b(c).p' \] and \[ q = a(d).\overline{d}de.q' \], and
suppose \[ p \mid q \overset{(b)\text{aab}}{\longrightarrow} b(c).p' \mid \overline{b}e.q' \mid [b/d] \] is allowed; now the two components of
the continuation can synchronize on \( b \) even if its scope extension has not actually
been accomplished, which is clearly incorrect.

The remaining rules are used to synchronize processes. The synchronization is
performed in two steps: 1) paths of parallel processes are collected through the rule
(par); 2) (com), (srv-in), (srv-out) and (srv-srv) take two compatible paths
out of the resulting multiset and replace them with their concatenation, without
modifying the source process; in other words, these rules synchronize two subpro-
cesses of the source process. The rule (com) covers all kinds of communications,
yielding a complete path: the continuation is suitably renamed and, if the involved
paths extrude some names, their restrictions are removed from the transition’s label
and added to the continuation, but only if there are no other paths extruding them.
The rules (srv-in), (srv-out) and (srv-srv) allow extending a path with a service
path. The premises of (com), (srv-in) and (srv-out) must have their concurrent
paths in canonical form: this is always possible, thanks to (par) side conditions.

The following proposition states that the transition system generated by these
rules is well-behaved.

**Proposition 4.5** If \( p \overset{\Lambda}{\longrightarrow} q \) then \( \Lambda \) is simple and well-formed, and \( q \) is well-formed.

We introduce the behavioral equivalence for NCPi processes, called network
conscious bisimilarity.

**Definition 4.6** A binary, symmetric and reflexive relation \( R \) is a network conscious
bisimulation if \( (p, q) \in R \) and \( p \overset{\Lambda}{\longrightarrow} p' \), with:

(i) \( \text{bn}(\Lambda) \cap \text{fn}(q) = \emptyset \);

(ii) \( l_{ab} \in \text{bn}(\Lambda) \cup \text{obj}_{\text{in}}(\Lambda) \Rightarrow \forall a'b' \neq ab : l_{a'b'} \notin \text{fn}(q) \)

implies that there is \( q' \) such that \( q \overset{\Delta}{\longrightarrow} q' \) and \( (p', q') \in R \). The bisimilarity is the
largest such relation and is denoted by \( \sim^{NC} \).

Condition (i) is standard, while (ii) rules out the transitions of \( p \) that \( q \) may not
be able to simulate due to well-formedness. Notice that a consequence of defining
the semantics up to structural congruence is that \( \equiv \subseteq \sim^{NC} \).

**Theorem 4.7** \( \sim^{NC} \) is a congruence with respect to all NCPi operators.

**Proof.** [Hint] This is proved by considering each possible elementary context and
defining a suitable bisimulation closed under that context. The difficult case is the
input prefix, since a renaming, possibly not injective, is involved. The key idea is that renaming a process may allow to apply more (com), (srv-in), (srv-out) or (srv-srv) rules, but the paths these rules concatenate are already observable in the original process, so the new transitions only depend on the original process’ ones. □

We can establish a relation between a subcalculus of the interleaving NC Pi and
the π-calculus.

**Proposition 4.8** Let linkless NC Pi be the subcalculus of NC Pi such that no links appear in processes and the output prefix is of the form \( aab \). Then there is a one-to-one correspondence:

1. between π-calculus processes and NC Pi processes;
2. between π-calculus transitions and NC Pi transitions with singleton labels.

This encoding maps \( ab \) to \( aab \) or \( \bullet aab \), depending on whether it is used as prefix or as action; the other cases are obvious. By homomorphic extension we get the encoding for processes and transitions. Notice that (ii) does not rule out transitions with non-singleton concurrent paths occurring at intermediate derivation steps, e.g. those inferred by (Par).

If we remove the restriction of having only singleton labels, we get a concurrent π-calculus transition system.

**Corollary 4.9 (of Theorem 4.7)** The bisimilarity on the concurrent π-calculus transition system is a congruence.

An evidence of this result is the classical counterexample not applying: we have \( \pi a r | a(x) \not\sim^{NC} aar.a(x) + a(x).aar \), because

\[
\begin{align*}
\pi a r | a(x) & \xrightarrow{\tau a r} aar.a(x) + a(x).aar \xrightarrow{\tau a r} 0 \\
\pi a r | a(x) & \xrightarrow{\tau a r} aar.a(x) + a(x).aar \xrightarrow{\tau a r} 0
\end{align*}
\]

This result is analogous to that in [15] but, as already mentioned, there the synchronization mechanism is not faithful to the π-calculus: in [15] the synchronization channel is observed unless restricted, for instance \( \pi | a \xrightarrow{\tau} 0 \), while for our calculus \( \pi | a \xrightarrow{\tau} 0 \), which corresponds to \( \tau \).

5 Example: routing protocol

In this section we model a simple routing protocol, similar to BGP [25]. This protocol assumes that the network is composed of disjoint groups of networks, each referring to a single administrative authority, called Autonomous Systems (AS). Some of the ASs’ routers act as gateways between the AS they belong to and other networks. The protocol takes care of the routing mechanism between ASs in a distributed manner: each gateway has a routing table, filled by the protocol, whose entries specify which is the next hop along the “best” path towards some destination; this information will be used to forward the incoming data.
In our model, both routers and hosts are represented as sites, and network connections are represented as links. Autonomous systems are generic processes whose links are all restricted, because these links represent local services. The forwarding behavior of each gateway \( g \) is modeled as a process of the form \( L(l_{gh}) \mid \cdots \mid L(l_{gh}) \), where \( L(l_{xy}) \) def \( l_{xy} \cdot L(l_{xy}) \), providing transportation services from \( g \) to other gateways. Routing tables are modeled as functions \( RT_g \) such that \( RT_g(a) \) is a link \( l_{gh} \) to some other gateway \( h \), representing the next hop of the best path towards \( a \). The forwarding is implemented at the SOS level by employing the following rule for gateways:

\[
\begin{align*}
\text{(forward)} & \quad \frac{p \quad (R) (\bullet; W; \text{far} | g; l_{gh}; h | \Theta)}{p'} \quad RT_g(a) = l_{gh} \\
& \quad \frac{p \quad (R) (\bullet; W; l_{gh}; \text{far} | \Theta)}{p'}
\end{align*}
\]

Now, consider the network depicted in Fig. 4. We have three ASs: an Italian one, a German one and an English one, whose gateways are respectively the sites \( \text{it} \), \( \text{de} \) and \( \text{en} \); and we have two processes willing to communicate from site \( a \) in \( A_{\text{it}} \) to site \( b \) in \( A_{\text{en}} \). Suppose there is a path from \( a \) to \( \text{it} \) in \( A_{\text{it}} \), the routing tables are such that \( RT_{\text{it}}(b) = l_{\text{it} \text{de}} \) and \( RT_{\text{de}}(b) = l'_{\text{de} \text{en}} \), and that there is a path in \( A_{\text{en}} \) from \( \text{en} \) to \( b \). Let \( G \) be the process that models forwarding from gateways, namely \( L(l_{\text{it} \text{de}}) \mid L(l'_{\text{it} \text{en}}) \mid L(l'_{\text{de} \text{en}}) \). Then we can infer

\[
A_{\text{it}} \mid A_{\text{en}} \mid A_{\text{de}} \mid G \xrightarrow{\bullet l_{\text{it} \text{de}}; l'_{\text{de} \text{en}}; \bullet} A_{\text{it}}' \mid A_{\text{en}}' \mid A_{\text{de}}' \mid G .
\]

Notice that only the part of the path between the gateways is observable.

6 Conclusions

In this paper we presented NCPi, an extension of \( \pi \)-calculus with an explicit notion of network. To achieve this the syntax is enriched with named connectors. From a semantic point of view, an observation is a snapshot of the traffic on the network, represented as the paths concurrently covered by the data. The semantics’ concurrent nature is the key feature that allows bisimilarity to be a congruence.

---

\[4\] Roles played by sites, such as “gateway”, are stated informally here, but they could be formalized through a type system.
6.1 Related Work

The works most closely related to ours are [10] and [23] where network-aware extensions of $D\pi$ [13] and KLAIM [22] are presented, called respectively $D\pi_F$ and $t$KLAIM. KLAIM is quite far from the synchronous $\pi$-calculus, because it models a distributed tuple-space modifiable through asynchronous primitives, but an encoding to the asynchronous $\pi$-calculus exists [7]. Both $D\pi_F$ and $t$KLAIM are located process calculi, which means that processes are deployed in locations, modeling physical network nodes. In NCPI, instead, processes access the network through sites, possibly more than one for each process, rather than being inside the network. However, locations can be easily introduced in NCPI by a typing mechanism which limits the number of subject names in processes. The network representations are quite different: in $D\pi_F$ locations are explicitly associated with their connectivity via a type system, $t$KLAIM has a special process to represent connections, while in our calculus connections are just names, so the available network nodes and connections correspond to the standard notion of free names. This brings simpler primitives, but also a higher level of dynamism: connections can be created and passed among processes, as shown in §2; this example, in our opinion, is not easily implementable in $t$KLAIM and $D\pi_F$. Moreover, our calculus is more programmable: processes explicitly activate transportation services over connections via the link prefix, while in the cited calculi the network is always available.

We can also cite [11,12,8] as examples of calculi where resources carry some extra information: they explicitly associate costs with $\pi$-calculus channels through a type system. In our case, links could also be typed in order to model services with different features, e.g. performance, costs and access rights.

6.2 Research Directions

Our calculus only captures point-to-point communication, but a network could be used for more complex forms of interaction, e.g. multicast. One possible development direction might be allowing different mechanisms of message exchanging. Moreover, one can think of complex QoS conditions on resources, e.g. restrictions on bandwidth or costs. There is also some room for asynchronous variations, for instance each hop of a routing path could be performed in different transitions. This would capture the step-by-step behavior of SDNs.

Network-awareness is only one form of resource-awareness, which is essential to adequately model new computational paradigms such as cloud computing. Future work includes also the development of an algebraic/coalgebraic categorical model of resource-aware nominal calculi. In particular, the approach based on presheaf models has been successfully applied to the $\pi$-calculus [9], the fusion calculus [17] and the explicit fusion calculus [2]. This approach is especially effective for nominal calculi, because it allows to model resources as a separate category, so to decouple the structure of resources from the syntax and semantics of processes using them. This permits to capture many alternatives with minimal changes. Moreover, coalgebras over a broad class of presheaves can be implemented as HD-automata [5,4], more
concrete operational models that allow for name deallocation and hence are suitable for verification purposes. In our case, the resources of a process are its free sites and links, which can be represented as a finite graph. Functors on the category of resources could allow to create new sites and new links, and to increase their capabilities, similarly to what happens with functor $\delta$ in the presheaf semantics of the $\pi$-calculus.

References


Continuation Models for the Lambda Calculus with Constructors

Barbara Petit

Focus - INRIA
University of Bologna
(Italy)

Abstract

The lambda calculus with constructors decomposes the pattern matching à la ML into some atomic rules. Some of them do not match with the usual computational intuitions (in particular with typing intuitions). However it is possible to define an abstract notion of model for the untyped calculus, that has a trivial syntactic instance.

Nevertheless, the question of devising a non-syntactic model for this calculus was still unresolved. In this paper we answer this question in the untyped setting, by going back to the first motivation of the lambda-calculus with constructors: the simulation of an abstract machine with two independent stacks. This provides immediately a CPS translation into the usual lambda calculus. At the semantic level, it appears that this translation transforms any continuation model of the untyped lambda calculus into a model of the lambda calculus with constructors. In particular, any Scott domain can be turned into such a model.

Keywords: Lambda calculus, Pattern matching, Continuation Passing Style transformation, Categorical semantics, Continuation model.

Introduction

Pattern matching is a key feature in modern functional programming languages (Haskell, Ocaml) and proof assistants (Agda, Coq, Twelf). Since the late 90’s, many formalisms have been proposed to integrate it with lambda calculus [3,9,1,2]. The syntactic properties of these calculi have been thoroughly studied, in both typed and untyped settings, and this led Jay to implement a programming language centred on pattern matching [8].

A more abstract approach to these formalisms could allow a deeper understanding of them, and possibly a comparison between them. As far as we know, no (non syntactical) denotational model has been defined for any of these calculi.

Owing to its simple syntax, the lambda calculus with constructors (or λε-calculus) may be the best one to start with. Indeed, whereas most calculi with

---

1 Email: barbara.petit@ens-lyon.org

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science URL: www.elsevier.com/locate/entcs
pattern matching require the definition of a powerful operation of pattern substitution, the operational semantics of the $\lambda_\xi$-calculus is composed of atomic rules: the pattern matching à la ML is decomposed into a simple analysis on constants (like the case instruction of Pascal), and a commutation rule between the case construction and the application (Sec. 1.2). Although this last rule is rather counter intuitive at first sight (it was presented as “ill-typed” in the introducing paper), the calculus is confluent and enjoys the separation property (in the spirit of Böhm’s theorem), and a type system has also been defined for it [13].

A naive definition of a model can be given in category theory for the untyped lambda calculus with constructors [12]. However it seems difficult to build non syntactic instantiations of this definition. This sends us back to one of the main challenges of theoretical computer science in the late 60’s: to build a denotational model for the pure lambda calculus (i.e. a mathematical structure with a reflexive object $D \cong D^D$). This problem was solved by Scott [16] in 1970, with the construction of a so-called $D_\infty$ domain. It appeared later that such domains are in fact continuation models (characterised by two objects $R$ and $C$ such that $C \cong R^C \times C$) of the pure lambda calculus [15].

The idea underlying these continuation models is to use a CPS translation of the pure lambda calculus into the simply typed lambda calculus with only two basic types (one for the continuations, and one for the responses), and then to use the standard interpretation of the simply typed lambda calculus in a Cartesian closed category. We use the same method in this paper: we define a CPS transformation of the lambda calculus with constructors into the lambda calculus, and then interpret the translated terms in a CCC (with some required isomorphisms). The main difficulty is to interpret the pattern analysers (called case bindings). Indeed, to keep the definition of the models conceptually simpler, we use an operation of composition of case bindings, that has a non trivial translation in the CPS. However, the translation is correct, and provides as expected a sound definition of continuation models for the lambda calculus with constructors.

Outline: In the first section, we give an intuitive presentation of the lambda calculus with constructors, by defining an abstract machine for it. A CPS translation naturally results from this machine; we formalise it in Sec. 2. In the last section, we give the categorical definition of models (Sec. 3.1), and of continuation models (Sec. 3.2) for the lambda calculus with constructors, and we show that the second ones form a subclass of the first ones (Sec. 3.3). Finally we show that good candidates for continuation models of the $\lambda_\xi$-calculus already exist (like Scott’s domains for instance).

1 Lambda calculus with constructors

1.1 First approach: a two stack abstract machine

We extend the syntax of the lambda calculus with a finite set $\mathcal{C}$ of constructors (c, d etc.) and a case construct $\{\theta\} \cdot t$, where $t$ is a term and $\theta$ a case binding, i.e. a
partial function from constructors to terms:

\[ \theta := \{ c_1 \mapsto t_1; \cdots ; c_k \mapsto t_k \} \]

The *domain* \( \{c_1; \cdots ; c_k\} \) of this case binding is denoted by \( \text{dom}(\theta) \), and \( \theta_{c_i} \) represents the term \( t_i \). Pattern matching occurs when such a case binding is associated to a constructor of its domain, just like a case analysis on constants:

\[ \{\theta\} \cdot c \rightarrow t \quad \text{if} \quad c \rightarrow t \in \theta \]

The conditional branching, testing a Boolean and returning \( t \) or \( u \) if it is *true* or *false* respectively (where *true* and *false* are constructors), is then written

\[ \text{if}t,u \text{false} = \lambda x.\{\text{true} \mapsto t; \text{false} \mapsto u\} \cdot x. \]

In this language, there are now two different kinds of values: the functions (the usual \( \lambda \)-abstractions) and the constructors\(^2\). Each of them can be evaluated by the corresponding construction in a context: the argument of an application and the case binding of a case construct respectively. Also we can extend the Krivine abstract machine [5] to this syntax, by replacing the stack of arguments originally composing the evaluation context by two stacks: one (say the “right stack”) for the arguments, and the other one (the “left stack”) for the case bindings. When a term is evaluated in this machine, it then interacts with the left stack if it is a case construct or a case binding, and with the right stack if it is an application or a \( \lambda \)-abstraction. This machine is formally defined in Fig. 1. Evaluating the term \( (\text{if}t,u \text{false}) \) in this machine (starting with two empty stacks) will indeed lead to the configuration \( \diamond \ast u \ast \diamond \). But this machine can also simulate the pattern matching on compound data structures.

---

| Terms: \( t, u := x \mid tu \mid \lambda x. t \mid c \mid \{\theta\} \cdot t \) |
| Case bindings: \( \theta, \phi := \{ c_1 \mapsto t_1; \cdots ; c_k \mapsto t_k \} \quad (k \geq 0) \) |
| Application stacks: \( \pi := \diamond \mid t \cdot \pi \) |
| Case stacks: \( \tau := \diamond \mid \tau \cdot \theta \) |
| Processes: \( s := \tau \ast t \ast \pi \) |

**Execution rules:**

\[
\begin{align*}
\text{Pop} & \quad \tau \ast \lambda x. t \ast u \cdot \pi & \Rightarrow & \tau \ast t[x := u] \ast \pi \\
\text{Push} & \quad \tau \ast tu \ast \pi & \Rightarrow & \tau \ast t \ast u \cdot \pi \\
\text{Pop}_c & \quad \tau \cdot \theta \ast c \ast \pi & \Rightarrow & \tau \ast \theta \ast \pi \\
\text{Push}_c & \quad \tau \ast \{\theta\} \cdot t \ast \pi & \Rightarrow & \tau \cdot \theta \ast t \ast \pi 
\end{align*}
\]

---

\( ^2 \) This second kind of values will be elaborated in Sec. 1.2.
Notice that constructors, just as any terms, can be applied to any number of arguments (they are variadic). We call a data structure a constructor possibly applied to some arguments. For instance, one can represent the natural numbers by data structures, using two constructors \( S \) and \( 0 \) and the unary encoding of natural numbers. The predecessor function is then written \( \text{pred} := \lambda x.\{0 \mapsto 0; S \mapsto \lambda z.z\}.x \). Its application to a number \( S^n \) is actually evaluated to \( n \) (we skip the first \( \beta \)-reduction steps):

\[
\begin{align*}
\circ \ast \text{pred } (S^n) & \ast \circ \\
| & \circ \ast \{\theta\} \cdot S^n \ast \circ \\
| & \circ \cdot \theta \ast S^n \ast n \circ \\
| & \circ \ast \lambda z.z \ast n \circ \\
| & \circ \ast n \ast \circ
\end{align*}
\]

(where \( \theta = \{0 \mapsto 0; S \mapsto \lambda z.z\} \)). More generally, any pattern matching with a branch \( \text{C}(x_1, \ldots, x_k) \rightarrow t \) in a ML-like program behaves like a term with a branch \( c \mapsto \lambda x_1 \ldots x_k.t \) evaluated in our machine. In this sense, the machine presented above is able to simulate pattern matching on elaborated data structures with a simple rule of constant analysis. The same idea is underlying the lambda calculus with constructors (or \( \lambda_C \)-calculus).

### 1.3 Operational semantics of the \( \lambda_C \)-calculus

The ML-style pattern matching is achieved in the double stack abstract machine by giving a double status to the constructors: they can be applied to some arguments to form a compound data structure (in this case they interact with the application stack), but they can also be seen as a constant to analyse by the case bindings (and then interact with the case stack). In the semantics setting, this context switching corresponds to the following commutation rule between case and application constructs:

\[
\{\theta\} \cdot (tu) \simeq (\{\theta\} \cdot t) \ u .
\]

This is a crucial rule of the lambda calculus with constructors (Fig. 2), called \textsc{CaseApp} (or \textsc{ca} for short). In addition to this rule, the calculus supports the usual \( \beta \) and \( \eta \)-reductions (resp. \textsc{al} and \textsc{la}), and the rule of constant analysis that we
have seen earlier (co). There are also a commutation rule between case construct and \(\lambda\)-abstractions (cl), and a composition rule for case bindings (cc) so that the \(\lambda\_C\)-calculus enjoys confluence and separation properties [1]. Writing \(\rightarrow^{*}\) the transitive closure of the reduction relation \(\rightarrow\), one can check that \(\text{pred} (Sn) \rightarrow^{*} n\), using rules al, ca and co.

Whereas the rule CaseApp does not match with the usual typing intuitions (the same subterm can be applied like a function, or pattern matched like a data structure), the case composition corresponds to a commutative conversion in logic [6, Sec. 10.4]:

\[
\{\theta\} \cdot \{c_1 \mapsto u_1; \ldots; c_n \mapsto u_n\} \cdot t \rightarrow \{\theta\} \cdot \{u_1; \ldots; u_n\} \cdot t
\]

Concerning evaluation contexts, this rule amounts to merging all case bindings of a case stack \(\diamond \cdot \theta_1 \cdots \theta_k\) in only one (optional) case binding \(\theta \circ \cdots \circ \theta_k\) (with left associativity of \(\circ\)). Also we consider the following alternative abstract machine for the \(\lambda\_C\)-calculus, that we call the KAM\(_{\lambda\_C}\):

**Definition 1.1** (The KAM\(_{\lambda\_C}\)). A *process* is a triple \(\langle \theta \rangle \star t \star \pi\), where \(\langle \theta \rangle\) is an optional case binding (\(\diamond\) or \(\theta\)), \(t\) is a term and \(\pi\) a stack of terms. The four *execution rules* are

\[
\begin{align*}
\langle \theta \rangle & \star \lambda x.t \star u \cdot \pi & \rightarrow & & \langle \theta \rangle & \star t[x := u] \star \pi \\
\langle \theta \rangle & \star tu \star \pi & \rightarrow & & \langle \theta \rangle & \star t \star u \cdot \pi \\
\theta & \star c \star \pi & \rightarrow & & \diamond & \star \theta_c \star \pi \\
\langle \theta \rangle & \star \{\phi\} \cdot t \star \pi & \rightarrow & & \langle \theta \rangle \circ \phi & \star t \star \pi
\end{align*}
\]

where \(\langle \theta \rangle \circ \phi\) is \(\phi\) if \(\langle \theta \rangle = \diamond\), and \(\theta \circ \phi\) if \(\langle \theta \rangle = \theta\).

Actually the rule CaseCase is not absolutely necessary from the computational point of view (it is only necessary for the separation property). Hence we could also consider the \(\lambda\_C\)-calculus without cc (the \(\lambda\_C\)-calculus), and the first version of abstract machine we have presented. This would also lead to a slightly different notion of models (see the footnotes 3 and 5).

In the next section, we will use this machine to translate the lambda calculus with constructors into the pure lambda calculus.

### 2 CPS translation

Plotkin [14] used stack abstract machines to define *continuation passing style* (cps) translations between the call-by-name and call-by-value \(\lambda\)-calculi. Indeed, the stack of the machine can be encoded with pairs in the \(\lambda\)-calculus. In the same way, we will define a CPS translation of the \(\lambda\_C\)-calculus into the lambda calculus with pairs (or \(\lambda\_p\)-calculus) based on the KAM\(_{\lambda\_C}\). A \(\lambda\_C\)-term \(t\) will be translated by a \(\lambda\_p\)-term \(t^*\) that takes an evaluation context \(k\) (the *continuation*) in argument, and that returns the result of the evaluation of \(t\) with context \(k\) in the machine.
A continuation is a pair \( \langle | \rangle \).

### 2.1 Target calculus

From now on, we write \( M \) respectively the case binding and the application stack of the evaluation context.

If \( \alpha \) is a \( \lambda \)-variable, a \( \lambda \) abstraction and an application exactly correspond (after forgiving the “case” part of the continuation) to the translation \( c.b.v.-c.b.n. \) of Plotkin [14]. The translation of a constructor \( c_i \) consists in giving the application context to the \( i^{th} \) component of the case context \( (x_i) \).

Terms of the \( \lambda \)-calculus are given by the following grammar and rules:

\[
M, N, P ::= \quad x \mid \lambda x.M \mid MN \mid * \mid \emptyset \mid \pi_i(M) \quad (i \in \{1,2\})
\]

\[
(\lambda x.M) N \rightarrow_P M[x := M] \quad ; \quad \lambda x.(Mx) \rightarrow_P M \quad (x \notin \text{fv}(M))
\]

\[
\pi_i(\emptyset, M_2) \rightarrow_P M_i \quad ; \quad \pi_i(M_1, M_2) \rightarrow_P M
\]

We use the same names for variables than in the \( \lambda \)-calculus, although we may also write \( k \) for some \( \lambda \)-variables (representing a continuation). We use the notations \( \emptyset, M_1, \ldots, M_\ell \) and \( \pi_1(M) \) for the usual encoding of tuples and generalised projections with pairs. We also write \( \text{let } \emptyset, x_1, \ldots, x_\ell \rightarrow P \) in \( M \) for the term \( (\lambda x_1, \ldots, x_\ell.M)\pi_1(P) \ldots \pi_\ell(P) \) (when \( \ell \) is not specified it is 2), so that \( \text{let } \emptyset, x_1, \ldots, x_\ell \rightarrow \emptyset, N_1, \ldots, N_\ell \rightarrow M \rightarrow_P x_1 := N_1 \ldots x_\ell := N_\ell \).

### 2.2 The CPS translation

The translation of a \( \lambda \)-term \( t \) in the \( \lambda \)-calculus is then given by:

\[
t^* := \lambda k.\text{let } \emptyset, x_\pi \rightarrow k \text{ in } \emptyset, t \rightarrow x_\pi
\]

where the result of \( t \) in context \( \emptyset \) (where \( M_\theta \) and \( M_\pi \) are two \( \lambda \)-terms), denoted by \( |\emptyset, t \rightarrow M_\theta \rangle \), is defined by induction in Fig. 3. The translations of a variable, a \( \lambda \)-abstraction and an application exactly correspond (after forgiving the “case” part of the continuation) to the translation \( c.b.v.-c.b.n. \) of Plotkin [14]. The translation of a constructor \( c_i \) consists in giving the application context to the \( i^{th} \) component of the case context \( (x_i) \).

Remark that no case context is given (we use the term \( * \)), since \( x_i \) comes with its own case context (Ex. 2.1). The translation of
a case construct amounts to composing the (translated) case binding with the case context.

**Example 2.1** Let $\psi = \{c_i \mapsto u_i/i \in S \subseteq [1..n]\}$ and $\phi = \{c_i \mapsto s_i/i \in S' \subseteq [1..n]\}$. Then the result of the term $u = \{\phi\} \cdot \{\psi\} \cdot (c_j \ t)$ (with $j \in \text{dom}(\psi)$) is:

$$|M_\theta \ast u \ast M_\pi| = \langle \langle N_1, \ldots, N_n \rangle_n \ast \{\psi\} \ast (c_j \ t) \ast M_\pi\rangle$$

(with $N_i = \lambda k . \text{let } \langle z_0, z_1 \rangle \equiv k \text{ in } |M_\theta \ast s_i \ast z_\pi| \text{ if } i \in \text{dom}(\phi)$)

$$|M_\theta \ast u \ast M_\pi| = \langle \langle P_1, \ldots, P_n \rangle_n \ast c_j \ t \ast M_\pi\rangle$$

$$|M_\theta \ast u \ast M_\pi| = \langle \langle P_1, \ldots, P_n \rangle_n \ast c_j \ \langle t^*, M_\pi\rangle\rangle$$

$$|M_\theta \ast u \ast M_\pi| = \langle \langle P_1, \ldots, P_n \rangle_n \ast c_j \ \langle t^*, M_\pi\rangle\rangle$$

$$|M_\theta \ast u \ast M_\pi| = \langle \langle P_1, \ldots, P_n \rangle_n \ast c_j \ \langle t^*, M_\pi\rangle\rangle$$

This translation enables the simulation of the lambda calculus with constructors in the lambda calculus with pairs (Theo. 2.4). This result derives from the following lemmas (proved by a trivial induction on $t$):

**Lemma 2.2** Let $t, t', u$ be three $\lambda_\eta$-terms, and $x$ a variable not free in $t'$. Then for any $\lambda_\nu$-terms $N, M_\theta, M_\pi$,

(i) $|M_\theta \ast t' \ast M_\pi| \ [x := N] = |M_\theta[x := N] \ast t' \ast M_\pi[x := N]|$

(ii) $|M_\theta \ast t \ast M_\pi| \ [x := u^*] \rightarrow^*_{\nu} |M_\theta[x := u^*] \ast t[x := u^*] \ast M_\pi[x := u^*]|$

(iii) $M_\theta \rightarrow^* M'_\theta \implies |M_\theta \ast t \ast M_\pi| \rightarrow^* |M'_\theta \ast t \ast M_\pi|$

$$M_\pi \rightarrow^* M'_\pi \implies |M_\theta \ast t \ast M_\pi| \rightarrow^* |M'_\theta \ast t \ast M'_\pi|$$

**Lemma 2.3** For any $\lambda_\eta$-terms $t, u$, any case-bindings $\phi, \psi$, and any $\lambda_\nu$-terms $M_\theta, M_\pi$,

$$|M_\theta \ast (\lambda x . t) u \ast M_\pi| \rightarrow^* |M_\theta \ast t[x := u] \ast M_\pi|$$

$$|M_\theta \ast \lambda x . t x \ast M_\pi| \rightarrow^* |M_\theta \ast t \ast M_\pi| \quad \text{if } x \notin \text{fv}(t)$$

$$|M_\theta \ast \{\phi\} \cdot c \ast M_\pi| \rightarrow^* |M_\theta \ast u \ast M_\pi| \quad \text{if } c \mapsto u \in \phi$$

$$|M_\theta \ast \{\phi\} \cdot t u \ast M_\pi| = |M_\theta \ast (\{\phi\} \cdot t) u \ast M_\pi|$$

$$|M_\theta \ast \{\phi\} \cdot \lambda x . t \ast M_\pi| = |M_\theta \ast \lambda x . \{\phi\} \cdot t \ast M_\pi| \quad \text{if } x \notin \text{fv}(\phi)$$

$$|M_\theta \ast \{\phi\} \cdot \{\psi\} \cdot t \ast M_\pi| = |M_\theta \ast \{\phi \circ \psi\} \cdot t \ast M_\pi|$$

Notice that the only $\lambda_\eta$-rules that are actually simulated by some reduction steps in the $\lambda_\nu$-calculus are the $\beta$ and $\eta$-reductions, and the constant analysis. The other rules correspond to the management of the stacks by the machine, and are simulated during the CPS translation.

**Theorem 2.4 (Correct simulation)** For any $\lambda_\eta$-terms $t, t'$,
2.3 Consequences for denotational models

The simulation theorem provides a sound interpretation of the $\lambda_{\ell}$-calculus in any model of the $\lambda_{p}$-calculus. Indeed, if $[\cdot]$ is an interpretation of the $\lambda_{p}$-terms that equalises the equivalent ones, then $t \simeq_{\lambda_{p}} t'$ implies $t^* \simeq_{\lambda_{p}} t'^*$ (by Theo. 2.4 and the Church-Rosser property) and thus $[t^*] = [t'^*]$ in the model (for each calculus $\mathcal{L}$ presented in this paper, we write $\simeq_{\mathcal{L}}$ the reflexive symmetric and transitive closure of its reduction rules).

In the next section, we give a categorical definition of models for the $\lambda_{\ell}$-calculus, and we show how to transform a model of the lambda calculus with pairs into a $\lambda_{\ell}$-model. This transformation of models will directly come from the CPS translation we have just presented.

3 Classical models for the $\lambda_{\ell}$-calculus

In this section we briefly present what is a categorical model for the $\lambda_{\ell}$-calculus (more details and proofs can be found in [12]), and we show that the continuation models of the pure lambda calculus have the good structure to be seen as $\lambda_{\ell}$-models.

Notations: In a Cartesian closed category (ccc), we write $Id_A$ the identity morphism on $A$, and $f;g$ the composition of $f$ and then $g$. We denote by $A \times B$ the product of two objects $A$ and $B$, and by $B^A$ their exponent, and by $1$ the terminal object. The $i^{th}$ projection morphisms over $k$ is written $\pi^k_i$ (or $\pi_i$ if $k = 2$), the pairing of $f$ and $g$ is $\langle f, g \rangle$, $ev$ is the evaluation morphisms and $\Lambda(f)$ the curried form of $f$.

3.1 Categorical models of the $\lambda_{\ell}$-calculus

In category theory, a model for the pure lambda calculus is a ccc with a reflexive object $D \cong D^D$. Indeed, $\lambda$-terms are interpreted in $D$, and points of $D^D$ are functions from terms to terms (i.e. open terms, abstracted over a free variable). Then a morphism $\text{lam} : D^D \to D$ enables to construct the denotation of $\lambda x.t$, from the representation of the function mapping $x$ to $t$. In the same way, a morphism $\text{app} : D \to D^D$ allows to interpret the application of any term to an other one. Also the equality $\text{app} \circ \text{lam} = Id$ ensures that the interpretation respects $\beta$-equivalence.

To interpret the $\lambda_{\ell}$-calculus in such a category, some extra morphisms are necessary (for the interpretation of the constructors and the case construct), as well as some equalities between them to validate the Case rules. A case binding $\theta$ will be interpreted in $D^n$: the $i^{th}$ component corresponds to $\theta c_i$ if it is defined, and is a special point $\sharp$ (meaning match failure) of $D$ otherwise. Then a morphism $\text{case} : (D^n \times D) \to D$ is required to interpret the case construct $(\theta \cdot t)$, given the denotation of $\theta$ in $D^n$ and the one of $t$ in $D$. We also need a point $c_i^*$ of $D$ for every constructor $c_i \in \mathcal{C}$.
**Definition 3.1 (λφ-model)** A categorical model for the untyped λφ-calculus is a structure \((C, D, \text{app}, \text{lam}, (c^*_i)_{i=1}^n, \dagger, \text{case})\) where

- \(C\) is a Cartesian closed category, and \(D\) is one of its object.
- \(\text{app} : D \to D^D\) and \(\text{lam} : D^D \to D\) form an isomorphism: \(D \cong D^D\).
- All the \(c^*_i\)'s and \(\dagger\) are points of \(D\), and \(\text{case}\) is a morphism of \(D^n \times D \to D\),
- The four diagrams of Fig. 4 commute ((\(D1\) commutes for every \(i \in [1..n]\)).

<table>
<thead>
<tr>
<th>CaseCons</th>
<th>CaseApp</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D1)</td>
<td>(D2)</td>
</tr>
<tr>
<td>[D^n \cong D^n \times 1]</td>
<td>[D^n \times D \rightarrow D^n \times (D \times D)]</td>
</tr>
<tr>
<td>(\pi^n_i)</td>
<td>(\text{case} \times \text{Id})</td>
</tr>
<tr>
<td>(D \leftarrow \text{case} D^n \times D)</td>
<td>(D \rightarrow \text{Id} \times (\text{app} \times \text{Id}))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CaseCase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D3)</td>
</tr>
<tr>
<td>[D^n \times 1 \rightarrow D^n \times D]</td>
</tr>
<tr>
<td>(\pi^2_2)</td>
</tr>
<tr>
<td>(1 \rightarrow \dagger)</td>
</tr>
<tr>
<td>(D \rightarrow \text{case} D^n \times D)</td>
</tr>
</tbody>
</table>

Figure 4. Commuting diagrams in a λφ-model

The equalities of morphisms described in Fig. 4 ensure that the interpretation we have informally presented respects λφ-equivalence. It is pretty clear how the commutation of diagrams \((D4)\) and \((D2)\) entail the validity of rules CaseCons and CaseApp respectively. The validity of the rule CaseCase is expressed through a morphism \(\bullet : D^n \times D^n \rightarrow D^n\) (\(D4\)), that represents the case binding composition in the categorical framework (Lem. 3.3). It is defined as the pairing of the morphisms \((\text{Id}_{D^n} \times \pi^n_i)\); \(\text{case}\), for \(1 \leq i \leq n\).

The diagram \((D3)\) is the only one that does not directly translate a reduction rule of the λφ-calculus. It expresses the equivalence between a match failure and the matching of a match failure\(^3\), and is necessary for the soundness \(w.r.t.\) the rule CaseCase.

\(^3\) If we enrich the λφ-calculus with explicit match failure (that is, a special constant \(\dagger\) and the rule \(\{\theta\} \cdot c \rightarrow \dagger\) if \(c \notin \text{dom}(\theta)\)), then we need an extra rule \(\{\theta\} \cdot \dagger \rightarrow \dagger\) for confluence (to close the critical pair...
Petit

Notice that there is no diagram corresponding to the rule CaseLam in the definition of a $\lambda \mathcal{E}$-model. In the same way that the rule $cl$ closes a critical pair between $ca$ and $al$, the commutation of the diagram corresponding to it ($D2'$) is induced by the commutation of ($D2$) and the reflexivity of $D$, as expressed by Lem. 3.2. This diagram uses a morphism $\text{case}^o : D^n \times D^D \to D^D$, that abstracts the case construct over a variable: it turns a case binding $\theta$ and a function mapping $x$ to $t$ into the function mapping $x$ to $\{\theta\} \cdot t$, and it is formally defined as the curried form of

$$(D^n \times D^D) \times D \cong D^n \times (D^D \times D) \xrightarrow{\text{Id}_{D^n} \times \text{ev}} D^n \times D \xrightarrow{\text{case}} D.$$ 

**Lemma 3.2 (Diagram ($D2'$))** If $(\text{app, lam})$ form an isomorphism between $D$ and $D^D$, then the commutation of ($D2$) is equivalent to the commutation of the following diagram:

$$
\begin{array}{ccc}
D^n \times D^D & \xrightarrow{\text{case}^o} & D^D \\
\downarrow \text{Id} \times \text{lam} & & \downarrow \text{lam} \\
D^n \times D & \xrightarrow{\text{case}} & D
\end{array}
$$

This enables to define a sound interpretation of $\lambda \mathcal{E}$-terms in any $\lambda \mathcal{E}$-model: if $t$ has its free variables included in $\Gamma = (x_1, \ldots, x_k)$, then its interpretation $[t]_\Gamma$ is defined by induction in Fig. 5.

\[
\begin{align*}
[x_i]_\Gamma &= \pi^k_i : D^k \to D \\
[tu]_\Gamma &= \begin{cases} 
D^k \times D^{\{[t]_\Gamma,[u]_\Gamma\}'} \to D \times D \xrightarrow{\text{app} \times \text{Id}_D} D^D \times D \xrightarrow{\text{ev}} D
\end{cases} \\
[\lambda x_{k+1}.t]_\Gamma &= D^k \xrightarrow{\Lambda(f_t)} D^D \xrightarrow{\text{lam}} D \\
where \ f_t &= D^k \times D \xrightarrow{\cong} D^{k+1} \xrightarrow{[t]_\Gamma,[x_{k+1}]} D \\
[c]_\Gamma &= D^k \xrightarrow{\text{ev}^*} 1 \xrightarrow{!} D \\
\{[\theta] : t\} &\xrightarrow{\Lambda(\theta)} D^k \times D \xrightarrow{\text{case}} D \\
\{\theta\} &\xrightarrow{\{f_1,\ldots,f_n\}} : D^k \to D^k, \text{ where } f_i = \begin{cases} 
[u_i]_\Gamma & \text{if } c_i \mapsto u_i \in \theta \\
!_{D^k} & \text{if } c_i \notin \text{dom}(\theta)
\end{cases}
\end{align*}
\]

**Figure 5.** Interpretation of $\lambda \mathcal{E}$-terms in a categorical model

**Lemma 3.3 (Categorical case composition)** If $\theta$ and $\phi$ are two case bindings whose free variables are all in $\Gamma$, then their interpretation in any $\lambda \mathcal{E}$-model satisfies

$$[\theta \circ \phi]_\Gamma = D^k \xrightarrow{\Lambda([\theta]_\Gamma,[\phi]_\Gamma)} D^n \times D^n \xrightarrow{\text{case}} D^n.$$ 

**Proposition 3.4 (Soundness)** If $(\mathcal{C},D,\text{app, lam},(c_i^*)_{i=1}^n,\text{case})$ is a $\lambda \mathcal{E}$-model, then Fig. 5 interprets each closed $\lambda \mathcal{E}$-term $t$ by a point $[t]_D$ of $D$ such that

with CC). The model we present here would still be sound for the extended calculus, and the diagram ($D3$) corresponds to this last rule. Alternatively, if we remove the rule $\text{CASECASE}$ form the calculus, then the commutation of ($D1$) and ($D2$) is sufficient.
In fact \( \lambda_{\&}\)-models are even complete for the sub calculus with no match failure [12].

3.2 Continuation models

A Cartesian closed category is a model of the pure \( \lambda \)-calculus if it has an object \( D \) equivalent to its function space, in which we can interpret the \( \lambda \)-terms. Among them are the \textit{continuation models} \(^4\) (see the excellent introduction of [15]): 

\[ \text{CCC with two objects } C \text{ and } R \text{ satisfying the equation } C \simeq C \times R. \]

Indeed, taking \( D = R \) fulfills the condition \( D \simeq D \), and leads to interpret the \( \lambda \)-terms by points of \( R \), \( \text{i.e. informally by functions taking a \textit{continuation argument} in } C, \text{ and returning a \textit{response} in } R. \) A functional term (\text{i.e. a point of } \( D \)) is interpreted in \( (R^C)^R \simeq R \), also the continuation argument of a function is a point of \( C \times R \). It represents a pair composed of \textit{later continuation} (in \( C \)) and a term (in \( R \)) that is the argument of the function. That is why we can see the continuations as \textit{stacks of arguments}, and the term interpretations in a continuation model as \textit{processes} in a stack abstract machine.

As we have seen earlier (Sec. 2.1), a continuation for the \( \lambda_{\&}\)-calculus should not be only a stack of arguments, but a pair composed of a case binding (\text{i.e. a point of } \( D_n \)) and a stack (\text{i.e. a point of some object } S \text{ satisfying the stack equation}). This gives rise to the following definition.

\textbf{Definition 3.5 (Continuation \( \lambda_{\&}\)-model)} A \( \text{CCC} \) is a \textit{continuation} \( \lambda_{\&}\)-model (or \textit{classical} \( \lambda_{\&}\)-model) if it has four objects \( R, C, S, D \) satisfying the following equations:

\[ D \simeq R^C; \quad C \simeq D_n \times S; \quad S \simeq D \times S. \]

In the next section, we show that every continuation \( \lambda_{\&}\)-model is actually a \( \lambda_{\&}\)-model in the sense of Def. 3.1. It might be a bit tedious to describe the morphisms \texttt{app}, \texttt{lam}, \texttt{c^*} and \( \downarrow \) in a continuation model (and still more to prove the diagrams commutation) with compositions and curried forms; and so we use the \( \lambda \)-calculus with pairs (as an \textit{internal language} for \text{CCCs}) to define those morphisms. Given a Cartesian closed category \( \mathbb{C} \), we call its internal language the \( \lambda_{\mathbb{C}} \)-calculus, and we write \( \simeq_{\mathbb{C}} \) the equivalence of terms in this language.

3.3 From continuation \( \lambda_{\&}\)-models to \( \lambda_{\&}\)-models

Let \( \mathbb{C} \) be a continuation \( \lambda_{\&}\)-model (Def. 3.5). We write \( \uparrow_s, \downarrow_s, \uparrow_c \) and \( \downarrow_c \) the terms (resp. of type \( S \to D \times S, D \times S \to S, C \to D_n \times S \) and \( D_n \times S \to C \)) corresponding to the morphisms that guarantee \( S \simeq D \times S \) and \( C \simeq D_n \times S \). We show that \( \mathbb{C} \) can be provided with the structure of a \( \lambda_{\&}\)-model (Def. 3.1). To do so, we refer

---

\(^4\) Also called \textit{classical models}, as their underlying logic is the classical logic [10].

\(^5\) Without the case composition, the case context would be not only a case binding (in \( D_n \)), but a stack of case bindings. Hence we would need a fourth object \( S' \) satisfying the equation \( S' \simeq D_n \times S' \), and the interpretation of terms in such a model would be more complex.
to the \( \lambda_c \)-terms defined in Fig. 6. The terms \( M_{\text{lam}}, M_{\text{app}} \) and \( M_{\text{case}} \) have a free variable \( z \), that will correspond (through the mapping from the \( \lambda_c \)-calculus to \( \mathbb{C} \)) to the arguments of \( \text{lam}, \text{app} \) and \( \text{case} \) respectively. Remark that there is a direct connection between the terms defined in Fig. 6 and the CPS translation of Sec. 2 (Fig. 3), given that \( |M_\theta \ast t \ast M_\pi| \simeq \lambda_c \uparrow t \langle M_\theta, M_\pi \rangle \).

**Definition 3.6** The morphisms defining a \( \lambda_c \)-model are given by the following derivable judgements:

\[
\begin{align*}
\text{lam} & \quad \vdash_M \lambda z. \pi_1(z) : D \\
\text{app} & \quad \vdash_M \lambda z. \pi_2(z) : D \\
\text{case} & \quad \vdash_M \lambda z. \pi_3(z) : D \\
\end{align*}
\]

**Theorem 3.7 (\( \mathbb{C} \) is a \( \lambda_c \)-model)** In a Cartesian closed category \( \mathbb{C} \) that is a continuation \( \lambda_c \)-model, the morphisms defined in Def. 3.6 satisfy the diagrams in Fig. 4, and the morphisms \( \text{lam} \) and \( \text{app} \) form an isomorphism between \( D \) and \( D^D \). Also \( (\mathbb{C}, D, \text{app}, \text{lam}, (c^*)^n, \text{case}) \) is a \( \lambda_c \)-model.

**Proof (sketch).** All the equalities on morphisms can be proved using the internal language: two morphisms are equal if their corresponding terms are convertible. For instance, the equality \( \text{lam}; \text{app} = \text{Id}_{D^D} \) follows from the \( \lambda_c \)-convertibility of \( \text{lam}; \text{app} \) (i.e. \( \lambda z. M_{\text{app}}[z := \text{lam}] \)) and \( \lambda z. z \), and inverse equation comes from \( \lambda z. M_{\text{app}}[z := \text{lam}] \simeq \mathbb{C} \lambda z. z \). In the same way, the commutation of the diagrams come from the following equivalences:

\[
\begin{align*}
\lambda z. M_{\text{case}}[z := \pi_1(z), M_{\text{lam}}] \simeq \mathbb{C} \lambda z. \pi_1^n((\pi_2(z))) \\
\lambda y. (M_{\text{app}}[z := M_{\text{case}}[z := \pi_1(y)])] \pi_2(y) \simeq \mathbb{C} \lambda y. M_{\text{case}}[z := \pi_1(y), M_{\text{app}}[z := \pi_2(y)]] \pi_2(y) \\
\lambda y. M_{\text{case}}[z := \pi_1(y), \pi_2(y)] \simeq \mathbb{C} \lambda y. M_{\text{case}}[z := \pi_1(y), \pi_2(y)] [z := \pi_2(y)] \\
\lambda y. M_{\text{i}} \simeq \mathbb{C} \lambda y. M_{\text{case}} [z := \pi_1(z), M_{\text{i}}] \\
\end{align*}
\]

In the commutation of (D4), the term \( M_{\bullet} \) corresponds to \( \bullet \), the pairing of the morphisms \( (\text{Id}_{D^n} \times \pi_1^n); \text{case} \):

317
$M_\bullet = \lambda z. \langle \text{case } z := \langle \pi_1(z), \pi_1^*(z) \rangle; \ldots; \text{case } z := \langle \pi_1(z), \pi_n^*(z) \rangle \rangle_n \square$

### 3.4 Non syntactical $\lambda C$-model

Although it is a priori not easy to construct a $\lambda C$-model (using Def. 3.1), some well-known categories are in fact continuation $\lambda C$-models. Indeed, every continuation model (i.e. every ccc with two objects $R$ and $C$ such that $C \cong R^C \times C$) happens to be a continuation $\lambda C$-model if we take (by definition) $S = C$ and $D = R^C$: we immediately have $S \cong D \times S$, and

$$C \cong R^C \times C \cong R^C \times (R^C \times C) \cong \ldots \cong (R^C)^n \times C = D^n \times S.$$

**Corollary 3.8** There is a sound interpretation of the $\lambda C$-calculus in any ccc with two objects $R$ and $C$ such that $C \cong R^C \times C$.

This is good news, since we know how to construct such mathematical structures. In particular, any Scott’s $D_\infty$ domain is suitable [15, Theo. 3.1].

Notice that conversely, every continuation $\lambda C$-model is a continuation model:

$$C \cong R^n \times S \cong D^n \times (D \times S) \cong (D^n \times S) \times D \cong C \times R^C.$$

By plugging this decomposition of isomorphism into the usual interpretation of the pure lambda calculus in continuation models, one actually obtains the morphisms $\text{lam}$ and $\text{app}$ as defined in Def. 3.1. Hence, using this isomorphism to transform a continuation $\lambda C$-model into a continuation model, and then interpreting pure $\lambda$-terms in it, amounts to the same as interpreting directly the $\lambda$-terms (seen as $\lambda C$-terms) in the continuation $\lambda C$-model.

### Conclusion and further work

We have shown how to construct an interpretation of the $\lambda C$-calculus in any continuation model (for instance in Scott’s domain): first use the isomorphism $C \cong R^C \times C$ to define the isomorphism $C \cong D^n \times S$ (with $S = C$ and $D = R^C$), and then use the isomorphisms in Def. 3.6 to define the morphisms $\text{lam}$, $\text{app}$, $\text{c}$, $\text{case}$ and $\text{?}$, $\text{?}$. Finally interpret the $\lambda C$-terms as in Fig. 5.

This work raises several questions. The first one concerns the interaction of the $\lambda C$-calculus with the $\lambda \mu$-calculus of Parigot [11], as a calculus corresponding to classical models. Is there a well-behaved calculus including both of them? Such a calculus would be of particular interest, since the $\lambda \mu$-calculus corresponds to classical logic, whereas pattern matching on data structures is usually associated to constructive proofs [4].

The second one is about the completeness of categorical models for the lambda calculus with constructors. Indeed, the $\lambda C$-models are complete for the $\lambda C$-calculus with no match failure (or with identification of all of them, as explained in the footnote p. 314). It is then natural to ask whether the continuation $\lambda C$-models are complete for the calculus; in other words, whether every $\lambda C$-model (in particular, the syntactic model of PErs) is equivalent to continuation $\lambda C$-model. If they are

---

6 Except the syntactical model in the category of Partial Equivalence Relations [12].
not, what would be an internal language for these categories? Maybe a kind of “$\lambda\mu_{C}$-calculus”, since continuation categories are complete for the $\lambda\mu$-calculus [7].

Last, the question of a denotational model for the typed $\lambda_{C}$-calculus is still pending. The syntax of this calculus is quite simple but the type system proposed for it [13] is not. To give a categorical definition of the data types seems especially not easy (the only denotational model for the typed calculus so far is the syntactic model of reducibilty candidates).

Acknowledgement

To Alexandre Miquel, for giving the idea of a double stack abstract machine for the $\lambda_{C}$-calculus, and for all the fruitful discussions we had on this work.

References


A Truly Concurrent Process Semantics over Multi-Pomsets of Consumable Resources

Dan Teodosiu

Paris Centre Mathematical Sciences
University Paris Diderot (Paris 7)
75205 PARIS Cedex 13, France

Abstract

This paper develops a truly concurrent semantical approach, whereby concurrency is notionally independent of nondeterminism, that allows describing the deterministically concurrent behaviour of recursive processes accessing consumable resources. The process semantics is based on the new coherently complete and prime algebraic domains of real and complex multi-pomsets. The process language that we study contains several deterministic quantitative process operators, namely a renaming, a hiding, a restriction, a serial and a parallel operator, as well as a recursion operator. The displayed deterministic structural operational machine engenders a linear and a complex operational semantics. A compositional denotational semantics is constructed, which uses a functional domain over environments of complex multi-pomsets. The robustness of the presented semantical work is established by proving that the denotational semantics is fully abstract with respect to both linear and complex operational semantics.

Keywords: process calculus, true concurrency, resource, consumption, quantification, pomset, denotational semantics, structural operational semantics, full abstraction.

1 Introduction

The seminal work of Hennessy & Plotkin [6] has shown how power domains could be employed to build semantic models that support parallel composition of processes. Supposing that a choice operator is part of the language, parallelism is simply reduced to interleaving and choice, thus introducing nondeterminism into the semantic models.

In this presentation, we follow a genuine approach to parallelism and concurrency, which avoids using choice and nondeterminism. We, thereby, rely on true concurrency, which has inspired a wealth of domain-theoretic approaches. These are mainly due to the insight that the prime event structures of Winskel [12] are domains that provide suitable models to express truly concurrent process combinations.

1 Email: dan.teodosiu@wanadoo.fr

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
However, to our knowledge, the only truly concurrent semantical approach that has actually developed an operational and matching denotational programming semantics, allowing for finitary process combinators, as well as recursion, originates in the work of Diekert & Gastin [2], Gastin & Teodosiu [5] and Gastin & Mislove [4]. The underlying denotational models are the pomsets advocated by Pratt [9] which, although being particular event structures, allow to simply express truly concurrent process combinations. Its intuition is based on the appealing interaction between processes and resources in any environment, a paradigm also driving the applied work in computer science of the last decades. One should also note the related work of Pym & Tofts [10] presenting a truly concurrent algebra and logic of processes and resources for a number of process combinators including choice.

A new stream in automata theory, as emphasized by the recent monograph on weighted automata [3], consists in attaching weights to transitions which express their cost or consumption in terms of some available resources and to extend these notions to recognized words and languages. Classical automata theory can in particular be recovered by considering unit weights, which is why the weighted view is more versatile, opening the way to new applications in engineering and economy.

The present process language combines the above truly concurrent approach on the denotational side with a weighted automata view on the operational side. To this end, the labeling of the denotational models and of the operational transition rules is quantified with the bounded or unbounded time or amount of resources being consumed. Quantification requires to replace the previously employed algebra of sets of resources, seen as vectors over the booleans, by an algebra of multi-sets of resources, seen as vectors over the extended positive reals, while taking care that relevant algebraic properties remain valid in this new setting. Furthermore, it leads to defining operators whose quantitative semantics enriches that of previously considered operators (serial, parallel), together with operators having a new, quantitative semantics (renaming, hiding, restriction, recursion).

Our semantics differs from previous approaches in several technical respects. Firstly, the representation of processes as complex multi-pomsets relies on a real part and a consumption part, which allows to trivially check the consumption commutation and continuity. Secondly, the new prefix partial order on complex multi-pomsets substantially simplifies the proofs, allowing to check the complex continuity solely for the real part. Finally, the semantics of recursion relies on the continuity with respect to two orders instead of one.

The paper is structured as follows. Section 2 recalls basic facts about partial orders. Section 3 introduces the domains of real and complex multi-pomsets. Section 4 presents the process language and defines the consumption semantics. Section 5 displays a deterministic structural operational machine, which engenders the linear and complex operational semantics. Section 6 is devoted to the compositional denotational semantics using a functional domain over environments of complex multi-pomsets. Section 7 presents the main results of congruence and full abstraction of the denotational semantics with respect to both linear and complex operational semantics.
The interested reader may find a complete version of our results in [11].

2 Partial Orders

For a detailed exposition we refer to the presentation of Abramsky & Jung [1].

A partial order (PO) is a pair \((X,\leq)\) where \(\leq\) is a reflexive, antisymmetric and transitive binary relation on \(X\). A subset \(Y \subseteq X\) is directed (coherent) iff it is non-empty and for all \(x,y \in Y\) there exists \(z \in Y\) (\(z \in X\)) such that \(x \leq z\) and \(y \leq z\). Any directed set is coherent. \((X,\leq)\) is a coherently complete PO (CCPO) (directed complete PO (DCPO)) iff every coherent subset (directed subset) has a least upper bound. Any CCPO is a DCPO.

An element \(x \in X\) is prime (compact) iff for all (directed) subsets \(Y \subseteq X\) having a least upper bound, \(x \leq \bigvee Y\) implies \(x \leq y\) for some \(y \in Y\). The set of all prime (compact) elements of \(X\) below \(y \in X\) is denoted by \(\Prm(y)\) (\(\Kmp(y)\)). A partial order \((X,\leq)\) is \(p\)-algebraic iff \(x = \bigvee \Prm(x)\) for all \(x \in X\). It is \(k\)-algebraic iff \(\Kmp(x)\) is directed and \(x = \bigvee \Kmp(x)\) for all \(x \in X\). In the case of CCPOs, \(p\)-algebraicity also implies \(k\)-algebraicity.

A \(k\)-algebraic DCPO is called a \((Scott-)domain\). A mapping \(F : (X,\leq) \rightarrow (X',\leq')\) is \((Scott-)continuous\) iff for all directed sets \(Y \subseteq X\), such that \(\bigvee Y\) exists, \(\bigvee F(Y)\) exists, and \(F(\bigvee Y) = \bigvee F(Y)\).

The set of finite ordinals (i.e. the least infinite ordinal) is denoted by \(\omega\).

3 Real and Complex Multi-Pomsets

We denote by \(\mathbb{R}_+\) the set of positive reals and by \(\mathbb{R}_+ = \mathbb{R}_+ \cup \{\infty\}\) the set of positive extended reals. The notations \(\mathbb{R}\) and \(\mathbb{C}\) are reserved for later purposes.

We fix in this section a countable set of resources \(\mathcal{R}\). The alphabet is the set of multi-sets of resources \(\mathbb{A} = \mathcal{R} \rightarrow \mathbb{R}_+\). We define \(0,\infty \in \mathbb{A}\) by \(0(\alpha) = 0\) and \(\infty(\alpha) = \infty\) for all \(\alpha \in \mathcal{R}\). The set of actions is \(\mathbb{A}_{\mathcal{R}} = \mathbb{A} \setminus \{0\}\).

The multiplicity attached by an action to a resource measures for instance the time consumed. The notion of time may just as well be replaced by amount, while the term consumed may be replaced by produced. If, for example, \(\mathcal{R} = \{A,B,C\}\) then an action consuming respectively 3.5, 5.7 and 7.3 time units of \(A\), \(B\) and \(C\) is denoted by the multi-set \(3.5A + 5.7B + 7.3C\). Concrete examples might come from computer science (an action consuming 5 processor, 10 channel and 2 memory time units) or workflow management (an action consuming 100 man, 5 tool and 10 object time units).

On \(\mathbb{A}\) we define componentwise the order \(\leq \subseteq \mathbb{A} \times \mathbb{A}\), the complement \(- : \mathbb{A} \rightarrow \mathbb{A}\), the sum \(+ : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}\), the infimum \(\wedge : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}\), the supremum \(\vee : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}\) and the skew difference \(\setminus : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}\), whereby for all \(n,m \in \mathbb{R}_+\) we set \(\overline{n} = \infty\) if \(n = 0\), \(\overline{n} = 0\) if \(n \neq 0\), \(n \setminus m = \infty\) if \(n = \infty\), \(n \setminus m = 0\) if \(n \neq \infty\) and \(m = \infty\), \(n \setminus m = (n - m) \vee 0\) if \(n,m \neq \infty\).

For \(a,b \in \mathbb{A}\) such that \(b \leq a\) let \(a - b = a \setminus b\). For \(a,b \in \mathbb{A}\) we define the independence \(a \perp b\) iff \(a \wedge b = 0\).
For \( a \in \mathbb{A} = \mathcal{R} \rightarrow \mathbb{R}_+ \) and \( S \in \mathcal{R} \times \mathcal{R} \rightarrow \mathbb{R}_+ \) we set \( S(a) = aS \) as a vector-matrix multiplication. The \textit{set of renamings} \( S_\mathcal{R} \subseteq \mathcal{R} \times \mathcal{R} \rightarrow \mathbb{R}_+ \) is defined by \( S \in S_\mathcal{R} \) iff \( (a \neq 0 \implies S(a) \neq 0 \) for all \( a \in \mathbb{A}_\mathcal{R} \) and \( (a \perp b \implies S(a) \perp S(b) \) for all \( a, b \in \mathbb{A}_\mathcal{R} \).

### 3.1 The Domain of Real Multi-Pomsets \( \mathbb{R} \)

A \textit{real multi-labelled partial order} is a triple \((E, \preceq, \rho)\), where

(i) the \textit{synchronization relation} \( \preceq \subseteq E \times E \) is a partial order on the \textit{set of events} \( E \) satisfying the \textit{past finiteness condition} that \( \{ f \in E \mid f \preceq e \} \) is finite for each \( e \in E \).

(ii) the \textit{event-labelling} \( \rho : E \rightarrow \mathbb{A}_\mathcal{R} \) satisfies the \textit{over-synchronization condition} that \( \rho(e) \wedge \rho(f) \neq 0 \implies e \preceq f \) or \( f \preceq e \) for each \( e, f \in E \).

A \textit{real multi-pomset} is the isomorphism class \([E, \preceq, \rho]\) of a real multi-labelled partial order \((E, \preceq, \rho)\). The \textit{set of real multi-pomsets} is denoted by \( \mathbb{R} \). A \textit{finite multi-pomset} is a multi-pomset whose event set is finite. The \textit{set of finite multi-pomsets} is denoted by \( \mathbb{F} \). The \textit{empty multi-pomset} is \( 0 = [([\emptyset], \emptyset, \emptyset)] \in \mathbb{R} \).

The synchronization relation reflects the temporal (or causal) order between the events of the multi-pomset. The past finiteness condition is a technical assumption that restricts the definition to real multi-pomsets, but can be relaxed if one wishes to deal with transfinite multi-pomsets.

The over-synchronization condition is equivalent to the fact that for each resource the events consuming it are \textit{sequentialized} (totally ordered). In particular it implies that the multi-pomset has no auto-concurrency, that is, for all \( a \in \mathbb{A}_\mathcal{R} \) the set \( \rho^{-1}(a) = \{ e \in E \mid \rho(e) = a \} \) is totally ordered by \( \preceq \). The \textit{number of occurrences} \( |a| : \mathbb{R} \rightarrow \omega + 1 \) of \( a \in \mathbb{A}_\mathcal{R} \) is defined for all \( a \in \mathbb{R} \) by \( |x|_a = \text{ord}(\rho^{-1}(a), \preceq \cap \rho^{-1}(a) \times \rho^{-1}(a)) \leq \omega \), that is the ordinal associated with the well-order induced by \( \preceq \) on \( \rho^{-1}(a) \).

In order to avoid cumbersome isomorphism proofs we define the \textit{standard representative} of \( x = (E, \preceq, \rho) \) as the unique isomorphic real multi-labelled partial order \( \hat{x} = (\hat{E}_x, \preceq_x, \rho_x) \), such that

(i) \( E_x = \phi_x(E) = \{ (a, n) \mid a \in \mathbb{A}_\mathcal{R} \text{ and } n < |x|_a \} \subseteq \mathbb{A}_\mathcal{R} \times \omega = \mathbb{E} \),

(ii) \( (a, n) \preceq_x (a, m) \) for all \( a \in \mathbb{A}_\mathcal{R} \) and \( n \leq m < |x|_a \),

(iii) \( \rho_x(a, n) = a \) for all \( a \in \mathbb{A}_\mathcal{R} \) and \( n < |x|_a \).

The \textit{past} in \( x \in \mathbb{R} \) of \( F \subseteq \mathbb{E} \) is \( \downarrow_x F = \{ e \in E_x \mid \exists f : e \preceq_x f \in F \} \). The \textit{restriction} of \( x \in \mathbb{R} \) to \( F \subseteq \mathbb{E} \) is \( x/F = [E_x \cap F, \preceq_x \cap F \times F, \rho_x \cap F \times \mathbb{A}_\mathcal{R}] \). The \textit{prefix} is defined for all \( x, y \in \mathbb{R} \) by \( x \preceq y \) iff \( E_x = \downarrow_y E_y \) and \( x = y/E_x \).

The next theorem shows that \((\mathbb{R}, \leq)\) is an interesting semantic domain.

**Theorem 3.1** \((\mathbb{R}, \leq)\) is a \textit{p-algebraic} CCPO, hence, it is a (Scott-)domain.
The alphabet $\text{alph} : \mathbb{R} \rightarrow P(\mathbb{A})$ is defined for all $x \in \mathbb{R}$ by $\text{alph}(x) = \{ p_x(e) \mid e \in E_x \}$. The consumption $\text{cons} : \mathbb{R} \rightarrow \mathbb{A}$ is defined for all $x \in \mathbb{R}$ by $\text{cons}(x) = \sum_{e \in E_x} \rho_x(e)$. It can be shown that $\text{cons} : (\mathbb{R}, \leq) \rightarrow (A, \leq)$ is continuous. For $x, y \in \mathbb{R}$ we define the independence $x \perp y$ iff $\text{cons}(x) \perp \text{cons}(y)$. The infinite consumption $\text{consinf} : \mathbb{R} \rightarrow \mathbb{A}$ is defined for all $x \in \mathbb{R}$ and $\alpha \in \mathbb{R}$ by $\text{consinf}(x)(\alpha) = \infty$ if $\text{cons}(x)(\alpha) = \infty$ and $\text{consinf}(x)(\alpha) = 0$ otherwise.

We need later on the following definitions. For $a \in A_R$ and $x \in \mathbb{R}$ satisfying the action prefix $a \leq x$ let the action residue be $a^{-1}x = x/(E_x \setminus \{(a, 0)\})$. For $u \in A_R$ and $x \in \mathbb{R}$ we define the linear prefix $u \leq x$ and the linear residue $u^{-1}x$ inductively on $u$ by:

- Let $0 \leq x$ and $0^{-1}x = x$,
- For $a \in A_R$ let $ua \leq x$ iff $u \leq x$, $a \leq u^{-1}x$, and let $(ua)^{-1}x = a^{-1}(u^{-1}x)$.

The set of linearizations of $r$ is $\text{Lin}(r) = \{ u \in A_R^* \mid u \leq x, u^{-1}x = 0 \}$.

We note that for any $a \in A_R \subseteq \mathbb{R}$ and $x \in \mathbb{R}$ we have $a \leq x$ iff $a \leq x$ iff $x$ contains a minimal event labelled with $a$, so in this case the prefix and linear prefix relation coincide.

The main deficiency of $(\mathbb{R}, \leq)$ is the fact that it is unsuitable to define continuous denotations for the operators of our process language, since for example the sequential composition is not monotone (hence, not continuous). This is indeed not surprising, since already the sequential composition (catenation) of strings is not monotone with respect to the prefix order, as can be easily seen on the example $a \leq a; a'$ and $b \leq b; b'$ but $a; b \not\leq (a; a'); (b; b')$ unless $a'$ is empty.

### 3.2 The Domain of Complex Multi-Pomsets $\mathbb{C}$

We surmount the above obstacle by introducing the domain of complex multi-pomsets.

A complex multi-pomset is a pair $x = (r, R)$, where $r \in \mathbb{R}$ is a real multi-pomset, and $R \in \mathbb{A}$ is a multi-set of resources, such that $\text{cons}(r) \leq R$. The set of complex multi-pomsets is denoted by $\mathbb{C}$. The multi-pomset $r$ is denoted by $\text{Re}(x)$ and called the real part of $x$. The multi-set $R$ is denoted by $\text{cons}(x)$ and called the consumption part of $x$.

The imaginary part of $x \in \mathbb{C}$ is $\text{Im}(x) = \text{cons}(x) - \text{cons}(\text{Re}(x))$. If its imaginary part $\text{Im}(x)$ is zero, the complex multi-pomset $x$ is called terminated. Note that, due to the convention regarding $\infty$ and the difference operator, we have $\text{consinf}(\text{Re}(x)) \leq \text{Im}(x)$.

The first component of a complex multi-pomset is a real multi-pomset describing the observed part of the process, while the second component is a multi-set of resources representing the quota actually consumed by the process during its execution.

The prefix is defined for all $(r, R), (s, S) \in \mathbb{C}$ by $(r, R) \leq (s, S) \leftrightarrow r \leq s$ and $R = S$. The underlying idea here is that we increase the information about a process by letting grow its observable part $r \leq s$, while preserving the quota $R = S$. 

324
that may be consumed during the execution.

The next theorem shows that \((\mathbb{C}, \leq)\) is a suitable semantic domain.

**Theorem 3.2** \((\mathbb{C}, \leq)\) is a \(p\)-algebraic CCPO, hence, it is a (Scott-)domain.

We need later on the following definitions. For \(u \in A_R^\mathbb{C}\) and \(x \in \mathbb{C}\) we extend the \textit{linear prefix} by \(u \preceq x\) iff \(u \preceq \text{Re}(x)\) and the \textit{linear residue} by \(u^{-1}x = (u^{-1}\text{Re}(x), \text{cons}(x) - \text{cons}(u))\).

The main virtue of \((\mathbb{C}, \leq)\) is the fact that it allows defining internal and continuous denotations for all process operators of our language.

## 4 The Process Language

We fix in the following a countable \textit{set of constants} \(\mathbb{C}\) and a disjoint countable \textit{set of variables} \(\mathbb{V}\). The \textit{set of resources} is \(\mathcal{R} = \mathbb{C} \cup \mathbb{V}\). The \textit{set of constant actions} is \(A_c = (\mathbb{C} \rightarrow \mathbb{N}) \setminus \{0\} \subseteq A_R\).

The \textit{language of terms} \(\mathcal{L}\) is generated by the following BNF-style grammar

\[
p ::= \text{SKIP} \mid c \mid S(p) \mid p \circ T \mid p \odot T \mid p \cdot p \mid p \parallel p \mid p \circ C \mid p \circ x \mid \text{rec}_x.p
\]

for all \(c \in A_c, S \in S_R, T \in A, C \subseteq A_R\) and \(x \in \mathbb{V}\).

Here, SKIP is the \textit{empty process}, \(c\) is a \textit{constant action}, \(S()\) is \textit{renaming} through \(S\), \(\odot T\) is \textit{hiding} of the consumption \(T\), \(\circ T\) is \textit{restriction} to the consumption \(T\), \(\cdot\) is \textit{serial composition}, \(\parallel\) is \textit{parallel composition} synchronized on the channels in \(C\), \(x \in \mathbb{V}\) is a \textit{variable} and \(\text{rec}_x\). is \textit{recursion} over \(x\).

The \textit{language of closed terms} \(\mathcal{L}_c\) is the set of terms without free variables.

The \textit{consumption} \(\text{Cons}(p) : A^\mathbb{V} \rightarrow A\) of a process term \(p \in \mathcal{L}\) is inductively defined for all \(\tau \in A^\mathbb{V}\) by

\[
\begin{align*}
\text{Cons}(\text{SKIP})(\tau) &= 0 \\
\text{Cons}(c)(\tau) &= c \\
\text{Cons}(S(p))(\tau) &= S(\text{Cons}(p)(\tau)) \\
\text{Cons}(p \circ T)(\tau) &= \text{Cons}(p)(\tau) \setminus T \\
\text{Cons}(p \odot T)(\tau) &= \text{Cons}(p)(\tau) \wedge T \\
\text{Cons}(p \cdot q)(\tau) &= \text{Cons}(p)(\tau) + \text{Cons}(q)(\tau) \\
\text{Cons}(p \parallel q)(\tau) &= \text{Cons}(p)(\tau) \lor \text{Cons}(q)(\tau) \\
\text{Cons}(x)(\tau) &= \tau(x) \\
\text{Cons}(\text{rec}_x.p)(\tau) &= \text{lfp}_< R.(\{x\} + \text{Cons}(p)(\tau[x \mapsto R]))
\end{align*}
\]

For \(\tau \in A^\mathbb{V}\) we define \(\tau[x \mapsto R]\) to be identical to \(\tau\) on all arguments except \(x\) that is assigned the value \(R\). The characteristic mapping \(\{x\} \in A\) is defined for all \(\alpha \in \mathcal{R}\) by \(\{x\}(\alpha) = 0\) for \(\alpha \neq x\) and \(\{x\}(\alpha) = 1\) for \(\alpha = x\). One may show by structural induction over \(p \in \mathcal{L}\) that \(\text{Cons}(p) : (A, \leq)^\mathbb{V} \rightarrow (A, \leq)\) is continuous. Thus, the above definition determines a compositional consumption semantics \(\text{Cons} : \mathcal{L} \rightarrow ((A, \leq)^\mathbb{V} \rightarrow (A, \leq))\).


5 The Operational Semantics

Table 1 presents the transition rules of our deterministic structural operational machine, whereby we let $c \in \mathbb{A}_C$, $T \in \mathbb{A}$, $C \subseteq \mathbb{A}_R$, $x \in \mathcal{V}$, $p,p',q,q' \in \mathcal{L}$, $a \in \mathbb{A}_R \cup \{0\}$, $\tau \in \mathbb{A}$. As usual, $p[q/x]$ denotes the term that is obtained from $p$ after substituting all occurrences of the variable $x$ by $q$. Note that recursion is modelled in an observable way, each unwinding producing as observation the variable being recursed, which may subsequently be renamed or hidden.

For $u \in \mathbb{A}_R^*$ and $p,p' \in \mathcal{L}$, it is defined on the length of $u$ by

- $p \xrightarrow{0} p'$
- $p \xrightarrow{a} p'$

For any fixed $E \subseteq \mathbb{E}$ and $p \xrightarrow{u} p'$, let the linear transition $p \xrightarrow{u} p'$ be inductively defined on the length of $u$ by

- $P \xrightarrow{0} P'$
- $P \xrightarrow{a} P'$

Using the linear transition we next define a linear and a complex operational semantics of closed process terms as follows.

The linear behaviour of $p \in \mathcal{L}_c$ is $\mathcal{A}_R^*(p) = \{ u \in \mathbb{A}_R^* | p \xrightarrow{u} \}.$

For any fixed $E \subseteq \mathbb{E}$ the intersection $\bigcap P$ of a set $P \subseteq \mathbb{R}$ such that for all $r \in P$ we have $E = E \cap P = \bigcap \{ r \mid r \in P \}$. The restriction of $p \in \mathcal{L}_c$ to $E$ is $p/E = \{ u \in \mathbb{A}_R^* | p \xrightarrow{u} \}.$

The complex behaviour of $p \in \mathcal{L}_c$ is $\mathcal{C}(p) = \{ (p/E, \text{Cons}(p)) \mid p \xrightarrow{u} \}.$

<table>
<thead>
<tr>
<th>RULE</th>
<th>DERIVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>$c \xrightarrow{c} \text{SKIP}$</td>
</tr>
<tr>
<td>REN</td>
<td>$p \xrightarrow{a} p'$</td>
</tr>
<tr>
<td>HID</td>
<td>$p \xrightarrow{a} p'$, $a \leq T$</td>
</tr>
<tr>
<td>RES</td>
<td>$\text{rec } x.p = q$</td>
</tr>
<tr>
<td>REC</td>
<td>$q \xrightarrow{(x)} p[q/x]$</td>
</tr>
<tr>
<td>SER1</td>
<td>$p \xrightarrow{a} p'$</td>
</tr>
<tr>
<td>SER2</td>
<td>$p \xrightarrow{a} q'$</td>
</tr>
<tr>
<td>PAR0</td>
<td>$p \parallel q \xrightarrow{a} p'$</td>
</tr>
<tr>
<td>PAR1</td>
<td>$p \parallel q \xrightarrow{a} p'$</td>
</tr>
<tr>
<td>PAR2</td>
<td>$p \parallel q \xrightarrow{a} p'$</td>
</tr>
</tbody>
</table>

Table 1
6 The Denotational Semantics

We next construct a denotational semantics for our process language using a functional domain over environments of complex multi-pomsets.

We endow the set of environments $\mathbb{C}^V$ with the product order $\leq$, that is, we set $(\mathbb{C}^V, \leq) = (C, \leq)^V$, $\text{cons}^V : \mathbb{C}^V \to \mathbb{A}^V$ is defined for all environments $\sigma \in \mathbb{C}^V$ and $x \in V$ by $\text{cons}^V(\sigma)(x) = \text{cons}(\sigma(x))$. We have a canonical embedding $\mathbb{A} \hookrightarrow \mathbb{C}$, $R \mapsto (0, R)$, which allows identifying $\mathbb{A}$ with its image in $\mathbb{C}$ and set $\mathbb{A} \subseteq \mathbb{C}$. Therefore, any function on $\mathbb{C}^V$ is a function on $\mathbb{A}^V$.

The functional domain $\mathbb{D}$ is the set of mappings $f : \mathbb{C}^V \to \mathbb{C}$ satisfying the following functional conditions

(i) consumption commutation: $\text{cons}(f(\sigma)) = f(\text{cons}^V(\sigma))$ for all $\sigma \in \mathbb{C}^V$,
(ii) consumption continuity: $f : (\mathbb{A}, \leq)^V \to (\mathbb{A}, \leq)$ is continuous,
(iii) complex continuity: $f : (\mathbb{C}, \leq)^V \to (\mathbb{C}, \leq)$ is continuous.

We pointwise lift the ordering $\leq$ from $\mathbb{C}$ to $\mathbb{D}$, that is, for $f, g \in \mathbb{D}$ we define $f \leq g$ iff $f(\sigma) \leq g(\sigma)$ for all $\sigma \in \mathbb{C}^V$.

**Proposition 6.1** $\mathbb{D}$ is closed by substitution and $(\mathbb{D}, \leq)$ is a DCPO.

The denotational semantics $[] : \mathcal{L} \to \mathbb{D}$ inductively defined below is uniquely determined by the denotation which will be defined in the next subsections for each finitary operator symbol as an operation on $\mathbb{C}$ having the same arity and satisfying the functional conditions of $\mathbb{D}$ (that is, we indeed have $[\text{SKIP}], [c], [S(p)], [p \odot T], \sigma \odot \tau, [p \cdot q], [p \parallel q] \in \mathbb{D}$ for $p, q \in \mathcal{L}$) and the denotation which will be defined for the infinitary recursion operator symbol as an operation on $\mathbb{D}$ (that is, we indeed have $[\text{rec } x.p] \in \mathbb{D}$ for $p \in \mathcal{L}$). Hereby, consumption commutation and continuity are straightforward to check whereas complex continuity is increasingly difficult to prove. From the above we may thus in advance state the following

**Theorem 6.2** The denotational semantics $[] : \mathcal{L} \to \mathbb{D}$ inductively defined by

\[
\begin{align*}
[\text{SKIP}] (\sigma) &= (0, 0) \\
[c] (\sigma) &= (c, c) \\
[S(p)] (\sigma) &= S([p] (\sigma)) \\
[p \odot T] (\sigma) &= [p] (\sigma) \odot T \\
[p \cdot q] (\sigma) &= [p] (\sigma) \cdot [q] (\sigma) \\
[p \parallel q] (\sigma) &= [p] (\sigma) \parallel [q] (\sigma) \\
[\text{rec } x.p] (\sigma) &= (\text{rec } x.[p]) (\sigma) \\
\end{align*}
\]

is well-defined, that is $[p] \in \mathbb{D}$ for all $p \in \mathcal{L}$.

Directly on the denotation of each of the operators SKIP, $c$, $S(\cdot)$, $\odot T$, $\odot T$, $\cdot$, $\parallel$ and rec $x$. we will also be able to inductively check the following

**Proposition 6.3** If $p \in \mathcal{L}$, $\sigma \in \mathbb{C}^V$ then $\text{cons}([p] (\sigma)) = \text{Cons}(p)(\text{cons}^V(\sigma))$.

We now proceed with the denotation of the operators of our language.
6.1 The Renaming Operator $S(x)$

Renaming amounts to a simple relabeling which preserves the events and their initial ordering. Simple as it may seem it nevertheless allows linear computations to take place on the labels of the events and may, therefore, be used to derive further operators from those of our language.

For every $S \in S_R$ the renaming operator $S() : R \rightarrow R$ is defined by $S([E, \preceq, \rho]) = [E, \preceq, S(\rho)]$, whereby $S(\rho)(e) = S(\rho(e))$ for all $e \in E$. For every $S \in S_R$ the renaming operator $S() : C \rightarrow C$ is defined for all $x \in C$ by $S(x) = (S(Re(x)), S(cons(x)))$.

6.2 The Hiding Operator $x \otimes T$

The hiding operator allows to internalize some given quota of resources and prevents other processes from synchronizing on events that make use of them. As usually, this expresses the need for local, as opposed to global, computation and communication.

For every $T \in A$ the hiding operator $\otimes T : R \rightarrow R$ is defined for all $[E, \preceq, \rho] \in R$ by $[E, \preceq, \rho] \otimes T = [E', \preceq', \rho']$ where

(i) $\rho'(e) = \rho(e) \setminus (T \setminus \sum_{f \preceq e} \rho'(f))$,  
(ii) $E' = \{ e \in E | \rho'(e) \neq 0 \}$,  
(iii) $\preceq' = \preceq \cap E' \times E'$.

For every $T \in A$ the hiding operator $\otimes T : C \rightarrow C$ is defined for all $x \in C$ by $x \otimes T = (Re(x) \otimes T, cons(x) \setminus T)$.

The hiding $x \otimes T$ erases a given additive consumption $T$ out of the past of each event of $x$, thereby rendering it unobservable.

6.3 The Restriction Operator $x \oslash T$

The restriction operator blocks a process on all but some given quota of resources. This is used for instance in order to assure confinement of that process to a certain safe environment and may be useful in security protocols.

For every $T \in A$ the restriction operator $\oslash T : R \rightarrow R$ is defined for all $x \in R$ by $x \oslash T = \bigvee \{ y \in R | y \leq x, cons(y) \leq T \}$. For every $T \in A$ the restriction operator $\oslash T : C \rightarrow C$ is defined for all $x \in C$ by $x \oslash T = (Re(x) \oslash T, cons(x) \wedge T)$.

One can easily see that $x \oslash T$ is the restriction of $x$ to the set of events having a past that additively consumes resources below $T$, that is, we have $x \oslash T = x/E'_x$ where $E'_x = \{ e \in E_x | \sum_{f \preceq e} \rho_x(f) \leq T \}$.

6.4 The Serial Composition $x \cdot y$

The following presentation of the serial composition is a generalization of the concatenation treated in [5]. The serial composition enforces synchronizations between the first and the second process only to prevent races for resources. Events at the end of the first and events at the beginning of the second process can thus occur
concurrently if they are independent. This construct may be of interest in automatic code parallelization and in transactional systems.

The sequential composition \( \cdot : \mathbb{R}^2 \to \mathbb{R} \) is defined for all \( x_1 = [E_1, \preceq_1, \rho_1] \in \mathbb{R} \) and \( x_2 = [E_2, \preceq_2, \rho_2] \in \mathbb{R} \) if \( \text{consinf}(x_1) \land \text{cons}(x_2) = 0 \) by \( x_1 \cdot x_2 = [E, \preceq, \rho] \), where

(i) \( E = E_1 \cup E_2 \),

(ii) \( \preceq = (\preceq_1 \cup \{ (e_1, e_2) \in E_1 \times E_2 | \rho_1(e_1) \land \rho_2(e_2) \neq 0 \}) \cup \preceq_2^* \),

(iii) \( \rho = \rho_1 \cup \rho_2 \).

The serial composition \( \cdot : \mathbb{C}^2 \to \mathbb{C} \) is defined for all \( x, y \in \mathbb{C} \) by \( x \cdot y = (\text{Re}(x) \cdot (\text{Re}(y) \circ \text{Im}(x)), \text{cons}(x) + \text{cons}(y)) \).

It can be shown that using the serial composition together with hidings enables us to denote all compact multi-pomsets by closed terms of the process language, which means that the denotational semantics is optimal.

### 6.5 The Sequential Composition \( x ; y \)

The sequential composition should be employed whenever there is a need to temporally completely synchronize two processes. The compound processes are scheduled such that the entire first process occurs before the entire second process, hence they are temporally ordered, even if independent.

The sequential composition \( ; : \mathbb{R}^2 \to \mathbb{R} \) is defined for all \( x_1 = [E_1, \preceq_1, \rho_1] \in \mathbb{R} \) and \( x_2 = [E_2, \preceq_2, \rho_2] \in \mathbb{R} \) by \( x_1 ; x_2 = [E_1 \cup E_2, \preceq_1 \cup E_1 \times E_2 \cup \preceq_2, \rho_1 \cup \rho_2] \). The sequential composition \( ; : \mathbb{C}^2 \to \mathbb{C} \) is defined for all \( x, y \in \mathbb{C} \) by

\[
\begin{align*}
  x ; y = \begin{cases} 
    (\text{Re}(x) ; \text{Re}(y), \text{cons}(x) + \text{cons}(y)) & \text{if } \text{Im}(x) = 0 \\
    (\text{Re}(x), \text{cons}(x) + \text{cons}(y)) & \text{otherwise}.
  \end{cases}
\end{align*}
\]

It can be shown that the sequential composition can be expressed as a renaming of the serial composition of renamings of the compound processes, hence it is a derived operator of our process language. We shall later on essentially use the sequential composition in order to define the denotational semantics of recursion.

### 6.6 The Parallel Composition \( x \parallel y \)

The following presentation of the parallel composition is a generalization of the one treated in [4]. The parallel composition is indexed by a set of channels that processes are supposed to employ in order to synchronize. Events accessing the channels are commonly processed by the compound processes, while events which make no use of the channels may be independently processed by each compound process. This construct may in particular be used to model data-parallel programs running on PRAMs.

Let \( x_1 = [E_1, \preceq_1, \rho_1] \), \( x_2 = [E_2, \preceq_2, \rho_2] \in \mathbb{R} \) be in standard representation. We define their parallel composition by \( x_1 \parallel x_2 = [E_1 \cup E_2, (\preceq_1 \cup \preceq_2)^*, \rho_1 \cup \rho_2] \). Note that \( x_1 \parallel x_2 \) may fail to be a real multi-pomset for two different reasons. First, \( \preceq \) may fail to be antisymmetric. This is the case for instance if \( x_1 = a ; b \) and \( x_2 = b ; a \). Second, \( \preceq \) may fail to be over-synchronized. This is the case for instance if \( x_1 = a \).
and \( x_2 = b \) with \( \neg(a \perp b) \).

Let \( x_1, x_2 \in \mathbb{C} \) and \( C \subseteq A_R \). We define \((r_1, r_2) \in \mathbb{R}_C(x_1, x_2) \subseteq \mathbb{R}^2 \) iff

(i) for all \( i \in \{1, 2\} \) we have \( r_i \leq \text{Re}(x_i) \),
(ii) \( |r_1|_a = |r_2|_a \) for all \( a \in C \),
(iii) for all \( \{i, j\} = \{1, 2\} \) and \( a \in \text{alph}(r_i) \) we have \( a \in C \) or \((a \perp C \) and \( a \perp x_j)\),
(iv) \( r_1 \parallel r_2 \in \mathbb{R} \).

The parallel composition \( \parallel : \mathbb{C}^2 \to \mathbb{C} \) is defined for all \( x_1, x_2 \in \mathbb{C} \) and \( C \subseteq A_R \) by \( x_1 \parallel x_2 = (r_1 \parallel r_2, \text{cons}(x_1) \lor \text{cons}(x_2)) \) where \((r_1, r_2) = \bigvee \mathbb{R}_C(x_1, x_2) \).

### 6.7 The Recursion Operator rec \( .f \)

Since the denotation of the recursion operator essentially differs from the least fixed point semantics, we shall explain in some detail how it is defined.

For \( \sigma \in A^V \) we define as usual \( \sigma[x \mapsto y] \) to be identical to \( \sigma \) on all arguments except \( x \), which is assigned the value \( y \).

For \( x \in V \) we define \( \Phi_x : D \times C^V \times C \to C \) by \( \Phi_x(f, \sigma, y) = \{x\} ; f(\sigma[x \mapsto y]) \), and \( \Psi_x : D \times C^V \times A \to A \) by \( \Psi_x(f, \sigma, R) = \Phi_x(\text{cons} \circ f, \sigma, R) = \{x\} ; \text{cons}(f(\sigma[x \mapsto R])) = \{x\} + f(\text{cons}^V(\sigma)[x \mapsto R]) = \Phi_x(f, \text{cons}^V(\sigma), R) \).

\( \Phi_x : (D, \leq) \times (C, \leq)^V \times (A, \leq) \to (C, \leq) \) can be easily shown to be continuous for each \( x \in V \). Since the mapping \( f \) satisfies the consumption continuity of \( D \), it follows that the mapping \( \Psi_x : D \times C^V \times (A, \leq) \to (A, \leq) \) is continuous in the last argument. Therefore, the least fixed point in this argument exists and may be computed for instance by means of \( \text{lfp}_{\leq} R, \Psi_x(f, \sigma, R) = \bigvee_{n<\omega} R_n \) where \( R_0 = 0 \) and \( R_{n+1} = \{x\} + f(\text{cons}^V(\sigma)[x \mapsto R_n]) \) for all \( n < \omega \). Hence, for each \( x \in V \) the mappings \( x_n : D \times C^V \to C \) for \( n < \omega \) may be defined by

\[
\begin{align*}
x_0(f, \sigma) &= \text{lfp}_{\leq} R, \Psi_x(f, \sigma, R) = \text{lfp}_{\leq} R . (\{x\} + f(\text{cons}^V(\sigma)[x \mapsto R])) \\
x_{n+1}(f, \sigma) &= \Phi_x(f, \sigma, x_n(f, \sigma)) = \{x\} ; f(\sigma[x \mapsto x_n(f, \sigma)])
\end{align*}
\]

for all \( f \in D \) and \( \sigma \in C^V \). The sequence of continuous mappings \( x_n : D \times C^V \to C \) is pointwise increasing and \((C, \leq) \) is a CCPO, hence, we may compute its pointwise supremum and define the recursion operator \( \text{rec}.x.f. : D \to D \) for all \( f \in D \) and \( \sigma \in C^V \) by \( (\text{rec}.x.f.)(\sigma) = \bigvee_{n<\omega} x_n(f, \sigma) \).

### 7 Congruence and Full Abstraction

We now arrive at the main results of the paper which require generalizing the arguments and constructions presented in [4].

We shall state in this section two results of full abstraction, a linear and a complex one. To this purpose we first exhibit linear translations to and from between linear transition on the operational side and linear prefix and residue on the denotational side.
The following hard technical lemma concerns the interaction of the operators with the action residue. Note the close resemblance of the denotational Table 2 to the operational Table 1. For all \( a \in \mathbb{A}_R, x, x' \in \mathbb{C} \) if \( a^{-1}x = x' \) then we suppose in particular that \( a \preceq x \).

**Lemma 7.1** For any \( a \in \mathbb{A}_R, x, x', y, y' \in \mathbb{C}, c \in \mathbb{A}_C, T \in \mathbb{A}, C \subseteq \mathbb{A}_R, f \in \mathbb{D}, \sigma \in \mathbb{C}^V \) we have the properties of Table 2.

<table>
<thead>
<tr>
<th>ACT</th>
<th>( c^{-1}c() = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>REN</td>
<td>( a^{-1}x = x' )</td>
</tr>
<tr>
<td>HID</td>
<td>( S(a)^{-1}S(x) = S(x') )</td>
</tr>
<tr>
<td>RES</td>
<td>( a^{-1}x = x', a \leq T )</td>
</tr>
<tr>
<td>REC</td>
<td>( {x}^{-1}y = f(\sigma[x \mapsto y]) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SER1</th>
<th>( a^{-1}x = x' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER2</td>
<td>( a^{-1}(x \cdot y) = x' \cdot y' )</td>
</tr>
<tr>
<td>PAR0</td>
<td>( a \in C, a^{-1}x = x', a^{-1}y = y' )</td>
</tr>
<tr>
<td>PAR1</td>
<td>( a^{-1}(x</td>
</tr>
<tr>
<td>PAR2</td>
<td>( a^{-1}(x</td>
</tr>
</tbody>
</table>

Table 2

The next proposition, which is easy to prove relying on proposition 6.3 and the previous lemma, allows us to translate linear transition on the operational side to linear residue on the denotational side. For all \( u \in \mathbb{A}_R^*, x, x' \in \mathbb{C} \) if \( u^{-1}x = x' \) then we suppose in particular \( u \preceq x \).

**Proposition 7.2** Let \( u \in \mathbb{A}_R^*, p, p' \in \mathbb{L}, \sigma \in \mathbb{C}^V \) and \( \tau = \text{cons}^V(\sigma) \). Then

\[
p \xrightarrow[\tau]{u} p' \implies u^{-1}[p](\sigma) = [p'](\sigma)
\]

The following easy technical lemma concerns the interaction of the operators with the action prefix.

**Lemma 7.3** For any \( a \in \mathbb{A}_R, x, y \in \mathbb{C}, c \in \mathbb{A}_C, T \in \mathbb{A}, C \subseteq \mathbb{A}_R, f \in \mathbb{D}, \sigma \in \mathbb{C}^V \) we have the properties of Table 3.

The next proposition, which is difficult to prove relying on proposition 6.3 and the previous lemma, allows us to translate linear prefix on the denotational side to linear transition on the operational side. The main difficulty resides in adequately treating the hiding operator which is the most difficult case. We only obtain the translation for a large subclass of terms that we call nice which do not contain hiding terms that hide variables of open subterms, a condition that is rather sensible to assume for any practical purposes.
\[ a \sqsubseteq c \Rightarrow a = c. \]

\[ a \sqsubseteq S(x) \Rightarrow b \sqsubseteq x \text{ for some } b \in A_R \text{ such that } a = S(b). \]

\[ a \sqsubseteq x \otimes T \Rightarrow u \sqsubseteq ub \sqsubseteq x \text{ for some } u \in A_R^* \text{ and } b \in A_R \text{ such that } u \otimes T = 0 \text{ and } (ub) \otimes T = a \]

\[ a \sqsubseteq x \otimes T \Rightarrow a \sqsubseteq x \text{ and } a \leq T. \]

\[ a \sqsubseteq x \cdot y \Rightarrow (a \sqsubseteq x) \text{ or } \text{(cons}(x) \perp a \text{ and } a \sqsubseteq y). \]

\[ a \sqsubseteq x \parallel y \Rightarrow (a \in C \text{ and } a \sqsubseteq x \text{ and } a \sqsubseteq y) \text{ or } (C, \text{cons}(y) \perp a \text{ and } a \sqsubseteq x) \text{ or } (C, \text{cons}(x) \perp a \text{ and } a \sqsubseteq y). \]

\[ a \sqsubseteq (\text{rec } x.f)(\sigma) \Rightarrow a = \{x\}. \]

---

Let \( \preceq \) denote the subterm ordering in \( L \). \( p \in L \) is a nice term iff for all subterms \( p_1 \otimes T \preceq p \) we have \( T \land V = 0 \) or \( p_1 \in L_c \). The set of nice terms is denoted by \( L_n \), the set of nice and closed terms is denoted by \( L_c,n \).

**Proposition 7.4** Let \( u \in A_R^* \), \( p \in L_n \), \( \tau \in A^V \). Then

\[ u \sqsubseteq [p](\tau) \Rightarrow p \xrightarrow{u \tau} \]

Using the linear translations we are able to state a denotational characterization of the linear behaviour, which observes only strings of actions.

**Theorem 7.5 (Linear Congruence)** For all \( p \in L_{c,n} \) we have

\[ A_R^*(p) = \{ u \in A_R^* \mid u \sqsubseteq [p] \} \]

As a main result, we next infer a linear full abstraction by probing processes terms in suitable hiding contexts. A context \( C(\_\_) \) is a term \( C \in L \) with one distinguished variable denoted by \( \_ \_ \). A context \( C(\_\_) \) is nice-preserving iff \( C(p) \in L_n \) whenever \( p \in L_{c,n} \).

**Theorem 7.6 (Linear Full Abstraction)** For all \( p,q \in L_{c,n} \) we have

\[ [p] = [q] \iff \text{for all nice-preserving } C(\_\_) \text{ we have } A_R^*(C(p)) = A_R^*(C(q)) \]

Using the fact that each multi-pomset \( x \in R \) is the intersection of its set of linearizations \( \text{Lin}(x) \subseteq A_R^* \), we derive a denotational characterization of the complex behaviour, which observes multi-pomsets of actions.

**Theorem 7.7 (Complex Congruence)** For all \( p \in L_{c,n} \) we have

\[ C(p) = \text{Kmp}([p]) \text{ and } [p] = \bigvee C(p) \]
This allows us to finally infer a complex full abstraction result.

**Theorem 7.8 (Complex Full Abstraction)** For all $p, q \in \mathcal{L}_{c,n}$ we have

$$[p] = [q] \iff \text{for all nice-preserving } C(\cdot) \text{ we have } C(C(p)) = C(C(q))$$

We conclude the presentation with a result relating the denotational semantics to bisimilarity and the consumption semantics.

A relation $S \subseteq \mathcal{L}_{c,n} \times \mathcal{L}_{c,n}$ is a *bisimulation* iff for all $u \in A^*_R$ we have

- $p S q$ and $q \xrightarrow{u} q'$ then $p \xrightarrow{u} p'$ and $p' S q'$ for some $p'$,
- $p S q$ and $p \xrightarrow{u} p'$ then $q \xrightarrow{u} q'$ and $p' S q'$ for some $q'$.

The coarsest bisimulation $\approx = \bigcup \{ S \mid S \text{ bisimulation} \}$, called bisimilarity, is known to be an equivalence relation (see for example [8]).

**Theorem 7.9** For all $p, q \in \mathcal{L}_{c,n}$ we have

$$[p] = [q] \iff \text{Cons}(p) = \text{Cons}(q) \text{ and } p \approx q.$$ 

### 8 Conclusion

We developed a truly concurrent semantics that allows describing the concurrent behaviour of recursive processes accessing consumable resources. We first presented the coherently complete and prime algebraic ground domains of real and complex multi-pomsets. The modelled process language contains several deterministic quantitative process operators as well as a recursion operator. Next, we displayed a deterministic structural operational machine that is straightforward to comprehend and allows extracting a linear and a complex behaviour. We then constructed a compositional denotational semantics using a functional domain over environments of complex multi-pomsets. The main results have finally shown that the denotational semantics is fully abstract with respect to both linear and complex operational semantics.

The only operator customary in classical process languages [7,8] that has been intentionally left out in ours is the *non-deterministic choice*. We think that using power-domains over complex multi-pomsets provides a clear way of handling choice at the expense of rendering the domain-theoretic tools more involved. Another possibility to achieve the same result could reside in enriching real multi-pomsets with a further relation on events expressing conflict of choices, thus imposing a modelling view closer to event structures.

Apart from the intended applications in engineering and economy the presented language can be also employed as a powerful formalism to abstractly specify and handle labelled partial orders which are far more complex than the series-parallel constructions usually considered in the literature.
Acknowledgement

I thank Paul Gastin for several fruitful discussions, Volker Diekert for pointing out interesting extensions, Daniele Varacca for constructive criticism, and Pierre-Louis Curien for various pertinent remarks.

References


A Characterisation of Expressivity for Coalgebraic Bisimulation and Simulation

Toby Wilkinson\textsuperscript{1,2}

University of Southampton, UK

Abstract

In a previous paper we introduced internal models for coalgebraic modal logics and showed how they characterise expressivity for bisimulation. Here we extend this work by enriching over preordered sets, and in so doing derive a characterisation that subsumes expressivity for both bisimulation and simulation.

Keywords: Enriched logical connection; Preordered set; Coalgebra; Bisimulation; Simulation.

1 Introduction

In [15] an abstract, category theoretic, characterisation of expressivity for coalgebraic bisimulation was given. This characterisation is in terms of internal models for a modal logic, and these were also introduced in [15]. In this paper we extend that work to include both bisimulation and simulation.

Our new characterisation is achieved by enriching over the preordered sets of [13], and thus the carrier of each coalgebra carries a preorder relation. It is this relation that gives our notion of simulation. This idea was first proposed in [6,7], where the authors enriched over \textbf{Pos} (partial orders), but we generalise this to incorporate bisimulation by noting (as in [13]) that sets can be viewed as preordered sets with the discrete preorder - i.e. \textbf{Set} \cong \textbf{DiscSetoid} (discrete setoids).

In fact we go further than this, and observe that by enriching over \textbf{Preord} (preorders), \textbf{Pos}, \textbf{Setoid} (setoids), and \textbf{DiscSetoid}, that we can characterise simulation, simulation where mutual simulation is bisimulation, mutual simulation, and bisimulation respectively.

\textsuperscript{1} Research supported by an EPSRC Doctoral Training Account.
\textsuperscript{2} Email: stw008r@ecs.soton.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science

URL: www.elsevier.com/locate/entcs
The notion of expressivity for a coalgebraic modal logic with respect to bisimulation states that, two states are logically equivalent if and only if they are behaviourally equivalent. Here logically equivalent means “have the same theory”, and behaviourally equivalent means “can be identified in a model”, where the identification is by means of coalgebra homomorphisms. Thus there is an implicit reliance on the equality relation associated with the set of all theories, and the equality relation associated with the carrier of each coalgebra. It is this that we generalise, and instead use the preorder relations that these objects now carry.

For brevity we summarise the important ideas from [15] that we need, and it should be consulted for further details.

A general outline of this paper is as follows. In Section 2 we give an overview of the framework in which we work. Then in Section 3 we define the preordered sets. In Section 4 we recall from [15] the definitions and results we need, and in Section 5 define our generalised notion of expressivity. Section 6 contains our main characterisation result, and finally in Section 7 we work through the well known expressivity result for simulation of finite branching labelled transition systems as an example.

2 Dual-Adjunction Framework

Increasingly, the standard approach to coalgebraic modal logic is to formulate it in a dual-adjunction framework [10,8,5]. In [12] this is extended to an enriched setting, where the enrichment is over a symmetric monoidal closed category that is complete and cocomplete. It this enriched version that we use, and in the remainder of this paper all categories, functors, etc. should be assumed to be enriched.

The framework consists of two categories \( A \) and \( X \), and two contravariant functors \( P \) and \( S \) that form a dual adjunction i.e. there exists a natural isomorphism

\[
\Phi: A(-1, P(-2)) \Rightarrow X(-2, S(-1))
\]

Such a dual-adjunction is often referred to as a logical connection [14], and we denote the unit and counit by

\[
\rho: id_A \Rightarrow PS \\
\sigma: id_X \Rightarrow SP
\]

The category \( X \) represents a collection of state spaces, and a collection of generalised transition systems is defined on these state spaces as coalgebras for an endofunctor \( T \). Similarly, the category \( A \) represents a collection of base logics to which modal operators are to be added. These are introduced via an endofunctor \( L \), and the corresponding modal logics are the \( L \)-algebras. The semantics of
these modal logics is given in two stages. First the dual adjunction gives a semantics for the base logics in terms of the state spaces, and then secondly, a natural transformation

$$\delta : L P \Rightarrow P T$$

gives the semantics of the modal operators in terms of the transition structures introduced by $T$ [9,11].

The literature contains many examples for the unenriched case, for example see [5], and in Example 3.4 we will develop an enriched example.

## 3 Enrichment over Preordered Sets

Recall the category $\text{Preord}$ of preordered sets and monotone functions, the objects of which are pairs consisting of a set, and a preorder relation on that set. Similarly, the categories $\text{Pos}$ (partially ordered sets), $\text{Setoid}$ (setoids), and $\text{DiscSetoid}$ (discrete setoids) have for objects pairs consisting of a set and respectively, a partial order, equivalence relation, or the equality relation, on that set. In [13] these examples are collectively known as the preordered sets, and they have significance for the coalgebraic analysis of simulation, which we shall come back to later.

We can consider these examples together by means of the following definition, where by a relation of “type $R$” we mean either a preorder, partial order, equivalence relation, or equality. The type is fixed, and every object in the category $\text{Set}_R$ must have a relation of that type.

**Definition 3.1** The category $\text{Set}_R$ has for objects pairs $(X, R_X)$ consisting of a set $X$, and $R_X$ a binary relation of type $R$ on $X$. The morphisms are the $R$-preserving functions i.e. $f : (X, R_X) \to (Y, R_Y)$ is a morphism if and only if for all $x, x' \in X$

$$x R_X x' \Rightarrow f(x) R_Y f(x')$$

To be explicit we have the following four cases:

(i) If $R$ is the type preorder, then $\text{Set}_R$ is $\text{Preord}$.

(ii) If $R$ is the type partial order, then $\text{Set}_R$ is $\text{Pos}$.

(iii) If $R$ is the type equivalence relation, then $\text{Set}_R$ is $\text{Setoid}$.

(iv) If $R$ is the type equality, then $\text{Set}_R$ is $\text{DiscSetoid}$ (which is obviously isomorphic to $\text{Set}$).

It is easy to verify that the forgetful from $\text{Set}_R$ to $\text{Set}$ creates limits and colimits - the product of $(X, R_X)$ and $(Y, R_Y)$ is given by $(X \times Y, R_{X \times Y})$, where $(x, y) R_{X \times Y} (x', y') \iff x R_X x'$ and $y R_Y y'$, and the final object is $(1, R_1)$, where $1$ is the singleton set, and $R_1 = 1 \times 1$.

It is also easy to verify that binary product and the final object form the tensor and unit of a symmetric monoidal category. To make $\text{Set}_R$ also closed we need internal hom objects $[\cdot, (X, R_X), (Y, R_Y)]$ such that $[(Y, -), \_]$ is right adjoint to $- \times (Y, R_Y)$. The obvious definition for $[(X, R_X), (Y, R_Y)]$ is the set of all $R$-preserving

337
functions from $X$ to $Y$ carrying the relation

$$fR_{[(X,R_X),(Y,R_Y)]}g \iff \forall x \in X \ f(x)R_Y g(x)$$

Further, we can define the unit of the adjunction

$$\eta_{(X,R_X)}: (X, R_X) \to [(Y, R_Y), (X, R_X) \times (Y, R_Y)]$$

by $\eta_{(X,R_X)}(x) = f_x: (Y, R_Y) \to (X, R_X) \times (Y, R_Y)$, where $f_x(y) = (x, y)$, and the counit of the adjunction

$$\varepsilon_{(Z,R_Z)}: [(Y, R_Y), (Z, R_Z)] \times (Y, R_Y) \to (Z, R_Z)$$

by $\varepsilon_{(Z,R_Z)}(g, y) = g(y)$. Thus we have the following proposition.

**Proposition 3.2** The category $\text{Set}_R$ is cartesian closed.

For the rest of this paper we make the following assumptions.

**Assumption 1** The categories $\mathbb{A}$ and $\mathbb{X}$ are enriched over $\text{Set}_R$ for some fixed type $R$ of relations. Further, the categories $\mathbb{A}$ and $\mathbb{X}$ are concrete categories i.e. the objects are sets with some additional structure and carry a relation of type $R$, and the morphisms have underlying $R$-preserving functions.

We do this for two reasons. Firstly enriching over $\text{Set}_R$ is an obvious generalisation of the approach of [6,7], and secondly to investigate expressivity we need to access individual states of objects in $\mathbb{X}$.

**Example 3.3** In the case where $R$ is the type equality, then enrichment over $\text{Set}_R$ is just ordinary category theory, and so we have all the examples of logical connections from the literature, see for example [5].

The following example is our leading example, and will feature prominently when we discuss expressivity.

**Example 3.4** There is a logical connection enriched over $\text{Set}_R$ between $\text{MSL}$ (meet semilattices with top) and $\text{Set}_R$ itself. To see this we need to observe that the objects of $\text{MSL}$ come with two built-in preorders. The first is the well known partial order defined by $x \leq y \iff x = x \wedge y$, and the second is equality. In what follows, if type $R$ represents preorders or partial orders, then objects of $\text{MSL}$ should be considered as having the standard partial order, and if $R$ represents equivalence relations or equality, then they should be considered as carrying the equality relation. The functor $P: \text{Set}_R \to \text{MSL}$ sends an object $(X, R_X)$ to the meet semilattice of its right $R$-closed subsets. A subset $U \subseteq X$ is right $R$-closed if $x \in U$ and $xR_X y$ implies $y \in U$. $P(X, R_X)$ is either ordered by inclusion or equality, depending upon the type $R$. The functor $S: \text{MSL} \to \text{Set}_R$ sends a meet semilattice $A$ to the set of its filters, again either ordered by inclusion or equality depending upon the type $R$. 
4 Models, Internal Models, and $R$-Models

In [15] we introduced the concepts of models and internal models for an $L$-algebra. We summarise these definitions below, and extend them to the case of enrichment over $\text{Set}_R$.

**Definition 4.1 [Models]** For an $L$-algebra $(A, \alpha)$ we define $\text{Mod}(A, \alpha)$, the category of models for $(A, \alpha)$, with objects given by pairs

$$(X, \gamma), f : X \to S(A)$$

where $(X, \gamma)$ is a $T$-coalgebra, and $f$ is a morphism such that

$$
\begin{array}{ccc}
X & \xrightarrow{f} & S(A) \\
\downarrow{\gamma} & & \downarrow{S(\alpha)} \\
T(X) & \xrightarrow{\delta_*} & TS(A)
\end{array}
$$

commutes. Such an $f$ is called a **theory map**. Here $\delta^* : TS \Rightarrow SL$ is defined as

$$\delta^* = SL\rho \circ \delta^* S$$

where $\rho$ is the unit of the logical connection, and $\delta^*$ is the adjunct of $\delta$ under the logical connection. The morphisms of $\text{Mod}(A, \alpha)$

$$g : ((X_1, \gamma_1), f_1) \to ((X_2, \gamma_2), f_2)$$

are given by $T$-coalgebra morphisms $g : (X_1, \gamma_1) \to (X_2, \gamma_2)$ such that $f_1 = f_2 \circ g$.

Like the categories $\text{Alg}(L)$ and $\text{CoAlg}(T)$, the category $\text{Mod}(A, \alpha)$ is enriched over $\text{Set}_R$, and the relation (of type $R$) on each homobject is given be the corresponding relation on the underlying homobject in $\text{Set}_R$.

**Definition 4.2 [Internal Models]** Given a class $M$ of monomorphisms in $X$, we define the category $\text{IntMod}_M(A, \alpha)$ to be the full subcategory of $\text{Mod}(A, \alpha)$ where the theory maps are in $M$, and write

$$G : \text{IntMod}_M(A, \alpha) \to \text{Mod}(A, \alpha)$$

for the corresponding inclusion functor.

We parameterise by the class $M$, as sometimes we require the morphisms of $M$ to have additional properties, for example, that the members of $M$ are preserved by $T$. In [15] this was exploited for the Giry functor which does not preserve all monomorphisms, but does preserve a particular subclass of them.
In [15] internal models were used to characterise expressivity for bisimulation. As we shall see in Section 6, to characterise simulation we shall need the following generalised definition.

**Definition 4.3** [R-Models] The category $\text{R–Mod}(A, \alpha)$ is the full subcategory of $\text{Mod}(A, \alpha)$ where the theory maps are $R$-reflecting. A function $f: X \to Y$ is $R$-reflecting if for all $x, y \in X$ if $f(x) R_Y f(y)$ then $x R_X y$. We write

$$H: \text{R–Mod}(A, \alpha) \to \text{Mod}(A, \alpha)$$

for the corresponding inclusion functor.

It should be noted that if the monomorphisms of the class $M$ are also $R$-reflecting, then the morphisms of $M$ correspond to embeddings of the type $R$ relations, and $\text{IntMod}_M(A, \alpha)$ is a full subcategory of $\text{R–Mod}(A, \alpha)$.

The following definition is very important for our characterisation of expressivity, and we shall make extensive use of it.

**Definition 4.4** We say a model $X$ in $\text{Mod}(A, \alpha)$ factors via the internal model $I$ in $\text{IntMod}_M(A, \alpha)$ if there exists a morphism $g: X \to G(I)$ in $\text{Mod}(A, \alpha)$. Similarly, $X$ factors via the $R$-model $J$ in $\text{R–Mod}(A, \alpha)$ if there exists a morphism $h: X \to H(J)$ in $\text{Mod}(A, \alpha)$.

In future sections we will also need colimits of models. Because we are working in an enriched setting the general case is that of weighted colimits, but we shall only need conical colimits i.e. those that correspond to colimits in the ordinary categorical sense. However, we must not forget that homobjects of mediating morphisms and cocones (for a fixed diagram and object) must be isomorphic as objects in $\text{Set}_R$, not just as sets.

The following two theorems follow immediately from the corresponding results of [15]. This is because created colimits are preserved, and the relation of type $R$ on the set of cocones from a diagram to an object, corresponds to the underlying one in $\text{Set}_R$.

**Theorem 4.5** The forgetful functor $U: \text{CoAlg}(T) \to X$ creates small conical colimits.

**Theorem 4.6** The forgetful functor $U: \text{Mod}(A, \alpha) \to X$ creates small conical colimits.

### 5 Expressivity

To develop a notion of expressivity that covers both bisimulation and simulation, we first need to extend the notions of logical equivalence and behavioural equivalence.

**Definition 5.1** Given two models $X_1, X_2$ in $\text{Mod}(A, \alpha)$, and states $x_1 \in X_1, x_2 \in X_2$, we say $x_1$ and $x_2$ are logically $R$-related if

$$f_1(x_1) R_{S(A)} f_2(x_2)$$
where $f_1$ and $f_2$ are the theory maps of $X_1$ and $X_2$ respectively.

**Definition 5.2** Given two models $X_1, X_2$ in $\text{Mod}(A, \alpha)$, and states $x_1 \in X_1, x_2 \in X_2$, we say $x_1$ and $x_2$ are **behaviourally** $R$-related if there exists in $\text{Mod}(A, \alpha)$ a cospan

$$X_1 \xrightarrow{f_1} X_3 \xleftarrow{f_2} X_2$$

such that $f_1(x_1) R X_3 f_2(x_2)$.

To see that these are the correct definitions we first consider the case where the type $R$ is equality. In this case logically $R$-related simply becomes equality of theories, as expected. For the definition of behaviourally $R$-related we see that the forgetful functor from $\text{Mod}(A, \alpha)$ to $\text{CoAlg}(T)$ yields the usual definition of behavioural equivalence as a cospan in $\text{CoAlg}(T)$ [9], but in addition, the forgetful functor to $X$ yields a condition that the theory maps are compatible. This is because we are working with arbitrary $L$-algebras, and not just the initial $L$-algebra, and is similar to the definition of bisimulation in [2]. Thus when the type $R$ represents equality we have, as in [15], the case of bisimulation.

In the case where the type $R$ represents preorders, we want to interpret the definition that $x$ is behaviourally $R$-related to $y$, as saying that $x$ is simulated by $y$. How is this so?

The usual way of approaching simulation in coalgebras is via something called a relator [4,3,13]. In the case of a functor $F: \text{Set} \to \text{Set}$, an $F$-relator is a functor $\Gamma: \text{Rel} \to \text{Rel}$ that satisfies certain additional properties. Using this a general notion of $\Gamma$-simulation for $F$-coalgebras can be defined. Further, associated with $F$ and $\Gamma$ is a functor $T: \text{Preord} \to \text{Preord}$ [4, Lemma 5.5] and [13, Definition 11] given by

$$T(X, R_X) = (F(X), \Gamma(R_X))$$

and under certain conditions [4, Theorem 9.4] the final $T$-coalgebra is the final $F$-coalgebra with the preorder given by the $\Gamma$-similarity relation. This final $T$-coalgebra characterises $\Gamma$-similarity of $F$-coalgebras as every set carries a discrete preorder (equality). Thus for every $F$-coalgebra there is a corresponding $T$-coalgebra, and given two $F$-coalgebras, the $\Gamma$-similarity relation on those two $F$-coalgebras is given by the preorder on the images of states under the corresponding unique cospan of morphisms to the final $T$-coalgebra [3, Remark 21].

Now in our general framework, for the initial $L$-algebra, every $T$-coalgebra has a unique theory map making it a model. Therefore if there exists a final $T$-coalgebra, it is a model, and moreover every other model factors uniquely via it. It is thus the final model $Z$. So for any cospan of models

$$X_1 \xrightarrow{f_1} X_3 \xleftarrow{f_2} X_2$$

such that $f_1(x_1) R X_3 f_2(x_2)$, there exists a unique model morphism $g: X_3 \to Z$, and this gives $g \circ f_1(x_1) R_Z g \circ f_2(x_2)$. So if $T$ is given by an $F$-relator as above, our

---

3 The use of arbitrary $L$-algebras allows propositional variables to be treated directly, rather than as nullary modalities.
notion of similarity coincides with $\Gamma$-similarity.

Our notion of simulation can thus be seen as taking the $F$-relator notion of simulation and extending it to arbitrary cospans in $\text{Mod}(A, \alpha)$, not just those with the final $T$-coalgebra as the target, and also to an arbitrary functor $T$, rather than one arising from a functor $F$ on $\textbf{Set}$ and an $F$-relator $\Gamma$.

The idea of approaching coalgebraic simulation via enrichment first appeared in [7], though there the authors enrich over $\textbf{Pos}$, not $\textbf{Preord}$. It is also in [7] that the link between enrichment over preordered sets and the relator approach to simulation was first made.

**Example 5.3** Consider the logical connection of Example 3.4, and where we fix the type $R$ to be preorders. Then logically $R$-related corresponds to inclusion of theories. To explain behaviourally $R$-related we can consider a generalisation of the finite powerset functor on $\text{Set}_R$. We define

$$P_{\text{fin}}(X, R_X) = (P_{\text{fin}}(X), R_{P_{\text{fin}}(X)})$$

where

$$UR_{P_{\text{fin}}(X)}V \iff \forall x \in U \exists y \in V. xR_X y$$

(this can be seen to be an example of the $\text{Sim}$ relator of [13]). Then using this we can define

$$T(X, R_X) = P_{\text{fin}}((\Sigma, R_\Sigma) \times (X, R_X))$$

making the $T$-coalgebras finite branching labelled transition systems with labels from $\Sigma$. Further, we assume that $R_\Sigma$ is the equality relation. Then we see that for a $T$-coalgebra $\gamma: (X, R_X) \rightarrow T(X, R_X)$, the fact that $\gamma$ must be $R$-preserving, means that $xR_X y$ implies for all $(l, z) \in \gamma(x)$ there exists $(l, z') \in \gamma(y)$ such that $zR_X z'$, which will be seen to be the usual notion that $x$ is simulated by $y$.

The other two case for the type $R$ are variations on simulation, and are summarised as follows:

(i) If $R$ is the type preorder, then we have simulation.

(ii) If $R$ is the type partial order, then we have simulation where mutual simulation is bisimulation.

(iii) If $R$ is the type equivalence relation, then we have mutual simulation.

(iv) If $R$ is the type equality, then we have bisimulation.

The type $R$ determines which type of behavioural relation we have e.g. simulation or bisimulation, but it is the definition of the functor $T$, specifically the way it defines an $R$ relation on the codomain (in practice probably via a relator), that determines what is actually meant by simulation for the $T$-coalgebras.

The following result is a simple consequence of the fact that morphisms in $\mathcal{X}$ are $R$-preserving.

**Proposition 5.4** Given two models $X_1, X_2$ in $\text{Mod}(A, \alpha)$, and states $x_1 \in X_1$, $x_2 \in X_2$, if $x_1$ and $x_2$ are behaviourally $R$-related then $x_1$ and $x_2$ are logically $R$-related.
Finally, our expressivity definition is as follows.

**Definition 5.5** An $L$-algebra $(A, \alpha)$ is **R-expressive** for $\text{Mod}(A, \alpha)$ if for all models in $\text{Mod}(A, \alpha)$, states are logically $R$-related if and only if they are behaviourally $R$-related.

### 6 Characterisation of Expressivity

In [15] we gave an abstract, category theoretic, characterisation of expressivity for bisimulation. Here we extend this to include simulation. In actual fact the changes are minimal, and amount to little more than replacing $\text{IntMod}_M(A, \alpha)$ with $R\text{-Mod}(A, \alpha)$, and injective functions with $R$-reflecting ones.

The following theorem gives sufficient conditions for $R$-expressivity.

**Theorem 6.1** Given an $L$-algebra $(A, \alpha)$, if the following hold:

(i) every model in $\text{Mod}(A, \alpha)$ factors via some model in $R\text{-Mod}(A, \alpha)$,

(ii) for every pair $I_1, I_2$ in $R\text{-Mod}(A, \alpha)$ there is a cospan $I_1 \to I_3 \leftarrow I_2$ in $R\text{-Mod}(A, \alpha)$,

then $(A, \alpha)$ is $R$-expressive for $\text{Mod}(A, \alpha)$.

**Proof.** Take any pair of models $X_1$ and $X_2$ in $\text{Mod}(A, \alpha)$. Then these factor via the $R$-models $I_1$ and $I_2$ respectively, and by assumption there exists an $R$-model $I_3$ such that there exists a cospan $I_1 \to I_3 \leftarrow I_2$ in $R\text{-Mod}(A, \alpha)$. Thus both $X_1$ and $X_2$ factor via $I_3$.

Spelling this out, the models $((X_1, \gamma_1), f_1)$ and $((X_2, \gamma_2), f_2)$ factor via the $R$-model $((I_3, \zeta_3), m_3)$ via $T$-coalgebra morphisms $g_1: (X_1, \gamma_1) \to (I_3, \zeta_3)$ and $g_2: (X_2, \gamma_2) \to (I_3, \zeta_3)$ such that $f_1 = m_3 \circ g_1$ and $f_2 = m_3 \circ g_2$.

Now suppose two states $x_1 \in X_1$ and $x_2 \in X_2$ are logically $R$-related for $(A, \alpha)$. Then $f_1(x_1) R_S(A) f_2(x_2)$, which means $m_3 \circ g_1(x_1) R_S(A) m_3 \circ g_2(x_2)$, and since $m_3$ is $R$-reflecting, $g_1(x_1) R_T g_2(x_2)$, and $x_1$ and $x_2$ are behaviourally $R$-related.

The converse direction is given by Proposition 5.4.

In addition to Assumption 1, for several of the results that follow we will also need to make assumptions about the category $\text{Mod}(A, \alpha)$. These assumptions contain the precise category theoretic properties that we require to prove our results, however, in Corollary 6.5 we shall see that these assumptions actually follow from more basic assumptions about the category $X$ and the functor $T$. Whenever we require these additional assumptions this will be indicated in the premises of the relevant proposition, lemma, or theorem.

**Assumption 2** Given an $L$-algebra $(A, \alpha)$ the category $\text{Mod}(A, \alpha)$ has small pushouts, a factorisation system $(E_{\text{Mod}(A, \alpha)}, M_{\text{Mod}(A, \alpha)})$, and is $E_{\text{Mod}(A, \alpha)}$-cowellpowered, where $M_{\text{Mod}(A, \alpha)}$ is a subclass of those morphisms in $\text{Mod}(A, \alpha)$ with $R$-reflecting underlying functions, and $E_{\text{Mod}(A, \alpha)}$ is a subclass of those morphisms in $\text{Mod}(A, \alpha)$ with surjective underlying functions.
Using Assumption 2 we can prove a converse to Theorem 6.1. The most difficult part is the proof that expressivity of \((A, \alpha)\) for \(\text{Mod}(A, \alpha)\) implies that all models factor via \(R\)-models.

**Theorem 6.2** Given an \(L\)-algebra \((A, \alpha)\), and assuming the conditions of Assumption 2 hold, if \((A, \alpha)\) is \(R\)-expressive for \(\text{Mod}(A, \alpha)\) then every model in \(\text{Mod}(A, \alpha)\) factors via an \(R\)-model in \(R\text{-Mod}(A, \alpha)\).

**Proof.** The proof is quite long and technical, but essentially the same as the bisimulation version of [15]. Thus we shall only sketch the outline:

(i) Observe that all model morphisms have an \((E_{\text{Mod}(A, \alpha)}, M_{\text{Mod}(A, \alpha)})\)-factorisation.

(ii) For a model \(((X, \gamma), f)\) use that \(\text{Mod}(A, \alpha)\) is \(E_{\text{Mod}(A, \alpha)}\)-cowellpowered to take the pushout \(((\coprod_{<e_j>} I_j, \zeta), f^\dagger)\) of the \(E_{\text{Mod}(A, \alpha)}\)-quotient objects of \(((X, \gamma), f)\).

(iii) Construct a model epimorphism \(h: ((X, \gamma), f) \to ((\coprod_{<e_j>} I_j, \zeta), f^\dagger)\).

(iv) Use the diagonalisation property of the factorisation system to show \(h, p_j \in E_{\text{Mod}(A, \alpha)}\) for all \(j \in J\).

(v) Use that morphisms in \(M_{\text{Mod}(A, \alpha)}\) have underlying functions that are \(R\)-reflecting to show that \(f^\dagger\) has an underlying function that is \(R\)-reflecting.

(vi) Observe that this makes \(((\coprod_{<e_j>} I_j, \zeta), f^\dagger)\) an \(R\)-model. 

\(\square\)

**Corollary 6.3** Given an \(L\)-algebra \((A, \alpha)\), and with the following assumptions:

(i) the conditions of Assumption 2 hold,

(ii) \(\text{Mod}(A, \alpha)\) has binary coproducts,

if \((A, \alpha)\) is \(R\)-expressive for \(\text{Mod}(A, \alpha)\) then for every pair \(I_1, I_2\) in \(R\text{-Mod}(A, \alpha)\) there is a cospan \(I_1 \to I_3 \leftarrow I_2\) in \(R\text{-Mod}(A, \alpha)\).

**Proof.** Given two \(R\)-models \(I_1\) and \(I_2\), since they are also models their coproduct exists, and by Theorem 6.2 the coproduct factors via an \(R\)-model, say \(I_3\), and this induces an obvious cospan between \(I_1\) and \(I_2\). 

\(\square\)

From Theorems 6.1, 6.2, and Corollary 6.3 we obtain our main expressivity result - an abstract, category theoretic, characterisation of \(R\)-expressivity.

**Theorem 6.4** Given an \(L\)-algebra \((A, \alpha)\), and with the following assumptions:

(i) the conditions of Assumption 2 hold,

(ii) \(\text{Mod}(A, \alpha)\) has binary coproducts,

\((A, \alpha)\) is \(R\)-expressive for \(\text{Mod}(A, \alpha)\) if and only if

(i) every model in \(\text{Mod}(A, \alpha)\) factors via an \(R\)-model in \(R\text{-Mod}(A, \alpha)\),

(ii) for every pair \(I_1, I_2\) in \(R\text{-Mod}(A, \alpha)\) there is a cospan \(I_1 \to I_3 \leftarrow I_2\) in \(R\text{-Mod}(A, \alpha)\).
The conditions of Assumption 2 follow from appropriate conditions on the category $X$ and the functor $T$. Essentially what is required is that $X$ have enough colimits, and that $X$ has a proper factorisation system, the monomorphisms of which are preserved by $T$.

**Corollary 6.5** Given an $L$-algebra $(A, \alpha)$, and with the following assumptions:

(i) $X$ is a concrete category (over $\text{Set}_R$) that has small conical colimits, has a factorisation system $(E_X, M_X)$, and is $E_X$-cowellpowered, where $M_X$ is chosen to be a subclass of those morphisms in $X$ that have underlying functions that are injective and $R$-reflecting, and $E_X$ is chosen to be a subclass of those morphisms in $X$ with underlying functions that are surjective,

(ii) $T$ preserves $M_X$, i.e. $m \in M_X \Rightarrow T(m) \in M_X$,

$(A, \alpha)$ is $R$-expressive for $\text{Mod}(A, \alpha)$ if and only if

(i) every model in $\text{Mod}(A, \alpha)$ factors via an $R$-model in $R^{-}\text{Mod}(A, \alpha)$,

(ii) for every pair $I_1, I_2$ in $R^{-}\text{Mod}(A, \alpha)$ there is a cospan $I_1 \to I_3 \leftarrow I_2$ in $R^{-}\text{Mod}(A, \alpha)$.

**Proof.** We have to show that the premises of Theorem 6.4 hold. Firstly we observe that by Theorem 4.6 $\text{Mod}(A, \alpha)$ has small conical colimits.

To show that the factorisation system of $X$ lifts to $\text{Mod}(A, \alpha)$ we note that in [5] it is observed that if $T$ preserves $M_X$, and the members of $M_X$ are monomorphisms, then the factorisation system of $X$ lifts to $\text{CoAlg}(T)$, and it is easy to see that this extends to $\text{Mod}(A, \alpha)$.

Finally, $\text{Mod}(A, \alpha)$ is $E_X$-cowellpowered since the morphisms in $E_X$ are epimorphisms, and this ensures that given a span in $\text{Mod}(A, \alpha)$ where the underlying morphisms are in $E_X$, there is an isomorphism between the two so defined $E_X$-quotient objects in $\text{Mod}(A, \alpha)$, if and only if, there is an isomorphism between the underlying $E_X$-quotient objects in $X$.  

7 A Simulation Example

In order to better illustrate what actually is going on with $R$-models, we shall cast into our framework the well known result that the logic given by the syntax

$$\mathcal{L} \ni \phi ::= \text{tt} \mid \phi \land \phi \mid \langle l \rangle \phi \quad \text{where } l \in \Sigma$$

is expressive for simulation of finite branching labelled transition systems. This continues Example 5.3, and again the type $R$ is fixed to be preorders.

What we aim to do is define an $L$ and $\delta$ such that $L$ is the initial $L$-algebra, then any $T$-coalgebra has a unique theory map that makes it a model for $L$. For
any such model $(((X, R_X), \gamma), f)$ we then factor it as follows

![Diagram](https://via.placeholder.com/150)

where $(((I, R_I), \zeta), m)$ is an $R$-model for $\mathcal{L}$.

We start by defining the modalities using the forgetful functor $U: \text{MSL} \to \text{Set}_R$ and its left adjoint $F: \text{Set}_R \to \text{MSL}$ that creates free meet semilattices with a top element. Specifically, $F(X, R_X)$ is the usual free meet semilattice $F(X)$, over the set of variables $X$, with the relation $R_{F(X)}$ given by the equality relation extended by $[x]R_{F(X)}[y] \iff xR_X y$ for all $x, y \in X$. The functor $L$ is then defined by $L(A) = F \bigsqcup_{I \in \Sigma} U(A)$, and we choose $\delta$ as follows

$$\delta_{(X, R_X)}: LP(X, R_X) \to PT(X, R_X)$$

$$\top_{LP(X, R_X)} \mapsto \mathcal{P}_{\text{fin}}((\Sigma, R\Sigma) \times (X, R_X))$$

$$[V_i]_{LP(X, R_X)} \mapsto \{W \in \mathcal{P}_{\text{fin}}((\Sigma, R\Sigma) \times (X, R_X)) \mid \exists (l, x) \in W. x \in V_i\}$$

$$[V_i \land V_i]_{LP(X, R_X)} \mapsto \delta_{(X, R_X)}([V_i]_{LP(X, R_X)}) \land \delta_{(X, R_X)}([V_i]_{LP(X, R_X)})$$

where the notation $V_i$ indicates that $V$ is from the copy of $UP(X, R_X)$ indexed by $l$. This corresponds to a modal operator $\langle l \rangle$ for each $l \in \Sigma$, where $\langle l \rangle a$ is satisfied at a state if there is an $l$ transition from that state to one where $a$ is satisfied. From this we get

$$\delta^*_A: TS(A) \to SL(A)$$

$$V \mapsto \{[W]_{L(A)} \in L(A) \mid V \in \delta_{S(A)} \circ L(\rho_A)([W]_{L(A)})\}$$

where $\rho_A(a) = \{s \in S(A) \mid a \in s\}$ is the unit of the logical connection, and so

$$\delta_{S(A)} \circ L(\rho_A): L(A) \to PTS(A)$$

$$\top_{L(A)} \mapsto \mathcal{P}_{\text{fin}}((\Sigma, R\Sigma) \times (S(A), R_{S(A)}))$$

$$[a_i]_{L(A)} \mapsto \{V \in \mathcal{P}_{\text{fin}}((\Sigma, R\Sigma) \times (S(A), R_{S(A)})) \mid \exists (l, s) \in V. a_i \in s\}$$

$$[a_i \land a_i]_{L(A)} \mapsto \delta_{S(A)} \circ L(\rho_A)([a_i]_{L(A)}) \land \delta_{S(A)} \circ L(\rho_A)([a_i]_{L(A)})$$

where again the notation $a_i$ indicates that $a$ is from the copy of $U(A)$ indexed by $l$.

To go back to our plan, what we need to do next is factor a theory map $f$ via an $R$-reflecting morphism $m$. We do this using the well known result that Hennessy-
Milner logic ($\mathcal{L}$ with negation) is expressive for bisimulation of finite branching labelled transition systems.

Specifically, using a functor $U: \mathbb{X} \to \mathbb{X}$ that assigns to every object the discrete preorder (in other words, forgets the current preorder), any model $((((X, R_X), \gamma), f)$ can be quotiented via a surjective $T$-coalgebra morphism $e: (X, \gamma) \to (I, \zeta)$, where $I$ is a subset of the ultrafilters (maximally consistent sets) of Hennessy-Milner logic. There is then an obvious function $m: I \to US(A)$ that maps an ultrafilter in Hennessy-Milner logic to the corresponding filter in $\mathcal{L}$ by throwing out all the formulae that contain negation, and moreover, $U(f) = m \circ e$. The way to think of this, is that a filter in $\mathcal{L}$ lists all the possible future things a state in a transition system can do, and an ultrafilter in Hennessy-Milner logic explicitly adds all the things it cannot do. Note: it is well known that mutual simulation is not the same as bisimulation, therefore $m$ is clearly not injective.

Now $S(A)$ is ordered by inclusion, and it is easy to see that $I$ can be given a preorder $R_I$ such that $e$ is $R$-preserving, and $m$ is both $R$-preserving and $R$-reflecting. Specifically, we can order the ultrafilters of $I$ by the inclusion order on their negation free subsets.

Further, since $e$ is surjective, $((I, U(\zeta)), U(m))$ is a model for $\mathcal{L}$. What remains to be shown is that $\zeta$ preserves the preorder $R_I$, for if that is the case, then $((I, R_I), \zeta), m)$ is an $R$-model for $\mathcal{L}$. It is easily seen that this is the case if $T$ preserves $R$-reflecting morphisms, and $\delta^*_A$ is $R$-reflecting. The former is not very hard to show, so what remains is to show that $\delta^*_A$ is $R$-reflecting. In fact we shall show this for an arbitrary $L$-algebra $(A, \alpha)$.

To do this suppose $V\mathcal{R}_{TS(A)}V'$, then

$$V\mathcal{R}_{TS(A)}V' \iff \exists (l, s) \in V. \forall (l, s') \in V' \text{ either } l \neq l' \text{ or } s\mathcal{R}_{S(A)}s'$$

Now, our plan is to find $[a_t]_{L(A)} \in L(A)$ such that $a_t \in s$, and for all $(l', s') \in V'$ either $l \neq l'$ or $a_t \notin s'$. If there is no $(l', s') \in V'$ such that $l = l'$ then we can take $a_t = (T_A)l$. If that is not the case, then there is a finite set of pairs $(l, s') \in V'$ such that $s\mathcal{R}_{S(A)}s'$. Now $s\mathcal{R}_{S(A)}s'$ means $s \notin s'$, so it is possible to find an element of $s$ that is not in any of the $s'$ (do it pairwise and then take the meet - we can do this as $V'$ is finite). Therefore $\delta_A(V) \not\subseteq \delta_A(V')$, which means $\delta_A(V)\mathcal{R}_{SL(A)}\delta_A(V')$, and thus $\delta_A$ is $R$-reflecting.

We have thus shown that every model for $\mathcal{L}$ factors via an $R$-model. Further, since $\textbf{Set}_R$ has coproducts, by Theorem 4.6 the coproduct of any pair of $R$-models, as models, exists, and since any model factors via an $R$-model, this yields a cospan of $R$-models. Therefore we have satisfied the premises of Theorem 6.1 and so we can conclude, as expected, that $\mathcal{L}$ is expressive for simulation of finite branching labelled transition systems.

**Acknowledgement**

The author would like to thank Corina Cîrstea for the many interesting discussions and her invaluable guidance. Special thanks also go to Alexander Kurz for sharing
an unpublished version of [7].

References