Contents

Preface v

DIRK PATTINSON, LUTZ SCHRÖDER
Sound and Complete Equational Reasoning over Comodels .......... 1

JEAN-PHILIPPE BERNARDY, THIERRY COQUAND, GUILHEM MOULIN
A Presheaf Model of Parametric Type Theory ......................... 17

KLAUS KEMEL
Healthiness Conditions for Predicate Transformers ................. 34

JAMES LAIRD
Sequential Algorithms for Unbounded Nondeterminism ............... 51

NEIL GHANI, PATRICIA JOHANN, FREDRIK NORDVALL FORSBERG, FEDERICO ORSANIGO, TIM REVELL
Bifibrational functorial semantics of parametric polymorphism .... 67

MARCO PAVIOTTI, RASMUS EJLERS MOGELBERG, LARS BIRKEDAL
A Model of PCF in Guarded Type Theory .......................... 84

MARC BAGNOL
Representation of Partial Traces ...................................... 100

ROBERT RAND, STEVE, ZDANCEWIC
VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs 113

BART JACOBS, BRAM WESTERBAAN
An Effect-Theoretic Account of Lebesgue Integration .............. 129

NILS JANSEN, BENJAMIN LUCIEN KAMINSKI, JOOST-PIETER KATOEN, FEDERICO OLMEDO, FRIEDRICH GRETZ, ANNABELLE McIVER
Conditioning in Probabilistic Programming ........................ 143

CHRIS HEUNEN, MARTTI KARVONEN
Reversible Monadic Computing ....................................... 159

KIM G. LARSEN, RADU MARDARE, BINGTIAN XUE
Alternation-Free Weighted Mu-Calculus: Decidability and Completeness .................................. 179

APIWAT CHANTAWIBUL AND PAVEL SOBOCIŃSKI
Towards Compositional Graph Theory ................................ 201
ALEŠ BIZJAK, RASMUS EJLERS MØGELBERG
A model of guarded recursion with clock synchronisation ............... 216

TARMO UUSTALU
Stateful runners of effectful computations .............................. 232

SERGEY GONCHAROV, CHRISTOPH RAUCH, LUTZ SCHRÖDER
Unguarded Recursion on Coinductive Resumptions ......................... 250

MATIJA PRETNAR (Invited tutorial paper)
An Introduction to Algebraic Effects and Handlers ......................... 265

VINCENT DANOS, ILIAS GARNIER
Dirichlet is natural ......................................................... 281

MATHYS RENNELA, SAM STATON
Complete positivity and natural representation of quantum computations ......................................................... 298

TAUS BROCK-NANNESTAD, NICOLAS GUENOT
Focused Linear Logic and the λ-calculus ................................. 314

A. W. ROSCOE
The expressiveness of CSP with priority ............................... 330

H. BEOHAR, P.J.L. CUIJPERS
Open maps in concrete categories and branching bisimulation for prefix orders ......................................................... 345

PEDRO DA ROCHA PINTO, THOMAS DINSDALE-YOUNG, PHILIPPA GARDNER (Invited tutorial paper)
Steps in Modular Specifications for Concurrent Modules .................. 360
Preface

This volume contains the proceedings of the Thirty-first Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). The conference is held in Nijmegen, The Netherlands, between June 22nd and June 25th. The conference is held jointly with the Sixth Conference on Algebra and Coalgebra in Computer Science (CALCO'15).

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 has particularly stressed 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 neighbouring areas.

The program committee of MFPS XXXI consisted of

- Achim Jung, Birmingham, UK
- Alan Jeffrey, Bell Labs, USA
- Alexandra Silva, Radboud U, NED
- Andrej Bauer, Ljubljana, SLO
- Andrew Pitts, Cambridge, UK
- Catherine Meadows, NRL, USA
- Dan R. Ghica, Birmingham, UK (Chair)
- Derek Dreyer, MPI-SWS, GER
- Franck van Breugel, York, CA
- F. Nordvall Forsberg, Strathclyde, UK
- Grigore Roșu, Illinois, USA
- James Brotherston, UCL, UK
- Joel Ouaknine, Oxford, UK
- Marco Gaboardi, Dundee, UK
- Michael Mislove, Tulane, USA
- Michele Pagani, Paris, FRA
- Nikos Tzevelekos, Queen Mary, UK
- Olivier Laurent, ENS Lyon, FRA
• Paul B. Levy, Birmingham, UK
• Prakash Panangaden, Montreal, CA
• Radu Mardare, Aalborg, DEN
• Radha Jagadeesan, DePaul, USA
• Ross Duncan, Strathclyde, UK
• Steve Brookes, CMU, USA
• Ugo Dal Lago, Bologna, ITA
• Ulrich Schöpp, Munich, GER

The papers were refereed by the program committee and by several outside referees, whose help is gratefully acknowledged.

The invited speakers of the conference were:

• Andy Pitts (Cambridge, UK), joint invited speaker with CALCO
• Thierry Coquand (Chalmers, Sweden)
• Paul B. Levy (Birmingham, UK)
• Sam Staton (Radboud University, The Netherlands).

The conference also included four special sessions featuring invited tutorials given by the special session organisers:

• Game Semantics on Monday, June 22nd, organised by Andrzej Murawski (Warwick, UK) which included talks by Nikos Tzevelekos (Queen Mary, UK), James Laird (Bath, UK) and Pierre Clairambault (ENS, Lyon).

• Concurrent programs and separation logic on Tuesday, June 23rd, organised by Philippa Gardner (Imperial College, UK) which included talks by Larks Birkedal (Aarhus, Denmark), Aleks Nanevski (IMDEA, Spain) and Azalea Raad (Imperial College, UK).

• Nominal techniques on Wednesday, June 24th, organised by Daniela Petrisan (Radboud University, The Netherlands) which included talks by Mikolaj Bojanczyk (Warsaw, Poland), Bartek Klin (Warsaw, Poland) and Paul-André Melliès (Paris Denis Diderot, France).

• Algebraic Effects on Thursday, June 25th, organised by Matija Pretnar (Ljubljana, Slovenia) which included talks by Ohad Kammar (Cambridge, UK), Conor McBride (Strathclyde, UK) and Tom Schrijvers (Leuven, Belgium).
MFPS gratefully acknowledges the financial support of the U.S. Office of Naval Research, Royal Society of the Netherlands (KNAW), the Dutch Organization for Scientific Research (NWO), and the Radboud University of Nijmegen. We thank the organisers Bart Jacobs, Alexandra Silva and Sam Staton for their effort. This event was organised in cooperation with ACM SIGLOG.

Nijmegen, June 22–25, 2015

Dan R. Ghica, PC Chair
Sound and Complete Equational Reasoning over Comodels

Dirk Pattinson
The Australian National University

Lutz Schröder
Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract
Comodels of Lawvere theories, i.e. models in \( \text{Set}^{op} \), model state spaces with algebraic access operations. Standard equational reasoning is known to be sound but incomplete for comodels. We give two sound and complete calculi for equational reasoning over comodels: an inductive calculus for equality-on-the-nose, and a coinductive/inductive calculus for equality modulo bisimulation which captures bisimulations syntactically through non-wellfounded proofs.

Keywords: Equational Logic, Comodels, Completeness, Bisimulation

1 Introduction
Comodels are an algebraic abstraction of the notion of global state, prominently used in the operational semantics of programming languages [13,15,11]. The most prominent example is the modelling of global state in imperative programs, where the explicit modelling of a store as a function that maps locations to values is replaced by algebraic operations that read and manipulate the values of global variables. Equations, in the standard universal-algebraic sense, ensure the intended semantics of these operations. Comodels are attractive for two reasons: first, the abstract implementation of state from the operational semantics, as state is not modelled explicitly, but only manipulated using operations. Second, the operations integrate seamlessly with programming language syntax. Some progress has been made towards the development of congruence formats in these settings [2]. While this builds the link between operational and denotational semantics, the link with

1 dirk.pattinson@anu.edu.au
2 lutz.schroeder@fau.de
3 Work supported by the DFG under project COAX (SCHR 1118/12-1)

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
comodels are essentially an algebraic concept, defined in terms of function symbols and equations, but the interpretation of function symbols takes place in the category \( \textbf{Set}^{\text{op}} \), the opposite category of the category \( \textbf{Set} \) of sets and maps. While a unary function symbol, say \( \text{wr}_v \), for writing a value \( v \), is still interpreted as a unary function \( \langle \text{wr}_v \rangle : C \rightarrow C \) on a set (of comodel states), general \( n \)-ary function symbols are interpreted as functions \( \langle f \rangle : C \rightarrow n \cdot C = C + \cdots + C \) (\( n \) times) and so are not understood as constructors, but as a combination of observation and state change. For example, a binary function symbol \( \text{rd} \) (that we think of as reading a binary value from a memory cell) receives the interpretation \( \langle \text{rd} \rangle : C \rightarrow C + C \) and so indicates the value of the cell being read (by choosing one of the alternatives in \( C + C \)) on top of a new state. As a consequence, the standard tools of universal algebra for proving completeness of equational logic, first and foremost the Lindenbaum construction, are not available in the setting of comodels. Moreover, it is easy to see that equational reasoning is sound, but incomplete over comodels. The easiest example is that of a theory comprising a nullary operation \( n \) and no equations: a comodel for this theory interprets \( n \) as a function \( \langle n \rangle : C \rightarrow 0 \cdot C = \emptyset \) and therefore is empty, hence validates all equations; but clearly not all equations are derivable from the empty set of equations by standard equational reasoning. Excluding nullary operations does not improve this situation: we give an example below, due to Power, that shows that the same effect happens for a commutative binary operation.

This situation is remedied in the present paper, where we provide sound and complete calculi for equational reasoning over comodels. The overall flavour of comodels is coalgebraic, with a very simple type functor but with added complexity creeping in via the algebraic equations, which, for instance, may relate terms of different lookahead. The semantics of equations over comodels therefore naturally comes in two variants: satisfaction on-the-nose, and satisfaction up to bisimilarity, inducing correspondingly different notions of logical consequence. For reasoning on-the-nose, we give a standard, purely inductive calculus. We formalize this calculus in the style of a labelled sequent system. Key rules of the system express that terms with disjoint sets of free variables can never be equal in the comodel interpretation, and that terms with \( n \) free variables are essentially \( n \)-fold case statements allowing for a corresponding case distinction. For reasoning modulo bisimilarity, we then extend this inductive calculus by a single coinductive rule that allows us to conclude that two comodel terms are equal if they have the same output and their successors are equal. This rule may be applied in non-wellfounded proofs, resulting in a mixed inductive/coinductive calculus.

**Related Work.** We have already mentioned [14] where the theory of arrays is developed in terms of comodels, and the use of comodels in the semantics of programming languages [12,15,2]. None of these papers is concerned with axiomatic semantics, i.e. the equational logic of comodels. The model/comodel duality is investigated in [9,8] on the basis of clones and establishes, in our terminology, a dual equivalence between categories of comodels and certain topological spaces, but does not investigate the logical aspects. The proof-theoretic analysis of circular coinduction [16] has a goal that is similar to the completeness of equational reasoning modulo bisimulation, and blocks the application of the congruence rule in coinductive reasoning.
steps to achieve soundness. We achieve a similar effect by including substitution
(which in our dualized setting plays, for purposes of coinductive proofs, an analo-
gous role as congruence does in standard equational reasoning) as an axiom rather
than a rule. As the proof calculus of op.cit. is purely inductive, no general com-
pleteness result can be established; this is remedied in our setup by using a mixed
inductive-coinductive calculus. Mixed inductive / coinductive definitions have been
investigated in type theory (e.g. [1,6]), and it appears that the modulo-bisimulation
calculus given here can be straightforwardly encoded, thus presenting another exam-
ple of the usefulness of these definition principles. An approach that is structurally
similar to ours has been put forward for equational reasoning over non-wellfounded
terms [7] where coinduction is used to capture non-wellfoundedness, whereas our
calculus derives equations between finite terms, and employs coinduction to char-
acterize validity modulo bisimulation. Non-wellfounded calculi have also been used
in [5,4] to formalize arguments by infinite descent for inductive definitions (rather
than up-to bisimulation arguments) where proofs are finite, possibly cyclic graphs
subject to an external well-formedness condition.

2 Preliminaries and Notation

Categorical Notions. We write + for (categorical) coproducts, and given $f_1, \ldots, f_n : A_i \to B$ we write $[f_1, \ldots, f_n] : A_1 + \cdots + A_n \to B$ for the induced co-tuple. For a set $V$ we write $V \cdot A = \bigsqcup_{v \in V} A$ for the $V$-th copower of $A$, inj$_v : A \to V \cdot A$ for the

Algebraic Notions. A signature is a set $\Sigma$ (of function symbols) equipped with a
function $ar : \Sigma \to \omega$ assigning arities to operations. We say that $f \in \Sigma$ is $n$-ary
if $ar(f) = n$. The set $T_\Sigma V$ of Terms over $\Sigma$ with variables in $V$ is defined in the
standard way. If $t \in T_\Sigma V$ is a term, and $f$ is unary, we often write $f.t$ for $f(t)$,
and if $t(x)$ is a term with a free variable $x$, we write $t.x$ for $t(x)$. A substitution is
a mapping $\sigma : V \to T_\Sigma V$; we write $t\sigma$ for the result of simultaneously replacing
every free variable $x$ in a term $t \in T_\Sigma V$ by $\sigma(x)$.

Fixpoints. If $M$ is a monotone operator on a complete lattice (we only consider
lattices of subsets in this paper), we write $\mu M$ or $\mu X.M(X)$ for its least fixpoint,
and $\nu M$, or $\nu X.M(X)$ for its greatest fixpoint.

3 Comodels

A comodel of an equational theory $T$ is a model of the Lawvere theory $L$ induced
by $T$ in the opposite category $\text{Set}^{op}$ of the category $\text{Set}$ of sets and functions. That
is, a comodel is a finite coproduct preserving functor $L^{op} \to \text{Set}$. In the area of
programming language semantics, comodels are one way to explain the meaning
of programs that change state. Given an algebraic signature $\Sigma$, a comodel for $\Sigma$
consists of a carrier set, say $C$, that we think of as a set of states, and a function $\langle f \rangle : C \to n \cdot C$ for every $n$-ary function symbol $f \in \Sigma$. As a consequence, a unary function
symbol \( f \in \Sigma \) is interpreted as a state-changing operation \( \langle f \rangle : C \to C \), whereas an \( n \)-ary function symbol \( g \) is an \( n \)-fold branching statement \( \langle g \rangle : C \to C + \cdots + C = n \cdot C \), for \( n \geq 1 \). That is, given a state \( c \in C \), \( \langle g \rangle \) delivers a new state in one of \( n \) alternative branches. The presence of nullary function symbols (or constants) in a signature immediately implies that all comodels are empty, as constants are interpreted as functions \( C \to 0 \cdot C = \varnothing \). In contrast to the evaluation of terms in universal algebra, the information flows from left to right when interpretation of terms over comodels. For example, when evaluating the term \( a \cdot \langle f \rangle \) over a comodel \( C \) by applying the interpretation of \( g \) \( f \) applies (the interpretation of) a tuple \( \langle A, C, \ldots, C \rangle \) to \( c \). Given a set \( V \) \( \langle \cdot \rangle \Sigma \) for short, is a tuple \( \langle C, \langle \cdot \rangle \rangle \) where \( C \) is a set, and \( \langle f \rangle : C \to n \cdot C \) is a function for all \( n \)-ary \( f \in \Sigma \). Given a set \( V \) \( \langle \cdot \rangle \Sigma \) engenders an interpretation of terms \( \langle \cdot \rangle \Sigma V : T_{\Sigma} V \to C \to C \cdot V \cdot C \) by
\[
\langle \cdot \rangle V = \text{inj}_V \quad \text{and} \quad \langle f(t_1, \ldots, t_n) \rangle V = \left[ \langle t_1 \rangle V, \ldots, \langle t_n \rangle V \right] \circ \langle f \rangle
\]
where \( T_{\Sigma} V \) denotes the set of \( \Sigma \)-terms with variables in \( V \). If clear from the context, we will elide the superscript \( V \). A morphism of comodels \( \langle C, \langle \cdot \rangle \rangle \) and \( \langle D, \langle \cdot \rangle \rangle \) is a function \( h : C \to D \) that commutes with the interpretation of function symbols, i.e. \( n \cdot h \circ \langle f \rangle_C = \langle f \rangle_D \circ h \) for all \( n \)-ary \( f \in \Sigma \).

A comodel \( \langle C, \langle \cdot \rangle \rangle \) satisfies an equation \( s = t \), where \( s, t \in T_{\Sigma} V \) are terms over the set \( V \) of variables, if \( \langle s \rangle V = \langle t \rangle V \), and if \( E \) is a set of equations, we say that \( \langle C, \langle \cdot \rangle \rangle \) is a \( \Sigma, E \)-comodel if it satisfies all equations in \( E \).

A simple example of a comodel is the following one-bit memory cell that supports operations for reading and writing.

**Example 3.2** Consider a one-bit memory cell, represented by comodels for the signature \( \Sigma = \{ \text{rd}, \text{wr}_0, \text{wr}_1 \} \) where \( \text{wr}_0 \) and \( \text{wr}_1 \) are unary and \( \text{rd} \) is binary. A comodel for \( \Sigma \) consists of a (state) set \( C \) and operations \( \langle \text{wr}_0 \rangle, \langle \text{wr}_1 \rangle : C \to C \) that we interpret as writing 0 (resp. 1) to the memory cell, and an operation \( \text{rd} : C \to C + C \) that will branch into the left hand component of the coproduct if evaluated at a cell storing 0, and into the right hand component, otherwise. Note that reading the cell may in general change its state. To ensure the intended behaviour of the memory cell, we stipulate the equations \( E = \{ \text{wr}_0.\text{rd}(x, y) = \text{wr}_0.x, \text{wr}_1.\text{rd}(x, y) = \text{wr}_1.y \} : \) the effect of reading a memory cell immediately after writing is completely determined by the bit written where the effect of writing to the cell is preserved.

More examples of comodels, primarily concerned with state, can be found in [14,12]. As mentioned at the beginning of this section, comodels are usually presented as finite coproduct-preserving functors \( \text{L}^\text{op} \to \text{Set} \) where \( \text{L} \) is a Lawvere theory. For the purposes of this paper, it is more convenient to work with comodels for equational theories, as the latter determine concrete syntax that can be manipulated in the equational calculi that we are about to give. The equivalence of comodels for a
Lawvere theory, and \( \Sigma, E \)-comodels is not relevant for the remainder of the paper, but included to justify our terminology. Recall that the Lawvere theory induced by a signature \( \Sigma \) and a set \( E \) of equations between \( \Sigma \)-terms is the category \( \mathcal{C} \) whose objects are the natural numbers, and whose morphisms \( \mathcal{C}(n, m) = \mathcal{K}(UF)(n, m) \) are the morphisms in the Kleisli category of the monad \( UF \) where \( U: \text{Alg}(\Sigma, E) \to \text{Set} \) is the forgetful functor and \( F \) its left adjoint.

**Proposition 3.3** Let \( E \) be a set of equations over a signature \( \Sigma \). Then the category of \( \Sigma, E \)-comodels is isomorphic to the category of comodels for the Lawvere theory induced by \( \Sigma \) and \( E \).

Soundness of equational reasoning over comodels is an immediate corollary.

**Corollary 3.4** Let \( \Sigma \) be a signature and \( E \) a set of \( \Sigma \)-equations. If an equality \( s = t \) is derivable from \( E \) in (standard) equational logic, then \( \langle s \rangle = \langle t \rangle \) in every \( \Sigma, E \)-comodel \( (C, \langle \cdot \rangle) \).

### 4 Labelled Tableau Equality On-The-Nose

We proceed to give a complete system for deriving equalities between comodel terms. The following example due to Power shows that the usual proof systems of equational logic in general fail to be complete over comodels, and the main contribution of this paper is a complete calculus.

**Example 4.1**

(i) Let \( \Sigma \) be a signature that contains a nullary function symbol \( c \).

As pointed out at the beginning of Section 3, the only \( \Sigma \)-comodel is (carried by) the empty set. Therefore, irrespective of the set \( E \) of equations, all equations are valid over \( \Sigma \)-comodels, but not all equations are derivable in general, e.g. for \( E = \emptyset \).

(ii) Consider the theory given by a single, commutative binary operation, i.e. \( \Sigma = \{ f \} \) with \( f \) binary, and \( E = \{ f(x, y) = f(y, x) \} \). If \( C \) is a comodel for \( \Sigma \) and \( E \), then necessarily \( C = \emptyset \): otherwise we could pick \( c \in C \), and supposing that \( \langle f(x, y) \rangle(c) = \text{inj}_y(d) \) we would obtain \( \text{inj}_x(d) = \langle f(x, y) \rangle(c) = \langle f(y, x) \rangle(c) = \text{inj}_y(d) \) whence \( x = y \) for distinct variables \( x \) and \( y \), contradiction. As the supposition \( \langle f(x, y) \rangle(c) = \text{inj}_y(d) \) leads to a similar contradiction, we obtain \( C = \emptyset \) and as a consequence, \( C \) satisfies \( s = t \) for all terms \( s, t \in T\Sigma(V) \). But clearly not all equations \( s = t \) are derivable from \( E \) in equational logic (keeping algebraists off the dole).

The complete system we are about to introduce manipulates labelled expressions of the form \( a.t = b.s \) where \( s, t \) are terms, and \( a, b \) are state variables, distinct from the variables from which we build terms. We interpret state variables \( a, b \) as elements of (the carrier of) the comodel, and read \( a.s = b.t \) as saying that the terms \( s \) and \( t \) have the same denotation (on the nose) if evaluated in state \( a \) and \( b \), respectively, of a given comodel. Throughout the section, we fix a signature \( \Sigma \) comprising terms with associated arities, a countable set \( V \) of (term) variables, and a countable set \( Z \) of state variables, disjoint from \( V \). We write \( T \) for the set of terms built from function symbols in \( \Sigma \) using the variables in \( V \), \( \text{FV}(t) \) for the set of (free) variables occurring in \( t \), and \( t(x_1, \ldots, x_n) \) to indicate that \( \text{FV}(t) \subseteq \{x_1, \ldots, x_n\} \).
Definition 4.2 A labelled term is a pair \((a, t) \in Z \times T\), written \(a.t\), where \(t\) is a term and \(a\) is a state variable. A labelled equation is a pair of labelled terms, written \(a.s = b.t\), with (free) state variables \(FS(a.s = b.t) = \{a, b\}\). A (comodel) sequent is a pair, written \(\Gamma \Rightarrow A\), where \(\Gamma\) is a set of labelled equations, and \(A\) is a labelled equation. We briefly write \(A\) for \(\varnothing \Rightarrow A\) and extend the notion of (free) state variables to sets of equations and comodel sequents by \(FS(\Gamma) = \bigcup\{FS(A) \mid A \in \Gamma\}\) and \(FS(\Gamma \Rightarrow A) = FS(\Gamma) \cup FS(A)\). A valuation for a comodel \((C, \theta)\) is a function \(\theta : Z \rightarrow C\), and we write \(C, \theta = a.s = b.t\) if \(\langle s\rangle(\theta a) = \langle t\rangle(\theta b)\); \(C, \theta = \Gamma\) if \(C, \theta = B\) for all \(B \in \Gamma\); and \(C, \theta = A\) if \(C, \theta = \Gamma\) implies \(C, \theta = A\). Finally, if \(E\) is a set of (unlabelled) equations, \(E = \Gamma \Rightarrow A\) if \(C, \theta = \Gamma \Rightarrow A\) for all comodel/valuation pairs \(C, \theta\) where \(C\) satisfies all equations in \(E\). A renaming is a function \(\tau : Z \rightarrow Z\), and we write \(\langle a.s = b.t \rangle \tau = \tau(a).s = \tau(b).t\) and \(\Gamma \tau\) for \(\{A \tau \mid A \in \Gamma\}\). Substitutions extend to labelled terms by \((a.t)\sigma = a.t\sigma\), to labelled equations by \((a.s = b.s)\sigma = (a.t = b.t)\sigma\), and to sets of labelled equations by \(\Gamma\sigma = \{A\sigma \mid A \in \Gamma\}\).

We specifically do not require the antecedent \(\Gamma\) in a comodel sequent to be finite. In the calculus for equality on-the-nose, this enables us to speak about strong completeness, and we will later need infinite sets of assumptions in the calculus for equality modulo bisimulation.

Remark 4.3 State variables could be internalized in the original term language. A state variable is essentially a unary function symbol that only appears at the head of a term. In particular, this overloads the dot-notation for the application of unary functions \(a.t = a(t)\) in a consistent way. All rules of the system we are about to introduce remain sound if we relax the interpretation of a state variable to be a unary function on states instead of a single state. (Completeness, proved later, transfers trivially to this more permissive semantics.) What distinguishes state variables from function symbols is that we have an infinite reservoir of state variables allowing us to pick fresh variables when necessary, as well as the possibility of renaming state variables.

As an equation \(s = t\) is satisfied by all \(\Sigma, E\)-comodels if and only if \(C, \theta = a.s = a.t\) for all \(\Sigma, E\)-comodels \(C\) and valuations \(\theta\), it suffices to derive all valid comodel sequents. The system that achieves this comprises the following rules. Substitution takes the form of an axiom

\[
\text{(subst)} \quad \Gamma, a.s(x_1, \ldots, x_n) = b.t(x_1, \ldots, x_n) \Rightarrow a.s(u_1, \ldots, u_n) = b.t(u_1, \ldots, u_n)
\]

where \(u_1, \ldots, u_n \in T_{\Sigma}V\) are terms, and stipulates that given that \(a.s\) and \(b.t\) take the same branches and end up in the same poststates, the same will hold if we postcompose with identical terms. Via the identity substitution, this implies in particular that every equation in \(\Gamma\) is derivable.

The fact that every term \(r(x_1, \ldots, x_n)\) evaluates to one of the alternatives \(x_1, \ldots, x_n\) is captured by the rule

\[
\text{(case)} \quad \{\Gamma, a.r(x_1, \ldots, x_n) = b.x_i \Rightarrow c.s = d.t \mid 1 \leq i \leq n\} \quad \{b \notin FS(\Gamma \Rightarrow c.s = d.t)\},
\]

which employs similar mechanisms as disjunction elimination and existential elim-
ination in natural deduction: to conclude \( c.s = d.t \) we have to derive \( c.s = d.t \)
assuming each of the possible outputs \( x_i \) of some labelled term \( a.r \) in turn, in each
case giving a fresh name \( b \) to the poststate reached by \( a.r \). We have a version of
falsum-elimination (on the left) and a rule that asserts validity of all substitution
instances of the axioms in \( E \) (on the right).

\[
\begin{align*}
\text{(disj)} & \quad \Gamma \Rightarrow a.s = b.t \\
& \quad \Gamma \Rightarrow c.u = d.v \\
& \quad (FV(s) \cap FV(t) = \emptyset) \\
\text{(E)} & \quad \Gamma \Rightarrow a.s \sigma = a.t \sigma \\
& \quad (s = t \in E)
\end{align*}
\]

The falsum elimination rule (disj) (for disjoint) reflects the fact that labelled terms
\( a.s \) and \( b.t \) that do not have any variable in common cannot be equal: if \( \{s\}(\theta a) = \( (x, \alpha) \) and \( \{t\}(\theta b) = \( (y, \beta) \) with \( x \in FV(s) \), \( y \in FV(t) \) then \( x \neq y \) as \( FV(s) \cap FV(t) = \emptyset \).

From such an impossible equality, the rule therefore allows us to draw arbitrary
conclusions \( c.u = d.v \), in analogy to the classical ex-falso-quodlibet principle. Axiom
(\( E \)) simply asserts that all substitution instances of equations are valid. These rules
are completed with the standard rules for equality

\[
\begin{align*}
\text{(sym)} & \quad \Gamma \Rightarrow a.s = b.t \\
& \quad \Gamma \Rightarrow b.t = a.s \\
\text{(trans)} & \quad \Gamma \Rightarrow a.s = b.t \\
& \quad \Gamma \Rightarrow b.t = c.u \\
& \quad \Gamma \Rightarrow a.s = c.u \\
\text{(ref)} & \quad \Gamma \Rightarrow a.t = a.t
\end{align*}
\]

ensuring symmetry, reflexivity and transitivity of equality, and the renaming rule

\[
\begin{align*}
\text{(ren)} & \quad \Gamma \Rightarrow A \\
& \quad \Gamma T \Rightarrow A T
\end{align*}
\]

as the only structural rule. We write \( E \vdash \Gamma \Rightarrow A \) if \( \Gamma \Rightarrow A \) can be derived using the
above rules.

**Remark 4.4** As the antecedent \( \Gamma \) of a comodel sequent \( \Gamma \Rightarrow A \) may be infinite,
an application of (case) could be blocked if \( \Gamma \) contains all state variables. The rule
(ren) allows us to free a state variable that we can then use as a fresh variable in
(case).

The next example revisits our examples of valid formulas not derivable in standard
equational reasoning (Example 4.1), showing that the extended system does handle
these examples.

**Example 4.5**  
(i) If \( \Sigma \) contains a nullary function symbol, say \( n \), we have \( E \vdash a.n = a.n \) by (ref) and as \( FV(n) \cap FV(n) = \emptyset \) we obtain \( E \vdash a.s = b.t \) for
arbitrary labelled terms \( a.s \) and \( b.t \) using (disj).

(ii) Consider a commutative binary function \( f \), that is, \( \Sigma = \{ f \} \) and \( E = \{ f(x,y) = f(y,x) \} \). We show that \( E \vdash c.s = d.t \) for all \( c,d \in Z \) and all \( s,t \in T \). Let \( a \in Z \)
be arbitrary and pick a fresh \( b \in Z \), i.e. \( b \notin \{a,c,d\} \). Fix \( \Gamma_1 = \{a.f(x,y) = b.x\} \).
We derive \( E \vdash \Gamma_1 \Rightarrow c.s = d.t \), eliding the leading \( E \vdash \), by

\[
\begin{align*}
\Gamma_1 & \Rightarrow a.f(x,y) = b.x \\
\Gamma_1 & \Rightarrow b.x = a.f(x,y) \\
\Gamma_1 & \Rightarrow b.x = a.f(x,y) = a.f(y,x) \\
\Gamma_1 & \Rightarrow a.f(x,y) = b.y \\
\Gamma_1 & \Rightarrow a.f(x,y) = a.f(y,x) = b.y \\
\Gamma_1 & \Rightarrow b.x = b.y \\
\Gamma_1 & \Rightarrow c.s = d.t
\end{align*}
\]
where the leftmost and rightmost leaves are by \((\text{subst})\) and we use \((\text{disj})\) in the last inference step. Taking \(\Gamma_2 = \{a.f(x, y) = b.y\}\) we obtain \(E \Rightarrow c.s = d.t\) by a symmetric derivation. We conclude, expanding the definitions of \(\Gamma_1\) and \(\Gamma_2\),

\[
\begin{align*}
  a.f(x, y) = b.x & \Rightarrow c.s = d.t \\
  a.f(x, y) = b.y & \Rightarrow c.s = d.t \\
  c.s = d.t
\end{align*}
\]

using \((\text{case})\).

Soundness is proved by a standard induction over derivations.

**Proposition 4.6 (Soundness)** The above system is sound for equality on-the-nose over comodels, i.e. \(E \models A\) whenever \(E \vdash \Gamma \Rightarrow A\).

The following technical preparations are needed for establishing completeness. We formulate substitution as an axiom rather than as a rule, as the substitution axiom (but not the substitution rule, see Remark 5.13) remains sound also for modulo-bisimulation reasoning. However, substitution as a rule is admissible.

**Lemma 4.7 (Admissibility of the Substitution Rule)** Suppose that \(E \vdash \Gamma \Rightarrow a.s = b.t\) and let \(\sigma : V \rightarrow T_2(V)\) be a substitution. Then \(E \vdash \Gamma \Rightarrow a.s \sigma = a.t \sigma\).

Moreover, cut is admissible.

**Lemma 4.8 (Cut Admissibility)** Suppose that \(E \vdash \Gamma \Rightarrow A\) and \(E \vdash \Gamma, A \Rightarrow B\). Then \(E \vdash \Gamma \Rightarrow B\).

Although not strictly necessary for the technical development, we show that entailment is also closed under substitution for state variables.

**Lemma 4.9** Suppose that \(E \vdash \Gamma \Rightarrow a.s_i = b.t_i\) for \(i = 1, \ldots, n\) and let \(a \notin FS(\Gamma)\). Then \(E \vdash \Gamma \Rightarrow c.u(s_1, \ldots, s_n) = c.u(t_1, \ldots, t_n)\).

The completeness proof for the system is partly similar to the classical Henkin construction. We keep the set \(E\) of equations fixed throughout, and show that every non-derivable sequent has a countermodel. Since sequents can be infinite, we add additional state variables to allow for a Lindenbaum lemma.

**Notation 4.10** We fix a second denumerable set \(Z'\) of extended state variables. We call a labelled term \(a.s\) extended if \(a \in Z \cup Z'\), and standard if \(a \in Z\). An extended labelled equation is a labelled equation between extended terms, and a labelled equation between standard terms is called standard. We write \(\vdash_{\text{ext}}\) for derivability of extended labelled equations, and (continue to) write \(\vdash\) for derivability of standard labelled equations.

We first establish that derivability in extended system is conservative. This is where we need renaming, as otherwise an application of \((\text{case})\)

\[
\frac{
\quad \Gamma, a.s(x_1, \ldots, x_n) = b.x_1 \Rightarrow A \\
\ldots \\
\quad \Gamma, a.s(x_1, \ldots, x_n) = b.x_n \Rightarrow A \\
}{\quad \Gamma \Rightarrow A}
\]

where \(b \in Z'\) and \(\Gamma\) contains all state variables in \(Z\) couldn’t be translated back to the standard system. Conservativity follows from the following:
Lemma 4.11 Suppose that \( E_{\text{ext}} \Gamma \Rightarrow A \) and \( \tau : Z \cup Z' \rightarrow Z \) is a bijective renaming. Then \( E \vdash \Gamma \tau \Rightarrow A \tau \).

Corollary 4.12 (Conservativity) Suppose that \( E_{\text{ext}} \Gamma \Rightarrow A \) and \( A \) are standard. Then \( E \vdash \Gamma \Rightarrow A \).

The countermodel construction will be based on witnessed sets of labelled equations.

Definition 4.13 A set \( \Gamma \) of extended labelled equations is \textit{witnessed} if for all extended labelled terms \( a.s \) there exists a variable \( x \in V \) and an extended state variable \( b \in Z' \) such that \( a.s = b.x \in \Gamma \).

Lemma 4.14 (Lindenbaum lemma) Let \( \Gamma \) be a set of standard labelled equations, and \( A \) be a standard labelled equation such that \( E_{\text{ext}} \Gamma \Rightarrow c.s = d.t \). Then \( E_{\text{ext}} \Gamma \Rightarrow A \) iff \( A \in \Gamma \).

Maximal sets are closed under derivation:

Lemma 4.15 (Derivability is containment) Let \( \Gamma \) be a set of (extended) labelled equations that is maximal with respect to \( E_{\text{ext}} \Gamma \Rightarrow c.s = d.t \). Then \( E_{\text{ext}} \Gamma \Rightarrow A \) iff \( A \in \Gamma \).

We now construct a countermodel from a witnessed set of extended labelled sequents as in the Lindenbaum lemma; the elements of the countermodel are equivalence classes of extended state variables. This construction is related to Henkin’s completeness proof of first-order logic.

Lemma 4.16 Let \( \Gamma \) be a witnessed set of labelled equations that is maximal with the property that \( E_{\text{ext}} \Gamma \Rightarrow c.s = d.t \). Then the following hold.

(i) For all labelled terms \( a.u \) there exists a unique \( x \in \text{FV}(u) \) and a (not necessarily unique) \( b \in Z' \) such that \( E_{\text{ext}} \Gamma \Rightarrow a.u = b.x \).

(ii) The relation \( \sim \) on the set \( Z' \) of extended state variables defined by

\[ a \sim b \iff E_{\text{ext}} \Gamma \Rightarrow a.x = b.x \text{ for some } x \in V \]

is an equivalence.

(iii) Putting \( ([f]_\Gamma[.])_\sim = \text{inj}_\Gamma([.]_\sim) \) iff \( a.f(x_1, \ldots, x_n) = b.x_i \in \Gamma \) yields a well-defined comodel structure on \( Z'/\sim \).

(iv) For all terms \( u \) and all \( a \in Z' \) we have \( ([u]_\Gamma[a]) = (x,[b]) \) iff \( E_{\text{ext}} \Gamma \Rightarrow a.u = b.x \).

(v) For the valuation \( \theta(a) = [a]_\sim \) we have \( Z'/\sim, \theta \models a.u = b.v \) iff \( E_{\text{ext}} \Gamma \Rightarrow a.u = b.v \).

(vi) The comodel \( Z'/\sim \) satisfies all equations in \( E \).

The first three items are straightforward, and the fourth is by induction on \( u \). The rule (case) is not used in the proof of Lemma 4.16; its role in the completeness proof is to enable the construction of witnessed sets, i.e. the Lindenbaum lemma.

Completeness is an immediate consequence of the countermodel construction:

Corollary 4.17 (Completeness for reasoning on-the-nose) Comodel reasoning is complete for on-the-nose equality over comodels, i.e. if \( E \models \Gamma \Rightarrow a.s = b.t \) then \( E \vdash \Gamma \Rightarrow a.s = b.t \).
5 Labelled Tableaux for Equality modulo Bisimulation

We now extend the reasoning system for equality on-the-nose introduced in the previous section to a system for equational reasoning modulo bisimilarity. Consider the following example.

Example 5.1 Recall the signature Σ and equations E describing a one-bit memory cell, introduced in Example 3.2. We intuitively expect that a second write overwrites the first, i.e. the equations \( \text{wr}_a \cdot \text{wr}_b \cdot x = \text{wr}_b \cdot x \) hold for \( a, b \in \{0, 1\} \). However, a comodel may internally record additional information beyond the content of the cell, for example the number of times the cell has been written to. Consider, for example, \( \text{comodel} \) may internally record additional information beyond the content of the cell, for example the number of times the cell has been written to. Consider, for example, \( C = \{0, 1\} \times \mathbb{N} \) with \( \langle \text{wr}_a \rangle(c, n) = (a, n + 1) \) for \( a = 0, 1 \) and \( \langle \text{rd} \rangle(c, n) = (c, (c, n)) \). Clearly \( \langle \text{wr}_0 \cdot \text{wr}_0 \cdot x \rangle(0, n) = (0, n + 2) \neq (0, n + 1) = \langle \text{wr}_0 \cdot x \rangle(0, n) \) for any \( n \in \mathbb{N} \) so that \( E \not\approx \text{wr}_0 \cdot \text{wr}_0 \cdot x = \text{wr}_0 \cdot x \). On the other hand, we cannot tell \( \text{wr}_0 \cdot \text{wr}_0 \cdot x \) and \( \text{wr}_0 \cdot x \) apart by (repeatedly) applying operations to both and observing that they give rise to different alternatives, as they behave identically under \( \text{rd} \).

This phenomenon of comodel states being different but observationally indistinguishable is entirely standard, and captured by the established notion of bisimilarity. We explicitly instantiate this concept to the particular notation of comodels. For simplicity of notation, we work with bisimilarity on a single comodel; in the comodel setting, this is without loss of generality as bisimilarity across two models is the same as bisimilarity in their coproduct.

Definition 5.2 Let \( C \) be a \( \Sigma \)-comodel. A relation \( B \subseteq C \times C \) is a comodel bisimulation if, for all \( (c, c') \in B \) and all \( n \)-ary \( f \in \Sigma \) we have that \( \langle f \rangle(c) B_n \langle f \rangle(c') \) where \( B_n = \{(i, c), (i, c') \mid c \text{Be } c' \text{ and } 0 \leq i < n\} \). We say that two comodel states \( c, c' \) are comodel bisimilar, and write \( c \approx c' \), if they are related by some comodel bisimulation.

For any two elements \( (c_1, c_2) \) to be bisimilar, they have to branch into the same alternative under any \( f \) producing elements that are again bisimilar. In Example 3.2 we obtain that two states are bisimilar if and only if they store the same bit. In Example 3.2 we obtain that two states are bisimilar if and only if they store the same bit.

Example 5.3 Let \( \Sigma \) and \( E \) be as in Example 3.2 above, and let \( C \) be a \( \Sigma \)-comodel that satisfies \( E \). For \( c \in C \) and \( i \in \{0, 1\} \) we write \( c \xrightarrow{\text{rd}} i \) if there exists \( d \in C \) such that \( \langle \text{rd} \rangle = (i, d) \), that is, \( c \) branches to the \( i \)-th alternative under \( \text{rd} \). Then the relation \( B \) defined by \( cBc' \) iff \( (c \xrightarrow{\text{rd}} i \iff c' \xrightarrow{\text{rd}} i \text{ for } i = 0, 1) \) is a comodel bisimulation: if \( cBc' \) we need to establish that \( \langle \text{wr}_0 \rangle(c) B_1 \langle \text{wr}_0 \rangle(c') \) and \( \langle \text{rd} \rangle(c) B_2 \langle \text{rd} \rangle(c') \) for \( i = 0, 1 \). The left hand relationship follows from the equation \( \text{wr}_i \cdot \text{rd}(x_0, x_1) = \text{wr}_i \cdot x_1 \), and the right hand relationship holds by definition of \( B \).

It is clear that unions of comodel bisimulations are again comodel bisimulations, so that the largest bisimulation on a comodel always exists, and coincides with comodel bisimilarity. We can quotient comodels by bisimilarity:
Lemma and Definition 5.4 Let $C$ be a $\Sigma$-comodel. Then bisimilarity $\approx$ is an equivalence relation on $C$, and putting

$$\langle f \rangle([c]_a) = \text{inj}_i([d]_a) \iff \langle f \rangle(c) = \text{inj}_i(d)$$

yields a well-defined comodel structure on $C/\approx$, the bisimulation quotient $C/\approx$ of $C$.

Continuing Example 5.3 we obtain that the bisimulation quotient of any comodel satisfies the equation $\text{wr}_a.\text{wr}_b.x = \text{wr}_b.x$.

Example 5.5 Let $C$ be a $\Sigma$-comodel that satisfies $E$ where $\Sigma$ and $E$ are as in Example 3.2. Then $B$ as described in Example 5.3 is easily seen to coincide with comodel bisimilarity $\approx$ on $C$. Hence, $C/\approx \cong \{0,1\}$ with $\langle \text{wr}_0 \rangle(\gamma) = 0$, $\langle \text{wr}_1 \rangle(\gamma) = 1$ and $\langle \text{rd} \rangle(\gamma) = \text{inj}_i(\gamma)$. In particular, $C/\approx = \text{wr}_a.\text{wr}_b.x = \text{wr}_b.x$ for all $a, b \in \{0,1\}$.

We now extend the labelled deduction system given in the previous paragraph to account for equality modulo comodel bisimulation. We formalize this semantically by replacing on-the-nose satisfaction of (labelled) equations in a comodel by satisfaction of (the same) equations in its bisimulation quotient.

Definition 5.6 Let $C$ be a $\Sigma$-comodel and $\theta : Z \rightarrow C$ a valuation. We write $C, \theta \models a.s \leftrightarrow b.t$ if $C/\approx, \theta \models a.s \leftrightarrow b.t$ where $\bar{\theta}(a) = [a]_a$ (and $C/\approx$ is the bisimulation quotient of $C$). This extends to comodel sequents so that $C, \theta \models \Gamma \Rightarrow A$ if $C, \theta \models A$ whenever $C, \theta \models B$ for all $B \in \Gamma$. We say that a comodel sequent is valid up to bisimilarity in the class of all comodels that satisfy $E$ up to bisimilarity, in symbols $E \models \Gamma \Rightarrow A$, if $(C, \theta) \models \Gamma \Rightarrow A$ whenever $C/\approx$ is a $\Sigma, E$-comodel and $\theta$ is a valuation.

We now extend the derivation system $E \vdash$ to a derivation system $E \models$ on comodel sequents to capture validity of equations modulo bisimulation. The ensuing system is a mixed inductive/coinductive proof system where the rules of $E \vdash$ may be applied inductively, and the following bisimulation rule

$$\frac{\text{(bisim)} \{ \Gamma \Rightarrow a.f(x_1, \ldots, x_n) = b.f(x_1, \ldots, x_n) \mid n \in \mathbb{N}, f \in \Sigma \text{n-ary} \}}{\Gamma \Rightarrow a.x = b.x}$$

is applied coinductively. Formally, we partition the set of rules into

- inductive rules comprising (subst), (case), (disj), (sym), (ref), (trans), (ren) and (E) and say that $\Gamma \Rightarrow A$ is an inductive consequence of a set $I$ of comodel sequents if it is the conclusion of an inductive rule with premises in $I$, and
- one coinductive rule (bisim), and say that $\Gamma \Rightarrow A$ is a coinductive consequence of a set $C$ of comodel sequents if it is a conclusion of (bisim) with premises in $C$.

We write $\text{Ind}(I)$ and $\text{Coind}(C)$ for the set of inductive / coinductive consequences of sets $I$ and $C$ of comodel sequents, respectively. We then take behavioural derivability $E \models \subseteq S$ to be the mixed fixpoint

$$E \models = \nu C. \mu I. (\text{Ind}(I) \cup \text{Coind}(C))$$

where we write $E \models \Gamma \Rightarrow A$ if $(\Gamma \Rightarrow A) \in E \models$. That is, a sequent is behaviourally derivable if it is the conclusion of a possibly non-wellfounded derivation where the
rule (bisim) can be applied infinitely often. The formulation of the proof system as a mixed fixpoint, with an inner inductive part, ensures that the inductive rules are only applied finitely many times between two successive instances of (bisim).

The soundness of the ensuing system is now no longer a simple matter of induction on derivations. Instead, we establish soundness in a step-by-step fashion using iterative approximations of behavioural derivability on the syntactic side, and iterative approximations of bisimilarity on the semantic side.

**Notation 5.7** Let $S$ be the set of comodel sequents, and consider the monotone operator $W : \mathcal{P}(S) \to \mathcal{P}(S)$, defined by

$$W(C) = \mu I.(\text{Ind}(I) \cup \text{Coind}(C))$$

and let $\vdash_0 = \mathcal{P}(S)$ and $\vdash_{n+1} = W(\vdash_n)$. We say that $\Gamma \Rightarrow A$ is $n$-step derivable if $\vdash_n \Gamma \Rightarrow A$. Similarly, for a comodel $C$, consider the monotone operator $W_C : \mathcal{P}(V \cdot C \times V \cdot C) \to \mathcal{P}(V \cdot C \times V \cdot C)$ defined by

$$W_C(R) = \{(x,c),(x,c') \mid (f(x_1,\ldots,x_n))(c)R(f(x_1,\ldots,x_n))(c') \text{ for all } f \in \Sigma\}$$

for $R \in V \cdot C \times V \cdot C$, and let $R_0 = (V \cdot C)^2$ and $R_{n+1} = W(R_n)$. We say that $(v,c)$ and $(v',c')$ are $n$-step bisimilar if $(v,c)R_n(v',c')$. If $C$ is a comodel for $\Sigma$ and $\Gamma \Rightarrow A$ is a comodel sequent, we say that $C,\theta \vdash_n a.s=b.t$ if $(a,s)R_n(b,t)$, and $C,\theta \vdash_n \Gamma \Rightarrow A$ if $C,\theta \vdash_n A$ whenever $C,\theta \vdash \Gamma$ (we explicitly require $C,\theta \not\vdash_n \Gamma$, not just $C,\theta \vdash_n \Gamma$).

We say that $\Gamma \Rightarrow A$ is $n$-step valid, and write $E \vdash_n \Gamma \Rightarrow A$, if $C,\theta \vdash_n \Gamma \Rightarrow A$ for all $\Sigma, E$-comodels $C$ and all valuations $\theta$.

This allows us to show soundness by establishing that $n$-step derivability implies $n$-step validity:

**Lemma 5.8** For states $c,c'$ in a $\Sigma$-comodel $C$, $c \approx c'$ if and only if $(x,c)R_n(x,c')$ for all $n \in \mathbb{N}$ and all $x \in V$.

It follows immediately that $\vdash$ is exhaustively approximated by the $\vdash_n$:

**Lemma 5.9** We have that $E \vdash = \cap_{n \in \mathbb{N}} E \vdash_n$.

For behavioural derivability, one inclusion suffices.

**Lemma 5.10** We have that $E \vdash \subseteq \cap_{n \in \mathbb{N}} E \vdash_n$.

The following is the key to soundness of behavioural derivability.

**Lemma 5.11** Let $n \in \mathbb{N}$ and let $\Gamma \Rightarrow A$ be a comodel sequent. Then $E \vdash_n \Gamma \Rightarrow A$ implies that $E \vdash_n \Gamma \Rightarrow A$.

**Proposition 5.12** Let $E$ be a set of $\Sigma$-equations. Then behavioural derivability is sound, i.e. $E \vdash \Gamma \Rightarrow A$ whenever $E \vdash \Gamma \Rightarrow A$.

**Proof.** Let $E \vdash \Gamma \Rightarrow A$. By Lemma 5.10, $E \vdash_n \Gamma \Rightarrow A$ for all $n$, so by Lemma 5.11, $E \vdash_n \Gamma \Rightarrow A$ for all $n$. By Lemma 5.9, it follows that $E \vdash \Gamma \Rightarrow A$. □
**Remark 5.13** The key to soundness, i.e. to Lemma 5.11, is that the substitution axioms \((\text{subst})\) are sound for \(\models_n\); the point is that \(\models_n\) requires the left-hand sides of comodel sequents to hold up to \(\approx\) and not just up to \(\equiv_n\). In contrast, the substitution rule
\[
(\text{subst}) \quad \Gamma \Rightarrow a.s(x_1,\ldots,x_n) = b.t(x_1,\ldots,x_n) \\
\Gamma \Rightarrow a.s(u_1,\ldots,u_n) = b.t(u_1,\ldots,u_n)
\]
is sound for \(\equiv\) and \(\models\) but not for \(\models_n\). And indeed, including this rule in the proof system for behavioural derivability is clearly unsound, as the following derivation (for \(\Sigma = \{f_1,\ldots,f_k\}\)) would establish \(a.x = b.x\) for arbitrary \(a,b \in Z\) where \(\overrightarrow{x}\) is a tuple of variables according to the arity of the preceding function symbol,
\[
(\text{∞}) \quad a.x = b.x \\
(\text{bisim}) \quad a.f_1(\overrightarrow{x}) = b.f_1(\overrightarrow{x}) \\
\vdots \\
(\text{∞}) \quad a.f_k(\overrightarrow{x}) = b.f_k(\overrightarrow{x}) \\
\]
and \((\text{∞})\) indicates a coinductive repeat of the derivation. This would be a legal derivation in the inductive/coinductive format, as it uses only finitely many inductive rules between successive applications of \((\text{bisim})\); but of course \(a.x = b.x\) should not be derivable for arbitrary \(a,b\).

For completeness, we need to show that the sequents \(\Gamma \Rightarrow A\) with \(E \models \Gamma \Rightarrow A\) are contained in the greatest fixpoint \(\nu C.\mu I.(\text{Ind}(I) \cup \text{Coind}(C))\). If \(V = \{\Gamma \Rightarrow A\mid E \models \Gamma \Rightarrow A\}\) are the comodel sequents that are universally valid modulo bisimulation, this follows (using Knaster-Tarski) if \(V \subseteq \mu I.(\text{Ind}(I) \cup \text{Coind}(V))\), that is, every \((\Gamma \Rightarrow A) \in V\) is inductively derivable from \(\text{Coind}(V)\). We follow the same approach as for completeness on-the-nose and use an additional, countable set of Henkin-constants.

**Notation 5.14** As in 4.10, extend the set \(Z\) of state variables by a second, countable set \(Z'\) and consider labelled terms of the form \(a.t\) where \(a \in Z \cup Z'\) and \(t \in T_\Sigma(V)\). As before, we call a labelled term (equation, set of equations) **standard** if they only mention state variables in \(Z\), and **extended** otherwise.

We write \(S = \{\Gamma \Rightarrow A\mid \Gamma \Rightarrow A\ \text{standard}, E \models \Gamma \Rightarrow A\}\) and \(X = \{\Gamma \Rightarrow A\mid \Gamma \Rightarrow A\ \text{extended}, E \models \Gamma \Rightarrow A\}\) for the set of standard (resp. extended) comodel sequents that are universally valid modulo bisimulation. Finally, \(E \vdash_S = \mu I.\text{Ind}(I) \cup \text{Coind}(S)\) is inductive entailment of standard labelled sequents from \(\text{Coind}(S)\), and \(E \vdash_{\text{ext}} X = \mu I.\text{Ind}(I) \cup \text{Coind}(X)\) is inductive derivability of extended labelled sequents from \(\text{Coind}(X)\).

Using this notation, our proof strategy indicated above is formalized as follows.

**Fact 5.15** We have that \((E \models \Gamma \Rightarrow A\ \text{whenever} \models \Gamma \Rightarrow A)\) if \(S \subseteq \mu I.(\text{Ind}(I) \cup \text{Coind}(S))\).

That is, to show completeness for the mixed inductive/coinductive system, we have to show completeness for a modified inductive system where we may use \(\text{Coind}(S)\) as additional assumptions. We proceed as for equality-on-the-nose, and re-visit the key lemmas.
Lemma 5.16. Suppose $E \vdash_{\text{ext}} \Gamma \Rightarrow A$ and $\tau : Z \cup Z' \Rightarrow Z$ is a bijective renaming. Then $E \vdash S \Gamma \Rightarrow A \tau$.

The proof of Corollary 4.12 translates directly to this new setting and we have:

Corollary 5.17. Suppose that $E \vdash_{\text{ext}} \Gamma \Rightarrow A$ and both $\Gamma$ and $A$ are standard. Then $E \vdash S \Gamma \Rightarrow A$.

Using the model construction of Lemma 4.16, we obtain the following.

Lemma 5.18. Suppose that $E \not\vdash S \Gamma \Rightarrow c.s = d.t$. Then there exists a witnessed set $\hat{\Gamma}$ of (extended) labelled equations that is maximal with the property $E \vdash_{\text{ext}} \hat{\Gamma} \Rightarrow c.s = d.t$ and a comodel structure on $Z'/\sim$ where $a \sim b$ iff $a.x = b.x \in \Gamma$ for some $x \in V$ and a valuation $\theta$ defined by $\theta(a) = [a]_\sim$ such that

(i) $Z'/\sim, \theta \vdash a.s = b.t$ iff $a.s = b.t \in \hat{\Gamma}$ and $Z'/\sim, \theta \not\vdash c.s = d.t$

(ii) If $B$ is a comodel bisimulation on $Z'/\sim$ then $cBc'$ implies that $c = c'$

The key step to showing that the model $Z'/\sim$ only admits the diagonal as a bisimulation is to show that any equation $A$ valid in $Z'/\sim$ is in fact a behavioural consequence of $\hat{\Gamma}$, that is, $\hat{\Gamma} \Rightarrow A$ is universally valid modulo bisimulation. This allows us to make use of the additional assumptions, i.e. the coinductive consequences of universally valid sequents, to establish that $A$ actually holds on-the-nose in $Z'/\sim$. From the above, completeness is immediate:

Corollary 5.19 (Completeness modulo bisimulation). Suppose that $E \vdash \Gamma \Rightarrow A$. Then $E \vdash \Gamma \Rightarrow A$.

Recall from Example 5.5 that $E \vdash a.wr_i,wr_j.x = b.wr_i.x$ under the axiomatization $E$ defined in Example 3.2. We give an example derivation of this equation.

Example 5.20. Let $\Sigma$ and $E$ be as in Example 3.2. For $\alpha = a_0 \ldots a_n \in \{0,1\}^+$ we write $wr_{\alpha}$ for $wr_{a_0} \ldots wr_{a_n}$ and $wr_{\alpha,\beta}$ stands for $wr_{\alpha} . wr_{\beta}$. We show, generalizing the original goal, that $E \vdash a.wr_{\alpha_i}.x = b.wr_{\beta_i}.x$ for all $\alpha,\beta \in \{0,1\}^*$, all $i = 0,1$, and all $a,b \in Z$. First note that for substitutions $\sigma$ and $\tau$, the rule

\[
\frac{\Gamma, a.s = c.u, b.t = d.v \Rightarrow a.s\sigma = b.t\tau}{\Gamma, a.s = c.u, b.t = d.v \Rightarrow c.u\sigma = d.v\tau}
\]

is derivable using (subst), (sym) and (trans). We first show that $E \vdash \Gamma \Rightarrow a.wr_{\alpha_i}.rd(x_0,x_1) = a.wr_{\alpha_i}.x_i$ for $i = 0,1$ and all sets $\Gamma$ of comodel sequents (generally, one can, of course, show that weakening is admissible and then restrict to $\Gamma = \emptyset$), in the derivation $D$

\[
\frac{\Gamma, a.wr_{\alpha_i}.x = c.x \Rightarrow c.wr_i.rd(x_0,x_1) = c.wr_i.x_i}{\Gamma, a.wr_{\alpha_i}.x = c.x \Rightarrow a.wr_{\alpha_i}.rd(x_0,x_1) = a.wr_{\alpha_i}.x_i}
\]

using $(E)$, $(\cdot)$ and (case). We now demonstrate that $\Gamma \Rightarrow a.wr_{\alpha_i}.rd(x_0,x_1) = b.wr_{\beta_i}.rd(x_0,x_1)$ is inductively derivable from $\Gamma \Rightarrow a.wr_{\alpha_i}.x_i = b.wr_{\beta_i}.x_i$. Building on $D$ and a variant $D'$ obtained by replacing $a$ with $b$, this is achieved, using symmetry and transitivity, by the derivation $\mathcal{E}$,
We finally show that $\Gamma \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x$ is behaviourally derivable, writing

$$\Gamma_1 = \Gamma, a.\text{wr}a_i.x = c.x \quad \text{and} \quad \Gamma_2 = \Gamma_1, b.\text{wr}b_i.x = d.x$$

to ease notation. If $c, d \notin FS(\Gamma)$, we have the following derivation where $(\infty)$ indicates a coinductive repeat of the same derivation with evident modifications.

$$(\infty) \quad \Gamma_2 \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x_i$$

$$(\infty) \quad \Gamma_2 \Rightarrow c.\text{wr}c_i.x_0 = d.\text{wr}d_i.x_0$$

$$\Gamma_2 \Rightarrow \text{wr}c_i.x_0 = d.\text{wr}d_i.x_0$$

$$\Gamma_2 \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x_i$$

$$\Gamma_2 \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x_i$$

$$\Gamma_2 \Rightarrow b.\text{wr}b_i.x_i = b.\text{wr}b_i.x_i$$

$$\Gamma_2 \Rightarrow b.\text{wr}b_i.x_i = b.\text{wr}b_i.x_i$$

$$\Gamma_2 \Rightarrow b.\text{wr}b_i.x_i = b.\text{wr}b_i.x_i$$

$$\Gamma_2 \Rightarrow \text{wr}c_i.x_0 = d.\text{wr}d_i.x_0$$

$$\Gamma_2 \Rightarrow \text{wr}c_i.x_0 = d.\text{wr}d_i.x_0$$

$$\Gamma_2 \Rightarrow c.\text{wr}c_i.x_0 = d.\text{wr}d_i.x_0$$

The ternary inference is (bisim), followed by (†) and (case) (twice). This last proof is an infinite derivation where (bisim) is used infinitely often, but only a finite number of applications of the inductive rules are used between two successive applications of (bisim); that is, it fits our inductive/coinductive format. Thus, $E \vdash \Gamma \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x_i$.

Generally, the way one will apply the inductive/coinductive calculus will be to identify a putative postfixed point, i.e., a set $W$ of comodel sequents, and then show in an inductive proof that $W$ can be derived from its coinductive consequences, i.e., that $W \subseteq \mu L. (\text{Ind}(I) \cup \text{Coind}(W))$. We have treated this principle informally in the above example; formally, we show in the example that the set $W$ of labelled sequents $\Gamma \Rightarrow a.\text{wr}a_i.x = b.\text{wr}b_i.x_i$, where $\Gamma$ is any set of labelled equations, $a, b \in Z$, and $a_i, b_i \in \{0, 1\}^+$, is a postfixedpoint.

**Conclusions**

We have given an inductive calculus for on-the-nose equational reasoning over co-models, and a mixed coinductive/inductive calculus for equational reasoning modulo bisimulation. We have done this in a bare bones setup without parametrized operations, e.g., using $n \text{ write}$ operations $\text{wr}_0, \ldots, \text{wr}_{n-1}$ to modify memory cells that can store $n$ distinct values, or one $\text{print}$ operation for every character in a given character set. Similarly, reading (a character or memory location) is expressed by a function of arity equal to the number of alternatives. One natural extension would therefore be to include parametrized operations. An orthogonal direction of future research is to automate comodel reasoning in the style of the CIRC theorem prover [10]. This requires bisimulations to be either finite, or at least sufficiently well-structured to be analysed automatically. A second topic is to use complete reasoning over co-models to bridge between the operational/denotational and the axiomatic semantics of stateful programs.
References


A Presheaf Model of Parametric Type Theory

Jean-Philippe Bernardy  Thierry Coquand  Guilhem Moulin

Chalmers University of Technology and University of Gothenburg
{bernardy,coquand,mouling}@chalmers.se

Abstract

We extend Martin-Löf’s Logical Framework with special constructions and typing rules providing internalized parametricity. Compared to previous similar proposals, this version comes with a denotational semantics which is a refinement of the standard presheaf semantics of dependent type theory. Further, this presheaf semantics is a refinement of the one used to interpret nominal sets with restrictions. The present calculus is a candidate for the core of a proof assistant with internalized parametricity.

Keywords:  Parametricity, Presheaf semantics, Type theory

1 Introduction

Reynolds [17] proved a general abstraction theorem (sometimes called parametricity theorem) about polymorphic functions. His argument is about a set theoretic semantic. As he stated it, the underlying idea is that the meanings of an expression in “related” environments will be “related” values. For instance, he proves that if \( t_X \) is a term of type \( X \to X \) and if we consider two sets \( A_0, A_1 \) and a relation \( R \subseteq A_0 \times A_1 \), then we have \( R([t_X]_{X=A_0}(a_0),[t_X]_{X=A_1}(a_1)) \) whenever \( R(a_0,a_1) \), where \([t_X]_{X=A}\) denotes the meaning of the expression \( t_X \) where \( X \) is interpreted by the set \( A \). As he noted, one can replace binary relations by \( n \)-ary relations in this statement, and in particular unary relations (predicates). In the latter case, the statement is the following: if \( A \) is a set and \( P \) is a predicate on \( A \), then we have \( P([t_X]_{X=A}(a)) \) whenever \( P(a) \) holds. Wadler [18] illustrates by many examples how this result is useful for reasoning about functional programs.

The argument and result of Reynolds are model-theoretic in nature. In the Logical Framework, it is possible to state such an abstraction result in a purely syntactical way. One states for example that if a function \( f \) has type \( (A : U) \to A \to A \) — the type of the polymorphic identity — then \( f \, A \, x \) is Leibniz-equal to \( x \), i.e., the following proposition holds:

\[
(A : U) \to (P : A \to U) \to (x : A) \to P \, x \to P(f \, A \, x)
\]
Indeed Bernardy et al. [9] prove such a result as a (syntactical) meta-theorem about type systems. However this result is not provable internally, i.e., the following proposition is not provable:

\[
(f : (A : U) \rightarrow A \rightarrow A) \rightarrow (A : U) \rightarrow (P : A \rightarrow U) \rightarrow (x : A) \rightarrow Px \rightarrow P(fAx) \quad (\star)
\]

Therefore users relying on the parametricity conditions have postulated the parametricity axiom [3, 11, 16]. However, because postulates do not have computational interpretations, such parametricity conditions can only be used in computationally-irrelevant positions.

Instead, one would like to be able to rely on parametricity conditions within the theory itself. Several attempts have been made [6, 7] — or are currently developed [2] — for designing an extension of dependent type theory in which such an internal form of parametricity holds. We propose another such system here. Our technical contributions are as follows:

- We present an extension of Martin-Löf’s Logical Framework (Sec. 2) which internalizes parametricity (as we show in Sec. 3 on page 22) and can be seen as a simplification and generalization of the systems of Bernardy and Moulin [6, 7]. In particular, we have a special construction \((a, p)\) which pairs a term \(a\) with its parametricity proof \(p\), as well as special projections to extract the proof. As we will show in Example 9, these new constructions enable us to prove the proposition \((\star)\) internally. (This is not possible with usual pairs and projections since the first projection does not commute with application.)

  The name \(i\) in the above construction is what we call a “color”; we want internalized parametricity not only for LF but also for the extended calculus, and as explained in [7], colors enable nested parametricity by keeping track of the different uses (this is analogous to building hypercubes and accessing their vertices as in [6]). However, unlike previous type theories with internalized parametricity [6, 7], the system presented here does not compute parametricity types: for instance, parametricity conditions are isomorphic to functions, rather than functions themselves. (As shown in Sec. 3, this does not appear to be an issue in practice.)

- We provide a denotational semantics, in the form of a presheaf model, for this type theory (Sec. 4 on page 26). This model is a refinement of the presheaf semantics used to interpret nominal sets with restrictions [10, 15].

We conjecture that conversion and type-checking are decidable for this system.

2 Syntax

In this section we define the syntax and typing rules of our parametric type theory, as well as the equality judgment.

We assume a special symbol ‘0’, and a countably infinite set \(\mathbb{I}\) of other symbols, called colors. The metasyntactic variables \(i, j, \ldots\) range over colors, while \(\varphi\) range over \(\mathbb{I} \cup \{0\}\). We further assume a fixed function \texttt{fresh}(\_\_) such that \(\texttt{fresh}(I) \in \mathbb{I} \setminus I\)
for any finite color set $I$. The main innovation of the type theory presented here is that terms may depend on (a finite number of) colors. For any term $a$, we note $\text{supp}(a)$ the set of free colors in $a$.

We do not attempt to explain what lead us to consider a colored type theory; for that we refer to [7] instead.

**Definition 1** (Syntax of terms and contexts).

$$A, B, P, T, a, p, t, u := x \quad \text{variable}$$

$$| \quad t u \quad \text{application}$$

$$| \quad \lambda x : A.t \quad \text{abstraction}$$

$$| \quad (x : A) \rightarrow B \quad \text{product}$$

$$| \quad |A| \quad \text{code}$$

$$| \quad \text{El}(A) \quad \text{decode}$$

$$| \quad U \quad \text{universe}$$

$$| \quad (a,i) \quad \text{colored pair}$$

$$| \quad (x : A) \times_i P \quad \text{colored type pair}$$

$$| \quad (t,i,u) \quad \text{colored function pair}$$

$$| \quad A \ni_i a \quad \text{parametricity type}$$

$$| \quad a \cdot i \quad \text{parametricity proof}$$

$$\Gamma, \Delta := () | \Gamma, x : A | \Gamma, i : \mathbb{I}$$

We give a few intuitions to interpret the novel syntax, before formally giving the typing rules of the system.

(i) Reynolds associates each type with a predicate. Here, each type is associated not with a single predicate, but many: one for every color. These multiple predicates are essential to interpret parametricity when it is nested. Indeed, using a single predicate yields inconsistencies. Furthermore these predicates are definable in the logic: the type $A \ni_i a$ expresses that $a$ satisfies the parametricity predicate associated with the type $A$ on color $i$. For each term $a$ and color $i$, the term $a(i0)$ is the erasure of $i$ in $a$. It is defined by induction on $a$ (Def. 2) and can be understood as a realizer [5] of $a$.

(ii) The term $a \cdot i$ yields a proof of $A \ni_i a(i0)$.

(iii) The forms $(a,i)p$, $(x : A)\times_i P$ and $(t,i,u)$ allow to locally associate parametricity proofs with a given realizer.

**Definition 2** (Color renaming and erasure). We consider a color $i \in \mathbb{I}$ and $\varphi \in \mathbb{I} \cup \{0\}$, and define the term $a(i\varphi)$ by induction on $a$.

$$x(i \varphi) = x$$

$$(t u)(i \varphi) = (t(i \varphi))(u(i \varphi))$$

$$(\lambda(x : A).t)(i \varphi) = \lambda(x : A(i \varphi)).t(i \varphi)$$

$$(x : A) \rightarrow B)(i \varphi) = (x : A(i \varphi)) \rightarrow (B(i \varphi))$$
\[|A|(i \varphi) = |A(i \varphi)|\]
\[\text{El}(A)(i \varphi) = \text{El}(A(i \varphi))\]
\[U(i \varphi) = U\]
\[(a \cdot i p)(i 0) = a\]
\[(a \cdot i p)(i j) = (a \cdot j p)\]
\[(a \cdot j p)(i \varphi) = (a(i \varphi).j p(i \varphi))\] if \(i \neq j\)
\[((x : A) \times_i P)(i 0) = A\]
\[((x : A) \times_i P)(i j) = (x : A) \times_j P\]
\[((x : A) \times_j P)(i \varphi) = (x : A(i \varphi)) \times_j P(i \varphi)\] if \(i \neq j\)
\[\langle t, i u \rangle(i 0) = t\]
\[\langle t, i u \rangle(i j) = \langle t, i u \rangle\]
\[\langle t, i u \rangle(i \varphi) = \langle t(i \varphi), i u(i \varphi)\rangle\] if \(i \neq j\)
\[(A \ni a)(i \varphi) = A A(i \varphi) \ni a(i \varphi)\]
\[(A \ni a)(i \varphi) = A(i \varphi) \ni a(i \varphi)\] if \(i \neq j\)
\[(a \cdot i)(i \varphi) = a(i j)(i \varphi) \cdot j\]
\[(a \cdot j)(i \varphi) = a(i \varphi) \cdot j\] if \(i \neq j\)

**Definition 3** (Typing judgements — à la Tarski).

\[
\begin{array}{c}
\text{Empty} & \text{NewVar} & \text{NewCol} \\
\hline
\Gamma \vdash & \Gamma, x : A \vdash & \Gamma, i : I \vdash \\
() \vdash & \Gamma, x : A \vdash & \Gamma, i : I \vdash \\
\end{array}
\]

\[
\begin{array}{c}
\text{Universe} & \text{Decode} & \text{Swap} & \text{Pi} & \text{Out} & \text{In-Pred} \\
\hline
\Gamma \vdash A : U & \Gamma, i : I, j : I, \Delta \vdash A & \Gamma \vdash A & \Gamma, x : A \vdash B \\
\Gamma \vdash U & \Gamma \vdash \text{El}(A) & \Gamma, j : I, i : I, \Delta \vdash A & \Gamma \vdash (x : A) \rightarrow B \\
\Gamma, i : I \vdash A & \Gamma \vdash a : A(i 0) & \Gamma, i : I \vdash (x : A) \times_i P \\
\Gamma \vdash A \ni a & \Gamma, x : A \vdash P & \\
\end{array}
\]
The parametricity constructions \( \cdot \) and \( \exists_i \) are color binders (they bring colors into scope), while the pairing constructs remove colors from scope. The equality relation used in the Conv rule is detailed below in Def. 5. The Swap rules allow us to use Out and Color-Elim with any free color, provided that no variable was introduced after that color (see e.g., Th. 12).

Additionally, for the above system to be well-founded, we need to distinguish small and big types, and allow only small types to be encoded in \( U \). Small types are closed under product, \( \times_i \) and \( \exists_i \). The distinction between big and small types being standard, and to keep the presentation concise, we leave it implicit in the syntax\(^1\).

**Theorem 4** (Color erasure and substitution preserve typing). If \( \Gamma, i : \mathbb{I} \vdash a : A \) then the terms \( a(i\varphi) \) and \( A(i\varphi) \) are defined and

- \( \Gamma \vdash a(i0) : A(i0) \), and
- \( \Gamma, j : \mathbb{I} \vdash a(ij) : A(ij) \).

**Proof.** By induction on the typing judgment. \( \square \)

\(^1\) Our rules are semantically justified in Sec. 4 on page 26; the use of codes enables a presentation à la Tarski, while avoiding us to split each constructor in two flavors, one for small types and one for large ones.
**Definition 5** (Conversion). The convertibility of types used in the \( \text{Conv} \) rule and written simply \( (=) \) is defined as the smallest reflexive-symmetric-transitive congruence containing the following rules.

- **Pair-Param**
  \[(a, i) \cdot i = p \]

- **Pair-App**
  \[\langle t, u \rangle a = (t a(i0), u a(i0)) (a \cdot i)\]

- **Pair-Pred**
  \[(x : A) \times i P[x] \ni a = P[a] \]

- **Surj-Param**
  \[t = (t(i0), t \cdot i)\]

- **Surj-Typ**
  \[T = (x : T(i0)) \times i (T(ij) \ni j x)\]

- **El**
  \[\text{El}(\mid A \mid) = A \quad |\text{El}(A)| = A \quad (\eta) \frac{t x = u}{t = \lambda x : A. u}\]

**Corollary 6** (**Surj-Fun**). \[t = \langle t(i0), \lambda xx', (t(x', x')) \cdot i \rangle\]

**Remark.** In order to be well-typed, any context for the conclusion of the **Pair-App**, **Surj-Param**, **Surj-Fun** and **Surj-Typ** rules needs to end with a color binding.

**Remark.** Although it looks as if \(\langle t, i u \rangle\) can be definable as \(\lambda x. (t x, u x x \cdot i)\), the latter rebinds \(i\), and does not allow us to prove parametricity for the Church-encoded naturals (Example 10) for instance.

Our conversion relation is intensional for functions, but extensional when it comes to dependencies on colors. Because there is at any point only a finite number of colors to consider, we conjecture that our conversion relation is decidable.

### 3 Parametricity

In this section we prove that our system properly internalizes unary parametricity; it could naturally be extended to the \(n\)-ary case by using further special symbols \(1, \ldots, n - 1\). We also illustrate the system by giving a few simple proofs relying on parametricity (including iterated parametricity). For the sake of readability, we leave out the distinction between types and their codes, which plays no role here.

Unlike previous type theories with internalized parametricity \([6, 7]\), the system presented here lacks equalities which allow to compute parametricity types. Expressed in our syntax, those equalities would become the conversion rules:

\[
U \ni a A = A \rightarrow U, \text{ and} \quad ((x : A) \rightarrow B[x]) \ni i f = (x : A) \rightarrow (x' : A \ni i x) \rightarrow B[(x, i x')] \ni i (f x).
\]

The absence of the above equalities allows for a simpler system, but how can we ensure that all parametricity theorems hold? The answer is that the above relationships hold as isomorphisms. We say that \(A\) is **isomorphic to** \(B\) iff:

1. There exist \(f : A \rightarrow B\), and
2. For any \(x\), \(f(gx) = x\), and
3. There exist \(g : B \rightarrow A\), and
4. For any \(x\), \(g(fx) = x\).
This notion of isomorphism is quite strong, because the equality used in its definition is the conversion relation (Def. 3).

**Theorem 7.** \( U 
\cong_i A \text{ is isomorphic to } A \rightarrow U. \)

**Proof.**

(i) \( f : (Q : U \cong_i A) \rightarrow A \rightarrow U \)
\[ f \ Q \ x = (A_i \ Q) \cong_i x \]

(ii) \( g : (P : A \rightarrow U) \rightarrow U \cong_i A \)
\[ g \ P = (\ (x : A) \times_i (P \ x) \ ) \cdot i \]

(iii) \( (A_i ((y : A) \times_i (P \ y)) \cdot i) \cong_i x = ((y : A) \times_i (P \ y)) \cong_i x = P \ x \) by \PAIR-PARAM then \PAIR-PRED, and we conclude by \( \eta \)-contraction.

(iv) \( ((x : A) \times_i (A_i \ Q) \cong_i x) \cdot i = (A_i \ Q) \cdot i = Q \) by \SURJ-TYP (indeed \( (x : A) \times_i (A_i \ Q) \cong_i x \) is typed in a context ending with \( i : \mathbb{I} \)) and \PAIR-PRED.

\( \square \)

**Theorem 8.** \( ((x : A) \rightarrow B[x]) \cong_i f \) is isomorphic to
\[(x : A) \rightarrow (x' : A \cong_i x) \rightarrow B[(x_i x')] \cong_i (f \ x) \]

**Proof.**

(i) \( f : (q : ((x : A) \rightarrow B[x]) \cong_i f) \rightarrow (x : A) \rightarrow (x' : A \cong_i x) \rightarrow B[(x_i x')] \cong_i (f \ x) \)
\[ f \ q \ x \ x' = ((f \ q) (x_i x')) \cdot i \]

(ii) \( g : ((x : A) \rightarrow (x' : A \cong_i x) \rightarrow B[(x_i x')] \cong_i (f \ x)) \rightarrow ((x : A) \rightarrow B[x]) \cong_i f \)
\[ g \ p = (f \ p) \cdot i \]

(iii) \( ((f \ p) (x_i x')) \cdot i = ((f \ p) (x_i x')) \cdot i = (f \ p \ x \ x') \cdot i = p \ x \ x' \) by \SURJ-PARAM then \PAIR-APP (indeed \( f \ p \) and \( f \ p \ (x_i x') \) are typed in a context ending with \( i : \mathbb{I} \)) and we conclude by \PAIR-PARAM.

(iv) \( (f, \ lambda x x'. ((f \ q) (x_i x')) \cdot i ; i) = (f, q) ; i = q \) by \SURJ-FUN (indeed \( f, q \) is typed in a context ending with \( i : \mathbb{I} \)) and we conclude by \PAIR-PARAM.

\( \square \)

In practice however, when carrying out parametricity proofs, many of the steps of the above isomorphisms cancel each other and one obtains a simpler proof. This behaviour is illustrated by the following examples: parametricity for the polymorphic identity and Church-encoded natural numbers.

**Example 9.** Any function \( f : (X : U) \rightarrow X \rightarrow X \) is the polymorphic identity, i.e., its output is Leibniz-equal to its second input. Assume a context
\[ \Gamma = (f : (X : U) \rightarrow X \rightarrow X, \ A : U, \ P : A \rightarrow U, \ a : A, \ p : P \ a) \]
Then \( \Gamma, i : \mathbb{I} \vdash (x : A) \times_i (P \ x) \) and by \PAIR-PRED \( \Gamma, i : \mathbb{I} \vdash (a, p) : (x : A) \times_i (P \ x) \), thus \( \Gamma, i : \mathbb{I} \vdash f ((x : A) \times_i (P \ x)) : (x : A) \rightarrow (P \ x) \) and finally
\[ \Gamma \vdash (f ((x : A) \times_i (P \ x)) (a, p)) \cdot i : ((x : A) \times_i (P \ x)) \cong_i (f ((x : A) \times_i (P \ x)) (a, p)) (i \ 0) \]
\[ = P (f ((x : A) \times_i (P \ x)) (a, p)) (i \ 0) = P (f A a) \]

23
Example 10. Let $N = (X : U) \to X \to (X \to X) \to X$. Proving (unary) parametricity for $N$ means that, assuming a context $\Gamma$

$$f : N, \ A : U, \ P : A \to U, \ z : A, \ z' : P z, \ s : A \to A, \ s' : (x : A) \to P x \to P (s x),$$

we can prove $P (f A z s)$.

Indeed $\Gamma, i : \mathbb{I} \vdash (x : A) \times_i (P x)$, and by Pair-Pred $\Gamma, i : \mathbb{I} \vdash (z, i z') : (x : A) \times_i (P x)$ and $\Gamma, i : \mathbb{I} \vdash (s, i s') : (x : A) \times_i (P x)$, thus

$$\Gamma, i : \mathbb{I} \vdash f ((x : A) \times_i (P x)) (z, i z') (s, i s') : (x : A) \times_i (P x),$$

and finally

$$\Gamma \vdash (f ((x : A) \times_i (P x)) (z, i z') (s, i s')) \cdot i : ((x : A) \times_i (P x)) \supset_i (f A z s) = P (f A z s)$$

As seen in Example 10, one needs to use $\langle t, u \rangle$ to pair a function with the parametricity proof of its type if one wants to apply that to pair and reduce the application. This is because as noted above, our system does not support direct computation of free theorems: in particular $(A \to B) \supset_i a$ does not reduce.

At this point one may wonder, since a new syntactic construction was introduced for function types, whether yet another construction is required for higher order functions. This objection was preemptively refuted by Th. 8: it turns out that $\langle t, u \rangle$ can be combined with $(a, p)$ to pair higher order functions with the parametricity proof of their type. The following example illustrates this technique:

Example 11. Let $F = (X : U) \to ((X \to X) \to X) \to X$. Proving (unary) parametricity for $F$ means that, assuming a context $\Gamma = f : F, \ A : U, \ P : A \to U, \ g : (A \to A) \to A, \ g' : (h : A \to A) \to ((x : A) \to P x \to P (h x)) \to P (g h)$, we can prove $P (f A g)$.

Let $T = (x : A) \times_i (P x)$. We have $\Gamma, i : \mathbb{I} \vdash T$ and

$$\Gamma, h : A \to A, h' : (T \to T) \supset_i h, x : A, x' : P x, i : \mathbb{I} \vdash (h, i h') : T \to T,$$

hence

$$\Gamma, h : A \to A, h' : (T \to T) \supset_i h, x : A, x' : P x \vdash ((h, i h') (x, i x')) \cdot i : T \supset_i h x = P (h x)$$

$$\Gamma, h : A \to A, h' : (T \to T) \supset_i h \vdash g' h (\lambda (x : A). \lambda (x' : P x). ((h, i h') (x, i x')) \cdot i : P (g h))$$

Let $g'' = \lambda h. \lambda h'. g' h \lambda (x : A). \lambda (x' : P x). ((h, i h') (x, i x')) \cdot i$. Since we have $\Gamma \vdash g'' : (h : A \to A) \to (T \to T) \supset h \to P (g h)$ we can pair it with $g$ and $\Gamma, i : \mathbb{I} \vdash \langle g, i g'' \rangle : (T \to T) \to T$. We can finally conclude as before, that $\Gamma \vdash (f T \langle g, i g'' \rangle) i : P (f A g)$.

3.1 Iterating Parametricity

In our system, one can use parametricity generically as follows:

$$p : (X : U) \to (x : X) \to X \supset_i x$$

$$p X x = x \cdot i$$

We have already seen that $A \supset_i$ corresponds to a parametricity predicate for the type $A$. As we hinted at in the introduction, the color index $i$ allows us to distinguish each application of parametricity. (As a side remark, since the COLOR-ELIM rule introduces a color, limiting the depth of nested applications of parametricity can trivially be enforced in our system by limiting the number of
free colors in the context.) We can iterate the operator $A \ni \cdot$ to construct relations between parametricity witnesses. That is, given a context with
\[ x : A, \ y : A \ni x, \ z : A \ni x, \]
the type $A \ni (x, y) \ni z$ is well formed ($\ni$ is left associative), and can be understood as a binary relation between the parametricity proofs $y$ and $z$. The following results about this relation illustrate the expressivity of our system.

**Theorem 12.** If the type $A$ does not depend on either $i$ or $j$, the relation $\lambda yz.A \ni (x, y) \ni z$ is symmetric.

*Proof.* We first construct the proof term:
\[
\sigma_1 : (x : A) \rightarrow (y : A \ni x) \rightarrow (z : A \ni x) \rightarrow A \ni (x, y) \ni z \rightarrow A \ni (x, z) \ni y
\]
\[
\sigma_1 x y z w = (((x, y), i) (z, w)) \cdot j \cdot i
\]
And, by $\alpha$-equivalence on colors, $A \ni (x, i) \ni z \ni y = A \ni (x, j) \ni z \ni y$. □

**Theorem 13.** If the type $A$ does not depend on either $i$ or $j$, then the types $A \ni (x, y) \ni z$ and $A \ni (x, i) \ni z$ are isomorphic.

*Proof.* We show that $\sigma_1 x y z (\sigma_1 x y z w) = w$. Let $t = ((x, j), i) (z, w))$, $w' = t \cdot j \cdot i$, $t' = ((x, i) z, y) (w')$. Then $t'(i, 0) = (x, j) = t(i, 0)$, $t'(j, 0) = (x, i) z = t(j, 0)$, and $(t \cdot j)(i, 0) = y$. We now continue to reason by deduction:
\[
w' = t \cdot j \cdot i \quad \text{by def.}
\]
\[
(y, i, w') = t \cdot j
\]
\[
t' \cdot j = t \cdot j \quad \text{because } (t \cdot j)(i, 0) = y
\]
\[
t' = t \quad \text{by def.}
\]
\[
t' = ((x, j) i (z, w)) \quad \text{by def.}
\]
\[
t' \cdot i = (z, i w)
\]
\[
t' \cdot j = = w \quad \text{□}
\]

**Remark.** At this point one may wonder if the system could have been set up to have $t \cdot i \cdot j = t \cdot j \cdot i$, and the equality between $A \ni (x, j) \ni z$ and $A \ni (x, i) \ni z$ rather than an isomorphism. The answer is that the equation
\[ A \ni (x, j) \ni z = A \ni (x, i) \ni z \ni y \]
is inconsistent: in particular for $A = U$ one gets
\[ U \ni (X, j) P \ni j Q = U \ni (X, i) Q \ni i P \]
for arbitrary $P$ and $Q$ of type $U \ni X$. The above equality in turn implies
\[ (x : X) \rightarrow P x \rightarrow Q x \rightarrow U = (x : X) \rightarrow Q x \rightarrow P x \rightarrow U \]
for arbitrary predicates $P$ and $Q$ over $X$, which is obviously inconsistent.

**Theorem 14.** If the type $A$ and the term $a$ do not depend on either $i$ or $j$, and $a' : A \ni a (not \ depending \ on \ i \ or \ j \ either)$, then $A \ni (a, j, a \cdot i) \ni j a'$. 25
Proof. We can construct the following closed term:

\[ q : (A : U) \rightarrow (x : A) \rightarrow (x' : A \ni x) \rightarrow A \ni (x_{i,j} x \cdot i) \ni_j x' \]

\[ q : (A : U) \rightarrow (x : A) \rightarrow (x' : A \ni x) \rightarrow A \ni x \ni_j x' \text{ by SURJ-PARAM} \]

The result is then obtained by substituting \( a \) for \( x \) and \( a' \) for \( x' \).

To conclude the section we note that by iterating parametricity \( n \) times, one creates \( n \)-ary relations between proofs of relations of arity \( n - 1 \). Furthermore, the above results carry over to the \( n \)-ary case. That is, for each \( k < n \), one can construct a function \( \sigma_k \), which exchanges the arguments \( k \) and \( k + 1 \) of a relation. Furthermore, these functions satisfy the laws of the generators of the symmetric group.

4 Presheaf model

In this section we show how to interpret our type theory by a presheaf model.

Definition 15. If \( I \) and \( J \) are two finite subsets of \( \mathbb{I} \), we call a color map any function \( f : I \rightarrow J \cup \{0\} \) such that \( i_1 = i_2 \) for any \( i_1, i_2 \in I \) with \( f(i_1) = f(i_2) \in J \).

Definition 16 (Category \( \mathbf{pI} \)). Let objects be finite color sets and morphisms be color maps (a.k.a. partial injections; the Hom-set \( I \rightarrow J \) denotes functions \( I \rightarrow J \cup \{0\} \)). If \( f : I \rightarrow J \) and \( g : J \rightarrow K \), we define the composition as the Kleisli one: \( fg : I \rightarrow K \) as \( fg(i) = 0 \) if \( f(i) = 0 \) and \( fg(i) = g(f(i)) \) if \( f(i) \in J \). We write \( 1_I : I \rightarrow I \) for the identity map. It is easy to check that \( \mathbf{pI} \) is a category (see [14, ex. 9.7 p. 176] for another description of this category).

If \( f : I \rightarrow J, i \notin I \) and \( j \notin J \), let \( (f, i = j) : I, i \rightarrow J, j \) (where \( I, i \) is a shorthand for \( I \cup \{i\} \)) denote the map defined by \((f, i = j)(i) = j \) and \((f, i = j)(k) = f(k) \) for every \( k \in I \).

If \( f : I, i \rightarrow J \) (resp. \( f : I, i \rightarrow J, j \)) is such that \( f(i) = 0 \) (resp. \( f(i) = j \)), let \( f - i : I \rightarrow J \) denote the map defined by \((f - i)(k) = f(k) \) for every \( k \in I \).

For any object \( I \) and \( i \notin I \), let \( \iota_i : I \rightarrow I, i \) denote the inclusion map, defined by \( \iota_i(k) = k \) for every \( k \in I \).

Definition 17 (Projection). We say that a morphism \( \alpha : I \rightarrow I_\alpha \) is a projection if \( I_\alpha \subseteq I \), \( \alpha(i) = 0 \) for each \( i \in I \setminus I_\alpha \), and \( \alpha(i) = i \) for each \( i \in I_\alpha \).

Definition 18 (Total maps). We say that a morphism \( h : I \rightarrow J \) is total, and note \( h : I \rightarrow J \), if it is injective, i.e., if \( h(i) \neq 0 \) for each \( i \in I \).

Remark (Morphism decomposition). Any morphism \( f : I \rightarrow J \) has a unique decomposition into a projection map \( \alpha : I \rightarrow I_\alpha \) and a total map \( h : I_\alpha \rightarrow J \).

Definition 19 (I-set). Let an I-element be any tuple indexed by the subsets of \( I \): \( (u_{i,j})_{i \subseteq I} \). An I-set is a set of I-elements. For instance, the elements of an \( \{i,j\} \)-set are of the form \( u = (u_{\emptyset}, u_i, u_j, u_{i,j}) \). Alternatively, such an element can be seen as a tuple \( (u_{i_{\alpha}}) \) indexed by the projections \( \alpha : I \rightarrow I_\alpha \).
If \(a, b\) are \(I\)-elements and \(j \not\in I\), we define the \((I, j)\)-element \((a, j, b)\) as \((a, j, b)_{I} := a_{I}\) if \(j \not\in J\) and \((a, j, b)_{J} := b_{J}\). Any \((I, i)\)-element can be written \(u = (u_{J})_{J \subseteq I, i} = (u_{J})_{J \subseteq I} \cup (u_{J,i})_{J \subseteq I};\) We can therefore define the \(I\)-elements \(u(i0) := (u_{J})_{J \subseteq I}\) and \(u \cdot i := (u_{J,i})_{J \subseteq I}\). (Hence by definition \(u = (u(i0), u \cdot i).\))

Recall that a presheaf \(F\) on \(\mathbf{pI}^{op}\) is given by a family of sets \(F(I)\) together with restriction maps \(F(I) \to F(J), u \mapsto uf\) for \(f : I \to J\) satisfying \(u_{I} = u\) and \((ufg) = u(fg)\). (Note that the category of presheaves on \(\mathbf{pI}^{op}\) is equivalent to the category \(\text{Res}\) of nominal restriction sets \([14, \text{rem. 9.9 p. 161}].\)) We use a refined presheaf on \(\mathbf{pI}^{op}\) by requiring two further conditions:

(i) for any object \(I, F(I)\) is an \(I\)-set; and

(ii) for any projection map \(\alpha : I \to I_{a}\), the restriction map \(F(I) \to F(I_{a}), u \mapsto u\alpha\) is the projection operation, i.e., \(u\alpha_{J} = u_{J}\) for any \(J \subseteq I\) (alternatively, seeing \(I\)-elements as tuples indexed by projection maps, \((u\alpha)_{\beta} = u_{\alpha\beta}\)).

Unless written otherwise, any presheaf in the remainder of this section is assumed to satisfy these conditions. The refinement is necessary for the interpretation of some of our syntactic constructions. Indeed, without it, it is not clear how to validate the equality \(\text{Pair-Pred}: ((x : A) \times_{I} P[x]) \ni a = P[a]\).

A context \(\Gamma \vdash\) is interpreted by a (non-refined) presheaf on \(\mathbf{pI}^{op}\), i.e., by a family of sets \(\Gamma(I)\) for each object \(I\), together with restriction maps \(\Gamma(I) \to \Gamma(J), \rho \mapsto \rho f\) for \(f : I \to J\) satisfying the conditions \(\rho 1 = \rho\) and \((\rho f)g = \rho(fg)\).

A type \(\Gamma \vdash A\) is interpreted by an \(I\)-set \(A\rho\) for each object \(I\) and \(\rho \in \Gamma(I)\), together with restriction maps \(A\rho \to A(\rho f), u \mapsto uf\) if \(f : I \to J\) satisfying \(u_{I} = u\) and \((uf)g = u(fg)\) for any \(g : J \to K\). Furthermore the map \(A\rho \to A(\rho a), u \mapsto ua\) is the projection operation.

A term \(\Gamma \vdash a : A\) is interpreted by an \(I\)-element \(a\rho \in A\rho\) for each object \(I\) and \(\rho \in \Gamma(I)\), such that \(a\rho f = a(\rho f)\) for any \(f : I \to J\).

If \(\Gamma \vdash\) and \(\Gamma \vdash A\) we define the interpretation of \(\Delta = (\Gamma, x : A)\) by taking \(\langle \rho, x = u \rangle \in \Delta(I)\) to mean \(\rho \in \Gamma(I)\) and \(u \in A\rho\). The restriction map is defined by \(\langle \rho, x = u \rangle f = \langle \rho f, x = uf \rangle\).

If \(\Gamma \vdash\) we define the interpretation of \(\Delta = (\Gamma, i : \mathbb{I})\) by taking \([\rho, i = \varphi] \in \Delta(I)\) to mean either \(\varphi = 0\) and \(\rho \in \Gamma(I)\), or \(\varphi = j \in I\) and \(\rho \in \Gamma(I \setminus \{j\})\). The restriction map is defined by \([\rho, i = 0] f = [\rho f, i = 0] \) and \([\rho, i = j] f = [\rho f - j, i = f(j)]\).

**Remark.** In other words, \(\Gamma, x : A\vdash\) is interpreted by the cartesian product \((\rho \in \Gamma) \times A\rho\) of the interpretations of \(\Gamma \vdash\) and \(\Gamma \vdash A\), while \(\Gamma, i : \mathbb{I} \vdash\) is interpreted by the separated product \([14, \text{sec. 3.4 p. 54}]\) \(\Gamma \ast \mathbb{I}\) of the interpretation of \(\Gamma \vdash\) and \(\mathbb{I} \cup \{0\}\):

\[
\Gamma \ast \mathbb{I}(I) = \{[\rho, i = 0] \mid \rho \in \Gamma(I)\} \cup \{[\rho, i = j] \mid j \in I, \rho \in \Gamma(I \setminus \{j\})\}
\]

We also note that \(\Gamma, i : \mathbb{I}, j : \mathbb{I} \vdash\) and \(\Gamma, j : \mathbb{I}, i : \mathbb{I} \vdash\) are respectively interpreted as the sets of \([\rho, i = \varphi, j = \varphi']\) and \([\rho, j = \varphi, i = \varphi']\), which are trivially isomorphic.

The semantics we define satisfy the substitution law. That is, if \(\Gamma, x : A \vdash B\) and \(\Gamma \vdash a : A\) then for any \(\rho \in \Gamma(I)\) we have \(B[a] \rho = B(\rho, x = a\rho)\). It also satisfies
In-Pred.

Assume \( \Pi. \) Assume \( \text{Universe.} \) The universe \( (\Pi. \) Assume \( \text{we have} \) \( A \) the substitution law on colors, \( \text{i.e., if} \Gamma, i : I \vdash A \) then for any \( \rho \in \Gamma(I) \) and \( j \not\in I \) we have \( A(i(0))\rho = A[\rho, i = 0] = A[\rho, i = j](j, 0). \) (Since \( [\rho, i = 0] \in \Gamma \ast \Pi(I) \) and \( [\rho, i = j] \in \Gamma \ast \Pi(I, j), \) \( A(i(0))\rho \) and \( A[\rho, i = 0] \) are \( I \)-sets while \( A[\rho, i = j] \) is a \( (I, j) \)-set.) For establishing these properties, we proceed as Aczel [1].

We proceed to interpret each type construction.

(1) Assume \( \rho \in \Gamma(I). \) We define \( ((x : A) \rightarrow B)\rho \) as a \( I \)-set. An \( I \)-element of \( ((x : A) \rightarrow B)\rho \) is defined as a tuple \( \lambda = (\lambda_\alpha), \) where each \( \lambda_\alpha \) is a family of elements indexed by a total map \( f : I_\alpha \rightarrow J: \)

\[
\lambda_\alpha f \in \prod_{u \in A(\rho \circ f)} B(\rho f, x = u)
\]

such that \( \text{app}(\lambda_\alpha f, u)g = \text{app}(\lambda_\alpha fg, ug) \) for \( f : I_\alpha \rightarrow J \) total and for any \( g : J \rightarrow K \) (where \( \text{app} \) is the semantic application). Because any map \( I \rightarrow J \) has an unique decomposition as a projection and a total map, we can consider \( \lambda_f \) for an arbitrary map \( f : I \rightarrow J. \)

If \( f : I \rightarrow J \) is an arbitrary map, we define \( \lambda f \) to be the tuple \( (\lambda f_{\beta_\gamma}) \) where \( \lambda f_{\beta_\gamma} \) is the family \( \lambda f_{\beta_\gamma} = \lambda f_{\gamma} \beta. \) With this definition, we directly have \( \lambda_\alpha \beta = \lambda_\alpha_\beta. \)

This is similar to the usual interpretation of dependent product in presheaf models [10, 12]; but to satisfy our first extra condition on presheaves we present each element as a tuple, which can be done naturally by repartitioning the family as follows: \( (\lambda f)_f : I \rightarrow J = (\lambda \alpha_{\alpha_f})_{I \subseteq I, g : I_\alpha \rightarrow J} = (g : I_\alpha \rightarrow J). \)

(2) \text{Universe.} \) The universe \( U \) is interpreted as a presheaf over \( \text{pI.} \) An element \( A \) of \( U(I) \) is a tuple \( (A_\alpha) \) where each \( A_\alpha \) is a family \( (A_\alpha f) \) of \( U \)-small sets (where \( U \) is a fixed Grothendieck universe) indexed by \( f : I_\alpha \rightarrow J \) total together with restriction maps \( A_{\alpha f} \rightarrow A_{\alpha f g}, u \rightarrow ug \) for \( f : I_\alpha \rightarrow J \) total and \( g : J \rightarrow K \) arbitrary, such that \( u1 = u \) and \( (ug)h = u(gh). \)

As before, such data define a set \( A_f \) for an arbitrary map \( f : I \rightarrow J \) with restriction maps \( A_f \rightarrow A_{fg} \) if \( g : J \rightarrow K. \)

If \( f : I \rightarrow J \) is an arbitrary map, we define \( A_f \) by taking \( A_{f \beta g} \) to be the set \( A_{f \beta g} \), together with restriction maps \( A_{f \beta g} \rightarrow A_{fg \beta h} \) defined as the given maps \( A_{f \beta g} \rightarrow A_{fg \beta h}. \) We can then check, as before, that we have \( A_{\alpha \beta} = A_{\alpha \beta}. \)

As before, this is similar to the usual interpretation of universe in presheaf models, where each element is presented as a tuple.

(3) Assume \( \rho \in \Gamma(I). \) We need to define the \( I \)-set \( (A \ni a)\rho. \) Let \( j = \text{fresh}(I). \)

We get a \( (I, j) \)-set \( A[\rho, i = j] \), and the \( I \)-element \( a \rho \) belongs to \( A(\rho, i = 0) = A[\rho, i = j](j, 0). \)

We define \( (A \ni a)\rho \) to be the set of \( I \)-elements \( v \) such that \( (a \rho, j) v \in A[\rho, i = j]. \) If \( v \) is such an element and \( f : I \rightarrow J \) and \( k = \text{fresh}(J), \) then \( vf \) is defined by the equation \( (apf_k \circ v) f(j, k). \)

(4) In-Pred. Assume \( [\rho, i = \varphi] \in \Gamma \ast \Pi(I). \) We define the \( I \)-set \( ((x : A) \times_i P)[\rho, i = \varphi] \) by case analysis on \( \varphi \in I \cup \{0\}. \) If \( \varphi = 0 \) then \( \rho \in \Gamma(I), \) and we define \( ((x : A) \times_i P)[\rho, i = 0] \) as the \( I \)-set \( Ap. \) If \( \varphi = j \in I \) then \( \rho \in \Gamma(I \setminus \{j\}), \) and we define \( ((x : A) \times_i P)[\rho, i = j] \) as the \( I \)-set of \( (u, j) v \) where \( u \in Ap \) and
Theorem 20 (Convertible terms are semantically equal)

Proof. By simultaneous induction on the derivation. We only show the conversion rules Pair-Param, Pair-Pred and Surj-Param here; other rules involving colors can be proven in a similar fashion, while \( \beta \) and \( \eta \) can be proven in the usual way.

**Pair-Param.** Let \( \rho \in \Gamma(I) \) and \( j = \text{fresh}(I) \). We have

\[
v \in \mathcal{P}(\rho, x = u).
\]

**Pair-Pred.** Let \( \rho \in \Gamma(I) \) and \( j = \text{fresh}(I) \). We have \((a,i)p\cdot i = (a,i)p\cdot i =\)
For each \( \rho \in \Gamma(I) \) we have \( (t(i0) t \cdot i)[\rho, i = 0] = t[i0] \rho = t[\rho, i = 0] \), and if \( j \not\in I \) then \( (t(i0) t \cdot i)[\rho, i = j] = (t[i0] \rho_j (t \cdot i) \rho) = (t[\rho, i = j][j0] \rho_j t[\rho, i = j] \cdot j) = t[\rho, i = j] \). Hence \( (t(i0) \cdot) \rho = t \rho \) for any \( \rho \in \Gamma \). □

**Remark.** As noted earlier, the types \( U \ni (X_{ij} P) \ni Q \) and \( U \ni (X_{iQ}) \ni P \) are not convertible. Their semantic interpretations are not equal either. Indeed taking \( \rho \in \Gamma(I), k = \text{fresh}(I) \) and \( l = \text{fresh}(I, k) \), we have (leaving out the context interpretation \( \rho \) for the sake of readability) on the one hand

\[
v \in (U \ni (X_{ij} P) \ni Q) \rho
\]

iff. \( ((X_{ij} P)[\rho, j = k]\rho_a, (P_{\rho, k} v)\rho_a) \in U(l, k)\)

while on the other hand

\[
v \in (U \ni (X_{iQ}) \ni P) \rho
\]

iff. \( ((X_{iQ})[\rho, i = k]\rho_a, (P_{\rho, k} v)\rho_a) \in U(k, l)\)

hence \( (U \ni (X_{ij} P) \ni Q) \rho \neq (U \ni (X_{iQ}) \ni P) \rho \) since the map \( U(l, k) \to U(k, l) \), \( u \mapsto ug \) where \( g(k) = l \) and \( g(l) = k \) is not the identity.

**Theorem 21** (Validity). If \( \Gamma \vdash a : A \) then \( a \rho \in A \rho \) for any \( \rho \in \Gamma(I) \).

**Proof.** By induction on the typing judgment. We only show the cases \( \text{In-Abs} \) and \( \text{Color-Elim} \). \( \text{In-Fun} \) is similar to the former, and the other cases match the usual proof (using Th. 20 for \( \text{Conv} \)).

**In-Abs.** Assume \( [\rho, i = \varphi] \in \Gamma \ast I(I) \). We proceed by case analysis on \( \varphi \in I \cup \{0\} \). If \( \varphi = 0 \) then \( \rho \in \Gamma(I) \), and we have \( (a_i p)[\rho, i = 0] = a \rho \in A(i0) \rho = A[\rho, i = 0] \).

If \( \varphi = j \in I \) then \( \rho \in \Gamma(I, \{j\}) \), and we have \( (a_i p)[\rho, i = j] = (a \rho_j p) \rho \).

Since by induction hypothesis \( p \rho_a \in (A \ni a) \rho \), we conclude by definition that \( (a \rho_j p) \rho_a \in A[\rho, i = j] \).

**Color-Elim.** Assume \( \rho \in \Gamma(I) \). We need to show that \( (a \cdot i) \rho \in (A \ni a(i0)) \rho \), i.e., that \( (a(i0) \rho_j (a \cdot i) \rho) \in A[\rho, i = j] \) where \( j = \text{fresh}(I) \). By induction hypothesis \( a[\rho, i = j] \in A[\rho, i = j] \), hence we have \( (a(i0) \rho_j (a \cdot i) \rho) = (a[\rho, i = j][j0] \rho_j a[\rho, i = j] \cdot j = a \in A[\rho, i = j] \). □

5 Related Work

Our own line of work

This work continues a line of work aiming at a smooth integration of parametricity with dependent types [5–9]. The present work offers two improvements over previous publications: 1. a denotational semantics, and 2. a much simplified syntax, suitable as the basis of a proof assistant.
The simplification of syntax is allowed by not requiring the preservation of functions by parametricity. We call preservation of functions by parametricity the property that if \( f \) were a function, then the canonical proof that \( f \) is parametric (denoted \( f \cdot i \) here) is also a function. To our knowledge, following Reynolds [17], all parametric models of parametricity (both syntactical and semantical ones) have this property. However, having this property in the syntax implies that certain function arguments must be swapped when performing the substitution of beta reduction, as identified by Bernardy and Moulin [6]. In the present system, the parametric interpretation of functions is instead merely isomorphic to a function, thanks to the \textsc{in-fun} rule (Th. 8). This isomorphism (rather than equality) means on the one hand that the swapping of arguments is handled by the usual rules of logic, instead of special-purpose ones. On the other hand, obtaining the usual parametric interpretation of types requires some purely mechanical work by the user of the logic.

**Parametric Models of Type Theory vs. Parametric Type Theories**

Two pieces of work propose alternative parametric models of type theory [4, 13], but do not integrate parametricity in the syntax of the calculus. This means that, while certain consequences of parametricity can be made available in the logic (e.g., via constants validated by the model), parametricity itself is not available. In this paper, we not only propose a parametric model, but also show how it can be used to interpret parametricity in the syntax of the type theory.

**Various kinds of models**

Another characterizing feature of proposals for parametricity is the kind of model underlying the semantics. Krishnaswami and Dreyer [13] propose a model based on Q-PER. Atkey et al. [4] propose a model based on reflexive graphs. The model that we use is based on cubes (functions from subsets of colors). In Bernardy and Moulin [6] the cubes were reified as syntax in an underlying calculus, while in the present work they refine a presheaf structure.

**Presheaf models**

The presheaf construction used in this paper follows a known template, used for example by Bezem et al. [10] and Pitts [15] to model univalence in type theory. Not only do both models use a presheaf, but they also use a category closely connected to the underlying category \( \mathbf{pI} \). This means that all these models have an additional cubical structure. We think that it is remarkable that cubical structures are useful for modeling both parametricity and univalence. Altenkirch and Kaposi [2] give a syntax for Bezem et al.’s Cubical Type Theory, effectively modelling univalence by internalization of their model. The present work further refines the model by interpreting terms as \( I \)-elements, which is essential to interpret our special-purpose pairing constructions.
6 Future work and conclusion

We have defined a new type theory with internalized parametricity. Thanks to our model construction, we have proved the consistency of the system. The missing piece to construct a type-checker is a decision algorithm for the conversion relation. This checker could then be used as a minimal proof assistant for a type theory with parametricity.

Acknowledgments: The fact that the category of partial bijections $pI$ should be relevant for internalization of parametricity became apparent through discussions between Thorsten Altenkirch and the second author about the paper [6].

We are extremely grateful to Simon Huber for finding a flaw in an earlier version of this paper, and helping us to fix it. We also thank Peter Dybjer, Patrik Jansson, Andrea Vezzosi and the anonymous referees for valuable feedback and discussion.

References


Healthiness Conditions for Predicate Transformers

Klaus Keimel 1,2

Fachbereich Mathematik
Technische Universität Darmstadt
64289 Darmstadt, Germany

Abstract

The behavior of a program can be modeled by describing how it transforms input states to output states, the state transformer semantics. Alternatively, for verification purposes one is interested in a 'predicate transformer semantics' which, for every condition on the output, yields the weakest precondition on the input that guarantees the desired property for the output.

In the presence of computational effects like nondeterministic or probabilistic choice, a computation will be modeled by a map $t: X \rightarrow T Y$, where $T$ is an appropriate computational monad. The corresponding predicate transformer assigns predicates on $Y$ to predicates on $X$. One looks for necessary and, if possible, sufficient conditions (healthiness conditions) on predicate transformers that correspond to state transformers $t: X \rightarrow T Y$.

In this paper we propose a framework for establishing healthiness conditions for predicate transformers. As far as the author knows, it fits to almost all situations in which healthiness conditions for predicate transformers have been worked out. It may serve as a guideline for finding new results.

Keywords: predicate transformers, healthiness conditions, continuation monad, entropic algebras

1 Introduction: An example

In denotational semantics we distinguish two complementary approaches that we shortly call state transformer semantics and predicate transformer semantics. Let us begin with the well-known example of angelic nondeterminism to explain our intentions. As semantic domains we will use directed complete partially ordered sets (dcpos), maps will be Scott-continuous, that is, they preserve the order and suprema of directed subsets.
In the presence of nondeterministic choice, running a program for an input \( x \) belonging to a domain \( X \) will lead to a set \( t(x) \) of possible outputs in a domain \( Y \). In the angelic interpretation of nondeterminism, \( t(x) \) will be a non-empty Scott-closed subset of the dcpo \( Y \) and \( t \) will be a Scott-continuous map from the dcpo \( X \) to the Hoare powerdomain \( \mathcal{H}Y \) of all nonempty Scott-closed subsets of \( Y \). The binary choice operator is interpreted by union on the Hoare powerdomain. Thus, a program will be interpreted by a state transformer, a Scott-continuous map \( t: X \to \mathcal{H}Y \).

Observable predicates on \( Y \) are Scott-open \( U \) subsets of \( Y \) (see, e.g., [25]). Thus, the complete lattice \( \mathcal{O}Y \) of all Scott-open subsets of \( Y \) represents the dcpo of predicates on \( Y \). A Scott-continuous map \( p: \mathcal{O}Y \to \mathcal{O}X \) transforming predicates on \( Y \) to predicates on \( X \) will be a predicate transformer.

To a state transformer \( t: X \to \mathcal{H}Y \) we associate the predicate transformer \( p: \mathcal{O}Y \to \mathcal{O}X \) defined by

\[
p(U) = \{ x \in X \mid t(x) \cap U \neq \emptyset \}
\]

the set of all points in \( X \) that lead to at least one output with the desired property \( U \) (the angelic point of view). The state transformer \( t \) can be recovered from the associated predicate transformer \( p \) by

\[
t(x) = \bigcap \{ Y \setminus U \mid x \notin p(U) \}
\]

We are concerned with the problem to find properties (healthiness conditions) that characterize those predicate transformers \( p: \mathcal{O}Y \to \mathcal{O}X \) that correspond to state transformers \( t: X \to \mathcal{H}Y \). The answer in this case is:

*The predicate transformers \( p: \mathcal{O}Y \to \mathcal{O}X \) that correspond to state transformers \( t: X \to \mathcal{H}Y \) are characterized by the properties:*

\[
p(\emptyset) = \emptyset, \quad p(U \cup U') = p(U) \cup p(U')
\]

*Equivalently, these are the maps \( p \) preserving arbitrary unions.*

The above considerations become more elegant, but less intuitive, by passing to a functional setting. We use that the category \text{DCPO} of dcpos and Scott-continuous maps is Cartesian closed. The exponential of two dcpos \( X \) and \( Y \), denoted by \( Y^X \) and equally by \( [X \to Y] \), is the dcpo of all Scott-continuous maps \( u: X \to Y \) with the pointwise defined order (one may consult [3] for background on dcpos).

We endow the two element domain \( 2 = \{ 0 < 1 \} \) with the structure of a unital semilattice with \( x \vee y = \max(x, y) \) and the constant (unit) \( 0 \). A predicate (a Scott-open subset \( U \)) of a dcpo \( Y \) is identified with the Scott-continuous map \( f_U: Y \to 2 \) with value 1 iff \( x \in U \). Thus the dcpo \( \mathcal{O}Y \) of predicates is identified with the function space \( 2^Y \). This function space is also a unital semilattice when equipped.
with the pointwise defined operation $\lor$ and the constant function $0$. The Scott-
continuous unital semilattice homomorphisms $\varphi: 2^Y \to 2$ form a dcpo $[2^Y \lor 0, 2]$ 
which is also a unital semilattice for the pointwise defined semilattice operations.
We will use that it is isomorphic to the Hoare powerdomain $H_Y$; indeed, these 
homomorphisms $\varphi$ correspond to the Scott-closed subsets of $Y$ by assigning to $\varphi$
the Scott-closed set $C = Y \setminus \bigcup\{U \in \mathcal{O}Y \mid \varphi(f_U) = 0\}$.

Now a state transformer will be a Scott-continuous map $t: X \to [2^Y \lor 0, 2]$ and a 
predicate transformer a Scott-continuous map $p: 2^Y \to 2^X$. For a state transformer 
$t$ the corresponding predicate transformer $p$ is given by 
\[
p(g)(x) = t(x)(g) \quad \text{for all } g \in [2^Y \lor 0, 2], x \in X
\]
We can recover $t$ from $p$ by reading this equation from right to left. For a state 
transformer $t: X \to [2^Y \lor 0, 2]$ the corresponding predicate transformer $p$ is a unital 
$\lor$-homomorphism. Indeed, using that $t(x)$ is a unital $\lor$-homomorphism for every 
$x \in X$, we have $p(0)(x) = t(x)(0) = 0$ and $p(g \lor g')(x) = t(x)(g \lor g') = t(x)(g) \lor 
t(x)(g') = p(g)(x) \lor p(g')(x) = (p(g) \lor p(g'))(x)$ for all $x \in X$, whence $p(0) = 0$
and $p(g \lor g') = p(g) \lor p(g')$. Conversely, a similar calculation shows that, given 
a Scott-continuous unital $\lor$-semilattice homomorphism $p: 2^Y \to 2^X$, the map $t(x)$ 
defined by 
\[
t(x)(g) = p(g)(x) \quad \text{for all } g \in 2^Y
\]
is indeed a unital $\lor$-homomorphism for every $x \in X$ and $t: X \to [2^Y \lor 0, 2]$ is a 
state transformer. Thus:

The predicate transformers $p: 2^Y \to 2^X$ corresponding to state transformers 
t: $X \to [2^Y \lor 0, 2]$ are characterized by the properties 
\[
p(0) = 0, \quad p(g \lor g') = p(g) \lor p(g')
\]
This functional approach is easily generalized to other situations. As above, in most 
applications an essential step will be the translation of the given situation into a 
functional setting.

2 The problem

We work in the category DCPO of dcpos and Scott-continuous functions, a category 
which is commonly used in semantics. We agree on some assumptions that will be 
tacitly assumed throughout the paper.

Convention 2.1 All maps between dcpos will tacitly supposed to be Scott-
continuous. All definitions of functions are expressible in the language of typed 
$\lambda$-calculus so that they are automatically Scott-continuous, since we are in a Carte-
sian closed category (see, for example, [19, Part I]). For this reason, we will never
verify continuity of functions explicitly.

\( R \) will be a fixed dcpo, called the dcpo of observations; 
\( X \) and \( Y \) denote arbitrary dcpos; 
\( x, y \) and \( r \) denote elements of \( X, Y \) and \( R \), respectively; 
\( u \) denotes Scott-continuous maps \( u: X \to Y \); 
\( f \) and \( g \) denote Scott-continuous maps \( f: X \to R \) and \( g: Y \to R \); 
\( \varphi \) denotes Scott-continuous maps \( \varphi: R^X \to R \).

Predicates on a dcpo \( Y \) will be \( R \)-valued\(^3\), that is, Scott-continuous functions \( g: Y \to R \). The function space (exponential) for which we will use two notations in parallel, \( R^Y = [Y \to R] \), will be the domain of predicates on \( Y \).

The contravariant functor \( R(-) \) assigns to every dcpo \( X \) the exponential \( R^X \) and to every map \( u: X \to Y \) of dcpos the map \( R^u: R^Y \to R^X \) defined by \( R^u(g) = g \circ u \) for all \( g \in R^Y \). Applying the contravariant functor \( R(-) \) twice yields a covariant functor \( R(R(-)) = [R(-) \to R] \), the **continuation monad**\(^4\) over \( R \). The unit of the monad represents \( X \) as a sub-dcpo of \( R^R^X \) and is given by the map \( \delta: X \to R^R^X \) that assigns to every \( x \in X \) the projection or evaluation map \( \delta_x: R^X \to R \) defined by
\[
\delta_x(f) = f(x)
\]

For a map \( t: X \to [R^Y \to R] \) its Kleisli lifting \( t^\dagger: [R^X \to R] \to [R^Y \to R] \) is given by
\[
t^\dagger(\varphi)(g) = \varphi(\lambda x. t(x)(g)).
\]

According to our setting, a program will be interpreted by a **state transformer**, a Scott continuous map \( t: X \to R^R^Y \). A predicate transformer will be a Scott-continuous map \( p: R^Y \to R^X \). To every state transformer corresponds a predicate transformer that assigns the weakest precondition to every postcondition:

**Lemma 2.2** The dcpos of state and predicate transformers are canonically isomorphic:

\[
[R^Y \to R]^X \cong [R^Y \to R^X]
\]

A state transformer \( t: X \to [R^Y \to R] \) and a predicate transformer \( p: R^Y \to R^X \) correspond to one another under this isomorphism if and only if
\[
t(x)(g) = p(g)(x) \quad \text{for } x \in X, g \in R^Y.
\]

---

\(^3\) Since \( R \) can be very different from the two-element set of truth values, this notion of a predicate is very wide, and one instead uses terms like ‘prevision’ [4], ‘expectation’ [21], ‘random variable’ [18] instead, depending on the concrete situation.

\(^4\) See [20], for example, for background on monads.
**Proof.** For a state transformer \( t: X \to [R^Y \to R] \) let \( p = P(t): R^Y \to R^X \) be defined by \( P(t) = \lambda g. \lambda x. t(x)(g) \). As we are in a Cartesian closed category and \( P(t) \) is defined by a \( \lambda \)-calculus term, \( P(t) \) is Scott-continuous. Similarly if, for a predicate transformer \( p: R^Y \to R^X \), we define \( t = T(p) = \lambda x. \lambda g. p(g)(x) \), then \( T(p)(x): R^Y \to R \) is in fact Scott-continuous for every \( x \in X \) and \( T(p) \) is a Scott-continuous map from \( X \) to \( [R^Y \to R] \). Moreover, \( P(T(p))(g)(x) = T(p)(x)(g) = p(g)(x) \) for every \( x \in X \) and every \( g \in R^Y \). Similarly, \( T(P(t)) = t \) for every state transformer \( t \). One may notice that equation (3) is an isomorphism of dcpos, since \( P \) and \( T \) are \( \lambda \)-definable hence Scott-continuous mutually inverse maps.

Our domain \( R \) of observations will carry additional structure, it will be a d-\( \Omega \)-algebra: a dcpo with an algebraic structure of signature \( \Omega \) which interprets the constructors in the programming language (see Section 3). The exponentials \( R^X \) and \( R^{R^X} \) become d-\( \Omega \)-algebras, too, with pointwise defined operations. A program will be interpreted by a state transformer \( t: X \to \mathcal{F}_R Y \), where \( \mathcal{F}_R Y \) is the d-\( \Omega \)-subalgebra of \( R^{R^Y} \) generated by the projections \( \delta_y, y \in Y \).

The assignment \( X \mapsto \mathcal{F}_R X \) gives rise to a monad that we call subordinate to the continuation monad. This paper deals with the

**Problem 2.3** Find conditions (healthiness conditions) that characterize the predicate transformers \( p: R^Y \to R^X \) that correspond to the state transformers \( t: X \to \mathcal{F}_R Y \).

We cannot offer a complete answer to this question. But we exhibit a framework which always yields necessary conditions that the predicate transformers corresponding to state transformers \( t: X \to \mathcal{F}_R Y \) must satisfy. And we give a criterion for when these necessary conditions are also sufficient. This criterion has to be checked separately in each special situation.

Let us make precise what we mean by a monad \( \mathcal{F} \) subordinate to the continuation monad: Suppose that \( \mathcal{F} \) assigns to every dcpo \( X \) a sub-dcpo \( \mathcal{F} X \) of \([R^X \to R]\) in such a way that the following properties are satisfied:

\[
\delta_x \in \mathcal{F} X \quad \text{for all} \quad x \in X;
\]

\[
t^\dagger(\mathcal{F} X) \subseteq \mathcal{F} Y \quad \text{for every} \quad t: X \to [R^Y \to R].
\]

Then \( R^{R^u} \) maps \( \mathcal{F} X \) into \( \mathcal{F} Y \) for every map \( u: X \to Y \); indeed, \( R^{R^u} \) is the Kleisli lifting of \( \delta \circ u: X \to [R^Y \to R] \). Thus \( R^{R^u} \) induces a map \( \mathcal{F} u: \mathcal{F} X \to \mathcal{F} Y \) in such a way that \( \mathcal{F} \) becomes a functor, and even a monad with (the corestriction of) \( \delta \) as unit and the (restriction-corestriction of the) Kleisli lifting \( t^\dagger|_{\mathcal{F} X}: \mathcal{F} X \to \mathcal{F} Y \) for
Our methods can be applied to quite some examples in the literature, in particular for nondeterminism and probability [4,5,9,13,14,15,16,18,21,25]. There, the monads are usually presented in the form of powerdomains. For applying our results one has to find functional representations for these powerdomains of the type $F_R X$, as we have seen in the Introduction. This paper is based on previous work by the author [11]. There, several examples are worked out explicitly, it is not possible in this paper because of space restrictions. The reader is invited to consult that source for examples.

Acknowledgments
The author would like to thank Gordon Plotkin for many insights. Thomas Streicher has listened to many questions and patiently discussed possible answers. The referees’ suggestions were quite helpful.

3 Algebraic structure
We recall a few concepts from universal algebra adapted to the category DCPO.

An operation of arity $n \in \mathbb{N}$ on a dcpo $A$ is a map $\omega: A^n \rightarrow A$. If $A$ and $B$ both carry an $n$-ary operation $\omega$, a map $h: A \rightarrow B$ is an $\omega$-homomorphism if, for all $(a_1, \ldots, a_n) \in A^n$, we have:

$$h(\omega(a_1, \ldots, a_n)) = \omega(h(a_1), \ldots, h(a_n))$$

Definition 3.1 A d-signature $\Omega$ is a sequence of dcpos $\Omega_n, n \in \mathbb{N}$. The elements $\omega \in \Omega_n$ are the operation symbols of arity $n$.

Definition 3.2 A d-algebra of d-signature $\Omega$ (a d-$\Omega$-algebra, for short) consists of a dcpo $A$ together with operations $\omega^A: A^n \rightarrow A$, one for each $\omega \in \Omega_n$, such that $(\omega, a_1, \ldots, a_n) \mapsto \omega^A(a_1, \ldots, a_n) : \Omega_n \times A^n \rightarrow A$

is Scott-continuous for every $n$. A map $u: A \rightarrow B$ of d-$\Omega$-algebras is an $\Omega$-homomorphism, if it is an $\omega$-homomorphism for every $\omega \in \Omega$.

We stress that the value $\omega^A(a_1, \ldots, a_n)$ depends continuously not only on the arguments $a_i$ but also on $\omega \in \Omega_n$. By choosing the $\Omega_n$ to be (unordered) sets we recover the usual notion of a signature $\Omega$ in universal algebra.

Convention 3.3 We will omit the superscript when denoting operations $\omega^A$ on a d-$\Omega$-algebra $A$ and simply write $\omega$ instead of $\omega^A$.

In proofs, we will use a binary operation, denoted by $+$, instead of an arbitrary $n$-ary operation $\omega$. In this way, proofs become easier to read. Of course, we will
not use any special property like commutativity that one usually associates with an operation +. This does not affect the general validity of our proofs; one just has to replace \( x_1 + x_2 \) by \( \omega(x_1, \ldots, x_n) \) in order to obtain the general proof.

We fix a d-signature \( \Omega \) and a d-\( \Omega \)-algebra \( R \). For every dcpo \( X \), the function space \( R^X \) also becomes a d-\( \Omega \)-algebra. For \( \omega \in \Omega_n \) the operation \( \omega \) on the function space \( R^X \) is defined pointwise: For all \( f_1, \ldots, f_n \in R^X \) and all \( x \in X \),

\[
\omega(f_1, \ldots, f_n)(x) = \omega(f_1(x), \ldots, f_n(x)).
\]

For every map \( u: X \to Y \), the map \( R^u: R^Y \to R^X \) is an \( \Omega \)-homomorphism. Thus, we may view \( R(-) \) to be contravariant functor from the category DCPO to the category of d-\( \Omega \)-algebras and \( \Omega \)-homomorphisms.

In the same way, the operations \( \omega \) can be extended to the function space \( R^{R^X} = \left[ R^X \to R \right] \) so that the latter becomes a d-\( \Omega \)-algebra, too, and the maps \( R^{R^u} \) are \( \Omega \)-homomorphisms.

**Lemma 3.4** For every \( t: X \to \left[ R^Y \to R \right] \), the Kleisli lifting \( t^\dagger: \left[ R^X \to R \right] \to \left[ R^Y \to R \right] \) is an \( \Omega \)-homomorphism.

**Proof.** We check that, for every binary operation + in \( \Omega_2 \) and all \( \varphi_1, \varphi_2 \), we have \( t^\dagger(\varphi_1 + \varphi_2) = t^\dagger(\varphi_1) + t^\dagger(\varphi_2) \). For every \( g \in R^Y \) we have indeed: \( t^\dagger(\varphi_1 + \varphi_2)(g) = (\varphi_1 + \varphi_2)(\lambda x. t(x)(g)) = \varphi_1(\lambda x. t(x)(g)) + \varphi_2(\lambda x. t(x)(g)) = t^\dagger(\varphi_1)(g) + t^\dagger(\varphi_2)(g) = (t^\dagger(\varphi_1) + t^\dagger(\varphi_2))(g) \).

### 4 \( R \)-Free algebras

We are interested in the monad that represents the free objects over a dcpo \( X \) relative to our d-\( \Omega \)-algebra \( R \) of observations that we keep fixed throughout this section.

A subalgebra of a d-\( \Omega \)-algebra \( A \) which is a sub-dcpo, too, is called a d-\( \Omega \)-subalgebra. The intersection of any family of d-\( \Omega \)-subalgebras is again a d-\( \Omega \)-subalgebra. Thus every subset of \( A \) generates a d-\( \Omega \)-subalgebra, the intersection of all d-\( \Omega \)-subalgebras containing the subset.

**Definition 4.1** The d-\( \Omega \)-subalgebra \( F_R X \) of \( \left[ R^X \to R \right] \) generated by the projections \( \delta_x, x \in X \), is called the free d-\( \Omega \)-algebra over \( X \) with respect to \( R \) or simply the \( R \)-free d-\( \Omega \)-algebra over \( X \).

For a map \( t: X \to F_R Y \subseteq \left[ R^Y \to R \right] \), the Kleisli lifting \( t^\dagger: \left[ R^X \to R \right] \to \left[ R^Y \to R \right] \) maps \( F_R X \) into \( F_R Y \), since \( t^\dagger \) is an \( \Omega \)-homomorphism by Lemma 3.4. This shows:
Proposition 4.2 \((\mathcal{F}_R, \delta, \delta)\) is a monad over the category DCPO subordinate to the continuation monad in the sense made precise at the end of Section 2, the Kleisli lifting of a map \(t:X \rightarrow \mathcal{F}_R Y\) being the restriction and corestriction of the Kleisli lifting \(t\) for the continuation monad \(R^{(-)}\).

Since we have a monad, the d-\(\Omega\)-algebras \(\mathcal{F}_R X\) are free for the class of Eilenberg-Moore algebras. It is a challenge to determine these Eilenberg-Moore algebras concretely. A natural conjecture would be that \(\mathcal{F}_R X\) is free over \(X\) for the class of d-\(\Omega\)-algebras determined by the (in)equational theory of the d-\(\Omega\)-algebra \(R\). This conjecture is supported by a theorem due to G. Birkhoff (see [2]) which tells us that, in the category SET, \(\mathcal{F}_R X\) is free over the set \(X\) in the class of all \(\Omega\)-algebras that satisfy the equational laws that hold in \(R\). Such a strong statement will not hold in the dcpo-setting, in general, although it holds in many examples.

The following proposition (that we state without proof) shows that \(\mathcal{F}_R X\) is free in the class of d-\(\Omega\)-algebras that are embeddable in some power of \(R\). This class is sometimes called the quasi-variety generated by \(R\). The algebras in this class satisfy not only all equational and inequational laws that hold in \(R\), but also all implications between two such laws (Horn formulas) that hold in \(R\).

**Proposition 4.3** Let \(u\) be a map from a dcpo \(X\) to a d-\(\Omega\)-algebra \(A\) that is embeddable in some \(R^Y\) as a d-\(\Omega\)-subalgebra. Then there is a unique \(\Omega\)-homomorphism \(\hat{u}:\mathcal{F}_R X \rightarrow A\) extending \(u\) along \(\delta\).

## 5 Homomorphism monads

We continue with a fixed d-\(\Omega\)-algebra \(R\) of d-signature \(\Omega\). In order to find properties characterizing the predicate transformers \(p:R^Y \rightarrow R^X\) that correspond to the state transformers \(t:X \rightarrow \mathcal{F}_R Y\), we need a second monad subordinate to the continuation monad.

For two d-\(\Omega\)-algebras \(A\) and \(B\), we denote by

\[
[A \xrightarrow{\Omega} B]
\]

the set of all \(\Omega\)-homomorphisms \(u:A \rightarrow B\). The pointwise supremum of a directed family of \(\Omega\)-homomorphisms is again an \(\Omega\)-homomorphism. Thus, \([A \xrightarrow{\Omega} B]\) becomes a dcpo, a sub-dcpo of the dcpo \([A \rightarrow B]\) of all Scott-continuous maps from \(A\) to \(B\).

**Proposition 5.1** For a d-\(\Omega\)-algebra \(R\), the assignment

\[
X \mapsto [R^X \xrightarrow{\Omega} R]
\]

41
yields a monad subordinate to the continuation monad. The unit is (the corestriction of) \(\delta\) and the Kleisli lifting of a map \(t: X \to [R^Y \xrightarrow{\Omega} R]\) is (the restriction-corestriction) \(t^1: [R^X \xrightarrow{\Omega} R] \to [R^Y \xrightarrow{\Omega} R]\).

**Proof.** We show that we are in a situation as described at the end of Section 2.

(a) Clearly, the projections \(\delta_x: R^X \to R\) are \(\Omega\)-homomorphisms for every \(x \in X\).

(b) For every state transformer \(t: X \to [R^Y \xrightarrow{\Omega} R]\), the Kleisli lifting \(t^1\) maps \([R^X \xrightarrow{\Omega} R]\) to \([R^Y \xrightarrow{\Omega} R]\). Indeed, let \(\varphi: R^X \to R\) be an \(\Omega\)-homomorphism. For a binary operation \(+\) in \(\Omega\) and \(g_1, g_2 \in R^Y\), we have:

\[
\begin{align*}
  t^1(\varphi)(g_1 + g_2) &= \varphi(\lambda x. t(x)(g_1 + g_2)) & \text{by the definition of } t^1 \\
  &= \varphi(\lambda x. (t(x)(g_1) + t(x)(g_2))) & \text{since } t(x) \text{ is a homomorphism} \\
  &= \varphi(\lambda x. t(x)(g_1) + \lambda x. t(x)(g_2)) & \text{since } + \text{ is defined pointwise} \\
  &= \varphi(\lambda x. t(x)(g_1)) + \varphi(\lambda x. t(x)(g_2)) & \text{since } \varphi \text{ is a homomorphism} \\
  &= t^1(\varphi(g_1)) + t^1(\varphi(g_2)) & \text{by the definition of } t^1.
\end{align*}
\]

The 'homomorphism monad' \(\langle [R^{(-)} \xrightarrow{\Omega} R], \delta, t^1 \rangle\) exhibited in the previous proposition behaves well with respect to the one-to-one correspondence between state and predicate transformers:

**Proposition 5.2** Let \(R\) be \(d\)-\(\Omega\)-algebra. Under the one-to-one correspondence between state transformers and predicate transformers in Lemma 2.2 the predicate transformers \(p: R^Y \to R^X\) corresponding to the state transformers \(t: X \to [R^Y \xrightarrow{\Omega} R]\) are characterized by the property of being \(\Omega\)-homomorphisms: \([R^Y \xrightarrow{\Omega} R]^X \cong [R^X \xrightarrow{\Omega} R^X]\)

**Proof.** Let \(t: X \to [R^Y \xrightarrow{\Omega} R]\) be a state transformer and \(p: R^Y \to R^X\) the corresponding predicate transformer according to Lemma 2.2. We show that, for a binary operation \(+\) in \(\Omega\), \(t(x)\) is a \(+\)-homomorphism for every \(x \in X\) if, and only if, \(p\) is a \(+\)-homomorphism.

If \(t(x)\) is a \(+\)-homomorphism for every \(x \in X\) then, for all \(g_1, g_2 \in R^Y\),

\[
  p(g_1 + g_2)(x) = t(x)(g_1 + g_2) = t(x)(g_1) + t(x)(g_2) = p(g_1)(x) + p(g_2)(x) = (p(g_1) + p(g_2))(x),
\]

whence \(p(g_1 + g_2) = p(g_1) + p(g_2)\), that is, \(p\) is a \(+\)-homomorphism. If conversely \(p\) is a \(+\)-homomorphism, then \(t(x)(g_1 + g_2) = p(g_1 + g_2)(x) = (p(g_1) + p(g_2))(x) = p(g_1)(x) + p(g_2)(x) = t(x)(g_1) + t(x)(g_2)\) which shows that \(t(x)\) is a \(+\)-submorphism for all \(x \in X\).
One may notice that the proof above is identical to the only proof that we gave in the Introduction for that special situation.

6 Commuting operations

We come back the monad \( F_R \) of section 4 for a given d-\( \Omega \)-algebra \( R \). We want to consider state transformers \( t: X \to F_R Y \) and the corresponding predicate transformers \( p: R^Y \to R^X \) according to Lemma 2.2. In order to apply the results obtained in the previous section with the homomorphism monad we have to introduce a new framework.

**Definition 6.1** Given two operations \( \sigma \) of arity \( m \) and \( \omega \) of arity \( n \) on a dcpo \( A \), we say that \( \sigma \) and \( \omega \) commute if for all ‘matrices’ \( (x_{ij})_{i=1,...,m, j=1,...,n} \) of elements in \( A \), we have:

\[
\omega(\sigma(x_{11},...,x_{m1}), \ldots, \sigma(x_{1n},...,x_{mn})) = \sigma(\omega(x_{11},...,x_{1n}), \ldots, \omega(x_{m1},...,x_{mn}))
\]

This is equivalent to the statement that \( \sigma: A^m \to A \) is an \( \omega \)-homomorphism, equivalently, that \( \omega: A^n \to A \) is a \( \sigma \)-homomorphism.

**Example 6.2** A constant \( c \) commutes with an \( n \)-ary operation \( \omega \) if and only if \( \omega(c,\ldots,c) = c \). Two commuting constants have to agree. Two unary operations \( \rho \) and \( \sigma \) commute if they commute as functions: \( \rho \circ \sigma = \sigma \circ \rho \). A unary operation \( \rho \) commutes with a binary operation \( + \) if and only if

\[
\rho(x + y) = \rho(x) + \rho(y)
\]

Two binary operation \( + \) and \( \ast \) commute if

\[
(x_1 \ast x_2) + (x_3 \ast x_4) = (x_1 + x_3) \ast (x_2 + x_4)
\]

In particular, a binary relation \( \ast \) commutes with itself if

\[
(x_1 \ast x_2) \ast (x_3 \ast x_4) = (x_1 \ast x_3) \ast (x_2 \ast x_4)
\]

Thus, every commutative, associative binary operation commutes with itself.

Now let \( \Omega \) be a d-signature and \( R \) a d-\( \Omega \)-algebra. Let

\[
\Sigma_n = [R^n \xrightarrow{\Omega} R]
\]

be the dcpo of all \( \Omega \)-homomorphisms \( \sigma: R^n \to R \), that is, \( \Sigma_n \) consists of all operations of arity \( n \) on \( R \) that commute with all \( \omega \in \Omega \). The \( \Sigma_n \) form a second d-signature \( \Sigma \) and \( R \) is a d-\( \Sigma \)-algebra, too. The fact that, on \( R \), the operations in \( \Sigma \) commute with those in \( \Omega \) is given by equational laws of the form in Definition 6.1. These equational laws are inherited by exponentials \( R^X \) and \( R^{RX} \) considered
as d-(Ω ∪ Σ)-algebras (with pointwise defined operations) so that the operations \(\omega \in \Omega\) commute with all the operations \(\sigma \in \Sigma\) on all exponentials of \(R\).

At this point it becomes clear, why we wanted to choose signatures which are dcpos and not simply (unordered) sets; indeed, our signature \(\Sigma\) is a dcpo in a natural way.

The homomorphisms between two \(\Omega\)-algebras do not form an \(\Omega\)-algebra, in general. The following observation was a surprise to me. But if you think about it, you might find that you always have known it:

**Lemma 6.3** Suppose that \(R\) is a d-\(\Omega\)-algebra and \(\Sigma\) a d-signature of operations \(\sigma\) on \(R\) that commute with all \(\omega \in \Omega\). Then the set \([R^X \xrightarrow{\Sigma} R]\) of all \(\Sigma\)-homomorphisms \(\varphi: R^X \to R\) is a d-\(\Omega\)-subalgebra of \([R^X \to R]\) containing the \(R\)-free algebra \(F_{RX}\).

**Proof.** If \(\varphi_1, \ldots, \varphi_n: R^X \to R\) are \(\Sigma\)-homomorphisms, then \(\omega(\varphi_1, \ldots, \varphi_n)\) is also a \(\Sigma\)-homomorphism for every \(\omega \in \Omega_n\). Indeed, if \(\omega\) is a binary operation \(+\) then, for every binary operation \(*\) in \(\Sigma_2\), hence commuting with \(+\), we have:

\[
(\varphi_1 + \varphi_2)(f_1 * f_2) = \varphi_1(f_1 * f_2) + \varphi_2(f_1 * f_2) \\
= \varphi_1(f_1) * \varphi_1(f_2) + \varphi_2(f_1) * \varphi_2(f_2)) \\
= (\varphi_1(f_1) + \varphi_2(f_1)) * (\varphi_1(f_2) + \varphi_2(f_2)) \\
= (\varphi_1 + \varphi_2)(f_1) * (\varphi_1 + \varphi_2)(f_2)
\]

Thus the \(\Sigma\)-homomorphisms \(\varphi: R^X \to R\) form an \(\Omega\)-subalgebra \([R^X \xrightarrow{\Sigma} R]\). Clearly all the projections \(\delta_x\) are \(\Sigma\)-homomorphisms. Hence, \([R^X \xrightarrow{\Sigma} R]\) contains \(F_{RX}\), the d-\(\Omega\)-subalgebra of \([R^X \to R]\) generated by the projections. \(\square\)

From the previous lemma and Proposition 5.2 we immediately deduce our main result on healthiness conditions:

**Theorem 6.4** Suppose that \(R\) is a d-\(\Omega\)-algebra and \(\Sigma\) a d-signature of operations on \(R\) that commute with all \(\omega \in \Omega\). Then the predicate transformers \(p: R^Y \to R^X\) corresponding to the state transformers \(t: X \to F_{RY}\) are necessarily \(\Sigma\)-homomorphisms. If \(F_{RY} = [R^Y \xrightarrow{\Sigma} R]\), then the \(\Sigma\)-homomorphisms \(p: R^Y \to R^X\) are precisely the predicate transformers corresponding to state transformers \(t: X \to F_{RY}\):

\[(F_{RY})^X \cong [R^Y \xrightarrow{\Sigma} R^X].\]

For applying this Theorem, the challenge is to find operations on \(R\) that commute with those in \(\Omega\). This then yields necessary healthiness conditions for the predicate transformers. It depends very much on the special situation whether these healthiness conditions are also sufficient: one has to show that the d-\(\Omega\)-algebra \([R^Y \xrightarrow{\Sigma} R]\)
is indeed generated by the projections $\delta_y, y \in Y$, hence equal to the $R$-free $d\Omega$-algebra $F_R Y$. The classical example of observable predicates can be treated in this way, and also the example of convex sets and effect modules as viewed by B. Jacobs [8]. In the first case role of $R$ is taken by the two element dcpo $2 = \{0 < 1\}$ without any algebraic structure, in the second case by the unit interval.

We now look at the special situation where the operations of the $d\Omega$-algebra commute with one another (see also [23]):

**Definition 6.5** A $d\Omega$-algebra is called *entropic* if any two operations $\sigma, \omega \in \Omega$ commute.

We note that the entropic $d\Omega$-algebras are the algebras of a commutative monad over the category DCPO in the sense of A. Kock [17].

As a particular case of Lemma 6.3 and Theorem 6.4 with $\Omega = \Sigma$ we have:

**Corollary 6.6** For an entropic $d\Omega$-algebra $R$, the $\Omega$-homomorphisms $\varphi: R^X \to R$ form a $d\Omega$-algebra $[R^X \xrightarrow{\Omega} R]$ containing the $R$-free algebra $F_R X$ as a $d\Omega$-subalgebra.

The predicate transformers $p: R^Y \to R^X$ corresponding to state transformers $t: X \to F_R Y$ are $\Omega$-homomorphisms. If $F_R Y = [R^X \xrightarrow{\Omega} R]$ these predicate transformers are precisely the $\Omega$-homomorphisms.

Entropicity is quite a special property. Using Example 6.2 we obtain examples of entropic algebras: commutative semigroups, commutative monoids, commutative groups, modules over commutative rings, semimodules over commutative semirings, semilattices and unital semilattices.

Corollary 6.6 can be used for deriving the healthiness criteria for angelic nondeterminism in the Introduction (Section 1). The only specific property to be proved is that every unital semilattice homomorphism $\varphi: 2^Y \to 2$ is the supremum of the projections $\delta_x$ with $\delta_x \leq \varphi$ which is equivalent to the property that every nonempty Scott-closed subset of a dcpo is the union of the $\downarrow x, x \in X$. In the same way this Corollary can be used for deriving healthiness criteria for predicate transformers in the case of demonic and erratic (the combination of angelic and demonic) nondeterminism as well as for probabilistic nondeterminism as in [9,10,25].

But the known results for predicate transformers in the presence of both nondeterministic and probabilistic choice do not fit into the framework developed above. The reason is that, for example, on the nonnegative reals, the operation of addition and the semilattice operation max and min do not commute. We therefore propose a relaxed framework.
Keimel

7 Relaxed morphisms and relaxed entropic algebras

We relax the previous framework by replacing equalities by inequalities (compare Definition 6.1):

**Definition 7.1** Let \( \omega \) be an operation of arity \( n \) defined on dcpos \( A \) and \( B \). A map \( h: A \to B \) is called an \( \omega \)-**submorphism** \(^5\) if

\[
h(\omega(x_1, \ldots, x_n)) \leq \omega(h(x_1), \ldots, h(x_n)) \quad \text{for all } x_1, \ldots, x_n \in A.
\]

An \( \omega \)-**supermorphism** is defined in the same way replacing the inequality \( \leq \) by its opposite \( \geq \).

For d-algebras of d-signature \( \Omega \), we want to distinguish some operations \( \omega \in \Omega \) for which we would like to consider relaxed morphisms. For this, we suppose that the d-signature \( \Omega \) is the union of two d-sub-signatures \( \Omega^\leq \) and \( \Omega^\geq \) which need not be disjoint.

**Definition 7.2** A map \( h: A \to B \) between d-algebras of d-signature \( \Omega = \Omega^\leq \cup \Omega^\geq \) is said to be a **relaxed** \( \Omega \)-**morphism** if \( h \) is an \( \omega \)-submorphism for all \( \omega \in \Omega^\leq \), but an \( \omega \)-supermorphism for \( \omega \in \Omega^\geq \). (For \( \omega \) in both \( \Omega^\leq \) and \( \Omega^\geq \), \( h \) will be an \( \omega \)-**homomorphism**.)

The pointwise supremum of a directed family of relaxed \( \Omega \)-morphisms is again a relaxed \( \Omega \)-morphism. Thus the set \( [A \rarrow \Omega \to B] \) of relaxed \( \Omega \)-morphisms from \( A \) to \( B \) is a sub-dcpo of the function space \( [A \to B] \). As in Proposition 5.1 and 5.2 we have:

**Proposition 7.3** Let \( R \) be a d-\( \Omega \)-algebra of d-signature \( \Omega = \Omega^\leq \cup \Omega^\geq \).

(a) For every state transformer \( t: X \to [R^Y \rarrow \Omega \to R] \), the Kleisli lifting \( t^\dagger: [RX \to R] \to [R^Y \to R] \) maps relaxed \( \Omega \)-morphisms to relaxed \( \Omega \)-morphisms, so that our continuation monad \( ([R^{\dagger -} \to R], \delta, \dagger) \) restricts to a monad \( ([R^{\dagger -} \to R], \delta, \dagger) \).

(b) Under the bijective correspondence of Lemma 2.2, the predicate transformers \( p: R^Y \to RX \) corresponding to the state transformers \( t: X \to [R^Y \rarrow \Omega \to R] \) are the relaxed \( \Omega \)-morphisms:

\[
[R^Y \rarrow \Omega \to R]^X \cong [R^Y \rarrow \Omega \to RX]
\]

The proofs are the same as for the corresponding claims in 5.1 and 5.2. We just have to replace the equality sign by the appropriate inequality \( \leq \) in case \( \omega \in \Omega^\leq \).

\(^5\) For the terminology we have been guided by a common terminology in analysis. A function on a vector space is subadditive if \( h(x + y) \leq h(x) + h(y) \) and superadditive if the reverse inequality holds.

46
and \( \geq \) in case \( \omega \in \Omega^\geq \) every time that we have used the homomorphism property there.

We now turn to the question under what circumstances the relaxed \( \Omega \)-morphisms form a subalgebra of \([R^X \rightarrow R]\).

**Definition 7.4** We will say that an operation \( \sigma \) of arity \( m \) on a dcpo \( R \) subcommutes with an operation \( \omega \) of arity \( n \) (equivalently, \( \omega \) supercommutes with \( \sigma \)) if, for all \( x_{ij} \in R, i = 1, \ldots, m, j = 1, \ldots, n \):

\[
\sigma(\omega(x_{11}, \ldots, x_{1n}), \ldots, \omega(x_{m1}, \ldots, x_{mn})) \leq \omega(\sigma(x_{11}, \ldots, x_{m1}), \ldots, \sigma(x_{1n}, \ldots, x_{mn}))
\]

This is equivalent to the statement that \( \sigma: R^m \rightarrow R \) is an \( \omega \)-submorphism, and also equivalent to the statement that \( \omega: R^n \rightarrow R \) is \( \sigma \)-supermorphism. Whenever this inequational law holds in \( R \), it also holds in \( R^X \) and in \( R^{R^X} \).

We now let \( R \) be a d-\( \Omega \)-algebra. For every natural number \( m \), we denote by \( \Sigma^\leq_m \) and \( \Sigma^\geq_m \) the dcpos of all operations \( \sigma: R^m \rightarrow R \) that subcommute, resp., supercommute, with all \( \omega \in \Omega \). These give rise to d-signatures \( \Sigma^\leq, \Sigma^\geq, \) and \( \Sigma = \Sigma^\leq \cup \Sigma^\geq \).

As in Lemma 6.3 we have:

**Lemma 7.5** The relaxed \( \Sigma \)-morphisms \( \varphi: R^X \rightarrow R \) form a d-\( \Omega \)-subalgebra \([R^X \overset{r\Sigma}{\longrightarrow} R]\) of \([R^X \rightarrow R]\).

We are mainly interested in the following situation where we can choose \( \Omega = \Sigma \):

**Definition 7.7** A d-\( \Omega \)-algebra \( R \) is said to be relaxed entropic, if every \( \sigma \in \Omega \) either subcommutes with every \( \omega \in \Omega \) or supercommutes with every \( \omega \in \Omega \).

From the preceding Theorem we deduce:

**Corollary 7.8** Let \( R \) be a relaxed entropic d-\( \Omega \)-algebra. The collection \([R^X \overset{r\Omega}{\longrightarrow} R]\) of all relaxed \( \Omega \)-morphisms \( \varphi: R^X \rightarrow R \) is a d-\( \Omega \)-subalgebra of \([R^X \rightarrow R]\).

The d-\( \Omega \)-subalgebra \( F_{R^X} \) of \([R^X \rightarrow R]\) generated by the projections \( \delta_x, x \in X \), is a d-\( \Omega \)-subalgebra of \([R^X \overset{r\Omega}{\longrightarrow} R]\).
The predicate transformers corresponding to state transformers $t: X \rightarrow \mathcal{F}_\Omega Y$ are relaxed $\Omega$-morphisms $p: R^I \rightarrow R^Y$. If $\mathcal{F}_\Omega Y = [R^Y \xrightarrow{\Omega} R]$, then these predicate transformers are precisely the relaxed $\Omega$-morphisms.

Whether we have equality $F_{R^Y} = [R^Y \xrightarrow{\Omega} R]$, has to be decided separately in each special case.

On the nonnegative real line addition subcommutes with the semilattice operation $x \vee y = \max(x, y)$ and it supercommutes with $x \wedge y = \min(x, y)$. One has indeed for arbitrary nonnegative real numbers:

\[(10) \ (x_1 + x_2) \vee (x_3 + x_4) \leq (x_1 \vee x_3) + (x_2 \vee x_4)\]
\[(11) \ (x_1 + x_2) \wedge (x_3 + x_4) \geq (x_1 \wedge x_3) + (x_2 \wedge x_4)\]

These simple facts allow us to use our relaxed setting for deriving healthiness conditions for predicate transformers in the presence of mixed nondeterministic and probabilistic choice as in [4,5,13,14,15,16,21].

8 Concluding remarks

The framework for deriving healthiness conditions for predicate transformers developed in this paper looks quite narrow, although it applies to almost all situations known to the author. (An exception is [7], where one meets a quite different notion of predicate.) Although there is no proof, there is some evidence that it may not be possible to characterize predicate transformers in situations that do not fit under this framework.

Nevertheless, our methods allows some straightforward extensions. We have not carried them out in this paper in order to keep it at a technically simple level.

Firstly we may allow infinite arities for signatures and consider operations $\omega: R^I \rightarrow R$ for infinite sets $I$. We may also allow arities to be dcpos; that is, a signature may contain operation symbols $\omega$ of arity $P$, where $P$ is a dcpo; then $\omega$ will be interpreted as a map $\omega: R^P \rightarrow R$. For example, we may choose $P$ to be the two element dcpo $\mathbf{2} = \{0 < 1\}$; an operation $\omega: R^2 \rightarrow R$ of arity $\mathbf{2}$ will be defined on the graph of the order of $R$ and not on all of $R \times R$.

We have worked in the category DCPO of directed complete partially ordered sets and Scott-continuous functions. The results apply in particular to the subcategory SET of sets. One can use the same arguments in other Cartesian closed categories as, for example, the category of qcb-spaces (quotients of countably based topological spaces [1]) and the category POSET of partially ordered sets and order preserving functions.

in a very general way. The results in his main examples on weakest precondition semantics for two player games [5, Sections 4 and 6] can be recovered by our methods if one transfers them to the category \textbf{POSET} which of course contains the category \textbf{SET}.

We can apply our methods also in situations where the ambient category is no longer Cartesian closed. Since we are working with exponentials of a fixed object \(R\) and certain subobjects thereof, we have to ensure that these exponentials exist and yield a model of simply typed \(\lambda\)-calculus. For this, an appropriate setting is provided by Hofmann and Streicher [6]. A category \(\mathcal{C}\) they call \textit{category with continuations} if \(\mathcal{C}\) has finite products and a subclass \(\mathcal{T}\) of objects with a distinguished object \(R \in \mathcal{T}\) of \textit{responses} such that every \(A \in \mathcal{T}\) has an exponential \(R^A \in \mathcal{T}\), with the property that \(R^A \times B \in \mathcal{T}\) for any \(B \in \mathcal{T}\). A simple example for this situation is the category of continuous dcpos and Scott-continuous maps provided that \(R\) is a continuous lattice. Another example is the category of topological spaces and continuous maps: For \(\mathcal{T}\) one may take the class of exponentiable spaces and for \(R\) a continuous lattice with the Scott topology.

The referees would have liked to see a new striking example, where our methods can be applied. But our Theorems 6.4 and 7.6 clearly indicate quite narrow limitations to the use of predicate transformer semantics. It is a quite rare phenomenon that operations commute or subcommute. One may not be able to go far beyond the known examples. And if the operations do not commute, one has to find a manageable collection of operations that commute with or subcommute with the given ones, a task that I have no idea how to be attacked except for some very simple cases.

References


Sequential Algorithms for Unbounded Nondeterminism

J. Laird

Department of Computer Science, University of Bath, UK

Abstract

We give extensional and intensional characterizations of higher-order functional programs with unbounded nondeterminism: as stable and monotone functions between the biorders of states of ordered concrete data structures, and as sequential algorithms (states of an exponential ocds) which compute them. Our fundamental result establishes that these representations are equivalent, by showing how to construct a unique sequential algorithm which computes a given stable and monotone function. We illustrate by defining a denotational semantics for a functional language with countable nondeterminism (“fair PCF”), with an interpretation of fixpoints which allows this to be proved to be computationally adequate. We observe that our model contains functions which cannot be computed in fair PCF, by identifying a further property of the definable elements, and so show that it is not fully abstract.

Keywords: Sequential Algorithms, Nondeterminism, Fairness, Biorders

1 Introduction

This paper develops a model of higher-order computation with unbounded nondeterminism. In this setting we may write programs which will always return a value but may take an unbounded number of steps to do so, corresponding to the notion of fairness [6]. A major challenge for capturing such programs is that they do not correspond to continuous functions, in general. In domain theory, this may be resolved by weakening the continuity properties required (e.g. to \(\omega_1\)-continuity [1]), although this admits many undefinable functions and leaves fewer principles with which to reason about program behaviour. A more intensional representation of programs (for example, as strategies in a games model) offers the possibility of studying unbounded nondeterminism in computation more directly, although traditional representations of strategies as collections of finite sequences of moves are insufficient to capture the distinction between infinite interactions and finite, but unbounded ones [7].

We take an approach which relates extensional and intensional representations of programs with unbounded nondeterminism: our main result is an equivalence be-

---

1 Research supported by UK EPSRC grant EP/K037633/1

---
tween stable and monotone functions and sequential algorithms on ordered concrete data structures. We show that these equivalent representations may be used to interpret a simple functional programming language with unbounded nondeterminism (fair PCF). We show that this model contains elements which are not definable as terms, leading to a failure of full abstraction and suggesting how it could be further refined.

1.1 Related Work

Our model is based on an intensional description of stable and monotone functions on biorders generated from ordered concrete data structures. Biorders, which combine some intensional information, in the form of the stable order, with the extensional (Scott) order, were introduced by Berry [2]. In previous work, the author has shown that stable and continuous functions on biorders with a (extensionally) greatest element are (Milner-Vuillemin) sequential, and used them to construct models of sequential languages such as the lazy \( \lambda \)-calculus [12], as well as \( \lambda \)-calculi with nondeterminism [11]. However, although these models technically carry information about program behaviour, they do not do so in a transparent way.

Concrete data structures were introduced by Kahn and Plotkin [9], as part of a definition of sequentiality for higher-order functionals, but the more intensional notion of sequential algorithm (a state of a “function-space” CDS) introduced by Berry and Curien [3] offers an appealing model of computation in its own right. On the one hand, concrete data structures correspond to a positional form of games, and sequential algorithms to positional strategies (see e.g. [8]). On the other, sequential algorithms may be related to purely extensional models: in the deterministic case, Cartwright, Curien and Felleisen [4] have established that they compute, and are equivalent to “observably sequential” functions; the author has given a more abstract representation of the latter as bistable functions on bistable biorders [12,10].

To interpret sequential, but nondeterministic programs (corresponding to stable and monotone functions on Berry-style biorders, which are sequential, but not strongly sequential) as sequential algorithms, we abandon the consistency condition on states (that any cell may be filled with at most one value). However, this also requires an ordering on cells and values (corresponding to game positions), to reflect the fact that (for example) any program which may diverge in response to a given argument may still diverge in response to an argument with a wider range of behaviours. This notion of an ordered concrete data structure was introduced in [13], in which stable and continuous functions were shown to correspond to finite-branching sequential algorithms. Here, we extend this correspondence to unbounded nondeterminism. This requires a new notion of ordinal-indexed interaction, to distinguish computations which are infinite from those which are finite but unbounded.

2 PCF With Unbounded Nondeterminism

In order to illustrate the interaction between higher-order functions, recursion and unbounded nondeterminism, we introduce a programming language which combines them — an extension of PCF with natural number choice — for which we will give
a denotational semantics. In other words, we add to the simply-typed \( \lambda \)-calculus over the single ground type \( \text{nat} \), the following constants\(^2\):

- **Numerals** \( 0 : \text{nat} \) and \( \text{suc} : \text{nat} \rightarrow \text{nat} \).
- **Conditional** \( \text{If0} : \text{nat} \rightarrow \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \).
- **Fixpoints**: \( Y : (T \rightarrow T) \rightarrow T \) for each type \( T \).
- **Error and Choice**: \( \top : \text{nat} \) and \( ? : \text{nat} \).

### 2.1 Operational Semantics

The (small-step) reduction relation for closed terms is defined in Table 1. We study a must-testing semantics (unbounded choice is definable from bounded choice up to may-testing equivalence: a sequential algorithms semantics for the latter was presented in [13]). Define \( \Downarrow \) (**must-convergence**) to be the least predicate on programs (closed terms of type \( \text{nat} \)) such that for any program \( M \), if \( M \Downarrow \) for all programs \( M' \) such that \( M \rightarrow M' \) then \( M \Downarrow \). (So, in particular, every numeral is must-convergent, and if \( M_1 \rightarrow M_2 \rightarrow \ldots \rightarrow M_n \rightarrow \ldots \) then none of the \( M_i \) are must-convergent.) Thus we may define must-approximation (\( \preceq \)) and must-equivalence (\( \simeq \)) as the least precongruence and congruence on terms (respectively) such that if \( M \preceq N \) or \( M \simeq N \) then \( M \Downarrow \) implies \( N \Downarrow \). Clearly, \( \preceq \) is a partial order on the \( \simeq \)-equivalence classes of closed terms at each type.

#### 2.2 Examples

Evidently, we may express bounded choice (up to must-equivalence) using countable choice — e.g. defining \( M \text{ or } N = \text{If0} \ ? \) then \( M \) else \( \lambda x.N \). But attempting to define countable choice using bounded choice — e.g. as \( Y \lambda f.\lambda x.(x \text{ or } (f(\text{suc}(x)))) \) — will fail (evaluation may always take the right hand branch and so diverge).

To express e.g. the **fair merge** of infinite streams of objects of type \( T \), the latter may be represented as objects of type \( \text{nat} \rightarrow T \) (i.e. \( \text{head}(M) = M_0 \), \( \text{tail}(M) = \lambda x.M(\text{suc}(x)) \) and \( M :: N = ((\text{If0} y) M) N \). The fair merge function,

\[ (\lambda x.M) N \rightarrow M[N/x] \quad ? \rightarrow_n n \in \mathbb{N} \quad \overline{Y} M \rightarrow \overline{M} (\overline{Y} M) \]

\[ \text{If0} 0 \rightarrow \lambda x.\lambda y.x \quad \text{If0} \text{suc}(n) \rightarrow \lambda x.\lambda y.(y n) \]

\[ M \rightarrow M' \quad M N \rightarrow M' N \quad \text{suc} M \rightarrow \text{suc} M' \quad \text{If0} M \rightarrow \text{If0} M' \]

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Operational Semantics for Fair PCF</th>
</tr>
</thead>
</table>

\(^2\) Our denotational semantics can also interpret the catch operator of observably sequential PCF, but we omit this since our focus is not on full abstraction.

\(^3\) If its first argument evaluates to \( \text{suc}(n) \), this passes \( n \) to its third argument, so we may define \( \text{pred} : \text{nat} \rightarrow \text{nat} = \lambda x.((\text{If0} x) 0) \lambda y.y \).
merge : (nat → T) → (nat → T) → nat → T returns any interleaving of its arguments which includes all entries from both lists by alternately taking a non-empty initial segment of unbounded length from each stream.

\( \langle Y \lambda f \lambda x \lambda u \lambda v. \text{If} (\text{succ}(u) \text{?} v u) \text{else} \lambda y. \text{head}(u) :: ((f y \text{tail}(u) v)) \rangle \)

Conversely, we may express countable choice in terms of fair merge — e.g. by returning the position of the first zero in a fair merge of the stream of 1s with the stream of 0s:

\( \langle Y \lambda f. \lambda x. \lambda y. \text{If} (0 \text{head}(x) \text{then} (f \text{tail}(x) (\text{suc}(y)))) \text{else} (\text{merge}(\lambda x. 0) (\lambda x. 1)) \rangle 0 \)

For each \( i \in \mathbb{N} \), define \( ?_i \) : nat by \( ?_0 = ? \) and \( ?_{i+1} = \text{suc}(?_i) \). Then (we claim: proof via the denotational semantics is straightforward) \( ?_i \preceq ?_{i+1} \) for each \( i \in \mathbb{N} \), and \( \top \) is a \( \preceq \)-least upper bound for the chain \( \langle ?_i | i \in \mathbb{N} \rangle \). This may be used to show that many first-order functions definable in fair PCF are not \( \preceq \)-continuous. For example, \( \lambda x. \text{If} (\text{if} x \text{then} 0 \text{else} \Omega) \) : nat → nat: If0 ? then 0 else Ω diverges for all \( i \), but If0 ⊤ then 0 else Ω converges.

Note that this example also shows that application is not \( \preceq \)-continuous with respect to functions as well as arguments — i.e. \( \lambda f.(f \top) \) is a \( \preceq \)-least upper bound for the chain of terms \( \langle M_i = \lambda f.(f ?_i) | i \in \omega \rangle \) but \( M_i \lambda x. \text{If} 0 x \text{then} 0 \text{else} \Omega \) diverges for all \( i \), whereas \( (\lambda f.\text{if} \top \text{then} \lambda x. \text{If} 0 x \text{then} 0 \text{else} \Omega \) converges. This creates difficulties for defining well-behaved least fixed points for functions, which we will resolve semantically by working with the stable order, for which application is continuous with respect to functions (although not arguments).

### 3 Complete Biorders

We generalize the notion of biorder [2,5] to infinite meets (corresponding to infinite branching under a must-testing interpretation).

**Definition 3.1** A complete biorder is a complete (meet) lattice \( |D|, \sqsubseteq \) with a second partial order \( \leq \) on \( |D| \) such that:

- If \( x \leq y \) then \( x \sqsubseteq y \).
- \( \bot \leq x \) for all \( x \in D \) (where \( \bot = \cap D \)).
- For any \( X, Y \subseteq D \), if \( X \leq Y \) (meaning: \( \forall x \in X \exists y \in Y. x \leq y \land \forall y \in Y. \exists x \in X. x \leq y \)) then \( \cap X \leq \cap Y \).

We write \( \uparrow X \) if \( X \subseteq |D| \) is bounded above in the stable order.

**Lemma 3.2** If \( \uparrow X \) then \( \cap X \) is the greatest lower bound for \( X \) in the stable order.

**Proof.** Supposing \( x \leq y \) for all \( x \in X \):

- For any \( x \in X, X \leq \{x, y\} \) and so \( \cap X \leq x = \cap \{x, y\} \).
- If \( z \leq x \) for all \( x \in X \), then \( \{z\} \leq X \) and so \( \cap \{z\} \leq \cap X \).

\( \square \)
**Definition 3.3** A function between biorders \( f : D \to E \) is said to be *monotone stable* if it preserves both orders, and is *conditionally multiplicative* — i.e. if \( \uparrow X \), then \( f(\prod X) = \prod f(X) \).

Let \( CB \) be the category in which objects are complete biorders and morphisms from \( D \) to \( E \) are monotone stable functions from \( D \) to \( E \).

**Proposition 3.4** \( CB \) is Cartesian closed.

**Proof.** Products are defined pointwise. The internal hom \( [D, E] \) is the lattice of monotone stable functions (i.e. \( (\prod F)(x) = \prod_{f \in F} f(x) \)), with the stable ordering defined:

\[
    f \leq_s g \text{ if for all } x, y \in D, x \leq_s y \implies f(x) \leq_s g(y) \text{ and } g(y) = f(y) \sqcap g(x).
\]

\( \Box \)

## 4 Ordered Concrete Data Structures

A concrete data structure \([9,3]\) consists of sets of *cells*, *values* and *events* (which are pairs of cells and values), and an *enabling relation* between events and cells. The idea is that each step of a sequential computation is represented as an event (the filling of a cell with a value), which may be dependent on some combination of previous events having occurred (as specified by the enabling relation). This may be considered as a two-player game between the *environment*, which may propose an enabled cell to be filled, and the *program*, which can then fill it with a value. *Deterministic* programs correspond to deterministic strategies for this game which specify a unique value for filling enabled cells. They are represented as *states*: sets of events which satisfy two conditions: *consistency* — every cell must be filled with a unique value — and *safety* — for every filled cell there is a finite chain of enablings of filled cells within the state, back to an “initial cell” which does not depend on any prior events. In order to model unbounded nondeterministic computation with must-testing we adapt this setting in the following ways:

- Removing the consistency condition on states, so that a cell may be filled with multiple different values.
- Placing an ordering on cells and values, (and thus events) and requiring states to be upwards closed under this ordering. This reflects the fact that (for example) the response of a function to an argument which is a nondeterministic choice of \( x \) and \( y \) must include all of its responses to both \( x \) and \( y \).
- Including a distinct element \( \perp \) — representing divergence — with which any cell may be filled (cf. the representation of divergence in the game semantics of must-testing in \([7]\)).
- Extending the safety condition to allow infinite chains of enabling events. (This captures the distinction between infinite interaction, and that which is finite but unbounded.)

**Definition 4.1** A *(filiform)* ordered concrete data structure (ocds) \( A \) is a tuple \( (C(A), V(A), \vdash_A, E(A)) \) where \( C(A), V(A) \) (the *cells* and *values* of \( A \)) are partial orders not containing the distinguished element \( \perp \), \( E(A) \subseteq C(A) \times V(A) \) is a set of
events and $\vdash A \subseteq (E(A) \cup \{\ast\}) \times C(A)$ is a relation (enabling) such that $(c, v) \vdash c'$ implies $c < c'$.

We write $E(A)_\bot$ for the partial order $E(A) \cup (C(A) \times \{\bot\})$, with $(c, \bot) \leq (c', v)$ if $c \leq c'$.

A simple example of an ocds, which we shall use to interpret the type of natural numbers, is $N = \langle \{c\}, N, \{(*, c)\}, \{c\} \times N \rangle$, which has a single initial cell which may be filled with any natural number value.

**Definition 4.2** A proof of an event $e$ is an ordinal sequence of events $\langle (c_\alpha, a_\alpha) \mid \alpha \leq \kappa \rangle$ such that $c_\kappa \leq e$ and for $\alpha \leq \kappa$:

- If $\alpha = 0$ then $\ast \vdash c_\alpha$ ($c_0$ is initial),
- If $\alpha = \beta + 1$ then $e_\beta \vdash c_\alpha$,
- If $\alpha = \bigvee_{\beta < \alpha} c_\beta$, then $c_\alpha = \bigvee_{\beta < \alpha} c_\beta$.

We write $x \vdash^* e$ if there is a proof of $e$, all of the elements of which are in $x$.

Note that for any proof $\langle (c_\alpha, v_\alpha) \mid \alpha \leq \kappa \rangle$, if $\beta < \alpha \leq \kappa$ then $c_\beta < c_\alpha$.

**Definition 4.3** A state of an ocds $A$ is a set of events $x \subseteq E(A)_\bot$ satisfying:

- **Upwards Closure** If $e \in x$ and $e \leq e'$ then $e' \in x$.
- **Safety** If $e \in x$ then $x \vdash^* e$.

We write $D(A)$ for the set of states of $A$. A state $x$ is total if $x \subseteq E(A)$ (i.e. no cell is filled with $\bot$ in $x$).

Thus, the total states in $D(N)$ are in one-one correspondence with the subsets of $\mathbb{N}$, and there is a single divergent state $\bot = E(N)_\bot$. For a state $x \in D(A)$, we define the following sets of cells (subsets of $C(A)$):

- $F(x) = \{c \in C(A) \mid \exists a \in V(A)_\bot. (c, a) \in x\}$ — the set of filled cells of $x$.
- $En(x) = \{c \in C(A) \mid x \vdash^* (c, \bot)\}$ — the set of enabled cells of $x$.
- $A(x) = En(x) \cup F(x)$ — the set of accessible cells of $x$.

If $c \in En(x)$ and $(c, a) \in E(A)_\bot$, we write $x + (c, a)$ for the state $x \cup \{e \in E(A)_\bot \mid (c, a) \leq e\}$. We will also write $x + (c, V)$ for $\bigcup_{v \in V} (x + (c, v))$.

### 4.1 Ordered Concrete Data Structures as Complete Biorders

Since any union of states satisfies the safety and up-closure conditions, $(D(A), \supseteq)$ is a complete (meet) lattice, (with least element $\bot_A = \bigcup D(A)$). We define the stable order on states:

$$x \leq_s y$$

if and only if $y \subseteq x$ and if $(c, v) \in x$ then $(c, v) \in y$ or $(c, \bot) \in x$.

**Proposition 4.4** For any OCDS $A$, $D(A)$ is a complete biorder.

**Proof.** By definition, the stable order is contained within the extensional order. $\bot_A \leq_s x$ for all $x$, since $(c, v) \in \bot_A$ implies $(c, \bot) \in \bot_A$. Suppose $X \leq_s Y$: then for all $(c, a) \in Y$, there exists $y \in Y$ with $(c, a) \in y$ and hence $x \subseteq X$ with $(c, a) \in x$ and so $(c, a) \in x \subseteq \bigcup X$. Conversely, if $(c, a) \in \bigcup X$ then there exists $x \subseteq X$ such that
Note that if \( \uparrow X \) then for all \( x, y \in X \), if \( (c, v) \in x \) then either \( (c, \perp) \in x \) or \( (c, v) \in y \). Given \( C \subseteq En(x) \), let \( x_C = \bigcup_{c \in C} (x + (c, \perp)) \). The stable order may be characterized as follows.

**Proposition 4.5** \( y \leq_s x \) iff \( y = x_C \) for some \( C \subseteq En(x) \).

**Proof.** Evidently, \( x_C \leq_s x \), so it suffices to show that every element \( y \leq_s x \) has this form. Let \( C = \{ c \in En(x) \mid (c, \perp) \in y \} \), so that \( x_C \subseteq y \). We claim that \( y \subseteq x_C \): suppose \( (c, a) \in y \), but \( (c, a) \notin x \). Let \( (c_\alpha, v_\alpha \mid \alpha < \kappa) \) be a proof of \( (c, a) \) in \( y \). Then there exists a least value \( \alpha \) such that \( (c_\alpha, v_\alpha) \notin x \). So \( c_\alpha \in E(x) \), and \( (c, \perp) \in y \) by stability and so \( c_\alpha \in C \). Since \( c_\alpha \subseteq c \), \( (c, a) \in x_C \) as required. \( \square \)

## 5 Sequential Algorithms

By Proposition 4.4, we may define a category \( \mathcal{OC} \) in which objects are ordered concrete data structures and morphisms from \( A \to B \) are monotone stable functions from \( D(A) \to D(B) \). This has cartesian products, given by the disjoint union of ocds: \((C_1, V_1, E_1, i_1) \times (C_2, V_2, E_2, i_2) = (C_1 \uplus C_2, V_1 \cup V_2, \{ (c, 1, v) \mid (c, v) \in E_1 \} \cup \{ (c, 2, v) \mid (c, v) \in E_2 \}, \{ (c, i, v) \mid (c, v, d) \in E_i, i \in \{1,2\} \}).\)

The fully faithful (identity-on-morphisms) functor \( D : \mathcal{OC} \to \mathcal{CB} \) which sends each ocds \( A \to D(A) \) preserves products. So to establish Cartesian closure of \( \mathcal{OC} \) it suffices to define an exponent ocds \( A \Rightarrow B \) for each \( A, B \), such that \( D(A \Rightarrow B) \cong [D(A), D(B)] \) in \( \mathcal{CB} \). This is a key result, since it establishes that every monotone stable function between the biorder of states of an ocds is computed by a unique state of \( A \Rightarrow B \) or sequential algorithm.

We define the ordered concrete data structure \( A \Rightarrow B \) (cf. the analogous definition of unordered concrete data structure [3]) as follows:

**Cells** A cell of \( A \Rightarrow B \) is given by a pair of a total state of \( A \) and a cell of \( B \):
\[
C(A \Rightarrow B) = (D(A) \cap \mathcal{P}(E(A))) \times C(B) \quad \text{— with } (x, c) \leq (x', c') \text{ if } x \subseteq x' \text{ and } c \leq c'.
\]

**Values** A value of \( A \Rightarrow B \) is either a cell from \( A \) or a value from \( B \) — the order being determined pointwise from that of \( V(B) \) and the dual of \( C(A) \):
\[
V(A \Rightarrow B) = C(A)^c \uplus V(B).
\]

**Events** A cell \((x, c)\) of \( A \Rightarrow B \) may be filled with either a cell accessible from \( x \) in \( A \) or a value in \( B \) which can fill \( c \): \( E(A \Rightarrow B) = \{(x, c, c') \mid (x, c) \in C(A \Rightarrow B) \land c' \in A(x)\} \cup \{(x, c, v) \mid (x, c) \in C(A \Rightarrow B) \land (c, v) \in E(B)\} \).

**Enabling** The event \(((x, c), c')\) enables the cell \((x', c)\) if \( x' \) is obtained from \( x \) by filling \( c' \):
The event \(((x, c), v)\) enables the cell \((x', c)\) if \( (c, v) \) enables \( c' \) in \( B \):
\[
\vdash_{A \Rightarrow B} = \{(((x, c), c'), (x', c)) \in E(A \Rightarrow B) \times C(A \Rightarrow B) \mid \exists V \subseteq V(A). x' = x + (c', V)\} \cup \{((x, c), (x, c')) \in E(A \Rightarrow B) \times C(A \Rightarrow B) \mid (c, v) \vdash_B c'\}.
\]

4 We will elide any explicit tagging, assuming that the sets of cells of \( A \) and values of \( B \) are disjoint.
A sequential algorithm from $A$ to $B$ is a state of $A \Rightarrow B$.

5.1 Stable Functions from Sequential Algorithms

We need to establish that $D(A \Rightarrow B)$ and $[D(A), D(B)]$ are isomorphic in $\mathcal{CB}$. We first show that every sequential algorithm $\sigma \in D(A \Rightarrow B)$ computes a monotone stable function $\text{fun}(\sigma)$ from $D(A)$ to $D(B)$. Given a state $x$, let $\text{fun}(\sigma)(x) =$

$$\{(c, a) \in E(B) \mid \exists x' \subseteq x. ((x', c), a) \in \sigma \lor \exists c', (c', \bot) \in x \land ((x', c), c') \in \sigma\}$$

We need to show that $\text{fun}(\sigma)$ is a well-defined, monotone stable function.

**Lemma 5.1** For any $x$, $\text{fun}(\sigma)(x)$ is a state.

**Proof.**

**Upwards closure** Suppose $(c', a') \geq (c, a) \in \text{fun}(\sigma)(x)$. If there exists $x' \subseteq x$ with $((x', c), a) \in \sigma$ then $((x', c'), a') \geq ((x', c), a)$ and so $((x', c'), a') \in \sigma$ and $(c', a') \in \text{fun}(\sigma)(x)$.

If there exists $(c'', \bot) \in x$ such that $((x', c), c'') \in \sigma$ then $((x', c'), c'') \geq ((x', c), c'')$ so $((x', c'), c'') \in \sigma$ and hence $(c', a') \in \text{fun}(\sigma)(x)$.

**Safety** Suppose $(c, a) \in \text{fun}(\sigma)(x)$. Then there exists $(y, c, b) \in \sigma$ with $y \subseteq x$, and a proof of $(c, y)$ in $\sigma$ which therefore restricts to a proof of $c$ in $f(y)$.

**Lemma 5.2** If $\uparrow X$ and $(c, a) \in \text{fun}(\sigma)(\bigcup X)$ then either $(c, a) \in X$ for all $x \in X$ or $(c, \bot) \in \text{fun}(\sigma)(x')$ for some $x' \in X$.

**Proof.** If $(c, a) \in \text{fun}(\sigma)(\bigcup X)$ then there exists an event $e \in \sigma$ such that either $e = ((w, c), a)$, where $w \subseteq \bigcup X$ or $e = ((w, c), c')$, where $w + (c', \bot) \subseteq \bigcup X$. If $w \subseteq x$ for every $x \in X$ then in the first case $(c, a) \in \bigcup \text{fun}(\sigma)(X)$, and in the second case there exists $x' \in X$ with $w + (c', \bot) \subseteq x'$ and so $(c, \bot) \in \text{fun}(\sigma)(x')$.

So suppose $w \not\subseteq x$ for some $x \in X$. Fixing a proof of $e \in \sigma$, let $e' = ((w', c'), a')$ be the first element in this proof such that $w' \not\subseteq x$ for some $x \in X$. Then there is an immediately preceding event $((w'', c'), c'')$ such that $w' = w'' + (c', V)$ for some set of values $V$, including a value $u$ such that $(c', u) \not\in x$. Because $w' \subseteq \bigcup X$, there exists $x' \in X$ with $(c', u) \in x'$. Since $x \uparrow x'$, therefore $(c', \bot) \in x'$, and hence $(c', \bot) \in \text{fun}(\sigma)(y)$. Since $c' \subseteq c$, we have $(c, \bot) \in \text{fun}(\sigma)(x')$ as required.

**Proposition 5.3** For any sequential algorithm $\sigma$, $\text{fun}(\sigma)$ is monotone stable.

**Proof.** Evidently, if $x \geq y$ then $\text{fun}(\sigma)(x) \geq \text{fun}(\sigma)(y)$. If $x \leq_s y$, then $f(x) \leq_s f(y)$: suppose $(c, v) \in f(x) = f(x \cup y)$ and so by Lemma 5.2, $(c, v) \in f(y)$ or $(c, \bot) \in f(y)$. If $\uparrow X$, then $\text{fun}(\sigma)(\bigcup X) = \bigcup \text{fun}(\sigma)(X)$: if $(c, a) \in \text{fun}(\sigma)(\bigcup X)$ then by Lemma 5.2, either $(c, a) \in \text{fun}(\sigma)(x)$ for all $x \in X$, and so $(c, a) \in \bigcup \text{fun}(\sigma)(X)$, or $(c, a) \geq (c, \bot) \in \bigcup \text{fun}(\sigma)(X)$.

We now show that fun itself is a monotone stable function.

**Lemma 5.4** If $\sigma \leq_s \tau$ then $\text{fun}(\sigma) \leq_s \text{fun}(\tau)$.

**Proof.**
• For all $x$, $\text{fun}(\sigma)(x) \leq_s \text{fun}(\tau)(x)$. Suppose $(c,v) \in \text{fun}(\sigma)(x)$ but $(c,\bot) \notin \text{fun}(\sigma)(x)$. Then there exists $x' \subseteq x$ with $((x',c),v) \in \sigma$ and $((x',c),\bot) \notin \sigma$ and so $((x',c),v) \in \tau$ and $(c,v) \in \text{fun}(\tau)(x)$ as required.

• For all $x \leq_s y$, $\text{fun}(\sigma)(x) = \text{fun}(\tau)(x) \cup \text{fun}(\sigma)(y)$. Suppose $(c,a) \in \text{fun}(\sigma)(x)$, we need to show that $(c,a) \in \text{fun}(\tau)(x)$ or $(c,a) \in \text{fun}(\sigma)(y)$. By Proposition 5.3, if $(c,a) \notin \text{fun}(\sigma)(y)$, then $(c,\bot) \in \text{fun}(\sigma)(x)$, and so we may assume that $a = \bot$.

So either there exists an event $((z,c),\bot) \in \sigma$ with $z \subseteq x$ or else there exists $z + (c',\bot) \subseteq x$ such that $((z,c),c') \in \sigma$. But this latter case reduces to the first one, since either $((z,c),c') \in \tau$ (and so $(c,\bot) \in \text{fun}(\tau)(x)$ and we are done), or else $((z,c),\bot) \in \sigma$.

Assuming $(c,\bot) \notin \text{fun}(\sigma)(y)$, let $P$ be a proof of $((z,c),\bot) \in \sigma$, and let $((z',c'),a)$ be the least element of $P$ such that $z' \subseteq y$. Then there must be an immediately preceding event in $P$ of the form $((z'',c'),c'')$, where $z' = z'' + (c'',V)$ for some $V$, and hence $z'' + (c'',\bot) \subseteq x$, as $\uparrow \{x,y\}$. If $((z'',c'),c'') \in \tau$ then $(c,\bot) \in \text{fun}(\tau)(x)$. Otherwise $((z'',c'),\bot) \in \sigma$ and so $(c',\bot) \in \text{fun}(\sigma)(y)$, and so $(c,\bot) \in \text{fun}(\sigma)(y)$ as required.

Noting that for any set of states $S \subseteq D(A \Rightarrow B)$, $\text{fun}(\bigcup S) = \bigcup_{\sigma \in S} \text{fun}(\sigma)$ by construction — i.e. $\text{fun}$ is additive — we have shown that:

**Proposition 5.5** $\text{fun} : D(A \Rightarrow B) \rightarrow [D(A), D(B)]$ is a monotone stable function.

### 5.2 Stable Functions and Sequentiality

Concrete data structures were introduced in order to give a description of sequentiality for higher-order functionals [9]. Essentially, a function between (the sets of states of) concrete data structures $A$ and $B$ is Kahn-Plotkin sequential if any argument (state) $x$ of $A$, and cell $c$ of $B$ which is filled in $f(y)$ for some $y$ which extends $x$, can be associated with a cell, accessible from $x$, which must be filled in any state $z$ (which extends $x$) such that $c$ is filled in $f(z)$. However, in this original setting, divergence is represented implicitly, by not filling an enabled cell (rather than as an explicit divergence by filling a cell with $\bot$), and inclusion of states corresponds to the stable order. Thus we translate this original definition of Kahn-Plotkin sequentiality to the current setting by (essentially) replacing the role of “accessible cell” with that of “cell filled with $\bot$”, and “filled cell” with “enabled cell not filled with $\bot$.” We define a partial order ($\preceq$) on total states (which plays the role of the stable order in the original definition of Kahn-Plotkin sequentiality): $x \preceq y$ if $x \subseteq y$ and if $c \in F(x)$ then $(c,v) \in y$ implies $(c,v) \in x$.

**Definition 5.6** A function $f : D(A) \rightarrow D(B)$ is explicitly sequential if whenever $x,y$ are total states such that $x \preceq y$ then for any event $(c,v) \in f(y)$, either $(c,v) \in f(x)$, or there exists $c' \in A(x) \cap F(y)$ such that if $\uparrow \{x,z\}$ and $(c',\bot) \in z$ then $(c,\bot) \in f(z)$.

We will now show that all monotone stable functions are explicitly sequential.

**Lemma 5.7** If $x \preceq y$ then $y \subseteq x_{A(x) \cap F(y)}$. 

59
Proof. Suppose \((c, v) \in y\) but \((c, v) \notin x\). Let \(P\) be a proof of \(c\) in \(y\), and \((c', v')\) the least element of \(P\) which is not in \(x\). Then \(c'\) is accessible (enabled but not filled) in \(x\), as \(x \preceq y\) — i.e. \(c' \in A(x) \cap F(y)\), so \((c, v) \geq (c', \perp) \in x_{A(x) \cap F(y)}\). □

Proposition 5.8 Any monotone stable function from \(D(A)\) to \(D(B)\) is explicitly sequential.

Proof. Suppose \(x \preceq y\) and \((c, v) \in f(y)\) but \((c, v) \notin f(x)\). Since \(y \subseteq x_{A(x) \cap F(y)}\), we have \((c, v) \in f(x_{A(x) \cap F(y)})\) and since \(f(x_{A(x) \cap F(y)}) \subseteq f(x)\), we have \((c, \perp) \in f(x_{A(x) \cap F(y)})\).

By conditional multiplicativity, \(f(x_{A(x) \cap F(y)}) = \bigcup_{c' \in A(x) \cap F(y)} f(x + (c', \perp))\) and so there is a cell \(c' \in A(x) \cap F(y)\) such that \((c, \perp) \in f(x + (c', \perp))\) as required. □

We establish the converse — that every explicitly sequential function is monotone stable — by showing below that every explicitly sequential function from \(D(A)\) to \(D(B)\) is computed by a state of \(A \Rightarrow B\), which corresponds via \fun\ to a monotone stable function from \(D(A)\) to \(D(B)\).

5.3 Sequential Algorithms from Stable and Monotone Functions

We will use the sequentiality property to establish that \fun\ : \(D(A \Rightarrow B) \to [D(A), D(B)]\) is an isomorphism by defining its inverse, \strat\ : \([D(A), D(B)] \to D(A \Rightarrow B)\). Given a monotone stable function \(f : D(A) \to D(B)\), define \strat(f) \in D(A \Rightarrow B)\) to be the set of events:

\[
\{(x, c) \in C(A \Rightarrow B) \mid (x, c) \in f(x)\} \\
\cup \{(x, c, c') \in C(A \Rightarrow B) \times C(A) \mid (c, \perp) \in f(x + (c', \perp))\}
\]

Lemma 5.9 \strat(f) is an upper set.

Proof. Suppose \((x', c', a') \geq (x, c, a) \in \strat(f)\). If \(a, a' \in f(B)\) — i.e. \((c, a) \in f(x)\) — then \((c', a') \geq (c, a) \in f(x') \supseteq f(x)\), and so \((x', c', a') \in \strat(f)\). If \(a, a' \in C(A)\), so that \(a' \preceq a\) and hence \(x + (a, \perp) \subseteq x' + (a', \perp)\), then \((c, \perp) \in f(x' + (a', \perp))\) and so \((x', c', a') \in \strat(f)\) as required. □

Lemma 5.10 \strat(f) satisfies the safety property.

Proof. We construct a proof of each event \((x, c, a) \in \strat(f)\) using the explicit sequentiality property for \(f\). Suppose \(a \in f(B)\) and fix a proof \(\langle c_\beta, v_\beta \mid \beta \leq \lambda\rangle\) of \((c, a)\) in \(f(x)\).

For each ordinal \(\alpha\), we define:

• A state \(x_\alpha \preceq x\).
• An ordinal \(\kappa(\alpha) \leq \alpha\)
• An event \(e_\alpha \in E(A \Rightarrow B)\) such that \(e_\alpha = ((x_\alpha, c_{\kappa(\alpha)}), v_{\kappa(\alpha)})\) or \(e_\alpha = ((x_\alpha, c_{\kappa(\alpha)}), c')\) for some \(c' \in f(x)\).

such that if \(\kappa(\alpha) < \lambda\), then \(\langle e_\gamma \mid \gamma \leq \alpha\rangle\) is a proof of \(e_\alpha\).

• Let \(x_0 = \{\}\) and \(\kappa(0) = 0\),
• For all \(\alpha\), if \((c_{\kappa(\alpha)}), v_{\kappa(\alpha)})) \in f(x_\alpha)\) then let \(e_\alpha = ((x_\alpha, c_{\kappa(\alpha)}), v_{\kappa(\alpha)})\) and \(x_{\alpha+1} = x_\alpha\) and \(\kappa(\alpha + 1) = \min\{\lambda, \kappa(\alpha) + 1\}\).
Otherwise, by the explicit sequentiality of $f$ (Proposition 5.8) there is a cell $c' \in A(x_j) \cap F(x)$ such that $(c_{\kappa(\alpha)}, \bot) \in f(x_a + (c', \bot))$ and so we may set $e_\alpha = ((x_a, c_{\kappa(\alpha)}), c')$, and $x_{\alpha+1} = \bigcup \{x_a + (c', v) \mid (c', v) \in x\}$ and $\kappa(\alpha+1) = \kappa(\alpha)$.

- If $\alpha = \bigcup \{\beta < \alpha\}$ then $x_\alpha = \bigcup_{\beta < \alpha} x_\beta$ and $\kappa(\alpha) = \bigcup \{\kappa(\beta) \mid \beta < \alpha\}$.

Since $\kappa(\alpha) \neq \lambda$ implies $e_\alpha \neq e_\beta$ for all $\beta < \alpha$, there must be some (least) $\alpha$ (no greater than the cardinality of $E(A \Rightarrow B)$) such that $\kappa(\alpha) = \lambda$ and so $\{e_j \mid \beta \leq \alpha\}$ is a proof of $((x, c), a)$ in $\text{strat}(f)$. The case in which $a$ is a cell in $C(A)$ is similar. □

**Proposition 5.11** For any monotone stable function $f : D(A) \rightarrow D(B)$, $\text{strat}(f)$ is a well-defined state of $A \Rightarrow B$.

We now show that $(\text{fun}, \text{strat})$ are an isomorphism between $D(A \Rightarrow B)$ and $[D(A), D(B)]$. Given a state $x \in D(A)$, let $x^\top$ be the upper closure of $\{(c, v) \in x \mid (c, \bot) \notin x\}$.

**Lemma 5.12** $x^\top$ is a well-defined (total) state.

**Proof.** $x^\top$ is by definition an upper set. For safety, suppose $(c, v) \in x$: for any event $(c', v')$ in any proof of $c, c' \leq c$ and so if $(c', v') \notin x^\top$ then $(c, v') \notin x^\top$ — i.e. if $(c, v) \in x$ then any proof of $c$ in $x$ is a proof of $c$ in $x^\top$.

By definition, if $(c, v) \in x$ then either $(c, \bot) \in x$ or $(c, v) \in x^\top$ — i.e. $x \leq_s x^\top$.

**Lemma 5.13** For any monotone stable function $f : D(A) \rightarrow D(B)$ and state $x \in D(A)$: $\text{fun}(\text{strat}(f))(x) = f(x)$.

**Proof.** Suppose $(c, a) \in \text{fun}(\text{strat}(f))(x)$. Then either:

- there exists a total $y \subseteq x$ with $((y, c), a) \in \text{strat}(f)$, and so $(c, a) \in f(y) \subseteq f(x)$.

- or there exists $y \subseteq x$ with $((y, c), c') \in \text{strat}(f)$ and $(c', \bot) \in x$, and so $(c, \bot) \in f(y + (c', \bot)) \subseteq f(x)$.

For the converse, suppose $(c, a) \in f(x)$. If $(c, a) \in f(x^\top)$ then $((x^\top, c), a) \in \text{strat}(f)$, and so $(c, a) \in \text{fun}(\text{strat}(f))(x)$ as required.

Otherwise $(c, \bot) \in f(x)$, and by the explicit sequentiality of $f$ (Proposition 5.8), there exists $c' \in A(x^\top) \cap F(y)$ such that $(c', \bot) \in x$ and $(c, \bot) \in f(x^\top + (c', \bot))$ and so $((x^\top, c), c') \in \text{strat}(f)$ and $(c, a) \in \text{fun}(\text{strat}(f))(x)$ as required. □

**Lemma 5.14** For all sequential algorithms $\sigma \in D(A \Rightarrow B)$, $\text{strat}(\text{fun}(\sigma)) = \sigma$.

**Proof.** Suppose $((x, c), a) \in E(A \Rightarrow B)_\bot$. Then:

- If $a \in V(B)_\bot$ then $((x, c), a) \in \sigma$ if and only if $(c, a) \in \text{fun}(\sigma)(x)$ and only if $((x, c), a) \in \text{strat}(\text{fun}(\sigma))$.

- If $a \in C(A)$ then $((x, c), a) \in \sigma$ if and only if $(c, \bot) \in \text{fun}(\sigma)(x + (c', \bot))$ if and only if $((x, c), a) \in \text{strat}(\text{fun}(\sigma))$. □

**Theorem 5.15** $\text{fun} : [D(A), D(B)] \rightarrow D(A \Rightarrow B)$ is an isomorphism, with inverse $\text{strat} : D(A \Rightarrow B) \rightarrow [D(A), D(B)]$. 61
6 Denotational Semantics for Fair PCF

Types are interpreted as ordered concrete data structures: the type \textbf{nat} denotes the OCDS \( N \) of natural numbers and \( S \to T \) the exponential OCDS \([ S ] \Rightarrow [ T ]\). Terms \( x_1 : S_1, \ldots, x_n : S_n \vdash M : T \) denote monotone stable functions from \([ S_1 ] \times \cdots \times [ S_n ]\) to \([ T ]\). \( \top \) denotes the (constant function returning the) empty state and \( \bot \) the state \( \{(c, i) \mid i \in \mathbb{N}\} \) in which the single cell is filled with every value (but not \( \bot \)). There are evident functions denoted by \textbf{suc} and \textbf{if0}, and so it remains to show that the constant \( Y \) may be interpreted as a fixed point. By the Tarski-Knaster theorem, any monotone function \( f : D(A) \to D(A) \) has a \( \sqsubseteq \)-least fixed point, since \( D(A) \) is a lattice under the extensional order. However, because function application is not continuous in the extensional order, we do not know how to prove that this fixed point delivers a \textit{computationally adequate} model. Instead, we will show that fixed points may be constructed as \textit{stable} least upper bounds of chains of approximants.

**Definition 6.1** Say that an ocds is \textit{stably complete} if for any stably directed set of states \( X \) (i.e. for all \( x, y \in X \) there exists \( z \in X \) with \( x, y \leq_s z \)), \( \bigcap X \) is a stable least upper bound for \( X \).

There are pathological examples of ocds in which this property fails, but we can show that it holds for every object denoting a PCF type (which is evident in the case of \textbf{nat}).

**Proposition 6.2** If \( B \) is stably complete, then \( A \Rightarrow B \) is stably complete.

**Proof.** We show that the internal hom \([ D(A), D(B) ]\) has suprema of stably directed sets, and hence so does \( D(A \Rightarrow B) \), and then observe that these are given by the intersection operation. Suppose \( F \subseteq [ D(A), D(B) ] \) is stably directed. For any \( x \in D(A) \), \( \{ f(x) \mid f \in F \} \) is stably directed, so we may define \( \bigvee F \langle x \rangle = \bigcap \{ f(x) \mid f \in F \} \). This is evidently monotone with respect to the extensional and stable orders. To establish conditional multiplicativity, we need to show that if \( \uparrow X \) then \( \bigvee F \langle \prod X \rangle \subseteq \prod (\bigvee F \langle x \rangle) \). Suppose \( e \notin \bigcup_{x \in X} (\bigvee F \langle x \rangle) \) and choose any \( x \in X \): there exists \( f \in F \) with \( e \notin f(x) \). Now consider any \( y \in X \): there exists \( g \in F \) such that \( e \notin g(y) \), and \( h \in F \) such that \( f, g \leq h \) and so \( e \notin g(x) \cup g(y) = h(x) \cup h(y) \). Then \( f(x) \cup f(y) = f(x + y) = f(x) \cup h(x + y) \), and so \( e \notin f(y) \). So \( e \notin \bigcup_{y \in X} f(x) = f(\prod X) \) and hence \( e \notin (\bigvee F)(\prod X) \) as required.

To show that \( \bigvee F \) is a stably least upper bound for \( F \), if \( x \leq y \) then for all \( f \in F \), \( f(x) = h(x) \cap f(y) \) and so \( (\bigvee F) \langle x \rangle = \bigcap \{ h(x) \cup f(y) \mid f \in F \} = h(x) \cap (\bigvee F)(y) \).

Finally, observe that \( \text{strat}(\bigvee F) = \bigcap \{ \text{strat}(f) \mid f \in F \} \), since \( (x, c, v) \in \{ \text{strat}(f) \mid f \in F \} \) if and only if for all \( f \in F \), \( (c, v) \in f(x) \), if and only if \( (x, c, v) \in \bigvee F \), and similarly \( (x, c, c') \in \{ \text{strat}(f) \mid f \in F \} \) if and only if for all \( f \in F \), \( (c, \bot) \in f(x + (c', \bot)) \), if and only if \( (x, c) \notin \bigvee F \).

**Proposition 6.3** If \( A \) is stably complete and \( f : D(A) \to D(A) \) is monotone stable, then \( f \) has a \( \leq_s \)-least fixedpoint.
Proof. Define the chain of stable approximants \( f^\lambda \in D(A) \) for each ordinal \( \lambda \) by:

- \( f^\lambda = \bot \), if \( \lambda = 0 \),
- \( f^\lambda = f(f^\kappa) \), if \( \lambda = \kappa + 1 \)
- \( f^\lambda = \bigvee_{\kappa < \lambda} f^\kappa \) if \( \lambda = \bigcup_{\kappa < \lambda} \kappa \).

By the bounded cardinality of \( D(A) \) this has a stationary point, which is a \( \leq_s \)-least fixed point for \( f \). \( \square \)

Thus, we may define the denotation of \( Y : (T \rightarrow T) \rightarrow T \) to be the \( \leq_s \)-least fixpoint of \( h : (T \Rightarrow T) \Rightarrow T \Rightarrow (T \Rightarrow T) \Rightarrow T \) such that \( h(x)(y) = \text{fun}(y)(\text{fun}(x)(y)) \).

6.1 Soundness

Straightforward analysis of the reduction rules establishes that:

**Lemma 6.4** For any reducible program \( M \), \([M] \subseteq \bigcup\{[N] \mid M \rightarrow N\}\).

**Proposition 6.5** If \( M \Downarrow \) then \([M] \neq \bot\).

**Proof.** By Lemma 6.4, if \((c, \bot) \notin [N]\) for all \( N \) such that \( M \rightarrow N \) then \((c, \bot) \notin [M]\). So by definition, the set of convergent programs is contained in \([M] \mid (c, \bot) \notin [M]\) \( \square \).

To prove the converse (computational adequacy), we define “approximation relations” in the style of Plotkin [14]: for each type \( T \) we define a relation \( \ll_T \) between elements of \([T]\) and closed terms of type \( T \):

- \( x \ll_{\text{nat}} M \) if \( M \not\Downarrow \) implies \((c, \bot) \in x\) and \( M \rightarrow^* n \) implies \((c, n) \in x\).
- \( f \ll_{S \rightarrow T} M \) if \( x \ll_S N \) implies \( f(x) \ll_T M N \).

Note that:

- If \( M \rightarrow N \) for some unique \( N \) such that \( e \ll_{\text{nat}} N \) then \( e \ll_{\text{nat}} M \)
- If \((f_\alpha \mid \alpha < \lambda)\) is a stable chain of functions such that \( f_\alpha \ll_{S \rightarrow T} M \) for all \( \alpha < \lambda \) then \( \bigvee_{\alpha < \lambda} f_\alpha \ll_{S \rightarrow T} M \).

**Lemma 6.6** \( [Y] \ll_{(T \rightarrow T) \rightarrow T} Y \)

**Proof.** Suppose \( T = T_1 \rightarrow \ldots \rightarrow T_k \rightarrow \text{nat} \). We show by ordinal induction that \( h^\lambda \ll_{(T \rightarrow T) \rightarrow T} Y \) for each \( \lambda \) — i.e. if, \( g \ll_{T \rightarrow T} M \) and \( e_i \ll_{T_i} N_i \) for \( 1 \leq i \leq k \) then \((h^\lambda g)e_1 \ldots e_k \ll_{\text{nat}} (YM)N_1 \ldots N_k \).

- For \( \lambda = 0 \), we have \((h^\lambda g)e_1 \ldots e_k = \bot \ll_{\text{nat}} M N_1 \ldots N_k \).
- For \( \lambda = \kappa + 1 \), by hypothesis \( h^\kappa \leq Y \), and so \((h^\lambda g)e_1 \ldots e_k = g(h^\kappa g)e_1 \ldots e_n = g(h^\kappa g)e_1 \ldots e_n \leq M(YM)N_1 \ldots N_k \rightarrow M(YM)N_1 \ldots N_k \) and so \( h^\lambda \leq Y \) as required.
- For \( \lambda = \bigcup_{\kappa < \lambda} \kappa \), we have \( h^\lambda = \bigvee_{\kappa < \lambda} h^\kappa \leq [M] \) by stable chain closure. \( \square \)

We then define \( f : [\Gamma] \rightarrow [T] \ll_{T \Gamma} \Gamma \vdash M : T \) if \( \Gamma = \lambda_1 : S_1, \ldots, x_n : S_n \) and \( e_1 \ll_{S_1} N_1, \ldots, e_n \ll_{S_n} N_n \) implies \( f(e_1, \ldots, e_n) \ll_T M[N_1/x_1, \ldots, N_n/x_n] \).
Proposition 6.7 (Adequacy) \([M] \neq \bot\) implies \(M \Downarrow\).

Proof. We prove that if \(\Gamma \vdash M : T\) then \([M] \ll_{\Gamma,T} M\) by structural induction. \(\Box\)

Hence, by a standard argument, our model is inequationally sound: if \([M] \subseteq [N]\) then \(M \lessapprox N\).

7 Conclusions

We conclude by considering the completeness problem for our model. By adding \textit{catch} (a simple, non-local control operator which can distinguish between different sequentializations of a function) to PCF with \textit{bounded nondeterminism} [13] we can show that every \textit{finite branching} nondeterministic sequential algorithm is the least upper bound of a chain of definable approximants and so prove that this model is \textit{fully abstract}. The situation in fair PCF is more complicated: since continuity fails, we cannot reduce full abstraction to the \textit{finite definability} property. Moreover, our model does not accurately reflect sequential testing of arguments with unbounded nondeterminism, as we can show by giving an example of a stable and monotone function which is not definable in fair PCF. Consider the function \(k : D(N) \rightarrow D(N)\) such that:

- \(k(x) = \top\), if \(x\) is finite,
- \(k(x) = \bot\) if \(x = \bot\),
- \(k(x) = 0\), otherwise

This is monotone and stable, since \(x \leq_s y\) in \(D(N)\) implies \(x = y\) or \(x = \bot\). However, it cannot be computed in fair PCF, since verifying that \(x\) contains infinitely many values requires infinitely many computation steps. (The function \(g : D(N) \rightarrow D(N)\) such that \(g(x) = \top\) if \(x\) is finite, and \(g(x) = \bot\), otherwise, is definable as \(\lambda x. (Y \lambda f. \lambda y. (f 0 (x < y) \text{ then } \top \text{ else } (f x) \text{ suc}(y))) 0\).)

We may prove that \(k\) is not definable in fair PCF by showing that all definable functions have the following property:

Definition 7.1 A stable function \(f : D \rightarrow E\) is \textit{weakly co-continuous} if for any downwards \(\subseteq\)-directed set \(X\), \(f(\bigsqcup X) \leq_s \bigsqcup f(X)\).

The function \(h\) is not weakly co-continuous: let \(X\) be the set of finite states of \(N\) — this is downwards \(\subseteq\)-directed, but \(\bigsqcup X\) is infinite and so \(h(\bigsqcup X) = 0 \not\leq_s \top = \bigsqcup h(X)\). But we can show that all terms denote weakly co-continuous functions.

Lemma 7.2 Let \(F\) be a (upwards) \(\leq_s\)-directed set of weakly co-continuous functions from \(D\) to \(E\) (which is stably complete). Then \(\bigsqcup F\) is weakly co-continuous.

Proof. \((\bigsqcup F)(\bigsqcup X) = \bigsqcup \{f(\bigsqcup X) | f \in F\} \leq_s \bigsqcup \{\bigsqcup f(X) | f \in F\}\). \(\Box\)

Proposition 7.3 Every term of fair PCF denotes a weakly co-continuous function.

Proof. For each type \(T\), we define a predicate (hereditary weak co-continuity) on the states of \([T]\), by induction, as follows:

- Every \(x \in D([N])\) is hereditarily weakly co-continuous.
• $\sigma \in D([S \rightarrow T])$ is hereditarily weakly co-continuous if $\text{fun}(\sigma)$ is weakly co-continuous and for any hereditarily weakly co-continuous $x \in D([S])$, $\text{fun}(\sigma)(x) \in D([T])$ is hereditarily weakly co-continuous.

We now show by structural induction that for any term $x_1 : S_1, \ldots, x_n : S_n \vdash M : T$, $\lambda x_1 \ldots x_n. M : S_1 \rightarrow \ldots \rightarrow S_n \rightarrow T$ is hereditarily weakly co-continuous (using Lemma 7.2 for the fixpoint combinator).

This is sufficient to show that our semantics is not fully abstract, independently of the absence of the $\text{catch}$ operators (which are weakly co-continuous).

**Proposition 7.4** The sequential algorithm semantics of fair PCF is not fully abstract.

**Proof.** Define $\text{bchoice} : \text{nat} \rightarrow \text{nat} = \lambda f. \lambda x. \text{If}0 x \text{ then } \top \text{ else } \lambda y.(y \text{ or } f y)$ — this evaluates its argument and nondeterministically returns a bounded choice over all smaller values. Define $\text{let } x = ? \text{ in } M$ to be $\text{If}0 ? \text{ then } M[0/x] \text{ else } \lambda y. M[\text{Suc}(y)/x]$ — i.e. nondeterministically choose a value and bind it to $x$ in $M$. Now define $M = \lambda f. \text{If}0 (f ?) \text{ then } (\text{let } x = ? \text{ in } f \text{ bchoice}(x)) \text{ else } (\text{let } x = ? \text{ in } f \text{ bchoice}(x))$ and $N = \lambda f.f ?$. These terms (of type $(\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$) are not equivalent in our model — they may be distinguished by application to $\text{strat}(k)$: $[M](\text{strat}(k)) = \top$, but $[N](\text{strat}(h)) = 0$ (applying $k$ to $[?]$ returns 0, but applying $k$ to $[\text{bchoice}(n)]$ returns $\top$ and so $[M](\text{strat}(k))$ returns $\bigcap_{n \in \mathbb{N}} k\{i < n\} = \top$.

However, $M$ and $N$ are observationally equivalent: Let $L : \text{nat} \rightarrow \text{nat}$ be any closed term of fair PCF. Then by Proposition 7.3 $L$ denotes (the sequential algorithm of) a weakly co-continuous function and so $[L ?] \leq_s [\text{let } x = ? \text{ in } L \text{ bchoice}(x)]$.

Thus either $[L ?] = \bot$, in which case $[M L] = \bot = [N L]$ or $[L ?] = [\text{let } x = ? \text{ in } L \text{ bchoice}(x)]$, in which case $[M L] = [\text{If}0 L ? \text{ then } L ? \text{ else } L ?] = [L?] = [N L]$. Therefore by adequacy of our semantics, and the Context Lemma for PCF (which extends straightforwardly to fair PCF), $M$ and $N$ are observationally equivalent.

### 7.1 Further Directions

We have established a fundamental relationship between extensional and intensional representations of higher-order functional computation with unbounded nondeterminism. Further study of this model may shed light on the debate over the relevance of the concept of fairness [6]. Questions posed more directly by our semantics include:

- Can the notion of weak co-continuity be completed to give a characterization of the sequential algorithms which are definable in fair PCF (with $\text{catch}$) ? (This will also require a characterization of the effectively computable nondeterministic sequential algorithms.)

- Can we build a graph games model of unbounded non-determinism based on ordered concrete data structures ?
References


Bifibrational functorial semantics of parametric polymorphism

Neil Ghani\textsuperscript{a} Patricia Johann\textsuperscript{b} Fredrik Nordvall Forsberg\textsuperscript{a} Federico Orsanigo\textsuperscript{a} Tim Revell\textsuperscript{a}

\textsuperscript{a} University of Strathclyde, UK
\textsuperscript{b} Appalachian State University, USA

Abstract

Reynolds' theory of parametric polymorphism captures the invariance of polymorphically typed programs under change of data representation. Reflexive graph categories and fibrations are both known to give a categorical understanding of parametric polymorphism. This paper contributes further to this categorical perspective on parametricity by showing the relevance of bifibrations. Using bifibrations, it develops a framework for models of System F that are parametric, in that they verify the Identity Extension Lemma and Reynolds' Abstraction Theorem. We also prove that our models satisfy expected properties, such as the existence of initial algebras and final coalgebras, and that parametricity implies dinaturality.

Keywords: Parametricity, logical relations, System F, fibred category theory.

1 Introduction

Strachey [30] called a polymorphic function parametric if its behaviour is uniform across all of its type instantiations. Reynolds [25] made this mathematically precise by formulating the notion of relational parametricity, in which the uniformity of parametric polymorphic functions is captured by requiring them to preserve all logical relations between instantiated types. Relational parametricity has proven to be a key technique for formally establishing properties of software systems, such as representation independence [1,6], equivalences between programs [15], and useful (“free”) theorems about programs from their types alone [31]. In this paper, we treat relational parametricity for the polymorphic $\lambda$-calculus System F [10], which forms the core of many modern programming languages and verification systems. Hermida, Reddy, and Robinson [14] give a good introduction to relational parametricity.

Since category theory underpins and informs many of the key ideas underlying modern programming languages, it is natural to ask whether it can provide a useful perspective on parametricity as well. Ma and Reynolds [19] developed the first categorical formulation of relational parametricity, but their models were complicated
and challenging to understand. Moreover, Birkedal and Rosolini discovered that not all expected consequences of parametricity necessarily hold in their models (see [4]).

Another line of work, begun by O’Hearn and Tennent [21] and Robinson and Rosolini [28], and later refined by Dunphy and Reddy [7], uses reflexive graphs to model relations and functors between reflexive graph categories to model types. This is the state of the art for functorial semantics for parametric polymorphism. Interpreting types as functors is conceptually elegant and Dunphy and Reddy show that this framework is powerful enough to prove expected results, such as the existence of initial algebras for strictly positive type expressions [5]. However, since reflexive graph categories are relatively unknown mathematical structures, much of this development has had to be carried out from scratch. We propose to instead take the more established fibrational view of logic from the outset, and thus to analyse parametricity through the powerful lens of categorical type theory [16].

In doing so, we follow an extensive line of work by Hermida [12,13] and Birkedal and Møgelberg [4], who use fibrations to construct sophisticated categorical models not only of parametricity, but also of its logical structure in terms of Abadi-Plotkin logic [24]. Abadi-Plotkin logic is a formal logic for parametric polymorphism that includes predicate logic and a polymorphic lambda calculus, and thus requires significant machinery to handle. Using this machinery, Birkedal and Møgelberg are able to go beyond Dunphy and Reddy’s results and, for instance, prove that all positive type expressions — not just the strictly positive ones as for Dunphy and Reddy — have initial algebras. However, these impressive results come at the price of the complexity of the notions involved. Our aim is to achieve the same results in a simpler setting, closer to Dunphy and Reddy’s functorial semantics. We end up with a notion of model in which each type is interpreted as an equality preserving fibred functor and each term is interpreted as a fibred natural transformation. This is quite similar to the models produced by the parametric completion process of Robinson and Rosolini [28] (see also Birkedal and Møgelberg [4, Section 8]) and to Mitchell and Scedrov’s relator model [20], but with a more general notion of relation given by a fibration. We thus combine the generality of Birkedal and Møgelberg’s fibrational models with the simplicity of Dunphy and Reddy’s functorial semantics.

Our central innovation is the use of bifibrations to achieve this “sweet spot” in the study of parametricity. This is not necessary for the definition of our framework, for which Lawvere equality [17] (i.e., preindexing along diagonals only) suffices, but it helps considerably with both the concrete interpretation of ∀-types [9] and the handling of graph relations. At a technical level, our strongest result is to use our simpler framework to recover all the expected consequences of parametricity that Birkedal and Møgelberg [4] prove using Abadi-Plotkin logic. In particular, we go beyond Dunphy and Reddy’s result by deriving, this time with a functorial semantics, initial algebras for all positive type expressions, rather than just for strictly positive ones. Nevertheless, this paper is in no way intended as the final word on fibrational parametricity. Instead, we hope the simple re-conceptualization of parametricity we offer here — replacing the usual categorical interpretations of types as functors and

\footnote{We stress again that we are not trying to model all of Abadi-Plotkin logic, but rather only type systems involving parametric polymorphism. Indeed, with respect to Abadi-Plotkin logic, we could not hope to improve upon the results of Birkedal and Møgelberg [4], who give a sound and complete semantics.}
terms as natural transformations with their fibred counterparts — will open the way to the study of parametricity in richer settings, e.g., proof-relevant ones.

**Structure of the paper:** In Section 2 we give a short introduction to fibrations. In Section 3 we study Reynolds’ Identity Extension Lemma, Abstraction Theorem, and relational interpretations of System F’s type constructors. In Section 4 we extract bifibrational generalisations of these and construct our parametric models. In Section 5 we show that our models behave as expected by deriving initial algebras for all definable functors and proving that parametricity implies (di)naturality. Finally, we instantiate our framework to derive both “standard” and new models of relational parametricity in Section 6. Section 7 concludes and discusses future work.

## 2 A Fibrational Toolbox for Relational Parametricity

We give a brief introduction to fibrations; more details can be found in, e.g., [16].

**Definition 2.1** Let $U : \mathcal{E} \to \mathcal{B}$ be a functor. A morphism $g : Q \to P$ in $\mathcal{E}$ is cartesian over $f : X \to Y$ in $\mathcal{B}$ if $Ug = f$ and, for every $g' : Q' \to P$ in $\mathcal{E}$ with $Ug' = f \circ v$ for some $v : UQ' \to X$, there exists a unique $h : Q' \to Q$ with $Uh = v$ and $g' = g \circ h$. A morphism $g : P \to Q$ in $\mathcal{E}$ is opcartesian over $f : X \to Y$ in $\mathcal{B}$ if $Ug = f$ and, for every $g' : P \to Q'$ in $\mathcal{E}$ with $Ug' = v \circ f$ for some $v : Y \to UQ'$, there exists a unique $h : Q \to Q'$ with $Uh = v$ and $g' = h \circ g$.

We write $f^\#: P \to Q$ for the cartesian morphism over $f$ with codomain $P$, and $f^\#: P \to Q$ for the opcartesian morphism over $f$ with domain $P$. Such morphisms are unique up to isomorphism. If $P$ is an object of $\mathcal{E}$ then we write $f^\#: P \to Q$ for the domain of $f^\#: P \to Q$ and $\Sigma_f P$ for the codomain of $f^\#: P \to Q$.

**Definition 2.2** A functor $U : \mathcal{E} \to \mathcal{B}$ is a fibration if for every object $P$ of $\mathcal{E}$ and every morphism $f : X \to UP$ in $\mathcal{B}$, there is a cartesian morphism $f^\#: P \to Q$ in $\mathcal{E}$ over $f$. Similarly, $U$ is an opfibration if for every object $P$ of $\mathcal{E}$ and every morphism $f : UP \to Y$ in $\mathcal{B}$, there is an opcartesian morphism $f^\#: P \to Q$ in $\mathcal{E}$ over $f$. A functor $U$ is a bifibration if it is both a fibration and an opfibration.

If $U : \mathcal{E} \to \mathcal{B}$ is a fibration, opfibration, or bifibration, then $\mathcal{E}$ is its total category and $\mathcal{B}$ is its base category. An object $P$ in $\mathcal{E}$ is over its image $UP$ and similarly for morphisms. A morphism is vertical if it is over id. We write $\mathcal{E}_X$ for the fibre over an object $X$ in $\mathcal{B}$, i.e., the subcategory of $\mathcal{E}$ of objects over $X$ and morphisms over $\text{id}_X$.

For $f : X \to Y$ in $\mathcal{B}$, the function mapping each object $P$ of $\mathcal{E}$ to $f^\#: P \to P'$ extends to a functor $f^* : \mathcal{E}_Y \to \mathcal{E}_X$ mapping each morphism $k : P \to P'$ in $\mathcal{E}_Y$ to the morphism $f^*k$ with $kf^\#: P \to f^\#: P'$. The universal property of $f^\#: P \to P'$ ensures the existence and uniqueness of $f^*k$. We call $f^*$ the reindexing functor along $f$. A similar situation holds for opfibrations; the functor $\Sigma_f : \mathcal{E}_X \to \mathcal{E}_Y$ extending the function mapping each object $P$ of $\mathcal{E}$ to $\Sigma_f P$ is the opreindexing functor along $f$.

We write $\mathcal{C}$ for the discrete category of $\mathcal{C}$. If $U : \mathcal{E} \to \mathcal{B}$ is a functor, then the discrete functor $|U| : |\mathcal{E}| \to |\mathcal{B}|$ is induced by the restriction of $U$ to $|\mathcal{E}|$. If $n \in \mathbb{N}$, then $\mathcal{E}^n$ denotes the $n$-fold product of $\mathcal{E}$ in $\mathbf{Cat}$. The $n$-fold product of $U$, denoted $U^n : \mathcal{E}^n \to \mathcal{B}^n$, is the functor defined by $U^n(X_1, ..., X_n) = (UX_1, ..., UX_n)$.
Lemma 2.3 If \( U : E \rightarrow B \) is a functor then \([U] : |E| \rightarrow |B|\) is a bifibration. If \( U \) is a (bi)fibration then so is \( U^n : E^n \rightarrow B^n \) for any natural number \( n \). □

To formulate Reynolds' relational parametricity categorically, we define the category \( \text{Rel} \) of relations over \( \text{Set} \) and the relations fibration on \( \text{Set} [16] \).

Definition 2.4 The category \( \text{Rel} \) has triples \((A, B, R)\) as objects, where \( A, B, \) and \( R \) are sets and \( R \subseteq A \times B \). A morphism \((A, B, R) \rightarrow (A', B', R')\) is a pair \((f, g)\), where \( f : A \rightarrow A' \) and \( g : B \rightarrow B' \), such that if \((a, b) \in R\) then \((fa, gb) \in R'\).

We write \((A, B, R)\) as just \( R \) when \( A \) and \( B \) are immaterial or clear from context. Note that \( \text{Rel} \) is not the category whose objects are sets and whose morphisms are relations, which also sometimes appears in the literature. Each set \( A \) has an associated equality relation defined by \( \text{Eq} A = \{(a, a) \mid a \in A\} \).

Example 2.5 The functor \( U : \text{Rel} \rightarrow \text{Set} \times \text{Set} \) sending \((A, B, R)\) to \((A, B)\) is called the relations fibration on \( \text{Set} \). To see that \( U \) is indeed a fibration, let \((f, g) : (X_1, X_2) \rightarrow (Y_1, Y_2)\) be a morphism in \( \text{Set} \times \text{Set} \) with \( UR = (Y_1, Y_2) \) for some \( R \) in \( \text{Rel} \). If we define \((f, g)^* R \subseteq X_1 \times X_2\) by \((x_1, x_2) \in (f, g)^* R\) iff \((fx_1, gx_2) \in R\), then \((f, g)\) is a cartesian morphism from \((f, g)^* R\) to \( R \) over \((f, g)\). It is also easy to see that \( U \) is an opfibration, with opreindexing given by forward image. Thus, \( U \) is a bifibration. We denote the fibre over \((A, B)\) in the relations fibration on \( \text{Set} \) by \( \text{Rel}(A, B) \).

Definition 2.6 Let \( U : E \rightarrow B \) and \( U' : E' \rightarrow B' \) be bifibrations. A fibred functor \( F : U \rightarrow U' \) comprises two functors \( F_0 : B \rightarrow B' \) and \( F_1 : E \rightarrow E' \) such that \( U' F_1 F_0 U \) and cartesian morphisms are preserved, i.e., if \( f \) is cartesian in \( E \) over \( g \) in \( B \) then \( F_1 f \) is cartesian in \( E' \) over \( F_0 g \) in \( B' \). If \( F' : U \rightarrow U' \) is another fibred functor, then a fibred natural transformation \( \eta : F \rightarrow F' \) comprises two natural transformations \( \eta_0 : F_0 \rightarrow F'_0 \) and \( \eta_1 : F_1 \rightarrow F'_1 \) such that \( U' \eta_1 \eta_0 U \).

In this paper we use fibred functors and fibred transformations to interpret System F types and terms, and show that under mild conditions this gives parametric models.

## 3 Reynolds’ Model of Relational Parametricity

We now describe Reynolds' set-theoretic model of relational parametricity: first concretely, and then in terms of the relations fibration \( \text{Rel} \rightarrow \text{Set} \times \text{Set} \). As Reynolds discovered, there are in fact no set-theoretic models if the meta-theory is classical logic [26], but the following makes sense in the (intuitionistic) internal language of a topos [22], or in the Calculus of Constructions with impredicative \( \text{Set} \). Throughout, we assume a standard syntax for System F.

### 3.1 Semantics of Types

Reynolds presents two “parallel” semantics for System F: a standard set-based semantics \([\_]_o\), and a relational semantics \([\_]_r\). Given \( \Gamma \vdash T \) type, where the type context \( \Gamma \) contains \(|\Gamma| = n \) type variables, Reynolds defines interpretations \( [T]_o : |\text{Set}|^n \rightarrow \text{Set} \) and \( [T]_r : |\text{Rel}|^n(A, B) \rightarrow \text{Rel}([T]_o A, [T]_o B) \) by structural induction on type judgements as follows:
• Type variables: \( [X_i]_o A = A_i \) and \( [X_i]_r A = R_i \)

• Arrow types:

\[
[T_1 \to T_2]_o A = [T_1]_o A \to [T_2]_o A
\]

\[
[T_1 \to T_2]_r R = \{(f, g) \mid (a, b) \in [T_1]_r R \Rightarrow (fa, gb) \in [T_2]_r R\}
\]

• For all types:

\[
[\forall X.T]_o A = \{f : \prod_{S : \text{Set}} [T]_o (A, S) \mid \forall R' \in \text{Rel}(A', B'). (f A', f B') \in [T]_r (\text{Eq} A, R')\}
\]

\[
[\forall X.T]_r R = \{(f, g) \mid \forall R' \in \text{Rel}(A', B'). (f A', g B') \in [T]_r (R, R')\}
\]

The definitions of \( [\forall X.T]_o \) and \( [\forall X.T]_r \) depend crucially on one another. Thus, we do not really have two semantics — one based on Set and one based on Rel — but rather a single semantics based on the relations fibration \( U : \text{Rel} \to \text{Set} \times \text{Set} \). In other words, Reynolds’ definitions of \( [-]_o \) and \( [-]_r \) entail the following theorem:

**Theorem 3.1 (Fibrational Semantics of Types)** Let \( U \) be the relations fibration on Set. Every judgement \( \Gamma \vdash T \) induces a fibred functor \( [T] : |U|^{|\Gamma|} \to U \).

\[
\begin{array}{ccc}
|\text{Rel}|^{|^\Gamma|} & \xrightarrow{[T]_r} & \text{Rel} \\
|U|^{|\Gamma|} \downarrow & & \downarrow U \\
|\text{Set}|^{|^\Gamma|} \times |\text{Set}|^{|^\Gamma|} & \xrightarrow{[T]_o \times [T]_r} & \text{Set} \times \text{Set}
\end{array}
\]

Since the domain of \( [T]_r \) is a discrete category, requiring that \( [T] \) is a fibred functor amounts simply to requiring that the above diagram commutes. In particular, no preservation of cartesian morphisms by \( [T]_r \) is needed. Reynolds does not give a functorial action of types on morphisms. This is reflected in the appearance of discrete categories in Theorem 3.1. As a result, Reynolds’ pointwise interpretation of function spaces is the exponential in the functor category \( |U|^{|\Gamma|} \to U \) [27]. How parametricity treats the action on morphisms will become clear in Section 5.1; instead of acting on morphisms, the interpretation of types act on graph relations induced by morphisms. For now, we simply note that the use of discrete domains does not take us out of the fibrational framework: Lemma 2.3 ensures that \( [T] \) is a functor between fibrations. The Identity Extension Lemma (IEL) is key for many applications of parametricity. It says that every relational interpretation preserves equality relations\(^2\):

**Lemma 3.2 (IEL)** If \( \Gamma \vdash T \) then \( [T]_r \circ |\text{Eq}|^{|^\Gamma|} = \text{Eq} \circ [T]_o \).

\(^2\) Reynolds’ approach also handles “identity relations” that are not equality relations, such as the information order on domains. In this paper, like many others [2,4,13,24], we only treat equality relations. In future work, we hope to give an axiomatic account of “identity relations” similar to that of Dunphy and Reddy [7].
3.2 Semantics of Terms

Reynolds’ main result is his Abstraction Theorem, stating that all terms send related environments to related values. Reynolds first gives set-valued and relational interpretations of term contexts \( \Delta = x_1 : T_1, \ldots, x_n : T_n \) by defining \( [[\Delta]]_o = \prod T_1 \times \cdots \times T_n \) and \( [[\Delta]]_r = \prod T_1 \times \cdots \times T_n \). This defines a fibred functor \( [[\Delta]] : [U|[\Gamma]] \to U \). Reynolds’ then interprets each judgement \( \Gamma; \Delta \vdash T \) as a family of functions \( [t]_o : [[\Delta]]_o S \to [[T]]_o S \) for each environment \( S \in |\text{Set}|^{[\Gamma]} \). We omit the standard definition of \( [t]_o \) here. Finally, Reynolds proves:

**Theorem 3.3 (Abstraction Theorem)** Let \( A, B \in \text{Set}^{[\Gamma]} \), \( R \in \text{Rel}^{[\Gamma]}(A, B) \), \( a \in [[\Delta]]_o A \), and \( b \in [[\Delta]]_o B \). For every term \( \Gamma; \Delta \vdash t : T \), if \( (a, b) \in [[\Delta]]_r R \), then \( ([t]_o A a, [t]_o B b) \in [[T]]_r R \). Or, more concisely, fibrationally: every judgement \( \Gamma; \Delta \vdash T \) is interpreted as a fibred natural transformation \( ([t]_o, [t]_r) : [[\Delta]] \to [[T]] \).

\[
\begin{array}{c}
|\text{Rel}|^{[\Gamma]} \xrightarrow{\downarrow [t]_r} \text{Rel} \\
|U|[\Gamma] \xrightarrow{\downarrow [\Delta]_r} \text{Set} \times \text{Set} \\
|\text{Set}|^{[\Gamma]} \times |\text{Set}|^{[\Gamma]} \xrightarrow{\downarrow [t]_o \times [t]_o} \text{Set} \times \text{Set}
\end{array}
\]

It is worthwhile to unpack the fibrational statement of the theorem: Since the domains of the functors \( [[\Delta]]_o \) and \( [[T]]_o \) are discrete, the interpretation \( [t]_o \) actually defines a (vacuously natural) transformation \( [t]_o : [[\Delta]]_o \to [[T]]_o \). By the definition of morphisms in the category \( \text{Rel} \), the existence of the (again, vacuously natural) transformation \( [t]_r \) over \( [t]_o \times [t]_o \) is exactly the statement that if \( (a, b) \in [[\Delta]]_r R \), then \( ([t]_o A a, [t]_o B b) \in [[T]]_r R \) — the verbose conclusion of the theorem.

Reynolds’ original formulation of the Abstraction Theorem makes it seem at first glance as though it asserts a property of \( [t]_o \). Surprisingly, however, our fibrational version makes it clear that the Abstraction Theorem actually states the existence of additional algebraic structure given by \( [t]_r \) and, more generally, the interpretation of terms as fibred natural transformations. Taking this point of view and exposing this heretofore hidden structure opens the way to our bifibrational generalisation of Reynolds’ model.

4 Bifibrational Relational Parametricity

Thus far we have only shown how to view Reynolds’ notion of parametricity in terms of the specific fibration \( U : \text{Rel} \to \text{Set} \times \text{Set} \). We now generalise this to other fibrations. This requires that we generalise \( [[-]]_o \) and \( [[-]]_r \) in such a way that the IEL and the Abstraction Theorem hold, which in turn requires that we define equality functors for these other fibrations. The construction of equality functors is standard in any fibration with the necessary infrastructure [16], but we briefly describe it here for completeness. The first step is to note that the relations fibration from Example 2.5 arises from the subobject fibration over \( \text{Set} \) by so-called change of base (or pullback), and to generalise that construction.
Definition 4.1 Let $U : \mathcal{E} \to \mathcal{B}$ be a fibration and suppose $\mathcal{B}$ has products. The fibration $\text{Rel}(U) : \text{Rel}(\mathcal{E}) \to \mathcal{B} \times \mathcal{B}$ is defined by the following change of base:

$$
\begin{array}{c}
\text{Rel}(\mathcal{E}) \xrightarrow{q} \mathcal{E} \\
\text{Rel}(U) \downarrow \downarrow \downarrow U \\
\mathcal{B} \times \mathcal{B} \xrightarrow{\times} \mathcal{B}
\end{array}
$$

We call $\text{Rel}(U)$ the relations fibration for $U$, and call the objects of $\text{Rel}(\mathcal{E})$ relations on $\mathcal{B}$, to emphasise that this construction generalises the relations fibration on $\text{Set}$. We say that a fibration $U : \mathcal{E} \to \mathcal{B}$ has fibred terminal objects if each fibre $\mathcal{E}_X$ of $\mathcal{E}$ has a terminal object, and if reindexing preserves these terminal objects. The map sending each object $X$ of $\mathcal{B}$ to the terminal object in $\mathcal{E}_X$ extends to a functor $K : \mathcal{B} \to \mathcal{E}$ called the truth functor for $U$. We can construct an equality functor for $\text{Rel}(U)$ from the truth functor for $U$ as follows:

Definition 4.2 Let $U : \mathcal{E} \to \mathcal{B}$ be a bifibration with fibred terminal objects. If $\mathcal{B}$ has products, then the map $X \mapsto \Sigma_{\delta_X}KX$, where $\delta_X : X \to X \times X$, extends to the equality functor $\text{Eq} : \mathcal{B} \to \text{Rel}(\mathcal{E})$ for $\text{Rel}(U)$.

For this definition, it is enough to ask for opreindexing along diagonals $\delta_X$ only (this is what Birkedal and Møgelberg [4] do to model equality). When dealing with graph relations in Section 5.1, though, we will use all of the opfibrational structure to opreindex along arbitrary morphisms. Our definition specialises to the equality relation $\text{Eq}A$ when instantiated to the relations fibration on $\text{Set}$. This equality functor is faithful, but not always full; a counterexample is the equality functor for the identity bifibration $\text{Id} : \text{Set} \to \text{Set}$, which gives a model with ad hoc, rather than parametric, polymorphic functions. We thus assume in the rest of this paper that equality functors are full. This is reminiscent of Birkedal and Møgelberg’s [4] assumption that the fibre has very strong equality, i.e., that internal equality implies external equality, in the following sense: fullness says that if $(f, g, \alpha) : 1 \to \text{Eq}Y$ (i.e., $\alpha$ shows that $f = g$ internally), then, since $1 = \text{Eq}1, (f, g, \alpha) = (h, h, \text{Eq}h)$ for some $h : 1 \to Y$ (i.e., $f = g$ externally). We use fullness of $\text{Eq}$ at several places in Section 5 below.

We now show how to interpret arrow types and forall types as fibred functors with discrete domains. We then show that a particular class of such functors forms a $\lambda$2-fibration and thus a model of System F which is, in fact, parametric.

4.1 Interpreting Arrow Types

The definition of $[T \to U]_o$ and $[T \to U]_r$ in Section 3.1 is derived from the cartesian closed structure of $\text{Set}$ and $\text{Rel}$, respectively. Moreover, the fibration $U : \text{Rel} \to \text{Set} \times \text{Set}$ preserves the cartesian closed structure, so that $[t]_r$ is over $[t]_o \times [t]_o$ as required by the Abstraction Theorem. Generalising from this fibration, we can model arrow types “parametrically”—i.e., in a way satisfying the Abstraction Theorem—in any fibration $U : \mathcal{E} \to \mathcal{B}$ in which $\mathcal{E}$ and $\mathcal{B}$ are cartesian closed categories (CCCs) and $U$ preserves cartesian closedness.
Definition 4.3 A fibration $U: \mathcal{E} \to \mathcal{B}$ is an arrow fibration if both $\mathcal{E}$ and $\mathcal{B}$ are CCCs, and $U$ preserves the cartesian closed structure. A relations fibration $\text{Rel}(U)$ is an equality preserving arrow fibration if it is an arrow fibration and $\text{Eq}: \mathcal{B} \to \text{Rel}(\mathcal{E})$ preserves exponentials.

One advantage of working with well-studied mathematical structures such as fibrations is that many of their properties can be found in the literature. This helps in determining when a relations fibration is an equality preserving arrow fibration:

Lemma 4.4 Let $U: \mathcal{E} \to \mathcal{B}$ be a bifibration with fibred terminal objects and $\mathcal{B}$ be a CCC.

(i) If $\text{Eq}: \mathcal{B} \to \text{Rel}(\mathcal{E})$ has a left adjoint $Q$, then $\text{Eq}$ preserves exponentials iff $Q$ satisfies the Frobenius property. Such a $Q$ exists if $U: \mathcal{E} \to \mathcal{B}$ has full comprehension, $\text{Eq}: \mathcal{B} \to \text{Rel}(\mathcal{E})$ is full and $\mathcal{B}$ has pushouts.

(ii) If $U: \mathcal{E} \to \mathcal{B}$ is a fibred CCC and has simple products (i.e., if, for every projection $\pi_B: A \times B \to A$ in $\mathcal{B}$, the reindexing functor $\pi^*_B$ has a right adjoint and the Beck-Chevalley condition holds), then $\mathcal{E}$ is a CCC and $U$ preserves the cartesian closed structure. \hfill $\Box$

Change of base preserves simple products and fibred structure, so $\text{Rel}(U)$ is a fibred CCC with simple products if $U$ is. Moreover, $\mathcal{B} \times \mathcal{B}$ is a CCC if $\mathcal{B}$ is. Lemma 4.4 thus derives structure in $\text{Rel}(U)$ from structure in $U$.

4.2 Interpreting Forall Types

We must generalise Reynolds’ definitions of $[-]_o$ and $[-]_r$ for forall types to relations fibrations in such a way that the Abstraction Theorem and IEL hold. The rules for type abstraction and type application suggest that we should interpret $\forall$ as right adjoint to weakening by a type variable. We may first try to look for such an adjoint on the base category, then another on the total category, and then try to link these adjoints. But this is the wrong idea; for the relations fibration of Example 2.5, this gives all polymorphic functions, not just the parametrically polymorphic ones. Instead, we require an adjoint for the combined fibred semantics.

Let $|\text{Rel}(U)|^n \xrightarrow{\text{Eq}} \text{Rel}(U)$ be the category whose objects are equality preserving fibred functors from $|\text{Rel}(U)|^n$ to $\text{Rel}(U)$ and whose morphisms are fibred natural transformations between them. Then:

Definition 4.5 $\text{Rel}(U)$ is a $\forall$-fibration if, for every projection $\pi_n: |\text{Rel}(U)|^{n+1} \to |\text{Rel}(U)|^n$, the functor $\bigcirc \pi_n: (|\text{Rel}(U)|^n \text{Rel}(U)) \to (|\text{Rel}(U)|^{n+1} \text{Rel}(U))$ has a right adjoint $\forall_n$ and this family of adjunctions is natural in $n$.

We write $\forall$ for $\forall_n$ when $n$ can be inferred. This definition follows, e.g., Dunphy and Reddy [7] by “baking the Identity Extension Lemma into” the definition of forall types — in the sense that the very existence of $\forall$ requires that if $F$ is equality preserving then so is $\forall F$ — rather than relegating it to a result to be proved post facto. If $U$ is faithful, then Definition 4.5 can be reformulated in terms of more basic concepts using its opfibrational structure. The IEL then becomes a consequence of the definition, rather than an intrinsic part of it [9]. For the purposes of this paper, this abstract specification is enough.
4.3 Fibred functors with discrete domains form a parametric model

A λ2-fibration, i.e., a fibration \( p : G \to S \) with fibred finite products, finite products in \( S \), fibred exponents, a generic object \( \Omega \), and simple \( \Omega \)-products, is a categorical model of System F. Seely [29] gives a sound interpretation of the calculus in such fibrations. We conclude this section with the following theorem:

**Theorem 4.6** If \( \text{Rel}(U) \) is an equality preserving arrow fibration and a ∀-fibration, then there is a λ2-fibration in which types \( \Gamma \vdash T \) are interpreted as equality preserving fibred functors \( [T] : |\text{Rel}(U)|^{|\Gamma|} \to \text{Rel}(U) \) and terms \( \Gamma; \Delta \vdash t : T \) are interpreted as fibred natural transformations \( [t] : [\Delta] \to [T] \).

Note that Lemma 4.4 gives conditions for \( \text{Rel}(U) \) to be an arrow fibration, and our other paper [9] similarly gives conditions for \( \text{Rel}(U) \) to be a ∀-fibration. Unwinding the interpretation of System F in λ2-fibrations [29], we see that we get the following for every fibration \( \mathcal{U} \to \mathcal{E} \) satisfying the hypotheses of the theorem: for every System F type \( \Gamma \vdash T \) and term \( \Gamma; \Delta \vdash t : T \), we get

(i) a standard interpretation of \( \Gamma \vdash T \) as a functor \([T] : [\text{Rel}(U)]_{\Gamma} \to B\);

(ii) a relational interpretation of \( \Gamma \vdash T \) as a functor \([T] : |\text{Rel}(_E)|^{|\Gamma|} \to \text{Rel}(_E)\);

(iii) a proof of the Identity Extension Lemma in the form of Lemma 3.2, i.e., a proof that \([T] \) is equality preserving;

(iv) a standard interpretation of \( \Gamma; \Delta \vdash t : T \) as a natural transformation \( [t] : [\Delta] \to [T] \); and

(v) a proof of the Abstraction Theorem in the form of Theorem 3.3, i.e., a proof that \( \Gamma; \Delta \vdash t : T \) has a relational interpretation as a natural transformation \( [t] : [\Delta] \to [T] \) over \([t] \times [t] \).

**Theorem 4.6** also gives a powerful internal language [16], where base types in type context \( \Gamma \) are given by fibred functors \( |\text{Rel}(U)|^{|\Gamma|} \to \text{Rel}(U) \), and base term constants in term context \( \Delta \) are given by fibred natural transformations \( [\Delta] \to [T] \).

Thus, we can use this language to reason about our models using System F. This will be used in the proofs of Theorems 5.7 and 5.11 below.

5 Consequences of parametricity

We use our new framework to derive expected consequences of parametricity. This serves as a “sanity check” for our new bifibrational conceptualisation, and shows that our framework is powerful enough to derive the same results as, e.g., Birkedal and Møgelberg [4]. At a high-level, our proof strategies are often similar to the ones found in the literature, while the proofs of individual facts are necessarily specific to our setting, and often fibrational in nature.

5.1 Graph Relations

In the fibration \( \mathcal{U} : \text{Rel} \to \mathcal{E} \) every function \( f : X \to Y \) defines a graph relation \( (f) = \{(x, y) \mid fx = y\} \subseteq X \times Y \). This generalises to the fibrational setting, where the graph of \( f : A \to B \) is obtained by reindexing the equality relation on \( B \).
Definition 5.1 Let $U : \mathcal{E} \to \mathcal{B}$ be a fibration with fibred terminal objects and products in $\mathcal{B}$. The graph of $h : X \to Y$ in $\mathcal{B}$ is $\langle h \rangle = (h, \text{id}_Y)^*(\text{Eq } Y)$ in $\text{Rel}(\mathcal{E})$.

The definition of $\langle h \rangle$ agrees with the set-theoretic one for the relations fibration on $\text{Set}$. Since reindexing preserves identities, $\langle \text{id}_A \rangle = (\text{id}_A, \text{id}_A)^*(\text{Eq } A) = \text{Eq } A$ for any object $A$ of $\mathcal{B}$. In a fibration, we can also define the graph of $f : A \to B$ in another, isomorphic way by using opfibrational structure to opreindex equality on $A$:

**Lemma 5.2 (Lawvere [17])** If $U : \mathcal{E} \to \mathcal{B}$ is a fibration with fibred terminal objects that satisfies the Beck-Chevalley condition [16, Section 1.8.11], and if $\mathcal{B}$ has products, then the graph of $h : X \to Y$ can also be described by $\langle h \rangle = \Sigma(\text{id}_X, h)(\text{Eq } X)$. \square

Being able to describe graph relations in terms of either reindexing or opreindexing in any fibration lets us use the universal properties of both when proving theorems about them. Graph relations are the key structures that turn morphisms in $\mathcal{B}$ into objects in $\text{Rel}(\mathcal{E})$ and, more generally, mediate the standard and relational semantics.

The graph functor for $\text{Rel}(U) : \text{Rel}(\mathcal{E}) \to \mathcal{B} \times \mathcal{B}$ is the functor $\langle \_ \rangle : \mathcal{B}^\to \to \text{Rel}(\mathcal{E})$ mapping $f : X \to Y$ in $\mathcal{B}$ to $\langle f \rangle$ in $\text{Rel}(\mathcal{E})$. To see how $\langle \_ \rangle$ acts on morphisms, recall that if $f : X \to Y$ and $f' : X' \to Y'$ are objects of $\mathcal{B}^\to$, then a morphism from $f$ to $f'$ is a pair of morphisms $g : X \to X'$ and $h : Y \to Y'$ such that $f' \circ g = h \circ f$.

The universal property of reindexing in $\text{Rel}(U)$ guarantees the existence of a unique morphism $\langle g, h \rangle : \langle f \rangle \to \langle f' \rangle$ over $(g, h)$ such that the following diagram commutes:

\[
\begin{array}{c}
\langle f \rangle \xrightarrow{(f, \text{id})} \text{Eq } Y \\
\exists (g, h) \downarrow \downarrow \downarrow \\
\langle f' \rangle \xrightarrow{(f', \text{id})} \text{Eq } Y'
\end{array}
\]

**Lemma 5.3** If the underlying bifibration satisfies the Beck-Chevalley condition, then $\langle \_ \rangle : \mathcal{B}^\to \to \text{Rel}(\mathcal{E})$ is full and faithful if $\text{Eq} : \mathcal{B} \to \text{Rel}(\mathcal{E})$ is. \square

The proof uses the opfibrational characterisation of the graph functor from Lemma 5.2. The main tool for deriving consequences of parametricity is the Graph Lemma, which relates the graph of the action of a functor on a morphism with its relational action on the graph of the morphism. Interestingly, although our setting is possibly proof-relevant (i.e., there can be multiple proofs that two elements are related), the following “logical equivalence” version of the Graph Lemma is strong enough for our applications. If $U : \mathcal{E} \to \mathcal{B}$ and $U' : \mathcal{E}' \to \mathcal{B}'$ are fibrations, we write $(F_o, F_r) : \text{Rel}(U) \to \text{Eq} \text{Rel}(U')$ to indicate that functors (not necessarily fibred) $F_o : \mathcal{B} \to \mathcal{B}'$ and $F_r : \text{Rel}(\mathcal{E}) \to \text{Rel}(\mathcal{E}')$ are such that $\text{Rel}(U') \circ F_r = (F_o \times F_o) \circ \text{Rel}(U)$, and $(F_o, F_r)$ is equality preserving, i.e., $F_r \circ \text{Eq} = \text{Eq} \circ F_o$.

**Theorem 5.4 (Graph Lemma)** Assume the underlying bifibration satisfies the Beck-Chevalley condition, and let $(F_o, F_r) : \text{Rel}(U) \to \text{Eq} \text{Rel}(U)$. For any $h : X \to Y$ in $\mathcal{B}$, there are vertical morphisms $\phi_h : \langle F_o h \rangle \to F_r \langle h \rangle$ and $\psi_h : F_r \langle h \rangle \to \langle F_o h \rangle$ in $\text{Rel}(\mathcal{E})$. \square

Our proof of the Graph Lemma is completely independent of the specific functor $(F_o, F_r)$, and so in particular does not proceed by induction on the structure of
types. This is a key reason why we can go beyond Dunphy and Reddy [7] and prove the existence of initial algebras of positive, rather than just strictly positive, type expressions.

5.2 Existence of Initial Algebras

Let $F : C \rightarrow C$ be an endofunctor. An $F$-algebra is a pair $(A, k_A)$ with $A$ an object of $C$ and $k_A : FA \rightarrow A$ a morphism. We call $A$ the carrier of the $F$-algebra and $k_A$ its structure map. A morphism $h : A \rightarrow B$ in $C$ is an $F$-algebra homomorphism $h : (A, k_A) \rightarrow (B, k_B)$ if $k_B \circ (Fh) = h \circ k_A$. An $F$-algebra $(Z, in)$ is weakly initial if, for any $F$-algebra $(A, k_A)$, there exists a mediating $F$-algebra homomorphism $fold[A, k_A] : (Z, in) \rightarrow (A, k_A)$. It is an initial $F$-algebra if $fold[A, k_A]$ is unique.

The literature contains other proofs that initial algebras exist in parametric models (e.g., [4,24]). Closest to our setting is Dunphy and Reddy [7], who show that strictly positive types have initial algebras. Under assumptions no stronger than theirs, we sharpen this result to all positive types, or, more generally, all functors on our parametric models that are strong (see below) and equality preserving.

Let $F = (F_o, F_r) : \text{Rel}(U) \rightarrow \text{Eq} \\text{Rel}(U)$ be a functor (note that the domain of $F$ is not discrete and that $F$ need not preserve cartesian morphisms) with a strength $t = (t_o, t_r)$, i.e., a family of morphisms $(t_o)_{A,B} : A \Rightarrow B \Rightarrow F o A \Rightarrow F o B$ and $(t_r)_{R,S} : R \Rightarrow S \Rightarrow F_r R \Rightarrow F_r S$ with $(t_r)_{R,S}$ over $((t_o)_{A,B}, (t_o)_{C,D})$ if $R$ is over $(A, B)$ and $S$ is over $(C, D)$, such that $t$ preserves identity and composition. A functor with a strength is said to be strong. Because of the discrete domains, $t$ is a natural transformation from $\Rightarrow$ to $\Rightarrow$ in $|\text{Rel}(U)|^2 \rightarrow \text{Eq} \\text{Rel}(U)$, and thus $\alpha, \beta : t : (\alpha \Rightarrow \beta) \rightarrow (F[\alpha] \Rightarrow F[\beta])$ represents the action of $F$ on morphisms in the internal language. All type expressions with one free type variable occurring only positively give rise to strong functors, but there are further examples of such functors, for instance if the model contains non-System F type constructions with natural functorial (and relational) interpretations — for example, those of dependent types in $\text{Set}$. We will define $Z$ by $(Z_o, Z_r) = [\forall X. (F X \rightarrow X) \rightarrow X]$.

**Lemma 5.5** $Z_o$ is the carrier of a weak initial $F_o$-algebra $(Z_o, in_o)$ with mediating morphism $fold_o[A, k]$ and $Z_r$ is the carrier of a weak initial $F_r$-algebra $(Z_r, in_r)$ with mediating morphism $fold_r[A, k]$. $\square$

To show that $fold_o$ is unique, we use the graph relations from Section 5.1. Recall that a category with a terminal object $1$ is well-pointed if, for any $f, g : A \rightarrow B$, we have $f = g$ if $f \circ e = g \circ e$ for all $e : 1 \rightarrow A$. Like Dunphy and Reddy [7], we only consider well-pointed base categories; well-pointedness is used to convert internal language reasoning in non-empty contexts to closed contexts, so that we can apply semantic techniques such as Theorem 5.4.

**Lemma 5.6** Assume that the underlying bifibration satisfies the Beck-Chevalley condition, and that $\text{Eq}$ is full.

(i) If $B$ is well-pointed, then $fold_o[Z_o, in_o] = id_Z$.  

77
(ii) For every $F_o$-algebra homomorphism $h : (Z_o, in_o) \to (A, k_A)$, we have that $h \circ \text{fold}_o[Z_o, in_o] = \text{fold}_o[A, k_A]$. \hfill $\square$

The proofs of the two parts of Lemma 5.6 are similar: both use the graph functor to map commuting diagrams in $B$ to morphisms in $\text{Rel}(E)$, and then use the Graph Lemma to see that these morphisms are $F_r$-algebras. Lemma 5.5 and Lemma 5.6 together now immediately imply the main result.

**Theorem 5.7** If the underlying bifibration satisfies the Beck-Chevalley condition, $\text{Eq}$ is full, and $B$ is well-pointed, then $(Z_o, in_o)$ is an initial $F_o$-algebra. \hfill $\square$

We show in Section 6 that these hypothesis cannot be weakened. One may wonder if the above result can be strengthened to get not only an initial $F_o$-algebra, but also an initial $F_r$-algebra. Certainly this is possible for the relations fibration $\text{Rel} \to \text{Set}$, since relations in $\text{Rel}$ are proof irrelevant: maps either preserve relatedness or not. This translates in the axiomatic bifibrational setting to requiring the fibration $\text{Rel}(E) \to B \times B$ to be faithful. When it is, the weak initial $F_r$-algebra is, in fact, initial: faithfulness implies the required uniqueness.

### 5.3 Existence of final coalgebras

We can also dualise the proof from the previous section to show the existence of final coalgebras in the usual manner [11]. As usual, this requires us to first encode products and existential types in System F. We encode products as $A \times B \to A \to B \to Y \to Y$. This supports the usual pairing and projection operations, as well as surjective pairing using parametricity. We encode existential types by $\exists X. T \to \forall X. (\forall X. (T \to Y)) \to Y$. We can support introduction and elimination rules

\[
\begin{align*}
\Gamma \vdash A \text{ type} & \quad \Gamma; \Delta \vdash u : T[A/X] \\
\hline
\Gamma; \Delta \vdash \langle A, u \rangle : \exists X. T(X)
\end{align*}
\]

with the conversion $\text{open} (A, t) \text{ as } (Z, y) \text{ in } s = s[X/A, y/t]$ by defining $(A, t) = \Lambda Y. \lambda f. f A t$ and $\text{open} t \text{ as } (Z, y) \text{ in } s = t V (\Lambda Z. \lambda y. s)$. Using parametricity we can prove the following commutation property and $\eta$-rule for existential types:

**Lemma 5.8** Assume the underlying bifibration satisfies the Beck-Chevalley condition, and that $\text{Eq}$ is full.

(i) Let $\Gamma; \Delta \vdash t : \exists X. T$, let $\Gamma, Z; \Delta, u : T[Z/X] \vdash s : S$ and let $\Gamma; \Delta \vdash f : S \to S'$ for a closed type $S'$. Then $[f(\text{open} t \text{ as } (Z, u) \text{ in } s)]_o = [\text{open} t \text{ as } (Z, u) \text{ in } f(s)]_o$.

(ii) If $\Delta; \Gamma \vdash t : \exists X. T$, then $[\text{open} t \text{ as } (Z, u) \text{ in } (Z, u)]_o = [t]_o$. \hfill $\square$

If $F : C \to C$ is an endofunctor, an $F$-coalgebra is a pair $(A, k_A)$ with $A$ an object of $C$ and $k_A : A \to FA$ a morphism. We call $A$ the carrier of the $F$-coalgebra and $k_A$ its structure map. A morphism $h : A \to B$ in $C$ is an $F$-coalgebra homomorphism if $h \circ k_A = (B, k_B)$. An $F$-coalgebra $(W, out)$ is weakly final if, for any $F$-coalgebra $(A, k_A)$, there exists a mediating $F$-coalgebra homomorphism $\text{unfold}[A, k_A] : (A, k_A) \to (W, out)$. It is a final $F$-coalgebra if $\text{unfold}[A, k_A]$ is unique.

Let $F = (F_o, F_r) : \text{Rel}(U) \to \text{Eq} \text{Rel}(U)$ be a functor with a strength $t$. We show that the final $F_o$-coalgebra exists. Again, we first construct a weakly final coalgebra.
by defining $W = (W_0, W_r) = \{ X \to F(X) \times X \}$.

**Lemma 5.9** $K_o$ is the carrier of a weakly final $F_o$-coalgebra $(K_o, out_o)$ with mediating morphism $\text{unfold}_o[A, k]$ and $K_r$ is the carrier of a weakly final $F_r$-coalgebra $(K_r, out_r)$ with mediating morphism $\text{unfold}_r[A, k]$.

We proceed similarly to Lemma 5.6. This time, we use the opfibrational part of the Graph Lemma to construct $F_r$-coalgebras.

**Lemma 5.10** Assume the underlying bifibration satisfies the Beck-Chevalley condition, and that $\text{Eq}$ is full.

(i) For every $F_o$-coalgebra morphism $h : (A, k_A) \to (B, k_B)$ we have $\text{unfold}_o[B, k_B] \circ h = \text{unfold}_o[A, k_A]$.

(ii) $\text{unfold}_o[K_o, out_o] = \text{id}_{K_o}$.

Putting things together, we have constructed a final coalgebra.

**Theorem 5.11** If the underlying bifibration satisfies the Beck-Chevalley condition, and if $\text{Eq}$ is full, then $(K_o, out_o)$ is a final $F_o$-coalgebra.

### 5.4 Parametricity Implies Dinaturality

We show that our axiomatic foundations for parametricity can be used to prove that dinaturality can be deduced from parametricity. We do this not because this result is unknown but because i) it shows our foundation passes this test; and ii) it highlights again the use of bifibrations to give two definitions of the graph of a function both of which are used in the proof. First, the definition of dinaturality:

**Definition 5.12** If $F, G : B^{\text{op}} \times B \to B$ are mixed variant functors, then a dinatural transformation $t : F \to G$ is a collection of morphisms $t_X : FXX \to GXX$ indexed by objects $X$ of $B$ such that, for every morphism $g : X \to Y$ of $B$, the following hexagonal diagram commutes:

\[
\begin{array}{c}
F(g, \text{id}_X) \\
\downarrow \\
FYY \\
\downarrow \\
FYX \\
\downarrow \\
F(g, \text{id}_Y) \\
\end{array}
\begin{array}{c}
GX X \\
\downarrow \\
GX Y \\
\downarrow \\
GYY \\
\downarrow \\
GY Y \\
\end{array}
\begin{array}{c}
\text{id}_{GXX} \\
\downarrow \\
\text{id}_{GYY} \\
\downarrow \\
\text{id}_{FYX} \\
\downarrow \\
\text{id}_{FYY} \\
\end{array}
\begin{array}{c}
GX X \\
\downarrow \\
GX Y \\
\downarrow \\
GYY \\
\downarrow \\
GY Y \\
\end{array}
\]

We note that our proof applies to all mixed variant functors with equality preserving liftings, not just strong such functors.

**Theorem 5.13** Let $(F_o, F_r), (G_o, G_r) : \text{Rel}(U)^{\text{op}} \times \text{Rel}(U) \to \text{Eq} \text{Rel}(U)$. Further, let $t^0_o : F_o AA \to G_o AA$ be a family indexed by objects $A$ of $B$, and $t^1_r : F_r RR \to G_r RR$ be a family indexed by objects $R$ of $\text{Rel}(E)$ such that if $R$ is over $(A, B)$, then $t^1_r$ is over $(t^0_o A, t^0_o B)$. Then $t^0_o$ is a dinatural transformation from $F_o$ to $G_o$.

Theorem 5.13 applies in particular to the interpretation of terms $t : \forall X. FXX \to GXX$ where $F$ and $G$ are type expressions with one free type variable. As is well known, dinaturality reduces to naturality when $F$ and $G$ are covariant.
6 Examples

The construction of examples remains delicate — for instance, there are no set-theoretic models with a classical meta-theory. We give five models: Examples 6.1, 6.3, 6.4 and 6.5 are to be regarded as being internal to the Calculus of Constructions with impredicative $\text{Set}$ (with $\rightarrow$-stable equality for Example 6.3), while Example 6.2 is internal to the category of $\omega$-sets.

Before doing so, we take a moment to emphasise the generality of our framework. Considering different fibrations, we can derive parametric models with very different flavours. For example, changing the base category of the fibration corresponds to changing the ‘standard’ model in which we interpret types and terms. Changing the total category and the fibration (i.e., the functor itself) corresponds to changing the relevant notion of relational logic. We take advantage of the possibility of non-standard relations in Examples 6.2, 6.3 and Non-example 6.5.

Example 6.1 Reynolds’ set-theoretic model is an instance of our framework via the relations fibration on $\text{Set}$. The equality functor is full and faithful in this bifibration, and $\text{Set}$ is well-pointed. Hence Theorems 5.7 and 5.13 ensure that initial algebras exist, and that all terms are interpreted as dinatural transformations.

Example 6.2 The PER model of Bainbridge et al. [2] is an instance of our framework, if bifibrations are understood as internal to the category of $\omega$-sets, so that natural transformations are uniformly realised (see also Longo and Moggi [18] for a detailed construction of a model using a category of PERs internal to $\omega$-sets).

An object of the category $\text{PER}_\mathbb{N}$ is a symmetric, transitive relation $R \subseteq \mathbb{N} \times \mathbb{N}$. A morphism from $R$ to $S$ is a function $f : \mathbb{N}/R \to \mathbb{N}/S$ that is tracked by some partial recursive function $\phi_k : \mathbb{N} \to \mathbb{N}$, i.e., such that $f([n]_R) = [\phi_k(n)]_S$ for all $[n]_R \in \mathbb{N}/R$. The appropriate notion of predicate with respect to a PER $R$ is that of a saturated subset, i.e., a subset $P \subseteq \mathbb{N}$ such that $P(x)$ and $R(x, x')$ implies $P(x')$. Saturated subsets form a bifibration over PERs with a full and faithful equality functor $\text{Eq}a = a$. The CCC structure of $\text{PER}_\mathbb{N}$ and $\text{SatRel}$ is standard; a bijective pairing function $(\cdot, \cdot) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ gives the product and recursion theory (the s-m-n Theorem) gives the exponential. The interesting case is that of forall types, which are interpreted as (cut-down, to ensure equality preservingness) intersections of PERs: $[\forall X. F]_o R = \{(n, k) \in \bigcap_{R: \text{PER}_\mathbb{N}} [F]_o ([R]_o R')\}$ for all $Q : \text{SatRel}(S, T). (n, n), (k, k) \in [F]_r (\text{Eq}([R], Q))$ and $[\forall X. F]_r P = \bigcap_{Q: \text{SatRel}(R, S)} [F]_r (P, Q)$. Since $\text{PER}_\mathbb{N}$ is also well-pointed, Theorems 5.7 and 5.13 again apply.

Example 6.3 The previous models are well-known, but our framework also suggests new ones. A relation $R \subseteq X \times Y$ can be understood classically as a function from $X \times Y$ to $\text{Bool}$. (Constructively, this only covers decidable relations.) Here, $\text{Bool}$ can be replaced with any constructively completely distributive [8] non-trivial lattice $\mathcal{V}$ of “truth values”, leading to “multi-valued parametricity”. For instance, the collection $\mathcal{D}(L)$ of all down-closed subsets of a complete lattice $L$ is constructively completely distributive, and classically, we recover $\text{Bool}$ as $\mathcal{D}(1)$. The category $\text{Fam}(\mathcal{V})$ has objects $(A, p)$, where $A$ is a set and $p : A \to \mathcal{V}$ is thought of as a $\mathcal{V}$-valued predicate. The families fibration $\pi : \text{Fam}(\mathcal{V}) \to \text{Set}$ is a bifibration with $\Sigma_f(Q)(y) = \sup_{f(x) = y} Q(x)$, fibred terminal objects $(X, \lambda, \top)$, where $\top$ is the greatest
element of $\mathcal{V}$, and comprehension given by $\{(A,p)\} = p^{-1}(\top)$. Since $\mathcal{V}$ is complete, it is a Heyting algebra, so that $\pi : \text{Fam}(\mathcal{V}) \to \text{Set}$ is a fibred CCC. Also, $\pi$ has simple products given by $\prod_{x \in E} p(a,b) = \inf_{x \in E} p(a,b)$. By Lemma 4.4, $\text{Rel}(\pi)$ is thus an equality preserving arrow fibration. Finally, the interpretation of forall types is given by $\forall X.F \models \bar{A} = \{ f : \prod_{S : \text{Set}} [F]_o(\bar{A}, S) | \inf_{P : X \to Y \to \mathcal{V}} [F]_r(Eq \bar{A}, P) = \top \}$ and $\forall X.F \models \bar{P} = \inf_{Q : X \to Y \to \mathcal{V}} [F]_r(P, Q)$. Distributivity is used to show that this functor is equality preserving. Fullness of $\text{Eq}$ is obvious by $\neg\neg$-stable equality.

The extra conditions we impose in Section 5 really are necessary: the following are examples of $\forall$- and arrow-fibrations where Theorems 5.7 and 5.13 fail.

**Non-example 6.4** Let $G$ be a fixed (non-trivial) group, and consider the category of $G$-sets, i.e., the category with objects $(X, \cdot_X)$, where $X$ is a set and $\cdot_X : G \times X \to X$ is a group action. Morphisms are functions between the carrier sets that respect the group action. Together with group action respecting relations, this forms a bifibration that is a model of System F in the sense of Theorem 4.6. However, the category is not well-pointed, and we can see that this makes Theorem 5.7 fail in our setting: the interpretation of $\forall X.X \to X$ is not the singleton $G$-set $1$ as expected, but instead contains all the elements of the group $G$. We conjecture that this non-example also extends to a constructive treatment of the category of nominal sets [23].

**Non-example 6.5** The identity fibration $\text{Id} : \text{Set} \to \text{Set}$ models *ad hoc* polymorphism: it is a $\forall$- and arrow-fibration, but the equality functor $\text{Eq}X = X \times X$ is not full. This explains why Theorem 5.13 fails: $\forall X.T \models_o$ includes *ad hoc* polymorphic functions, so that e.g. $\forall X.X \to X \models_o$ contains non-natural transformations such as $\eta$, where $\eta_{\text{Bool}}(x) = \neg x$ and $\eta_X(x) = x$ for $X \neq \text{Bool}$.

### 7 Conclusions and future work

Our interpretation of types and terms as fibred functions and fibred natural transformations shows that parametricity entails replacing the usual categorical semantics involving categories, functors, and transformations with one based on fibrations, fibred functors, and fibred transformations. The results in Section 5 show that our new approach based on bifibrations hits the sweet spot of a light structure that still suffices to prove key results. Work is ongoing in using the bifibrational framework to develop new notions such as proof-relevant parametricity, and higher order parametricity with interesting links to cubical sets that also appear in the semantics of Homotopy Type Theory [3].

### Acknowledgement

We thank the reviewers of this and previous versions of the paper for their comments and suggestions. We especially thank Uday Reddy for extremely valuable advice and encouragement. Research supported by EPSRC grants EP/K023837/1 (NG, FNF), EP/M016951/1 (NG), NSF award 1420175 (PJ), and SICSA (FO).
References


Abstract
Guarded recursion is a form of recursion where recursive calls are guarded by delay modalities. Previous work has shown how guarded recursion is useful for constructing logics for reasoning about programming languages with advanced features, as well as for constructing and reasoning about elements of coinductive types. In this paper we investigate how type theory with guarded recursion can be used as a metalanguage for denotational semantics useful both for constructing models and for proving properties of these. We do this by constructing a fairly intensional model of PCF and proving it computationally adequate. The model construction is related to Escardo’s metric model for PCF, but here everything is carried out entirely in type theory with guarded recursion, including the formulation of the operational semantics, the model construction and the proof of adequacy.

Keywords: Denotational semantics, guarded recursion, type theory, PCF, synthetic domain theory

1 Introduction

Variations of type theory with guarded recursive types and guarded recursively defined predicates have proved useful for giving abstract accounts of operationally-based step-indexed models of programming languages with features that are challenging to model, such as recursive types and general references [1,6], countable nondeterminism [7], and concurrency [15]. Following observations of Nakano [13] and Atkey and McBride [2], guarded type theory also offers an attractive type-based approach to (1) ensuring productivity of definitions of elements of coinductive types [12], and (2) proving properties of elements of coinductive types [8]. One
of the key features of guarded type theory is a modality on types, denoted \( \triangleright \) and pronounced ‘later’. This modality is used to guard recursive definitions and the intuition is that elements of type \( \triangleright A \) are elements of \( A \) only available one time step from now.

In this paper, we initiate an exploration of the use of guarded type theory for denotational semantics and use it to further test guarded type theory. More specifically, we present a model of PCF in guarded dependent type theory. To do so we, of course, need a way to represent possibly diverging computations in type theory. Here we follow earlier work of Escardo [10] and Capretta [9] and use a lifting monad \( L \), which allows us to represent a possibly diverging computation of type \( X \) by a function into \( L(X) \). In Capretta’s work, \( L \) is defined using coinductive types. Here, instead, we use a guarded recursive type to define \( L \). Using this approach we get a fairly intensional model of PCF which, intuitively keeps track of the number of computation steps, similar to [10]. We show this formally by proving that the denotational model is adequate with respect to a step-counting operational semantics. The definition of this step-counting operational semantics is delicate — to be able to show adequacy the steps in the operational semantics have to correspond to the abstract notion of time-steps used in the guarded type theory via the \( \triangleright \) operator. Our adequacy result is related to one given by Escardo in [10]. To show adequacy, we define the operational semantics in guarded type theory and also define a logical relation in guarded type theory to relate the operational and denotational semantics. To carry out the logical relations proof, we make crucial use of some novel features of guarded dependent type theory recently proposed in [8], which, intuitively, allow us to reason now about elements that are only available later.

The adequacy result of this paper may be seen as a version of Plotkin’s classic result from domain theory [14] set in guarded type theory. There has been work to formalise domain theory in Coq [4], however, this is difficult due to the use of classical mathematics. In fact, [4] uses a coinductively defined lifting monad similar to that of Capretta [9]. We believe that guarded type theory is more suitable for encoding in proof assistants such as Coq or Agda, and thus this work can be seen as a step towards enabling the use of the models for formal reasoning.

The remainder of the paper is organized as follows. In Section 2 we recall the core parts of guarded dependent type theory and the model thereof in the topos of trees [6,8]. Then we define a step-counting operational semantics of PCF in Section 3 and the denotational semantics is defined in Section 4. We prove adequacy in Section 5. In Section 6 we use the topos of trees model of the guarded type theory to summarize briefly what the results proved in guarded type theory mean externally, in standard set theory. Finally, we conclude and discuss future work in Section 7.

2 Guarded recursion

In this paper we work in a type theory with dependent types, natural numbers, inductive types and guarded recursion. The presentation of the paper will be informal, but the results of the paper can be formalised in gDTT as presented in [8] (we
do not need the □ modality of gDTT). We start by recalling the core of this type theory (as described in [6]), introducing further constructions later on as needed.

A guarded recursive definition is a recursive definition where the recursive calls are guarded by time steps. The time steps are introduced via a type modality ⊢ pronounced ‘later’. If A is a type then ⊢A is the type of elements of A available only one time step from now. The type constructor ⊢ is an applicative functor in the sense of [11], which means that there is a term next: A → ⊢A freezing an element of A so that it can be used one time step from now, and a ‘later application’ ⊙: ⊢(A → B) → ⊢A → ⊢B written infix, satisfying next(f) ⊙ next(t) = next(f(t)) among other axioms (see also [5]). In particular, ⊢ extends to a functor mapping f: A → B to λx: ⊢A.next(f)⊛x.

Recursion on the level of terms is given by a fixed point operator fix: (⊢A → A) → A satisfying f(next(fix(f))) = fix(f). Intuitively, fix can compute the fixed point of any recursive definition, as long as that definition will only look at its argument later. This fixed point combinator is particularly useful in connection with guarded recursive types, i.e., types where the recursion variable occurs only guarded under a ⊢ as, e.g., in the type of guarded streams:

\[ \text{Str}_A^g ≃ A \times ⊢ \text{Str}_A^g \]

The cons operation cons\(^g\) for Str\(^g\)_A has type A → ⊢ Str\(^g\)_A → Str\(^g\)_A. Hence, we can define, e.g., constant streams as constant a = fix(λxs: ⊢ Str\(^g\)_A. cons\(^g\) a xs).

Guarded recursive types can be constructed using universes and fix as we now describe [5]. We shall assume a universe type \(\hat{\mathcal{U}}\) closed under both binary and dependent sums and products as usual, and containing a type of natural numbers. We write \(\hat{\mathbb{N}}\) for the code of natural numbers satisfying El(\(\hat{\mathbb{N}}\)) ≃ \(\mathbb{N}\) and likewise \(\hat{\times}\) for the code of binary products satisfying El(A \(\hat{\times}\) B) ≃ El(A) \(\times\) El(B). The universe is also closed under ⊢ in the sense that there exists an \(\hat{\circ}\): \(\circ\mathcal{U} → \mathcal{U}\) satisfying

\[ \text{El}(\hat{\circ}(\text{next}(A))) ≃ \text{El}(A). \]  

Using these, the type \(\text{Str}_\mathbb{N}^g\) can be defined as El(\(\text{Str}_\mathbb{N}^g\)) where \(\text{Str}_\mathbb{N}^g = \text{fix}(\lambda B: \circ\mathcal{U}. \hat{\mathbb{N}} \hat{\times} \circ\hat{\mathbb{N}})\). Note that this satisfies the expected equality because

\[ \text{El}(\hat{\text{Str}}_\mathbb{N}^g) ≃ \text{El}(\hat{\mathbb{N}} \hat{\circ}\hat{\circ}(\text{next}(\hat{\text{Str}}_\mathbb{N}^g))) ≃ \text{El}(\hat{\mathbb{N}}) \times \text{El}(\hat{\circ}(\text{next}(\hat{\text{Str}}_\mathbb{N}^g))) ≃ \mathbb{N} \circ \circ \text{El}(\hat{\text{Str}}_\mathbb{N}^g) \]

Likewise, guarded recursive (proof-relevant) predicates on a type A, i.e., terms of type A → \(\mathcal{U}\) can be defined using fix as we shall see an example of in Section 5.

Note that we just assume a single universe and that the above only allows us to solve type equations that can be expressed as endomorphisms on this universe. All the type equations considered in this paper are on this form, but we shall not always prove this explicitly, and often work with types rather than codes, in order to keep the presentation simple.

\(^4\) It is also sound to add guarded recursive types as primitives to the type theory without use of universes, see [6]
2.1 The topos of trees model

The type theory gDTT can be modelled in the topos of trees [6], i.e., the category of presheaves over $\omega$, the first infinite ordinal. Since this is a topos, it is a model of extensional type theory. A closed type is modelled as a family of sets $X(n)$ indexed by natural numbers together with restriction maps $r_n : X(n+1) \to X(n)$. We think of $X(n)$ as how the type looks if we have $n$ computation steps to reason about it.

Using the propositions-as-types interpretation, we say that $X$ is true at stage $n$ if $X(n)$ is inhabited. Note that if $X$ is true at stage $n$, it is also true at stage $k$ for all $k \leq n$. Thus, the intuition of this model is that a proposition is initially considered true and can only be falsified by further computation.

In the topos of trees model, the $\vdash$ modality is interpreted as $\vdash X(0) = 1$ and $\vdash X(n+1) = X(n)$, i.e., from the logical point of view, the $\vdash$ modality delays evaluation of a proposition by one time step. For example, if 0 is the constantly empty presheaf (corresponding to a false proposition), then $\vdash 0$ is the proposition that appears true for the first $n$ computation steps and is falsified after $n+1$ steps.

3 PCF

This section defines the syntax, typing judgements, and operational semantics of PCF. These should be read as judgements in guarded type theory, but as stated above we work informally in type theory, which here means that we ignore standard problems of representing syntax up to $\alpha$-equality. Note that this is a perpendicular issue to the one we are trying to solve here.

Unlike the operational semantics to be defined below, the typing judgements of PCF are defined in an entirely standard way, see Figure 1. In the figure, $v$ ranges over values of PCF, i.e., terms of the form $v = n$, where $n$ is a natural number or $v = \lambda x.M$. Note that we distinguish notationally between a natural number $n$ and the corresponding PCF value $n$. We denote by $\text{Type}_{PCF}$, $\text{Term}_{PCF}$, and $\text{Value}_{PCF}$ the types of PCF types, closed terms, and closed values of PCF.

3.1 Big-step semantics

The big-step operational semantics defined in Figure 2 is a relation between terms, numbers and predicates on values. The statement $M \Downarrow^k Q$ should be read as $M$ evaluates in $k$ steps to a value satisfying $Q$. The relation can either be defined

\[
\begin{align*}
\Gamma, x : \sigma, \Delta \vdash x : \sigma & \quad \Gamma \vdash (\lambda x : \sigma.M) : \sigma \to \tau \\
\Gamma \vdash M : \sigma \to \tau & \quad \Gamma \vdash N : \sigma \\
\Gamma \vdash MN : \tau & \\
\Gamma \vdash M : \text{nat} & \quad \Gamma \vdash N : \text{nat} \\
\Gamma \vdash Y_{\sigma} : \text{nat} & \\
\Gamma \vdash \text{ifz} L M N : \sigma & \\
\end{align*}
\]

Fig. 1. PCF typing rules
\[ \triangleright : \text{Term}_{\text{PCF}} \times \mathbb{N} \times (\text{Value}_{\text{PCF}} \to \mathcal{U}) \to \mathcal{U} \]

\[ v \downarrow^0 Q \overset{\text{def}}{=} Q(v) \]

\[ \text{pred } M \downarrow^k Q \overset{\text{def}}{=} M \downarrow^k (\lambda x. \Sigma n : \mathbb{N}. x = n \text{ and } Q(n-1)) \]

\[ \text{succ } M \downarrow^k Q \overset{\text{def}}{=} M \downarrow^k (\lambda x. \Sigma n : \mathbb{N}. x = n \text{ and } Q(n+1)) \]

\[ Y_\sigma M \downarrow^{k+1} Q \overset{\text{def}}{=} (M(Y_\sigma M) \downarrow^k Q) \]

\[ MN \downarrow^{k+m} Q \overset{\text{def}}{=} M \downarrow^k Q' \]

where \( Q'(\lambda x. L) = L[N/x] \downarrow^m Q \)

\[ \text{ifz } L M N \downarrow^{k+m} Q \overset{\text{def}}{=} L \downarrow^k Q' \]

where \( Q'(\underline{0}) = M \downarrow^m Q \) and \( Q'(n+1) = N \downarrow^m Q \)

Fig. 2. Step-indexed Big-Step Operational Semantics for PCF

by a combination of guarded recursion and induction on \( M \), or simply by ordinary induction first on \( k \) then on \( M \).

Figure 2 uses standard syntactic sugar, for example, only non-empty cases are mentioned, e.g., \( v \downarrow^k Q \) is defined to be 0 in case \( k > 0 \), and the case of function application should be read as

\[ MN \downarrow^l Q \overset{\text{def}}{=} \Sigma k, m : \mathbb{N}. (k + m = l) \text{ and } M \downarrow^k Q' \]

Note in particular that this means that \( Y_\sigma M \downarrow^0 Q \) is always false.

As mentioned in the introduction, the formulation of the big-step operational semantics is quite delicate – the wrong definition will make the adequacy theorem false. First of all, the definition must ensure that the steps of PCF are synchronised with the steps on the meta level. This is the reason for the use of \( \triangleright \) in the case of the fixed point combinator. Secondly, the use of predicates on values on the right hand side of \( \downarrow \) rather than simply values is necessary to ensure that the right hand side is not looked at before the term is fully evaluated. For example, a naive definition of the operational semantics using values on the right hand side and the rule

\[ \text{succ } M \downarrow^k v \overset{\text{def}}{=} \Sigma n : \mathbb{N}. (v = n + 1) \text{ and } M \downarrow^k n \]

Would make \( \text{succ } (Y_N (\lambda x : \mathbb{N}. x)) \downarrow^{42} 0 \) false, but to obtain computational adequacy, we need this statement to be true for the first 42 steps before being falsified. (For an explanation of this point, see Remark 5.8 below.) In general, \( M \downarrow^k Q \) should be defined in such a way that in the topos of trees model it is true at stage \( n \) (using vocabulary from Section 2.1) iff either

- \( k < n \) and \( M \) evaluates in precisely \( k \) steps to a value satisfying \( Q \), or
- \( k \geq n \) and evaluation of \( M \) takes more than \( k \) steps.

In particular, if \( M \) diverges, then \( M \downarrow^k Q \) should be true at stages \( n \leq k \) false for \( n > k \).

The use of predicates means that partial results of term evaluation are ignored,
Fig. 3. Step-Indexed Small Step semantics of PCF. In the rules, \( k \) can be 0 or 1.

and comparison of the result to the right hand side of \( \llcorner \) is postponed until evaluation of the term is complete. The more standard big-step evaluation of terms to values can be defined as

\[ M \llcorner^k v \overset{\text{def}}{=} M \llcorner^k \lambda v. v = v \]

### 3.2 Small-step semantics

Figure 3 defines the small-step operational semantics. Just like the big step semantics, the small step semantics counts unfoldings of fixed points. The small steps semantics will be proved equivalent to the big-step semantics, but is introduced because it is more suitable for the proofs of soundness and computational adequacy.

Note the following easy lemma.

**Lemma 3.1** The small-step semantics is deterministic: if \( M \rightarrow^k N \) and \( M \rightarrow^{k'} N' \), then \( k = k' \) and \( N = N' \).

The transitive closure of the small step semantics is defined using \( \triangleright \) to ensure that the steps of PCF are synchronised with the steps of the meta language.

**Definition 3.2** Denote by \( \rightarrow^0 \) the reflexive, transitive closure of \( \rightarrow^0 \). The closure of the small step semantics, written \( M \Rightarrow^k Q \) is a relation between closed terms, natural numbers, and predicates on closed terms, defined by induction on \( k \) as

\[ M \Rightarrow^0 Q \overset{\text{def}}{=} \Sigma N : \text{Term}_{\text{pcr}} \rightarrow^0 N \text{ and } Q(N) \]

\[ M \Rightarrow^{k+1} Q \overset{\text{def}}{=} \Sigma M', M'' : \text{Term}_{\text{pcr}} \rightarrow^0 M' \text{ and } M' \rightarrow^1 M'' \text{ and } \triangleright(M'' \Rightarrow^k Q) \]

Similarly to the case of the big-step semantics we define \( M \Rightarrow^k v \overset{\text{def}}{=} M \Rightarrow^k \lambda N.v = N \)

We will now prove the correspondence between the big-step and the small step operational semantics. First we need the following lemma.

**Lemma 3.3** Let \( M, N \) be closed terms of type \( \tau \), and let \( Q : \text{Term}_{\text{pcr}} \rightarrow U \).

(i) If \( M \rightarrow^0 N \) and \( N \llcorner^k Q \) then \( M \llcorner^k Q \)
(ii) If $M \rightarrow^1 N$ and $\triangleright(N \triangleright^k Q)$ then $M \triangleright^{k+1} Q$

Proof sketch

(i) By induction on $M \rightarrow^0 N$. We consider the case ifz $L M N \rightarrow^0$ ifz $L' M N$. Assume ifz $L' M N \triangleright^k Q$. By definition $L' \triangleright^k Q'$. By induction hypothesis $L \triangleright^k Q'$ and by definition ifz $L M N \triangleright^k Q$. All the other cases are similar.

(ii) By induction on $M \rightarrow^1 N$. The base case is $Y_{\sigma} M \rightarrow^1 M(Y_{\sigma} M)$. Assume $\triangleright(M(Y_{\sigma} M) \triangleright^k Q)$. Then by definition $Y_{\sigma} M \triangleright^k Q$. We consider now the inductive cases pred $M \rightarrow^1$ pred $M'$. Assume $\triangleright$(pred $M' \triangleright^k Q$). By definition $\triangleright(M' \triangleright^k \lambda x.Q(x - 1))$ and by induction hypothesis $M \triangleright^{k+1} \lambda x.Q(x - 1)$. By definition pred $M \triangleright^k Q$.

Lemma 3.4 Let $M$ be a closed term and $Q : \text{Value}_{\text{PCF}} \rightarrow \mathcal{U}$ a relation on values. If $M \triangleright^k (\lambda N. N \triangleright^m Q)$ then $M \triangleright^{k+m} Q$

Proof. The proof is by induction on $k$. In the case where $k = k' + 1$ we have as assumptions that $M \rightarrow^0 N$ and $N \rightarrow^1 N'$ and $\triangleright(N' \triangleright^{k'+m} (\lambda N. N \triangleright^m Q))$. By induction we have $\triangleright(N' \triangleright^{k+m} Q)$ and now by repeated application of Lemma 3.3 also $M \triangleright^{k+m} Q$ as desired.

Now we can state the correspondence. Note that we have to massage the predicate of the $\Rightarrow$ relation to make things type check properly.

Lemma 3.5 If $M : \text{Term}_{\text{PCF}}$ and $Q : \text{Value}_{\text{PCF}} \rightarrow \mathcal{U}$, then $M \triangleright^k Q$ iff $M \Rightarrow^k (\lambda N. \Sigma v.N = v \land Q(v))$

Proof. We consider implication from left to right in the case of the fix-point. Assume $Y_{\sigma} M \triangleright^{k+1} Q$. By definition $\triangleright(M Y_{\sigma} M \triangleright^k Q)$. By induction hypothesis $\triangleright(M Y_{\sigma} M \Rightarrow^k (\lambda N. \Sigma v.N = v \land Q(v)))$. As $Y_{\sigma} M \rightarrow^1 M Y_{\sigma} M$ by definition $Y_{\sigma} M \Rightarrow^{k+1} (\lambda N. \Sigma v.N = v \land Q(v))$. The case from right to left follows from Lemma 3.4 together with the fact that $v \triangleright^0 v$.

The following is the standard statement for operational correspondence and follows directly from Lemma 3.5.

Corollary 3.6 $M \triangleright^k v \iff M \Rightarrow^k v$

4 Denotational semantics

We now define the denotational semantics of PCF. For this, we use the guarded recursive lifting monad on types, defined as the guarded recursive type$^5$

$$LA \overset{\text{def}}{=} \text{fix}.(A + \triangleright X).$$

Let $i : A + \triangleright LA \cong LA$ be the isomorphism, let $\theta : \triangleright LA \rightarrow LA$ be the right inclusion composed with $i$ and let $\eta : A \rightarrow LA$ (the unit of the monad) denote the left inclusion composed with $i$. Note that any element of $LA$ is either of the form $\eta(a)$ or $\theta(r)$.

$^5$ Since guarded recursive types are encoded using universes, $L$ is strictly an operation on $\mathcal{U}$. We will only apply $L$ to types that have codes in $\mathcal{U}$.
We can describe the universal property of $LA$ as follows. Define a $\triangleright$-algebra to be a type $B$ together with a map $\theta_B : \triangleright B \to B$. The lifting $LA$ as defined above is the free $\triangleright$-algebra on $A$. Given $f : A \to B$ with $B$ a $\triangleright$-algebra, the unique extension of $f$ to a homomorphism of $\triangleright$-algebras $\hat{f} : LA \to B$ is defined as

\[
\hat{f}(\eta(a)) \overset{\text{def}}{=} f(a) \\
\hat{f}(\theta(r)) \overset{\text{def}}{=} \theta_B(\text{next}(\hat{f}) \odot r)
\]

which can be formally expressed as a fixed point of a term of type $\triangleright (LA \to B) \to LA \to B$.

The intuition the reader should have for $L$ is that $LA$ is the type of computations possibly returning an element of $A$, recording the number of steps used in the computation. The unit $\eta$ gives an inclusion of values into computations, the composite $\delta = \theta \circ \text{next} : LA \to LA$ is an operation that adds one time step to a computation, and the bottom element $\bot = \text{fix}(\theta)$ is the diverging computation. In fact, any $\triangleright$-algebra has a bottom element and an operation $\delta$ as defined above, and homorphisms preserve this structure.

4.1 Interpretation

The interpretation function $[\cdot] : \text{Type} \to \mathcal{U}$ is defined by induction.

\[
[\text{nat}] \overset{\text{def}}{=} LN \\
[\tau \to \sigma] \overset{\text{def}}{=} [\tau] \to [\sigma]
\]

The denotation of every type is a $\triangleright$-algebra: the map $\theta_\sigma : \triangleright [\sigma] \to [\sigma]$ is defined by induction on $\sigma$ by

\[
\theta_{\sigma \to \tau} = \lambda f : \triangleright ([\sigma] \to [\tau]), \lambda x : [\sigma], \theta_\tau(f \odot \text{next}(x))
\]

Typing judgements $\Gamma \vdash M : \sigma$ are interpreted as usual as functions from $[\Gamma]$ to $[\sigma]$, where the interpretation of contexts is defined as $[x_1 : \sigma_1, \ldots, x_k : \sigma_k] \overset{\text{def}}{=} [\sigma_1] \times \cdots \times [\sigma_n]$. Figure 4 defines the interpretation of judgements. Below we often write $[M]$ rather than $[\Gamma \vdash M : \sigma]$. Natural numbers in PCF are computations that produce a value in zero step, so we interpret them by using $\eta$. In the case of $Y_\sigma$ we have by induction a map $[M](\gamma)$ of type $[\sigma] \to [\sigma]$. Morally, $[\Gamma \vdash Y_\sigma M](\gamma)$ should be the fixed point of $[M](\gamma)$ composed with $\delta$, ensuring that each unfolding of the fixed point is recorded as a step in the model, but to get the types correct, we have to apply the functorial action of $\triangleright$ to $[M](\gamma)$ and compose with $\theta$ instead of $\delta$. The intuition given above is captured in the following lemma.

**Lemma 4.1** Let $\Gamma \vdash M : \sigma \to \sigma$ then $[Y_\sigma M] = \delta_\sigma \circ [M(Y_\sigma M)]$

We now explain the interpretation of $\text{ifz} \ L M N$. Define first a semantic $\text{ifz} : [\sigma] \to [\sigma] \to N \to [\sigma]$ operation by

\[
\text{ifz} \ x \ y \ 0 \overset{\text{def}}{=} x \\
\text{ifz} \ x \ y \ (n + 1) \overset{\text{def}}{=} y
\]
\[ \{x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash x_i(\gamma) = \pi_i \gamma \} \]

\[
\begin{align*}
\Gamma \vdash n : \text{nat}(\gamma) &= \eta(n) \\
\Gamma \vdash X : \text{nat}(\gamma) &= (\text{fix}_\sigma)(\lambda x : \sigma. \sigma(\gamma :: (\text{next}(\| M \| (\gamma)) \otimes x))) \\
\Gamma \vdash \lambda x : \sigma. \lambda y : \tau. M(\gamma) &= \lambda x. \| M \| (\gamma, x) \\
\Gamma \vdash M N(\gamma) &= \| M \| (\gamma) \| N \| (\gamma) \\
\Gamma \vdash \text{succ} M(\gamma) &= L(\lambda x. x + 1)(\| M \| (\gamma)) \\
\Gamma \vdash \text{pred} M(\gamma) &= L(\lambda x. x - 1)(\| M \| (\gamma)) \\
\Gamma \vdash \text{ifz} L M N(\gamma) &= (\text{ifz}(\| M \| (\gamma), \| N \| (\gamma)))(\| L \| (\gamma))
\end{align*}
\]

Fig. 4. Interpretation of terms

The operation \( \text{ifz} : [\sigma] \to [\sigma] \to [\text{nat}] \to [\sigma] \) is defined by \( \text{ifz} \ x \ y \) being the extension of \( \text{ifz} \ x \ y \) to a homomorphism of \( \triangleright \)-algebras. As a direct consequence of this definition we get

**Lemma 4.2**  
(i) \[
\| \lambda x : \text{nat}. \text{ifz} \ x \ M \ N(\theta(\gamma)) = \theta(\text{next}(\| \lambda x : \text{nat}. \text{ifz} \ x \ M \ N(\gamma))) \oplus r)
\]

(ii) If \( \| L(\gamma) = \delta(\| L' \| (\gamma)) \), then \( \| \text{ifz} \ L \ M \ N(\gamma) = \delta(\| \text{ifz} \ L' \ M \ N(\gamma)) \)

### 4.2 Soundness

The soundness theorem states that if a program \( M \) evaluates to a value \( v \) in \( k \) steps then the interpretation of \( M \) is equal to the interpretation of \( v \) delayed \( k \) times by the semantic delay operation \( \delta \). Thus the soundness theorem captures not just extensional but also intensional behaviour of terms.

The theorem is proved using the small-step semantics. We first need a lemma for the single step reduction.

**Lemma 4.3** Let \( M \) be a closed term of type \( \tau \). If \( M \to_k N \) then \( \| M \| (\ast) = \delta^k \| N \| (\ast) \)

**Proof.** The proof goes by induction on \( M \to_k N \), and here we only consider two cases. The case of \( Y_\sigma \ M \to^1 M(Y_\sigma \ M) \) follows from Lemma 4.1. In the case of \( \text{ifz} M_1 N_1 N_2 \to^1 \text{ifz} M_2 N_1 N_2 \), the induction hypothesis gives \( \| M_1 \| = \delta \circ \| M_2 \| \), and now Lemma 4.2 applies proving the case. \( \square \)

We prove it now for \( \Rightarrow^k \).

**Lemma 4.4** Let \( M \) be a closed term of type \( \tau \), if \( M \Rightarrow^k N \) then \( \| M \| (\ast) = \delta^k \| N \| (\ast) \)

**Proof.** By induction on \( k \). The case \( k = 0 \) follows from Lemma 4.3. Assume \( k = k' + 1 \). By definition we have \( M \to^0_{\ast} M' \) and \( M' \to^1 M'' \) and \( \triangleright(M'' \Rightarrow^{k'} N) \). By repeated application of Lemma 4.3 we get \( \| M \| (\ast) = \| M' \| (\ast) \) and \( \| M' \| (\ast) = \delta(\| M'' \| (\ast)) \).

92
By induction hypothesis we get $\triangleright (\llbracket M'' \rrbracket (\ast) = \delta^k \llbracket N \rrbracket (\ast))$. By $\text{gDTT}$ rule $\text{TY} - \text{COM}$, this implies $\text{next}(\llbracket M'' \rrbracket (\ast)) = \text{next}(\delta^k \llbracket N \rrbracket (\ast))$ and since $\delta = \theta \circ \text{next}$, this implies $\delta \llbracket M'' \rrbracket (\ast) = \delta^k \llbracket N \rrbracket (\ast)$. By putting together the equations we get finally $\llbracket M \rrbracket (\ast) = \delta^k \llbracket v \rrbracket (\ast)$.

The Soundness theorem follows from the fact that the small-step semantics is equivalent to the big step, which is Corollary 3.6.

Theorem 4.5 (Soundness) Let $M$ be a closed term of type $\tau$, if $M \Downarrow^k v$ then $\llbracket M \rrbracket (\ast) = \delta^k \llbracket v \rrbracket (\ast)$

5 Computational Adequacy

In this section we prove that the denotational semantics is computationally adequate with respect to the operational semantics. At a high level, we proceed in the standard way, by constructing a logical relation $\mathcal{R}_\sigma$ between denotations $\llbracket \sigma \rrbracket$ and terms $\text{Term}_{PCF}$ and then proving that open terms and their denotation respect this relation (Lemma 5.6 below). We define our logical relation in guarded dependent type theory, so formally, it will be a map into the universe $U$ of types. Thus we work with a proof-relevant logical relation, similar to what was recently done in work of Benton et al. [3].

To formulate the definition of the logical relations and also to carry out the proof of the fundamental theorem of logical relations, we need some more sophisticated features of $\text{gDTT}$, which we now recall.

5.1 Guarded Dependent Type Theory

We recall some key features of $\text{gDTT}$; see [8] for more details.

As mentioned in Section 2, the later functor $\triangleright$ is an applicative functor. Guarded dependent type theory extends the later application $\otimes: \triangleright (A \rightarrow B) \rightarrow \triangleright A \rightarrow \triangleright B$ to the dependent case using a new notion of delayed substitution: if $\Gamma \vdash f : \Pi(x: A).B$ and $\Gamma \vdash t : \triangleright A$, then the term $f \otimes t$ has type $\triangleright [x \leftarrow t].B$, where $[x \leftarrow t]$ is a delayed substitution. Note that since $t$ has type $\triangleright A$, and not $A$, we can not substitute $t$ for $x$ in $B$. Intuitively, $t$ will eventually reduce to some value next $u$, and at that time the resulting type should be $\triangleright B[u/x]$. But when $t$ is an open term, we can not perform this reduction, and thus can not type this term. Hence we use the type mentioned earlier $\triangleright [x \leftarrow t].B$, in which $x$ is bound in $B$. Definitional equality rules allow us to simplify this type when $t$ has form next $u$, i.e.,

$$\triangleright [x \leftarrow \text{next } u].B \simeq \triangleright B[u/x]$$

as expected. Here we have just considered a single delayed substitution, in general, we may have sequences of delayed substitutions (such as $\triangleright [x \leftarrow t, y \leftarrow u].C$).

Delayed substitutions can also occur in terms, e.g., if $\Gamma, x: A \vdash t : B$ and $\Gamma \vdash u : \triangleright A$, then $\Gamma \vdash \text{next } [x \leftarrow u].t : \triangleright [x \leftarrow u].B$. Using this, one can express a generalisation of the rule (1)

$$\text{El}(\triangleright (\text{next } \xi.A)) \simeq \triangleright \xi. \text{El}(A) \quad (2)$$
where $\xi$ ranges over delayed substitutions. We recall the following rules from [8] which we will need in the development below. The notation $\xi[x \leftarrow t]$ means the extension of the delayed substitution $\xi$ with $[x \leftarrow t]$.

\[
\begin{align*}
\text{next } \xi[x \leftarrow \text{next } \xi.t],u &= \text{next } \xi.(u[t/x]) \quad (3) \\
\text{next } \xi[x \leftarrow t].x &= t \quad (4) \\
\text{next } \xi[x \leftarrow t].u &= \xi.u \quad \text{if } x \text{ not free in } u \quad (5) \\
\text{next } \xi[x \leftarrow t].(ux) &= (\text{next } \xi.u) \odot t \quad (6)
\end{align*}
\]

Of these, (3) and (4) can be considered $\beta$ and $\eta$ laws, and (5) is a weakening principle.

Recall from Section 2 that $\text{next}(f(t)) = \text{next } f \odot \text{next } t$. From the above rules we can derive a generalisation of this as follows

\[
\begin{align*}
\text{next } \xi.(ft) &= \text{next } \xi[x \leftarrow \text{next } \xi.t](fx) \\
&= (\text{next } \xi.f) \odot (\text{next } \xi.t)
\end{align*}
\]

5.2 Logical Relation

In this section we define a logical relation to prove the adequacy theorem. This relation is a function to $\mathcal{U}$.

We introduce the following notation:

**Notation 1** Let $\mathcal{R} : A \to B \to \mathcal{U}$ be a relation from $A$ to $B$, $t$ of type $\triangleright A$ and $u$ of type $\triangleright B$. Define $t \triangleright \mathcal{R} u \overset{\text{def}}{=} \triangleright [x \leftarrow t, y \leftarrow u].(x \mathcal{R} y)$

More precisely, we can define $t \triangleright \mathcal{R} u$ as a term of type $\mathcal{U}$ by defining it to be $\hat{\triangleright}(\text{next } [x \leftarrow t, y \leftarrow u].(x \mathcal{R} y))$, what we have defined above are the elements of this term. From this, one can prove that

\[
((\text{next } \xi.t) \triangleright \mathcal{R} (\text{next } \xi.u)) \simeq \triangleright \xi.(tRu) \quad (7)
\]

using (3) and (2).

**Lemma 5.1** The mapping $\lambda \mathcal{R}. \triangleright \mathcal{R} : (A \to B \to \mathcal{U}) \to \triangleright A \to \triangleright B \to \mathcal{U}$ is contractive, i.e., can be factored as $F \circ \text{next}$ for some $F : \triangleright (A \to B \to \mathcal{U}) \to \triangleright A \to \triangleright B \to \mathcal{U}$.

**Proof.** Define $F(S)(x,y) = \hat{\triangleright}(S \odot x \odot y)$. \hfill $\square$

**Definition 5.2** [Logical Relation] The logical relation $\mathcal{R}_\tau : [\tau] \times \text{Term}_{\text{PCF}} \to \mathcal{U}$ is inductively defined on types.

\[
\begin{align*}
\eta(v) \mathcal{R}_\text{nat} M &\overset{\text{def}}{=} M \Downarrow^0 v \\
\theta_\text{nat}(r) \mathcal{R}_\text{nat} M &\overset{\text{def}}{=} \Sigma M', \text{M}'' : \text{Term}_{\text{PCF}}. M \rightarrow^0 M' \text{ and } M' \rightarrow^1 M'' \text{ and } r \triangleright \mathcal{R}_\text{nat} \text{ next}(M'') \\
f \mathcal{R}_\tau \rightarrow_{\sigma} M &\overset{\text{def}}{=} \Pi : [\tau], N : \text{Term}_{\text{PCF}}. \alpha \mathcal{R}_\tau N \implies f(\alpha) \mathcal{R}_\sigma (MN)
\end{align*}
\]

The definition of $\mathcal{R}_\text{nat}$ is by guarded recursion using Lemma 5.1.

We now prove a series of lemmas needed for the proof of computational adequacy. The first states that the applicative functor action $\odot$ respects the logical relation.

94
Lemma 5.3 If \( f \triangleright R_{\tau \rightarrow \sigma} \) \( \text{next}(M) \) and \( r \triangleright R_{\tau} \text{next}(L) \) then \( (f \circ r) \triangleright R_{\sigma} \text{next}(ML) \).

Proof. The first hypothesis unfolds to
\[
\triangleright [g \leftarrow f].(g \ R_{\tau \rightarrow \sigma} M) \simeq \triangleright [g \leftarrow f].(\Pi(y : \mathbb{\sigma})(L : \text{Term}_{\text{ecr}}).y \ R_{\tau} L \rightarrow g(y) \ R_{\sigma} ML)
\]
By delayed application of this to \( r \), \( \text{next}(L) \) and the second hypothesis we get
\[
\triangleright [g \leftarrow f, y \leftarrow r].(g(y) \ R_{\sigma} ML), \text{which by (7) reduces to}
\]
\[
\text{next} [g \leftarrow f, y \leftarrow r].(g(y)) \triangleright R_{\sigma} \text{next} [g \leftarrow f, y \leftarrow r].(ML) \simeq (f \circ r) \triangleright R_{\sigma} \text{next}(ML).
\]
\( \square \)

The following lemma generalises the second case of \( R_{\text{nat}} \) to all types.

Lemma 5.4 Let \( \alpha \) of type \( \triangleright \mathbb{\sigma} \) and two terms \( N \) and \( M \), if \( (\alpha \triangleright R_{\sigma} \text{next}(N)) \) and \( M \rightarrow^1 N \) then \( \theta_{\sigma}(\alpha) \triangleright R_{\sigma} M \)

Proof. The proof is by induction on \( \sigma \). The case \( \sigma = \text{nat} \) is by definition of \( R_{\text{nat}} \).

For the induction step, suppose \( \alpha \) of type \( \triangleright \mathbb{\sigma} \), and \( M, N \) are closed terms such that \( \alpha \triangleright R_{\tau_1 \rightarrow \tau_2} \text{next}(N) \) and \( M \rightarrow^1 N \). We must show that if \( \beta : \triangleright \mathbb{\sigma} \), \( P : \text{Term}_{\text{ecr}} \) and \( \beta \ R_{\tau_1} P \) then \( (\theta_{\tau_1 \rightarrow \tau_2}(\alpha))(\beta) \triangleright R_{\tau_2} (MP) \).

So suppose \( \beta \ R_{\tau_1} P \), and thus also \( \triangleright (\beta) \triangleright R_{\tau_1} \text{next}(P) \). By applying Lemma 5.3 to this and \( \alpha \triangleright R_{\tau_1 \rightarrow \tau_2} \text{next}(N) \) we get
\[
\alpha \circ (\text{next}(\beta)) \triangleright R_{\tau_2} \text{next}(NP)
\]
Since \( M \rightarrow^1 N \) also \( MP \rightarrow^1 NP \), and thus, by the induction hypothesis for \( \tau_2 \), \( \theta_{\tau_2}(\alpha \circ (\text{next}(\beta))) \triangleright R_{\tau_2} MP \). Since by definition \( \theta_{\tau_1 \rightarrow \tau_2}(\alpha)(\beta) = \theta_{\tau_2}(\alpha \circ (\text{next}(\beta))) \), this proves the case. \( \square \)

Lemma 5.5 If \( M \rightarrow^0 N \) then \( \alpha \ R_{\sigma} M \) iff \( \alpha \ R_{\sigma} N \)

Proof. The proof is by induction on \( \sigma \). We show the left to right implication in the case \( \sigma = \text{nat} \). We proceed by case analysis on \( \alpha \) and show the case of \( \alpha = \theta_{\text{nat}}(r) \). From the assumption \( \alpha \ R_{\sigma} M \) we have that there exists \( M' \) and \( M'' \) such that \( M \rightarrow^0 M' \) and \( M' \rightarrow^1 M'' \) and \( \alpha \ R_{\text{nat}} \text{next}(M'') \). By determinism of the small-step semantics (Lemma 3.1) the reduction \( M \rightarrow^0 M' \) must factor as \( M \rightarrow N \rightarrow^0 M' \) and thus \( \alpha \ R_{\text{nat}} N \) as desired. \( \square \)

We can now finally prove the fundamental lemma, which can be thought of as a strengthened induction hypothesis for computational adequacy, generalised to open terms.

Lemma 5.6 (Fundamental Lemma) Let \( \Gamma \vdash t : \tau \), suppose \( \Gamma \equiv x_1 : \tau_1, \ldots , x_n : \tau_n \) and \( t : \tau_i, \alpha_i : \mathbb{\tau_i} \) and \( \alpha_i \ R_{\mathbb{\tau_i}} t_i \) for \( i \in \{1, \ldots , n\} \), then \( \triangleright \mathbb{\tau}(\alpha) \ R_{\tau} t(t/x) \)

Proof. The proof is by induction on the height of the typing judgement, and we just show the two most difficult cases.

We start off by the case \( \Gamma \vdash Y_{\sigma} M : \sigma \). The argument is by guarded recursion: we assume
\[
\triangleright (\mathbb{\tau}(\alpha) \ R_{\tau} Y_{\sigma}(Y_{\sigma} M)([t/x]))
\]
and prove $\llbracket Y_\sigma M \rrbracket (\alpha) R_\sigma (Y_\sigma M) ([t/x])$. By induction hypothesis we know $\llbracket M \rrbracket (\alpha) R_\sigma \rightarrow_\sigma M[t/x]$, hence we derive $\triangleright (\llbracket M \rrbracket (\alpha) R_\sigma \rightarrow_\sigma (M[t/x])N)$, i.e.,

$$\triangleright (\llbracket M \rrbracket (\alpha)(\llbracket Y_\sigma M \rrbracket (\alpha) R_\sigma \rightarrow_\sigma (M[t/x])N) \tag{9}$$

Applying (9) to (8) we get

$$\triangleright (\llbracket M \rrbracket (\alpha)(\llbracket Y_\sigma M \rrbracket (\alpha) R_\sigma \rightarrow_\sigma (M[t/x])N) \tag{9}$$

which is equal as types to

$$\triangleright (\llbracket M(Y_\sigma M) \rrbracket (\alpha) R_\sigma \rightarrow_\sigma (Y_\sigma M) ([t/x])\tag{10}$$

Thus, by Lemma 5.4

$$\theta_\sigma (\text{next}(\llbracket M(Y_\sigma M) \rrbracket (\alpha))) R_\sigma (Y_\sigma M) ([t/x])$$

and as $\delta_\sigma = \theta_\sigma \circ \text{next}$, by Lemma 4.1

$$\llbracket Y_\sigma M \rrbracket (\alpha) R_\sigma (Y_\sigma M) ([t/x])$$

as desired.

Now the case of $\Gamma \vdash \text{ifz } L M N : \sigma$. This case can be shown by showing that

$$\llbracket \lambda y. \text{ifz } y M N \rrbracket (\alpha) R_{\text{nat} \rightarrow_\sigma} (\lambda y. \text{ifz } y M N)[t/x]$$

and then applying this to the induction hypothesis $\llbracket L \rrbracket (\alpha) R_{\text{nat} \rightarrow_\sigma} L[t/x]$. The argument is by guarded recursion. Assume

$$\triangleright (\llbracket \lambda y. \text{ifz } y M N \rrbracket (\alpha) R_{\text{nat} \rightarrow_\sigma} (\lambda y. \text{ifz } y M N)[t/x]) \tag{10}$$

We must show that if $\beta : \llbracket \text{nat} \rrbracket$, $L : \text{Term}_{\text{pcr}}$ and $\beta R_{\text{nat} \rightarrow_\sigma} L$ then

$$\llbracket \lambda y. \text{ifz } y M N \rrbracket (\alpha)(\beta) R_\sigma ((\lambda y. \text{ifz } y M N)[t/x](L))$$

We proceed by case analysis on $\beta$. The interesting case is $\beta = \theta_{\text{nat}}(r)$. Here $r$ is of type $\triangleright \llbracket \text{nat} \rrbracket$ and $L : \text{Term}_{\text{pcr}}$. The hypothesis $\theta_{\text{nat}}(r) R_{\text{nat} \rightarrow_\sigma} L$ states that there exist $L', L'' : \text{Term}_{\text{pcr}}$ s.t. $L \rightarrow_\psi^0 L', L' \rightarrow_1 L''$ and

$$r \triangleright R_{\text{nat} \rightarrow_\sigma} \text{next}(L'') \tag{11}$$

Since (10) is equal to

$$(\text{next}(\llbracket \lambda y. \text{ifz } y M N \rrbracket (\alpha))) \triangleright R_{\text{nat} \rightarrow_\sigma} \text{next}((\lambda y. \text{ifz } y M N)[t/x])$$

We can apply Lemma 5.3 to that and (11) to get (using Lemma 5.5)

$$(\text{next}(\llbracket \lambda y. \text{ifz } y M N \rrbracket (\alpha)) \oplus r) \triangleright R_{\sigma} \text{next}(\text{ifz } L'' M[t/x] N[t/x])$$

96
By Lemma 5.4 with $L' \rightarrow^1 L''$ this implies

$$\theta_\sigma(\text{next}(\|\lambda y. \text{ifz } y M N\|)(\alpha)) \odot r) \mathcal{R}_\sigma (\text{ifz } L' M[t/x] N[t/x])$$

and by Lemma 4.2 along with repeated application of Lemma 5.5 this implies

$$\|\lambda y. \text{ifz } y M N\|((\alpha))(\beta) \mathcal{R}_\sigma (\lambda y. \text{ifz } y M N)[t/x](L)$$

thus getting what we wanted.

We have now all the pieces in place to prove adequacy.

**Theorem 5.7 (Computational Adequacy)** If $M$ is a closed term of type $\text{nat}$ then $M \Downarrow^k v$ iff $\|M\|(*) = \delta^k[v]$.

**Proof.** The left to right implication is soundness (Theorem 4.5). For the right to left implication note first that the Fundamental Lemma (Lemma 5.6) implies $\delta^k[v] \mathcal{R}_{\text{nat}} M$. An easy induction on $k$ then proves that $M \Downarrow^k v$.

**Remark 5.8** In the topos of trees model $\|\text{nat}\|(n) \cong \{1, \ldots, n\} \times \mathbb{N} + \{\bot\}$. Values are modelled as elements of the form $(1, k)$ and $\delta$ is defined as $\delta(j, k) = (j + 1, k)$ if $j < n$ and $\delta(n, k) = \bot$. Thus, if a term $M$ diverges, then $\|M\|(*) = \delta^k[v]$ holds at stage $n$ whenever $k > n$ explaining the need for $M \Downarrow^k v$ to be true also at stage $n$ when $k > n$.

6 The external viewpoint

The adequacy theorem is a statement formulated entirely in $\mathsf{gDTT}$, relating two notions of semantics also formulated entirely in $\mathsf{gDTT}$. While we believe that $\mathsf{gDTT}$ is a natural setting to do semantics in, and that the result therefore is interesting in its own right, it is still natural to ask what we proved in the “real world”. One way of formulating this question more precisely is to use the interpretation of $\mathsf{gDTT}$ in the topos of trees (henceforth denoted by $\mathcal{L}$). For example, the sets of PCF types, terms and values are inductively defined types, which are interpreted as constant presheaves over the corresponding sets of types, terms and values. Types of PCF as understood in set theory, thus correspond bijectively to global element of $\mathcal{L}_{\text{ref}}$, which by composing with the interpretation of PCF defined in $\mathsf{gDTT}$ gives rise to an object in the topos of trees. Likewise, a PCF term gives rise to a morphism in the topos of trees. Thus, essentially by composing the interpretation of PCF given above with the interpretation of $\mathsf{gDTT}$ in the topos of trees, we get an interpretation of PCF into the topos of trees, which we will denote by $\|\cdot\|_{\text{ext}}$.

We denote by $M \Downarrow^k_{\text{ext}} v$ the usual external formulation of the big-step semantics for PCF obtained from Figure 2 by removing $\Delta$s and replacing dependent sums by existential quantifiers (see e.g. [10]).

**Lemma 6.1** The type $\langle M \Downarrow^k Q \rangle$ is globally inhabited iff there exists a value $v$ such that $M \Downarrow^k_{\text{ext}} v$ and $\langle Q(v) \rangle$ is globally inhabited.

The lemma can be proved by induction over first $k$ then $M$. 97
As a special case, Theorem 5.7 states that $\langle M \, \Downarrow \, v \rangle$ is inhabited by a global element iff $\llbracket [M] (\ast) = \delta^k [v] \rrbracket$ is inhabited by a global element. Since the topos of trees is a model of extensional type theory, the latter holds precisely when $\llbracket [M]_{\text{ext}} = \delta^k [v] \rrbracket$.

**Theorem 6.2 (Computational Adequacy, externally)** If $\vdash M : \sigma$ with $\sigma$ a ground type, then $M \Downarrow^k v$ iff $\llbracket [M]_{\text{ext}} (\ast) = \delta^k [v] \rrbracket$.

7 Discussion and Future Work

In earlier work, it has been shown how guarded type theory can be used to give abstract accounts of operationally-based step-indexed models [6,15]. There the operational semantics of the programming language under consideration is also defined inside guarded type theory, but there are no explicit counting of steps (indeed, part of the point is to avoid the steps). Instead, the operational semantics is defined by the transitive closure of a single-step relation — and, importantly, the transitive closure is defined by a fixed point using guarded recursion. Thus some readers might be surprised why we use a step-counting operational semantics here. The reason is simply that we want to show, in the type theory, that the denotational semantics is adequate with respect to an operational semantics and since the denotational semantics is intensional and steps thus matter, we also need to count steps in the operational semantics to formulate adequacy.

In previous work [6] we have studied the internal topos logic of the topos of trees model of guarded recursion and used this for reasoning about advanced programming languages. In this paper, we could have likewise chosen to reason in topos logic rather than type theory. We believe that the proofs of soundness and computational adequacy would have gone through also in this setting, but the interaction between the $\nabla$ type modality and the existential quantifiers in the topos of trees, makes this an unnatural choice. For example, one can prove the statement $\exists k. \exists v. \, Y_{\text{nat}} \, (\lambda x. x) \Downarrow^k v$ in the internal logic using guarded recursion as follows: assume $\nabla (\exists k. \exists v. \, Y_{\text{nat}} \, (\lambda x. x) \Downarrow^k v)$. Because $\text{nat}$ is total and inhabited we can pull out the existentials by Theorem 2.7.4 in [6] and derive $\exists k. \exists v. (Y_{\text{nat}} \, (\lambda x. x) \Downarrow^k v)$ which implies $\exists k. \exists v. Y_{\text{nat}} \, (\lambda x. x) \Downarrow^k v$. The corresponding statement in type theory: $\sum k, v. Y_{\text{nat}} \, (\lambda x. x) \Downarrow^k v$ is not derivable as can be proved using the topos of trees. Intuitively the difference is the constructiveness of the dependent sum, which allows us to extract the witnesses $k$ and $n$.

In future work, we would like to explore models of FPC (i.e., PCF extended with recursive types) and also investigate how to define a more extensional model by quotienting the present intensional model. The latter would be related to Escardo’s results in [10].

**Acknowledgement**

We thank Aleš Bizjak for fruitful discussions.
References


Abstract
The notion of trace in a monoidal category has been introduced to give a categorical account of a situation occurring in very different settings: linear algebra, topology, knot theory, proof theory... with the trace operation understood as a feedback operation.

Partially traced categories were later introduced to account for cases where the trace is not always defined, and it was shown that partially traced category can always be seen as a subcategory of a totally traced one. We give a new proof of this representation theorem, using a construction that is different from the original one. However, since they both satisfy the same universal property, the two constructions yield categories which embed into each other.

Keywords: monoidal category, trace, feedback, representation theorem.

Introduction
Traced monoidal categories were introduced by A. Joyal, R. Street and D. Verity [9] as a common categorical axiomatization of a structure that occurs in very different settings such as linear algebra, topology, knot theory, proof theory... In particular, traced monoidal categories constitute the basis of the categorical approach [1,5] to J.-Y. Girard’s geometry of interaction program [3].

The basic idea is that a trace is an operation associating to any \( f : A \otimes U \rightarrow B \otimes U \) in a monoidal category, a new morphism \( \text{Tr}^U(f) : A \rightarrow B \), this operation being understood as a feedback along \( U \), which is acknowledged in the graphical language for these categories [9] by depicting \( \text{Tr}^U(f) \) as

This operation has to satisfy a number of axioms that capture formally what is expected of such a notion of feedback.

©2015 Published by Elsevier Science B. V.
More recently, E. Haghverdi and P. J. Scott [6] introduced the notion of partial trace, accounting for the fact that the trace operation can be only partially defined. This is a situation that occurs very naturally in practice: think of the trace in infinite-dimensional Hilbert spaces 1 or feedback loops in synchronous circuits, for instance. Later on, O. Malherbe, P. J. Scott and P. Selinger [11] showed a representation theorem for partial traces, relating them to total traces: any partially traced category embeds in a totally traced one, with an embedding reflecting the partial trace. Their construction is based on P. Freyd’s paracategories [7] and a partial version of the Int(·) construction [9]. It enjoys a universal property factoring any functor reflecting the partial trace structure.

In this article, we will give a new proof of this result, via a different construction based on tools used in the categorical approach to equivalence of automata [2]. Our proof is more straightforward and does not rely on a delicate argumentation about partially defined operations. As a consequence, the formulation of the universal property we obtain is also more direct as it does not involve compact closed categories as an intermediate step. However, the two constructions satisfy in the end the same universal property and must therefore be related by mutual embeddings preserving their trace structure.

Outline of the article

In section 1 we fix the notations and vocabulary used in the rest of the article; we recall the definition of a partially traced category, and the associated notion of traced functor.

We then introduce in section 2 the key ingredient of our construction: the dialect construction, that allows morphisms to have private interfaces. With this construction comes a hiding operation which sets the basis for total extensions of partial traces.

A congruence is then defined in section 3 and its interplay with the monoidal and traced structures is explored. We will show that quotienting the dialect category by this equivalence turns the hiding operation into a total trace (section 3.1).

In section 3.2 and section 4.1, we will show that the original partially traced category embeds in this quotiented category, via an embedding reflecting the partial trace. Finally, we show that our construction enjoys the expected universal property in section 4.2.

1 Partially traced categories

We begin by setting notations and recalling some background definitions.

Notation 1.1 We write the composition of morphisms in a category in the usual order, omitting the \( \circ \) symbol: if we have \( f : A \to B \) and \( g : B \to C \) then \( gf : A \to C \). We may also omit the parenthesis when applying functors to objects and morphisms: if we have \( F : C \to D \) and \( f : A \to B \) in \( C \), then \( Ff : FA \to FB \).

1 Although this case actually fails to satisfy axiom (v) of definition 1.5 and would need a relaxing of the framework (and a non-trivial, if possible, adaptation of our proof of the representation theorem) to be considered as a partial trace in the categorical sense.
When manipulating two categories with similar operations, we will use subscripts to indicate in which category an operation occurs when not clear from the context.

**Definition 1.2** A **monoidal category** is a category $\mathcal{C}$ together with a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and a distinguished object $1$ satisfying:

- The functor $\otimes$ is associative: for any objects $A, B, C$ of $\mathcal{C}$, $A \otimes (B \otimes C) = (A \otimes B) \otimes C$, which we then write $A \otimes B \otimes C$, and the same holds for morphisms.
- The object $1$ is neutral: for any object $A$, $A \otimes 1 = 1 \otimes A = A$, and $\text{Id}_1 \otimes f = f \otimes \text{Id}_1 = f$ for any $f$.

It is called a **symmetric** monoidal category if it enjoys a natural family of isomorphisms $\sigma_{A,B} : A \otimes B \to B \otimes A$ such that $\sigma_{B,A}^{-1} = \sigma_{A,B}$ and $\sigma_{A \otimes B,C} = (\sigma_{A,C} \otimes \text{Id}_B)(\text{Id}_A \otimes \sigma_{B,C})$.

**Remark 1.3** We consider, as in the work of O. Malherbe, P. J. Scott and P. Selinger [11], the **strict** variant of symmetric monoidal categories to work with lighter notations. This is relatively harmless since any symmetric monoidal category is equivalent to a strict one [10] and hence our results can be extended (with a bit of extra work) in a non-strict setting building upon this this equivalence.

As we will manipulate operations that are only partially defined, we will use the **Kleene equality** notation to describe situations where one of the two equated expressions might be undefined.

**Notation 1.4** When handling equality between potentially undefined expressions, we will use the following notations:

- $E \xRightarrow{\text{def}} E'$ means that if $E$ is defined, then $E'$ is and in that case $E = E'$.
- $E \xRightarrow{\text{def}} E'$ means that if $E'$ is defined, then $E$ is and in that case $E = E'$.
- $E \xRightarrow{\text{def}} E'$ means that $E$ is defined if and only if $E'$ is and in that case $E = E'$.

When both sides are always defined, we simply write $E = E'$.

Let us now introduce the notion of partial trace in a symmetric monoidal category. As mentioned in the introduction, an intuitive way to understand it is to think of a feedback operator. Most of the axioms are very natural from that perspective, especially if written in the associated graphical language [9], see fig. 1.

**Definition 1.5** [11] A **partial trace** in a symmetric monoidal category is a partially defined operation on morphisms (parametrized by an object $U$ and two objects $A, B$ which we leave implicit) $\text{Tr}^U[\cdot]$ that inputs a morphism $f : A \otimes U \to B \otimes U$ and outputs (when defined) $\text{Tr}^U[f] : A \to B$.

It must satisfy the following axioms:

(i) **Superposing:** for all $f : A \otimes U \to B \otimes U$ and $g : C \to D$, we have

$$\text{Tr}^U[f] \otimes g = \text{Tr}^U[f \otimes g]$$

(ii) **Tightening:** for all $f : A \otimes U \to B \otimes U$, $g : A' \to A$ and $h : B \to B'$, we have

$$h \text{Tr}^U[f] g = \text{Tr}^U[(h \otimes \text{Id}_U)f(g \otimes \text{Id}_U)]$$
Figure 1. The axioms of partial traces, graphically.
(iii) Sliding: for all \( f : A \otimes U \to B \otimes U \) and \( g : U \to U' \), we have
\[
\text{Tr}^U[f(\text{Id}_A \otimes g)] = \text{Tr}^{U'}[(\text{Id}_B \otimes g)f]
\]
(iv) Vanishing: for all \( f : A \otimes 1 \to B \otimes 1 \), we have
\[
\text{Tr}^1[f] = f
\]
(v) Associativity: for all \( f : A \otimes U \otimes V \to B \otimes U \otimes V \), if \( \text{Tr}^V[f] \) is defined we have
\[
\text{Tr}^{U \otimes V}[f] = \text{Tr}^U[\text{Tr}^V[f]]
\]
(vi) Yanking: for any object \( A \), we have
\[
\text{Tr}^A[\sigma_{A,A}] = \text{Id}_A
\]

A symmetric monoidal category equipped with a partial trace will be called a partially traced category. In case \( \text{Tr}^U[\cdot] \) is always defined, we call it a totally traced category.

**Definition 1.6** Let us also set a weaker variant of axiom (iii):

(iii)' Weak sliding: for all \( f : A \otimes U' \to B \otimes U \) and isomorphism \( g : U \to U' \), we have
\[
\text{Tr}^U[f(\text{Id}_A \otimes g)] = \text{Tr}^{U'}[(\text{Id}_B \otimes g)f]
\]

With these notions of categories come notions of functors preserving their structures. The concept of traced embeddings allows to formulate precisely how a partially traced category embeds into a total one.

**Definition 1.7** A symmetric monoidal functor \( F : C \to D \) between symmetric monoidal categories is a functor together with a natural family of isomorphisms \( m_{A,B} : FA \otimes_D FB \to F(A \otimes_C B) \) and an isomorphism \( m_1 : 1_D \to F(1_C) \) satisfying some coherence axioms expressing compatibility with the symmetric monoidal structure [10].

If moreover \( C \) and \( D \) are partially traced, it will be called a traced functor if it is compatible with the partial trace: \( \text{Tr}^D_F[Ff] = Fg \) if and only if \( \text{Tr}^C[Gf] = g \).

A functor will be called an embedding if it is injective both on objects and morphisms.

**Remark 1.8** The traced property means that the functor \( F \) reflects the trace: in \( D \), the trace of \( Ff \) is defined and yields a morphism in the image of \( F \) exactly when the trace of \( f \) is defined. In particular, when \( D \) is a totally traced category, \( \text{Tr}^D_F[Ff] \) is always defined and is equal to some \( Fg \) exactly when \( \text{Tr}^U[f] \) is defined.

---

\( ^2 \) We actually define here strong (as opposed to lax) monoidal functors. As we will only consider strong functors in this work, we will not need to specify this later on.
2 The dialect construction

We introduce now the notion of the dialect category built out of a (strict) monoidal category. The basic idea is the following: a morphism from $A$ to $B$ in the category $\mathbf{D}(\mathcal{C})$ is a morphism $f : A \otimes U \rightarrow B \otimes U$ in the category $\mathcal{C}$, the object $U$ being thought of as a state space which is private to $f$. The composition in $\mathbf{D}(\mathcal{C})$ is given following this intuition: when composing two morphisms with a state space, neither of them has access to information about the state space of the other. This appears more clearly when looking at the graphical representation of composition in fig. 2.

This construction has been used in categorical approaches to automata equivalence [2] and — while it has not been emphasized as a generic construction in these cases — when dealing with the additive connectives of linear logic both in the context of proofnets [8], where it is related to the notion of slice, and in geometry of interaction [4].

**Definition 2.1** Given a symmetric monoidal category $\mathcal{C}$ we define its dialect category $\mathbf{D}(\mathcal{C})$ as follows:

- The objects of $\mathbf{D}(\mathcal{C})$ are the objects of $\mathcal{C}$.
- The morphisms of $\mathbf{D}(\mathcal{C})$ are of the form $(f, U) : A \rightarrow B$ where $U$ is an object of $\mathcal{C}$ and $f : A \otimes U \rightarrow B \otimes U$ is a morphism of $\mathcal{C}$.
- The identity on $A$ is the morphism $(\text{Id}_A, 1)$.
- Composition of $(f, U) : A \rightarrow B$ and $(g, V) : B \rightarrow C$ is defined as

$$ (g, V)(f, U) = \left( (\text{Id}_C \otimes \sigma_{V,U})(g \otimes \text{Id}_U)(\text{Id}_B \otimes \sigma_{U,V})(f \otimes \text{Id}_V), U \otimes V \right) $$

**Remark 2.2** The fact that $\mathbf{D}(\mathcal{C})$ is a category without needing further modification is a consequence of the strictness of $\mathcal{C}$, which we assume in this article. Indeed, if for instance we do not have $(U \otimes V) \otimes W = U \otimes (V \otimes W)$ but only an isomorphism between the two, then we get the associativity of composition $f(gh) = (fg)h$ only up to isomorphism.

![Figure 2. Composition in $\mathbf{D}(\mathcal{C})$. The exerted eye might notice the similarity with the composition in the case of the $\text{Int}(\cdot)$ construction [9].](image)

Now we can define an operation in $\mathbf{D}(\mathcal{C})$ that will, after proper quotienting, become a total trace: hiding. Given a morphism in $\mathbf{D}(\mathcal{C})$, one can always decide to privatize part of its interface, moving it to the state space.

**Definition 2.3** If $(f, V) : A \otimes U \rightarrow B \otimes U$ is a morphism in $\mathbf{D}(\mathcal{C})$, then we define

$$ \mathbb{H}^U[f, V] = (f, U \otimes V) : A \rightarrow B $$

We cannot say much on this operation for the moment, since we do not have yet any monoidal structure for which it could be candidate to be a trace. This is the topic of the next section.
3 Congruences and partial traces

In this section we will use the concept of quotient category that allows us to identify morphisms of $\mathbf{D}(\mathcal{C})$ to get a monoidal structure for which the hiding operation is a trace.

A congruence is the general notion of equivalence relation that can be used in the quotient category construction: it must be compatible with composition.

**Definition 3.1** If $\mathcal{C}$ is a category, an equivalence relation on morphisms (with same domain and codomain) $\ast$ is said to be a *congruence* if $f \ast f' \implies fg \ast f'g$ and $hf \ast hf'$ for all $g, h$.

**Definition 3.2** If $\mathcal{C}$ is a category and $\ast$ a congruence, the *quotient category* $\mathcal{C}/\ast$ is defined as

- Objects of $\mathcal{C}/\ast$ are the objects of $\mathcal{C}$.
- Morphisms of $\mathcal{C}/\ast$ are the morphisms of $\mathcal{C}$ modulo $\ast$.
- Composition and identities are induced by $\mathcal{C}$ modulo $\ast$.

We then set up some basic relations capturing what we would like to be equalities in the quotient of $\mathbf{D}(\mathcal{C})$ we are going to consider: morphisms with a state space should be considered equal up to isomorphism and tracing (when defined) on their state part. We will then quotient by the induced equivalence relation.

**Definition 3.3** Let $\mathcal{C}$ be a partially traced category, $(f, U)$ and $(g, V)$ morphisms in $\mathbf{D}(\mathcal{C})$. We define the following relations:

- If there is an isomorphism $\varphi : U \to V$ in $\mathcal{C}$ such that $(\text{Id}_B \otimes \varphi)f(\text{Id}_A \otimes \varphi^{-1}) = g$ then $(f, U) \sim (g, V)$.
- If $U = V \otimes U''$ and $\text{Tr}^{U''}[f] = g$ then $(g, V) \prec (f, U)$ and $(f, U) \succ (g, V)$.
- The relation $\approx$ is the equivalence relation generated by $\sim, \prec$ and $\succ$.

**Remark 3.4** Note that $\prec$ and $\succ$ are reflexive: $(f, U) = (f, U \otimes 1) \succ (f, U)$ by the vanishing axiom ((iv) of definition 1.5).

Now the first thing to check is that this $\approx$ equivalence relation is indeed a congruence, so that we can look at what happens when quotienting by it. We do this by showing that its basic components are all compatible with composition.

**Lemma 3.5** If $\ast$ is $\sim, \prec$ or $\succ$, then $(f, U) \ast (f', U')$ implies $(f, U)(g, V) \ast (f', U')(g, V)$ and $(h, W)(f, U) \ast (h, W)(f', U')$ for all $(g, V)$ and $(h, W)$.

**Proof** The case of $\sim$ is straightforward, therefore let us have look at $\prec$ ($\succ$ being similar): if $(f, U) \prec (f', U')$, then $U' = U \otimes U''$ and $\text{Tr}^{U''}[f'] = f$. Applying the superposing and tightening axioms, we get the required relations. \hfill $\Box$

**Corollary 3.6** The above defined $\approx$ is a congruence.

**Proof** If $(f, U) \approx (f', U')$, we have a chain of relations from $(f, U)$ to $(f', U')$.

Applying repeatedly lemma 3.5 we can obtain an identical chain for both $(f, U)(g, V) \approx (f', U')(g, V)$ and $(h, W)(f, U) \approx (h, W)(f', U')$. \hfill $\Box$
We finally set the category that will be shown to be totally traced and in which \( C \) embeds: the dialect category quotiented by \( \simeq \).

**Definition 3.7** If \( C \) is a partially traced category, we define \( T(C) = D(C)/\simeq \) and keep writing \( H[\cdot] \) the induced operation on morphisms of \( T(C) \).

The operation \( H[\cdot] \) can indeed be transported to \( T(C) \) because it is compatible with \( \approx \) as \( (f, U) \approx (g, V) \) implies \( H^W[f, U] \approx H^W[g, V] \) for any suitable \( W \).

### 3.1 From partial to total

We will now show that \( T(C) \) is a totally traced category. Let us first identify its monoidal structure, induced by that of \( C \).

**Definition 3.8** If \( C \) is a partially traced category, we define the following operation \( \otimes \) on morphisms of \( D(C) \): if we have \( (f, U) : A \to B \) and \( (g, V) : C \to D \)

\[
(f, U) \otimes (g, V) = ((\text{Id}_B \otimes \sigma_{D,U} \otimes \text{Id}_V)(f \otimes g)(\text{Id}_A \otimes \sigma_{U,C} \otimes \text{Id}_V), U \otimes V)
\]

Moreover, we set \( \tau_{A,B} = (\sigma_{A,B}, 1) \).

![Figure 3. Monoidal structure in \( D(C) \).](image)

**Lemma 3.9** If \( * \) is either \( \sim \), \( \prec \) or \( \succ \), then \( (f, U) * (f', U') \) implies that both \( ((f, U) \otimes (g, V)) * ((f', U') \otimes (g, V)) \) and \( ((g, V) \otimes (f, U)) * ((g, V) \otimes (f', U')) \) for any \( (g, V) \).

**Proof** This is similar to the proof of lemma 3.5, using the superposing axiom instead of tightening in the cases of \( \prec \) and \( \succ \). \( \square \)

The lemma above ensures that \( \otimes \) induces a well-defined operation on \( T(C) \). Note that before quotienting by \( \simeq \), this operation is not a bifunctor since the compositions

\[
(\text{Id} \otimes (g, V))(f, U) \otimes \text{Id}) = (\cdots, U \otimes V)
\]

\[
(\text{Id} \otimes (g, V))(f, U) \otimes \text{Id}) = (\cdots, V \otimes U)
\]

cannot be equal (but are related by \( \sim \)).

**Proposition 3.10** The category \( T(C) \) is symmetric monoidal.

**Proof** The monoidal structure is given by the above defined \( \otimes \) operation (which is a bifunctor once we quotient by \( \simeq \)) and the object \( 1 \). The symmetries are the \( \tau_{A,B} \). \( \square \)

We finally turn to proving that the hiding operation on \( T(C) \) is actually a trace which encompasses the partial trace of \( C \) given how we defined the relation \( \simeq \).
We notice first that once we quotient by \( \approx \), \( H[:\cdot:] \) immediately enjoys some of the properties of a trace.

**Proposition 3.11 (feedback properties)** The operation \( H[:\cdot:] \) in \( T(C) \) satisfies all axioms of definition 1.5 except (iii) (sliding) and (vi) (yanking). It also satisfies (iii’) (weak sliding).

Now let us mention the following lemma saying that it is enough to have sliding on isomorphisms in presence of the other axioms of the trace to get full sliding.

**Lemma 3.12 ([9, lemma 2.1])** In presence of the other axioms of the trace, the weak sliding axiom (iii’) implies its full variant.

Putting all this together, we can finally show that \( H[:\cdot:] \) is indeed a trace in \( T(C) \).

**Corollary 3.13** The category \( T(C) \) is totally traced with \( H[:\cdot:] \) as a trace.

**Proof** Proposition 3.11 lists the properties of a trace \( H[:\cdot:] \) satisfies: all of them but (iii) and (vi). It also satisfies (iii’). As \( Tr[:\cdot:] \) satisfies axiom (vi), we have in addition that \( H^A[\tau_{A,A}] = (\sigma_{A,A}, A) \succ (\text{Id}_A, 1) \), that is to say \( H[:\cdot:] \) also satisfies (vi) in \( T(C) \).

By lemma 3.12, \( H[:\cdot:] \) satisfies the full axiom (iii) in \( T(C) \). Therefore \( H[:\cdot:] \) is a total trace in \( T(C) \).

3.2 Embedding in \( T(C) \)

We now turn to the question of embedding \( C \) into \( T(C) \), the basic idea being that \( f \) in a partially traced category \( C \) will be represented as \( (f, 1) \) in \( T(C) \). The main difficulty here is to make sure that the equivalence relation we introduced does not happen to equate two morphisms that were different in \( C \). In order to prove this, we have a series of lemmas that allow us to progressively rewrite chains of equivalences into assertions that some trace is defined when morphisms of the form \( (f, 1) \) are involved.

**Lemma 3.14** If \( (f, 1) \sim (g, U) \), \( (f, 1) \succ (g, U) \) or \( (f, 1) \prec (g, U) \) then \( Tr^U[g] = f \) (i.e. \( Tr^U[g] \) is defined and equal to \( f \)).

**Proof** If \( (f, 1) \succ (g, U) \), it means that \( 1 = U \otimes V \) with \( Tr^V[f] \) defined, equal to \( g \). By associativity and vanishing, \( Tr^U[g] \equiv Tr^U[Tr^V[f]] = Tr^U\otimes V[f] \equiv Tr^1[f] = f \). Therefore \( Tr^U[g] \) is defined and equal to \( f \). The case of \( (f, 1) \prec (g, U) \) is similar.

If \( (f, 1) \sim (g, U) \), we get \( Tr^U[g] \equiv Tr^1[f] = f \) by sliding and vanishing.

**Lemma 3.15** If \( Tr^U[g] = f \) and \( (g, U) \sim (h, V) \), then \( Tr^V[h] = f \).

**Proof** By the sliding axiom, \( Tr^V[h] \equiv Tr^U[g] = f \).

**Lemma 3.16** If \( Tr^U[g] = f \) and \( (g, U) \prec (h, V) \), then \( Tr^V[h] = f \).

**Proof** We have \( V = U \otimes V' \) and \( Tr^V'[h] = g \). By the associativity axiom, we get that \( Tr^V[h] \equiv Tr^U \otimes V'[h] \equiv Tr^U[Tr^V'[h]] \equiv Tr^U[g] = f \).

**Lemma 3.17** If \( Tr^V[g] = f \) and \( (g, V) \succ (h, U) \), then \( Tr^U[h] = f \).
We have \( V = U \otimes V' \) and \( \text{Tr}^{V'}[g] = h \). By the associativity axiom, we get that \( \text{Tr}^{U}[h] = \text{Tr}^{U}[\text{Tr}^{V'}[g]] = \text{Tr}^{U \otimes V'}[g] = \text{Tr}^{V'}[g] = f \).

Combining all this, we get that equivalence to a morphism of the form \((f, 1)\) implies the definiteness of the trace. As a consequence, we will be able to show that we have indeed an embedding of \( \mathcal{C} \) into \( \mathcal{T}(\mathcal{C}) \) that reflects the partial trace.

**Proposition 3.18** If \((f, 1) \approx (g, U)\), then \( \text{Tr}^{U}[g] = f \).

In particular, if \((f, 1) \approx (g, 1)\) then \( f = g \).

**Proof** Given two equivalent morphisms \((f, 1) \approx (g, U)\), we can repeatedly apply lemmas 3.14–3.17 to transform (starting from \( \text{Tr}^{1}[f] = f \)) the chain of relations between them into the required \( \text{Tr}^{U}[g] = f \).

In the particular case where \((f, 1) \approx (g, 1)\) we get \( g = \text{Tr}^{1}[g] = f \).

**Corollary 3.19 (embedding)** The functor \( E_{\mathcal{C}} : \mathcal{C} \to \mathcal{T}(\mathcal{C}) \) defined as

\[
E_{\mathcal{C}}(A) = A \quad \text{(on objects)} \\
E_{\mathcal{C}}(f) = (f, 1) \quad \text{(on morphisms)}
\]

is a symmetric monoidal embedding.

**Proof** As it acts as the identity on objects, \( E_{\mathcal{C}} \) is what is called a strict symmetric monoidal embedding: the \( m \) isomorphisms are actually identities and moreover \( E_{\mathcal{C}}(f \otimes g) = E_{\mathcal{C}}(f) \otimes E_{\mathcal{C}}(g) \) for any morphisms \( f, g \). The fact that it is an embedding follows from the second part of proposition 3.18.

## 4 Representation theorem

We finally come to formulate and prove the main result of this article, the representation theorem for partially traced categories.

It is split into two parts: first we show that the embedding defined in the previous section is traced, which means that any partially traced category embeds (reflecting the trace) into a totally traced one by the \( \mathcal{T}(\cdot) \) construction; second, we show that this construction enjoys a universal property ensuring that any traced functor from a partially traced category to a total one factors as the embedding followed by a traced functor.

### 4.1 Reflecting the trace

A consequence of **propoosition 3.18** is the following lemma, from which the traced nature of \( E_{\mathcal{C}} \) follows: the trace of a morphism in \( \mathcal{C} \) is defined exactly when its hiding in \( \mathcal{T}(\mathcal{C}) \) is equivalent to some \((g, 1)\), that is to say exactly when it is in the image of the embedding of \( \mathcal{C} \) into \( \mathcal{T}(\mathcal{C}) \).

**Lemma 4.1** Given a \( f : A \otimes U \to B \otimes U \) in a partially traced category \( \mathcal{C} \), we have that \( \text{Tr}^{U}[f] \) is defined if and only if there is a morphism of the form \((g, 1)\) in \( \mathcal{T}(\mathcal{C}) \) such that \((f, U) \approx (g, 1)\).
The fact that $\text{Tr}^U[f]$ is defined implies $(f, U) \succ (\text{Tr}^U[f], 1)$ by definition. Conversely, if $(f, U) \approx (g, 1)$ the first part of proposition 3.18 tells us that $\text{Tr}^U[f]$ is defined.

**Theorem 4.2** The functor $E_C : C \to T(C)$ is a traced embedding.

**Proof** We already know by corollary 3.19 that it is a (strict) monoidal embedding, therefore we only have to check its traced nature, which is an immediate consequence of lemma 4.1.

4.2 Universal property

Now we can consider equivalent morphisms $(f, U) \approx (g, V)$ and show that the traces of their images through any traced functor to a totally traced category must be equal. This will allow us to properly define the factorization through $T(C)$ in the universal property.

**Lemma 4.3** If we have $f : A \otimes U \to B \otimes U$ and $g : A \otimes V \to B \otimes V$ in a partially traced category $C$ such that $(f, U) \approx (g, V)$ and a traced functor $F : C \to D$ with $D$ totally traced, then $\text{Tr}^F(U)(Ff) = \text{Tr}^F(V)(Fg)$.

**Proof** It is enough to show that $(f, U) \sim (g, V)$, $(f, U) \prec (g, V)$ and $(f, U) \succ (g, V)$ all imply $\text{Tr}^F(U)(Ff) = \text{Tr}^F(V)(Fg)$, which are straightforward consequences of the axioms of trace and of the functor being traced.

**Theorem 4.4** If $C$ is a partially traced category, $D$ a totally traced category and $F$ a traced functor from $C$ to $D$, then $F$ factors uniquely as

$$F = G \circ E_C$$

with $G : T(C) \to D$ a traced functor.

**Proof** Define $G$ as

$$GA = FA \quad \text{(on objects)}$$

$$G(f, U) = \text{Tr}^F(f) \quad \text{(on morphisms)}$$

the $m$ isomorphisms are those of $F$

This is well-defined thanks to lemma 4.3. Checking that $G$ is traced is routine, using the fact that $F$ is traced and the axioms of trace. Moreover for any $f : A \to B$ we have

$$(G \circ E_C)(f) = G(f, 1) = \text{Tr}^F(f) = \text{Tr}^1[\text{Id}_B \otimes m_1^{-1}(Ff)(\text{Id}_A \otimes m_1)]$$

using the isomorphism $m_1 : 1 \to F1$ and the sliding axiom. By the vanishing axiom and the coherence properties of $m_1$, $(G \circ E_C)(f) = (\text{Id}_B \otimes m_1^{-1})(Ff)(\text{Id}_A \otimes m_1) = Ff$. 

110
Now for the uniqueness of \( G \), suppose we have two traced functors \( G, K \) making our diagram commute. We readily see that we must have \( GA = KA = FA \), so \( G \) and \( K \) agree on objects. Moreover, the diagram tells us that they must agree on morphisms of the form \( (f, 1) \). Now, given \( (f, U) \) in \( T(C) \) we have that \( (f, U) = H_U[f, 1] \) and because \( G \) and \( K \) are traced we get

\[
G(f, U) = G(H_U[f, 1]) = Tr^{GU}[G(f, 1)] = Tr^{KU}[K(f, 1)] = K(H_U[f, 1]) = K(f, U)
\]

Conclusion

We gave a new proof of the representation theorem for partially traced categories, using an approach that is quite different from the original proof of the result. The interest of this theorem remains the same: all equational reasoning that can be carried out in totally traced categories (in particular using their graphical language) will be valid in partially traced categories provided the equated expressions are defined.

One can wonder about the relation between the two constructions, ours based on hiding and quotients and the previous one based on paths in paracategories, which must be related by a traced equivalence of categories, since they satisfy the same universal property. Another direction for future work would be to give a relaxed definition of partial traces that still enjoy the representation theorem and encompasses more general situations, such as infinite-dimensional Hilbert spaces, as mentioned in the introduction.

Acknowledgement

The author wishes to thank Phil Scott for his feedback and discussions on this topic and the anonymous referee for their useful comments and crucial suggestions.

References


VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs

Robert Rand$^2$ Steve Zdancewic$^3$

Computer and Information Sciences
University of Pennsylvania
Philadelphia, PA

Abstract

We introduce a Hoare-style logic for probabilistic programs, called VPHL, that has been formally verified in the Coq proof assistant. VPHL features propositional, rather than additive, assertions and a simple set of rules for reasoning about these assertions using the standard axioms of probability theory. VPHL’s assertions are partial correctness assertions, meaning that their conclusions are dependent upon (deterministic) program termination. The underlying simple probabilistic imperative language, PrImp, includes a probabilistic toss operator, probabilistic guards and potentially-non-terminating while loops.

Keywords: Hoare Logic, Formal Verification, Coq, Probabilistic Programming, Non-termination

1 Introduction

Hoare logic has long been used as a means of formally verifying programs and several papers have presented variants to reason about probabilistic programs [6,7,10,20]. There are significant differences between these approaches but all embrace certain design choices: They reason about sub-distributions instead of full distributions and they restrict the possibility of non-termination. The first limits us from introducing assertions like $Pr(2 = 2) = 1$, which frequently precedes the variable assignment $x := 2$, into our deductions. It also prevents us from taking the complements of our probabilities, which is critical for probabilistic reasoning. Eliminating while loops, or restricting us to those guaranteed to terminate, not only limits the kinds of programs we can analyze, it removes a core feature of Hoare logic: partial correctness assertions. We introduce a Verified Probabilistic Hoare Logic (VPHL) that reasons exclusively about full distributions and applies to potentially non-terminating programs. Importantly, VPHL is itself formally verified in the Coq proof assistant [9].

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
As probabilistic Hoare logics are increasingly used to verify critical code (for example, in the Easycrypt project [2]), it is important that the logic itself should rest on the firmest foundations.

Classical Hoare logic reasons about *program states*, mappings from identifiers (or variables) to values. Commands are partial functions from one state to another: for example \( x := 1 \) takes a state \( \theta \) to a state that maps \( x \) to 1 and is otherwise identical. The triple \( \{ x = 1 \} \ c \ { z = 3 \} \) asserts that if a state maps \( x \) to 1 and then we run the program \( c \) from that state, the resulting state will map \( z \) to 3, assuming that the program terminates.

In designing a Hoare logic for probabilistic programs, we would like to introduce triples with probabilistic propositions such as \( \{ \Pr(x = 1) > \frac{1}{2} \} \ c \ { \Pr(z = 3) = \frac{2}{3} } \). Since states are deterministic (a state either maps \( x \) to 1 or it doesn’t) we will have to reason about *state distributions*: the set of states a program may be in at a given point, and their associated weights (or probabilities). For example, if we toss a fair coin \( T \) in a deterministic state, we arrive at a state distribution: One state has \( T \) mapped to heads, and the other has \( T \) mapped to tails; each state has a weight of \( \frac{1}{2} \).

In section 2, we formalize our notion of a distribution. In the following section we introduce our simple probabilistic imperative language \( \text{PrImp} \). Sections 4–7 deal with \( \text{VPHL} \) itself, with a particular focus on the If and While rules. Following that (section 8), we discuss how termination analysis can be combined with our partial correctness assertions to yield precise characterizations of generally undecidable problems. We conclude with an example of our logic in practice, discussion of possible extensions to the logic, and a review of the related work in the area.

We will occasionally refer to an expanded version of this paper [21], which includes additional examples and discussion of alternative If and While rules. The underlying Coq development is online at [https://github.com/rnrand/VPHL](https://github.com/rnrand/VPHL).

### 2 Modeling Distributions

To begin, we need to formalize our notions of distributions and state distributions. A *distribution with finite support* is a multiset of elements \( \{ x_1, x_2, \ldots, x_n \} \) along with associated weights \( \{ w_1, w_2, \ldots, w_n \} \) in which every \( w_i \in [0, 1] \) and \( \sum_{i=1}^{n} w_i = 1 \). In our development, we will be concerned exclusively with distributions over program states.

We define distributions (ranged over by \( \Theta \)) inductively by means of a weighted tree structure. The leaves of the distribution contain states (mappings from identifiers to numeric and boolean values), denoted \( \theta \), and we use \( \text{Unit} \ \theta \) to lift a state to a one-element distribution. The *combine* operator \( \oplus_p \) takes two trees \( \Theta_1 \) and \( \Theta_2 \) and combines them to make one bigger tree, associating weight \( p \in (0, 1) \) to \( \Theta_1 \) and weight \( (1 - p) \) to \( \Theta_2 \).

For example, suppose we want to give \( \theta_1 \) and \( \theta_2 \) a weight of \( \frac{1}{2} \) each and give the remaining two-thirds of the weight to \( \theta_3 \). We can represent this distribution with
\[
A ::= \text{n} \mid \text{v} \mid A + A \mid A - A \mid A \ast A
\]
\[
B ::= \text{t} \mid \text{f} \mid A = A \mid A < A \mid B \land B \mid B \lor B
\]
\[
\mathcal{P}, \mathcal{Q} ::= Pr(B) = p \mid Pr(B) < p \mid Pr(B) > p \mid \mathcal{P} \land \mathcal{P} \mid \mathcal{P} \lor \mathcal{P}
\]

Fig. 1. \text{PrImp} Expressions and the Probabilistic Assertions of \text{VPHL}

either of the following trees:

either of the following trees:

which correspond to \((\text{Unit } \theta_1 \oplus_{\frac{1}{3}} \text{Unit } \theta_2) \oplus_{\frac{1}{5}} \text{Unit } \theta_3\) and \(\text{Unit } \theta_1 \oplus_{\frac{1}{5}} (\text{Unit } \theta_2 \oplus_{\frac{1}{5}} \text{Unit } \theta_3)\), respectively.

This gives rise to our notion of distribution equivalence. We say that \(\Theta_1 \equiv \Theta_2\) if the total weight given to every \(\theta\) in the distributions’ supports is equal.\footnote{In our Coq development we do not posit the decidability of state equivalence, hence distribution equivalence is defined in terms of boolean predicates over states, for which we can state equivalent lemmas.}

We now define probability over distributions. In the general case, for any distribution \(X\) over \(\mathcal{X}\) and function \(f : \mathcal{X} \rightarrow \text{Bool}\), \(\text{Pr}_X(f) = \sum \{w_i : f(x_i) = \text{t}\}\).

We can use this construction to derive all the standard laws of discrete probability theory (see the expanded paper and Coq development for more details).

In our logic, the elements of the distributions will always be states. The probability functions \(f\) are the lifted boolean expressions \(B\) from \text{PrImp} (see figure 1),
Where $B(\theta)$ is the value of $B$ in the given state. For example, let $\Theta$ be our distribution above and suppose that $\theta_1(x) = 2$, $\theta_2(x) = 1$ and $\theta_3(x) = 2$. Consider the boolean expression $b \equiv (x = 2)$. Then $b(\theta_1) = \mathbf{t}$, $b(\theta_2) = \mathbf{f}$ and $b(\theta_3) = \mathbf{t}$. Hence, $Pr_B(b) = \frac{1}{6} + 0 + \frac{2}{3} = \frac{5}{6}$.

### 3 A Simple Probabilistic Imperative Language

VPHL will let us prove assertions about PrImp, a probabilistic variant of the simple imperative language Imp from Software Foundations [19], with the addition of a coin flip operator toss. We present the big-step operational semantics of PrImp in figure 2 where $c / \Theta \Downarrow \Theta'$ means that if we evaluate $c$ in the state distribution $\Theta$ we arrive at the new distribution $\Theta'$. PrImp’s semantics are designed to satisfy the following principles:

(i) PrImp should contain an embedding of a deterministic programming language.
(ii) Deterministic commands should preserve probabilities.
(iii) Any program with a non-terminating branch should not terminate.

We satisfy these principles by “lifting” Imp such that every command behaves in its traditional way on Unit (or single-state) distributions. The Combine rule recursively applies a given command to all states in the support, terminating if and only if every such state terminates on the command, satisfying principle (iii) (we discuss the rationale for this specification in the following section). The new command $p := \text{toss}(y)$ splits a Unit distribution into two Unit states, the first with weight $p$ and $y$ set to $\mathbf{t}$ and the second with weight $(1 - p)$ and $y$ set to $\mathbf{f}$.

The following two lemmas follow directly from our operational semantics but are worth making explicit:

**Lemma 3.1 (Decomposition)** For any $c, p, \Theta_1, \Theta_2, \Theta'_1, \Theta'_2$,

\[
\frac{c / \Theta_1 \Downarrow \Theta'_1}{c / \Theta_1 \Downarrow \Theta'_1 \Downarrow \Theta'_2} \iff \frac{c / \Theta_1 \Downarrow \Theta'_2 \Downarrow \Theta_2}{c / \Theta_1 \Downarrow \Theta'_1 \Downarrow \Theta'_2}.
\]

**Lemma 3.2 (Step Determinism)** For any $c, \Theta, \Theta', \Theta''$,

\[
c / \Theta \Downarrow \Theta' \land c / \Theta \Downarrow \Theta'' \implies \Theta' = \Theta''.
\]
4 Hoare Logic Semantics

We now have sufficient background to formally define our Hoare triples. Let us begin with the formal definition of a classical (non-probabilistic) Hoare triple. Here $P$ and $Q$ are assertions about states, or more formally, mappings from states to propositions:

**Definition 4.1** We say that a classical Hoare Triple $\{P\} c \{Q\}$ is valid if $\forall \theta, \theta' : P(\theta) \land c / \theta \Downarrow \theta'$ implies $Q(\theta')$.

We call $P$ the precondition and $Q$ the postcondition.

In our Hoare logic, all assertions relate directly to probabilities, using the probabilistic assertions defined in figure 1. Note that the arithmetic and boolean expressions are the same ones used in our definition of probability over state distributions above, and in the commands of PrImp itself. In particular $[Pr(\mathcal{B}) = p](\Theta)$ translates into $Pr_{\Theta}(\mathcal{B}) = p$ and likewise for $<, >$, $\land$ and $\lor$ distribute as you would expect, and we use $\neq, \leq, \geq, \neg$ and $\rightarrow$ as abbreviations for the corresponding disjunctions. We express that a given boolean expression $b$ is true throughout a distribution by $Pr(\mathcal{B}) = 1$, which we abbreviate $\lceil b \rceil$, and call these assertions non-probabilistic.

**Definition 4.2** We say that a Hoare Triple $\{P\} c \{Q\}$ is valid in PrImp if $\forall \Theta, \Theta' : P(\Theta) \land c / \Theta \Downarrow \Theta'$ implies $Q(\Theta')$.

As mentioned in the previous section, a PrImp while loop terminates on a given distribution if and only if it terminates on every state in the distribution’s support. Hence, the following two programs do not terminate in our language, even though the second would traditionally terminate with probability 1:

\begin{align*}
y : = \text{toss}(\frac{3}{4}); & \text{ if } y \text{ then } x : = 4 \text{ else while } t \text{ do skip} & \quad (1) \\
y : = \text{toss}(\frac{2}{3}); & \text{ while } y \text{ do } y : = \text{toss}(\frac{2}{3}) & \quad (2)
\end{align*}

This is partly motivated by our approach to the Hoare If rule (see figure 4 later in this paper). We separately reason about each branch of the if statement, and then take the conjunction of their conclusions. If either of these conclusions is only vacuously true (since the branch doesn’t terminate), and the program is deemed to terminate, our conjunction will be false. Hence, we call such a program non-terminating as well.
5 Basic Hoare Logic Rules

We can now introduce VPHL itself. Figure 3 presents the basic rules of VPHL. The rules for if and while commands are presented in figures 4 and 5, with discussion deferred until later in the paper. We will present our soundness result up front:

**Theorem 5.1 (Soundness)** All of the VPHL rules presented in this paper are sound with respect to the semantics of PrImp.

We prove this theorem in the Coq development, where each rule is individually verified to be sound.

We do not claim completeness (that is, that everything possible to derived can be derived via our Hoare logic) in this paper, though we do demonstrate the usability of VPHL via examples (section 10).

The basic rules (with the exception of Toss) are preserved from classical Hoare logic. The toss command assigns an identifier y to either t or f, with probability p and (1 − p) respectively. For our Hoare logic, we restrict y from appearing in the precondition P. Since a freshly tossed y is necessarily independent of all previous probabilities, we can then update all statements of the form \( Pr(b) = q \) with \( Pr(b \land y) = pq \).

We define \( P \leq_P^y \), read “P conditioned on y with probability p”, inductively as follows:

\[
\begin{align*}
(Pr(b) = q) \leq_P^y & \equiv Pr(b \land y) = pq \land Pr(b \land \neg y) = (1 - p)q \\
(Pr(b) < q) \leq_P^y & \equiv Pr(b \land y) < pq \land Pr(b \land \neg y) < (1 - p)q \\
(Pr(b) > q) \leq_P^y & \equiv Pr(b \land y) > pq \land Pr(b \land \neg y) > (1 - p)q \\
(P_1 \land P_2) \leq_P^y & \equiv P_1 \leq_P^y \land P_2 \leq_P^y \\
(P_1 \lor P_2) \leq_P^y & \equiv P_1 \leq_P^y \lor P_2 \leq_P^y
\end{align*}
\]

Conveniently, the resulting rule is lossless. When we apply the Toss rule \( \{P \} y := \text{toss}(p) \{P \leq_P^y \} \), any proposition that was true in the precondition will remain true in the postcondition. To formalize this:

**Lemma 5.2** For all P, \( P \leq_P^y \) entails P.

We can show this by simply marginalizing over y in each atomic proposition. For instance, \( P(b \land y) = \frac{1}{5} \) and \( P(b \land \neg y) = \frac{2}{5} \) imply that \( P(b) = \frac{1}{5} + \frac{2}{5} = \frac{3}{5} \). We will use this technique, often implicitly, throughout the paper.

6 Conditioning on Probabilistic Guards

The if command is the most difficult to reason about for straightforward reasons. Unlike the previous commands, which behave identically on all of the states in the distribution’s support, the if command will run one command wherever the guard is true, and another whenever the guard is false.

In the simplest case, the value of the guard will take on the same value throughout the distribution (we say that the guard is deterministic), so it’s sensible to have a usable If rule, specific to that case. In figure 4 we present such a rule.
Lemma 6.1

For any state distribution $\Theta$ where $Pr(y) = p$ and $p \in (0, 1)$, $\exists \Theta_1, \Theta_2$ such that $Pr_{\Theta_1}(y) = 1, Pr_{\Theta_2}(y) = 0$ and $\Theta \equiv \Theta_1 \oplus_p \Theta_2$.

We can now reason about the sub-distribution in which $y$ is true and the one in which it’s false separately, and then recombine their postconditions into a single postcondition.

In order to ensure that we’re reasoning about the right part of the distribution, we split the assertion $P$ into two parts, $P_1 \mid y$ and $P_1 \mid \neg y$ (shown in figure 4) as defined below. Any part of the assertion that does not mention a value for the guard is discarded.

\[
\begin{align*}
(Pr(X) = q) \mid y &\equiv Pr(X \land y) = q \\
(Pr(X) < q) \mid y &\equiv Pr(X \land y) < q \\
(Pr(X) > q) \mid y &\equiv Pr(X \land y) > q \\
(P_1 \land P_2) \mid y &\equiv P_1 \mid y \land P_2 \mid y \\
(P_1 \lor P_2) \mid y &\equiv P_1 \mid y \lor P_2 \mid y
\end{align*}
\]

For reasoning about sub-distributions, we take the approach of scaling both sub-distributions up to complete distributions, allowing us to reason normally about both without having to restrict our logic. We achieve this via the scaling operator $*$ that scales up all of probabilities in preconditions and postconditions by $\frac{1}{p}$ or $\frac{1}{1-p}$ where $p$ is the probability of the guard. Since we’re looking to reason about sub-distributions (not just sub-assertions), we need the following lemmas:

Lemma 6.2

For any $\Theta_1, \Theta_2$ where $Pr_{\Theta_1}(y) = 1$ and $Pr_{\Theta_2}(y) = 0, [P_1 | y](\Theta_1 \oplus_p \Theta_2)$ if and only if $[\frac{1}{p} \times P](\Theta_1)$

Lemma 6.3

For any $\Theta_1, \Theta_2$ where $Pr_{\Theta_1}(y) = 1$ and $Pr_{\Theta_2}(y) = 0, [P_2 | \neg y](\Theta_1 \oplus_p \Theta_2)$ if and only if $[\frac{1}{1-p} \times P_2](\Theta_2)$
\[
\begin{array}{c}
P \to D \quad \{ P \land [y] \} \ c \ \{ P \} \\
D \to D' \quad \{ D \land [y] \} \ c \ \{ D \land ([y] \lor [-y]) \} \quad D \text{ is non-probabilistic} \\
\{ P \land ([y] \lor [-y]) \} \ \text{while } y \ c \ \{ P \land [-y] \}
\end{array}
\]

Fig. 5. The While Rule

We demonstrate that if \( P \mid y \) holds of the full distribution, then \( p \ast P \) holds of the sub-distribution in which \( y \) is true. We can then reason about the application of \( c_1 \) and \( c_2 \) to those distributions separately. In order to combine the two postconditions, we add back the \([y] \lor [-y]\) to all the probabilistic expression, ensuring that they do not overlap. In order to prevent statements of the form \( P(\cdot \land \neg y) = p \) from appearing in the combined postconditions, we restrict \( y \) from being reassigned in \( c_1 \) or \( c_2 \).

The reader may observe that the rules of VPHL are scale-invariant – that is, they do not depend on the value \( p \) on the right hand side of \( Pr(X) = p \). This suggest that instead of scaling our \( P_1 \) and \( P_2 \), we could simply reason about \( P_1 \mid y \) and \( P_2 \mid \neg y \) separately. Indeed, with minor restrictions such an If rule would be sound, and we discuss this rule in the expanded paper.

7 The While Rule

In order to explain and justify the form of our While rule (see figure 5), let us consider the classical Hoare logic While rule, and why it fails for probabilistic programs:

\[
\{ P \land y \} \ c \ \{ P \} \\
\{ P \} \ \text{while } y \ c \ \{ P \land [-y] \}
\]

Consider the following Hoare triple:

\[
\begin{align*}
\{ Pr(y') = \frac{1}{2} \land [y \rightarrow y'] \} \\
\text{while } y \ c \ (y' := \text{toss}(\frac{1}{2}); y := (y' \land i < 5); i++) \\
\{ Pr(y') = \frac{1}{2} \land [y \rightarrow y'] \land [-y] \}
\end{align*}
\]

There are two major problems with the derivation of this triple. The first is that a contradiction appears in the precondition: \( Pr(y') = \frac{1}{2} \land [y \rightarrow y'] \land [y] \) implies that \( Pr(y') = \frac{1}{2} \) and \( Pr(y') = 1 \). Moreover, the post condition is false – if initially \( i = 1 \) we run the loop up to 5 times, and the probability of \( y' \) coming out as true is \( 2^{-5} = \frac{1}{32} \).

This illustrates that probabilistic invariants are not guaranteed to hold if different branches of the distribution traverse the loop a different number of times. Hence we introduce the restriction (figure 5) that the value of \( y \) must remain non-probabilistic (that is, either \([y]\) or \([-y]\)) upon the completion of every iteration loop, ensuring that if the loop terminates, all branches terminate concurrently.

Or rather, all branches should terminate concurrently but we run into some difficulty: While \( P \) may hold for a given distribution \( \Theta_1 \oplus_p \Theta_2 \) and be sufficient to preserve both \( P \) and the determinism of \( y \) upon running \( c \), there’s no guarantee
that either $\Theta_1$ or $\Theta_2$ satisfy $P$ and thereby preserve determinism. However, we have a nice subset of non-probabilistic assertions (that is, assertions where every probability is either zero or one) that do satisfy this property.

**Lemma 7.1** For any non-probabilistic assertion $P$, $P(\Theta_1 \oplus_p \Theta_2)$ implies $P(\Theta_1)$ and $P(\Theta_2)$ for any $p \in (0, 1)$.

Our While rule therefore requires a non-probabilistic invariant that preserves the determinism of the guard to be exhibited along with the probabilistic one. In practice (as opposed to in general) this proves to be fairly straightforward, and often quite convenient. For example, we may only need to show that our counter deterministically takes on a specific value in $\{0, 1, \ldots, n\}$ and that $y$ depends only on $i$, for example when analyzing a random walk on an $n$-vertex graph. We separately reason about the main invariant, which may include probabilistic propositions.

### 8 Reasoning About Probabilistically Terminating Programs

Consider again the following simple program from section 4.

```plaintext
y := toss(\frac{2}{3}); if y then x := 4 else while t do skip
```

We call this program $c_{\text{partial}}$. According to the operational semantics of $PrImp$, this program doesn’t terminate, hence $\{P\} c_{\text{partial}} \{Q\}$ is valid (though generally not derivable in $VPHL$) for any assertions $P$ and $Q$. However, if we consider the truly probabilistic program (one with a seed of random numbers and probabilistic steps) that corresponds to $c_{\text{partial}}$, we know that $Pr(x = 4 \land y) = 1$ upon successful termination. More generally, for an arbitrary `else` branch which doesn’t reassign $y$ we know the following:

$$Pr(x = 4 \land y) = \begin{cases} 
1 & \text{if the `else` branch terminates with probability 0} \\
\frac{2}{3} & \text{if the `else` branch terminates with probability 1} \\
p \in (\frac{2}{3}, 1) & \text{otherwise}
\end{cases}$$

This suggest that if we have information about the probabilities of each branch terminating, we can combine this with a straightforward analysis of each branch to derive precise probabilities for the outcomes. This requires that (1) every branch terminates with probability zero or one and (2) that every branch is analyzed independently and has its own post-conditions.

We now note four important features of our logic.

(i) Each branch of an `if` statement has its own, independent derivation

(ii) `if` statement post-conditions explicitly mention the value of the guard

(iii) The While rule requires that the loop terminates with probability 0 or 1

(iv) There is no way to vacuously derive False from non-termination

Hence, with the simple restrictions that the guards on `if` statements are not reassigned in the program or marginalized over in the Hoare logic derivation, we
can combine our logic with termination analysis to precisely characterize probabilistically terminating programs.

Consider the following example, where $c_1$ through $c_4$ are programs that do not modify $y_1, y_2$ or $y_3$ and are not guaranteed to terminate:

$$y_1 := \text{toss}(p); \ y_2 := \text{toss}(q); \ y_3 := \text{toss}(r);$$

if $y_1$ then
  if $y_2$ then $c_1; x := 1$ else $c_2; x := 2$
else
  if $y_3$ then $c_3; x := 3$ else $c_4; x := 4$

Assuming that $c_1, c_2, c_3$ and $c_4$ are susceptible to analysis by $VPHL$ (i.e. their while loops terminate or loop deterministically), we should be able to derive the following post-condition:

$$Pr(x = 1 \land y_1 \land y_2) = pq \quad \land \quad Pr(x = 2 \land y_1 \land ¬y_2) = p(1 − q) \quad \land$$

$$Pr(x = 3 \land ¬y_1 \land y_3) = (1 − p)r \quad \land \quad Pr(x = 4 \land ¬y_1 \land ¬y_3) = (1 − p)(1 − r)$$

Now imagine we know that only $c_3$ never halts. Then the prior probabilities of $x$ taking on the values 1, 2, and 4 are $pq$, $p(1 − q)$ and $(1 − p)(1 − r)$ respectively, corresponding to the values in our post-condition. The prior probability of the program looping is $(1 − p)r$. Upon the program’s successful termination, we have $x = 1$ with probability $pq \frac{p}{1−(1−p)r}$ and similarly for the other terminating outcomes.

On the other hand, if we don’t know whether any of the branches terminate we can’t come to any conclusion regarding the probabilities of the outcomes. This isn’t surprising: it follows directly from the fundamental results of computability theory. In the general case, the probability of any proposition upon program termination depends directly upon the weight of the branches that do terminate, reducing the problem of assigning probabilities directly to the Halting Problem.

9 Extending VPHL

Before we use $VPHL$ to verify a few sample programs, we will introduce some notations that will make our task easier. The first is drawing from a uniform distribution. We define $\text{UNIFORM}$ as syntactic sugar for a series of tosses:

$$\begin{align*}
  x & := \text{UNIFORM}(1) \equiv x := 1 \\
  x & := \text{UNIFORM}(N) \equiv u := \text{toss}(\frac{1}{N}) \\
  \text{if } u \text{ then } x & := N \text{ else } x := \text{UNIFORM}(N − 1)
\end{align*}$$

where $u$ is a reserved boolean variable.

We can prove the associated Hoare rule\footnote{This is meant to illustrate what we can in principle prove in the logic, though this rule, and the examples given have not been verified in Coq.}

$$\frac{x \text{ free in } P}{\{P\} x := \text{UNIFORM}(N) \{P \land \frac{x}{N} \} \text{ Uniform}}$$
where $P_{\leq N}$ is the analogue to $P_{\leq N}^p$ with $[P(b) = p]_{\leq N} \equiv P(b \land x = 1) = \frac{1}{N} \land \cdots \land P(b \land x = N) = \frac{1}{N}$.

Another useful feature missing from the specification of VPHL is the ability to include identifiers on the right side of probabilistic assertions. There is an obvious reason for this: Probabilistic Assertions refer to a distribution, and different states in that distribution may map a given identifier to different values. At the same time, some identifiers will be deterministically set in our program, and we might like to reference those. We can express the desired assertions as follows: $\forall k < N, Pr(i = k) = 1 \rightarrow Pr(b) = f(k)$, where $b$ is any boolean expression, $i$ is the identifier we want to include in the probability, and $f$ is the real-valued function that we want to depend on $i$. The universal quantifier here represents a series of conjunctions.

Similarly, we would like to be able to say that an identifier $i$ deterministically takes on some value in $\{1, 2, \ldots, n\}$. We write this as $i \in \{1, 2, \ldots, n\}$ which is shorthand for $[i = 1] \lor [i = 2] \lor \cdots \lor [i = n]$.

Both of the above constructs require us to have an upper bound on the possible values for $i$, but this is often the case, as in the examples we analyzed.

10 Simulating A Uniform Distribution

As a simple demonstration of our logic, let us attempt to prove the Hoare logic rule above. Proving the rule in the general case would require both induction over the precondition and induction over $N$. Instead let’s prove the simple case of a uniform distribution for $N = 3$. In order to use our If rule we will replace $u$ with $u_1$ and $u_2$.

Algorithm 1. \textsc{uniform}(3)
\begin{verbatim}
  $u_1 := \text{toss}(1/3)$;
  if $u_1$ then
    $x := 3$
  else
    $u_2 := \text{toss}(1/2)$
    if $u_2$ then
      $x := 2$
    else
      $x := 1$
  end if
end if
\end{verbatim}

We will now try to prove that $\{P\} x := \text{UNIFORM}(N) \{P_{\leq N}\}$ for an atomic proposition $P$ of the form $Pr(b) = p$. We will implicitly use the consequence rule to transform $Pr(b) = p$ to $Pr(b \land 2 = 2) = p$ or $Pr(b) = p \land Pr(\tau) = 1$ as needed throughout the program. We will occasionally make this explicit by means of an arrow symbol ($\rightarrow$). We show sub-derivations indented and inline, as is common for Hoare logic analysis:
Algorithm 2. Uniform Derivation

\{ Pr(b) = p \}

\[ u_1 := \text{toss}(1/3); \]

\{ Pr(u_1) = 1/3 \land Pr(b \land u_1) = p/3 \land Pr(b \land \neg u_1) = 2p/3 \}

\textbf{if} \ u_1 \ \textbf{then}

\{ 3/1 \ast [Pr(b) = p/3] \land [u_1] \} \rightarrow \{ Pr(b \land 3 = 3) = p \}

\[ x := 3 \]

\{ Pr(b \land x = 3) = p \}

\textbf{else}

\{ 3/2 \ast [Pr(b) = 2p/3] \land [\neg u_1] \} \rightarrow \{ Pr(b) = p \}

\[ u_2 := \text{toss}(1/2); \]

\{ Pr(u_2) = 1/2 \land Pr(b \land u_2) = p/2 \land Pr(b \land \neg u_2) = p/2 \}

\textbf{if} \ u_2 \ \textbf{then}

\{ Pr(b) = p \}

\[ x := 2 \{ Pr(b \land x = 2) = p \} \]

\textbf{else}

\{ Pr(b) = p \}

\[ x := 1 \{ Pr(b \land x = 1) = p \} \]

\textbf{end if}

\{ Pr(b \land x = 2 \land u_2) = p/2 \land Pr(b \land x = 1 \land \neg u_2) = p/2 \}

\textbf{end if}

\{ Pr(b \land x = 3 \land u_1) = p/3 \land Pr(b \land x = 2 \land u_2 \land \neg u_1) = p/3 \land Pr(b \land x = 1 \land \neg u_2 \land \neg u_1) = p/3 \}

Note that the \( u_i \)'s are still present in the conclusion. We could in principle conclude only that \( Pr(b \land x = 1) = 1/3 \land Pr(b \land x = 2) = 1/3 \land Pr(b \land x = 3) = 1/3 \) by carrying \( P(b) = p \) all the way to the postcondition and then noticing that the three conjuncts add up the probability of \( b \) itself, but typically we want to maintain the values of the guards, and we don’t wish to complicate things unnecessarily.

In the expanded version of this paper [21], we show how this proof is simplified by means of an alternative If rule that doesn’t require scaling. We further illustrate the use of the While rule by analyzing a random walk.

11 Related Work

The most significant work in representing distributions in Coq was made by the ALEA project [18] based on the work of [1]. ALEA introduces its own axiomatic library for the unit interval and multiple notions of distributions. ALEA is designed to reason directly about probabilistic programs, and forms a foundation of the Certicrypt cryptographic tool [4]. Our goals in this paper were far more limited than ALEA’s in terms of what we aimed to represent, namely discrete distributions with finite support. For these, a simple tree based structure of objects proved sufficient and (significantly) easy to reason about. Our unit intervals are based on the Coq real number library, restricted to \([0, 1]\).

Hoare Logic for deterministic programs was introduced in a foundational paper by C. A. R. Hoare [12]. The first attempt to extend this style of reasoning to probabilistic programs appeared in Ramshaw’s thesis [20], which was based upon...
Kozen Semantics [14] and featured the following rule for conditionals:

$$\frac{\{P | b\} \ c_1 \ \{Q_1\} \ \{P | \neg b\} \ c_2 \ \{Q_2\}}{\{P\} \text{ if } b \text{ then } \ c_1 \ \text{else } \ c_2 \ \{Q_1 + Q_2\}}$$

where \(A | b\) (pronounced “\(A\) restricted to \(b\)”) breaks up the predicate into “frequencies” (or sub-distributions) in which \(b\) is true and in which \(b\) is false. The plus operator in the conclusion means that some part of the distribution satisfies \(Q_1\) and another satisfies \(Q_2\). Reasoning about sub-distributions brings with it a number of difficulties that we set out to avoid, including the restriction that \(Pr(t) = 1\) is not true in any strict sub-distribution.

Den Hartog and De Vink’s logic \(pH\) [10] has a similar construction for the If rule but with the following \(\oplus\) operator:

$$\frac{\{b \oplus P\} \ c_1 \ \{Q_1\} \ \{-b \oplus P\} \ c_2 \ \{Q_2\}}{\{P\} \text{ if } b \text{ then } \ c_1 \ \text{else } \ c_2 \ \{Q_1 + Q_2\}}$$

Interestingly, \(\oplus\) is primarily defined on state distributions, not assertions. Modifying their example (using multiset notation), if \(\Theta = \{\{\theta_1, \frac{1}{2}\}, \{\theta_2, \frac{1}{4}\}, \{\theta_3, \frac{1}{4}\}\}\) where \(\theta_1\) and \(\theta_3\) satisfy \(b\), then \(b?\ \Theta = \{\{\theta_1, \frac{1}{2}\}, \{\theta_3, \frac{1}{4}\}\}\) and \(-b?\ \Theta = \{\{\theta_2, \frac{1}{4}\}\}\). Then \(b? \ P(\Theta)\) asserts that there is a state \(\Theta'\) such that \(b?\ \Theta' = \Theta\) and \(P(\Theta)\). How to integrate this with the rest of the logic isn’t made clear, but it would need to involve weakening. Again, the postcondition refers to two sub-distributions.

There are two sufficient conditions for application of the \(pH\) While rule: Either the loop is “terminating” (guaranteed to terminate on all states) or it is “\(c, s\)-closed”, meaning there is a lower bound on the probability of termination on each iteration. While the first condition seems difficult to guarantee, the latter allows us to reason about a range of programs that lie outside the scope of \(PrImp\). On the other hand, it has the significant limitation that it cannot reason about potentially non-terminating programs.

Chadha et al. [6] take an approach that is similar to ours for a language without While loops, and demonstrate the completeness of their logic. They take an interesting approach to the Toss rule, which, like the assignment rule, requires us to rewrite the precondition in the form of the postcondition. For example, we use the following triple to attain a postcondition of \(P(y) = \frac{1}{3}\) upon tossing a coin with bias one-third:

$$\left\{ \frac{1}{3}P(t) + \frac{2}{3}P(t) = \frac{1}{3} \right\} y := \text{toss}(\frac{1}{3}) \ \{P(y) = \frac{1}{3}\}.$$  

Given that the identifier cannot appear in the precondition here either (unlike assignment, where we might assign \(x := x + 1\)), it’s not clear that this adds expressivity, but reasoning backwards in this fashion may help with weakest precondition proofs.

Their If rule takes the following form:

$$\frac{\{P_1\} \ c_1 \ \{Pr(X) = p_1\} \ \{P_2\} \ c_2 \ \{Pr(X) = p_2\}}{\{P_1/b \land P_2/\neg b\} \text{ if } b \text{ then } \ c_1 \ \text{else } \ c_2 \ \{Pr(X) = p_1 + p_2\}}$$

\(P/b\) is similar to our \(P | y\), inserting \(\land b\) into all the probabilistic terms in the assertion. However, the sub-assertions are not scaled, instead we again reason about

125
them in a sub-probability space. In order to allow us to combine postconditions, it includes the significant restriction that both branches’ postconditions must refer to the same probability – often this will require significant weakening. In order to derive a complex postcondition, we need to apply the If rule repeatedly, and join the results via conjunction and disjunction rules.

Interestingly, this restriction wasn’t present in an earlier version of the logic [7]. That version used an alternative If command that took in a free boolean identifier and set it to \( t \) or \( f \) depending on which branch was taken. This allowed a postcondition of the form \( P_1/y \land P_2/\neg y \) (where \( y \) was the new identifier) at the cost of a non-standard If rule and the proliferation of fresh identifiers.

There has also been considerable work done in related formal systems, including Probabilistic Guarded Command Language [17] (formalized in HOL4 [13] and Isabelle/HOL [8]), Dynamic Logic [11, 15] and Kleene Algebra with Tests [16]. We refer the reader to Vasquez et al. [22] for a comparison of Probabilistic Hoare Logic and pGCL and to the related work section of Chadha et al. [6] for a broader discussion of the approaches to probabilistic verification.

Finally, related to the Certicrypt project mentioned above [4], the EasyCrypt cryptographic tool [2] is based upon two logical systems: pRHL, a Probabilistic Relational Hoare Logic for reasoning about two programs simultaneously and pHL, a Probabilistic Hoare Logic for reasoning about a single program. Introduced in Barthe et al. [3], pRHL is based upon Benton’s Relational Hoare Logic [5] and uses a technique called “lifting” to avoid talking directly about probabilities allow us to derive probabilities within sub-distributions. pHL (discussed briefly in the Easycrypt tutorial [2] and elsewhere) reasons about transitions from one state to another in probabilistic terms, but a full account of its semantics and underlying logic has not yet been published. Easycrypt demonstrates the utility of a probabilistic Hoare logic in a cryptographic setting.

12 Future Work

Both PrImp and VPHL are limited by design. PrImp expressions are limited to boolean and natural numbers; for ease of analysis we haven’t included data structures, function calls or recursion, among other language features. VPHL is similarly limited, primarily by its lack of quantification. Existential quantifiers would allow us to express crucial ideas like independence: \( A \) and \( B \) are independent in \( \Theta \) iff \( \exists p, q \text{ s.t. } Pr(A) = p \land Pr(B) = q \land Pr(A \land B) = pq \).

VPHL is meant to provide the groundwork for the further study of probabilistic Hoare logics and for their application. It is extensible, meaning that we can add new rules without impacting the language. The new Hoare rules would fall into one of two categories: core rules and derived rules.

A core rule is one that is sound within the Hoare logic, but not redundant in the context of the existing rule. Any new core rules would probably be similar in form to the rules in [6]. They would emphasize completeness over usability, and create a set of rules that can be used to deduce any valid Hoare triple.

A derived rule is a rule that can be derived using existing rules from the logic. Though in the strictest sense these rules are redundant, they enable us to efficiently
and intuitively reason about programs. Adding such rules to our logic could have a substantial impact on its usability.

In this paper we’ve walked a line between usability and completeness, and there is still a long way to go on the usability front. We envision a bevy of rules for a variety of probabilistic constructs, extending what we did with the \textsc{Uniform} rule above. There are a number of areas, including cryptography, privacy, machine learning and randomized algorithms, which beg for formal analysis methods, and a corresponding universe of domain-specific logics that can be tailored to these problems. \textsc{VPHL} should provide a solid foundation for this future work.

References


An Effect-Theoretic Account of Lebesgue Integration

Bart Jacobs and Bram Westerbaan

Institute for Computing and Information Sciences (iCIS), Radboud University Nijmegen, The Netherlands.

Abstract

Effect algebras have been introduced in the 1990s in the study of the foundations of quantum mechanics, as part of a quantum-theoretic version of probability theory. This paper is part of that programme and gives a systematic account of Lebesgue integration for \([0,1]\)-valued functions in terms of effect algebras and effect modules. The starting point is the ‘indicator’ function for a measurable subset. It gives a homomorphism from the effect algebra of measurable subsets to the effect module of \([0,1]\)-valued measurable functions which preserves countable joins. It is shown that the indicator is free among these maps: any such homomorphism from the effect algebra of measurable subsets can be thought of as a generalised probability measure and can be extended uniquely to a homomorphism from the effect module of \([0,1]\)-valued measurable functions which preserves joins of countable chains. The extension is the Lebesgue integral associated to this probability measure. The preservation of joins by it is the monotone convergence theorem.

Keywords: Effect algebra, effect module, Lebesgue integration

1 Introduction

Integration is a fundamental mathematical technique developed to compute quantities such as lengths of curves, areas of surfaces, volumes of solids, averages of distributions, Fourier transforms of functions, solutions to differential equations, and so on. Roughly speaking, the integral assigns to a function the area under its graph (counting the area under the \(x\)-axis negatively). The notation \(\int f(x) \, dx\) for the integral of \(f\) suggests that it should be thought of as a sum ("\(\int\)" is an elongated "s") of uncountably many rectangles \(f(x) \, dx\) of infinitesimal width \(dx\). While this makes for an elegant picture, a formal definition of the integral requires a different approach: for instance, by approximating \(f\) by basic functions for which the integral is easily determined.

In probability theory, integration is used for calculating probabilities of events and expected values of random variables (among many other things). In the theory

\footnote{bart@cs.ru.nl and bram.westerbaan@cs.ru.nl}

This paper is electronically published in

Electronic Notes in Theoretical Computer Science

URL: www.elsevier.nl/locate/entcs
of continuous probabilistic computation, integration is used for sequential composition (of Markov kernels, or coalgebras of the Giry monad), see e.g. [17,19]. Integration is also used for calculation weakest preconditions of quantitative predicates (random values), see e.g. [14].

This paper gives an elementary account of Lebesgue integration, using basic measure theory. It is restricted to measurable functions $X \to [0,1]$ to the unit interval, which may be understood as fuzzy predicates. What distinguishes our account from the traditional one is that it makes systematic use of the notions of effect algebra and effect module, where an effect module is an effect algebra with scalar multiplication, where scalars are taken from $[0,1]$. These effect structures emerged in the foundations of quantum mechanics, as part of a quantum-theoretic version of probability theory (see [6] for an overview). It turns out that the basic notions of Lebesgue integration can be formulated very naturally in terms of $\omega$-(complete)effect algebras and $\omega$-effect modules. For instance, for a measurable space $X$, with set $\Sigma_X$ of measurable subsets,

- the $\sigma$-algebra $\Sigma_X$ of measurable subsets is an $\omega$-effect algebra;
- the set $\text{Meas}(X,[0,1])$ of measurable functions $X \to [0,1]$ is an $\omega$-effect module;
- the indicator function gives a map $1_{(-)} : \Sigma_X \to \text{Meas}(X,[0,1])$ which is a homomorphism of $\omega$-effect algebras — where $1_M(x) = 1$ if $x \in M$ and $1_M(x) = 0$ if $x \notin M$;
- moreover, this indicator map is free in the following sense: for every $\omega$-complete effect module $E$, and for each probability measure (homomorphism of $\omega$-effect algebras) $\phi : \Sigma_X \to E$, there is a unique homomorphism of effect modules $\overline{\phi} : \text{Meas}(X,[0,1]) \to E$ with $\overline{\phi} \circ 1_{(-)} = \phi$. This free extension $\overline{\phi}$ precisely is Lebesgue integration! It sends $p \in \text{Meas}(X,[0,1])$ to the integral $\overline{\phi}(p) = \int p \, d\phi \in E$.

These bullet points summarise the main contributions of the paper. The definition of the integral $\int p \, d\phi \in E$ proceeds in two stages, as usual, namely first for step functions (using the effect module structure of $E$), and then for any measurable $p$ function by writing $p$ as an $\omega$-join $\bigvee$ of an ascending chain of step functions (using the $\omega$-completeness of $E$). Much of the work of the paper goes into verifying that the usual arguments can be adapted to the setting of $\omega$-effect modules.

In the end one may wonder how much of a restriction our use of $[0,1]$-valued functions is. These functions form an effect module. In [16] it is shown that the category of effect modules is equivalent to the category of order unit spaces, via a process called totalisation. By applying such totalisation one obtains the bounded $\mathbb{R}$-valued functions from the $[0,1]$-valued ones. In this way one can extend integration from $[0,1]$-valued functions to bounded $\mathbb{R}$-valued functions.

2 Effect algebras and effect modules

Effect algebras have been introduced in mathematical physics [8] (and also [3,10]), in the investigation of quantum probability, see [6] for an overview. An effect algebra is a partial commutative monoid $(M,0,\odot)$ with an orthocomplement $(-)\perp$.  

130
One writes \( x \perp y \) if \( x \otimes y \) is defined. The formulation of the commutativity and associativity requirements are a bit involved, but essentially straightforward. The orthocomplement satisfies \( x^{\perp \perp} = x \) and \( x \otimes x^\perp = 1 \), where \( 1 = 0^\perp \). There is always a partial order, given by \( x \leq y \) iff \( x \otimes z = y \), for some \( z \). Then: \( x \perp y \) iff \( x \leq y^\perp \).

The main example is the unit interval \([0,1]\) \( \subseteq \mathbb{R} \), where addition + is obviously partial, commutative, associative, and has 0 as unit; moreover, the orthocomplement is \( r^\perp = 1 - r \). An \( \omega \)-effect algebra (also called \( \sigma \)-effect algebra) additionally has joins \( \bigvee_n x_n \) of countable ascending chains \( x_1 \leq x_2 \leq \cdots \). We write \( \mathbf{EA} \) for the category of effect algebras, with as morphisms maps preserving \( \otimes \) and 1 — and thus all other structure. The morphisms in the subcategory \( \omega \)-\( \mathbf{EA} \) \( \hookrightarrow \mathbf{EA} \) of \( \omega \)-effect algebras are those that preserve joins of \( \omega \)-chains.

For each set \( X \), the set \([0,1]^X\) of fuzzy predicates on \( X \) is an \( \omega \)-effect algebra, with pointwise operations. Each Boolean algebra \( B \) is an effect algebra with \( x \perp y \) iff \( x \wedge y = \perp \); then \( x \otimes y = x \vee y \). In a quantum setting, the main example is the set of effects \( \mathcal{E}_f(\mathcal{H}) \) on a Hilbert space \( \mathcal{H} \) (that is, bounded linear operators \( A: \mathcal{H} \rightarrow \mathcal{H} \) with \( 0 \leq A \leq I \), see e.g [6,13]).

An effect module is an ‘effect’ version of a vector space. It involves an effect algebra \( E \) with a scalar multiplication \( s \cdot x \in E \), where \( s \in [0,1] \) and \( x \in E \). This scalar multiplication must preserve 0, \( \otimes \) in each variable separately. The sets \([0,1]^X\) and \( \mathcal{E}_f(\mathcal{H}) \) are clearly such effect modules. In the subcategory \( \mathbf{EMod} \hookrightarrow \mathbf{EA} \) of effect modules, maps additionally commute with scalar multiplication. We use \( \omega \)-\( \mathbf{EMod} \hookrightarrow \mathbf{EMod} \) for the subcategory of \( \omega \)-complete effect modules, with effect module maps that preserve joins of \( \omega \)-chains.

We need the following results about effect modules.

**Lemma 2.1** For elements \( x, y \) in an effect module, and for scalars \( r, s \in [0,1] \),

(i) \( (r \cdot x)^\perp = (r^\perp \cdot x) \otimes x^\perp \);

(ii) \( x \perp y \) implies \( r \cdot x \perp s \cdot y \).

**Proof** We obtain \( (r \cdot x)^\perp = r^\perp \cdot x \otimes x^\perp \) by uniqueness of orthocomplements:

\[
r \cdot x \otimes r^\perp \cdot x \otimes x^\perp = (r \otimes r^\perp) \cdot x \otimes x^\perp = 1 \cdot x \otimes x^\perp = x \otimes x^\perp = 1.
\]

Next, if \( x \perp y \), then \( x \leq y^\perp \), and thus \( r \cdot x \leq x \leq y^\perp \). Taking complements, we see that \( s \cdot y \leq y \leq (r \cdot x)^\perp \). This means \( r \cdot x \perp s \cdot y \).

**3 Measurable spaces and functions**

A measurable space \( (X, \Sigma_X) \) (or simply \( X \)) is a pair consisting of a set \( X \) and a \( \sigma \)-algebra \( \Sigma_X \subseteq \mathcal{P}(X) \). The latter is a collection of measurable subsets closed under \( \emptyset \), complements (negation), and countable unions. The measurable subsets form a Boolean algebra in which countable joins exist — so \( \Sigma_X \) is an \( \omega \)-effect algebra.

A function \( f: X \rightarrow Y \) between measurable spaces — that is, from \( (X, \Sigma_X) \) to \( (Y, \Sigma_Y) \) — is called measurable if \( f^{-1}(M) \in \Sigma_X \) for each \( M \in \Sigma_Y \). This yields a category \( \mathbf{Meas} \), which comes with a functor \( \Sigma_{(-)}: \mathbf{Meas} \rightarrow \omega \)-\( \mathbf{EA}^{\text{op}} \). With each topological space \( X \) one associates the least \( \sigma \)-algebra containing all open subsets,
called the Borel algebra/space on $X$. In particular the unit interval $[0, 1]$ forms a measurable space. Its measurable subsets are generated by the intervals $(q, 1]$, where $q$ is a rational number in $[0, 1]$.

Measurable functions have more order structure than continuous ones: they are closed under countable joins.

**Lemma 3.1** Let $X$ be a measurable space, and $Y$ a topological space.

(i) The set $\text{Meas}(X, [0, 1])$ of measurable functions $X \to [0, 1]$ is an $\omega$-effect module. In particular, it is closed under joins of ascending $\omega$-chains.

(ii) The set $\text{Top}(Y, [0, 1])$ of continuous functions $Y \to [0, 1]$ is an effect module, but not always an $\omega$-effect module: some ascending $\omega$-chains of continuous functions have no join.

These mappings $X \mapsto \text{Meas}(X, [0, 1])$ and $Y \mapsto \text{Top}(Y, [0, 1])$ yield functors:

$$
\begin{align*}
\text{Meas} & \longrightarrow \omega\text{-EMod}^{\text{op}} \\
\text{Top} & \longrightarrow \text{EMod}^{\text{op}}
\end{align*}
$$

**Proof** The measurable functions $X \to [0, 1]$ form an effect module, using pointwise the effect module structure from the unit interval $[0, 1]$. To show that they are closed under joins let $p_1 \leq p_2 \leq p_3 \leq \cdots$ be measurable functions $p_n: X \to [0, 1]$. We must show that the (pointwise) join $p = \bigvee_n p_n$ in $[0, 1]^X$ is again measurable. Since subsets of the form $(r, 1]$ with $r \in [0, 1]$ generate the Borel $\sigma$-algebra on $[0, 1]$ it suffices to show that $p^{-1}(r, 1])$ is measurable. Note that for $x \in X$ and $r \in [0, 1]$ we have $p(x) = \bigvee_n p_n(x) > r$ if and only if there is $n$ with $p_n(x) > r$. Thus $p^{-1}(r, 1]) = \bigcup_n p_n^{-1}(r, 1])$. Since each $p_n^{-1}(r, 1])$ is measurable, so is $p^{-1}(r, 1])$, and the join $p = \bigvee_n p_n$ is measurable.

We thus get a functor $\text{Meas}(-, [0, 1]): \text{Meas} \to \omega\text{-EMod}^{\text{op}}$. The $\omega$-effect module structure is preserved by pre-composition, since it is defined pointwise.

The set $\text{Top}(Y, [0, 1])$ of continuous functions $Y \to [0, 1]$ is an effect module, but in general has no $\omega$-joins. Take for instance $Y = [0, 2]$, and consider the continuous functions $f_1 \leq f_2 \leq \cdots \leq f: [0, 2] \to [0, 1]$ defined by:

$$
 f_n(y) = \begin{cases} 
 1 - y^n & \text{if } y \in [0, 1) \\
 0 & \text{if } y \in [1, 2] 
\end{cases}
$$

and

$$
 f(y) = \begin{cases} 
 1 & \text{if } y \in [0, 1) \\
 0 & \text{if } x \in [1, 2] 
\end{cases}
$$

Since $\lim_{n \to \infty} y^n = 0$ for $y \in [0, 1)$ we see that $f$ is the pointwise join of $f_1, f_2, \ldots$. Clearly, this join $f$ is not continuous, and so it cannot be the join of $f_1, f_2, \ldots$ in $\text{Top}(Y, [0, 1])$. Even more: we claim there is no least continuous function above $f$. Thus $f_1, f_2, \ldots$ has no join at all in $\text{Top}(Y, [0, 1])$.

Indeed, if $g: [0, 2] \to [0, 1]$ is continuous and $g \geq f$, then $g^2 \geq f^2 = f$ as well (where $f^2 = f$ because $f$ is $\{0, 1\}$-valued). On the other hand $g$ is not $\{0, 1\}$-valued because $g$ is continuous at $1$. Thus $g^2 < g$. Hence $g$ is not the least continuous function above $f$, and thus $g$ is not the join of $f_1, f_2, \ldots$. \qed

For each measurable space $(X, \Sigma)$ there is the ‘indicator’ function $1_{(-)}: \Sigma \to \text{Meas}(X, [0, 1])$, given by $1_M(x) = 1$ if $x \in M$ and $1_M(x) = 0$ if $x \notin M$ where $M \in \Sigma_X$. Then $1_{(-)}$ is a homomorphism of $\omega$-effect algebras.
The next result neatly organises the situation so far. It turns out that this situation has an additional freeness property that is the essence of Lebesgue integration. This will be elaborated in the next section (see Theorem 4.12).

**Lemma 3.2** Sending a measurable subset $M$ to its indicator function $1_M$ is a natural transformation in:

$$\omega\text{-EA}^{\text{op}} \xrightarrow{\Sigma(-)} \text{U Meas}(-, [0,1])$$

where $U: \omega\text{-EMod} \to \omega\text{-EA}$ is the forgetful functor. The (possibly unexpected) direction of the arrow $\leftarrow$ is explained by the $(-)^{\text{op}}$.

**Proof** Let $(X, \Sigma_X)$ be a measurable space. We show that the mapping $M \mapsto 1_M$ is a homomorphism of $\omega$-effect algebras $1(-): \Sigma_X \to \text{Meas}(X, [0,1])$, and leave naturality to the reader. Clearly, the unit is preserved, since $1_X$ is the constant function $x \mapsto 1$. Also, if $M \perp M'$ in $\Sigma_X$, that is, $M \cap M' = \emptyset$, then $1_{M \cup M'} = 1_M + 1_{M'} = 1_M \otimes 1_{M'}$. It is easy to see that $\omega$-joins are preserved: $\bigvee_n 1_{M_n} = 1_{\bigcup_n M_n}$. □

**Lemma 3.3** Hom-ing into $[0,1]$ yields an adjunction between $\omega$-effect modules and measurable spaces:

$$\omega\text{-EMod}^{\text{op}} \xleftarrow{\text{Hom}(-, [0,1])} \text{Meas}$$

**Proof** In order to do this, we first need to provide the homset $\omega\text{-EMod}(E, [0,1])$ with a $\sigma$-algebra. We take the least $\sigma$-algebra that makes for each $e \in E$ the evaluation map $\text{ev}_e: \omega\text{-EMod}(E, [0,1]) \to [0,1]$, given by $\text{ev}_e(\omega) = \omega(e)$, measurable. This is functorial, since for $f: E \to D$ in $\omega\text{-EMod}$, the map $(-) \circ f: \omega\text{-EMod}(D, [0,1]) \to \omega\text{-EMod}(E, [0,1])$ is measurable.

We get an adjunction since there is a natural bijective correspondence:

$$E \xrightarrow{f} \text{Meas}(X, [0,1]) \quad \text{in } \omega\text{-EMod}$$

$$X \xleftarrow{g} \omega\text{-EMod}(E, [0,1]) \quad \text{in } \text{Meas}$$

This is done via a simple swapping of arguments. □

Later on, in Corollary 4.14, we shall see that the monad on the category $\text{Meas}$ induced by this adjunction is the well-known Giry monad [9].

### 4 Lebesgue integration in $\omega$-effect modules

Our approach to integration is on the one hand more restricted than usual, and on the other hand more general. The restriction lies in the fact that we define integration for $[0,1]$-valued functions, and not for more general functions. The extension involves using probability measures $\phi: \Sigma \to E$ into an $\omega$-effect module $E$, instead of into $[0,1]$ as is commonly done.
Traditionally, a measure space consists of a measurable space \((X, \Sigma_X)\) with a function \(\phi: \Sigma_X \to [0, \infty]\) which satisfies \(\phi(\emptyset) = 0\) and is countably additive:

\[
\phi \left( \biguplus_{n \in \mathbb{N}} M_n \right) = \sum_{n \in \mathbb{N}} \phi(M_n) = \bigvee_{n \in \mathbb{N}} \sum_{i \leq n} \phi(M_i),
\]

for each pairwise disjoint, countable collection of measurable \(M_n \in \Sigma_X\). Here we use \(\biguplus\) for disjoint union, where \(\Sigma_X\) is understood as an effect algebra. Such a measure \(\phi\) is called a probability measure if \(\phi(X) = 1\), so that \(\phi\) can be restricted to a function \(\Sigma_X \to [0, 1]\).

Below is a well-known observation (see e.g. [21, Thm. 4.4]) that justifies our generalisation of probability measures to other codomains than \([0, 1]\).

**Lemma 4.1** Let \(X\) be a measurable space, with a function \(\phi: \Sigma_X \to [0, 1]\). The following points are then equivalent:

(i) \(\phi\) is a probability measure, that is, \(\phi(\emptyset) = 0\) and \(\phi(X) = 1\) and \(\phi\) is countably additive as in \((2)\);

(ii) \(\phi\) is a homomorphism of \(\omega\)-effect algebras \(\Sigma_X \to [0, 1]\). \(\square\)

**Definition 4.2** Let \(X\) be a measurable space, and \(E\) a \(\omega\)-effect module. An \(E\)-valued probability measure, or simply an \(E\)-probability measure is a map \(\phi: \Sigma_X \to U(E)\) in the category \(\omega\-EMod\) of \(\omega\)-effect algebras — where \(U: \omega\-EMod \to \omega\-EA\) is the forgetful functor.

For each element \(x \in X\) we write \(\eta(x): \Sigma_X \to E\) for the probability measure given by \(\eta(x)(M) = 1\) if \(x \in M\) and \(\eta(x)(M) = 0\) if \(x \notin M\).

Examples of probability measures with values in an \(\omega\)-effect module are POVMs: Positive Operator-Valued Measures, see e.g. [13, Defn. 3.5]. Such a POVM is a map of \(\omega\)-effect algebras \(\Sigma_X \to \mathcal{E}(\mathcal{H})\) with the effects of a Hilbert space \(\mathcal{H}\) as codomain. We will return to POVMs in Example 4.15 below.

**Remark 4.3** While \(\Sigma_X\) and \([0, 1]\) are MV-algebras (see [4]), a probability measure \(\phi: \Sigma_X \to [0, 1]\) need not be an homomorphism of MV-algebras, that is, preserve binary joins \(\lor\).

Indeed, since in an MV-algebra we have the identity \(x \lor y = x + (y^\perp + x)^\perp\) a homomorphism of MV-algebras preserves finite joins. (In fact, a homomorphism of effect algebras between MV-algebras is a homomorphism of MV-algebras precisely when it preserves finite joins.) The standard probability measure \(\mu\) on \([0, 1]\) does not preserve finite joins \(\mu([0, \frac{1}{2}]) \cup [\frac{1}{2}, 1]) = \frac{1}{2} \neq 1 = \max \{\mu([0, \frac{1}{2}]), \mu([\frac{1}{2}, 1])\}\) and is thus not a homomorphism of MV-algebras.

The probability measures \(\phi: \Sigma_X \to [0, 1]\) which preserve joins are in fact quite special. Indeed, for such \(\phi\) we have \(\phi(A \cup B) = \max\{\phi(A), \phi(B)\}\) for all \(A, B \in \Sigma_X\), and also \(\phi(A \cap B) = \min\{\phi(A), \phi(B)\}\). Taking \(B = A^\perp\), we see that either \(\phi(A) = 1\) (and \(\phi(A^\perp) = 0\)) or \(\phi(A^\perp) = 1\) (and \(\phi(A) = 0\)). Thus \(\{A \in \Sigma_X: \phi(A) = 1\}\) is an ultrafilter on \(\Sigma_X\).

Extending measure to integral is done in two parts, first for step functions, and then for all measurable functions, as joins of \(\omega\)-chains of step functions.

**Definition 4.4** Let \(X\) be a measurable space.
(i) A step function \( x \rightarrow [0,1] \) is a function that can be written as finite linear combination \( r_1 \cdot 1_{M_1} + \cdots + r_k \cdot 1_{M_k} = \bigotimes_i r_i \cdot 1_{M_i} \in \text{Meas}(X, [0,1]) \) of indicator functions \( 1_{M_i} \) and scalars \( r_i \in [0,1] \), where the \( M_i \in \Sigma_X \) are pairwise disjoint measurable subsets satisfying \( \bigotimes_i M_i = X \).

(ii) Let \( \phi : \Sigma_X \rightarrow E \) be a probability measure. The interpretation of \( \int s \, d\phi \) for a step function \( s = \bigotimes_i r_i 1_{M_i} \) is

\[
\int s \, d\phi = \bigotimes_i r_i \cdot \phi(M_i) \in E.
\]  

(There is no ambiguity, see Lemma 4.5 below.)

Since these \( M_i \) form a partition, they are \( k \)-test in the effect algebra \( \Sigma_X \). Also, the set of step functions can be described as tensor product \( \Sigma_X \otimes U(E) \), where \( \otimes \) is the tensor of effect algebras, see [15].

In the second point we use the property that in an effect module \( x \perp y \) implies \( r \cdot x \perp t \cdot y \) for all scalars \( r, t \in [0,1] \), see Lemma 2.1.

We will first show that the integral \( \int s \, d\phi \) in (3) is independent of the representation of the step function \( s \), see e.g. [21, Lemma 9.1]. We elaborate the details in order to show that this works in effect modules too.

**Lemma 4.5** Let \( X \) be a measurable space, and \( \phi : \Sigma_X \rightarrow E \) a probability measure. Consider two step functions \( \bigotimes_i r_i \cdot 1_{M_i} \leq \bigotimes_j s_j \cdot 1_{N_j} \in \text{Meas}(X, [0,1]) \). Then \( \bigotimes_i r_i \cdot \phi(M_i) \leq \bigotimes_j s_j \cdot \phi(N_j) \) in \( E \).

**Proof** Since \( \bigotimes_i M_i = X = \bigotimes_j N_j \) by Definition 4.4 we have \( M_i = \bigotimes_j M_i \cap N_j \) and \( N_j = \bigotimes_i N_j \cap M_i \). Thus:

\[
\sum_i r_i \phi(M_i) = \sum_i r_i \phi( \bigotimes_j M_i \cap N_j ) = \sum_{i,j} r_i \phi(M_i \cap N_j) \\
\leq \sum_{i,j} s_j \phi(M_i \cap N_j) \quad \text{see below}
\]

\[
= \sum_j s_j \phi( \bigotimes_i M_i \cap N_j ) = \sum_j s_j \phi(N_j). 
\]

We used the fact that \( r_i \phi(M_i \cap N_j) \leq s_j \phi(M_i \cap N_j) \) for all \( i \) and \( j \). Indeed, this inequality holds when \( M_i \cap N_j = \emptyset \). Otherwise, we have for \( x \in M_i \cap N_j \),

\[
r_i = \bigotimes_j r_i 1_{M_i}(x) \leq \bigotimes_j s_j 1_{N_j}(x) = s_j.
\]

Thus \( r_i \phi(M_i \cap N_j) \leq s_j \phi(M_i \cap N_j) \). \( \square \)

A basic observation is that each measurable predicate can be described as join of an ascending \( \omega \)-chain of step functions (see e.g. [21, Thm. 8.8]).

**Lemma 4.6** For each measurable function \( p : X \rightarrow [0,1] \) there is an \( \omega \)-chain \( s_1 \leq s_2 \leq \cdots \) of step functions \( s_n \leq p \) with \( p = \bigvee s_n \).

**Proof**

Lemma 4.6 is the key to the meaning of \( \int p \, d\phi \) when \( p \) is an arbitrary measurable function in \( \text{Meas}(X, [0,1]) \). Indeed, we shall have \( \int p = \bigvee_n \int s_n \) when \( s_1 \leq s_2 \leq \cdots \) are step functions with \( \bigvee_n s_n = p \). However, before we can cast this observation into a definition we must check that there is no ambiguity by proving that \( \bigvee_n \int s_n = \int p = \bigvee s_n \).
\[ \bigvee_n \int t_n \text{ whenever } t_1 \leq t_2 \leq \cdots \text{ and } s_1 \leq s_2 \leq \cdots \text{ are step functions with } \bigvee_n s_n = \bigvee_n t_n. \] This fact will follow from a far more general statement (see Proposition 4.8) about the following notion.

**Definition 4.7** Let \( \phi \) be an \( E \)-valued probability measure on a measurable space \( X \). An elementary extension of \( \phi \) is a map \( \Phi: S \to E \) defined on a collection of measurable functions \( S \subseteq \text{Meas}(X, [0, 1]) \) such that:

(i) \( 1_M \in S \) and \( \Phi(1_M) = \phi(M) \) for all \( M \in \Sigma_X \);
(ii) \( S \) is a sub-effect module of \( \text{Meas}(X, [0, 1]) \), and \( \Phi: S \to \text{Meas}(X, [0, 1]) \) is a homomorphism of effect modules.
(iii) \( s \cdot 1_M \in S \) for all \( M \in \Sigma_X \) and \( s \in [0, 1] \).

The integral \( \int (\cdot) \, d\phi \), defined on the sub-effect module of step functions is an elementary extension of \( \phi \). But also integration on all measurable maps will be an elementary extension. This abstraction allows us to apply the following result both to integration of step functions and of all measurable functions.

**Proposition 4.8** Let \( \Phi: S \to E \) be an elementary extension of an \( E \)-valued probability measure \( \phi \) on a measurable space \( X \).

(i) Let \( s \) and \( t_1 \leq t_2 \leq \cdots \) be from \( S \) with \( s \leq \bigvee t_n \). Then \( \Phi(s) \leq \bigvee_n \Phi(t_n) \).
(ii) Let \( s_1 \leq s_2 \leq \cdots \) and \( t_1 \leq t_2 \leq \cdots \) be from \( S \). Then \( \bigvee s_n \leq \bigvee t_n \) implies \( \bigvee \Phi(s_n) \leq \Phi(t_n) \).

**Proof** We will only prove point (i) since point (ii) is an easy consequence.

Writing \( a_m = 1 - \frac{1}{m} \in [0, 1] \) for \( m \geq 1 \) we have \( \bigvee_m a_m = 1 \). Thus to prove \( \Phi(s) \leq \bigvee_n \Phi(t_n) \) it suffices to show that \( a_m \cdot \Phi(s) \leq \bigvee_n \Phi(t_n) \) for all \( m \). Since then \( \Phi(s) = 1 \cdot \Phi(s) = (\bigvee_m a_m) \cdot \Phi(s) = \bigvee_m a_m \cdot \Phi(s) \leq \bigvee_n \Phi(t_n) \).

Let \( m \) be given. The trick is to consider the sets

\[ M_n = \{ x \in X \mid a_m \cdot s(x) \leq t_n(x) \}. \]

It is not difficult to prove that: (1) each subset \( M_n \subseteq X \) is measurable (since \( s, t_1, t_2, \ldots \) are measurable functions); that (2) the \( M_n \) form an ascending chain with \( \bigcup_n M_n = X \) (since \( a_m \cdot s(x) < s(x) \leq \bigvee t_n(x) \) for each \( x \in X \)); and that (3) \( a_m \cdot (s \cdot 1_{M_n}) \leq t_n \) for all \( n \). The latter implies \( a_m \cdot \bigvee_n \Phi(s \cdot 1_{M_n}) \leq \bigvee_n \Phi(t_n) \) in \( E \). So it suffices to prove that \( \Phi(s) = \bigvee_n \Phi(s \cdot 1_{M_n}) \), or in other words, \( \bigwedge_n \Phi(s \cdot 1_{M_n}) = 0 \). Since \( s \cdot 1_{-M_n} \leq 1_{-M_n} \), for all \( n \) we have \( \bigwedge_n \Phi(s \cdot 1_{-M_n}) \leq \bigwedge_n \Phi(1_{-M_n}) = \bigwedge_n \phi(-M_n) = \bigwedge_n 1 - \phi(M_n) = 1 - \bigvee_n \phi(M_n) = 1 - \phi(\bigvee_n M_n) = 1 - \phi(X) = 1 - 1 = 0 \).

We can now define the (Lebesgue) integral taking its values in an \( \omega \)-effect algebra, for measurable predicates.

**Definition 4.9** Let \( \phi \) be an \( E \)-valued probability measure on a measurable space \( X \). For measurable function \( p: X \to [0, 1] \) we define the integral by

\[
\int p \, d\phi = \bigvee_n \int s_n \, d\phi \in E, \tag{4}
\]
where \( s_1 \leq s_2 \leq \cdots \) is a chain of step functions with \( \bigvee_n s_n = p \). Such a chain exists by Lemma 4.6 and there is no ambiguity by Proposition 4.8.

We list some basic well-known properties of integration, formulated here in effect-theoretic terms.

**Proposition 4.10** Let \( X \) be a measurable space, together with a probability measure \( \phi : \Sigma_X \to E \) in an \( \omega \)-effect module \( E \).

(i) \( \int (-) \, d\phi \) on \( \text{Meas}(X, [0,1]) \) is an elementary extension of \( \phi \). In particular, sending \( p \mapsto \int p \, d\phi \) yields a homomorphism of effect modules \( \text{Meas}(X, [0,1]) \to E \).

(ii) (‘Levi’s Theorem’) For all \( p_1 \leq p_2 \leq \cdots \) in \( \text{Meas}(X, [0,1]) \),

\[
\int \bigvee_n p_n \, d\phi = \bigvee_n \int p_n \, d\phi.
\]

The latter two points say that \( \int (-) \, d\phi \) is a morphism \( \text{Meas}(X, [0,1]) \to E \) in the category \( \omega \text{-EMod} \) of \( \omega \)-effect modules.

(iii) For maps \( f : X \to Y \) in \( \text{Meas} \) and \( g : E \to D \) in \( \omega \text{-EMod} \),

\[
\int (g \circ f) \, d\phi = \int g \, d(\phi \circ f^{-1}) = g \left( \int p \, d\phi \right) = \int p \, d(U(g) \circ \phi),
\]

where \( U : \omega \text{-EMod} \to \omega \text{-EA} \) is the forgetful functor.

(iv) For each \( x \in X \) and \( p \in \text{Meas}(X, [0,1]) \) one has:

\[
\int p \, d\eta(x) = p(x),
\]

where \( \eta(x) : \Sigma_X \to E \) is as described in Definition 4.2.

**Proof** (i) We only show that \( \int (-) \, d\phi \) is a homomorphism of effect modules. The other requirements for \( \int (-) \, d\phi \) to be an elementary extension of \( \phi \) (see Definition 4.7) are either trivial to verify or follow immediately from the fact that the integral on step functions is an elementary extension of \( \phi \).

Since \( 1_X \) is a step function, and \( \int (-) \, d\phi \) extends the integral on step functions, and we already know that that the integral on step functions is a homomorphism of effect modules, we get \( \int 1_X \, d\phi = 1 \).

Let \( p, q \in \text{Meas}(X, [0,1]) \) with \( p \perp q \). We must show that \( \int p \otimes q \, d\phi = \int p \, d\phi \otimes \int q \, d\phi \). By Lemma 4.6 there are step functions \( s_0 \leq s_1 \leq \cdots \) and \( t_1 \leq t_2 \leq \cdots \) such that \( p = \bigvee s_n \) and \( q = \bigvee t_n \). Then \( \int p \, d\phi = \bigvee_n \int s_n \, d\phi \) and \( \int q \, d\phi = \bigvee_n \int t_n \, d\phi \) by Definition 4.9. Then \( s_n \perp t_n \) for all \( n \) and \( p \otimes q = \bigvee_n s_n \otimes t_n \) so \( \int p \otimes q \, d\phi = \bigvee_n \int s_n \otimes t_n \, d\phi \). Since \( s_n \) and \( t_n \) are step functions, we already know that \( \int s_n \otimes t_n \, d\phi = \int s_n \, d\phi \otimes \int t_n \, d\phi \). Thus,

\[
\int p \, d\phi \otimes \int q \, d\phi = \left( \bigvee_n \int s_n \, d\phi \right) \otimes \left( \bigvee_n \int t_n \, d\phi \right) = \bigvee_n \left( \int s_n \, d\phi \otimes \int t_n \, d\phi \right) = \bigvee_n \int (s_n \otimes t_n) \, d\phi = \int p \otimes q \, d\phi.
\]
By a similar reasoning using that scalar multiplication preserves suprema of \(\omega\)-joins we get \(\int r \cdot p \, d\phi = r \cdot \int p \, d\phi\) for all \(p \in \text{Meas}(X, [0, 1])\), \(r \in [0, 1]\). Thus \(\int (\cdot) \, d\phi\) is a homomorphism of effect modules.

(ii) This is a consequence of Proposition 4.8 since \(\int (\cdot) \, d\phi\) is an elementary extension of \(\phi\).

(iii) For measurable \(f : X \to Y\) and step function \(s = \biguplus_i r_i 1_{N_i}\) in \(\text{Meas}(Y, [0, 1])\) we have:

\[
\int (s \circ f) \, d\phi = \int \left( \biguplus_i r_i 1_{f^{-1}(N_i)} \right) \, d\phi \quad \text{by naturality of } 1_{(-)}, \text{ see Lemma 3.2}
\]

\[
= \biguplus_i r_i \cdot \phi(f^{-1}(N_i))
\]

\[
= \int \left( \biguplus_i r_i 1_{N_i} \right) \, d(\phi \circ f^{-1})
\]

\[
= \int s \, d(\phi \circ f^{-1}).
\]

The required result for an arbitrary predicate \(p \in \text{Meas}(X, [0, 1])\) now follows like in (i) using that \((\cdot) \circ f\) preserves suprema of \(\omega\)-chains.

The second equation is also first obtained for step functions.

(iv) For a step function \(s = \biguplus_i r_i 1_{M_i}\) we have:

\[
\int s \, d\eta(x) = \biguplus_i r_i \eta(x)(M_i) = \sum_i r_i 1_{M_i}(x) = s(x).
\]

Hence for a join \(p = \bigvee_n s_n\) of step functions \(s_n\) we get:

\[
\int p \, d\eta(x) = \bigvee_n \int s_n \, d\eta(x) = \bigvee_n s_n(x) = p(x).
\]

\(\square\)

Remark 4.11 Let \(\phi : \Sigma_X \to E\) be a probability measure on a measurable space \(X\) where \(E\) is an \(\omega\)-effect module. Many optional features of \(\phi\) carry over to the integral \(\overline{\phi} = \int (\cdot) \, d\phi\). We give two examples.

(i) If \(E\) is an MV-algebra and \(\phi\) is a homomorphism of MV-algebras, then \(\overline{\phi} = \int (\cdot) \, d\phi : \text{Meas}(\Sigma_X, [0, 1]) \to E\) is a homomorphism of MV-modules. We sketch a proof, but leave the details to the reader.

Note the \(\overline{\phi}\) is a homomorphism of MV-algebras iff it preserves binary meets. Given a step function \(s = \biguplus_i s_i 1_{M_i}\) the sets \(M_i\) are pairwise disjoint and so \(s = \bigvee_i s_i 1_{M_i}\). Thus, by distributivity of \(\wedge\) over \(\vee\), we see that for step functions \(s = \biguplus_i s_i 1_{M_i}\) and \(t = \biguplus_j t_j 1_{N_j}\) we have \(s \wedge t = \bigvee_{i,j} (s_i \wedge t_j) 1_{M_i \cap N_j}\). To see that the integral on step functions preserves binary meets, integrate, use the facts that \(\phi\) preserves binary meets, and rewrite.

To see that \(\int (\cdot) \, d\phi\) preserves binary meets, note that for step functions \(s_1 \leq s_2 \leq \cdots\) and \(t_1 \leq t_2 \leq \cdots\) we have \((\bigvee_n s_n) \wedge (\bigvee_m t_m) = \bigvee_n s_n \wedge t_n\).

Now, integrate, use that the integral on step functions preserves binary meets, rewrite, and finish the proof with an appeal to Lemma 4.6.

(ii) If \(E\) is endowed with a suitable product \(\odot\) and \(\phi\) preserves ‘products’ (that is, \(\phi(A \cap B) = \phi(A) \odot \phi(B)\) for all \(A, B \in \Sigma_X\)), then \(\overline{\phi} = \int (\cdot) \, d\phi\) preserves products. We leave the proof to the reader.

By a suitable product, we mean an associative map \(\odot : E \times E \to E\) such that for every \(a \in E\) the maps \(a \odot (\cdot)\) and \((\cdot) \odot a\) preserve sum \(\oplus\), scalar multiplication, and countable joins, and \(1 \odot a = a = a \odot 1\).
The natural transformation \( 1 \): \( \Sigma(-) \Rightarrow U(M) \) is universal, in the following sense. For each functor \( F: \text{Meas} \rightarrow \omega\text{-EMod} \) with a natural transformation \( \tau: \Sigma(-) \Rightarrow U \), there is a unique \( F(-) \) \( \in \omega\text{-EMod} \), such that \( F(1(-)) = \tau \), in:

\[
\begin{array}{ccc}
\Sigma(-) & \xrightarrow{1(-)} & U(M) \\
\tau & \downarrow & \downarrow \phi \\
\text{Meas}(-, [0,1]) & \xrightarrow{\tau} & E
\end{array}
\]

(iii) The natural transformation \( 1(-): \Sigma(-) \Rightarrow U\text{Meas}(-, [0,1]) \) from Lemma 3.2 is universal, in the following sense. For each functor \( F: \text{Meas} \rightarrow \omega\text{-EMod} \) with a natural transformation \( \tau: \Sigma(-) \Rightarrow U \), there is a unique \( \tau: \text{Meas}(-, [0,1]) \Rightarrow F \) with \( U\tau \circ 1(-) = \tau \), in:

\[
\begin{array}{ccc}
\Sigma(-) & \xrightarrow{1(-)} & U\text{Meas}(-, [0,1]) \\
\tau & \downarrow & \downarrow \phi \\
\text{Meas}(-, [0,1]) & \xrightarrow{U\tau} & F
\end{array}
\]

Proof (i) Take \( \phi = \int (-) \phi \); it is a homomorphism of \( \omega\)-effect modules with \( \int M \phi = \phi(M) \) for all \( M \in \Sigma \) by Proposition 4.10.

For uniqueness, let \( \xi: \text{Meas}([0,1], [0,1]) \rightarrow E \) be a homomorphism of \( \omega\)-effect modules such that \( \xi(1_M) = \phi(M) \) for all \( M \in \Sigma \). Let \( p \in \text{Meas}([0,1], [0,1]) \) be given. We must show that \( \xi(p) = \int p \phi \).

We prove this first for step functions. If \( p = \bigotimes_i r_i 1_{M_i} \) then we have that \( \xi(p) = \bigotimes_i r_i \xi(1_{M_i}) = \bigotimes_i r_i \phi(M_i) = \int p \phi \).

Now, for arbitrary \( p \) there are step functions \( s_1 \leq s_2 \leq \cdots \) with \( p = \bigvee_n s_n \) by Lemma 4.6 and so we have \( \xi(p) = \bigvee_n \xi(s_n) = \bigvee_n \int s_n \phi = \int p \phi \).

(ii) This follows categorically from point (i), see [18, Thm. IV.1.2 (ii)].

(iii) Define \( \tau_X = \tau_X \), given by \( \tau_X(p) = \int p \tau_X \) as in point (i). \( \square \)

Remark 4.13 Theorem 4.12 says that for a measurable space \((X, \Sigma_X)\), the predicates \( \text{Meas}(X, [0,1]) \) form the free \( \omega\)-effect module on the \( \omega\)-effect algebra \( \Sigma_X \). In
essence, for $[0,1]$-valued functions, Lebesgue integration is thus the extension of a map of $\omega$-effect algebras $\Sigma_X \rightarrow E$ to a map of $\omega$-effect modules $\text{Meas}(X, [0,1]) \rightarrow E$.

Such free modules are obtained by tensoring $[0,1] \otimes (-)$ with the scalars involved. This description is used by Gudder in [11, Theorem 6.8]. He proves an isomorphism $[0,1] \otimes \Sigma_X \cong \text{Meas}(X, [0,1])$, which implies that the predicates $\text{Meas}(X, [0,1])$ form the free $\omega$-effect module on $\Sigma_X$. There are a few more things to say.

- Gudder does not use $\omega$-effect modules, only $\omega$-effect algebras. He proves that there is a suitable bihomomorphism of $\omega$-effect algebras $[0,1] \times \Sigma_X \rightarrow \text{Meas}(X, [0,1])$. He does not prove the existence of tensors of $\omega$-effect algebras in general. He only shows the existence of this particular one in $[0,1] \otimes \Sigma_X \cong \text{Meas}(X, [0,1])$.
- Gudder does not use categorical language, and so the formulation of integration as free construction (as in Theorem 4.12) does not occur in [11].

The most common instance of the $\omega$-effect module $E$ in Theorem 4.12 uses $E = [0,1]$. But as we shall see later, we can also use the effects of a Hilbert space or of a von Neumann algebra. Thus, the generality of using $E$-valued probability measures $\Sigma_X \rightarrow E$ pays off.

**Corollary 4.14** The monad $\omega$-$\text{EMod}(\text{Meas}(X, [0,1]), [0,1])$ on the category $\text{Meas}$ of measurable spaces induced by the adjunction $\omega$-$\text{EMod}^{\text{op}} \dashv \text{Meas}$ from Lemma 3.3 is (isomorphic to) the Giry monad $\mathcal{G}$ (from [9]).

**Proof** Since by Theorem 4.12, with $E = [0,1]$ we have:

$$\mathcal{G}(X) \overset{\text{def}}{=} \omega\text{-EA}(\Sigma_X, [0,1]) \cong \omega\text{-EMod}(\text{Meas}(X, [0,1]), [0,1]).$$

\[\square\]

**Example 4.15** One application of the general mechanism of Theorem 4.12 is the formulation of the spectral theorem for effects on a Hilbert space $\mathcal{H}$. Recall that a bounded self-adjoint linear map $A$ on $\mathcal{H}$ is called an effect when $0 \leq \langle Ax, x \rangle \leq 1$ for all $x \in \mathcal{H}$. These effects form an $\omega$-effect module, which we denote by $\mathcal{E}(\mathcal{H})$.

Let $A \in \mathcal{E}(\mathcal{H})$ be an effect. The spectrum $\sigma_A$ of $A$ (i.e. all $\lambda \in \mathbb{C}$ such that $A - \lambda \cdot 1$ is not invertible) inherits the topology of $\mathbb{C}$. Since $A$ is an effect, we get $\sigma_A \subseteq [0,1]$. Endow $\sigma_A$ with the $\sigma$-algebra of Borel measurable subsets of $\mathbb{C}$, so that $\sigma_A$ becomes a measurable space.

Recall that an $\omega$-effect algebra homomorphism $\phi: \Sigma_{\sigma_A} \rightarrow \mathcal{E}(\mathcal{H})$ is called a POVM (positive operator valued measure). We are interested in POVMs $\phi: \Sigma_{\sigma_A} \rightarrow \mathcal{E}(\mathcal{H})$ such that $\phi(M)$ is a projection for all $M \in \Sigma_{\sigma_A}$. Such a $\phi$ is called a spectral measure on $\sigma_A$, and by Theorem 4.12, it has a unique extension to an $\omega$-effect module homomorphism $\int (-) \, d\phi: \text{Meas}(\sigma_A, [0,1]) \rightarrow \mathcal{E}(\mathcal{H})$.

(Note that while $\phi$ is projection-valued the integral $\int (-) \, d\phi$ is not: $\int 1/2 \, d\phi = 1/2$ is not a projection. Also, the set of projections does not form an $\omega$-effect module.)

The spectral theorem states that there is a unique spectral measure $\phi$ on $\sigma_A$ which satisfies the following requirements (see [12, §43 and §39]).

(i) $A = \int id \, d\phi$ where $id: \sigma_A \rightarrow [0,1]$ is given by $id(x) = x$ for $x \in \sigma_A$. This means that the effect $A$ has a ‘spectral decomposition’ as an integral over projections.

(ii) For any open subset $G$ of $\sigma_A$ with $\phi(G) = 0$ we have $G = \emptyset$.  

140
In fact, we may replace the latter requirement by the following weaker form.

(ii') The complement of $\bigcup \{ G \mid G \subseteq [0,1] \text{ open and } \phi(G) = 0 \}$ in $[0,1]$ is compact. Moreover, such a spectral measure $\phi$ has the following properties.

(iii) $(\int f \, d\phi) \cdot (\int g \, d\phi) = \int f \cdot g \, d\phi$ for all $f, g \in \text{Meas}(\sigma_A, [0,1])$, see [12, §37, Thm. 3].

(iv) Let $B$ be a bounded linear operator on $H$; then $B$ commutes with $A$ if and only if $B$ commutes with $\phi(M) \in \mathcal{E}(H)$ for all $M \in \Sigma_{\sigma_A}$, via a combination of Theorem 2 from §41 and Theorem 4 from §37 of [12].

The spectral theorem is one of the great achievements of 20th century mathematics. It reveals that effects behave somewhat like measurable functions to $[0,1]$; the integral $\int (\cdot) \, d\phi$ provides the translation from measurable functions to effects.

5 Perspectives and future work

By Theorem 4.12 an $E$-probability measure can be extended to an integral. But how does one obtain an $E$-probability measure? Carathéodory’s extension theorem guarantees that given a measurable space $(X, \Sigma_X)$ any homomorphism of effect algebras $\mu : S \rightarrow [0,1]$ defined on a Boolean subalgebra $S$ of a $\sigma$-algebra $\Sigma_X$ on a set $X$ can be extended uniquely to a probability measure $\tilde{\mu} : \Sigma_X \rightarrow [0,1]$ provided that $\mu(\bigcup_n A_n) = \bigvee_n \mu(A_n)$ for all $A_1 \subseteq A_2 \subseteq \cdots$ from $\Sigma_X$ with $\bigcup_n A_n \in \Sigma_X$.

We do not know if a similar theorem holds for $E$-valued homomorphisms $\mu$ where $E$ is an arbitrary $\omega$-effect module. Our attempts to generalise existing proofs are blocked by the potential lack of a complete metric on $E$, which leads us to the following problem.

**Problem 5.1** Let $E$ be an Archimedean $\omega$-effect module. Is the metric on $E$ complete? (See [16], Equation (10) for the definition of the metric on $E$.)

Other questions remain: for example, can we fit Fubini (integration over product spaces) in our general framework?

Of the numerous generalisations of the formal definition of integral given by Riemann our work is perhaps most similar in setup and breadth to the vector valued variations on the Lebesgue integral studied by Bochner [2] and Pettis [20]. Their integrals takes values from a Banach space while our integral takes values from an $\omega$-effect module. They exploit the uniform structure on a Banach space, while we use the order structure of an $\omega$-effect module. An order-theoretic approach to integration has also been considered by Alfsen (for real-valued lattice valuations, see [1]), the second author (for lattice valuations taking their values from a suitable lattice-ordered abelian group, see [23]), and others [5,22].

Traditionally, countable chains take centre stage in the theory of measure and integral as opposed to the directed sets of domain theory. To see why, note that the Lebesgue measure on $[0,1]$ does not preserve joins of directed sets as any (measurable) set is the union of the directed set of its finite (and thus negligible) subsets. Nevertheless, there are connections between integration and domain theory. For example, the measurable subsets on $[0,1]$ modulo negligibility form a complete lat-
Jacobs and Westerbaan

tice, and the real-valued Riemann integration of continuous functions on a compact metric space can be related to the probabilistic power domain (see [7]).

References


Conditioning in Probabilistic Programming

Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Federico Olmedo
RWTH Aachen University, Aachen, Germany

Friedrich Gretz, Annabelle McIver
Macquarie University, Sydney, Australia

Abstract
In this paper, we investigate the semantic intricacies of conditioning in probabilistic programming, a major feature, e.g., in machine learning. We provide a quantitative weakest pre-condition semantics. In contrast to all other approaches, non-termination is taken into account by our semantics. We also present an operational semantics in terms of Markov models and show that expected rewards coincide with quantitative pre-conditions. A program transformation that entirely eliminates conditioning from programs is given; the correctness is shown using our semantics. Finally, we show that an inductive semantics for conditioning in non-deterministic probabilistic programs cannot exist.

Keywords: Probabilistic Programming, Semantics, Conditional Probabilities, Program Transformation

1 Introduction
In recent years, interest in probabilistic programming has rapidly grown [9,11]. This is due to its wide applicability, for example in machine learning for describing distribution functions; Bayesian inference is pivotal in their analysis. It is used in security for describing both cryptographic constructions such as randomized encryption and experiments defining security properties [4]. Probabilistic programs, being extensions of familiar notions, render these fields accessible to programming communities. A rich palette of probabilistic programming languages exists including Church [8] as well as modern approaches like probabilistic C [23], Tabular [10] and R2 [22].

Probabilistic programs are sequential programs having two main features: (1) the ability to draw values at random from probability distributions, and (2) the ability to condition the value of variables in a program through so-called observations. The semantics of languages without conditioning is well-understood: In his seminal work, Kozen [19] considered denotational semantics for probabilistic

* This work was supported by the Excellence Initiative of the German federal and state government.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
programs without non–determinism or observations. One of these semantics—the expectation transformer semantics—was adopted by McIver and Morgan [21], who added support for non–determinism; a corresponding operational semantics is given in [13]. Other relevant works include probabilistic power–domains [17], semantics of constraint probabilistic programming languages [15,14], and semantics for stochastic λ–calculi [26].

Semantic intricacies. The difficulties that arise when program variables are conditioned through observations is less well–understood. This gap is filled in this paper. Previous work on semantics for programs with observe statements [22,16] do neither consider the possibility of non–termination nor the powerful feature of non–determinism. In contrast, we thoroughly study a more general setting which accounts for non–termination by means of a very simple yet powerful probabilistic programming language supporting non–determinism and observations. Let us first analyze a few examples illustrating the different problems. We start with the problem of non–termination; consider the two program snippets

\[ x := 2 \quad \text{and} \quad \{ x := 2 \} \left[ \frac{1}{2} \right] \{ \text{abort} \} . \]

The program on the left just assigns the value 2 to the program variable \( x \), while the program on the right tosses a fair coin—which is modeled through a probabilistic choice—and depending on the outcome either performs the same variable assignment or diverges due to the abort instruction. The semantics given in [22,16] does not distinguish these two programs and is only sensible in the context of terminating programs. A programmer writing only terminating programs is already unrealistic in the non–probabilistic setting. Our semantics does not rely on the assumption that programs always terminate and is able to distinguish these two programs.

To discuss observations, consider the program snippet \( P_{obs1} \)

\[ \{ x := 0 \} \left[ \frac{1}{2} \right] \{ x := 1 \} : \text{observe} \ (x=1), \]

which assigns zero to the variable \( x \) with probability \( \frac{1}{2} \) while \( x \) is assigned one with the same likelihood, after which we condition to the outcome of \( x \) being one. The observe statement blocks all invalid runs violating its condition and renormalizes the probabilities of the remaining valid runs. This differs, e.g., from program annotations like (probabilistic) assertions [25] as we will see later. The interpretation of the program is the expected outcome conditioned on the valid runs. For \( P_{obs1} \), this yields the outcome \( 1 \cdot 1 \)—there is one valid run that happens with probability one, with \( x \) being one.

More involved problems arise when programs are infeasible meaning all runs are blocked. Consider a slight variant of the program above, called \( P_{obs2} \):

\[ \{ x := 0; \ \text{observe} \ (x=1) \} \left[ \frac{1}{2} \right] \{ x := 1; \ \text{observe} \ (x=1) \} \]

The left branch of the probabilistic choice is infeasible. Is this program equivalent to the sample program \( P_{obs1} \)? It will turn out that this is the case, meaning that setting an infeasible program into context can render it feasible.

The situation becomes more complicated when considering loopy programs that
may diverge. Consider the following two programs:

\begin{align*}
P_{\text{div}} : & \quad x := 1; \text{ while } (x=1) \{ x := 1 \} \\
P_{\text{andiv}} : & \quad x := 1; \text{ while } (x=1) \{ \{ x := 1 \} \{ x := 0 \}; \text{ observe } (x=1) \}
\end{align*}

Program \( P_{\text{div}} \) diverges as \( x \) is set to one in every iteration. This yields a null expected outcome. Due to the conditioning on \( x=1 \), \( P_{\text{andiv}} \) has just a single (valid)—non–terminating—run, but this run almost surely never happens, i.e. it happens with probability zero. The conditional expected outcome of \( P_{\text{andiv}} \) can thus not be measured. Our semantics can distinguish these programs while programs with (probabilistic) assertions must be loop–free to avoid similar problems [25]. Other approaches insist on the absence of diverging loops [5]. Neither of these assumptions are realistic.

Non–determinism is a powerful means to deal with unknown information, as well as to specify abstractions in situations where implementation details are unimportant. This feature turns out to be intricate in combination with conditioning.\(^1\) Consider the program \( P_{\text{nondet}} \)

\[
\{ \{ x := 5 \} \oplus \{ x := 2 \} \} \{ 1/4 \} \{ x := 2 \}; \text{ observe } (x>3),
\]

where with probability \( 1/4 \), \( x \) is set either to 5 or to 2 non–deterministically (denoted \( \{ x := 5 \} \oplus \{ x := 2 \} \)), while \( x \) is set to 2 with likelihood \( 3/4 \). Resolving the non–deterministic choice in favor of setting \( x \) to five yields a conditional expectation of 5 for \( x \), obtained as \( 5 \cdot \frac{1}{4} \) rescaled over the single valid run of \( P_{\text{nondet}} \). Taking the right branch however induces two invalid runs due to the violation of the condition \( x>3 \), yielding a non–measurable conditional outcome.

Contributions. The above issues—non–termination, loops, and non–determinism—indicate that conditioning in probabilistic programs is far from trivial. This paper presents a thorough semantic treatment of conditioning in a probabilistic extension of Dijkstra’s guarded command language (known as \( \text{pGCL} \) [21]), an elementary though foundational language that includes (amongst others) parametric probabilistic choice. We take several semantic viewpoints.

We first provide a conditional version of a weakest pre–condition (wp) semantics à la [21]. This is typically defined inductively over the structure of the program. We show that combining both non–determinism and conditioning cannot be treated in this manner. To treat possibly non–terminating programs, due to e.g., diverging loops or abortion, this is complemented by a weakest liberal pre–condition (wlp) semantics. Moreover, our \( w(l)p \) semantics is backward compatible with the original \( \text{pGCL} \) semantics for programs without conditioning; this does not apply to alternative approaches such as \( \text{R2} \) [22].

Furthermore, Markov Decision Processes (MDPs) [24] are used as the basis for an operational semantics. This semantics is simple and elegant while covering all aforementioned phenomena, including non–determinism. We show that conditional

\footnote{As stated in [11], “representing and inferring sets of distributions is more complicated than dealing with a single distribution, and hence there are several technical challenges in adding non–determinism to probabilistic programs”.

---

145
expected rewards in the MDP–semantics correspond to (conditional) wp in the de- 
notational semantics, extending a similar result for pGCL [13].

Finally, we present a program transformation which entirely eliminates condi-
tioning from any program and prove its correctness using our semantics.

Summarized, after introducing pGCL (Section 2), we give a denotational seman-
tics for fully probabilistic programs (Section 3). We provide the first operational se-
mantics for imperative probabilistic programming languages with conditioning and 
both probabilistic and non–deterministic choice (Section 4). Our semantics enables 
us to prove the correctness of a program transformation that eliminates observe 
statements (Section 5). Finally, we show that it is not possible to provide an inductive 
semantics for programs that include both conditioning and non–determinism 
(Section 6).

An extended version of this paper including all proofs and further program 
transformations for eliminating observe statements is available in [12].

2 The Programming Language

In this section we briefly present the probabilistic programming language used for 
our development. The language is an extension of the probabilistic guarded command 
language (pGCL) of McIver and Morgan [21]. The original pGCL is given by syntax

\[
P ::= \text{skip} \mid \text{abort} \mid x := E \mid \{P\} \mid \text{ite}(G)\{P\} \{P\} \\
\quad \mid \{P\} \{p\} \{P\} \mid \{P\} \Box \{Q\} \mid \text{while}(G)\{P\}
\]

and constitutes a plain extension of Dijkstra’s guarded command language (GCL) [7] 
with a binary probabilistic choice operator. Here, \(x\) belongs to \(V\), the set of program 
variables; \(E\) is an arithmetical expression over \(V\); \(G\) a Boolean expression over \(V\); 
and \(p\) a real–valued parameter with domain \([0,1]\). Most of the pGCL instructions 
are self–explanatory; we elaborate only on the following: \(\{P\} \{p\} \{Q\}\) is a proba-
bilistic choice where program \(P\) is executed with probability \(p\) and program \(Q\) with 
probability \(1–p\); \(\{P\} \Box \{Q\}\) is a non–deterministic choice between \(P\) and \(Q\); finally 
\text{abort} is syntactic sugar for the diverging program \(\text{while}(\text{true})\{\text{skip}\}\).

To model probabilistic programs with conditioning we extend pGCL with obser-
vations, leading to the conditional pGCL (cpGCL). At the syntactic level, an 
observation is introduced with the instruction \text{observe} \((G)\), \(G\) being a Boolean expression over \(V\). The effect of such an instruction is to block all invalid program 
executions violating \(G\) and rescale the probability of the remaining executions so 
that they sum up to one.

As an illustrative example consider the following pair of programs:

\[
P_1: \{x := 0\} \{p\} \{x := 1\}; \{y := 0\} \{q\} \{y := -1\}
\]

\[
P_2: \{x := 0\} \{p\} \{x := 1\}; \{y := 0\} \{q\} \{y := -1\}; \text{observe} \ (x+y=0)
\]

Program \(P_1\) admits all (four) runs, two of which satisfy \(x=0\); for this program 
the probability of \(x=0\) is \(p\). Program \(P_2\)—due to the \text{observe} statement requiring 
\(x+y=0\)—admits only two runs, only one of them satisfying \(x=0\); for this program
the probability of $x=0$ is $\frac{pq}{pq+(1-p)(1-q)}$.

Note that there exists a connection between the observe statement used in our work and the well-known assert statement. Both statements observe ($G$) and assert ($G$) block all runs violating $G$. The crucial difference, however, is that observe ($G$) normalizes the probability of the unblocked runs while assert ($G$) does not, yielding then a sub-distribution of total mass possibly less than one [20,4].

3 Denotational Semantics for Conditional pGCL

In this section we recall the expectation transformer semantics of pGCL and extend it to conditional programs in the fully probabilistic fragment of cpGCL.

3.1 Expectation Transformers in pGCL

Expectation transformers are a quantitative version of predicate transformers [7] used to endow probabilistic pGCL programs a formal semantics. Loosely speaking, they capture the average or expected outcome of a program, measured w.r.t. a utility or reward function over the set of final states. To make this more precise, let $S$ be the set of program states, where a program state is a variable valuation. Now assume that $P$ is a fully probabilistic program, i.e. a program without non-deterministic choices. Intuitively, we can think of $P$ as a mapping from an initial state $\sigma \in S$ to a distribution of final states $J_P(\sigma)$; its formal semantics is captured by a transformer $wp[P]$, which acts as follows: Given a random variable $f: S \rightarrow \mathbb{R}_{\geq 0}$, $wp[P](f)$ maps every initial state $\sigma$ to the expected value $E_{J_P(\sigma)}(f)$ with respect to the distribution of final states $J_P(\sigma)$. Symbolically,

$$wp[P](f)(\sigma) = E_{J_P(\sigma)}(f).$$

In particular, if $f = \chi_A$ is the characteristic function of some event $A$, $wp[P](f)$ retrieves the probability that the event occurred after the execution of $P$. (Moreover, if $P$ is a deterministic program in GCL, $E_{J_P(\sigma)}(\chi_A)$ is $\{0, 1\}$–valued and we recover the ordinary notion of predicate transformers introduced by Dijkstra [7].)

For a program $P$ including non-deterministic choices, the execution of $P$ yields a set of final distributions. To account for this, we assume that $wp[P](f)$ gives the tightest lower bound $\inf_{\mu \in J_P(\sigma)} E_{\mu}(f)$ for the expected value of $f$. This corresponds with the notion of a demonic adversary resolving the non-deterministic choices.

We follow McIver and Morgan [21] and use the term expectation to refer to a random variable mapping program states to real values. The expectation transformer $wp$ then transforms a post–expectation $f$ into a pre–expectation $wp[P](f)$ and can be defined by induction on the structure of $P$, following the rules in Figure 1. The transformer $wp$ also admits a liberal variant $wlp$, which differs from $wp$ in the way in which non–termination is treated.

Formally, the transformer $wp$ operates on unbounded expectations in $E = S \rightarrow \mathbb{R}_{\geq 0}$ and $wlp$ operates on bounded expectations in $E_{\leq 1} = S \rightarrow [0, 1]$. Here $\mathbb{R}_{\geq 0}^\infty$ denotes the set of non–negative real values with the adjoined $\infty$ value. In order to guarantee the well–definedness of $wp$ and $wlp$ we need to provide $E$ and $E_{\leq 1}$ the structure of a directed–complete partial order. Expectations are ordered pointwise,
i.e. \( f \subseteq g \) iff \( f(\sigma) \leq g(\sigma) \) for every state \( \sigma \in S \). The least upper bound of directed subsets is also defined pointwise.

In the remainder we make use of the following notation related to expectations. We use bold fonts for constant expectations, e.g. \( 1 \) denotes the constant expectation 1. Given an arithmetical expression \( E \) over program variables we simply write \( E \) for the expectation that in state \( \sigma \) returns \( E(\sigma) \). Given a Boolean expression \( G \) over program variables we use \( \chi_G \) to denote the \( \{0, 1\} \)-valued expectation that returns 1 if \( \sigma \models G \) and 0 otherwise.

### 3.2 Conditional Expectation Transformers

We now study how to extend the notion of expectation transformers to conditioned probabilistic programs without non–determinism in \( \text{cpGCL} \). To illustrate the intuition behind our solution, consider the following scenario: Assume we want to measure the probability that some event \( A \) occurs after the execution of a conditioned program \( P \). Since \( P \) contains observations, its execution leads to a conditional distribution \( \mu_{|O} \) of final states. Now the conditional probability that \( A \) occurs (given that \( O \) occurs) is given as the quotient of the probabilities \( \Pr[\mu \in A \land O] \) and \( \Pr[\mu \in O] \). Motivated by this observation, we introduce an expectation transformer \( \text{cwp}[\cdot] : \mathbb{E} \times \mathbb{E}_{\leq 1} \rightarrow \mathbb{E} \times \mathbb{E}_{\leq 1} \), whose application \( \text{cwp}[P](\chi_A, 1) \) will yield the desired pair of probabilities \( \langle \Pr[\mu \in A \land O], \Pr[\mu \in O] \rangle \). We are only left to define a transformer \( \text{cwp}[P] \) that computes the corresponding quotient. Formally, we let

\[
\text{cwp}[P](f) \triangleq \frac{\text{cwp}_1[P](f, 1)}{\text{cwp}_2[P](f, 1)},
\]

where \( \text{cwp}_1[P](f, g) \) (resp. \( \text{cwp}_2[P](f, g) \)) denotes the first (resp. second) component of \( \text{cwp}[P](f, g) \). If \( \text{cwp}_2[P](f, 1)(\sigma) = 0 \), then \( \text{cwp}[P](f) \) is not well–defined in \( \sigma \) (in the same way as the conditional probability \( \Pr(A|B) \) is not well–defined when \( \Pr(B) = 0 \)) and we say that program \( P \) is infeasible from state \( \sigma \), meaning that all its executions are blocked by observations.

As so defined, \( \text{cwp}[P](f) \) represents the weakest conditional pre–expectation of \( P \) with respect to post–expectation \( f \) and \( \text{cwp}[\cdot] \) generalizes the transformer \( \wp[\cdot] \) to conditioned programs. The weakest liberal conditional pre–expectation \( \text{cwplp}[P](f) \) is defined analogously, in terms of the transformer \( \text{cwplp}[P] : \mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1} \rightarrow \mathbb{E}_{\leq 1} \times \mathbb{E}_{\leq 1} \).

We are only left to provide definitions for \( \text{cwp}[P] \) and \( \text{cwplp}[P] \). Both transformers are defined by induction on the structure of \( P \), following the rules in Figure 1. Let us briefly explain these rules. \( \text{cwp}[\text{skip}] \) behaves as the identity since \( \text{skip} \) has no effect. \( \text{cwp}[\text{abort}] \) maps any pair of post–expectations to the pair of constant pre–expectations \( (0, 1) \). Assignments induce a substitution on expectations, i.e. \( \text{cwp}[x := E] \) maps \( (f, g) \) to pre–expectation \( (f[x/E], g[x/E]) \), where \( h[x/E](\sigma) = h(\sigma[x/E]) \) and \( \sigma[x/E] \) denotes the usual variable update on states. \( \text{cwp}[P_1; P_2] \) is obtained as the functional composition (denoted \( \circ \)) of \( \text{cwp}[P_1] \) and \( \text{cwp}[P_2] \). \( \text{cwp}[\text{observe } (G)] \) restricts post–expectations to those states that satisfy \( G \); states that do not satisfy \( G \) are mapped to 0. \( \text{cwp}[\text{ite } (G) \{P_1\} \{P_2\}] \) behaves

\[\footnote{In the continuous setting we could define a conditional density even when conditioning on events with 0 measure using the Radon–Nikodym theorem. However, our programs generate discrete distributions only.} \]
The computation of \(cwp\)\{(P_1 \cup P_2)\}(f, g)

\[
\begin{array}{|c|c|}
\hline
P & wp[P](f) & cwp[P](f, g) \\
\hline
skip & f & (f, g) \\
abort & 0 & (0, 1) \\
x := E & f[x/E] & (f[x/E], g[x/E]) \\
observe (G) & – not defined – & \chi_G \cdot (f, g) \\
\{P_1\} \cup \{P_2\} & (wp[P_1] \circ wp[P_2])(f) & (cwp[P_1] \circ cwp[P_2])(f, g) \\
\text{ite}(G)\{(P_1)\} & \chi_G \cdot wp[P_1](f) + \chi_{\neg G} \cdot wp[P_2](f) & \chi_G \cdot cwp[P_1](f, g) + \chi_{\neg G} \cdot cwp[P_2](f, g) \\
\{P_1\} \cup \{P_2\} & p \cdot wp[P_1](f) + (1 - p) \cdot wp[P_2](f) & p \cdot cwp[P_1](f, g) + (1 - p) \cdot cwp[P_2](f, g) \\
\{P_1\} \cup \{P_2\} & \lambda \sigma, \min(wp[P_1](f/\sigma), wp[P_2](f/\sigma)) & \text{not defined} \\
\text{while}(G)\{P\} & \mu f, \left(\chi_G \cdot wp[P](f) + \chi_{\neg G} \cdot f\right) & \mu_{\leq}(f, g) \cdot \left(\chi_G \cdot cwp[P](f, g) + \chi_{\neg G} \cdot (f, g)\right) \\
\hline
\end{array}
\]

Fig. 1. Definitions for the \(wp/wlp\) and \(cwp/cwlp\) operators. The \(wp\) (\(cwlp\)) operator differs from \(wp\) (\(cwlp\)) only for \(abort\) and the \(while\)-loop. Multiplication \(h \cdot (f, g)\) is meant componentwise yielding \((h \cdot f, h \cdot g)\). Likewise, addition \((f, g) + (f', g')\) is meant componentwise yielding \((f + f', g + g')\).

either as \(cwp[P_1]\) or \(cwp[P_2]\) according to the evaluation of \(G\). \(cwp[\{P_1\} \cup \{P_2\}]\) is obtained as a convex combination of \(cwp[P_1]\) and \(cwp[P_2]\), weighted according to \(p\). \(cwp[\text{while}(G)\{P\}]\) is defined using standard fixed point techniques.\(^3\) The \(cwlp\) transformer follows the same rules as \(cwp\), except for the \(abort\) and \(while\) statements. \(cwlp[\text{abort}]\) takes any post-translation to pre-translation \((1, 1)\); \(cwlp[\text{while}(G)\{P\}]\) is defined in terms of a greatest rather than a least fixed point.

Observe that Figure 1 presents no rule for the non-deterministic choice operator. Therefore our conditional expectation transformers \(cwp/cwlp\) can only handle fully probabilistic cpGCL programs. In Section 6 we elaborate on this limitation.

**Example 3.1** Assume we want to compute the expected value of the expression 10\(+x\) after executing program \(P'\) given as:

1. \(\{x := 0\} \{1/2\} \{x := 1\};\)
2. \(\text{ite}(x = 1) \{\{y := 0\} \{1/2\} \{y := 2\}\} \{\{y := 0\} \{4/5\} \{y := 3\}\};\)
3. \(\text{observe} \ (y=0)\)

The computation of \(cwp[P'](10+x, 1)\) goes as follows:

\[
cwp[P'](10+x, 1) = cwp[P'_{1,2}](cwp[\text{observe} \ (y=0)](10+x, 1))
= cwp[P'_{1,2}](f, g) \text{ where } (f, g) = \chi_{y=0} \cdot (10+x, 1)
= cwp[P'_{1,2}](cwp[\text{ite}(x=1) \{\ldots\} \{\ldots\}](f, g))
= cwp[P'_{1,2}](\chi_{x=1} \cdot (h, i) + \chi_{x \neq 1} \cdot (h', i') \text{ where}
(\ h, i \) = cwp[\{\(y := 0\) \{1/2\} \{y := 2\}\}(f, g) = \frac{1}{2} \cdot (10 + x, 1), \text{ and}
(\ h', i' \) = cwp[\{\(y := 0\) \{4/5\} \{y := 3\}\}(f, g) = \frac{4}{5} \cdot (10 + x, 1)
= \frac{1}{2} \cdot \frac{4}{5} \cdot (10 + 0, 1) + \frac{1}{2} \cdot \frac{1}{2} \cdot (10 + 1, 1) = (\frac{27}{4}, \frac{13}{20})\).
\]

The expected value of 10\(+x\) is then given by \(cwp[P'](10+x) = \frac{27}{4} / \frac{13}{20} = \frac{135}{13} \approx 10.38\).

\(^3\) We define \(cwlp[\text{while}(G)\{P\}]\) as the least fixed point w.r.t. the order \(\subseteq\) in \(E \times E_{\leq}\). This way we encode the greatest fixed point in the second component w.r.t. the order \(\leq\) over \(E_{\leq}\) as the least fixed point w.r.t. the dual order \(\supseteq\).
In the rest of this section we investigate some properties of the expectation transformer semantics (of the fully probabilistic fragment) of \( \text{cpGCL} \). As every fully probabilistic \( \text{pGCL} \) program is contained in \( \text{cpGCL} \), we begin by studying the relation between the \( \text{w(l)p} \)-semantics of \( \text{pGCL} \) and the \( \text{cw(l)p} \)-semantics of \( \text{cpGCL} \). To that end, we extend the \( \text{w(l)p} \) operator to \( \text{cpGCL} \) by the clauses \( \text{wp[observe } (G)) (f) = \chi_{G} \cdot f \) and \( \text{wlp[observe } (G)) (f) = \chi_{G} \cdot f \). Our first result says that \( \text{cw} \) (resp. \( \text{cwlp} \)) can be decoupled as the product \( \text{wp} \times \text{wlp} \) (resp. \( \text{wlp} \times \text{wlp} \)).

**Lemma 3.2 (Decoupling of \( \text{cw(l)p} \))** Let \( P \) be a fully probabilistic \( \text{cpGCL} \) program, \( f \in \mathbb{E} \) and \( f' \), \( g \in \mathbb{E}_{\leq 1} \). Then \( \text{cw}[P](f, g) = (\text{wp}[P](f), \text{wlp}[P](g)) \) and \( \text{cwlp}[P](f', g) = (\text{wlp}[P](f'), \text{wlp}[P](g)) \).

Our next result shows that the \( \text{cw} \)-semantics is a conservative extension of the \( \text{wp} \)-semantics for the fully probabilistic fragment of \( \text{pGCL} \). The same applies to the weakest liberal pre–expectation semantics.

**Theorem 3.3 (Compatibility with the \( \text{w(l)p} \)-semantics)** Let \( P \) be a fully probabilistic \( \text{pGCL} \) program, \( f \in \mathbb{E} \), and \( g \in \mathbb{E}_{\leq 1} \). Then \( \text{wp}[P](f) = \text{cw}[P](f) \) and \( \text{wlp}[P](g) = \text{cwlp}[P](g) \).

**Proof.** By Lemma 3.2 and the fact that \( \text{wlp}[P](1) = 1 \) (see Lemma 3.4). \( \square \)

We now show that \( \text{cw} \) and \( \text{cwlp} \) preserve the so–called healthiness conditions of \( \text{wp} \) and \( \text{wlp} \).

**Lemma 3.4 (Healthiness conditions of \( \text{cw} \) and \( \text{cwlp} \))** For every fully probabilistic \( \text{cpGCL} \) program \( P \) with at least one feasible execution (from every initial state), every \( f, g \in \mathbb{E} \) and non–negative real constants \( \alpha, \beta \):

i) \( f \sqsubseteq g \) implies \( \text{cw}[P](f) \sqsubseteq \text{cw}[P](g) \) and likewise for \( \text{cwlp} \) (monotonicity).

ii) \( \text{cw}[P](\alpha \cdot f + \beta \cdot g) = \alpha \cdot \text{cw}[P](f) + \beta \cdot \text{cw}[P](g) \) (linearity).

iii) \( \text{cw}[P](0) = 0 \) and \( \text{cwlp}[P](1) = 1 \).

**Proof.** Using Lemma 3.2 one can show that the transformers \( \text{cw} \) and \( \text{cwlp} \) inherit these properties from \( \text{wp} \) and \( \text{wlp} \). For details see [12, p. 15]. \( \square \)

We conclude this section by discussing alternative approaches for providing an expectation transformer semantics for \( P \in \text{cpGCL} \). By Lemma 3.2, the transformers \( \text{cw}[P] \) and \( \text{cwlp}[P] \) can be recast as

\[
\begin{align*}
    f & \mapsto \frac{\text{wp}[P](f)}{\text{wlp}[P](1)} \quad \text{and} \quad f \mapsto \frac{\text{wlp}[P](f)}{\text{wlp}[P](1)},
\end{align*}
\]

respectively. An alternative is to normalize using \( \text{wp} \) instead of \( \text{wlp} \) in the denominator, yielding the two transformers

\[
\begin{align*}
    i) \ f & \mapsto \frac{\text{wp}[P](f)}{\text{wp}[P](1)} \quad \text{and} \quad ii) \ f \mapsto \frac{\text{wlp}[P](f)}{\text{wp}[P](1)}.
\end{align*}
\]

Transformer ii) is not meaningful, as the denominator \( \text{wp}[P](1)(\sigma) \) may be smaller than the numerator \( \text{wlp}[P](f)(\sigma) \) for some state \( \sigma \in S \). This might lead to probabilities exceeding one. Transformer i) normalizes w.r.t. the terminating executions.
This interpretation corresponds to the semantics of the probabilistic programming language R2 [22,16] and is only meaningful if programs terminate almost surely (i.e. with probability one). A noteworthy consequence of adopting transformer $i)$ is that observe ($G$) is equivalent to while ($¬G$) {skip} [16], see the discussion in Section 5.

Let us briefly compare the four alternatives by means of a concrete program $P$:

\[
\{ \text{abort} \} \left( \frac{1}{2} \right) \{ x := 0 \} \left( \frac{1}{2} \right) \{ x := 1 \}; \{ y := 0 \} \left( \frac{1}{2} \right) \{ y := 1 \}; \text{observe} (x=0 \lor y=0) \]

$P$ tosses a fair coin and according to the outcome either diverges or tosses a fair coin twice and observes at least once heads ($y=0 \lor x=0$). We measure the probability that the outcome of the last coin toss was heads according to each transformer:

\[
\begin{align*}
\wp[P](x=0) &= \frac{2}{7} & \wp[P](x=0) &= 6 \quad \frac{2}{7} & \wp[P](x=0) &= \frac{2}{3} \quad \wp[P](x=0) &= 2
\end{align*}
\]

As mentioned before, the transformer $ii)$ is not significant as it yields a “probability” exceeding one. Note that our w2p–semantics yields that the probability of $y=0$ after the execution of $P$ while passing all observe–statements is $\frac{2}{7}$. As shown before, this is a conservative and natural extension of the wp–semantics. This does not apply to the R2–semantics, as this would require an adaptation of rules for abort and while.

4 Operational Semantics for Conditional pGCL

This section presents an operational semantics for pGCL using Markov decision processes (MDPs) as underlying model. We begin by recalling the notion of MDPs. For that, let $\text{Distr}(S)$ denote the set of distributions $\mu : S \to \mathbb{R}$ over $S$ with $\sum_{s \in S} \mu(s) = 1$.

**Definition 4.1** An MDP is a tuple $\mathfrak{M} = (S, s_I, \text{Act}, \mathcal{P}, L)$ with a countable set of states $S$, an initial state $s_I \in S$, a finite set of actions $\text{Act}$, a transition probability function $\mathcal{P} : S \times \text{Act} \to \text{Distr}(S)$ with $\sum_{s' \in S} \mathcal{P}(s, \alpha)(s') = 1$ for all $(s, \alpha) \in S \times \text{Act}$ and a labeling function $L : S \to 2^{\text{Act}}$ for a set of atomic propositions $\text{AP}$.

A function $r : S \to \mathbb{R}_{\geq 0}$ is used to add rewards to an MDP. A path of $\mathfrak{M}$ is a finite or infinite sequence $\pi = s_0 a_0 s_1 a_1 \ldots$ such that $s_i \in S$, $a_i \in \text{Act}$, $s_0 = s_I$, and $\mathcal{P}(s_i, a_i)(s_{i+1}) > 0$ for all $i \geq 0$. The $i$-th state $s_i$ of $\pi$ is denoted $\pi(i)$. The set of all paths of $\mathfrak{M}$ is denoted by $\text{Paths}^{\mathfrak{M}}$. $\text{Paths}^{\mathfrak{M}}(s, s')$ is the set of all finite paths starting in $s$ and ending in $s'$. This is also lifted to sets of states. We sometimes omit superscript $\mathfrak{M}$ in $\text{Paths}^{\mathfrak{M}}$.

MDPs operate by a non–deterministic choice of an action $\alpha \in \text{Act}$ that is enabled at state $s$ and a subsequent probabilistic determination of a successor state according to $\mathcal{P}(s, \alpha)$. For resolving the non–deterministic choices, so-called schedulers are used. Here, deterministic and memoryless schedulers suffice which are functions $\mathcal{S} : S \to \text{Act}$. Let $\text{Sched}^{\mathfrak{M}}$ denote the class of all such schedulers for $\mathfrak{M}$.

For MDP $\mathfrak{M}$, the fully probabilistic system $\mathcal{E}\mathfrak{M}$ induced by a scheduler $\mathcal{S} \in \text{Sched}^{\mathfrak{M}}$ is called the induced Markov Chain (MC) on which a probability measure over paths is defined. The measure $\Pr^\mathfrak{M}$ for MC $\mathfrak{M}$ is given by $\Pr^\mathfrak{M} : \text{Paths}^\mathfrak{M} \to [0, 1] \subseteq \mathbb{R}$ with $\Pr^\mathfrak{M}(\pi) = \prod_{i=0}^{n-1} \mathcal{P}(s_i, s_{i+1})$, for a finite path $\pi = s_0 \ldots s_n$. This is
lifted to infinite paths using the standard cylinder set construction, see [2, Ch. 10].

The cumulated reward of a finite path \( \vec{s} = s_0 \ldots s_n \) is given by \( r(\vec{s}) = \sum_{i=0}^{n-1} r(s_i) \).
Note that in our special setting the cumulated reward will not be infinite.

We consider reachability properties \( \Diamond T \) for a set of target states \( T \subseteq S \) where \( \Diamond T \) also denotes all paths that reach \( T \) from the initial state \( s_I \). Analogously, the set \( \neg \Diamond T \) contains all paths that never reach a state in \( T \).

First, consider reward objectives for MCs. The expected reward for a countable set of paths \( \Diamond T \) is given by \( \text{ExpRew}^R(\Diamond T) = \sum_{\vec{s} \in \Diamond T} \text{Pr}^R(\vec{s}) \cdot r(\vec{s}) \). For a reward bounded by one, the notion of the liberal expected reward also takes the mere probability of not reaching the target states into account: \( \text{LExpRew}^R(\Diamond T) = \text{ExpRew}^R(\Diamond T) + \text{Pr}^R(\neg \Diamond T) \). To exclude the probability of paths that reach “undesired” states, we let \( U = \{ s \in S | \not\in L(s) \} \) and define the conditional expected reward for the condition \( \Diamond U \) by \(^4\)

\[
\text{CExpRew}^R(\Diamond T \mid \Diamond U) \triangleq \frac{\text{ExpRew}^R(\Diamond T \cap \Diamond U)}{\text{Pr}^R(\neg \Diamond U)}.
\]

Reward objectives for MDPs are now defined using a demonic scheduler \( S \in \text{Sched}^\mathbb{R} \) minimizing probabilities and expected rewards for the induced MDP \( \mathcal{G}^S \). For the expected reward this yields \( \text{ExpRew}^\mathbb{R}(\Diamond T) = \inf_{S \in \text{Sched}^\mathbb{R}} \text{ExpRew}^S(\Diamond T) \). For conditional expected reward properties, the value of the quotient is minimized:

\[
\text{CExpRew}^\mathbb{R}(\Diamond T \mid \Diamond U) \triangleq \inf_{S \in \text{Sched}^\mathbb{R}} \frac{\text{ExpRew}^S(\Diamond T \cap \Diamond U)}{\text{Pr}^S(\neg \Diamond U)}.
\]

The liberal reward notions for MDPs are analogous. Regarding the quotient minimization we assume “\( \frac{0}{0} \) is undefined” as we see \( \frac{0}{0} \) being undefined—to be less favorable than \( 0 \). For details about conditional probabilities and expected rewards see [3].

The structure of the operational MDP of a cpGCL program is depicted on the right. Terminating runs eventually end up in the \( \langle \text{sink} \rangle \) state; other runs are diverging (never reach \( \langle \text{sink} \rangle \)). A program terminates either successfully, i.e. a run passes a \( \checkmark \) -labelled state, or terminates due to a violation of an observation, i.e. a run passes \( \langle \text{diverge} \rangle \). Squiggly arrows indicate reaching certain states via possibly multiple paths and states; the clouds indicate that there might be several states of the particular kind. The \( \checkmark \) -labelled states are the only ones with positive reward. Note that the sets of paths that eventually reach \( \langle z \rangle \), or \( \checkmark \), or diverge are pairwise disjoint.

**Definition 4.2** [Operational cpGCL semantics] The operational semantics of \( P \in \text{cpGCL} \) for \( \sigma \in S \) and \( f \in E \) is the MDP \( \mathcal{G}^\sigma_f[P] = (S, \langle P, \sigma \rangle, \text{Act}, P, L, r) \), such that \( S \) is the smallest set of states with \( \langle z \rangle \in S \), \( \langle \text{sink} \rangle \in S \), and \( \langle Q, \tau \rangle, \langle \downarrow, \tau \rangle \in S \)

\(^4\) Note that strictly formal one would have to define the intersection of sets of finite and possibly infinite paths by means of a cylinder set construction considering all infinite extensions of finite paths.
for $Q \in \mathit{pGCL}$ and $\tau \in S$. $(P, \sigma) \in S$ is the initial state. $Act = \{\text{left}, \text{right}\}$ is the set of actions. A state of the form $(\downarrow, \tau)$ denotes a terminal state in which no program is left to be executed. $P$ is formed according to SOS rules given in [12, p. 5].

For some $\tau \in S$, the labelling and the reward function is given by:

$$L(s) \triangleq \begin{cases} \{\checkmark\}, & \text{if } s = (\downarrow, \tau) \\ \{\text{sink}\}, & \text{if } s = (\text{sink}) \\ \{\downarrow\}, & \text{if } s = (\downarrow) \\ \emptyset, & \text{otherwise} \end{cases} \quad r(s) \triangleq \begin{cases} f(\tau), & \text{if } s = (\downarrow, \tau) \\ 0, & \text{otherwise} \end{cases}$$

To determine the conditional expected outcome of program $P$ given that all observations are true, we need to determine the expected reward to reach $(\text{sink})$ from the initial state conditioned on not reaching $(\text{.down\_right})$ under a demonic scheduler. For $\mathcal{R}_x^P$ this is given by $\text{CExpRew}^{\mathcal{R}_x^P}((\text{sink}) | (\downarrow) )$. Recall for the condition (\text{down\_right} that all paths not eventually reaching $(\text{down\_right})$ either diverge (thus collect reward 0) or pass by a $\checkmark$–labelled state and eventually reach $(\text{sink})$. This gives us:

$$\text{CExpRew}^{\mathcal{R}_x^P}((\text{sink}) | (\downarrow) ) = \inf_{\mathcal{S} \in \text{Sched}^{\mathcal{R}_x^P}} \frac{\text{ExpRew}^{\mathcal{R}_x^P}((\text{sink}) \cap (\downarrow))}{\Pr^{\mathcal{R}_x^P}(\downarrow)}$$

The liberal version $\text{CLEExpRew}^{\mathcal{R}_x^P}((\text{sink}) | (\downarrow) )$ is defined analogously.

**Example 4.3** Consider the program $P \in \mathit{cpGCL}$:

$$\{\{x := 5\} \square \{x := 2\}\} [q] \{x := 2\} ; \text{observe } (x > 3)$$

where with parametrized probability $q$ a non–deterministic choice between $x$ being assigned 2 or 5 is executed, and with probability $1 - q$, $x$ is directly assigned 2. Let for readability $P_1 = \{x := 5\} \square \{x := 2\}$, $P_2 = x := 2$, $P_3 = \text{observe } (x > 3)$, and $P_4 = x := 5$. The operational MDP $\mathcal{R}_{\sigma_I}^P$ for an arbitrary initial variable valuation $\sigma_I$ and post–expectation $x$ is depicted below:
The only state with positive reward is \( s' := (\downarrow, \sigma_I[x/5]) \) and its reward is indicated by number 5. Assume first a scheduler choosing action left in state \((P_1; P_3, \sigma_I)\). In the induced MC the only path accumulating positive reward is the path \( \pi \) going from \((P, \sigma_I)\) via \( s' \) to \( \langle \text{sink} \rangle \) with \( r(\pi) = 5 \) and \( \Pr(\pi) = q \). This gives an expected reward of \( 5 \cdot q \). The overall probability of not reaching \( \langle \text{☇} \rangle \) is also \( q \). The conditional expected reward of eventually reaching \( \langle \text{sink} \rangle \) given that \( \langle \text{☇} \rangle \) is not reached is hence \( \frac{5 \cdot q}{q} = 5 \).

Assume now the demonic scheduler choosing right at state \((P_1; P_3, \sigma_I)\). In this case there is no path having positive accumulated reward in the induced MC, yielding an expected reward of 0. The probability of not reaching \( \langle \text{☇} \rangle \) is also 0. The conditional expected reward in this case is undefined \((0/0)\) and thus the right branch is preferred over the left branch. In general, the operational MDP need not be finite, even if the program terminates almost–surely (i.e. with probability 1).

We now investigate the connection to the denotational semantics of Section 3, starting with some auxiliary results. First, we establish a relation between (liberal) expected rewards and weakest (liberal) pre–expectations.

**Lemma 4.4** For any fully probabilistic \( P \in \text{cpGCL} \), \( f \in E \), \( g \in E_{\leq 1} \), and \( \sigma \in S \):

\[
\text{ExpRew}^R_{\downarrow} [P] (\Diamond \langle \text{sink} \rangle) = \text{wp}[P](f)(\sigma) \quad \text{(i)}
\]
\[
\text{LExpRew}^R_{\downarrow} [P] (\Diamond \langle \text{sink} \rangle) = \text{wlp}[P](g)(\sigma) \quad \text{(ii)}
\]

Moreover, the probability of never reaching \( \langle \text{☇} \rangle \) in the MC of program \( P \) coincides with the weakest liberal pre–expectation of \( P \) w.r.t. post–expectation 1:

**Lemma 4.5** For any fully probabilistic \( P \in \text{cpGCL} \), \( g \in E_{\leq 1} \), and \( \sigma \in S \) we have \( \Pr^R_{\downarrow} [P] (\neg \Diamond \text{☇}) = \text{wlp}[P](1)(\sigma) \).

We now have all prerequisites in order to present the main result of this section: the correspondence between the operational and expectation transformer semantics of fully probabilistic \( \text{cpGCL} \) programs. It turns out that the weakest (liberal) pre–expectation \( \text{cwlp}[P](f)(\sigma) \) (resp. \( \text{cwlp}[P](f)(\sigma) \)) coincides with the conditional (liberal) expected reward in the RMC \( R^R_{\downarrow} [P] \) of terminating while never violating an observe–statement, i.e., avoiding the \( \langle \text{☇} \rangle \) states.

**Theorem 4.6** (Correspondence theorem) For any fully probabilistic \( P \in \text{cpGCL} \), \( f \in E \), \( g \in E_{\leq 1} \), and \( \sigma \in S \),

\[
\text{CExpRew}^R_{\downarrow} [P] (\Diamond \text{sink} | \neg \Diamond \text{☇}) = \text{cwlp}[P](f)(\sigma)
\]
\[
\text{CLEExpRew}^R_{\downarrow} [P] (\Diamond \text{sink} | \neg \Diamond \text{☇}) = \text{cwlp}[P](g)(\sigma).
\]

**Proof.** The proof makes use of Lemmas 4.4, 4.5, and Lemma 3.2 which are themselves proven by induction on the structure of \( P \). For details see [12, p. 13-14, 16-21].

**5 Program Transformation**

In this section we present a program transformation for removing observations from fully probabilistic \( \text{cpGCL} \) programs and use the expectation transformer semantics
As a sanity check note that the expected value of $10+x$ in this program is equal to $10 \cdot \frac{3}{5} + 11 \cdot \frac{5}{11} = \frac{135}{11}$, which agrees with the result obtained by analyzing the original program. Formally, the program transformation is given by a function

$$\mathcal{T} : \text{cpGCL} \times \mathbb{E}_{\leq 1} \to \text{pGCL} \times \mathbb{E}_{\leq 1}.$$ 

To apply the transformation to a program $P$ we need to determine $\mathcal{T}(P, \mathbf{1})$, which gives the semantically equivalent program $\hat{P}$ and the expectation $\hat{h}$.

The transformation is defined in Figure 2 and works by inductively computing the weakest pre–expectation that guarantees the establishment of all $\text{observe}$– statements and updating the probability parameter of probabilistic choices so that the pre–expectations of their branches are established in accordance with the original probability parameter. The computation of these pre–expectations is performed following the same rules as the $\text{wp}$ operator. The correctness of the transformation is established by the following Theorem, which states that a program and its transformed version share the same terminating and non–terminating behavior.

**Theorem 5.1 (Program Transformation Correctness)** Let $P$ be a fully probabilistic cpGCL program that admits at least one valid run for every initial state, and let $T(P, \mathbf{1}) = (\hat{P}, \hat{h})$. Then for any $f \in \mathbb{E}$ and $g \in \mathbb{E}_{\leq 1}$, we have $\text{wp}[\hat{P}](f) = \text{cwP}[P](f)$ and $\text{wp}[\hat{P}](g) = \text{cwP}[P](g)$.

---

**Fig. 2.** Program transformation for eliminating $\text{observe}$ statements in fully probabilistic cpGCL programs.
Proof. See [12, p. 21].

A similar program transformation has been given by Nori et al. [22]. Whereas they use random assignments to introduce randomization in their programming model, we use probabilistic choices. Consequently, they can hoist observe-statements only until the occurrence of a random assignment, while we are able to hoist observe-statements over probabilistic choices and completely remove them from programs. Another difference is that the semantics of Nori et al. only accounts for terminating program behaviors and thus they can guarantee the correctness of the program transformation for almost–surely terminating programs only. Our semantics is more expressive and enables establishing the correctness for non–terminating program behavior, too.

6 Denotational Semantics for Full cpGCL

In this section we argue why (under mild assumptions) it is not possible to provide a denotational semantics in the style of conditional pre–expectation transformers (CPETs for short) for full cpGCL, i.e. including non–determinism. To show this, it suffices to consider a simple fragment of cpGCL containing only assignments, observations, probabilistic and non–deterministic choices. Let $x$ be the only program variable that can be written or read in this fragment. We denote this fragment with respect to some $\sigma_0$ and let $\llbracket \cdot \rrbracket : D \to \mathbb{R} \cup \{\perp\}$ be an interpretation function such that for any $d \in D$ we have that $\llbracket d \rrbracket$ is equal to the (possibly undefined) conditional expected value of $x$.

Definition 6.1 [Inductive CPETs] A CPET is a function $\text{cwp}^* : \text{cpGCL}^- \to D$ such that for any $P \in \text{cpGCL}^-$, $\llbracket \text{cwp}[P] \rrbracket = \text{CExpRew}^{\text{cpGCL}}[P] (\otimes \text{sink} \ |
eg \otimes \text{ok})$. $\text{cwp}^*$ is called inductive, if there exist two functions $\mathcal{K} : \text{cpGCL}^- \times [0, 1] \times \text{cpGCL}^- \to D$ and $\mathcal{N} : \text{cpGCL}^- \times \text{cpGCL}^- \to D$, such that for any $P_1, P_2 \in \text{cpGCL}^-$ we have $\text{cwp}^*[\{P_1\} \{p\} \{P_2\}] = \mathcal{K}(\text{cwp}^*[P_1], p, \text{cwp}^*[P_2])$ and $\text{cwp}^*[\{P_1\} \sqcap \{P_2\}] = \mathcal{N}(\text{cwp}^*[P_1], \text{cwp}^*[P_2])$, where $\forall d_1, d_2 \in D, \mathcal{N}(d_1, d_2) \in \{d_1, d_2\}$.

This definition suggests that the conditional pre–expectation of $\{P_1\} \{p\} \{P_2\}$ is determined only by the conditional pre–expectation of $P_1$ and $P_2$, and the probability $p$. Furthermore the above definition suggests that the conditional pre–expectation of $\{P_1\} \sqcap \{P_2\}$ is also determined by the conditional pre–expectation of $P_1$ and $P_2$ only. Consequently, the non–deterministic choice can be resolved by replacing it either by $P_1$ or $P_2$. While this might seem like a strong limitation, the above definition is compatible with the interpretation of non–deterministic choice as demonic choice: The choice is deterministically driven towards the worst option. The requirement $\mathcal{N}(d_1, d_2) \in \{d_1, d_2\}$ is also necessary for interpreting non–deterministic choice as an abstraction where implementation details are not important.

As we assume a fixed initial state and a fixed post–expectation, the non–deterministic choice turns out to be deterministic once the pre–expectations of $P_1$ and $P_2$ are known. Under the above assumptions (which do apply to the $\text{wp}$ and $\text{wlp}$ transformers) we claim:

156
**Theorem 6.2** There exists no inductive CPET.

**Proof Sketch** (for details, see [12, p. 11]). By contradiction: Consider the program $P = \{P_1\} [1/2] \{P_5\}$ with

- $P_1: \ x := 1$
- $P_3: \ \{P_2\} \Box \{P_1\}$
- $P_2: \ x := 2$
- $P_4: \ \{\text{observe false}\} [1/2] \{P_3\}$
- $P_3: \ x := 2.2$

A schematic depiction of the $R_{cwp}^x[P]$ is given in Figure 3. Assume there exists an inductive CPET $cwp^*$ over some appropriate domain $D$. With the program given above, one can get to the contradiction $\downarrow cwp^*[P_5] > cwp^*[P_2] = cwp^*[P_3]$. □

As an immediate corollary of Theorem 6.2 we obtain the following result:

**Corollary 6.3** We cannot extend the $cwp$ or $cwlp$ rules in Figure 1 for non–deterministic programs such that Theorem 4.6 extends to full cpGCL.

This result is related to Varacca and Winskel’s work [27], who have already noticed the difficulties that arise when trying to integrate non–determinism and probabilities, even in the absence of conditioning. When conditioning is taken into account, Andrés and van Rossum [1] have also observed that positional schedulers—i.e. the kind of schedulers implicitly considered in the expectation transformer semantics—are not sufficient for minimizing probabilities. In contrast to our work, their development is done in the context of temporal logics.

## 7 Conclusion and Future Work

This paper presented an extensive treatment of semantic issues in probabilistic programs with conditioning. Major contributions are the treatment of non–terminating programs (both operationally and for weakest liberal pre–expectations), our results on combining non–determinism with conditioning, as well as the presented program transformation. We firmly believe that a thorough understanding of these semantic issues provides a main cornerstone for enabling automated analysis techniques such as loop invariant synthesis [5,18], program analysis [6] and model checking [3] to the class of probabilistic programs with conditioning. Future work consists of investigating conditional invariants and a further investigation of non–determinism in combination with conditioning.

**Acknowledgments.** We would like to thank Pedro d’Argenio and Tahiry Rabehaja for the valuable discussions preceding this paper.
References


Reversible Monadic Computing

Chris Heunen\textsuperscript{1,2} \\
\textit{Department of Computer Science} \hfill \textit{University of Oxford} \hfill \textit{United Kingdom}

Martti Karvonen\textsuperscript{3} \\
\textit{Department of Mathematics and Systems Analysis} \hfill \textit{Aalto University} \hfill \textit{Finland}

Abstract

We extend categorical semantics of monadic programming to reversible computing, by considering monoidal closed dagger categories: the dagger gives reversibility, whereas closure gives higher-order expressivity. We demonstrate that Frobenius monads model the appropriate notion of coherence between the dagger and closure by reinforcing Cayley’s theorem; by proving that effectful computations (Kleisli morphisms) are reversible precisely when the monad is Frobenius; by characterizing the largest reversible subcategory of Eilenberg–Moore algebras; and by identifying the latter algebras as measurements in our leading example of quantum computing. Strong Frobenius monads are characterized internally by Frobenius monoids.

Keywords: Frobenius monad, dagger category, reversible computing, quantum measurement

1 Introduction

The categorical concept of a \textit{monad} has been tremendously useful in programming, as it extends purely functional programs with nonfunctional effects. For example, using monads one can extend a functional programming language with nondeterminism, probabilism, stateful computing, error handling, read-only environments, and input and output [51]. Haskell incorporates monads in its core language. On the theoretical side, there are satisfyingly clean categorical semantics. Simply typed \textit{\lambda}-calculus, that may be regarded as an idealized functional programming language, takes semantics in Cartesian closed categories [31]. The functional programming concept of a monad is modeled by the categorical concept of a monad [36].

\textsuperscript{1} Supported by the Engineering and Physical Sciences Research Council Fellowship EP/L002388/1. We thank an anonymous referee for Example 6.4, and Jorik Mandemaker, Sean Tull, and Maciej Pirog for helpful discussions.

\textsuperscript{2} Email: heunen@cs.ox.ac.uk

\textsuperscript{3} Email: martti.karvonen@aalto.fi

This paper is electronically published in \textit{Electronic Notes in Theoretical Computer Science} \\
URL: www.elsevier.nl/locate/entcs
In classical computation it is not always possible to reconstruct the input to an algorithm from its output. However, by using auxiliary bits, any classical computation can be turned into a reversible one [48]. Such a computation uses invertible primitive gates, and composition preserves invertibility. As discarding information requires work, reversible computations could in principle be implemented at higher speeds. The only operation costing power is the final discarding of auxiliary bits.

This is brought to a head in quantum computing, where any deterministic evolution of quantum bits is invertible, unlike the eventual measurement that converts quantum information to classical information. Another novelty in quantum computing is that it is impossible to copy or delete quantum information. This leads to a linear type theory of resources rather than a classical one [47]: quantum computing takes semantics in monoidal categories, rather than Cartesian ones [2].

Led by quantum computing, this article extends the categorical semantics of monadic programming to reversible computing. To allow for a linear type theory we consider monoidal closed categories. To allow for reversible computations, we consider dagger categories; in general these correspond to bidirectional computations rather than invertible ones, which in the quantum case comes down to the same thing. To allow for monadic effects, we introduce Frobenius monads. In the presence of a dagger, any monad gives rise to a comonad; a Frobenius monad is one that interacts with its comonad counterpart via the following Frobenius law:

\[ \begin{align*}
\begin{array}{c}
\begin{array}{c}
\downarrow \\
\downarrow
\end{array}
\end{array}
\end{align*}
\]

Here we used the graphical calculus for monoidal categories [44,34], that will be explained further in Section 2, along with several examples.\(^4\)

Our main contribution is to take reversal as a primitive and so justify the claim that Frobenius monads are precisely the right notion as follows:

- Section 3 justifies the Frobenius law as a necessary (and sufficient) consequence of coherence between the dagger and closure. In a reversible setting, it is natural to consider involutive monoids. In a monoidal closed category, any monoid embeds into a canonical one by Cayley’s theorem. We prove that this embedding preserves the involution induced by the dagger if and only if the monoid satisfies the Frobenius law. This derivation from first principles is a noncommutative generalization of [41, Theorem 4.3] with a new proof.

- Section 4 characterizes Frobenius monads internally. Monads are an external notion. A good example is the writer monad, that allows programs to keep auxiliary output alongside the computation. These values accumulate according to some monoid. Any monoid gives rise to a strong monad, and Frobenius monoids give rise to strong Frobenius monads. In general this is merely an adjunction and not an equivalence, but we work out that the converse holds in the Frobenius setting.

\(^4\) We often need to reason simultaneously about morphisms in a monoidal category and endofunctors on it. Unfortunately there is no sound and complete graphical proof calculus that would handle this yet. Therefore we cannot use the graphical calculus exclusively and also have to use traditional commutative diagrams.
This is a noncommutative generalization of [41, Corollary 4.5]. It also generalizes
the classic Eilenberg–Watts theorem from homological algebra to categories that
are not necessarily abelian. As Frobenius monoids satisfy the very same law (1)
as Frobenius monads, only interpreted in a category rather than by endofunctors
on it, this also exhibits that reversible settings are closed under categorification.

• We show that the extension of reversible pure computations with effects modeled
by a monad results in reversible effectful computations if and only if the monad
is a Frobenius monad. More precisely, Section 5 shows that a monad on a dagger
category is a Frobenius monad if and only if the dagger extends to the category
of Kleisli algebras. This reinforces that Frobenius monads model the right notion
of effects for reversible computing. Section 6 identifies the largest subcategory
of all algebras with this property, which we call Frobenius–Eilenberg–Moore al-
gebras. Section 7 exemplifies them in the quantum setting by arguing that they
correspond precisely to measurements via effect handlers [42].

Frobenius monads have been studied before [46,32], and monads have been used
as semantics for quantum computing before [15,4,3], but not in a dagger setting,
except for [41] that deals with the commutative case abstractly. Conversely, re-
versible programming has been modeled in dagger categories [6], but not using
monads. Daggers and monads were combined before in coalgebra [20,24], quantum
programming languages programming languages [14,45], and matrix algebra [11].
The current work differs by systematically starting from first principles. We intend
to fit probabilistic programming in our setup in future work.

2 Dagger categories

Let us model types as objects $A, B, C, \ldots$ in a category, and computations as mor-
phisms $f, g, h, \ldots$. To model composite types, we consider monoidal categories,
where one can not only compose computations in sequence $A \xrightarrow{f} B \xrightarrow{g} C$, but also in
parallel $A \otimes B \xrightarrow{f \otimes g} C \otimes D$. This much is standard [5]. To model reversible computations,
we need an operation turning a computation $A \xrightarrow{f} B$ into a computation $B \rightarrow A$, such that reversing twice doesn’t do anything.

Definition 2.1 A dagger is a functor $\dagger: C^{op} \rightarrow C$ satisfying $A^\dagger = A$ on objects
and $f^\dagger\dagger = f$ on morphisms. A dagger category is a category equipped with a dagger.

Dagger categories can behave quite different from ordinary (non-dagger) ones,
see e.g. [49, 9.7]. They are especially useful as semantics for quantum comput-
ing [19]. Note that reversible computing does not mean computations are invertible.
An invertible morphism $f$ in a dagger category is unitary when $f^\dagger = f^{-1}$. Similarly,
an endomorphism $f$ is self-adjoint when $f = f^\dagger$. As a rule, any structure in sight
should cooperate with the dagger.

Definition 2.2 A monoidal category is called a monoidal dagger category when
$(f \otimes g)^\dagger = f^\dagger \otimes g^\dagger$, and all coherence isomorphisms $A \otimes (B \otimes C) \xrightarrow{\vartriangleleft} (A \otimes B) \otimes C$,
$I \otimes A \xrightarrow{\triangleright} A$, and $A \otimes I \xrightarrow{\triangleleft} A$, are unitary. In a symmetric monoidal dagger category
additionally the swap maps $A \otimes B \xrightarrow{\triangleright} B \otimes A$ are unitary.

We will mainly consider the following two examples.
Example 2.3 The symmetric monoidal dagger category $\text{Rel}$ has sets as objects. Morphisms $A \to B$ are relations $R \subseteq A \times B$, with composition $S \circ R = \{(a, c) \mid \exists b: (a, b) \in R, (b, c) \in S\}$. The dagger is given by $R^t = \{(b, a) \mid (a, b) \in R\}$, and the monoidal structure is given by Cartesian products. We may think of $\text{Rel}$ as modeling nondeterministic computation [22].

Example 2.4 The symmetric monoidal dagger category $\text{FHilb}$ has finite-dimensional complex Hilbert spaces as objects and linear maps as morphisms. The dagger is given by adjoints: $f^\dagger$ is the unique linear function satisfying $\langle f(x) | y \rangle = \langle x | f^\dagger(y) \rangle$; in terms of matrices it is the conjugate transpose. The monoidal structure is given by tensor products of Hilbert spaces. This models quantum computation [2].

There are many other examples. Reversible probabilistic computation is modelled by the category of doubly stochastic maps [7, 2.3.5]; this generalizes to labelled Markov chains [38]. Universal constructions can generate examples with specific properties [40]. Finally, one can formally add daggers to a category in a free or cofree way [16, 3.1.17 and 3.1.19]. We will be interested in the following way to turn a monoidal dagger category into a new one of endofunctors on the old one. It could be regarded as modeling second-order computation, because the computations in the new category may refer to computations in the old one (but not to themselves).

Example 2.5 A functor $C \xrightarrow{F} D$ between dagger categories is a dagger functor when $F(f^\dagger) = (F(f))^\dagger$ on morphisms. Let $C$ be a monoidal dagger category. If $F \xrightarrow{\beta_A} G \xrightarrow{\beta_A} C$, then so is $G \xrightarrow{\beta_A} F \xrightarrow{\beta_A} C$, thus the category $(C, C)^\dagger$ of dagger functors $C \to C$ is again a monoidal dagger category by $G \otimes F = (G \otimes F)^\dagger$.

Monoidal dagger categories have a sound and complete graphical calculus, that we briefly recall; for more details, see [44]. A morphism $A \xrightarrow{f} B$ is represented as $\xrightarrow{f}$, and composition, the tensor product, and the dagger, become:

\[
\begin{array}{c}
g \circ f = \xrightarrow{g} \xrightarrow{f} \\
\xrightarrow{A} \xrightarrow{A}
\end{array}
\quad
\begin{array}{c}
f \otimes g = \xrightarrow{f} \xrightarrow{g} \\
\xrightarrow{A \otimes C} \xrightarrow{A} \xrightarrow{C}
\end{array}
\quad
\begin{array}{c}
f^\dagger = \xrightarrow{f} \\
\xrightarrow{B} \xrightarrow{B}
\end{array}
\]

Notice that the output wire $B \otimes D$ of a morphism $A \xrightarrow{f} B \otimes D$ becomes a pair of wires labelled $B$ and $D$ coming out of the box labelled $f$. Also, the dagger reflects in the horizontal axis, which is why we draw the boxes asymmetrically. Distinguished morphisms are often depicted with special diagrams instead of generic boxes as above. For example, the identity $A \to A$ is just the line $1$; the (identity on) the monoidal unit object $I$ is drawn as the empty picture, and the swap map of symmetric monoidal categories becomes $\times$. Soundness and completeness means that any equality between morphisms one can prove algebraically using the axioms
of monoidal dagger categories can equivalently and rigorously be proven graphically by isotopies of the graphical diagram.

To model higher order computation, we need function types. This is usually done by requiring closed monoidal categories, where the functors $- \otimes B$ have right adjoints $B \Rightarrow -$. That is, there is a natural bijective correspondence between morphisms $B \otimes A \xrightarrow{f} C$ and their curried version $A \xrightarrow{\Lambda(f)} B \Rightarrow C$. In the reversible setting of monoidal dagger categories, this closure operation should cooperate with the dagger: since $B \Rightarrow C$ is the type of computations $B \xrightarrow{f} C$, and those computations can be reversed to $C \xleftarrow{f^\dagger} B$, there should be an operation $(B \Rightarrow C) \rightarrow (C \Rightarrow B)$ modelling this internally (we will see this in more detail in Section 3). Therefore we demand that $B \Rightarrow -$ are dagger functors. It follows that they are not just right adjoint to $- \otimes B$, but also left adjoint. Now it is a small step to so-called compact dagger categories \cite{33,27}, which we make here for the sake of simplicity.

**Definition 2.6** A compact dagger category is a symmetric monoidal dagger category in which every object $A$ has a chosen dual object $A^*$ and a morphism $I \xrightarrow{u} A^* \otimes A$, drawn as $\sbullet$, satisfying $(u^\dagger \circ \sigma) \otimes \text{id} \circ (\text{id} \otimes u) = \text{id}$ and its dual:

\[
\begin{align*}
A & \quad A^* \quad A^- \\
A^* & \quad A \quad A^*^-
\end{align*}
\]

Compact dagger categories are automatically closed monoidal, with $(B \Rightarrow C) = B^* \otimes C$. Think of dual objects $B^*$ as input types, and primal objects $C$ as output types. By convention we choose $A^{**} = A$ and $(A \otimes B)^* = B^* \otimes A^*$.

Our previous examples in fact already satisfy this closure property of higher order computation: Rel and FHilb are compact dagger categories as follows. In Rel we can take $A^* = A$ and $u = \{(*,(a,a)) \mid a \in A\}$ for $I = \{\ast\}$. In FHilb we can take $H^*$ to be the dual Hilbert space of $H$; if $H$ has an orthonormal basis $\{e_1,\ldots,e_n\}$, then $H^*$ has an orthonormal basis $\{e_1^*,\ldots,e_n^*\}$, and we can take $u(1) = \sum_{i=1}^n e_i^* \otimes e_i$.

There is also a free compact dagger category on a given (dagger) category $C$ \cite{1}.

Let us conclude this preparatory section by contrasting reversible computing and invertible computing. A groupoid is a category where any morphism is invertible; it is always a dagger category with $f^\dagger = f^{-1}$. Any symmetric monoidal closed groupoid $G$ is a so-called compact category with $A^* = (A \Rightarrow I)$, as follows. Closure gives isomorphisms $(A \Rightarrow B) \otimes A \xrightarrow{\Lambda(ev)} B$ for all objects $A$ and $B$; in particular, $I \cong A^* \otimes A$. The morphisms $\Lambda(ev)$ are isomorphisms $A \cong A^{**}$, making $G$ into a so-called *-autonomous category \cite{5}. Because $G$ is symmetric monoidal, there are isomorphisms $A^* \otimes B^* \xrightarrow{\Lambda(ev \otimes ev)} (A \otimes B)^*$, making $G$ a compact category. However, this is not a compact dagger category unless all swap maps $\sigma$ are identities.

## 3 Frobenius monoids

This section considers monoids in monoidal dagger categories. We will see that, in the higher order setting of closed monoidal categories, our rule of thumb that
everything should cooperate with the dagger means considering Frobenius monoids.

**Definition 3.1** A **monoid** in a monoidal category is an object $A$ with morphisms $\cdot : A \otimes A \to A$ and $\mathbb{1} : I \to A$, satisfying:

\[
\begin{align*}
(\mathbb{1} \otimes f)(\cdot) &= f \\
(f \otimes \mathbb{1})(\cdot) &= f
\end{align*}
\]

It is **commutative** when $\cdot = \cdot \circ \sigma$. A **Frobenius monoid** is a monoid in a monoidal dagger category satisfying (1). It is **special** when $\cdot \circ (\cdot)^\dagger = \text{id}_A$.

A **comonoid** in $C$ is a monoid in $C^{\text{op}}$. The Frobenius law (1) makes sense for pairs of a monoid and comonoid on the same object, and most of Section 4 holds in that generality. Each side of the Frobenius law (1) equals $(\cdot)^\dagger \circ \cdot$; one of these equations is equivalent to (1). It is mostly motivated by observing that Frobenius monoids in specific categories are appropriate well-known mathematical structures.

**Example 3.2** Frobenius monoids in $\mathcal{FHilb}$ correspond to finite-dimensional $C^*$-algebras [50, Theorem 4.6]. These play a major role in quantum computing [28], but also as semantics for labelled Markov processes with bisimulations [35,43,30,37] and as operational semantics of probabilistic languages [12,13]. Commutative Frobenius monoids in $\mathcal{FHilb}$ therefore correspond to orthonormal bases when special [9].

**Example 3.3** Frobenius monoids in $\mathcal{Rel}$ correspond to (small) groupoids [18,39], which are important to invertible computing.

**Example 3.4** In a compact dagger category, $A^* \otimes A$ is a Frobenius monoid with $\mathbb{1} = u$, and $\cdot$ being the **pair of pants**:

\[
\begin{align*}
A^* & \quad A^* \\
\downarrow & \quad \downarrow \\
A^* & \quad A^*
\end{align*}
\]

This is precisely the monoid $A \rightarrow A$ of computations $A \rightarrow A$ under composition.

Pair of pants are universal, as the following generalization of Cayley’s theorem shows. A **monoid homomorphism** $f$ satisfies $\mathbb{1} = f \circ \mathbb{1}$ and $f \circ \cdot = \cdot \circ (f \otimes f)$.

**Lemma 3.5** Any monoid $(A,\cdot,\mathbb{1})$ in a compact category allows a monic monoid homomorphism $R$ into $A^* \otimes A$.

**Proof.** The following is a monoid homomorphism by (1):

\[
\begin{align*}
\begin{array}{ccc}
A^* & \rightarrow & A \\
\downarrow & & \downarrow \\
R & \rightarrow & A
\end{array}
\end{align*}
\]

(3)

It is monic because it has a left inverse $(\cdot)^\dagger \otimes \text{id}) \circ R$. 

\[\square\]
We will prove that the Cayley embedding of the previous lemma respects daggers precisely when the monoid is a Frobenius monoid. To make precise what it means to respect daggers, we need to internalize the operation $f \mapsto f^\dagger$ from $A \hookrightarrow A$ to the monoid $A \to A$. But the former might not be a well-defined morphism; for example, in $\mathbf{FHilb}$, taking conjugate transpose matrices is anti-linear, not linear, and hence a morphism $(A \to A) \to (A \to A)^*$ rather than an endomorphism. In a compact category, this is modeled by

$$
A^* \overset{f^*}{\rightarrow} A^*
$$

for $A \hookrightarrow B$. The operation $f \mapsto f^\dagger$ additionally is contravariant: $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$. So for it to be a monoid homomorphism the codomain has to have opposite multiplication as the domain.

**Lemma 3.6** If $(A, \cdot, \cdot, e)$ is a monoid in a compact category, then so is $(A^*, \cdot, \cdot, e^*)$, called the opposite monoid.

**Proof.** The functor $f \mapsto f^*$ is (strong) monoidal.

**Definition 3.7** A monoid $(A, \cdot, \cdot)$ in a compact dagger category is an involutive monoid when it is equipped with an involution: a monoid homomorphism $A \to A^*$ satisfying $i \circ i = \text{id}$. A homomorphism of involutive monoids is a monoid homomorphism $A \to B$ satisfying $i \circ f = f^* \circ i$.

Note that there is a canonical choice of involution:

$$
A^* \overset{i}{\rightarrow} A
$$

For the groupoids of Example 3.3, it is $g \mapsto g^{-1}$. For the C*-algebras of Example 3.2, it is $a \mapsto a^*$. The following theorem justifies the Frobenius law from first principles, generalizing [41, Theorem 4.3] noncommutatively.

**Theorem 3.8** A monoid in a compact dagger category is a Frobenius monoid if and only if (4) makes it involutive and (3) a homomorphism of involutive monoids.

**Proof.** Write $(A, \cdot, \cdot, e)$ for the monoid, and $i$ for (4). If $A$ is a Frobenius monoid, it follows from (1) that $i$ is indeed an involution. Observe that the involution on $A^* \otimes A$ is the identity because of our convention $(A \otimes B)^\dagger = B^* \otimes A^*$. So (3) preserves involutions when $R_s \circ i = R$:

$$
A^* \overset{R_s}{\rightarrow} A^* = A^* \overset{i}{\rightarrow} A^* = A^* \overset{R_s}{\rightarrow} A^* = A
$$

165
Conversely, assuming $R_* \circ i = R$:

\[
\begin{array}{c}
\text{(*)}
\end{array}
\]

Hence, by associativity:

\[
\begin{array}{c}
\text{(*)}
\end{array}
\]

But this is equivalent to (1).

4 Frobenius monads

A monad is a functor $\mathcal{C} \xrightarrow{T} \mathcal{C}$ with natural transformations $T(T(A)) \xrightarrow{\mu_A} T(A)$ and $A \xrightarrow{\eta_A} T(A)$ satisfying certain laws. It is well-known that monads are precisely monoids in categories of functors $\mathcal{C} \rightarrow \mathcal{C}$: Definition 3.1 unfolds to the monad laws

\[
\begin{align*}
\mu_A \circ T(\mu_A) &= \mu_A \circ \mu_{T(A)} , \\
\mu_A \circ T(\eta_A) &= id_{T(A)} = \mu_A \circ \eta_{T(A)}. 
\end{align*}
\]

There is a dual notion of a comonad. Daggers make any monoid (monad) give rise to a comonoid (comonad). Thus the Frobenius law (1) lifts to monads as follows.

**Definition 4.1** A Frobenius monad on a dagger category $\mathcal{C}$ is a Frobenius monoid in $[\mathcal{C}, \mathcal{C}]_\dagger$; explicitly, a monad $(T, \mu, \eta)$ on $\mathcal{C}$ with $T(f^\dagger) = T(f)^\dagger$ and

\[
T(\mu_A) \circ \mu_{T(A)} = \mu_{T(A)} \circ T(\mu_A).
\]

It is special when $\mu_A \circ \mu_A^\dagger = id_{T(A)}$.

Frobenius monads have been studied before by Street [46,32]. His definition does not take daggers into account, and concerns a monad rather than a monad-comonad pair. However, the natural generalization of the above definition to (non-dagger) monad-comonad pairs results in an equivalent notion to the one studied by Street. The primary example of a Frobenius monad is taking tensor products with a Frobenius monad.

**Example 4.2** If $(B, \delta, d)$ is a Frobenius monoid in a monoidal dagger category $\mathcal{C}$, then the functor $\mathcal{C} \xrightarrow{\otimes B} \mathcal{C}$, given by $A \mapsto A \otimes B$ and $f \mapsto f \otimes id$, is a Frobenius monad on $\mathcal{C}$ with:

\[
\mu_A = \begin{array}{c}
A \\
B
\end{array} \quad \text{and} \quad \eta_A = \begin{array}{c}
A \\
B
\end{array}
\]
Proof. The Frobenius monad law simply comes down to the Frobenius monoid law:

\[
T\mu \circ \mu^\dagger = \left( \begin{array}{ccc}
A & B & B \\
A & B & B
\end{array} \right) = \left( \begin{array}{ccc}
A & B & B \\
A & B & B
\end{array} \right) = \mu_T \circ T\mu^\dagger
\]

The monad laws become the monoid laws. Taking \( A = I \), we thus see that \( - \otimes B \) is a Frobenius monad if and only if \( B \) is a Frobenius monoid. \( \square \)

This section characterizes Frobenius monads of this form. There are, however, also other Frobenius monads, as in the following example.

Example 4.3 Consider the monoid \( \text{Rel}(\mathbb{N}, \mathbb{N}) \) of all relations \( \mathbb{N} \to \mathbb{N} \) as a single-object category. The following define a Frobenius monad on this category:

\[
T(R) = \{(2m, 2n) \mid (m, n) \in R\} \\
\cup \{(2m + 1, 2n + 1) \mid (m, n) \in R\} \\
\eta = \{(2n, 2n + 1) \mid n \in \mathbb{N}\} \\
\mu = \{(4n, 2n) \mid n \in \mathbb{N}\} \cup \{(4n + 3, 2n + 1) \mid n \in \mathbb{N}\}
\]

The functor \( - \otimes B \) comes with a natural transformation \( \alpha_{-,-,B} \), making it a strong functor. This natural transformation respects the monoid structure on \( B \).

Before recording some folklore results, we first define what this means for monads.

Definition 4.4 A functor \( F \) between monoidal categories is strong when it is equipped with a natural transformation \( A \otimes F(B) \xrightarrow{\text{st}\ A,B} F(A \otimes B) \) satisfying \( \text{st} \circ \alpha = F(\alpha) \circ \text{st} \circ (\text{id} \otimes \text{st}) \) and \( F(\lambda) \circ \text{st} = \lambda \). A morphism of strong functors is a natural transformation \( F \xrightarrow{\beta} G \) satisfying \( \beta \circ \text{st} = \text{st} \circ (\text{id} \otimes \beta) \). A strong monad is a monad \( (T, \mu, \eta) \) that is a strong functor satisfying \( \text{st} \circ (\text{id} \otimes \mu) = \mu \circ T(\text{st}) \circ \text{st} \) and \( \text{st} \circ (\text{id} \otimes \eta) = \eta \). A morphism of strong monads is a natural transformation, which is a morphism of the underlying monads and the underlying strong functors.

Proposition 4.5 Let \( C \) be a monoidal category. The operations \( B \mapsto - \otimes B \) and \( T \mapsto T(I) \) define an adjunction between monoids in \( C \) and strong monads on \( C \), with \( B \mapsto - \otimes B \) being the left adjoint.

Proof. See [52]. The unit of the adjunction is \( I \otimes B \xrightarrow{\lambda_B} B \). The counit is determined by \( A \otimes T(I) \xrightarrow{T(\rho) \circ \text{st}} T(A) \). \( \square \)

In the case of symmetric monoidal categories, there is also a notion of commutativity for strong monads [29,23]. Given a strong monad \( T \), one can define a natural transformation \( T(A) \otimes B \xrightarrow{\text{st}_{A,B}} T(A \otimes B) \) by \( T(\sigma_{B,A}) \circ \text{st}_{B,A} \circ \sigma_{T(A),B} \), and

\[
dst_{A,B} := \mu_A \otimes B \circ T(st'_{A,B}) \circ st_{T(A),B} \\
dst'_{A,B} := \mu_A \otimes B \circ T(st_{A,B}) \circ st'_{T(A),B}
\]

A strong monad is commutative when these coincide. Proposition 4.5 restricts to an adjunction between commutative monoids and commutative monads [52].
Definition 4.6 A costrong functor \( C \xrightarrow{F} D \) between monoidal categories is a functor that is strong when considered as a functor \( C^{\text{op}} \to D^{\text{op}} \). Explicitly, it has a natural transformation \( F(A \otimes B) \xrightarrow{\text{cst}_{A,B}} A \otimes F(B) \) satisfying \( F(\lambda) = \lambda \circ \text{cst}_{I,A} \) and \( \text{cst}_{A,B,C} \circ F(\alpha) = \alpha \circ (\text{id} \otimes \text{cst}_{B,C}) \circ \text{cst}_{A,B \otimes C} \). A morphism of costrong functors is a natural transformation \( F \xrightarrow{\beta} G \) satisfying \( \text{cst} \circ \beta = (\text{id} \otimes \beta) \circ \text{cst} \). A costrong comonad is a comonad \( (T, \delta, \varepsilon) \) that is a costrong functor, such that \( (\text{id} \otimes \varepsilon) \circ \text{cst} = \varepsilon \) and \( \text{cst} \circ T(\text{cst}) \circ \delta = (\text{id} \otimes \delta) \circ \text{cst} \). A morphism of costrong comonads is a natural transformation, which is a morphism of the underlying comonads and the underlying costrong functors.

Corollary 4.7 Let \( C \) be a monoidal category. The operations \( B \mapsto - \otimes B \) and \( T \mapsto T(I) \) form an adjunction between comonoids in \( C \) and costrong comonads on \( C \), but this time \( B \mapsto - \otimes B \) is the right adjoint. \( \square \)

In our reversible setting of dagger categories, any strong monad \( T \) is automatically a costrong comonad under \( \text{cst} = \text{st}^\dagger \), \( \delta = \mu^\dagger \), and \( \varepsilon = \eta^\dagger \). According to our motto that everything in sight should cooperate with the dagger, the reverse \( \text{cst} \) of \( \text{st} \) should in fact be its inverse, leading to the following definition.

Definition 4.8 A strong Frobenius monad on a monoidal dagger category \( C \) is a Frobenius monad \( (T, \mu, \eta) \) that is simultaneously a strong monad, such that each \( \text{st} \) is unitary. A morphism of strong Frobenius monads is just a morphism of the underlying strong monads.

The following theorem promotes the adjunction of Proposition 4.5 and Corollary 4.7 into an equivalence in the dagger setting. It generalizes [41, Theorem 4.5] noncommutatively. It also generalizes the classic Eilenberg–Watts theorem, that characterizes certain endofunctors on abelian categories as being of the form \( - \otimes B \) for a monoid \( B \), to monoidal dagger categories; note that there are monoidal dagger categories that are not abelian, such as \( \text{Rel} \) and \( \text{Hilb} \) [17, Appendix A].

Theorem 4.9 Let \( C \) be a monoidal dagger category. The operations \( B \mapsto - \otimes B \) and \( T \mapsto T(I) \) define an equivalence between Frobenius monoids in \( C \) and strong Frobenius monads on \( C \).

Proof. We already saw in Example 4.2 that \( B \mapsto - \otimes B \) preserves the Frobenius law. We prove that \( T \mapsto T(I) \) preserves the Frobenius law, too, in Lemma A.2 in the Appendix. It remains to prove that they form an equivalence. Clearly the unit of the adjunction, \( I \otimes B \xrightarrow{\text{st}^\dagger} B \), is a natural isomorphism. To prove that the counit \( A \otimes T(I) \xrightarrow{T(\mu)\text{cst}} T(A) \) is also a natural isomorphism, notice that by definition it is a morphism of strong monads. In Lemma A.3 in the Appendix we prove that it is also a morphism of comonads. But homomorphisms of Frobenius monoids must be isomorphisms by Lemma A.1. \( \square \)

The previous theorem restricts to an equivalence between commutative/special Frobenius monoids and commutative/special strong Frobenius monads (see Corollary A.4 in the Appendix).

One might think it too strong to require \( \text{st} \) to be unitary. The following counterexample shows that Theorem 4.9 would fail if we abandoned that requirement.
Example 4.10 Let’s call a Frobenius monad rather strong when it is simultaneously a strong monad. The operations of Theorem 4.9 do not form an adjunction between Frobenius monoids and rather strong Frobenius monads, because the counit of the adjunction would not be a well-defined morphism. To produce a counterexample where the counit does not preserve comultiplication comes down to finding a rather strong Frobenius monad with $T(\eta_A) \circ \eta_A \neq \mu_A^\dagger \circ \eta_A$ for some $A$. This is the case when $T$ is $- \otimes B$ for a Frobenius monoid $B$ with $\delta \otimes \delta \neq (\delta_\otimes)^\dagger \circ \delta$. Such Frobenius monoids certainly exist: if $G$ is any nontrivial group, regarded as a Frobenius monoid in $\text{Rel}$ via Example 3.3, then $\delta \otimes \delta$ is the relation $\{(\ast, (1, 1))\}$, but $(\delta_\otimes)^\dagger \circ \delta = \{((\ast, (g, g^{-1})) \mid g \in G\}$.

5 Kleisli algebras

One of the standard categorical constructions when given a monad $T$ is to consider the category $\mathcal{C}_T$ of its Kleisli algebras. In monadic programming, this category gives semantics for computations with effects modeled by $T$, whereas the base category $\mathcal{C}$ only gives semantics for pure computations [25]. In this section we show that if $T$ is a Frobenius monad, then $\mathcal{C}_T$ is a dagger category. In fact we also show the converse, under a natural condition about cooperation with daggers. Thus effects modeled by a monad can be added without leaving the setting of reversible computations precisely when the monad is a Frobenius monad.

Definition 5.1 If $\mathcal{C} \xrightarrow{T} \mathcal{C}$ is a monad, its Kleisli category $\mathcal{C}_T$ is defined as follows. Objects are the same as in $\mathcal{C}$. A morphism $A \rightarrow B$ in $\mathcal{C}_T$ is a morphism $A \xrightarrow{f} T(B)$ in $\mathcal{C}$. Identities are given by $\eta$, and composition of $g$ and $f$ in $\mathcal{C}_T$ is given by $\mu \circ T(g) \circ f$.

There is a forgetful functor $\mathcal{C}_T \rightarrow \mathcal{C}$ given by $A \mapsto T(A)$ on objects and $f \mapsto \mu \circ T(f)$ on morphisms. It has a left adjoint $\mathcal{C} \rightarrow \mathcal{C}_T$ given by $A \mapsto A$ on objects and $f \mapsto \eta \circ f$ on morphisms.

We now show that for Frobenius monads the Kleisli construction preserves daggers.

Lemma 5.2 If $T$ is a Frobenius monad on a dagger category $\mathcal{C}$, then $\mathcal{C}_T$ carries a dagger that commutes with the canonical functors $\mathcal{C}_T \rightarrow \mathcal{C}$ and $\mathcal{C} \rightarrow \mathcal{C}_T$.

Proof. A straightforward calculation establishes that

$$(A \xrightarrow{f} T(B)) \mapsto (B \xrightarrow{\mu^\dagger} T^2(B) \xrightarrow{T(f)} T(A))$$

is a dagger on $\mathcal{C}_T$ commuting with the canonical functors $\mathcal{C} \rightarrow \mathcal{C}_T$ and $\mathcal{C}_T \rightarrow \mathcal{C}$.$\square$

The following theorem proves a converse of the previous lemma, under the natural condition that the “reverse identity morphisms” of the Kleisli category equal their own dagger. This gives another characterization of Frobenius monads, in terms of reversibility of their effectful computations.

Theorem 5.3 A monad $T$ on a dagger category $\mathcal{C}$ is a Frobenius monad if and only if $\mathcal{C}_T$ has a dagger such that:

- the functors $\mathcal{C} \rightarrow \mathcal{C}_T$ and $\mathcal{C}_T \rightarrow \mathcal{C}$ are dagger functors;
The morphisms $\mu_A^\dagger : T(A) \to T^2(A)$ of $C$ are self-adjoint when regarded as morphisms $T(A) \to T(A)$ of $C_T$.

**Proof.** One direction follows from Lemma 5.2 and the observation that with that dagger the morphism $\mu_A^\dagger : T(A) \to T^2(A)$ is self-adjoint in $C_T$. For the other direction, we wish to show that the following diagram commutes for arbitrary $A$.

$$
\begin{array}{ccc}
T^2(A) & \xrightarrow{T(\mu_A^\dagger)} & T^3(A) \\
\mu_T(A)^\dagger \downarrow & & \downarrow \mu_T(A) \\
T^3(A) & \xrightarrow{T(\mu_A)} & T^2(A)
\end{array}
$$

Write $C \xrightarrow{F} C_T$ and $C_T \xrightarrow{G} C$ for the canonical functors. Note that if we consider $\text{id}_{T^2(A)}$ and $\eta_{T(A)} \circ \mu_A$ as morphisms of $C_T$, then we have $G(\text{id}_{T^2(A)}) = \mu_{T(A)}$, $G(\eta_{T(A)} \circ \mu_A) = T(\mu_A)$, and $F(\mu_A) = \eta_{T(A)} \circ \mu_A$. As $G$ is a dagger functor, we have found preimages of all the morphisms in the diagram. More explicitly, we know that

$$
G(\text{id}_{T^2(A)} \circ F(\mu_A^\dagger)) = \mu_{T(A)} \circ T(\mu_A^\dagger),
$$

$$
G(F(\mu_A) \circ \text{id}_{T^2(A)}) = T(\mu_A) \circ \mu_{T(A)}.
$$

Hence it suffices to show $\text{id}_{T^2(A)} \circ F(\mu_A^\dagger) = F(\mu_A) \circ \text{id}_{T^2(A)}$. As the left hand side is the dagger of the right hand side and $\mu_A^\dagger$ is self-adjoint in $C_T$, it suffices to show that either equals $\mu_A^\dagger$. The following calculation does this for the left-hand side:

$$
\text{id}_{T^2(A)} \circ F(\mu_A^\dagger) = \mu_{T(A)} \circ T(\text{id}_{T^2(A)}) \circ \eta_{T^2(A)} \circ \mu_A^\dagger
$$

$$
= \mu_{T(A)} \circ \eta_{T^2(A)} \circ \mu_A^\dagger
$$

This completes the proof. \qed

Kleisli categories of commutative monads on symmetric monoidal categories are again symmetric monoidal [10]. This extends to the reversible setting.

**Theorem 5.4** If $T$ is a commutative strong Frobenius monad on a symmetric monoidal dagger category $C$, then $C_T$ is a symmetric monoidal dagger category.

**Proof.** The monoidal structure on $C_T$ is given by $A \otimes_T B = A \otimes B$ on objects and by $f \otimes_T g = \text{dst} \circ (f \otimes g)$ on morphisms. The coherence isomorphisms of $C_T$ are images of those in $C$ under the functor $C \to C_T$. This functor preserves daggers and hence unitaries, making all coherence isomorphisms of $C_T$ unitary. It remains to check that the dagger on $C_T$ satisfies $(f \otimes_T g)^\dagger = f^\dagger \otimes_T g^\dagger$. By Theorem 4.9, $T$ is isomorphic to $- \otimes T(I)$, and it is straightforward to check that this induces an isomorphism between the respective Kleisli categories that preserves daggers and monoidal structure on the nose. Thus it suffices to check that this equation holds on $C_{-\otimes T(I)}$, which can be done with a straightforward graphical argument. \qed

170
6 Frobenius–Eilenberg–Moore algebras

The other canonical standard categorical construction when given a monad \( T \) is to consider the category \( C^T \) of its Eilenberg–Moore algebras. In monadic programming, these are understood to expand effectful computations to pure computations [25]. This section identifies the largest full subcategory of \( C^T \) that is still reversible.

**Definition 6.1** An *Eilenberg–Moore algebra* \((A, a)\) for a monad \( T \) is a morphism \( T(A) \xrightarrow{a} A \) satisfying \( a \circ T(a) = a \circ \mu \) and \( a \circ \eta = \text{id} \). A *morphism of Eilenberg–Moore algebras* \((A, a) \to (B, b)\) is a morphism \( A \xrightarrow{f} B \) satisfying \( b \circ T(f) = f \circ a \). These form a category \( C^T \).

We will again need cooperation of such algebras with daggers when present.

**Definition 6.2** Let \( T \) be a monad on a dagger category \( C \). A *Frobenius–Eilenberg–Moore algebra*, or *FEM-algebra* for short, is an Eilenberg–Moore algebra \((A, a)\) that makes the following diagram commute.

\[
\begin{array}{ccc}
T(A) & \xrightarrow{T(a)^\dagger} & T^2(A) \\
\downarrow \mu^\dagger & & \downarrow \mu \\
T^2(A) & \xrightarrow{T(a)} & T(A)
\end{array}
\]

We call this the *Frobenius law* for Eilenberg–Moore algebras.

**Example 6.3** The Kleisli category \( C_T \) of any monad \( T \) sits inside \( C^T \) as the free algebras \((T(A), \mu_A)\). If \( T \) is a Frobenius monad on a dagger category \( C \), any free algebra is an FEM-algebra.

**Proof.** The Frobenius law for the free algebra is the Frobenius law of the monad. □

There are many EM-algebras that are not FEM-algebras; a family of examples can be derived from [41, Theorem 6.4]. Here is a concrete example.

**Example 6.4** Let \( A = M_2(\mathbb{C}) \) be the Hilbert space of 2-by-2-matrices, with inner product \( \langle a, b \rangle = \frac{1}{2} \text{Tr}(a^\dagger \circ b) \). Matrix multiplication gives a map \( m: A \otimes A \to A \) making \( A \) a Frobenius monoid in \( \text{FHilb} \), so that \( T = - \otimes A \) is a Frobenius monad. Let \( u \in A \) be a unitary matrix, and define \( U: A \to A \) by \( U(a) = u^\dagger \circ a \circ u \). Now \( U^\dagger(a) = u a u^\dagger \) and \( U \) is an endomorphism of the monoid \( A \), making \( h = m \circ (\text{id} \otimes U) \) an EM-algebra. It is an FEM-algebra if and only if \( u = u^\dagger \).

**Proof.** The Frobenius law (5) means:

\[
\begin{array}{c}
\text{U} \\
\downarrow \text{U} \text{^dagger} \\
\text{U}
\end{array} =
\begin{array}{c}
\text{U} \\
\downarrow \text{U} \text{^dagger} \\
\text{U}
\end{array}
\]

This comes down to \( U = (U^*)^\dagger \), that is, \( u = u^\dagger \). □
The following two results highlight the importance of FEM-algebras to daggers. First, extending from pure computations to FEM–computations is still reversible.

**Proposition 6.5** Let $T$ be a Frobenius monad on a dagger category $\mathbf{C}$. The dagger on $\mathbf{C}$ induces a dagger on the category of FEM-algebras of $T$.

**Proof.** Let $f : (A, a) \rightarrow (B, b)$ be a morphism of FEM-algebras; we have to show that $f^\dagger$ is a morphism $(B, b) \rightarrow (A, a)$. It suffices to show that $b \circ T(f) = f \circ a$ implies $a \circ T(f^\dagger) = f^\dagger \circ b$. Consider the following diagram:

Region (i) is the Frobenius law of $(B, b)$; commutativity of (ii) follows from the assumption that $f$ is a morphism $(A, a) \rightarrow (B, b)$ by applying $T$ and $\dagger$; (iii) is naturality of $\mu$; (iv) is the Frobenius law of $(A, a)$; (v) commutes since $T$ is a comonad; (vi) and (vii) commute by naturality of $\eta^\dagger$.

Second, FEM–computations are the largest class that stays reversible.

**Theorem 6.6** FEM-algebras form the largest full subcategory of $\mathbf{C}^T$ containing $\mathbf{C}_T$ that carries a dagger commuting with the forgetful functor $\mathbf{C}^T \rightarrow \mathbf{C}$.

**Proof.** Suppose that an EM-algebra $(A, a)$ is such that for any free algebra $(T(B), \mu_B)$ and any morphism $f : T(B) \rightarrow A$, $f$ is a morphism of EM-algebras $(T(B), \mu_B) \rightarrow (A, a)$ iff $f^\dagger$ is a morphism $(A, a) \rightarrow (T(B), \mu_B)$ of EM-algebras. Now $(A, a)$ being an EM-algebra implies that $a$ is a morphism $(T(A), \mu_A) \rightarrow (A, a)$. Thus by assumption $a^\dagger$ is a morphism $(A, a) \rightarrow (T(A), \mu_A)$, which implies that $(A, a)$ is an FEM-algebra.

7 Quantum measurement

This final section exemplifies the relevance of FEM-algebras to quantum computation, by indicating how quantum measurement fits neatly in effectful functional programming as handlers of Frobenius monads [42,26].

**Example 7.1** Let $B$ be a finite-dimensional Hilbert space. A choice of orthonormal basis makes $B$ a commutative Frobenius monoid in $\mathbf{FHilb}$ via Example 3.2. Hence $T = [- \otimes B]$ is a (commutative strong) Frobenius monad on $\mathbf{FHilb}$ by Theorem 4.9.

Traditionally, effectful computations are modelled as morphisms in the Kleisli category [36,51]. In the above example, those are just morphisms $A \rightarrow A \otimes B$
in $\text{FHilb}$. Quantum measurements are indeed morphisms of this type, but they satisfy more requirements, such as von Neumann’s projection postulate: repeating a measurement is equivalent to copying the outcome of the first measurement. These requirements make the dagger of the morphism $A \to A \otimes B$ precisely an FEM-algebra, see [8, Theorems 1.5 and 1.6]. The following proposition summarizes.

**Proposition 7.2** Quantum measurements with outcomes modeled by a commutative strong Frobenius monad on $\text{FHilb}$ correspond precisely to its FEM-algebras. □

Consider the exception monad $T$ that adds exceptions from a set $E$ to a computation by $T(A) = A + E$. Intercepting exceptions means executing a computation $f_e$ for each $e \in E$, and a computation $f$ if no exception is raised. Thus a handler for $T$ specifies an EM-algebra $(A, a)$ and a map $f : A \to A$ making the triangle left below commute.

$$
\begin{array}{ccc}
A & \xrightarrow{f} & A \\
\eta_A \downarrow & & \downarrow \eta_A \\
A + E & \xrightarrow{\cdot} & (A, a)
\end{array}
$$

This extends to arbitrary algebraic effects $T$ [42]. In particular, it makes sense for quantum measurement, as in the right diagram above. The Frobenius monad $- \otimes B$ modeling quantum measurement with outcomes in $B$ is similar to ‘raising exceptions $B$', the vertical arrows are Kleisli morphisms, and the lower right handling construct is an FEM-algebra $A \otimes B \xrightarrow{\cdot} A$ that ‘handles exceptions $B$': it involves the unique dashed arrow, that is induced by the free property of the Kleisli algebra $A \otimes B$, and is a morphism of FEM-algebras by Example 6.3. Intuitively, Kleisli morphisms $A \to T(B)$ are constructors that ‘build’ an effectful computation, whereas FEM algebras $T(B) \to B$ are destructors that ‘handle’ the effects.

Thus in general, effectful reversible computation takes place in the category of FEM-algebras of a Frobenius monad, rather than its subcategory of Kleisli algebras. See also [21] for a similar reasoning in different language.

### 8 Conclusion

We have proposed Frobenius monads as the appropriate notion to model computational effects in the reversible setting of dagger categories. We have justified their definition from first principles, characterized them internally, shown that their Kleisli categories are again reversible, and identified the largest reversible subcategory of their Eilenberg–Moore categories. As an example we phrased quantum measurement in the category of such Frobenius–Eilenberg–Moore algebras.

More examples should be studied. Specifically, noncommutative Frobenius monoids on $\text{FHilb}$ might induce monads modelling partial quantum measurement. Also, the relationship between nondeterministic computation in $\text{Rel}$ and groupoids should be explored. Finally, we leave probabilistic computation to future work.

---

5 Technically, the monad has to be lifted to a category of so-called completely positive maps, see [8].
References


175
A Proofs

This appendix verifies steps used in proofs in Section 4.

Lemma A.1 A monoid homomorphism between Frobenius monoids in a monoidal dagger category, that is also a comonoid homomorphism, is an isomorphism.

Proof. Construct an inverse to \( A \xrightarrow{f} B \) as follows:

The composite with \( f \) gives the identity in one direction:

The third equality uses the Frobenius law (1) and unitality. The other composite is the identity by a similar argument. \( \square \)

Lemma A.2 The functor \( T \mapsto T(I) \) preserves the Frobenius law.

Proof. Consider the diagram in Fig. A.1. Region (i) commutes because \( T \) is a Frobenius monad, (ii) because \( \mu^\dagger \) is natural, (iii) because \( \rho^{-1} \) is natural, (iv) because \( st^\dagger \) is natural, (v) is a consequence of \( T \) being a strong monad, (vi) commutes as \( \rho \) is natural, (vii) and (viii) because \( st \) is natural, (ix) commutes trivially and (x) because \( st \) is natural. Regions (ii)’-\( x \)’ commute for dual reasons. Hence the outer diagram commutes, and \( T \mapsto T(I) \) preserves the Frobenius law. \( \square \)

Lemma A.3 If \( T \) is a strong Frobenius monad, the counit of the adjunction of Proposition 4.5 is a morphism of comonads.

Proof. First we show that the counit of the adjunction preserves counits of the comonads. It suffices to see that

commutes. But the rectangle commutes because \( \eta^\dagger \) is natural, and the triangle commutes because \( T \) is a strong monad and \( st \) is an isomorphism.
To see that the counit of the adjunction preserves the comultiplication, consider the following diagram:

\[
\begin{array}{cccc}
A \otimes T(I) & \xrightarrow{id \otimes \mu^!} & A \otimes T^2(I) & \xrightarrow{id \otimes T(\rho^{-1})} & A \otimes T(T(I) \otimes I) & \xrightarrow{id \otimes st^!} & A \otimes (T(I) \otimes T(I)) \\
\downarrow{id} & & \downarrow{id \otimes T(\rho)} & & \downarrow{id \otimes st} & & \downarrow{\alpha} \\
A \otimes T^2(I) & \leftarrow A \otimes T(T(I) \otimes I) & \leftarrow (A \otimes T(I)) \otimes T(I) \\
\downarrow{st} & & \downarrow{id} & \downarrow{id} & \downarrow{id} & \downarrow{id} \\
T(A \otimes I) & \xrightarrow{T(id \otimes \rho)} & T(A \otimes T(I)) & \xrightarrow{T(id \otimes \rho)} & T(A \otimes T(I)) \\
\downarrow{T(\rho)} & & \downarrow{\mu^!} & & \downarrow{\mu^!} \\
T(A) & \xrightarrow{T(\rho)} & T^2(A) & \xrightarrow{T(\rho)} & T^2(A) \\
\end{array}
\]

Commutativity of region (i) is a consequence of \( T \) being a strong monad, and \( st \) being an iso, (ii) commutes by definition, (iii) commutes as \( st \) is natural, (iv) because \( T \) is a strong functor, (v) by coherence and finally (vi) by naturality of \( \mu^! \). Hence the outer diagram commutes, and the counit of the adjunction preserves the comultiplication.

**Corollary A.4** The equivalence of Theorem 4.9 restricts to an equivalence between special Frobenius monoids and special strong Frobenius monads.

**Proof.** The commutative case follows from [52]. If the Frobenius monoid is special, so is the monad, by trivial graphical manipulation of Example 4.2. Conversely, if the Frobenius monad \( T \) is special, the following diagram commutes:

\[
\begin{array}{cccc}
T(I) & \xrightarrow{\mu^!} & T^2(I) & \xrightarrow{T(\rho^{-1})} & T(T(I) \otimes I) & \xrightarrow{st^{-1}} & T(I) \otimes T(I) \\
\downarrow{id} & & \downarrow{id} & & \downarrow{id} & & \downarrow{id} \\
T(I) & \xleftarrow{\mu} & T^2(I) & \xleftarrow{T(\rho)} & T(T(I) \otimes I) \\
\end{array}
\]

and so \( T(I) \) is special. □

177
Fig. A.1. Diagram proving that $T \mapsto T(I)$ preserves the Frobenius law.
Alternation-Free Weighted Mu-Calculus: Decidability and Completeness

Kim G. Larsen  Radu Mardare  Bingtian Xue

Aalborg University, Denmark

Abstract

In this paper we introduce WMC, a weighted version of the alternation-free modal mu-calculus for weighted transition systems. WMC subsumes previously studied weighted extensions of CTL and resembles previously proposed time-extended versions of the modal mu-calculus. We develop, in addition, a symbolic semantics for WMC and demonstrate that the notion of satisfiability coincides with that of symbolic satisfiability. This central result allows us to prove two major meta-properties of WMC. The first is decidability of satisfiability for WMC. In contrast to the classical modal mu-calculus, WMC does not possess the finite model-property. Nevertheless, the finite model property holds for the symbolic semantics and decidability readily follows; and this contrasts to resembling logics for timed transitions systems for which satisfiability has been shown undecidable. As a second main contribution, we provide a complete axiomatization, which applies to both semantics. The completeness proof is non-standard, since the logic is non-compact, and it involves the notion of symbolic models.

Keywords: weighted modal Mu-Calculus, non-compact modal logics, weighted transition systems, satisfiability, complete axiomatization.

1 Introduction

For more than two decades, specification and modelling formalisms have been sought that address essential non-functional properties of embedded and cyber-physical systems. In particular, timed automata [4] were used for expressing and analysing timing constraints of systems with respect to timed logics such as TCTL [3], $T_{\mu}$ [17], $L_{\nu}$ [23] and MTL [19]. However, equally important non-functional properties of embedded or cyber-physical systems are related to consumption of resources, in particular that of energy. This lead initially to weighted extensions of timed automata [5, 6] and most recently to energy automata [9]. However, whereas the problems of cost-optimal reachability and infinite runs have been shown to be efficiently computable, the general model checking problem with respect to a weighted extension of TCTL turns out to be undecidable [11].

In this paper, we consider the purely weighted setting, in which the quantitative information of systems is modelled as weighted transition systems (WTSs) with transitions being

---

1 This research is partially supported by the Sino-Danish Basic Research Center IDEA4CPS.
2 Email: kgl,mardare,bingt@cs.aau.dk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
decorated with non-negative reals besides actions. We study the problems of satisfiability and axiomatization of weighted logics in the most general setting. We develop WMC, a weighted version of the alternation-free modal mu-calculus, that subsumes WCTL and resembles the previously studied timed extension of the modal mu-calculus $T_{\mu}$ and $L_{\nu}$. WMC is a multi-modal logic with fixed-point operators, where modalities either constrain discrete transitions or the amount of resources in a given state. For the latter, WMC uses resource-variables, similar to the clock-variables used in timed logics, see e.g. [10].

Our first main contribution is to show decidability of satisfiability for WMC. In previous work [27], we proved decidability and finite model property for restriction of WMC with only one resource-variable for each resource and only maximal fixed points. This restriction bounds severely the expressiveness of the logic. In [25, 26], we studied two sub-logics of WMC with multiple resource-variables for each resource and only maximal fixed points. These logics are shown decidable by using the filtration construction, but are significantly weaker than WMC in that resource-variables are restricted to be event-recording. In contrast to these fragments and to modal mu-calculus, WMC does not posses finite model property, thus decidability does not follow from classical arguments. As an alternative, we propose here notions of symbolic model and semantics for which the finite model property does hold. Fortunately – as demonstrated in the paper – the notion of satisfiability coincides with that of symbolic satisfiability, from which our decidability result follows. This should be contrasted to the resembling timed logics for which satisfiability is undecidable.

The fact that the two semantics have the same validities is a remarkable property and a powerful tool that allows us to transport meta-results between the two semantics, in particular computability and complexity results for satisfiability checking and completeness results for proof systems.

Our second main contribution is a complete axiomatization of WMC, allowing all valid properties to be derived as theorems. At the best of our knowledge, this is the first complete axiomatization for a fixed point weighted modal logic in the literature. The axiomatization is remarkably simple, combining modal axioms of non-recursive weighted logic with classic axioms of fixed points [20, 28, 30]. The finite model property provides the arguments to demonstrate that the axiomatization is complete for the symbolic semantics and hence, the completeness result can be extrapolated to the WTS-semantics.

Our third main contribution is the completeness proof itself, which is non-standard and novel in many aspects. Since the logic is non-compact, it requires infinitary proof rules. To cope with this, we involve topological techniques for model theory, inspired by the work of Rasiowa and Sikorski [16, 29]. These techniques were previously developed by the first two authors in collaboration with Kozen and Panangaden for proving the strong completeness for Markovian logics [21, 22]. Our completeness proof avoids the tableaux method used in [32] for the general Mu-Calculus and it is sufficiently robust to be reused in similar contexts. On the other hand, our proof is designed for alternation-free calculi and it is not clear whether it can be used in a general unrestricted context.
2 Alternation-Free Weighted Mu-Calculus

Definition 2.1 A weighted transition system is a tuple \( W = (M, \mathcal{K}, \Sigma, \theta) \) where \( M \) is a non-empty set of states, \( \mathcal{K} = \{e_1, \ldots, e_k\} \) is a finite set of resources, \( \Sigma \) a non-empty set of actions and \( \theta : M \times \Sigma \times (\mathcal{K} \to \mathbb{R}_{\geq 0}) \to 2^M \) is a labelled transition function.

Instead of \( m' \in \theta(m, a, f) \), we write \( m \xrightarrow{f,a} m' \) and we call \( f \) the weight function. For simplicity, in what follows we assume that \( \mathcal{K} \) is a singleton and we use the transition functions \( \theta : M \times (\Sigma \times \mathbb{R}_{\geq 0}) \to 2^M \). However, the work can be straightforwardly extended to include multiple resources and all the following results hold in the extended case.

Alternation-Free Weighted Mu-Calculus (WMC) encodes properties of WTSs and involves modal operators and resource-variables similar to the ones used in timed logics [1, 3, 17]:

(i) transition modalities of type \([a] \) for \( a \in \Sigma \);
(ii) recursive-variables that range over the set \( X \); they are used to define simultaneous recursive equations to express maximal and minimum fixed points, in the style of [12, 13, 24];
(iii) resource-variables ranging over the set \( \mathcal{V} \);
(iv)) state modalities of type \( x \preceq r \) for \( r \in \mathbb{R}_{\preceq} \) and \( r \in \mathbb{Q}_{\geq 0} \), which approximates the resource-variable \( x \in \mathcal{V} \);
(v) reset operators of type \( x \) in for the resource-variable.

Notation: we use both \( \preceq \) and \( \geq \) to range over the set \( \{\preceq, \geq\} = \{\leq, \geq\} \). Similarly, we use \( < \) and \( > \) to range over the set \( \{<, >\} \).

Definition 2.2 [Syntax] The formulas of WMC are defined by the following grammar, for arbitrary \( r \in \mathbb{Q}_{\geq 0}, a \in \Sigma, x \in \mathcal{V}, \preceq \in \{\preceq, \geq\} \) and \( X \in \mathcal{X} \).

\[
\mathcal{L} : \quad \phi ::= \ x \preceq r \mid \neg \phi \mid \phi \lor [a] \phi \mid x \ \text{in} \ \phi \mid X.
\]

We also consider the De Morgan duals of \( x \preceq r \) and \([a] \), defined by

\[
x \preceq r = \neg(x \geq r) \quad \text{and} \quad [a] = \neg([a] = \neg \phi)\text{ respectively}.
\]

Given \( \phi_1, \phi_2, \ldots, \psi_n \in \mathcal{L} \) and \( X_1, \ldots, X_n \in \mathcal{X} \), let \( \phi_1/X_1, \ldots, \psi_n/X_n \) be the formula obtained by substituting each occurrence of the variable \( X_i \) in \( \phi \) with \( \psi_i \) for each \( i = 1 \ldots n \). If \( \bar{\psi} = (\psi_1, \ldots, \psi_n) \) and \( \bar{X} = (X_1, \ldots, X_n) \), let \( \phi(\bar{\psi}/\bar{X}) \) denote \( \phi(\psi_1/X_1, \ldots, \psi_n/X_n) \). Following [12, 13], we allow sets of the maximal or minimal blocks of mutually recursive equations in WMC.

Definition 2.3 [Equation Blocks] An equation block \( B \) over the set \( X_B = \{X_1, \ldots, X_n\} \) of pairwise distinct variables has one of two forms – \( \min(E) \) or \( \max(E) \), where \( E \) is a system of (mutually recursive) equations such that for any \( i, j \in \{1, \ldots, m\} \), \( \phi_i \) is monotonic in \( X_j \).

\[
E : \quad \{ X_1 = \phi_1, \ldots, X_n = \phi_n \}
\]

If \( B = \max(E) \) or \( B = \min(E) \), the elements of \( X_B \) are called max-variables or min-variables respectively. Given the system \( E \) of equations in the previous definition, its dual is

\[
\bar{E} : \quad \{ X_1 = \neg \phi_1(\neg X_1/X_1, \ldots, \neg X_n/X_n), \ldots, X_n = \neg \phi_n(\neg X_1/X_1, \ldots, \neg X_n/X_n) \}
\]

If \( B = \max(E) \) or \( B = \min(E) \), then its dual is \( \bar{B} = \min(\bar{E}) \) or \( \bar{B} = \max(\bar{E}) \) respectively.

Given a block \( B \), a formula \( \phi \in \mathcal{L} \) depends on \( B \) if it involves variables in \( X_B \). Given two blocks \( B \) and \( B' \) such that \( X_B \cap X_{B'} = \emptyset \), we say that \( B \) is dependent on \( B' \) if the right hand side formulas of the equations of \( B \) depend on \( B' \).
**Definition 2.4** [Alternation-Free Block Sequence] A sequence \( \mathcal{B} = B_1, \ldots, B_m \) of \( m \geq 1 \) pairwise-distinct equation blocks is an alternation-free block sequence given that

(i) \( X_{B_i} \cap X_{B_j} = \emptyset \) for \( i \neq j \); \hspace{1em} (ii) if \( i < j \), then \( B_i \) is not dependent on \( B_j \).

A formula \( \phi \in \mathcal{L} \) is dependent on \( \mathcal{B} \) if it is dependent of each block in the sequence.

**Example 2.5** Anticipating the semantics, the sequence of blocks in WMC can be used to encode, for instance, the formula \( A(\phi_1 U [r] \neg \phi_2) \) of WCTL: let \( \phi = X \) be dependent on the alternation-free sequence \( \mathcal{B} = B_1, B_2 \) defined as follows

\[
B_1 = \min \{ Y = (\phi_2 \land r \leq x \leq r') \lor (\phi_1 \land \neg a \in \Sigma [a] Y) \} \\
B_2 = \max \{ X = \neg a \in \Sigma [a] \land \land \phi_1 \}
\]

\( B_1 \) is a minimal equation block and \( B_2 \) is a maximal one. \( B_2 \) is dependent on \( B_1 \).

### 3 Weighted Semantics for WMC

To provide a semantics for WMC in terms of WTSs, we define the notions of resource valuation, extended states and environments.

A **resource valuation** is a function \( l : \mathcal{V} \to \mathbb{R}_{\geq 0} \) that assigns (non-negative) real numbers to the resource-variables in \( \mathcal{V} \). We denote by \( L \) the class of resource valuations. For \( l \in L \), \( x \in \mathcal{V} \) and \( s \in M \), let \( l(x \mapsto s) \in L \) be defined by \( l(x \mapsto s)(x) = s \) and \( l(x \mapsto s)(y) = l(y) \) for \( y \neq x \); let \( l + s \in L \) be defined by \( (l + s)(x) = l(x) + s \).

Given a WTS \( \mathcal{W} = (\mathcal{M}, \Sigma, \emptyset) \), \( m \in M \) and \( l \in L \), the pair \((m, l)\) is called an **extended state** of \( \mathcal{W} \). Transitions between extended states are defined by:

\[
(m, l) \to (m', l') \text{ iff } m \xrightarrow{a} m' \text{ and } l' = (l + u).
\]

Given a WTS \( \mathcal{W} = (\mathcal{M}, \Sigma, \emptyset) \), an **environment** is a function \( \rho : X \to 2^{M \times L} \) that interpret the recursive-variables as sets of extended states. We use 0 as the empty environment that associates \( \emptyset \) to all recursive-variables. Given an environment \( \rho \) and \( S \subseteq M \times L \), let \( \rho[X \mapsto S] \) be the environment that interprets \( X \) as \( S \) and all the other recursive-variables as \( \rho \) does. Similarly, for a pairwise-disjoint tuple \( \overline{X} = (X_1, ..., X_n) \in X^n \) and \( \overline{S} = (S_1, ..., S_n) \subseteq (M \times L)^n \), let \( \rho[\overline{X} \mapsto \overline{S}] \) be the environment that interprets \( X_i \) as \( S_i \) for all \( i = 1..n \) and all the other variables as \( \rho \) does.

Given a WTS \( \mathcal{W} = (\mathcal{M}, \Sigma, \emptyset) \) and an environment \( \rho \), the WTS-semantics for \( \mathcal{L} \) is defined, on top of the classic semantics for Boolean logic, as follows.

\[
\begin{align*}
\mathcal{W} \models (m, l), \rho \models s \iff l(s) \leq r; \\
\mathcal{W} \models (m, l), \rho \models [a] \phi \iff \text{for any } (m', l') \in M \times L \text{ s.t. } (m, l) \to (m', l'), \mathcal{W} \models (m', l'), \rho \models \phi; \\
\mathcal{W} \models (m, l), \rho \models x \phi \iff \text{for } \mathcal{W} \models (m, l[x \mapsto 0]), \rho \models \phi; \\
\mathcal{W} \models (m, l), \rho \models X \phi \iff \text{for } (m, l) \in \rho(X).
\end{align*}
\]

Let \( \llbracket \phi \rrbracket_{\rho} = \{(m, l) \in M \times L \mid \mathcal{W} \models (m, l), \rho \models \phi\} \).

Following [12, 13, 24], we extend now the semantics to include the restrictions imposed by an alternation-free sequence of blocks and obtain the so-called block-semantics.

Given a set of equations \( E \) with variables \( \overline{X} = (X_1, ..., X_n) \), an environment \( \rho \) and \( \overline{T} = (T_1, ..., T_n) \subseteq (M \times L)^n \), let the function \( f^\rho : (2^{M \times L})^n \to (2^{M \times L})^n \) be defined as follows:
Consider a weighted system that can perform three actions under any resource valuation and any environment.

We say that a formula \( \phi \) is \( B \)-validity, written \( \models_B \phi \), if it is satisfied in all states of any WTS under any resource valuation and any environment.

We observe that \((2^{M \times L})^n\) forms a complete lattice with the ordering, join and meet operations defined as the point-wise extensions of the set-theoretic inclusion, union and intersection, respectively. Moreover, for any \( E \) and \( \rho \), \( f_{E}^\rho \) is monotonic with respect to the order of the lattice and therefore, it has a greatest fixed point denoted by \( \nu X. f_{E}^\rho \) and a least fixed point denoted by \( \mu X. f_{E}^\rho \) [12]. These can be characterized as follows:

\[
\nu X. f_{E}^\rho = \bigcup \{ E | \langle X, \rho \rangle \subseteq f_{E}^\rho (\langle X, \rho \rangle) \}, \quad \mu X. f_{E}^\rho = \bigcap \{ E | \langle X, \rho \rangle \subseteq f_{E}^\rho (\langle X, \rho \rangle) \}.
\]

The blocks \( \max\{E\} \) and \( \min\{E\} \) define environments that satisfy all the equations in \( E \); \( \max\{E\} \) is the greatest fixed point and \( \min\{E\} \) is the least fixed point. The environment defined by the block \( B \) is denoted by \( \llbracket B \rrbracket_\rho \).

**Definition 3.1 [Block-Semantics]** Given an alternation-free block sequence \( B = B_1, \ldots, B_m \) and an environment \( \rho_0 \), let \( \rho_1, \ldots, \rho_m \) be defined by \( \rho_i = \llbracket B_i \rrbracket_{\rho_{i-1}} \) for \( i = 1, \ldots, m \). The semantics of \( B \) is then given by \( \llbracket B \rrbracket_{\rho_0} = \rho_m \).

We say that a formula \( \phi \) is \( B \)-satisfiable if there exists at least one WTS that satisfies it for the alternation-free block sequence \( B \) in one of its states under some resource valuation and some environment; \( \phi \) is a \( B \)-validity, written \( \models_B \phi \), if it is satisfied in all states of any WTS under any resource valuation and any environment.

### 4 Symbolic Semantics for WMC

Consider an weighted system that can perform three actions \( a, b \) and \( c \), and suppose that we are interested in the following specifications of the system:

1. It can do an \( a \)-action followed by an infinite sequence of alternations of the actions \( b \) and \( c \) with non-zero cost;
2. After an \( a \)-transition, the overall behaviour costs less than one unit of resource.

These requirements can be encoded in WMC, by using three resource-variables \( x_a, x_b \) and \( x_c \), as follows:

\[
\phi = \langle a \rangle (x_a \text{ in } X),
\]

\[
B = \max\{X = x_a < 1 \land \langle b \rangle (x_b \text{ in } (Y \land x_c > 0)), \quad Y = x_a < 1 \land \langle c \rangle (x_c \text{ in } (X \land x_b > 0))\}
\]

We can see that there exists a WTS satisfying \( \phi \) under the assumptions of \( B \). But it cannot be satisfied by a finite WTS, since it must have at least one infinite trace of non-zero cost transitions with a bounded overall cost. However, all the WTSs that satisfy the requirements encoded by \( \phi \) have something in common: the way the resource-variables behave under certain resource valuations and as a result of resetting.

This observation motivates the development of symbolic weighted transition systems (SWSs), which are similar to the ones used with timed automata in [2, 4, 23]. These are abstractions of WTSs: a symbolic model is a labelled transition system representing an
The equivalence classes under $N \in \mathbb{R}_\geq 0$ consisting of all the resource valuations of a tuple $[\Sigma]$. Given $N$, $l'$ is equivalent w.r.t. $N$, denoted by $l \equiv_N l'$ iff:

1. $\forall x \in \mathcal{V}$, $l(x) > N$ if $l'(x) > N$;
2. $\forall x \in \mathcal{V}$ s.t. $0 \leq l(x) \leq N$, $[l(x)] = [l'(x)]$ and $l(x) = 0 \equiv [l'(x)] = 0$;
3. $\forall x, y \in \mathcal{V}$ s.t. $0 \leq l(x), l(y) \leq N$, $(|[l(x)]| \leq |l(y)|) \equiv ([l'(x)] \leq [l'(y)])$.

The equivalence classes under $\equiv = \equiv_N$ are called $N$-regions. Let $[l]$ be the region containing $l$ and $\mathcal{R}_N$ be the set of all $N$-regions for the set $\mathcal{V}$ of resource-variables and the constant $N$. For a given $N \in \mathcal{R}_N$, the $\mathcal{R}_N$ is finite whenever $\mathcal{V}$ is finite.

For $\delta \in \mathcal{R}_N$, a successor region is the region $\delta'$ s.t. for any $l \in \delta$, there exists $d \in \mathbb{R}_\geq 0$ s.t. $l + d \in \delta'$, denoted by $\delta \xrightarrow{d} \delta'$. For $\delta \in \mathcal{R}_N$, $x \in \mathcal{V}$ and $n \in \mathbb{N}$, $\delta[x \mapsto n]$ denotes the region consisting of all the resource valuations $l$ for which there exists $l' \in \delta$ s.t. $l = [l'[x \mapsto n]$.

Example 4.2 In Figure 1 are represented some regions for $N = 1$ and $\mathcal{V} = \{x_a, x_b, x_c\}$.

$\delta_0 = [x_a = x_b = x_c = 0]$ $\delta_1 = [0 < x_a = x_b = x_c < 1]$
$\delta_2 = [x_a = 0, 0 < x_a = x_b < 1]$ $\delta_3 = [0 < x_a < x_b = x_c < 1]$
$\delta_4 = [x_a = 0, 0 < x_b < x_a < 1]$ $\delta_5 = [0 < x_b < x_c < x_a < 1]$
$\delta_6 = [x_b = 0, 0 < x_c < x_a < 1]$ $\delta_6 = [0 < x_b < x_c < x_a < 1]$

$\delta_0$ is a successor of $\delta_0$, $\delta_2 = \delta_1[x_b \mapsto 0]$ and $\delta_3$ is a successor of $\delta_2$. Similarly, $\delta_3$ is a successor of $\delta_3$ and $\delta_7$ is a successor of $\delta_6$. Moreover, $\delta_2 = \delta_3[x_b \mapsto 0]$, $\delta_4 = \delta_3[x_c \mapsto 0] = \delta_5[x_c \mapsto 0] = \delta_7[x_c \mapsto 0]$. $\delta_0 = \delta_3[x_b \mapsto 0] = \delta_7[x_b \mapsto 0]$. In what follows, we consider an extension of the concept of region to also include the case when $N = \dfrac{1}{q}$ with $p, q \in \mathbb{N}$. We firstly construct the regions for $p$ and then divide each of the resource-valuation in it by $q$ – the resulting set will be a region for $N = \dfrac{1}{q}$. For instance, if we take $N = \dfrac{1}{2}$ in Example 4.2, then $\delta_1 = [0 < x_a = x_b = x_c < \dfrac{1}{2}]$ and $\delta_2 = [x_b = 0, 0 < x_a = x_c < \dfrac{1}{2}]$ are regions in $\mathcal{R}_{1/2}$.

Definition 4.3 [Symbolic Model] Given $\mathcal{R}_N$ and a non-empty set $S$, a symbolic weighted transition system (SWS) is a tuple $\mathcal{W} = (\Pi^i, \Sigma^i, \theta^i)$ where $\Pi^i \subseteq S \times \mathcal{R}_N$ is a non-empty set of symbolic states, $\Sigma^i = \{e_x \mid x \in \mathcal{V}\} \cup \Sigma$ a non-empty set of actions, and $\theta^i : \Pi^i \times \Sigma^i \rightarrow 2^{\Pi^i}$ is a labeled transition function such that:

1) if $(s, \delta) \rightarrow_a (s', \delta')$ for $a \in \Sigma_a$, then $\delta \xrightarrow{\delta'}$; 2) if $(s, \delta) \rightarrow_{e_x} (s, \delta')$ then $\delta' = \delta[x \mapsto 0]$.
Note that if \((s, \delta) \rightarrow_{e_i} (s, \delta)\), then for any \(l \in \delta\), \(l(x) = 0\).

For a given SWS \(W^s = (\Pi^s, \Sigma^s, \theta^s)\), a symbolic environment is a function \(\rho^s : \mathcal{X} \rightarrow 2^{\Pi^s}\) which interprets the recursive-variables as sets of symbolic states.

The symbolic satisfiability relation \(\models^s\) is defined for the non-Boolean operators as follows.

\[
\begin{align*}
W^s, \pi, \rho^s \models^s \phi &\iff \text{for any } l \in \delta, l(x) \leq r; \\
W^s, \pi, \rho^s \models^s [a] \phi &\iff \text{for arbitrary } \pi' \in \Pi^s \text{ such that } \pi \rightarrow_a \pi', \text{ we have } W^s, \pi', \rho^s \models^s \phi; \\
W^s, \pi, \rho^s \models^s \phi &\iff \text{if there exists } \pi' \in \Pi^s \text{ such that } \pi \rightarrow_{e_i} \pi' \text{ and } W^s, \pi', \rho^s \models^s \phi; \\
W^s, \pi, \rho^s \models^s X &\iff \pi \in \rho^s(X).
\end{align*}
\]

Similarly as in Section 3, for a given alternation-free sequence of blocks \(\mathcal{B}\) we can define the symbolic \(\mathcal{B}\)-semantics based on the \(\mathcal{B}\)-satisfiability relation \(\models^s_{\mathcal{B}}\), as follows:

\[
W^s, \pi, \rho \models^s_{\mathcal{B}} \phi \iff W^s, \pi, \llbracket \mathcal{B} \rrbracket^s_{\rho} \models^s \phi.
\]

5 The Equivalence of the Two Semantics

In this section we prove that the two semantics introduced for WMC are equivalent, in the sense that the set of the WTS-validities coincides with the set of the SWS-validities. This result has important consequences: (i) if the satisfiability problem is decidable for one semantics, then it is also decidable for the other; and (ii) an axiomatization that is sound and complete for one semantics is sound and complete also for the other semantics. To prove the equivalence, we show that for any formula \(\phi \in \mathcal{L}\) dependent on \(\mathcal{B}\), if \(\phi\) has a WTS-model, then we can also construct an SWS-model for it; and conversely, if it has an SWS-model, then we can construct a WTS-model for it.

Construction A: Given a WTS \(W = (M, \Sigma, \theta)\) and \(\mathcal{R}^V_N\), we construct the SWS \(W^s = (\Pi^s, \Sigma^s, \theta^s)\), where \(\Pi^s = M \times \mathcal{R}^V_N\), \(\Sigma^s = \{e_i \mid x \in \mathcal{V}\} \cup \Sigma\) and \(\theta^s\) is defined as follows:

1. \((m, [l]) \rightarrow_a (m', [l'])\) iff \((m, l) \rightarrow_a (m', l')\);
2. \((m, [l]) \rightarrow_{e_i} (m, [l'])\) iff \([l'] = [l][x \mapsto 0]\).

We call \(W^s\) the symbolic model of \(W\) w.r.t. \(\mathcal{R}^V_N\), denoted by \(\mathbb{S}(W, \mathcal{R}^V_N)\).

Construction B: Given an SWS \(W^s = (\Pi^s, \Sigma^s, \theta^s)\) on \(\mathcal{R}^V_N\) with \(\Sigma^s = \{e_i \mid x \in \mathcal{V}\} \cup \Sigma\), let \(W = (M, \Sigma, \theta)\) be a WTS s.t.

- the states are sets of type \((s, \delta_1, l_1), \ldots, (s, \delta_k, l_k)\) where \((1) \ (s, \delta_i) \in \Pi^s\) and \(l_i \in \delta_i; \ (2)\) for any \(i \in \{1, \ldots, k\}\) there exist \(j \in \{1, \ldots, k\}\) and \(x \in \mathcal{V}\) s.t. either \(\delta_j = \delta_i[x \mapsto 0]\) and \(l_j = l_i[x \mapsto 0]\), or \(\delta_i = \delta_j[x \mapsto 0]\) and \(l_i = l_j[x \mapsto 0]\).
- \(\theta\) is defined for any \(m_1, m_2 \in M\), \(m_1 \stackrel{a}{\rightarrow}_u m_2\) iff there exist \((s_1, \delta_1, l_1) \in m_1\) and \((s_2, \delta_2, l_2) \in m_2\) s.t. \((s_1, \delta_1) \rightarrow_u (s_2, \delta_2)\) and \(l_2 = (l_1 + u)\).

We call \(W\) the concrete model of \(W^s\) on \(\mathcal{R}^V_N\), denoted by \(\mathbb{C}(W^s, \mathcal{R}^V_N)\).

We prove that the constructions preserve the \(\mathcal{B}\)-satisfiability of WMC properties, i.e., a formula \(\phi\) is \(\mathcal{B}\)-satisfiable in the WTS-semantics iff it is \(\mathcal{B}\)-satisfiable in the SWS-semantics.

Consider an arbitrary formula \(\phi \in \mathcal{L}\) dependent on \(\mathcal{B}\).

- Let \(\mathcal{V}[\phi, \mathcal{B}]\) be the set of the resource-variables in \(\phi\) and \(\mathcal{B}\). For any \(x \in \mathcal{V}[\phi, \mathcal{B}]\), let \(Q[\phi, \mathcal{B}] \subseteq \mathbb{Q}_{\geq 0}\) be the set of all \(r \in \mathbb{Q}_{\geq 0}\) that occur in a construct of type \(x \leq r\) in \(\phi\) or \(\mathcal{B}\).
- Let \(g\) be the least common denominator of the elements of \(Q[\phi, \mathcal{B}]\).
- Let \(\mathcal{R}[\phi, \mathcal{B}]\) denote the set \(\mathcal{R}^V_{\mathcal{R}[\phi, \mathcal{B}]}/g\)-regions, where \(g = \max Q[\phi, \mathcal{B}]\).
Theorem 5.1 Let $\phi$ depending of the alternating-free sequence of blocks $B = B_1, \ldots, B_m$.  
1. If $\mathcal{W}, (m, l), \rho \models_B \phi$, then $\mathcal{W}', (m, [l]), \rho' \models_{B'} \phi$, where $\mathcal{W}' = \Xi(\mathcal{W}, \mathcal{R}[\phi, B])$ and $\rho'(X) = [(m, [l]) | (m, l) \in \rho(X)]$ for any $X \in \mathcal{X}$.
2. If $\mathcal{W}', (s, \delta, l), \rho' \models_{B'} \phi$, then $\mathcal{W}, (m, l), \rho \models_B \phi$, where $\mathcal{W} = \Xi(\mathcal{W}', \mathcal{R}[\phi, B])$, $m \in \mathcal{M}, (s, \delta, l) \in m$ and for any $X \in \mathcal{X}$, $\rho(X) = [(m, l) | (s, \delta) \in \rho'(X), (s, \delta, l) \in m]$.

Consequently, the $B$-validities for WTC-semantics coincide with that of SWS-semantics.

6 Decidability and finite symbolic model property

In this section, we prove that WMC enjoys the finite model property against the SWS-semantics, by involving the region construction technique and adapting the classical tableau method. A consequence of this result is that the $B$-satisfiability problem for the SWS-semantics is decidable. In the light of Theorem 5.1, this means that $B$-satisfiability is decidable also for the WTS-semantics even if, as we have emphasized in Section 4, WMC does not enjoy the finite model property for the WTS-semantics.

Given $\phi \in \mathcal{L}$ that depends on an alternation-free sequence $B$, let $\Sigma[\phi, B]$ be the set of all actions $a \in \Sigma$ that appears in some transition modality of type $\langle a \rangle$ or $\{ a \}$ in $\phi$ or $B$; let $Q[\phi]$ and $\mathcal{R}[\phi]$ be defined as in Section 5. Observe that $\Sigma[\phi]$, $Q[\phi]$ and $\mathcal{R}[\phi]$ are finite or empty.

We fix $\phi_0 \in \mathcal{L}$ dependent on $B_0$. Let $\mathcal{L}[\phi_0, B_0]$ be the set of the sub-formulas of $\phi_0$ or $B_0$. Let $\Omega[\phi_0, B_0] \subseteq 2^{\mathcal{L}[\phi_0, B_0]} \times \mathcal{R}[\phi_0, B_0]$. Since $\mathcal{L}[\phi_0, B_0]$ and $\mathcal{R}[\phi_0, B_0]$ are both finite, $\Omega[\phi_0, B_0]$ is finite. We construct a tableau for $\phi_0$, which is similar to the standard construction with extra focus on the quantities.

The nodes of a tableau are pairs $(\Delta, \delta) \in \Omega[\phi_0, B_0]$ and the tableau rules are listed in Table 1, where $[\phi, \Delta]$ denotes $\{ \phi \} \cup \Delta$.

<table>
<thead>
<tr>
<th>Tableau System $T^\phi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\phi] \cup { \langle a \rangle</td>
</tr>
</tbody>
</table>

Because of the quantitative requirements must also be satisfied, not any pair $(\Delta, \delta)$ is a node in the tableau. A tableau $T(\phi, \delta)$ derived from the previous rules must be region consistent, meaning that any node $(\Delta, \delta') \in T(\phi, \delta)$ must satisfy the following conditions:

(i) for any $x \leq r \in \Delta$ and $l \in \delta$, $l(x) \leq r$;
(ii) if $\{ l(x) \phi, \Delta \}, \delta$ is the conclusion and $\{ [\phi], \Delta, \delta' \}$ is the assumption of (Res), then $\delta' = \delta[x \mapsto 0]$;
(iii) if $(\Delta, \delta)$ is the conclusion of (Mod), then $\delta \xrightarrow{\Delta} \delta'$ for any assumption $(\Delta', \delta')$.

If (Mod) is applied for an action $a$ at the node $t$, the node obtained is called an $\langle a \rangle$-son of $t$. The tableaux may be infinite. However, because $\Omega[\phi_0]$ is finite, the pairs from $\Omega[\phi_0]$ that appear in $T(\phi, \delta)$ are finitely many.
As in the classic method for mu-calculus [20, 31, 32], we use max-trace, min-trace to capture the idea of a history of the regeneration of a formula and markings, consistent markings to characterize B-satisfiability of a formula in a state of an SWS (these classic definitions can be found in the appendix).

**Lemma 6.1** \( \phi_0 \) is satisfied at state \( \pi_0 = (s_0, \delta_0) \) in an SWS \( \mathcal{W}' = (\Pi', \Sigma', \theta') \) if and only if there is a consistent marking of \( T(\phi_0, \delta_0) \) respect to \( \mathcal{W}' \) and \( \pi_0 \).

The proof of Lemma 6.1 relies on notion of signature, similar to that considered by Streett and Emerson [31]. These notions come from the characterization of fixed point formulas by means of transfinite chains of approximations, which have been extended to the setting with fixed points defined with blocks in [12, 13]. Involving these, the previous lemma is proven similarly to the case of classic \( \mu \)-calculus [20, 31, 32]. The correctness of the cases with weight is guaranteed by the region consistency.

This lemma allows us to prove the finite model property for SWS-semantics, by following the classic proof strategy of [20]; the only difference consists in managing the reset actions.

**Theorem 6.2 (Finite Symbolic Model Property)** Let \( \phi_0 \in \mathcal{L} \) be a formula that depends of \( \mathcal{B}_0 \). If \( \phi_0 \) is \( \mathcal{B}_0 \)-satisfiable, then there exists a finite SWS \( \mathcal{W}^f_\phi = (\Pi^f_\phi, \Sigma^f_\phi, \theta^f_\phi) \) with \( \pi_f \in \Pi^f_\phi \) and a symbolic environment \( \rho^f_\phi \) such that \( \mathcal{W}^f_\phi, \pi_f, \rho^f_\phi \models_\mathcal{B}_0 \phi_0 \).

According to Lemma 6.1 and Theorem 6.2, we can have an algorithm to decide the satisfiability of a given WMC formula. The following example shows how this works.

**Example 6.3** Suppose that we want to verify the \( \mathcal{B} \)-satisfiability of the property discussed at the beginning of Section 4.

\[
\phi = (a)(x_a \text{ in } X),
B = \max\{X = x_a < 1 \land (b)(x_b \text{ in } (Y \land x_c > 0)), Y = x_a < 1 \land (c)(x_c \text{ in } (X \land x_b > 0))\}.
\]

In Figure 2 shows \( T(\phi, \delta_0) \). There is only one infinite trace – max-trace. We construct \( \mathcal{W}^s \):

\[
\Sigma^s = \{a, b, c, e_{x_a}, e_{x_b}, e_{x_c}\}, \Pi^s = \{(t_0, \delta_0), (t_3, \delta_0), (t_6, \delta_1), (t_6, \delta_2), (t_9, \delta_3), (t_9, \delta_4), (t_12, \delta_5)\},
\]

![Fig. 2. Tableau T(\phi, B)](image)

![Fig. 3. SWS for \( \phi \) dependent on \( B \)](image)
In order to state the axioms for WMC we need to establish some notations.

\[ l_0 = (0, 0, 0) \]
\[ l_1 = (0.3, 0.3, 0.3) \]
\[ l_2 = (0.3, 0.0, 0.3) \]
\[ l_3 = (\frac{0}{3}, \frac{0}{3}, \frac{0}{3}) \]
\[ l_4 = (\frac{0}{3}, 0, \frac{0}{3}) \]
\[ l_5 = (0.1, 0.1, 0.1) \]
\[ l_6 = (0.1, 0.0, 0.1) \]
\[ l_7 = (0.5, 0.2, 0.5) \]
\[ l_8 = (0.5, 0.2, 0.0) \]
\[ l_9 = (0.3 + \frac{0}{3}, \frac{0}{3}, 0.3 + \frac{0}{3}) \]
\[ l_{10} = (0.3 + \frac{0}{3}, 0, \frac{0}{3}) \]
\[ l_{11} = (0.3, 0.2, 0.3) \]
\[ l_{12} = (0.3, 0.2, 0.0) \]
\[ l_{13} = (0.6, 0.3, 0.1) \]
\[ l_{14} = (0.6, 0.6, 0.1) \]
\[ l_{15} = (0.5, 0.4, 0.2) \]
\[ l_{16} = (0.5, 0.5, 0.2) \]
\[ l_{17} = (0.75, 0.15, 0.25) \]
\[ l_{18} = (0.75, 0.15, 0.0) \]
\[ l_{19} = (0.6, 0.1, 0.3) \]
\[ l_{20} = (0.6, 0.1, 0.0) \]

It is not difficult to verify that it is a model for \( \phi \).

**Theorem 6.4 (Decidability of \( B \)-Satisfiability)** For any alternation-free block sequence \( B \), the \( B \)-satisfiability problem for WMC is decidable for both WTS- and SWS-semantics.

## 7 Axiomatization

In this section, we focus on developing a sound and complete axiomatization for the validities of WMC with respect to the two semantics. Recall that the two sets of validities coincide. In the light of Theorem 5.1, it is sufficient to find such an axiomatization for the SWS-semantics and it is then sound and complete also for the WTS-semantics.

### 7.1 Sound axiomatization

In order to state the axioms for WMC we need to establish some notations.

- The **modal prefixes** are words \( w \in \text{Mod}^* \) over the alphabet of modal operators of \( \mathcal{L} \), \( \text{Mod} = \{ [a] | a \in \Sigma \} \cup \{ x \in \mathcal{V} | x \in \mathcal{V} \} \). E.g., \( [a], x \in [a][a], [a]x \in \mathcal{C} \), \( \in \text{Mod}^* \).
- A **context** \( C \) is a word formed by a modal prefix \( w \in \text{Mod}^* \) concatenated with the metavariable \( \mathcal{X} \); e.g., \( [a][a], x \in [a][a][a], [a]x \in \mathcal{X} \), \( [a]x \in \mathcal{X} \) are contexts. To emphasize the presence of the metavariable we will use the functional representation of type \( C[\mathcal{X}] \) for contexts; this will allow us to instantiate the metavariable with elements from \( \mathcal{L} \). E.g.,
Given a maximal equation block \( B = \max\{X_1 = \phi_1, \ldots, X_n = \phi_n\} \) and an arbitrary classical deducibility relation \( \vdash^* \), we define the deducibility relation \( \vdash^B \) as the extension of \( \vdash^* \) given by the axioms and rules in Table 3, which are the equation-version of the fixed points axioms of Mu-calculus [20, 28, 30]. These are stated for arbitrary \( \phi \in \mathcal{L} \) and \( \Psi = (\psi_1, \ldots, \psi_n) \in \mathcal{L}^n \), where \( \mathcal{X} = (X_1, \ldots, X_n) \). Similarly, we define a classical deducibility relation \( \vdash^B \) for a minimal equation block \( B = \min\{X_1 = \phi_1, \ldots, X_n = \phi_n\} \) based on \( \vdash^* \) by using the axioms and rules in Table 4.
1. we construct a canonical model. The construction will go as follows: $L$ corresponds to a function from the set of valuations of states in the canonical WTS model. This is because any state in a given WTS corresponds to a function $\Theta$. The model construction is not standard, in the sense that we will not use the state replacement approach.

For some set $S$ depending on a fixed alternation-free sequence $B = B_1, ..., B_m$, we define the classical deducibility relations $\vdash_{B_0}, \vdash_{B_1}, ..., \vdash_{B_m}$ as follows: $\vdash_{B_0} = \vdash$, $\vdash_{B_{i-1}} \vdash_{B_i}$ for $i = 1, ..., m$. Consequently, $\vdash_B = \vdash_{B_m}$.

As usual, we say that a formula $\phi$ (or a set $\Phi$ of formulas) is $B$-provable, denoted by $\vdash_B \phi$ (respectively $\vdash_B \Phi$), if it can be proven from the given axioms and rules of $\vdash_B$. We denote by $\Psi = \{\phi \in L | \phi \vdash_B \phi\}$.

An induction on the structure of the alternation-free blocks shows that all the theorems of $\vdash_B$ are sound in the WTS-semantics, hence also in the SWS-semantics.

**Theorem 7.2 (Extended Soundness)** The axiomatic system of $\vdash_B$ is sound with respect to the semantics based on WTSs, i.e., for arbitrary $\phi \in L$,

$\vdash_B \phi$ implies $\vdash B \phi$.

### 7.2 Completeness

In the rest of this section we prove that the axiomatic system of $\vdash_B$ is not only sound, but also complete for the two semantics, meaning that all the $B$-validities can be proved, as theorems, from the proposed axioms and rules, i.e., for arbitrary $\phi \in L$, $\vdash_B \phi$ implies $\vdash_B \phi$. To complete this proof it is sufficient to show that any $B$-consistent formula has a model.

For some set $S \subseteq L$, $\Phi$ is $(S, B)$-maximally consistent if it is $B$-consistent and no formula of $S$ can be added to $\Phi$ without making it inconsistent. $\Phi$ is $B$-maximally-consistent if it is $(L, B)$-maximally-consistent.

In the following we fix a consistent formula $\phi_0$ depending on a fixed alternation-free sequence $B_0$ and we construct a model. Let $\Theta$ be the set of $B_0$-maximally consistent sets.

The model construction is not standard, in the sense that we will not use $\Theta$ as the set of states in the canonical WTS model. This is because any state in a given WTS corresponds to a function from the set of valuations $L$ to $\Theta$: each resource valuation identifies a $B_0$-maximally-consistent set of formulas satisfied by that model under the given resource valuation. Consequently, to construct the canonical model we will need to take as states not $B_0$-maximally-consistent sets of formulas (as usual in modal logics), but some particular functions from $L$ to $B_0$-maximally-consistent sets, called coherent functions. Then, the construction will go as follows:

1. we construct a canonical model which takes coherent functions as states, similar to the construction made in [18] for timed logic;
2. we construct an SWS from the above model and prove the truth lemma, where the symbolic finite model property is used;
3. according to Theorem 5.1, there exists a WTS for any $B_0$-consistent formula.

**Lemma 7.3** For arbitrary $\Lambda \in \Theta$ and $x \in V$,

$$\sup\{r \in \mathbb{Q}^+ | x \geq r \in \Lambda\} = \inf\{r \in \mathbb{Q}^+ | x \leq r \in \Lambda\} \in \mathbb{R}_{\geq 0}.$$
The previous lemma demonstrates that each $B_0$-maximally-consistent set corresponds to a unique resource valuation of resource-variables, that we will identify using the function $\mathcal{F}: \Theta \rightarrow L$ defined for arbitrary $\Lambda \in \Theta$ and $x \in \mathcal{V}$ by:

$$\mathcal{F}(\Lambda)(x) = \sup\{r \in \mathbb{Q}^+ \mid x \geq r \in \Lambda\} \in \mathbb{R}_{\geq 0}.$$ 

Since $\mathcal{F}(\Lambda)$ synthesizes only the information regarding the resource-variables, there exist distinct sets $\Lambda_1, \Lambda_2 \in \Theta$ s.t. $\mathcal{F}(\Lambda_1) = \mathcal{F}(\Lambda_2)$; this defines an equivalence relation on $\Theta$ and the equivalence classes are in one to one correspondence with the resource valuation in $L$.

Observe that not just any function $\gamma: L \rightarrow \Omega$ is a good candidate for becoming a state in the canonical model. To better understand this, we emphasize the essential role of resource valuations in the semantics of WMC. We start from analyzing how the formulas satisfied by a given WTS under a certain resource valuation change with the change of the valuation.

Let $\mathcal{F}(\phi)$ be the set of the free resource-variables in $\phi \in L$ (i.e., those that are not bounded by reset operator $x \in \mathcal{V}$) defined by: $\mathcal{F}(\bot) = \mathcal{F}(X) = \emptyset$, $\mathcal{F}(x \leq r) = \{x\}$, $\mathcal{F}(\phi \lor \psi) = \mathcal{F}(\phi) \cup \mathcal{F}(\psi)$, $\mathcal{F}(-\phi) = \mathcal{F}([-a] \phi) = \mathcal{F}(\phi)$, $\mathcal{F}(x \in \phi) = \mathcal{F}(\phi) \setminus \{x\}$. Similarly, we denote the set of the free resource-variables in $\phi_0$ and $\mathcal{B}$ by $\mathcal{F}([\phi_0, \mathcal{B}])$.

For $y \in \mathcal{V}$ that does not appear in the syntax of $\phi$ and $x \in \mathcal{F}(\phi)$, we denote by $\phi[y/x]$ the formula obtained by uniformly substituting all the occurrences of $x$ in $\phi$ by $y$.

**Definition 7.4** Let $f_-, f_+: \mathcal{V} \rightarrow \mathbb{Q}$ be two rational resource valuations. For any formula $\phi \in L$, let $\phi^+ \vdash \phi$ be defined as follows, where $x \leq t$ for $t < 0$ should be read as $x \geq 0$:

- $\bot \vdash \bot$
- $(\phi \lor \psi)^+ \vdash (\phi^+ \lor \psi^+)$
- $(x \leq r)^+ \vdash x \leq (r + f_+(x))$
- $(-\phi)^+ \vdash -(\phi^+)$
- $(\phi \lor \psi)^+ \vdash (\phi^+ \lor \psi^+)$
- $(\phi \lor \psi)^+ \vdash (\phi^+ \lor \psi^+)$
- $(x \in \phi)^+ \vdash x \in (\phi^+)$
- $\mathcal{V} \vdash X$

Given a list of equations $E = (X_1 = \phi_1, \ldots, X_n = \phi_n)$, let $E^+ \vdash \phi$ be defined as $E \vdash \phi$, let $E^+ = (X_1 = \phi_1^+, \ldots, X_n = \phi_n^+)$.

Given an equation block $B = \max\{E\} \text{ or } \min\{E\}$, we define $B^+ \vdash E$, to be $\max\{E^+\}$ or $\min\{E^+\}$ respectively. Given an alternation-free block sequence $\mathcal{B} = B_1, \ldots, B_m$, let $\mathcal{B}^+ \vdash E^+ = B_1 +^+ \ldots, B_m +^+$.

Whenever $f_- = f_+ = f$, we write $+ f$.

For $S \subseteq L$ and $\delta: \mathcal{V} \rightarrow \mathbb{R}$, let

$$S \sqsupseteq \delta = \{\phi^+ \vdash \phi \mid \phi \in S, f_-, f_+ : \mathcal{K} \rightarrow \mathbb{Q} \text{ s.t. } f_- < \delta < f_+\}.$$ 

**Definition 7.5** [Coherent function] A function $\gamma: L \rightarrow \Theta$ is coherent, if for any $l, l' \in L$,

1. $(\mathcal{F} \circ \gamma)(l) = l$;
2. $\gamma(l) \sqsupseteq (l' - l) \subseteq \gamma(l')$.

The first fundamental result is that any $B_0$-maximally-consistent set $\Lambda$ belongs to the image $\gamma(L)$ of a coherent function $\gamma$. Eventually, we will construct a symbolic model from the WTS on the set of coherent functions, and this result will guarantee that any $B_0$-maximally-consistent set is satisfied.

**Lemma 7.6** For any $\Lambda \in \Theta$, there exists a coherent function $\gamma$ such that $\gamma(\mathcal{F}(\Lambda)) = \Lambda$.

Firstly, we define a WTS using the state space

$$\Gamma = \{\gamma: L \rightarrow \Theta \mid \gamma \text{ is a coherent function}\}$$
and the transitions defined by
\[ \gamma \xrightarrow[^u]{a} \gamma' \] if \([\forall l \in L, [a] \phi \in \gamma(l) \Rightarrow \phi \in \gamma'(l + u)]\).

Secondly, we apply Construction A from Section 5 and construct a SWS \(W' = (\Pi^s, \Sigma^s, \theta')\) for the above WTS w.r.t. \(\phi_0\) that depends on \(\mathcal{B}_0\), for a set of regions \(\mathcal{R}[\phi_0, \mathcal{B}_0]\). We get \(\Pi^s = \Gamma \times \mathcal{R}[\phi_0, \mathcal{B}_0], \Sigma^s = \Sigma[\phi_0, \mathcal{B}_0] \cup \{e_0 \mid x \in \mathcal{V}[\phi_0, \mathcal{B}_0]\}\) and
1. \((\gamma, [l]) \xrightarrow{a} (\gamma', [l'])\) iff \(\gamma \xrightarrow{a} \gamma'\) and \(l' = l + u\); 2. \((\gamma, [l]) \xrightarrow{e_i} (\gamma', [l'])\) iff \([l'] = [l][x \mapsto 0]\).

Let \(\mathcal{L}[\phi_0, B]\) be defined as:
\[
\mathcal{L}[\phi_0, B] = \{\phi \in \mathcal{L} \mid [\Sigma[\phi, B] \subseteq [\Sigma[\phi_0], \mathcal{Q}[\phi, B] \subseteq \mathcal{Q}[\phi_0, B]\}. 
\]

Let \(\rho_0^i\) be the symbolic environment defined for any \(X \in \mathcal{X}\), by \(\rho_0^i(X) = \{(\gamma, [l]) \mid X \in \gamma(l)\}\).

Firstly, we prove the restricted truth lemma that does not consider recursive constructs. Its proof is similar to the proof presented in [18] for timed modal logic.

**Lemma 7.7 (Restricted Truth Lemma)** For \(\phi \in \mathcal{L}[\phi_0, \mathcal{B}_0], l \in L\) and \(\pi = (\gamma, [l]) \in \Pi^s, W'^s, \pi, \rho_0^i \models \phi \iff \phi \in \gamma(l)\).

On the restricted truth lemma we can base the following two results that indicate how we can extend the results to include the recursive cases.

**Lemma 7.8** Let \(B = \max\{X_1 = \phi_1, \ldots, X_n = \phi_n\}\) be an equation block in the sequence \(\mathcal{B}_0\) and \(\rho^s\) a symbolic environment such that \(\rho^s(X_i) = \{(\gamma, [l]) \mid X_i \in \gamma(l)\}\) for any \(i = 1, \ldots, n\). For any \(\phi \in \mathcal{L}[\phi_0, \mathcal{B}_0], l \in L\) and \(\pi = (\gamma, [l]) \in \Pi^s, \)
\(\phi \models^{\rho^s} \phi \iff \phi \models^{\rho_0^i} \phi \iff \phi \in \gamma(l)\).

**Proof.** Induction on \(\phi\). We prove here the case of the recursive-variables \(X_i, i = 1, \ldots, k\).

\((\Rightarrow)\) Because WMC enjoys the finite symbolic model property, there exists a finite ordinal \(k_0\) s.t. for all \(i = 1, \ldots, n\), \(W'^s, \pi, [B] \models^\rho X_i \iff W'^s, \pi, [\rho^s] \models^{\rho_0^i} \phi_i\), where all \(i = 1, \ldots, n\), \(\phi_i\) are defined simultaneously by \(\phi_0^i = \bot\) and \(\phi_0^i = \phi_i(\overrightarrow{X}/\overrightarrow{X}),\) where \(\overrightarrow{X} = (X_1, X_2, \ldots)\). It is clear that in \(\phi_i\), there is no recursive-variable from \(\{X_1, \ldots, X_n\}\). For any recursive-variable \(X\) other than \(X_1, \ldots, X_n\), \([B] \models^{\rho^s} X \iff X_i \models^{\rho^s}\). Hence, \(W'^s, \pi, [B] \models^\rho X_i \iff X_i \models^{\rho^s}\). Then, \(\phi_i \in \gamma(l)\).

The finite symbolic model property also guarantees that for any \(\pi' = \Gamma\) and any \(i = 1, \ldots, n,\)
\(W'^s, \pi', \rho^s \models^{\rho_0^i} \phi_i(\overrightarrow{X}/\overrightarrow{X})\).
So, for any \(i = 1, \ldots, n, \phi_i \rightarrow \phi_i(\overrightarrow{X}/\overrightarrow{X}) \in \gamma(l)\) for any \(\gamma', [l']\) \(\in \Gamma\). This further implies that \(\Gamma \cup \{\phi_i \rightarrow \phi_i(\overrightarrow{X}/\overrightarrow{X})\}\) is present in all the maximal-consistent sets. Hence, using (max-R2), for any \(i, \phi_i \rightarrow X_i \in \gamma(l)\) for any \(\gamma', [l']\) \(\in \Gamma\).

As already proven above, \(W'^s, \pi, [B] \models^\rho X_i \iff X_i \models^{\rho_0^i} \phi_i \in \gamma(l)\). Together with \(\phi_i \rightarrow X_i \in \gamma(l)\) for any \(\gamma', [l']\) \(\in \Gamma\), provided by (max-A1), we get that \(X_i \in \gamma(l)\).

\((\Leftarrow)\) We prove that \(\rho^s\) is a post-fixed point of \(B\) as follows:

For any \(X_i, i = 1, \ldots, n\), suppose \(W'^s, \pi, \rho^s \models X_i\). Then \(X_i \in \gamma(l)\), which implies that
\[ \phi_i \in \gamma(l) \text{ by (max-A1). So } W', \pi, \rho^s \models \phi_i. \text{ Since } \rho^s \subseteq \llbracket B \rrbracket_{\rho^s}. \text{ Therefore, } W', \pi, \rho^s \models \phi \text{ implies } W', \pi, \llbracket B \rrbracket_{\rho^s} \models \phi. \]

Since the minimal blocks are dual of the maximal blocks, we have a similar lemma for minimal blocks.

**Lemma 7.9** Let \( B = \min\{X_1 = \phi_1, \ldots, X_n = \phi_n\} \) be an equation block in the sequence \( B_0 \) and \( \rho^s \) a symbolic environment such that \( \rho^s(X_i) = (\gamma_i, [l]) \mid X_i \in \gamma(l) \) for any \( i = 1, \ldots, n \). For any \( \phi \in \mathcal{L}[\phi_0, B_0], l \in L \) and \( \pi = (\gamma, [l]) \in \Pi^s \), if \( \llbracket W', \pi, \rho^s \rrbracket \models \phi \iff \phi \in \gamma(l) \), then \( \llbracket W', \pi, \llbracket B \rrbracket_{\rho^s} \models \phi \iff \phi \in \gamma(l) \).

These lemmas allow us to prove the stronger version of the truth lemma.

**Theorem 7.10 (Extended Truth Lemma)** For \( \phi \in \mathcal{L}[\phi_0, B_0], l \in L \) and \( \pi = (\gamma, [l]) \in \Pi^s \),

\[ \llbracket W', \pi, \rho^s \rrbracket \models \phi \iff \phi \in \gamma(l), \quad \llbracket W', \pi, \llbracket B \rrbracket_{\rho^s} \models \phi \iff \phi \in \gamma(l). \]

A direct consequence of Theorem 7.10 is the completeness\(^3\) of the axiomatic system.

**Theorem 7.11 (Completeness)** The axiomatic system \( \vdash_B \) is complete with respect to the WTS-semantics, i.e., for arbitrary \( \phi \in \mathcal{L} \),

\[ \models_B \phi \text{ implies } \vdash_B \phi. \]

### 8 Conclusions

In this paper we have investigated the alternation-free weighted mu-calculus (WMC) for which we presented two semantics: one based on weighted transition systems (WTSs) and one based on the symbolic models (SWSs). We have demonstrated that the two semantics are equivalent in the sense that the WTS-validities coincide with the SWS-validities. This is a remarkable result that allows us to transport metareresults between the two semantics.

We firstly proved that even if WMC does not enjoy the finite model property for the WTS-semantics, it enjoys it for the SWS-semantics and thus we prove that satisfiability is decidable in both cases. To prove this we involve the tableau method. We suspect that a similar result can be extended to the entire weighted Mu-Calculus without the alternation-free restriction, but for now we have no evidence in this sense.

The finite model property is also used to prove that the axiomatization that combines modal axioms of weighted logic with the axioms of fixed points is complete for the SWS-semantics. Since the SWS-validities coincide with the WTS-validities, the completeness result can be extrapolated for the TWS-semantics.

The development of symbolic semantics that induces the same validities as the classic semantics is a powerful tool with potential applications also in other contexts. We intend to further apprehend these results to understand if some general technique can be proposed.

\(^3\) In this context by completeness we mean the weak-completeness. Since WMC is not compact, the weak- and strong-completeness do not coincide.

193
References


Appendix

Related definitions for the tableau method

**Definition** [Trace] Given a path $P = t_0 t_1 \ldots$ of a tableau $T(\phi, \delta)$, a trace on $P$ is a function $T$ assigning a formula to every node $t = (\Delta, \delta)$ in some initial segment of $P$ (possibly to all of $P$), satisfying the following conditions:
- (i) if $T(t)$ is defined, $T(t) \in \Delta$;
- (ii) if $T(t)$ is defined and $t' \in P$ is a son of $t$; if a rule applied at $t$ does not reduce the formula $T(t)$ then $T(t') = T(t)$; if $T(t)$ is reduced in $t$ then $T(t')$ is one of the results of the reduction.

We say that there is a regeneration of a recursive-variable $X$ on a trace $T$ on some path of a tableau, if for $t$ and its son $t'$ on the path, $T(t) = X$ and $T(t') = \phi$, where $X = \phi \in B$.

**Definition** [max-Trace and min-Trace] We call a trace a max-trace iff it is an infinite trace (defined for the whole path) on which the recursive-variable regenerated infinitely often is a max-variable.

Similarly, a trace will be called a min-trace iff it is an infinite trace where the recursive-variable regenerated infinitely often is a min-variable.

Every infinite trace is either a max-trace or a min-trace; all the rules except (Reg) decrease the size of formulas; hence, every formula is eventually reduced.

**Definition** [Marking] For a tableau $T(\phi, \delta)$, we define its marking with respect to an SWS $\mathcal{W}^\prime = (\Pi^\prime, \Sigma^\prime, \theta^\prime)$ and state $\pi_0 \in \Pi^\prime$ to be a relation $\mathcal{M} \subseteq \Pi^\prime \times T(\phi, \delta)$ satisfying the following conditions:
- (i) $(\pi_0, t_0) \in \mathcal{M}$, where $t_0$ is the root of $T(\phi, \delta)$;
- (ii) if some pair $(\pi, t) \in \mathcal{M}$ and a rule other than (mod) was applied at $t$, then for some son $t'$ of $t$, $(\pi, t') \in \mathcal{M}$;
- (iii) if $(\pi, t) \in \mathcal{M}$ and rule (mod) was applied at $t$, then for every action $a$ for which exists $\langle \! \langle a \rangle \! \rangle \psi \in \Delta(t)$:
  - (a) for every $\langle a \rangle$-son $t'$ of $t$, there exists a state $\pi'$ s.t. $\pi \xrightarrow{a} \pi'$ and $(\pi', t') \in \mathcal{M}$, and
  - (b) for every state $\pi$ s.t. $\pi \xrightarrow{a} \pi'$, there exists a $\langle a \rangle$-son $t'$ of $t$ s.t. $(\pi', t') \in \mathcal{M}$.

**Definition** [Consistent Marking] A marking $\mathcal{M}$ of $T(\phi, \delta)$ is consistent with respect to $\mathcal{W}^\prime = (\Pi^\prime, \Sigma^\prime, \theta^\prime)$ and $\pi \in \Pi^\prime$ if and only if $\mathcal{M}$ satisfies the following conditions:
- local consistency: for any node $t = (\Delta, \delta) \in T(\phi, \delta)$ and state $\pi' = (s', \delta') \in \Pi^\prime$, if $(\pi', t) \in \mathcal{M}$ then $\delta_t = \delta'$ and for any $\psi \in \Delta(t)$, $\mathcal{W}^\prime, \pi' \models^{\Delta(t)} \psi$;
- global consistency: for every path $P = t_0, t_1, \ldots$ of $T(\phi, \delta)$ s.t. there exist $\pi_i$ with $(\pi_i, t_i) \in \mathcal{M}$ for $i = 0, 1, \ldots$, there is no min-trace on $P$.

Detailed Proofs

**Proof.** [Proof of Theorem 5.1] 1. $\mathcal{W}, (m, l), \rho \models B \phi$ iff there exist $\rho_0, \rho_1, \ldots, \rho_m$ s.t.
  - $\rho_0 = \rho$ and for any $i = 1, \ldots, m$, $\rho_i = \| B_l \|_{\rho_{i-1}}$;
  - $\mathcal{W}, (m, l), \rho_m \models \phi$.

Let $\rho_i^X$ for any $i = 0, \ldots, m$ be defined as: $\rho_i^X(X) = (m, [l]) \models (m, l) \in \rho_i(X)$ for any $X \in \mathcal{V}$.

It is not difficult to verify that $\rho_0^X = \rho^X$ and $\rho_i^X = \| B_l \|_{\rho_{i-1}^X}$ for any $i = 1, \ldots, m$. 

196
We can prove that for any $i = 0, \ldots, m$, if $\mathcal{W}, (m, l), \rho_i \models \phi$, then $\mathcal{W}^i, (m, [l]), \rho_i^i \models \phi$ by induction on $\phi$. Moreover, $\mathcal{W}^i, (m, [l]), \rho_i \models \phi$ if $\mathcal{W}^i, (m, [l]), [\mathcal{B}]_{\rho_i} \models \phi$, where $[\mathcal{B}]_{\rho_i} = \rho_i^0$. Hence, $\mathcal{W}, (m, l), \rho \models \phi$ implies $\mathcal{W}^i, (m, [l]), \rho_i \models \phi$.

2. $\mathcal{W}^i, (s, \delta), \rho^i \models \phi$ if there exist $\rho_0, \rho_1^i, \ldots, \rho_m^i$ s.t.
   - $\rho_0 = \rho^i$ and for any $i = 1, \ldots, m$, $\rho_i^i = [\mathcal{B}]_{\rho_{i-1}}$;
   - $\mathcal{W}^i, (s, \delta), \rho_m^i \models \phi$.

Let $\rho_i$ for any $i = 0, \ldots, m$ be defined as: $\rho_i = \{(m, l) \mid (s, \delta, l) \in m\}$ for any $X \in \mathcal{V}$. It is not difficult to verify that $\rho_0 = \rho$ and $\rho_i = [\mathcal{B}]_{\rho_{i-1}}$ for any $i = 1, \ldots, m$.

We can prove that for any $i = 0, \ldots, m$, if $\mathcal{W}^i, (s, \delta), \rho_i^i \models \phi$, then $\mathcal{W}, (m, l), \rho_i \models \phi$ by induction on $\phi$. Moreover, $\mathcal{W}, (m, l), \rho \models \phi$ if $\mathcal{W}^i, (m, l), [\mathcal{B}]_{\rho} \models \phi$, where $[\mathcal{B}]_{\rho} = \rho^m$.

Hence, $\mathcal{W}^i, (s, \delta), \rho \models \phi$ implies $\mathcal{W}, (m, l), \rho \models \phi$.

Proof. [Proof of Theorem 6.2] Suppose $\phi_0 = (\pi_0, \delta_0)$ is satisfied at state $\pi_0$ in $\mathcal{W}^i$ under environment $\rho^i$. According to the above lemma, there is a consistent marking $\mathfrak{M}$ of $\mathcal{T}(\phi_0, \delta_0)$ respect to $\mathcal{W}^i$ and $\pi_0$. We construct a finite SWS $\mathcal{W}_f = (\Pi_f, \Sigma_f, \theta_f)$, with $\Sigma_f = \Sigma[\phi_0] \cup \{t \mid x \in \mathcal{V}\}$.

Let $A, B$ and $C$ be the set of $\mathcal{T}(\phi_0, \delta_0)$ nodes that are leaves, where the (mod) rule is applied and where the (res) rule is applied respectively. For $t \in A \cup B \cup C$, let $U(t)$ be the set of nodes of $\mathcal{T}(\phi_0, \delta_0)$ consisting of $t$ and all ancestors on the path back up to, but not including, the most recent ancestor in $A \cup B \cup C$; or back up to and including the root if no ancestor of $t$ is in $A \cup B \cup C$. Similarly for $t \in A \cup B$, let $U'(t)$ be the set of nodes of $\mathcal{T}(\phi_0, \delta_0)$ consisting of $t$ and all ancestors on the path back up to, but not including, the most recent ancestor in $A \cup B$; or back up to and including the root if no ancestor of $t$ is in $A \cup B$.

Let $\Pi_1 = \{(t, \delta) \mid t = (\Delta, \delta) \in A \cup B\}$ and $\Pi_2 = \{(t, \delta') \mid t = (\Delta, \delta) \in A \cup B, t' = (\Delta', \delta') \in C \cap U'(t)\}$. The state set $\Pi_f = \Pi_1 \cup \Pi_2$. Notice that $\Omega[\phi]$ is finite, so $\Pi_f$ is finite.

Then the transition relation $\theta_f$ is defined as:
- for any $(t_1, \delta_1), (t_2, \delta_2) \in \Pi_1$, $(t_1, \delta_1) \rightarrow_a (t_2, \delta_2)$ iff there exists an $\langle a \rangle$-son $t'$ of $t_1$ s.t. $t' \in U(t_2)$;
- for any $(t_1, \delta_1) \in \Pi_1$ and $(t_2, \delta_2') \in \Pi_2$, $(t_1, \delta_1) \rightarrow_a (t_2, \delta_2)$ iff there exists an $\langle a \rangle$-son $t'$ of $t$ and $t_2' = (\Delta_2, \delta_2') \in C$ s.t. $t_2' \in U'(t)$ and $t' \in U(t_2')$;
- for any $(t_1, \delta_1), (t_2, \delta_2') \in \Pi_2$, $(t_1, \delta_1) \rightarrow_a (t_2, \delta_2)$ iff there exist $t' = (\Delta', \delta') \in C$ and $(t, \delta) \in \Pi_1$ s.t. $t', t' \rightarrow_a (t, \delta, \delta') \rightarrow_a (t, \delta)$ iff there exist $t' = (\Delta', \delta') \in C$ s.t. $t \in U'(t)$.

For any $X \in \mathcal{X}$, let $\rho_0^i(X) = \{t = (\Delta, \delta) \mid X \in \Delta\}$. We need to prove that for any $\phi \in \mathcal{L}[\phi_0, \mathcal{B}_0]$ and $t = (\Delta, \delta) \in \Pi_f'$,

$\theta_f(t) = (\Delta', \delta') \in U(t), \phi \in \Delta'$ implies $\mathcal{W}^i, (t, \delta'), \rho_i \models \phi$.

This can be done in a similar way to that in [20, 32]. The correctness of the cases with weight is guaranteed by the region consistency.

Proof. [Proof of Lemma 7.3] Let $A = \{r \in \mathbb{Q}^+ \mid x \geq r \in \Lambda\}$ and $B = \{r \in \mathbb{Q}^+ \mid x \leq r \in \Lambda\}$.

(A1) guarantees that $A \neq \emptyset$ and if $B = \emptyset$, we can derive a contradiction from (R3) for $C[\mathcal{X}] = \mathcal{X}$. 197
Since the two sets are non-empty, the sup and inf exist. Moreover, (R3) can also be used to prove that sup \( A < \infty \). Let sup \( A = u \) and inf \( B = v \). If \( u < v \), there exists \( r \in \mathbb{Q}^+ \) such that \( u < r < v \). Hence, \( x \leq r \in \Lambda \), which contradicts \( r \leq v \). If \( v < u \), there exists \( r_1, r_2 \in \mathbb{Q}^+ \) such that \( v < r_1 < r_2 < u \). Hence, \( x \leq r_i \), \( x \geq r_i \in \Lambda \) for \( i = 1, 2 \). Since \( r_2 - r_1 > 0 \), (A3)− \( x \geq r_2 \rightarrow \neg(x \leq r_1) \), which proves the inconsistency of \( \Lambda \) - contradiction.

**Proof.** [Proof of Lemma 7.6] We prove the following properties first:

- For any \( S \subseteq \mathcal{L} \) and \( \delta, \delta_1, \delta_2 : \mathcal{V} \rightarrow \mathbb{R} \) such that \( \delta = \delta_1 + \delta_2 \), \( S \equiv \delta = (S \equiv \delta_1) \equiv \delta_2 \).

**Proof:** (⇒) Suppose \( \psi \in S \equiv \delta \). Then there must exist \( \psi \in S \), \( f_-, f_+ : \mathcal{V} \rightarrow \mathbb{Q} \) s.t. \( \delta < f_- \) and \( \psi' = \psi + f_-/f_+ \). Since \( \delta = \delta_1 + \delta_2 \), there exist \( g_-, g_+, h_-, h_+ : \mathcal{V} \rightarrow \mathbb{Q} \) s.t. \( \delta_1 < g_+ \) and \( g_- = g_+ + h_- f_- = g_+ + h_+ \). So \( \psi' = \psi + g_-/g_+, h_- = \psi + h_-/g_+ \). Since \( \psi + h_-/g_+ \in S \equiv \delta_1 \) by definition, we have \( \psi + h_-/g_+ \in (S \equiv \delta_1) \equiv \delta_2 \).

Hence, \( \psi' \in (S \equiv \delta_1) \equiv \delta_2 \).

(⇐) Suppose \( \psi' \in (S \equiv \delta_1) \equiv \delta_2 \). Then there must exist \( \psi \in S \), \( g_-, g_+, h_-, h_+ : \mathcal{V} \rightarrow \mathbb{Q} \) s.t. \( \delta < \delta_1 < g_+ \) and \( \psi' = \psi + g_-/g_+, h_- = \psi + h_-/g_+ \). Since \( \delta = \delta_1 + \delta_2 \), there exist \( f_-, f_+ : \mathcal{V} \rightarrow \mathbb{Q} \) s.t. \( \delta < f_+ \) and \( f_- = f_+ + h_+ \). So \( \psi' = \psi + h_+-f_+/f_+ \). Hence, \( \psi' \in S \equiv \delta \).

- 2. Let \( \Lambda_1, \Lambda_2 \in \Theta \) such that \( \{\Lambda_1, \Lambda_2\} \) is coherent. Then, for any \( l \in L \), \( \Lambda_1 \equiv (l - \mathcal{I}(\Lambda_1)) \equiv \Lambda_2 \equiv (l - \mathcal{I}(\Lambda_2)) \).

**Proof:** Let \( l_1 = \mathcal{I}(\Lambda_1), l_2 = \mathcal{I}(\Lambda_2) \).

(⇒) \( \Lambda_1 \equiv (l - l_1) \equiv \Lambda_1 \equiv ((l_2 - l_1) + (l - l_2)) \), which implies \( \Lambda_1 \equiv (l - l_1) = (\Lambda_1 \equiv (l_2 - l_1)) \equiv (l - l_2) \) by the above property. Since \( \{\Lambda_1, \Lambda_2\} \) is coherent, \( \Lambda_1 \equiv (l_2 - l_1) \subseteq \Lambda_2 \). So \( (\Lambda_1 \equiv (l_2 - l_1)) \equiv (l - l_2) \subseteq \Lambda_2 \equiv (l - l_2) \).

Similarly for the other direction.

With these properties, we can prove the lemma.

**I.** Firstly, observe that \( C \subseteq \Theta \) is coherent iff for any \( \Lambda_1, \Lambda_2 \in C \), with \( l_1 = \mathcal{I}(\Lambda_1), l_2 = \mathcal{I}(\Lambda_2) \),

\[ \Lambda_1 \equiv (l_2 - l_1) \subseteq \Lambda_2 \text{ and } \Lambda_2 \equiv (l_1 - l_2) \subseteq \Lambda_1. \]

Moreover, \( \Lambda_1 \equiv (l_2 - l_1) \subseteq \Lambda_2 \equiv (l_1 - l_2) \subseteq \Lambda_1 \).

**II.** Secondly, we observe that all the infinitary rules of our axiomatization have countable sets of instances. We consider the Boolean-completion of \( \mathcal{L} \) with the same axiomatization (see [14]), namely (an isomorphic copy of) the Boolean algebra of complete ideals in \( \mathcal{L} \).

The completion is a complete Boolean algebra. Every element in the completion is the supremum (in the completion) of the set of elements in \( \mathcal{L} \) that are below it. Moreover, \( \mathcal{L} \) is a dense subset of its completion in the sense that every non-zero element in the completion is above a non-zero element in \( \mathcal{L} \). Since the axiomatization is countable, the Rasiowa-Sikorski lemma [15, 29] applied to the completion guarantees that any non-zero element of the completion belongs to an ultrafilter (of the completion). Since any consistent set \( S \) of \( \mathcal{L} \) corresponds to a non-zero element \( \bigwedge S \) in the completion, by applying Rasiowa-Sikorski lemma to the completion of \( \mathcal{L} \), we obtain that there exists an ultrafilter \( u \) of the completion
containing \( \land S \). This is equivalent to the fact that there exists an ultrafilter \( u \cap \mathcal{L} \) of \( \mathcal{L} \) that includes \( S \).

**III.** We prove that if \( l = \mathcal{I}(\Lambda) \) and \( l' \in L \), then there exists \( \Lambda' \in \Theta \) s.t. \( \mathcal{I}(\Lambda') = l' \) and \( \{ \Lambda, \Lambda' \} \) is coherent. To prove this, we firstly need to prove that \( \Lambda \equiv (l' - l) \) is consistent. The following two properties guarantee the consistency, which can be proved by induction on the structure of the formulas:

(a) If \( \phi \in \Lambda \) and \( f_-, f_+ : \mathcal{V} \to \mathbb{Q} \) s.t. for any \( x \in \mathcal{V}(\phi) \), either \( f_- (x) = f_+(x) = 0 \) or \( f_- (x) < (l' - l)(x) < f_+(x) \), then,

\[
\vdash (\phi + f_+ / f_-) + - f_- / f_+ \rightarrow \phi.
\]

(b) For any \( x \leq r \in \mathcal{L} \),

\[
\{ (x \leq r) + f_+ / f_- | f_-, f_+ : \mathcal{V} \to \mathbb{Q}, f_- < 0 < f_+ \} \vdash x \leq r.
\]

Since \( \Lambda \equiv (l' - l) \) is consistent, applying II, it must have a \( \mathcal{B}_0 \)-maximal-consistent extension \( \Lambda' \). Since \( \Lambda \equiv (l' - l) \subseteq \Lambda' \), we also have \( \Lambda' \equiv (l' - l) \subseteq \Lambda \). Hence, \( \{ \Lambda, \Lambda' \} \) is coherent.

**IV.** Suppose \( C = \{ \Lambda_0, \Lambda_1, \ldots, \Lambda_k, \ldots \} \) is a coherent set (possibly infinite), \( l_i = \mathcal{I}(\Lambda_i), i = 1, \ldots, k, \ldots \) and \( l \in L \). Similarly with III, we can prove that \( \Lambda_i \equiv (l - l_i) \) is consistent. By Property 2 proven above, we have that \( \Lambda_1 \equiv (l - l_1) = \Lambda_2 \equiv (l - l_2) = \ldots = \Lambda_k \equiv (l - l_k) = \ldots \)

Hence, in order to get a coherent function \( \gamma \), we only need to get \( \Lambda \equiv (l' - l) \) for any \( l' \in L \), and extend it to \( \mathcal{B}_0 \)-maximal-consistent set \( \Lambda_F \) by applying II. Let \( \gamma(l') = \Lambda_F \). Obviously, \( \gamma \) is a coherent function. \( \square \)

**Proof.** [Proof of Lemma 7.7] Induction on \( \phi, \phi \lor \psi, \neg \phi \) and \( X \) cases are straightforward.

**[The case \( x \leq r \):]**

\((\Rightarrow)\) \( \mathcal{W}^x, \pi, \rho_0^l \models x \leq r \) implies for any \( l' \in [l], l'(x) \leq r \). So \( l(x) \leq r \), which implies that \( x \leq r \in \gamma(l) \).

\((\Leftarrow)\) \( x \leq r \in \gamma(l) \) implies \( l(x) \leq r \). Because \( x \in \mathcal{V}[\phi_0, \mathcal{B}] \), so \( r \in Q[\phi_0, \mathcal{B}] \). And since either \( [l] = \phi_0 \) or \( [l] = (\phi_0, \phi_0 \lor \psi, \neg \phi) \), it is obvious that for any \( l' \in [l], l'(x) \leq r \). Hence \( \mathcal{W}^x, \pi, \rho_0^l \models x \leq r \).

**[The case \( [a] \phi \):]**

\( \mathcal{W}^x, \pi, \rho_0^l \models [a] \phi \) iff for any \( \pi' = (\gamma', [l']) \in \Pi^s \) s.t. \( \pi \rightarrow_o \pi', \mathcal{W}^{x'}, \pi', \rho_0^l \models \phi \) iff \( \phi \in \gamma'(l') \) by induction hypothesis.

\((\Rightarrow)\) \( \langle a \rangle \neg \phi \in \gamma(l) \).

If \( \gamma \) cannot do any \( a \)-transition, then there should be no formula like \( \langle a \rangle \psi \) in \( \gamma(l) \) for all \( l \in L \) — contradiction!

Suppose \( \gamma \not\vdash_a \gamma' \). Let \( A_I = \langle \neg \phi \rangle \cup \psi \cup [a] \psi \in \gamma(l) \cup \mathcal{T}_{t+u} \) and \( A_F = \{ \psi | [a] \psi \in \gamma(l') \} \cup \mathcal{T}_{t+u} \) for any \( l' \neq l \), where \( \mathcal{T}_{t+u} = \bigcup_{x \in \mathcal{V}} \{ x \leq r \mid r \geq l'(x) \} \cup \{ x \geq r \mid r \leq l'(x) \} \).

It is easy to see that \( \{ \psi | [a] \psi \in \gamma(l) \} \cup \mathcal{T}_{t+u} \) and \( A_F \) for any \( l' \neq l \) are consistent.

Suppose that \( A_I \) is inconsistent. Then there exists a set \( F \subseteq A_I \) s.t. \( F \vdash \phi \). If \( F \) is finite, (R1) guarantees that \( [a]F \vdash [a] \phi \), where \( [a]F = \{ [a] \psi \mid \psi \in F \} \). Otherwise, \( F \vdash \phi \) is (modulo Boolean reasoning possible involving infinite meets) an instance of one of the rules (R2)-(R3); in all these cases, \( [a]F \vdash [a] \phi \) is an instance of the same rule for the
context \( C[\mathcal{X}] = [a]\mathcal{X} \). Since \( F \subseteq A_l, [a]F \subseteq \gamma(l) \) implying \([a]\phi \in \gamma(l)\), which contradicts the consistency of \( \gamma(l) \). Hence, \( A_l \) is consistent.

Now we prove that for any \( l_1, l_2 \in L, A_{l_1} \) and \( A_{l_2} \) are such that \( A_{l_1} + (l_2 - l_1) \subseteq A_{l_2} \). If \( l_1 \neq l \), then for arbitrary \( \psi' \in A_l \), either \([a]\psi' \in \gamma(l_1)\), or \( \psi' = x \leq r \).

In the first case, \([a]\psi' + /\\psi_r \in \gamma(l_2)\), for all \( f_- \leq l_2 - l_1 \leq f_+ \). So, \( \psi' + /\\psi_r \in A_{l_2} \).

In the second case, since \( \psi' = x \leq r \) is closed under any resource valuation transformation, \( f_- \leq l_2 - l_1 \leq f_+ \Rightarrow \psi' + /\\psi_r \in A_{l_2} \).

If \( l_1 = i \), consider an arbitrary \( \psi' \in A_{l_1} \). If \( \psi' \neq \phi \), we get a similar case as above. Otherwise, \( (a)\psi' \in \gamma(l) \), which implies \( (a)\psi' + /\\psi_r \in \gamma(l_2) \) for all \( f_- \leq l_2 - l_1 \leq f_+ \). So, \( \psi' + /\\psi_r \in A_{l_2} \).

At this point we can use a similar strategy as in Theorem 7.6 to prove that there exists \( \gamma'' \in \Gamma \) s.t. for any \( l' \in L, A_{l'} \subseteq \gamma''(l') \). Hence, \( \neg \phi \in \gamma''(l + u) \). According to the definition of the model, \( \gamma \xrightarrow{u} \gamma'' \), which implies \( \phi \in \gamma''(l + u) \) - contradiction!

Hence, \([a]\phi \in \gamma(l)\).

\((\Longleftarrow)\) derives from the definition of \( \theta^a \).

**The case \( x \text{ in } \phi \):**

\((\Longrightarrow)\) \( \mathcal{W}_l, \pi, \rho_0^l \models x \text{ in } \phi \) implies that there exists \( \pi' \in \Pi^x \) s.t. \( \pi \rightarrow \epsilon, \pi' \) and \( \mathcal{W}_l', \pi', \rho_0^l \models \phi \), which implies that \( \phi \in \gamma(l[x \mapsto 0]) \) by inductive hypothesis. Since \( l[x \mapsto 0](x) = 0 \), we have \( x \text{ in } \phi \in \gamma(l[x \mapsto 0]) \). Because \( \gamma \) is coherent function, it is not difficult to prove that \( x \text{ in } \phi \in \gamma(l) \).

\((\Longleftarrow)\) \( x \text{ in } \phi \in \gamma(l) \) implies that \( x \text{ in } \phi \in \gamma(l[x \mapsto 0]) \) by Definition 7.5. Therefore, \( \phi \in \gamma(l[x \mapsto 0]) \) by (A11). By inductive hypothesis, \( \mathcal{W}_l', (\gamma, [I], [x \mapsto 0]), \rho_0^l \models \phi \), which implies \( \mathcal{W}_l', (\gamma, [I]), \rho_0^l \models x \text{ in } \phi \).

**Proof. [Proof of Theorem 7.10]** By the semantics of the alternation-free block sequence, given an environment \( \rho_0, \mathcal{B} \) defines a series of environments: \( \rho^0, \ldots, \rho^m \), where \( \rho^i = \spadesuit [\mathcal{B}] \rho^i \), for any \( i = 1, \ldots, m \). And \( \spadesuit [\mathcal{B}] \rho^m = \rho^m \).

We prove that for \( \rho^i, i = 0, 1, \ldots, m \),

\( \mathcal{W}_l, \pi, \rho^i \models \phi \) iff \( \phi \in \gamma(l) \)

by induction on \( i \). The case \( i = 0 \) is given by Lemma 7.7. Suppose the statement holds for \( k \geq 0 \). Then it is still true according to Lemma 7.8 and Lemma 7.9.

And \( \mathcal{W}_l', \pi, \rho^0 \models \gamma \) iff \( \mathcal{W}_l', \pi, \rho^m \models \phi \). Therefore, \( \mathcal{W}_l', \pi, \rho^0 \models \phi \) iff \( \phi \in \gamma(l) \).
Towards Compositional Graph Theory

Apiwat Chantawibul and Paweł Sobociński

ECS, University of Southampton

Dedicated to the memory of R.F.C. Walters.

Abstract

Decomposing graphs into simpler graphs is one of the central concerns of graph theory. Investigations have revealed deep concepts such as modular decomposition, tree width or rank width, which measure—in different ways—the structural complexity of a graph’s topology. Courcelle and others have shown that such concepts can be used to obtain efficient algorithms for families of graphs that are amenable to decomposition (e.g. those that have bounded tree-width). These algorithms, in turn, are of course of use in computer science, where graphs are ubiquitous. In this paper we take the first steps towards understanding notions of decomposition in graph theory compositionally, and more generally, in a categorical setting: category theory, after all, is the mathematics of compositionality.

We introduce the concept of ∪−matrices (cup-matrices). Like ordinary matrices, ∪-matrices are the arrows of a PROP: we give a presentation, extending the work of Lafont, and Bonchi, Zanasi and the second author. A variant of ∪-matrices is then used in the development of a novel algebra of simple graphs, the lingua franca of graph theory. The algebra is that of a certain symmetric monoidal theory: ∪-matrices—akin to adjacency matrices—encode the graphs’ topology.

1 Introduction

When category theorists talk about the category of graphs, they usually mean the presheaf category Graph = Set^op. The objects of this category, however, are not what graph theorists would typically refer to as graphs tout court – rather, as directed multigraphs, because there may be more than one directed edge between any two vertices. The most basic—and important—objects of study in graph theory are simple graphs: these are undirected with at most one edge between any two vertices, and no self loops. In this paper, we are concerned with simple graphs.

A structural metric of a graph is a way of assigning a numerical value, typically a natural number, to a graph. The intention is for the number to indicate, in some way, the graph’s inherent structural complexity. Some well-known structural metrics in graph theory include path-width, tree-width, branch-width, clique-width and rank-width. The notion of tree-width is perhaps the best known in Computer Science through the work of Courcelle [6] who showed that monadic second-order logic can be decided in linear time for families of graphs with bounded tree-width. Courcelle’s theorem has found several algorithmic applications (see e.g. [13]).
The structural metric rank-width, due to Oum and Seymour, has been a hot topic in graph theory over the last ten years and is of particular relevance for us. We refer to [17, 18] for the technical details, here we give an intuitive description. A rank-decomposition of a simple graph with vertex set $V$ can be considered as a binary tree, where the tree-nodes are labelled with nonempty subsets of $V$ and the tree-edges are labelled with natural numbers, such that:

- the labels (vertex sets $W_1, W_2$) of the children $w_1, w_2$ of any tree-node $w$ are a (binary) partition of the label (vertex set $W$) of the parent tree-node: i.e. $W = W_1 \cup W_2$ and $W_1 \cap W_2 = \emptyset$,
- the root is labelled with $V$,
- the leaves are labelled with singletons,
- the tree-edge from a parent to a child labelled with vertex set $W$ is labelled with the rank of the $|V\setminus W| \times |W|$ adjacency $\mathbb{Z}_2$-matrix that tabulates the edges from $W$ to the remainder of the graph. Note that matrix algebra is performed over the field $\mathbb{Z}_2$, i.e. $1 + 1 = 0$.

The width of a particular rank-decomposition is its maximum edge label. The rank-width of a graph is then the width of an optimal rank-decomposition: one with the smallest width. Discrete graphs and cliques both enjoy a rank-width of 1.

The concepts of rank-width and other structural metrics have proved to be very important in graph theory and related areas. There are two shortcomings, however, where category theory can help:

- **Generality.** The definition, as stated, is specialised to simple graphs. Yet, the underlying concept is quite robust and can be stated mutatis mutandis for other kinds of graphical structures: multigraphs, directed graphs (bi-rank-width), hypergraphs, Petri nets etc. This suggests that the fundamental theory ought to be done in a more general setting.

- **Compositionality.** The notion of a rank-decomposition (and equivalent notions for other structural metrics) is not particularly compositional in the sense that knowing how to decompose a graph $G$ may not help in constructing decompositions of a graph $H$ that has $G$ as a sub-component. Intuitively speaking, rank-decompositions forget too much: only rank is recorded – the adjacency information ought to be recorded as well, in some form.

What, then, is a fruitful way of treating simple graphs categorically, in a way that will lead us to understanding structural metrics generally and compositionally? Graphs as objects and homomorphisms as arrows is the traditional approach, yet it does not immediately yield an algebra of graphs: for that, we need graphs to be the arrows. Cospans are a standard technique for turning objects into arrows, and the algebra of cospans of directed graphs was considered in [9], which is close in spirit to our work, although we do not consider cospans.

Similarly to Fiore and Campos’ work on an algebra of directed acyclic graphs [8], our approach is to use symmetric monoidal theories, by which we mean presentations of PROPs with generators and equations. The paper culminates with the symmetric monoidal theory $\text{ABUV}$, which we consider as an algebra of simple graphs. The role of matrix algebra is central: we build on Lafont’s [15] and Bonchi, Zanasi and the
second author’s [3] work on presentations of the PROP of matrices.

Indeed, in order to understand \( \mathbf{ABUV} \), we introduce the concept of \( \cup \)-matrices ("cup" matrices) that generalise matrix algebra and somewhat relax the inherent straightjacket of directionality in ordinary matrices. The main technical contribution in this paper is a presentation of \( \cup \)-matrices. We conclude with some remarks on a structural metric, monoidal width, for symmetric monoidal theories.

The use of the algebra of monoidal categories to model systems of various kinds was pioneered by R.F.C. Walters and collaborators [11, 12]. Although we do not concentrate on applications in this paper, we believe that our theory will be relevant in several settings as there has been a recent surge in the applications of symmetric monoidal theories: amongst other works we mention signal flow graphs [1, 4], Petri nets [19–21], asynchronous circuits [10] and quantum circuits [5, 7].

Structure of the paper.

In §2 we recall the presentation of the PROP of matrices. In §3 we introduce \( \cup \)-matrices and develop their algebra. In §4 we give a presentation, and in §5 we use the earlier technical developments to introduce an algebra of simple graphs.

Notation and background.

Given an \( m_1 \times n_1 \) matrix \( A \) and \( m_2 \times n_2 \) matrix \( B \), we write \( A \oplus B \) for the \((m_1 + m_2) \times (n_1 + n_2)\) matrix \( \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \). If \( A \) is an \( m \times n \) matrix, \( A^T \) denotes its transpose, an \( n \times m \) matrix, where \( A_{ij}^T = A_{ji} \). A matrix \( A \) is upper triangular when it has only 0 entries below the main diagonal, i.e. \( A_{ij} = 0 \) if \( i > j \).

A PROP [14,16] is a symmetric strict monoidal category with natural numbers as objects and with addition as the monoidal product on objects: \( m \oplus n = m + n \). PROP homomorphisms are identity-on-objects strict symmetric monoidal functors: that is they preserve objects, composition, monoidal products and symmetries on the nose. We reserve the term symmetric monoidal theory for PROPs that are freely generated from a presentation: a pair \( (\Sigma, E) \) where \( \Sigma \) is the set of generators and \( E \) is a set of equations (some authors say relations).

2 Matrices as a Symmetric Monoidal Theory

The results in this section can be found in [3], based on the work of [15].

Fix a ring \( R \) (e.g. the integers); this is the source for the entries of our matrices, which we will not explicitly reference during the development in order to reduce the number of subscripts. We use \( r, s \) to range over \( R \).

**Definition 2.1** The PROP \( \text{Mat} \) has \( m \times n \) matrices as arrows \( n \to m \); composition is matrix multiplication \( B \circ A = BA \). The symmetries are permutation matrices.

**Definition 2.2** (HA) The symmetric monoidal theory \( \text{HA} \) is the free PROP on generators \( \{ \Delta : 1 \to 2, \bot : 1 \to 0, \nabla : 2 \to 1, \top : 0 \to 1 \} \cup \{ r : 1 \to 1 \}_{r \in R} \) and equations listed in Fig. 1. In string diagrams the generators are rendered:

\[
\Delta = \begin{array}{c}
\bullet \\
\end{array} \\
\bot = \bullet \\
\nabla = \begin{array}{c}
\bullet \\
\end{array} \\
\top = \bullet \\
r = \begin{array}{c}
\square
\end{array}
\]

203
\[ \nabla \circ (\nabla \oplus id) = \nabla \circ (id \oplus \nabla) \quad (A1) \]

\[ \nabla \circ (id \oplus \top) = id \quad (A2) \]

\[ \nabla \circ tw = \nabla \quad (A3) \]

\[ (\Delta \oplus id) \circ \Delta = (id \oplus \Delta) \circ \Delta \quad (A4) \]

\[ (id \oplus \bot) \circ \Delta = id \quad (A5) \]

\[ tw \circ \Delta = \Delta \quad (A6) \]

\[ \bot \circ \nabla = \bot \oplus \bot \quad (A7) \]

\[ \Delta \circ \top = \top \oplus \top \quad (A8) \]

\[ \Delta \circ \nabla = (\nabla \oplus \nabla) \circ (id \oplus tw \oplus id) \circ (\Delta \oplus \Delta) \quad (A9) \]

\[ \bot \circ \top = id_0 \quad (A10) \]

\[ r \circ \nabla = \nabla \circ (r \oplus r) \quad (A11) \]

\[ r \circ \top = \top \quad (A12) \]

\[ \Delta \circ r = (r \oplus r) \circ \Delta \quad (A13) \]

\[ \bot \circ r = \bot \quad (A14) \]

\[ \nabla \circ (r \oplus s) \circ \Delta = r + s \quad (A15) \]

\[ 0 = \bot \circ \top \quad (A16) \]

\[ s \circ r = sr \quad (A17) \]

\[ 1 = id \quad (A18) \]

Fig. 1. Equations of HA.

Theorem 2.3 ([3,15]) HA \cong \text{Mat}.

\[ \square \]
3 Union-Matrices

Matrices are common in graph theory: for example, they are a convenient tool to record the connectivity information within a graph (adjacency matrices), and they enjoy a rich and ubiquitously used algebra on which several techniques and results rely.

In our categorical algebra of graphs, introduced in §5, an arrow \( G : m \to n \) combines three elements:

- a simple graph \( G \) on \( k \) vertices,
- a \( m \times k \) matrix,
- a \( n \times k \) matrix.

The idea is that the two matrices tell us how the vertices of \( G \) connect with other graphs that may be attached, at either end, through composition. Since \( G \) can itself be represented as a \( k \times k \) adjacency matrix, we are dealing with three matrices. We will see that relaxing the “directed” nature of matrices allows us to roll the three matrices into one union-matrix (pronounced “cup”-matrix).

3.1 Turns

Before introducing \( \cup \)-matrices, we first need one important component, the notion of turn – an equivalence class of \( n \times n \) matrices that is closely connected with the concept of adjacency matrix in graph theory: two matrices are turn equivalent exactly when they encode the same connectivity information (in an undirected setting).

**Definition 3.1 (Turn equivalence, Turn)** We say that two square \( k \times k \) matrices \( K \) and \( L \) are turn-equivalent when \( K + K^T = L + L^T \), and write \( K \bowtie L \). Clearly \( \bowtie \) is an equivalence relation. Note that if \( K \bowtie L \) then \( K \) and \( L \) have equal size.

By a \((k-)\)turn we mean a \( \bowtie \) equivalence class of some \( k \times k \) matrix. We use \( M, N \) to range over turns, and given square \( K \), we denote the induced turn by \( [K] \).

In certain cases, a turn has an obvious representative: an upper triangular matrix.

**Lemma 3.2** Given a turn \([K]\), there exists an upper triangular matrix \( K' \) such that \([K] = [K']\). Suppose also that \( \forall r, s \in R, \) we have \( 2r = 2s \) implies \( r = s \). If \( K \) and \( L \) are upper triangular square matrices, and \([K] = [L]\), then \( K = L \).

**Proof.** The first claim follows easily from the fact that addition in \( R \) is an abelian group. Next, if \([K] = [L]\) then \( K + K^T = L + L^T \). Since \( K \) and \( L \) are upper triangular, they clearly agree in all non diagonal entries. The \( i \)th diagonal entry of \( K + K^T = L + L^T \) is \( 2K_{ii} = 2L_{ii} \). Using the additional assumption on \( R \), \( K_{ii} = L_{ii} \).\( \square \)

The additional assumption on \( R \) in Lemma 3.2 excludes the case \( R = \mathbb{Z}_2 \) where we have \([1] = [0]\) so the second statement does not hold. In fact, the case of \( \mathbb{Z}_2 \) is of particular importance in for our applications, and we will return to it in §5.

---

1 This is the case, for instance, for the integers, or any field of characteristic not equal to 2.
3.2 \( \cup \)-Matrices

A \( m \times n \) \( \cup \)-matrix consists of an ordinary \( m \times n \) matrix together with an \( n \)-turn.

**Definition 3.3 (\( \cup \)-matrix)** An \( m \times n \) \( \cup \)-matrix \( U \) is a pair \((A, M)\) where \( A \) is an \( m \times n \) matrix and \( M \) is an \( n \)-turn. We will refer to \( A \) as the underlying matrix of \( U \) and to \( M \) as its turn. We will use \( U, V \) to range over \( \cup \)-matrices.

It turns out that there is a very natural algebra of \( \cup \)-matrices that extends that of ordinary matrices. We develop this algebra below.

Given an \( m \times n \) \( \cup \)-matrix \( V = (B, [L]) \) and \( n \times p \) \( \cup \)-matrix \( U = (A, [K]) \) their multiplication, written \( VU \), is defined as the \( m \times p \) \( \cup \)-matrix:

\[
(BA, [K + A^T L A])
\]

**Lemma 3.4** Multiplication of \( \cup \)-matrices is well-defined.

**Proof.** Suppose that \( K \uplus K' \) and \( L \uplus L' \). We need \( K + A^T L A \uplus K' + A^T L' A \).

\[
K + A^T L A + (K + A^T L A)^T = K + K^T + A^T L A + A^T L'^A \\
= K + K^T + A^T (L + L^T) A \\
= K' + K'^T + A^T (L' + L'^T) A \\
= K' + A^T L' A + (K' + A^T L' A)^T
\]

\( \Box \)

**Lemma 3.5** Multiplication of \( \cup \)-matrices is associative.

**Proof.** This is true of the underlying matrices, hence it suffices to check associativity on the turns. Suppose that \( W = (C, [M]) \), \( V = (B, [L]) \), \( U = (A, [K]) \) are \( \cup \)-matrices on which \( WV \) and \( VU \) are defined. Then their turns are, respectively, \([L + B^T MB]\) and \([K + A^T L A]\). The turn of \((WV)U\) is thus the equivalence class of

\[
K + A^T (L + B^T MB) A = K + A^T LA + A^T B^T MBA \\
= K + A^T LA + (BA)^T M(BA)
\]

which is matrix that induces the turn of \( W(VU) \).

\( \Box \)

The \( n \times n \) identity \( \cup \)-matrix is \((I_n, [0_n])\) where \( I_n \) is an identity matrix and \( 0_n \) is a zero matrix. We abuse notation by also referring to the identity \( \cup \)-matrix as \( I_n \).

**Lemma 3.6** Suppose that \( U \) is an \( m \times n \) \( \cup \)-matrix. Then \( I_m U = U I_n = U \).

**Proof.** Suppose \( U = (A, [K]) \). Clearly it suffices to focus on the turns. The turn of \( I_m U \) \([K + A^T 0_m A] = [K] \). The turn of \( U I_n \) is \([0_n + I_n^T 0_n K I_n] = [K] \).

\( \Box \)

**Definition 3.7 (Category of \( \cup \)-matrices, UMat)** UMat has:

- as its set of objects, the set of natural numbers,
- and as arrows from \( n \) to \( m \), \( m \times n \) \( \cup \)-matrices.

Composition is \( \cup \)-matrix multiplication: \( U \circ V \overset{\text{def}}{=} UV \), with identity arrows the identity \( \cup \)-matrices.

206
Given an $m_1 \times n_1 \cup$-matrix $U = (A, [K])$ and $m_2 \times n_2 \cup$-matrix $V = (B, [L])$, their direct sum, written $U \oplus V$, is the $(m_1 + m_2) \times (n_1 + n_2) \cup$-matrix

$$(A \oplus B, [K \oplus L]).$$

This is well-defined—if $K \bowtie K'$ and $L \bowtie L'$ then clearly $K \oplus L \bowtie K' \oplus L'$. The direct sum yields a strict monoidal product on $\text{UMat}$.

There is an faithful identity-on-objects monoidal functor $U : \text{Mat} \to \text{UMat}$ defined $U(A) = (A, [0_n])$. We have seen that $\text{Mat}$ is a PROP (Definition 2.1); in fact, $\text{UMat}$ is also a PROP, inheriting its permutations from $\text{Mat}$ via $U$.

**Lemma 3.8** $\text{UMat}$ is a PROP.

**Proof.** It suffices to show that symmetries are natural, i.e. the following diagram commutes for any $m' \times m \cup$-matrix $U = (A, [K])$ and $n' \times n \cup$-matrix $V = (B, [L])$.

$$
\begin{array}{ccc}
m + n & \xrightarrow{\sigma_{m,n}} & n + m \\
U \oplus V & \downarrow & \downarrow V \oplus U \\
m' + n' & \xrightarrow{\sigma_{m',n'}} & n' + m'
\end{array}
$$

On the underlying matrices we clearly have $\sigma_{m',n'}(A \oplus B) = (B \oplus A)\sigma_{m,n}$. Notice that $\sigma^T_{k,l} = \sigma_{l,k}$ for any $k, l \in \mathbb{N}$. We can use this to show that the turns agree:

$$\sigma^T_{m,n}(L \oplus K)\sigma_{m,n} = \sigma_{n,m}(L \oplus K)\sigma_{m,n} = K \oplus L$$

Indeed, the functor $U : \text{Mat} \to \text{UMat}$ is a PROP homomorphism.

We end this section with a simple, yet useful way to factorise $\cup$-matrices. By a **pure turn** we mean an arrow of type $m \to 0$ in $\text{UMat}$, i.e. a $0 \times m \cup$-matrix. By a **turn-free** $\cup$-matrix we mean an arrow in the image of $U : \text{Mat} \to \text{UMat}$, i.e., a $\cup$-matrix of the form $(A, [0])$. In the following, let $\Delta_n$ denote the turn-free $(n + n) \times n \cup$-matrix $((l_n, l_n), [0_n])$.

**Lemma 3.9** Suppose that $U$ is an $m \times n \cup$-matrix. Then

$$U = (A \oplus P)\Delta_n$$

where $P$ is a pure turn and $A$ is turn free.

**Proof.** Easy calculation: $(A, [K]) = ((A, [0]) \oplus ([0], [K]))\Delta_n$.  

## 4 \cup-Matrices as a Symmetric Monoidal Theory

The goal of this section is to give a presentation of $\text{UMat}$, extending the presentation of $\text{Mat}$ recalled in §2. We introduce a symmetric monoidal theory, $\text{HAU}$, and prove that $\text{HAU} \cong \text{UMat}$ as PROPs.
Definition 4.1 (HAU) The symmetric monoidal theory HAU is the free PROP on generators \( \{ \Delta : 1 \to 2, \perp : 1 \to 0, \nabla : 2 \to 1, \top : 0 \to 1, \cup : 2 \to 0 \} \cup \{ r : 1 \to 1 \}_{r \in R} \) and the set of equations listed in Fig. 1 together with the additional equations concerning \( \cup \) (pronounced “cup”) listed in Fig. 2. In string diagrams, \( \cup \) is drawn \( \bigcirc \). 

Our main goal for this section is to prove the following.

Theorem 4.2 Suppose that \( R \) satisfies the condition \( 2r = 2s \) implies \( r = s \), for all \( r, s \in R \). Then \( \text{HAU} \cong \text{UMat} \).

The remainder of this section consists of a proof of the above result; we thus assume that \( R \) satisfies the required condition for the remainder of this section.

We start by defining a PROP morphism \( \Theta : \text{HAU} \to \text{UMat} \) recursively.

\[
\Delta \mapsto \left( \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{bmatrix} 0 \end{bmatrix} \right) \quad \nabla \mapsto \left( \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right) \quad \cup \mapsto \left( !, \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right) \\
\perp \mapsto (!, [0]) \quad \top \mapsto (i, [0])
\]

Note that we are being somewhat sloppy with notation since we denote by \( ! \) all unique arrows to 0 and by \( i \) all unique arrows from 0 in \( \text{Mat} \). Since we are defining a symmetric monoidal functor, the recursion is forced, e.g.:

\[
\Theta(q \circ p) = \Theta(q) \circ \Theta(p) \quad \Theta(p \oplus q) = \Theta(p) \oplus \Theta(q)
\]

This procedure is well-defined: since both \( \text{HAU} \) and \( \text{UMat} \) are symmetric monoidal it suffices to show that all equations in \( \text{HAU} \) hold also in \( \text{UMat} \). This is the case for \( \text{HA} \) and \( \text{Mat} \) (as shown in [3]) – it suffices to check the equations involving \( \cup \).

Lemma 4.3

(i) \( (\Theta \cup)(\Theta \nabla \oplus id) = (\Theta \cup)(id \oplus \Theta \cup \oplus id)(id_2 \oplus \Theta \Delta) \)

(ii) \( (\Theta \cup)(id \oplus \Theta \top) = \Theta \perp \)

(iii) \( (\Theta \cup)(\Theta tw) = \Theta \cup \)

(iv) \( (\Theta \cup)\Theta(id \oplus r) = (\Theta \cup)\Theta(r \oplus id) \).
Proof. Straightforward calculations. \hfill \qed}

**Lemma 4.4** The following diagram commutes.

\[
\begin{array}{c}
\text{HA} \xrightarrow{\Phi} \text{HAU} \\
\downarrow \quad \downarrow \\
\text{Mat} \xrightarrow{U} \text{UMat}
\end{array}
\tag{1}
\]

**Proof.** Since the diagram consists of PROP homomorphisms, it suffices to verify that it is the case for the generators of HA. \hfill \qed

The above observation allows us to conclude that the additional axioms of HAU are conservative wrt the equations in HA.

**Lemma 4.5** Suppose \( t, u \in \text{HA} \). Then \( t =_{\text{HA}} u \) iff \( t =_{\text{HAU}} u \).

**Proof.** The ‘only if’ direction is obvious. For the ‘if’, observe that in the diagram (1), \( \Phi \) is an isomorphism and \( U \) is faithful, so their composition is faithful. Since the diagram commutes, the homomorphism \( \text{HA} \to \text{HAU} \) is faithful. \hfill \qed

Since \( \Theta \) is identity on objects, to show it is an isomorphism it suffices to show that it is full and faithful.

**Proposition 4.6** \( \Theta \) is full.

**Proof.** Suppose that \( U = (A, [K]) \) is an \( m \times n \cup \)-matrix. We must construct a term \( t_U \) of HAU such that \( \Theta(t_U) = U \). Using the factorisation of Lemma 3.9, we have \( U = ((A, [0_n]) \oplus ([!, [K]]) \Delta_n \). We can break up the problem into two parts: construct \( t_A : n \to m \) such that \( \Theta(t_A) = (A, [0_n]) \) and \( t_K : n \to n \) such that \( \Theta(t_K) = (\text{id}_n, [K]) \). Since \( \text{HA} \cong \text{Mat} \), it is straightforward to obtain \( t_A \).

We are left with \((!, [K])\), where w.l.o.g. we assume that \( K \) is upper triangular. The problem becomes: given an upper triangular \( n \times n \) \( K \), construct a term of \( t_K \) such that \( \Theta(t_K) = (\text{id}_n, [K]) \). We can do this by constructing two terms in HAU:

- given \( p < q \leq n \), a term \( \text{inc}_{p,q} : n \to n \), satisfying, for any \((B, [L])\)

\[
\Theta(\text{inc}_{p,q})(B, [L]) = (B, [L']) \text{ where } L'_{ij} = \begin{cases} L_{ij} + 1 & \text{if } i = p \text{ and } j = q \\ L_{ij} & \text{otherwise.} \end{cases}
\]

- given \( p \leq n \), \( \text{inc}_i : n \to n \), satisfying,

\[
\Theta(\text{inc}_p)(B, [L]) = (B, [L']) \text{ where } L'_{ij} = \begin{cases} L_{ij} + 1 & \text{if } i = j = p \\ L_{ij} & \text{otherwise.} \end{cases}
\]

For sake of simplicity, we only give the appropriate string diagrams below and let the reader verify that \( \Theta(\text{inc}_{p,q}) \) and \( \Theta(\text{inc}_p) \) work as advertised:

\[
\text{Diagram A}
\]

\[
\text{Diagram B}
\]

209
Then, clearly, $t_K$ can be constructed by an appropriate composition of these two families of terms, finally composing with $\bot_n$ to obtain $t_K$. 

Proving that $\Theta$ is faithful is slightly more involved. We start by showing how turns “add” in HAU; the proofs are simple exercises in diagrammatic reasoning.

**Lemma 4.7**

The crux of the proof is a certain “normal form” for representations of pure turns in HAU. Basically, one can write any such term without using $\nabla$ or $\top$.

**Lemma 4.8**

The following are the rewrite rules that we will apply:

at each point obtaining a step $t_{k+1}^{k+1} \circ t_{\bot}^{k+1} \circ t'_{\bot} \Rightarrow t_{\bot}^{k+1} \circ t'_{\bot}$ where $t_{\bot}^{k+1}$ results from $t'_{\bot}$ by putting it in matrix form.

The only tricky issue is proving termination: we can show it using the concept of monoid depth. Given a term $t_{\bot} : k \rightarrow l$, we let the monoid depth of right port $0 \leq i < l$ of $md(t_{\bot}, i)$ be the number of non-zero entries in the $i$th row of the
matrix that corresponds to $t_{HA}$. The isomorphism $HA \cong \text{Mat}$ guarantees that this definition makes sense. The goal is to reach a term where all ports have monoid depth 1, i.e. one whose matrix has exactly one non-zero entry in each row.

Now, given two ports with monoid depth $k > 1, l$ connected by a turn, each of the first two rewrite rules decrements the monoid depth of $k$, and adds another port of depth $l$. The last two rules simply remove a port of multiplication depth 0 (in other words, remove a zero row from the matrix). By always rewriting at a port with the highest monoid depth, and cancelling any ports of depth 0, clearly one eventually arrives at a term with all ports of monoid depth 1.

Lemma 4.10 (Turn matrix form) Any $t : k \rightarrow 0$ can be written in the form

$$t = \Upsilon_k \circ \left( \bigoplus_{1 \leq i \leq k} \bigoplus_{1 \leq j \leq k+1} r_{i,j} \right) \circ \left( \bigoplus_{1 \leq i \leq k} \Delta_{k+1} \right)$$

where $r_{i,j} = 1$ whenever $j = k + 1$ or $i > j$ and $\Upsilon : k(k+1) \rightarrow 0$ is constructed by connecting, for each $1 \leq i \leq k$:

- the $(i-1)(k+1) + i$th port and $((i-1)(k+1) + k+1)$th port, and
- for each $j > i$, the $(i-1)(k+1) + j$th port with the $(j-1)(k+1) + i$th port with cups. A recursive definition can be given, but it is unenlightening: the diagram below does a better job of illustrating the structure of $\Upsilon_k$.

Let $K$ be the upper triangular matrix with $j, i$th entry $r_{i,j}$ when $i \leq j$ and 0 otherwise. Then $\Theta(t) = (!, [K])$.

**Proof.** Using the procedure of Lemma 4.9 we can write any term $k \rightarrow 0$ in the form $t_{ij} \circ t_\Delta$, then add up the coefficients using the equations of Lemmas 4.8 and 4.7. □
Example 4.11 The turn matrix form of \[ \begin{array}{l}
\end{array} \] is calculated below:

\[ \begin{array}{l}
\end{array} \Rightarrow \begin{array}{l}
\end{array} \Rightarrow 2 \]

where we did not draw the 0-coefficient turns. It is easily checked that this gives the upper triangular matrix

\[
\begin{pmatrix}
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}
\]

Proposition 4.12 \( \Theta \) is faithful.

Proof. Suppose that \( t, u : n \to m \) such that \( \Theta(t) = \Theta(u) \). Using the laws of HA we can write \( t = (t_{HA} \oplus t') \circ \Delta_n \) and \( u = (u_{HA} \oplus u') \circ \Delta_n \), where \( t_{HA}, u_{HA} : n \to m \) and \( t', u' : n \to 0 \).

Now \( \Theta(\Delta_n) \) is mono, thus \( \Theta(t) = \Theta(u) \) implies

\[
\Theta(t_{HA} \oplus t') = \Theta(u_{HA} \oplus u').
\]

By definition of direct sum in UMat, we have

\[
\Theta(t_{HA}) = \Theta(u_{HA}) \quad \Theta(t') = \Theta(u')
\]

Since HA \( \cong \text{Mat} \) and \( U : \text{Mat} \to \text{UMat} \) is faithful, we have \( t_{HA} = u_{HA} \). It remains to show that \( \Theta(t') = \Theta(u') \) implies that \( t' = u' \). Transform \( t' \) and \( u' \) into turn matrix form. Since \( (!, [K]) = \Theta(t') = \Theta(u') = (!, [L]) \), we must have \( [K] = [L] \). Since \( K \) and \( L \) are upper triangular, the conclusion of Lemma 3.2 yields \( K = L \). Thus \( t' = u' \).

We have shown that \( \Theta \) is full in Proposition 4.6, and that it is faithful in Proposition 4.12. This completes the list of ingredients needed for the proof of Theorem 4.2.

5 A compositional theory of simple graphs: AUV

In this section we focus on the case \( R = \mathbb{Z}_2 \). We give a presentation for \( \text{UMat}_{\mathbb{Z}_2} \), and show how one can use this to obtain an algebra of simple graphs.

5.1 \( \cup \)-Matrices over \( \mathbb{Z}_2 \)

Definition 5.1 (AB and ABU) The symmetric monoidal theory AB is the free PROP on generators \( \{ \Delta : 1 \to 2, \perp : 1 \to 0, \triangledown : 2 \to 1, \top : 0 \to 1 \} \), and equations (A1)-(A10) listed in Fig. 1 and (C1) in Fig. 3.
The symmetric monoidal theory \( \text{ABU} \) is the free PROP on generators \( \{ \Delta : 1 \to 2, \bot : 1 \to 0, \nabla : 2 \to 1, \top : 0 \to 1, \cup : 2 \to 0 \} \), the equations of \( \text{AB} \), equations (B1)-(B3) of Fig. 2 and (C2) in Fig. 3.

First recall [2, 15] that \( \text{AB} \cong \text{Mat}_{\mathbb{Z}_2} \). Likewise, presenting \( \text{UMat}_{\mathbb{Z}_2} \) is simpler than the general case: there is no need for additional scalars, and the equation \( 1 + 1 = 0 \) is captured by the “anti-separable” law – equation (C1) in Fig. 3 [2]. The equation (C2) may seem surprising, since it does not hold in the general case. Indeed, the requirement of “no self loops” in simple graphs is already taken care of by the algebra of \( \cup \)-matrices with \( \mathbb{Z}_2 \) entries.

\[ \Theta(\cup \circ \Delta) = \Theta(\top) \]

**Proof.** For \( \mathbb{Z}_2 \)-matrices we have \((0) \bowtie (1)\), so \( [(0)] = [(1)] \). \( \square \)

**Theorem 5.3** \( \text{ABU} \cong \text{UMat}_{\mathbb{Z}_2} \).

**Proof.** Proceeds along the lines of the proof of Theorem 4.2: the only difference is that instead of using upper triangular matrices as representatives of turns, we use upper triangular matrices with 0 diagonals. We omit the details. \( \square \)

### 5.2 An algebra of simple graphs

We will use the theory \( \text{ABU}(\cong \text{UMat}_{\mathbb{Z}_2}) \) as an adjacency algebra of simple graphs. The only thing missing is the vertices.

**Definition 5.4** (ABUV) *The symmetric monoidal theory \( \text{ABUV} \) is the free PROP on generators \( \{ \Delta : 1 \to 2, \bot : 1 \to 0, \nabla : 2 \to 1, \top : 0 \to 1, \cup : 2 \to 0, v : 0 \to 1 \} \), and the equations of \( \text{ABU} \). In string diagrams, we will draw \( v : 0 \to 1 \) as follows:*

\[ \]

Since there are no additional equations involving \( v \), \( \text{ABUV} \) is the coproduct (in the category of PROPs) of \( \text{ABU} \) and the free PROP on generator \( v \) and no equations.

**Definition 5.5** Let \( C\text{Graph} \) be the 2-PROP with arrows \( n \to m \) pairs \( (k, U) \) where

* \( k \in \mathbb{N} \) and
* \( U \) is an \( m \times (n + k) \cup \)-matrix.

Composition \( (V, l) \circ (U, k) \) is

\[ (k + l, V(U_1 \oplus I_k)) \]

For the 2-cells, given a bijection \( \sigma : k \to k \), let \( I_{\sigma} \) denote the permutation matrix
induced by $\sigma$. Then $\sigma : (U, k) \Rightarrow (U', k) : n \to m$ is a 2-cell if

$$U'(I_n \oplus I_\sigma) = U.$$  

Note that all 2-cells are invertible. Let $\text{CGraph}_\sim$ denote the category obtained by identifying isomorphic 1-cells in $\text{CGraph}$.

**Theorem 5.6** $\text{ABUV} \cong \text{CGraph}_\sim$.

**Proof.** Omitted.

**Corollary 5.7** There is a 1-1 correspondence between $\text{ABUV}[0,0]$ and (isomorphism classes of) simple graphs.

**Proof.** A simple graph $G$ on $k$ vertices can be identified with an upper-triangular $k \times k$ $\mathbb{Z}_2$ matrix $M_G$ with 0s on the diagonal—its adjacency matrix. Two simple graphs $G, G'$ are isomorphic precisely when there exists bijection $\sigma : k \to k'$ s.t.

$$M_G = I_\sigma^T M_G I_\sigma.$$  

The statement thus follows directly from Theorem 5.6.

We conclude the paper with an illustrative, yet simple example of how the algebra of $\text{ABUV}$ can be used to decompose graphs. It is well-known that cliques have unbounded tree-width, but are very simple from the point of view of rank-width: their rank-width is 1. This is easy to observe: suppose that $\{V_1, V_2\}$ is any partition of the vertex set $V$ of a clique. Then the $|V_2| \times |V_1|$ adjacency matrix that describes how vertices of $V_1$ connect to the vertices of $V_2$ is simply the matrix with 1 in each entry, and this matrix has rank 1.

Clique also enjoy a very simple description in $\text{ABUV}$, as we show in the following.

**Example 5.8** [Clique] The clique on $n$ vertices can be constructed as follows

$$\bot \circ c^n \circ \top$$

where $c$ is the following term:

![Diagram](image)

Here we do not define “simplicity”, which is measured with a structural metric called *monoidal width*; it will be introduced in a sequel to this paper. The basic idea is the following: a *monoidal decomposition* is simply a binary tree with internal nodes labelled with $\circ$ or $\oplus$, and leaves generators, identities and twists. The *width* associated to the decomposition tied to the *largest object* along which a composition is ever performed when evaluating the tree as a term of the ambient symmetric monoidal theory. Monoidal width of a term is then the width of an optimal decomposition, i.e. one that has the smallest width.

As we shall show, monoidal width in $\text{ABUV}$ is closely related to rank-width, and concepts such as tree-width can be characterised in a similarly natural way by restricting to sub-PROPs of $\text{ABUV}$. Note also that the definition of monoidal
width is not tied to ABUV and makes sense in any monoidal theory, which solves the problem of generality outlined in the Introduction. Our claims are that (i) ABUV is a canonical, compositional algebra of simple graphs, and (ii) monoidal width is a robust concept that we believe will be useful in a number of different applications areas [1,4,7,10,19] where monoidal theories are used.

References

A model of guarded recursion with clock synchronisation

Aleš Bizjak

Department of Computer Science, Aarhus University, Denmark

Rasmus Ejlers Møgelberg

IT University of Copenhagen, Denmark

Abstract

Guarded recursion is an approach to solving recursive type equations where the type variable appears guarded by a modality to be thought of as a delay for one time step. Atkey and McBride proposed a calculus in which guarded recursion can be used when programming with coinductive data, allowing productivity to be captured in types. The calculus uses clocks representing time streams and clock quantifiers which allow limited and controlled elimination of modalities. The calculus has since been extended to dependent types by Møgelberg. Both works give denotational semantics but no rewrite semantics.

In previous versions of this calculus, different clocks represented separate time streams and clock synchronisation was prohibited. In this paper we show that allowing clock synchronisation is safe by constructing a new model of guarded recursion and clocks. This result will greatly simplify the type theory by removing freshness restrictions from typing rules, and is a necessary step towards defining rewrite semantics, and ultimately implementing the calculus.

Keywords: Guarded recursion, coinductive types, type theory, categorical semantics.

1 Introduction

Guarded recursion [17] is an approach to solving recursive type equations where the type variable appears guarded by a ▶ (pronounced “later”) modal type operator. In particular the type variable could appear positively or negatively or both, e.g. the equation \( \sigma = 1 + ▶(\sigma \rightarrow \sigma) \) has a unique solution [6]. On the term level the guarded fixed point combinator \( \text{fix}_r : (▶τ \rightarrow τ) \rightarrow τ \) satisfies the equation \( f (\text{next} (\text{fix}_r f)) = \text{fix}_r f \) for any \( f : ▶τ \rightarrow τ \). Here \( \text{next} : τ \rightarrow ▶τ \) is an operation that “freezes” an element that we have available now so that it is only available in the next time step.

One situation where guarded recursive types are useful is when faced with an unsolvable type equation. These arise for example when modelling programming

1 Email: abizjak@cs.au.dk
2 Email: mogel@itu.dk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
languages with sophisticated features. In this case a solution to a guarded version of the equation often turns out to suffice, as shown in [6].

But guarded recursive versions of polymorphic type equations are also useful in type theory, even in settings where inductive and coinductive solutions to these equations are assumed to exist. To see this, consider the coinductive type of streams \( \text{Str} \), i.e., the final coalgebra for the functor \( S(X) = \mathbb{N} \times X \). Proof assistants like Coq [14] and Agda [18] allow programmers to construct streams using recursive definitions, but to ensure consistency, these must be \textit{productive}, i.e., one must be able to compute the first \( n \) elements of a stream in finite time. Coq and Agda inspect recursive definitions for productivity by a \textit{syntactic property} that is often overly conservative and does not interact well with higher-order functions.

Using the type of \textit{guarded streams} \( \text{Str}_g \), i.e., the unique type satisfying the equation \( \text{Str}_g = \mathbb{N} \times \nu \text{Str}_g \), one can encode productivity in types: a productive recursive stream definition is exactly a term of type \( \nu \text{Str}_g \rightarrow \text{Str}_g \). To combine the benefits of coinductive and guarded recursive types, Atkey and McBride [3] suggested a simply typed calculus with clock variables \( \kappa \) representing time streams, each with associated \( \nu \kappa \) type constructors, and universal quantification over clocks \( \forall \kappa \). If we think of the type \( \tau \) as being time-indexed along \( \kappa \), then the type \( \forall \kappa.\tau \) contains only elements which are available for all time steps. The relationship between the two notions of streams can then be captured by the encoding of the coinductive stream type as \( \text{Str} = \forall \kappa.\text{Str}_g \kappa \). This encoding works for a general class of coinductive types including those given by polynomial functors, and these results were since extended to the dependently typed setting by Møgelberg [16]. In both cases the encodings were proved sound with respect to a denotational model and no rewrite semantics was given. This paper is part of ongoing work to construct just that.

\textbf{Clock synchronisation}

In the calculus for guarded recursion with clocks, typing judgements are given in a context of clocks \( \Delta \), which is just a finite set of names for clocks, as well as a context of term variables \( \Gamma \). Clock variables \( \kappa \) are simply names, there are no constants or operations on them, and there is no type of clocks. The introduction and elimination rules for \( \forall \kappa \) as defined by Atkey and McBride [3] are

\[
\begin{align*}
\Delta, \kappa, \Gamma \vdash t : \tau & \quad \Delta, \kappa' \vdash t : \forall \kappa.\tau \kappa' \not\in \forall \kappa.\tau \\
\Delta \vdash \Lambda \kappa.t : \forall \kappa.\tau & \quad \Delta, \kappa' \vdash t[\kappa'] : \tau[\kappa'/\kappa]
\end{align*}
\]

(1)

These rules are very similar those for polymorphic types in System F [8], except for the freshness side condition on the elimination rule ensuring that the clocks \( \kappa \) and \( \kappa' \) are not synchronised in \( \tau \). The side condition makes the rule syntactically not well-behaved. For instance it is not clear that \( \beta \)-rule for clock application preserves types.

This becomes a more serious problem in dependent type theory. The rule Møgelberg [16] considers for clock instantiation is

\[
\begin{align*}
\kappa \not\in \text{fc} (\Gamma) & \quad \Delta, \kappa, \Gamma \vdash \tau \\
\Delta, \kappa', \Gamma \vdash t : \forall \kappa.\tau & \quad \Delta, \kappa' \vdash t[\kappa'] : \tau[\kappa'/\kappa]
\end{align*}
\]

217
where the side condition requires that none of the types $\tau$ depends on contain the clock $\kappa$. The reason for the additional clock context $\Gamma'$ is to ensure that the calculus is closed under weakening. However, closure under substitution was overlooked and the rules do not appear to be sufficient to derive the substitution property like

$$\Delta \mid \Gamma, x : \tau \vdash t : \sigma \quad \Delta \mid \Gamma \vdash s : \tau$$

$$\frac{}{\Delta \mid \Gamma \vdash t[s/x] : \sigma[s/x]}$$

which is necessary for a well-behaved dependent type theory.

The restriction on clock instantiation comes from the denotational models of guarded recursion. The original work on guarded recursion [5,6] models a type as a presheaf over the ordered natural numbers, i.e., a diagram of the form

$$X(1) \leftarrow X(2) \leftarrow X(3) \leftarrow \cdots$$

For example, the guarded recursive type of streams satisfying $\text{Str}_g = \mathbb{N} \times \triangleright \text{Str}_g$ is modelled by the presheaf with $X(n) = \mathbb{N}^n$. In this model $\triangleright$ shifts a type one step to the right inserting a singleton set in the end of the sequence.

This model was generalised by Mogelberg [16] (Atkey and McBride [3] use essentially the same idea) to multiple clocks by simply indexing by multiple copies of natural numbers. Thus, conceptually, a type with clocks $\kappa_1$ and $\kappa_2$ was modelled as a two dimensional diagram of sets (as in the left hand part of Figure 1). In this model $\triangleright$ shifts a type one step to the right inserting a singleton set in the end of the sequence.

We propose a new model which supports clock substitution that preserves all the constructs of type theory in the correct way. The model verifies soundness (up to solving the coherence problem, see Section 4) of the rules (1) as understood in dependent type theory, but without the freshness side condition on the elimination rule. In the new model a type depending on two clocks $\kappa_1$ and $\kappa_2$ is modelled as a commutative diagrams of the form in Figure 1: the two dimensional grid on the left represents the type $X$ when clocks $\kappa_1$ and $\kappa_2$ are not identified and the vertical diagram on the right represents the type $X$ when clocks $\kappa_1$ and $\kappa_2$ become
synchronised. The arrows inside the two and one dimensional diagrams describe the evolution of elements when the clocks decrease and the arrows from the diagonal of the diagram on the left to the diagram on the right describe how the elements change when the clocks are synchronised. This also explains why there are no arrows from the vertical diagram on the right to the diagram on the left. Once the clocks are identified there is no way to disentangle them. To model the substitution $\kappa_1/\kappa_2$ we simply take the right vertical part of the diagram.

With more clocks the denotation of a type becomes more complex. For instance when we have three clocks the denotation will have a three dimensional diagram (representing the state when none of the clocks are identified), three two dimensional diagrams (representing the state when two of the clocks are identified) and a one dimensional diagram, representing the state when all of the clocks are identified. Arrows between the different diagrams are given according to the following schema

$$\begin{align*}
\kappa_1 = \kappa_2, \kappa_3 &\quad \kappa_1 = \kappa_3, \kappa_2 &\quad \kappa_1, \kappa_2 = \kappa_3 \\
\kappa_1 = \kappa_2, \kappa_3 &\quad \kappa_1 = \kappa_3, \kappa_2 &\quad \kappa_1, \kappa_2 = \kappa_3 \\
\kappa_1 = \kappa_2 = \kappa_3 &\quad \kappa_1 = \kappa_2, \kappa_3 &\quad \kappa_1, \kappa_2 = \kappa_3
\end{align*}$$

where, for example, $\kappa_1 = \kappa_2, \kappa_3$ represents the diagram where clocks $\kappa_1$ and $\kappa_2$ are identified, and $\kappa_3$ is independent of the two.

**Related work**

The calculus considered in this paper can be understood as a modal variant of sized types [1,2]. The modal aspect of $\forall \kappa$ is investigated by Clouston et. al. [7] which replaces clocks and quantification $\forall \kappa$ by a single comonadic modality $\Box$. This corresponds to having exactly one clock always available. And indeed the calculus is modelled in the topos of trees. The paper provides operational semantics for the calculus and a logic, which is essentially the internal language of the topos of trees with some additional constructs and rules, for reasoning about equality of programs.

**2 Rules of the type theory**

Due to space restrictions we only give a brief overview of some of the type and term constructs which are not part of basic dependent type theory. For details on how to use the terms we refer to Møgelberg [16] and Atkey and McBride [3].

The new types in addition to standard constructs of dependent type theory are

$$\begin{align*}
\Delta | \Gamma \vdash \tau &\quad \kappa \in \Delta \\
\Delta | \Gamma \vdash \Box^\kappa \tau &\quad \Delta, \kappa | \Gamma \vdash \tau &\quad \kappa \notin \Delta \\
\Delta' \subseteq \Delta &\quad \Delta | \Gamma \vdash U_{\Delta'}
\end{align*}$$

The first two rules introduce $\Box^\kappa$ and $\forall \kappa$. type formers. The third rule gives universes. The reason we need universes $U_{\Delta'}$ for each $\Delta' \subseteq \Delta$ is to ensure that they are preserved by clock substitution, in particular by weakening. Clock substitution from clock context $\Delta_1$ to clock context $\Delta_2$ is given by a function $f : \Delta_1 \rightarrow \Delta_2$, e.g. a substitution $\kappa_1/\kappa_2$ from clock context $\kappa_1, \kappa_2$ to clock context $\kappa_1$ is given by the
unique function. We point out in particular how clock substitution on universes is defined, on other constructs it is standard. If \( f : \Delta_1 \to \Delta_2 \) is a clock substitution and \( \Delta' \subseteq \Delta_1 \) then we define \( f(\mathcal{U}_\Delta) = \mathcal{U}_{f[\Delta']} \), where \( f[\Delta'] \) denotes the image of the set \( \Delta' \) by \( f \). For example \((\mathcal{U}_{\kappa_1, \kappa_2})[\kappa_1/\kappa_2] = \mathcal{U}_{\kappa_3} \). Note that this would not make sense if we only had one universe \( \mathcal{U}_\Delta \) in each clock context \( \Delta \), since \( f \) might not be surjective. See Section 3.5 and also Møgelberg [16] for semantic reasons why these additional universes are needed.

The new model

Møgelberg [16] explains in detail how this is done.

The main terms introducing and eliminating the new constructs are

\[
\frac{\Delta \mid \Gamma \vdash t : \tau}{\Delta \mid \Gamma \vdash \text{next}^\kappa t : \text{next}^\kappa \tau} \quad \kappa \in \Delta
\]

\[
\frac{\Delta \mid \Gamma \vdash \Delta, \kappa \mid \Gamma \vdash t : \tau}{\Delta \mid \Gamma \vdash \Lambda \kappa.t : \forall \kappa.\tau} \quad \kappa \not\in \Delta
\]

\[
\frac{\Delta \mid \Gamma \vdash t : \forall \kappa.\tau}{\Delta \mid \Gamma \vdash t[\kappa'] : \forall \kappa'[\kappa'] \quad \kappa' \in \Delta}
\]

\[
\frac{\Delta \mid \Gamma; x : \text{next}^\kappa \tau \vdash t : \tau}{\Delta \mid \Gamma \vdash \text{fix}^\kappa x.t : \tau} \quad \kappa \in \Delta
\]

The constructs \text{next}^\kappa and \oplus^\kappa are part of the applicative functor [15] structure of \( \text{next}^\kappa \). The second line contains introduction and elimination forms for the \( \forall \kappa \) type.

The term \text{fix}^\kappa x.t is the unique fixed point of \( t \).

In addition to standard rules these constructs satisfy type isomorphisms

\[
\forall \kappa.\tau \cong \forall \kappa.\tau \quad \text{if } \kappa \not\in \tau
\]

\[
\forall \kappa.\tau + \forall \kappa.\sigma \cong \forall \kappa. (\tau + \sigma) \quad \forall \kappa.\tau \cong \forall \kappa.\text{next}^\kappa \tau \quad \text{for } \kappa \not\neq \kappa'
\]

which are needed for encoding coinductive types using guarded recursive types.

The directions from left to right are definable in the calculus with only the standard introduction and elimination forms, but the inverses need to be added as additional terms, together with definitional equalities stating that they are inverses. Møgelberg [16] explains in detail how this is done.

3 The new model

We fix a countable set of clocks \( \text{CV} = \{ \kappa_1, \kappa_2, \ldots \} \). The model we construct can be briefly described as follows. We build an indexed category \( \mathfrak{S} \mathcal{R} \), indexed by the opposite of the full subcategory of \( \text{Set} \) on \( \text{finite subsets} \) of \( \text{CV} \). For each finite set of clocks \( \Delta \), the category \( \mathfrak{S} \mathcal{R}(\Delta) \) is a model of extensional dependent type theory: term variable contexts \( \Delta \vdash \Gamma \), types \( \Delta \mid \Gamma \vdash A \) and terms \( \Delta \mid \Gamma \vdash t : A \) are interpreted in \( \mathfrak{S} \mathcal{R}(\Delta) \). For any \( f : \Delta_1 \to \Delta_2 \) the reindexing functor \( \mathfrak{S} \mathcal{R}(f) : \mathfrak{S} \mathcal{R}(\Delta_1) \to \mathfrak{S} \mathcal{R}(\Delta_2) \), which is used to model clock substitution, preserves all the structure required for modeling dependent type theory. Finally, for any inclusion \( i : \Delta \to \Delta, \kappa \) the reindexing functor \( \mathfrak{S} \mathcal{R}(i) : \mathfrak{S} \mathcal{R}(\Delta) \to \mathfrak{S} \mathcal{R}(\Delta, \kappa) \) has a right adjoint \( \forall \kappa \) which is used to interpret quantification over clocks. Due to space restrictions we cannot describe the model in whole, but we only provide definitions
of constructs used to interpret ▶κ, ∀κ and the universes and proof sketches of important points.

3.1 The indexed category \( \mathfrak{S} \mathfrak{R} \)

The category \( \mathfrak{S} \mathfrak{R} (\Delta) \) is the category of presheaves over the poset \( \mathcal{I} (\Delta) \) which we describe first. To understand the definition of the poset \( \mathcal{I} (\Delta) \) it is useful to keep in mind the example in Figure 1. Let \( \Delta \) be a finite set of clocks. An element \( \mathcal{I} (\Delta) \) should indicate what is the state of clocks, i.e. which clock are identified, and it should indicate how much time is left on each clock. Hence elements of \( \mathcal{I} (\Delta) \) should be pairs \((E, \delta)\) of an equivalence relation \( E \) on \( \Delta \) and a function \( \delta : \Delta \rightarrow \mathbb{N} \). Since identified clocks should have the same amount of time remaining, the function \( \delta \) should preserve \( E \). The order on \( \mathcal{I} (\Delta) \) should allow us to get from state represented by \((E, \delta)\) to \((E', \delta')\) whenever \( E' \) identifies more clocks than \( E \) and there is no more time left on \( \delta' \) than on \( \delta \). This makes sense because we want to be able to substitute clocks, and substitution, in general, identifies clocks. On the other hand once the clocks are identified we can no longer separate them, hence we should not be able to get from a state where more clocks are identified to a state where fewer of them are. With this in mind, here are the precise definitions.

**Definition 3.1** For \( \Delta \subseteq ^{\text{fin}} CV \) let \( \mathcal{E} (\Delta) \) be the set of equivalence relations on \( \Delta \) (considered as subsets of \( \Delta \times \Delta \)).

The order relation on \( \mathcal{E} (\Delta) \) is the opposite of the refinement order, concretely \( E \geq E' \iff E \subseteq E' \) (note the reverse inclusion). Or in other words, \( E' \leq E \) if whenever two elements are related by \( E \), they are also related by \( E' \).

The top element for this ordering is the diagonal relation \( \delta_\Delta \). The bottom element is the relation that equates everything.

For a function \( f : \Delta_1 \rightarrow \Delta_2 \) let \( \mathcal{E} (f) : \mathcal{E} (\Delta_2) \rightarrow \mathcal{E} (\Delta_1) \) be the function defined by pullback as \( \mathcal{E} (f) (E) = \{ (\kappa_1, \kappa_2) \mid (f(\kappa_1), f(\kappa_2)) \in E \} \), i.e. clocks \( \kappa_1 \) and \( \kappa_2 \) are related by \( \mathcal{E} (f) (E) \) if they become equated in \( E \) after substitution with \( f \).

**Definition 3.2** Let \( \Delta \) be a finite set of clocks. The poset \( \mathcal{I} (\Delta) \) has elements pairs \((E, \delta)\) where \( E \in \mathcal{E} (\Delta) \) is an equivalence relation and \( \delta : \Delta \rightarrow \mathbb{N} \) is a function that respects \( E \). This means that if \( (\kappa_1, \kappa_2) \in E \) then \( \delta (\kappa_1) = \delta (\kappa_2) \).

The order on \( \mathcal{I} (\Delta) \) is component-wise: \( (E, \delta) \geq (E', \delta') \iff E \geq E' \wedge \delta \geq \delta' \). where the ordering on functions is pointwise.

For a function \( f : \Delta_1 \rightarrow \Delta_2 \) the function \( \mathcal{I} (f) : \mathcal{I} (\Delta_2) \rightarrow \mathcal{I} (\Delta_1) \) is defined as \( \mathcal{I} (f) (E, \delta) = (\mathcal{E} (f) (E), \delta \circ f) \).

**Definition 3.3** Let \( \Delta \) be a finite set of clocks. The category \( \mathfrak{S} \mathfrak{R} (\Delta) \) is the category Set\(^{\mathcal{I} (\Delta)^{op}} \) of (contravariant) \( \mathcal{I} (\Delta) \)-indexed set valued presheaves.

For a function \( f : \Delta_1 \rightarrow \Delta_2 \) let \( \mathfrak{S} \mathfrak{R} (f) : \mathfrak{S} \mathfrak{R} (\Delta_1) \rightarrow \mathfrak{S} \mathfrak{R} (\Delta_2) \) be the functor defined by precomposition with \( \mathcal{I} (f) \). Concretely

\[
\mathfrak{S} \mathfrak{R} (f) (X) = X \circ \mathcal{I} (f) \quad \text{and} \quad \mathfrak{S} \mathfrak{R} (f) (\alpha)_{(E, \delta)} = \alpha_{\mathcal{I} (f) (E, \delta)}
\]

where \( X \) is an object of \( \mathfrak{S} \mathfrak{R} (\Delta_1) \), \( \alpha \) is a natural transformation in \( \mathfrak{S} \mathfrak{R} (\Delta_1) \) and \((E, \delta) \in \mathcal{I} (\Delta_2) \). We will also use \( f^* \) for the functor \( \mathfrak{S} \mathfrak{R} (f) \).

221
For use in Section 3.5 below we record a property of surjective substitutions.

**Lemma 3.4** Let $f : \Delta_1 \to \Delta_2$ be a function between clock contexts. If $f$ is surjective then $E(f)$ and $I(f)$ are injective.

### 3.2 Basic properties of $\mathfrak{GR}$

For each finite set of clocks the category $\mathfrak{GR}(\Delta)$ is a presheaf topos, hence it is a model of extensional dependent type theory. As mentioned above we aim to use the functors $\mathfrak{GR}(f)$ to interpret clock substitution and this means that these functors must preserve constructs used to interpret dependent type theory.

The first property we show is that all the functors $\mathfrak{GR}(f)$ are locally cartesian closed functors. This property is not so straightforward to show and requires some preparations. First, because the functors $\mathfrak{GR}(f)$ are given by precomposition, they have left and right adjoints [13, Theorem VII.2.2]. Hence they preserve all limits and colimits and in fact they preserve the natural choice of these on the nose, a property that simplifies some proofs. To show that they also preserve exponentials and local exponentials we require some preparation.

**Definition 3.5** Let $P$ and $Q$ be two posets. An order-preserving function $\phi : P \to Q$ is a fibration if for every $p \in P$ and $q \in Q$ such that $q \leq \phi(p)$ the set

$$B_{p,q} = \{ p' \leq p \mid \phi(p') = q \}$$

has a top element $u(p,q)$ and moreover whenever $q_1 \leq q_2$, also $u(p,q_1) \leq u(p,q_2)$.

This definition is equivalent to a standard definition of a fibration [10], but we found it useful to have names for the top element $u(p,q)$.

One of the reasons fibrations are useful is the following property.

**Proposition 3.6** Let $P$ and $Q$ be two posets and $\phi : P \to Q$ a fibration. The functor $\phi^* : \text{Set}^{Q^{\text{op}}} \to \text{Set}^{P^{\text{op}}}$ given by precomposition with $\phi$, i.e. $\phi^*(X) = X \circ \phi$, is a locally cartesian closed functor.

**Proof sketch** It is possible to show this directly, but $\phi$ being a fibration implies the assumption of Lemma C.3.3.8.(ii) of Johnstone [11] which shows in particular that the functor $\phi^*$ is locally cartesian closed by Proposition C.3.3.1 of loc. cit. $\square$

Next, the crucial property.

**Lemma 3.7** Let $\Delta_1, \Delta_2$ be two finite sets of clocks and $f : \Delta_1 \to \Delta_2$ a function. Then $E(f) : \mathfrak{E}(\Delta_2) \to \mathfrak{E}(\Delta_1)$ and $I(f) : \mathfrak{I}(\Delta_2) \to \mathfrak{I}(\Delta_1)$ are both fibrations.

**Proof sketch** Let $E \in \mathfrak{E}(\Delta_2)$ and $\mathfrak{E}(\Delta_1) \ni F \leq \mathfrak{E}(f)(E)$. Define $u(E,F) \in \mathfrak{E}(\Delta_2)$ as the transitive closure of the relation

$$E_b = \{ (\kappa, \kappa') \mid (\kappa, \kappa') \in E \vee (\exists (\kappa_1, \kappa_2) \in F, f(\kappa_1) = \kappa \land f(\kappa_2) = \kappa') \}$$

For the second part let $(E, \delta) \in \mathfrak{I}(\Delta_2)$ and $\mathfrak{I}(\Delta_1) \ni (F, \gamma) \leq \mathfrak{I}(f)(E, \delta)$. Define
\(\delta' : \Delta_2 \to \mathbb{N}\) as

\[
\delta'(\kappa) = \begin{cases} 
\gamma(\kappa_1) & \text{if } \exists \kappa_1 \in \Delta_1, (\kappa, f(\kappa_1)) \in E \\
\delta(\kappa) & \text{otherwise}
\end{cases}
\]

Then define \(u((E, \delta), (F, \gamma)) = (u(E, F), \delta')\) where \(u(E, F)\) is the element given by the first part of the proof.

Verification that these are well-defined and satisfy the correct properties requires some work, but we must omit it here due to lack of space.

These two results above allow us to prove the following.

**Theorem 3.8** Let \(f : \Delta_1 \to \Delta_2\) be a function between clock contexts. The functor \(\mathfrak{G}R(f)\) is a locally cartesian closed functor.

**Remark 3.9** As we mentioned already the functors \(\mathfrak{G}R(f)\) do preserve the natural choice of limits and colimits on the nose. However there does not appear to be a natural choice of exponentials or dependent products such that \(\mathfrak{G}R(f)\) would preserve them on the nose. As a consequence we have some technical problems with coherence, which we comment on in Section 4 below.

### 3.3 The ▷\(\kappa\) functors

Let \(\Delta\) be a clock context and \(\kappa \in \Delta\). We now define the functor ▷\(\kappa\) on \(\mathfrak{G}R(\Delta)\) and the natural transformation next\(^\kappa\) : id\(\mathfrak{G}R(\Delta)\) → ▷\(\kappa\) such that the triple \((\mathfrak{G}R(\Delta), ▷\kappa,\text{next}^{\kappa})\) is a model of guarded recursive terms [6, Definition 6.1].

**Example 3.10** To understand the definition recall the diagram \(X\) with two clocks in Figure 1. We wish clock substitution to preserve ▷ in the sense that \((▷^{\kappa_1}▷^{\kappa_2}X)[\kappa_1/\kappa_2]\) is the same as ▷\(^{\kappa_1}\)▷\(^{\kappa_1}\)\(X[\kappa_1/\kappa_2]\) and so the diagram ▷\(^{\kappa_1}\)▷\(^{\kappa_2}\)\(X\) should be

\[
\begin{array}{ccccccccc}
& & & & & & & & & \\
\downarrow & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow & \downarrow & \\
1 & \leftarrow & X(1,2) & \leftarrow & X(2,2) & \leftarrow & \cdots & \leftarrow & X(1) \\
\downarrow & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow & \downarrow & \\
1 & \leftarrow & X(1,1) & \leftarrow & X(2,1) & \leftarrow & \cdots & \leftarrow & 1 \\
\downarrow & \downarrow & \downarrow & \downarrow & \cdots & \downarrow & \downarrow & \downarrow & \\
1 & \leftarrow & 1 & \leftarrow & 1 & \leftarrow & \cdots & \leftarrow & 1 \\
\end{array}
\]

In particular notice that the one dimensional diagram on the left is delayed twice, because it represents the state when \(\kappa_1\) and \(\kappa_2\) are identified.

To define ▷\(\kappa\) in general we start with an auxiliary definition.

**Definition 3.11** Let \(\kappa \in \Delta \subseteq^{\text{fin}} \text{CV}\), \(E \in \mathcal{E}(\Delta)\) and \(\delta : \Delta \to \mathbb{N}\). The function \(\delta^{-\kappa} : \Delta \to \mathbb{N}\) is defined as

\[
\delta^{-\kappa}(\kappa') = \begin{cases} 
\max\{1, \delta(\kappa) - 1\} & \text{if } (\kappa, \kappa') \in E \\
\delta(\kappa') & \text{otherwise}
\end{cases}
\]
The thing to notice in this definition is that all the clocks equivalent to $\kappa$ have their remaining time decreased by 1. This is crucial for clock substitution to commute with $\uparrow^\kappa$ in the appropriate way, as illustrated in Example 3.10 above. Observe that $(E, \delta^{-\kappa}) \leq (E, \delta)$ and this assignment is also order preserving. It is easy to check that if $\delta$ preserves $E$ then so does $\delta^{-\kappa}$, which implies $(E, \delta^{-\kappa}) \in \mathcal{I}(\Delta)$. Moreover, this assignment commutes with reindexing $\mathcal{I}$ in the appropriate way.

**Lemma 3.12** Let $f : \Delta_1 \to \Delta_2$ be a function and $(E, \delta) \in \mathcal{I}(\Delta_2)$. For any $\kappa \in \Delta_1$ the pairs $(\mathcal{E}(f)(E), \delta^{-f(\kappa)} \circ f)$ and $(\mathcal{E}(f)(E), (\delta \circ f)^{-\kappa})$ are in $\mathcal{I}(\Delta_1)$ and moreover they are equal.

The definition of $\uparrow^\kappa : \mathcal{S}\mathcal{R}(\Delta) \to \mathcal{S}\mathcal{R}(\Delta)$ is now simple.

**Definition 3.13** Let $\kappa \in \Delta \subseteq^{\text{fin}} CV$ and $X$ an object of $\mathcal{S}\mathcal{R}(\Delta)$. The action of the functor $\uparrow^\kappa$ on objects is

$$\uparrow^\kappa(X)(E, \delta) = \begin{cases} 1 & \text{if } \delta(\kappa) = 1 \\ X(E, \delta^{-\kappa}) & \text{otherwise} \end{cases}$$

$$\uparrow^\kappa(X)((E_1, \delta_1) \leq (E_2, \delta_2)) = \begin{cases} 1 & \text{if } \delta_1(\kappa) = 1 \\ X((E_1, \delta_1^{-\kappa}) \leq (E_2, \delta_2^{-\kappa})) & \text{otherwise} \end{cases}$$

where $1$ is the singleton set $\{\ast\}$ and $!$ is the unique arrow to $1$. On morphisms

$$\uparrow^\kappa(\alpha)_{E, \delta} = \begin{cases} \text{id}_1 & \text{if } \delta(\kappa) = 1 \\ \alpha_{E, \delta^{-\kappa}} & \text{otherwise} \end{cases}$$

There is an associated natural transformation $\text{next}^\kappa : \text{id}_{\mathcal{S}\mathcal{R}(\Delta)} \to \uparrow^\kappa$

$$\text{next}^\kappa_{E, \delta}(x) = \begin{cases} \ast & \text{if } \delta(\kappa) = 1 \\ X((E, \delta^{-\kappa}) \leq (E, \delta))(x) & \text{otherwise} \end{cases}$$

It is easy to see that $\uparrow^\kappa$ preserves all limits, since these are given pointwise and any limit of any diagram of terminal objects is a terminal object. It does not preserve colimits, however. For example it does not preserve the initial object.

**Proposition 3.14 (Properties of $\uparrow$)** Let $\Delta_1$ and $\Delta_2$ be two clock contexts and $f : \Delta_1 \to \Delta_2$ a function between them. Let $\kappa \in \Delta_1$ be a clock. The following properties hold.

(i) Let $X, Y$ be two objects in $\mathcal{S}\mathcal{R}(\Delta_1)$ and $\alpha : Y \times \uparrow^\kappa(X) \to X$ a natural transformation. There exists a unique $\beta : Y \to X$ such that $\alpha \circ \text{id}_Y \circ \beta = \beta$. We write $\text{fix}^\alpha(\kappa)$ for this unique fixed point. Moreover, for any $\gamma : Z \to Y$ $\text{fix}^\alpha(\kappa) \circ \gamma = \text{fix}^\alpha(\kappa \circ \gamma \times \text{id}_Y)$ which expresses naturality of fixed points.

(ii) Clock substitution preserves $\uparrow$, i.e. $f^* \circ \uparrow^\kappa = \uparrow^{f(\kappa)} \circ f^*$, and for every $X \in \mathcal{S}\mathcal{R}(\Delta_1)$, $f^* (\text{next}^\kappa_X) = \text{next}^{f(\kappa)}_{f^*X}$.

(iii) Let $\alpha : Y \times \uparrow^\kappa X \to X$ be a morphism in $\mathcal{S}\mathcal{R}(\Delta_1)$. From the fact that $f^*$ preserves products on the nose and the previous item the morphism $f^*(\alpha)$ has type $f^*(Y) \times \uparrow^{f(\kappa)} f^*(X) \to f^*(X)$ and moreover $f^* (\text{fix}^\alpha(\kappa)) = \text{fix}^{f(\kappa)} (f^*(\alpha))$. 224
Proof sketch The fixed point $\beta$ at $(E, \delta)$ is defined by induction on $\delta(\kappa)$ as

$$
\beta_{E, \delta}(y) = \begin{cases} 
\alpha_{E, \delta}(y, \ast) & \text{if } \delta(\kappa) = 1 \\
\alpha_{E, \delta}(y, \beta_{E, \delta - \kappa}(Y((E, \delta - \kappa) \leq (E, \delta))(y))) & \text{otherwise}
\end{cases}
$$

Item (ii) is shown by simple unfolding of definitions. Item (iii) is shown by establishing that the term on the left is a fixed point of $f^*(\alpha)$ and then using uniqueness of fixed points.

The facts above show that for each clock context $\Delta$ and $\kappa \in \Delta$, the triple $(\mathcal{GR}(\Delta), \uparrow^\kappa, \text{next}^\kappa)$ is a model of guarded recursive terms [6, Definition 6.1]. Hence for each object $X \in \mathcal{GR}(\Delta)$ the slice category $\mathcal{GR}(\Delta)/X$ also admits a $\uparrow^\kappa_X$ functor defined by pullback [6, Theorem 6.3]

$$
\begin{array}{c}
\uparrow_X^\kappa Y \\ X \downarrow \text{next}^\kappa
\end{array} \quad \xrightarrow{f^*} \\
\begin{array}{c}
\uparrow^\kappa f(Y) \\ \text{next}^\kappa
\end{array}
$$

This comes with the associated morphism $\text{next}^\kappa_X$ in $\mathcal{GR}(\Delta)/X$. Moreover, for $f : \Delta \to \Delta'$ we easily conclude from Proposition 3.14 and the fact that the functor $f^*$ preserves all limits that

$$
f^*(\uparrow_X^\kappa Y) \cong \uparrow_{f(\kappa)} f^*(Y).
$$

and similarly for $f^*(\text{next}^\kappa_X)$ so clock substitution behaves well also with respect to $\uparrow^\kappa$ and $\text{next}^\kappa$ in slices.

3.4 Clock quantification

For any clock context $\Delta$ and $\kappa \notin \Delta$ the inclusion function $\iota : \Delta \to \Delta, \kappa$ gives rise to the weakening functor $\iota^* : \mathcal{GR}(\Delta) \to \mathcal{GR}(\Delta, \kappa)$. Because $\iota^*$ is defined by precomposition with $J(\iota)$ it has a right (as well as left) adjoint [13, Theorem VII.2.2]. We shall call this right adjoint $\forall \kappa$ and in this section we provide a more explicit description of it, which will provide some more intuition behind it and its relation to coinductive types.

To understand the definition it is again useful to consider the case with two clocks from Figure 1. The object $\forall_{\kappa_2}X$ is a one dimensional diagram and at stage $n$ it is the limit (in Set) of the diagram

$$
X(n, 1) \leftarrow X(n, 2) \leftarrow X(n, 3) \leftarrow X(n, 4) \leftarrow \cdots
$$

The idea is that the type $(\forall_{\kappa_2}X)(n)$ contains information about $X(n, k)$ for all times $k$. Note that in particular the one dimensional diagram which represents the state of $X$ when the clocks $\kappa_1$ and $\kappa_2$ are identified is ignored. This is because the clock $\kappa_2$ is no longer free and no substitution will able to equate it to some other clock, i.e. substitution is capture avoiding.

To define the right adjoint of the inclusion in general we need some auxiliaries.
Lemma 3.15 Let $\Delta$ be a clock context and $\iota : \Delta \to \Delta, \kappa$ the inclusion. Then $\mathcal{E}(\iota) : \mathcal{E}(\Delta, \kappa) \to \mathcal{E}(\Delta)$ has a right adjoint $\iota^!$ defined explicitly as
\[\iota^!(E) = E \cup \{(\kappa, \kappa)\}.\]

In contrast the function $\mathcal{I}(\iota)$ does not have a right adjoint, the reason being that $\mathbb{N}$ does not have a top element. However for each $n \in \mathbb{N}$ we can define a function $\iota^!_n$ using the explicit description of $\iota^!$ in Lemma 3.15 it is easy to see that $\delta^!_n$ preserves $\iota^!(E)$. We record some useful properties for use below.

Lemma 3.16 Let $\Delta$ be a clock context, $\kappa \notin \Delta$ and $\iota : \Delta \to \Delta, \kappa$ the inclusion

(i) If $n \leq m$ and $(E, \delta) \leq (E', \delta')$ then $\iota^!_n(E, \delta) \leq \iota^!_m(E', \delta').$

(ii) For any $(E, \delta) \in \mathcal{I}(\Delta, \kappa)$ we have $(E, \delta) \leq \iota^!_{\delta(\kappa)}(\mathcal{I}(\iota)(E, \delta)).$

(iii) For any $(E, \delta) \in \mathcal{I}(\Delta)$ and any $n \in \mathbb{N}$ we have $\mathcal{I}(\iota)(\iota^!_n(E, \delta)) = (E, \delta).

(iv) For any $(E, \delta) \in \mathcal{I}(\Delta, \kappa)$ and $\kappa' \in \Delta$, $\delta^!_n(\kappa') = \frac{\delta(\kappa')}{n}$ if $\kappa' \in \Delta$.

We are now ready to describe the right adjoint $\forall \kappa$ to $\iota^*$. Let $\Delta$ be a clock context, $\kappa$ a clock not in $\Delta$ and $\iota : \Delta \to \Delta, \kappa$ the inclusion.

Define $\forall \kappa : \mathfrak{DR}(\Delta, \kappa) \to \mathfrak{DR}(\Delta)$ on an object $X \in \mathfrak{DR}(\Delta, \kappa)$ at stage $(E, \delta) \in \mathcal{I}(\Delta)$ by taking the limit (in Set) of the diagram of restrictions
\[X((\iota^!_1(E, \delta)) \leftarrow X((\iota^!_2(E, \delta)) \leftarrow X((\iota^!_3(E, \delta)) \leftarrow \cdots\]

where the arrows are $X$’s restrictions using Lemma 3.16. The restrictions of $\forall \kappa(X)$ and the action of $\forall \kappa$ on morphisms are determined purely formally from the universal property of limits. The unit $\eta$ of the adjunction is constructed using the universal property of the limit using Lemma 3.16.(iii) which shows that the diagram
\[\iota^*(X)((\iota^!_1(E, \delta)) \leftarrow \iota^*(X)((\iota^!_2(E, \delta)) \leftarrow \iota^*(X)((\iota^!_3(E, \delta)) \leftarrow \cdots\]

is a constant diagram. The counit $\varepsilon$ is constructed with the projections of the limit together with Lemma 3.16.(ii). Since the diagram is in Set we could describe the limit very explicitly as the set of compatible sequences. This is useful for checking some properties, but we omit it here due to lack of space.

Equipped with this description of $\forall \kappa$ we are able to show the necessary properties for interpreting the rules of the type theory.

Proposition 3.17 (Properties of $\forall \kappa$) Let $\Delta$ be a clock context and $\kappa \in \mathcal{C} \forall \kappa$ a clock not in $\Delta$. The functor $\forall \kappa$ satisfies

(i) The unit $\eta$ of the adjunction $\iota^* \vdash \forall \kappa$ is a natural isomorphism. Hence $\iota^*$ is a full and faithful functor witnessing that $\mathfrak{DR}(\Delta)$ is a full subcategory of $\mathfrak{DR}(\Delta, \kappa)$. 226
(ii) The functor \( \forall \kappa \) preserves all coproducts, but not colimits in general.

(iii) For any object \( X \in \mathcal{G} \mathcal{O}(\Delta, \kappa) \) the canonical morphism \( c : \forall \kappa.X \to \forall \kappa.(\kappa^* X) \) defined as \( c = \forall \kappa.(\text{next}^\kappa) \) is an isomorphism.

(iv) (Beck-Chevalley condition for \( \forall \kappa \)) Let \( f : \Delta_1 \to \Delta_2 \) be a function between two clock contexts, and let \( \kappa \notin \Delta_1 \cup \Delta_2 \) be a clock. Let and \( \iota_1 : \Delta_1 \to \Delta_1, \kappa \) and \( \iota_2 : \Delta_2 \to \Delta_2, \kappa \) be the two inclusions. For every \( X \in \mathcal{G} \mathcal{O}(\Delta_1, \kappa) \) the presheaves \( f^*(\forall \kappa.X) \) and \( \forall \kappa.(f + id_\kappa)^*(X) \) are equal and the canonical morphism \( \forall \kappa.((f + id_\kappa)^*(\varepsilon)) \circ \eta^*(\forall \kappa.X) \) from \( f^*(\forall \kappa.X) \) to \( \forall \kappa.(f + id_\kappa)^*(X) \) is the identity.

(v) Let \( \Delta \) be a clock context, \( \kappa' \in \Delta \), \( \kappa \notin \Delta \) and \( X \in \mathcal{G} \mathcal{O}(\Delta, \kappa) \) the canonical morphism \( \forall \kappa.(\kappa^\prime(\varepsilon)) \circ \eta : \kappa^\prime(\forall \kappa.X) \to \forall \kappa.\kappa^\prime X \) is an isomorphism.

Proof sketch

(i) Using Lemma 3.16.(iii) the object \( \forall \kappa.t^*(X) \) at stage \((E, \delta)\) is the limit of the constant diagram (2). Because the diagram is connected its limit is isomorphic to \( X(E, \delta) \) by the unique mediating map, which is by definition the unit \( \eta \). The second part is a standard fact about adjoint functors [12, Theorem IV.3.1].

(ii) The reason this property holds is that coproducts are given pointwise and that in Set coproducts commute with connected limits.

(iii) The arrow \( \forall \kappa.(\text{next}^\kappa) \) at stage \((E, \delta) \in \mathcal{I}(\Delta)\) is by definition the mediating map from the limit of

\[
X(\iota_1(E, \delta)) \leftarrow X(\iota_2(E, \delta)) \leftarrow X(\iota_3(E, \delta)) \leftarrow \cdots
\]

to the limit of

\[
1 \leftarrow X(\iota_1(E, \delta)) \leftarrow X(\iota_2(E, \delta)) \leftarrow X(\iota_3(E, \delta)) \leftarrow \cdots
\]

so it is an isomorphism.

(iv) The proof is somewhat technical due to the amount of notation involved, but essentially straightforward. Lemma 3.16 is used.

(v) Follows by computation and Lemma 3.16.(iv). Note that to even state it Proposition 3.14 is used to get \( t^* \circ \kappa^\prime = \kappa^\prime \circ t^* \) so we could apply the counit \( \varepsilon \). \( \square \)

Extension of \( \forall \kappa \) to slices proceeds exactly as before [16, Proposition 1]. The interpretation of the clock instantiation \( t[\kappa'] \) now proceeds as follows. A term \( \Delta \vdash \Gamma \vdash t : \forall \kappa.\tau \) corresponds to a morphism from \((\text{the interpretation of}) \\Gamma \to \forall \kappa.\tau \) in \( \mathcal{G} \mathcal{O}(\Delta) \). Transposing along the adjunction \( \iota^* \dashv \forall \kappa \) we get a morphism \( t' \) from \( \iota^*(\Gamma) \to \tau \) in \( \mathcal{G} \mathcal{O}(\Delta, \kappa) \). Let \( f : \Delta, \kappa \to \Delta \) be the identity on \( \Delta \) and map \( \kappa \) to \( \kappa' \). Applying \( \mathcal{G} \mathcal{O}(f) \) to \( t' \) we get a morphism from \( \mathcal{G} \mathcal{O}(f)(\mathcal{G} \mathcal{O}(\iota)(\Gamma)) \) to \( \mathcal{G} \mathcal{O}(f)(\tau) \) in \( \mathcal{G} \mathcal{O}(\Delta) \) which we define to be the interpretation of \( t[\kappa'] \). Notice that \( \mathcal{G} \mathcal{O}(f)(\mathcal{G} \mathcal{O}(\iota)(\Gamma)) \) is just \( \Gamma \) and by definition \( \mathcal{G} \mathcal{O}(f)(\tau) \) is the interpretation of \( \tau[\kappa'/\kappa] \), so the interpretation is consistent.

Remark 3.18 This interpretation is standard, see e.g. Jacobs [10], but note that it is crucial that we have general clock substitution \( \mathcal{G} \mathcal{O}(f) \), for arbitrary \( f \), and this is
precisely the ingredient that was missing in previous models, hence the restrictions on clock instantiation rules.

3.5 Universes

We follow previous work [5,16] and use Hofmann and Streicher’s construction of universes in presheaf toposes from universes in Set [9] which we now recall instantiated to our special case. We first recall what a semantic universe is.

Definition 3.19 Let $\mathbb{C}$ be a locally cartesian closed category with coproducts and $\text{el} : E \to U$ a morphism in $\mathbb{C}$. A morphism $f : A \to \Gamma$ is small with respect to $\text{el}$ if there is a morphism $f : \Gamma \to U$ such that $f$ is appears as the pullback of $\text{el}$ along $f$. The morphism $f$ is called a code of $f$. An object $\Gamma$ is small if the unique map $\Gamma \to 1$ is small.

The map $\text{el}$ is a universe if the objects 0, 1, $\mathbb{N}$ are small and the notion of smallness is closed under composition, finite coproducts and small dependent products.

Let $U$ be a Grothendieck universe in Set such that $\mathbb{N} \in U$ and let $\Delta$ be a finite set of clocks.

Definition 3.20 The presheaf $V^\Delta \in \mathfrak{SR}(\Delta)$ is defined as $V^\Delta(E, \delta) = U(\downarrow E, \delta)^{op}$ where $\downarrow (E, \delta)^{op}$ is the set of elements of $J(\Delta)$ below $(E, \delta)$ and $U(\downarrow E, \delta)^{op}$ is the set of presheaves $D$ on $\downarrow (E, \delta)^{op}$ such that for all $(E', \delta') \leq (E, \delta)$ we have $D(E', \delta') \in U$.

The action of $V^\Delta$ on morphisms is by precomposition: $V^\Delta((E_1, \delta_1) \leq (E_2, \delta_2)) (D) = D \circ \iota$ where $\iota$ is the inclusion of $\downarrow (E_1, \delta_1)$ to $\downarrow (E_2, \delta_2)$.

The presheaf of elements $E^V_\Delta$ is defined as $E^V_\Delta(E, \delta) = \sum_{D \in V^\Delta(E, \delta)} D(E, \delta)$ with restrictions $E^V_\Delta((E_1, \delta_1) \leq (E_2, \delta_2)) (D, x) = (D \circ \iota, D((E_1, \delta_1) \leq (E_2, \delta_2))(x))$.

The universe is the first projection $u^\Delta : E^V_\Delta \to V^\Delta$ defined as $u^\Delta_{E,\delta}(D, x) = D$.

Hofmann and Streicher [9] show that the universe $u^\Delta$ is closed under the usual constructs used to model dependent type theory, provided $U$ is. What remains is to show that they are also closed under $\forall \kappa$ and $\forall^* \kappa$ and that they are suitably preserved by reindexing functors $\mathfrak{SR}(f)$. The first two of these properties follow exactly as before [16] so we focus on the last.

The functors $\mathfrak{SR}(f)$ do not in general preserve the universes. In particular the inclusion $\iota^* : \mathfrak{SR}(\Delta) \to \mathfrak{SR}(\Delta, \kappa)$ does not map $V^\Delta$ to (an object isomorphic to) $V^{\Delta,\kappa} \in \mathfrak{SR}(\Delta, \kappa)$. However surjective substitutions do preserve universes in the appropriate sense.

Lemma 3.21 Let $s : \Delta \to \Delta'$ be a surjective function between clock contexts $\Delta$ and $\Delta'$. There exist natural isomorphisms $c^V : s^* (V^\Delta) \to V^{\Delta'}$ and $c^E : s^* (E^V_\Delta) \to E^V_{\Delta'}$ such that the diagram

$$
\begin{array}{ccc}
\downarrow s^* \downarrow u^\Delta & & \downarrow \downarrow u^\Delta' \\
E^V_{\Delta'} & & V^{\Delta'}
\end{array}
$$

commutes.
Proof sketch From Lemma 3.4 we have \( \mathcal{J}(s) \) injective and from Lemma 3.7 we have that \( \mathcal{J}(s) \) is a fibration. Thus \( \mathcal{J}(s) \) restricted to a function \( \downarrow (E, \delta) \rightarrow \downarrow \mathcal{J}(s)(E, \delta) \) is a bijection with an order preserving inverse given by the assignment \( u((E, \delta), -) \).

Moreover, because the bijection is given by a restriction of a single function \( \mathcal{J}(s) \) it is natural in \((E, \delta)\). We thus have

\[
s^*(V^\Delta)(E, \delta) = V^\Delta(\mathcal{J}(s)(E, \delta)) = \bigcup \downarrow \mathcal{J}(s)(E, \delta)^{op} \cong \bigcup \downarrow (E, \delta)^{op} = V^\Delta'(E, \delta)
\]

where the bijection \( \bigcup \downarrow \mathcal{J}(s)(E, \delta)^{op} \cong \bigcup \downarrow (E, \delta)^{op} \) is natural in \((E, \delta)\). Thus \( s^*(V^\Delta) \cong V^\Delta' \) as presheaves in \( \mathfrak{R}(\Delta') \). The map \( c^E \) is defined similarly. \( \square \)

Remark 3.22 Inspection of the proof also shows why for the inclusion \( \iota : \Delta \rightarrow \Delta, \kappa, \) the reindexing \( \iota^* \) does not preserve universes in this way. This is consistent with the situation as it was in Mogelberg’s previous model [16] and so following loc. cit. we add additional universes in each \( \mathfrak{R}(\Delta) \).

Definition 3.23 Let \( \Delta \) and \( \Delta' \) be clock contexts such that \( \Delta' \subseteq \Delta \). Let \( \iota : \Delta' \rightarrow \Delta \) be the inclusion. We define the universe \( (u^\Delta_\Delta, \mathcal{E}^\Delta_\Delta, U^\Delta_\Delta) \) as

\[
U^\Delta_\Delta = \iota^*(V^\Delta'), \quad \mathcal{E}^\Delta_\Delta = \iota^*(E^V_\Delta), \quad u^\Delta_\Delta = \iota^*(u^\Delta').
\]

Theorem 3.24 The triple \( (u^\Delta_\Delta, \mathcal{E}^\Delta_\Delta, U^\Delta_\Delta) \) is a universe closed under dependent products, sums, \( \forall \kappa \) and \( \triangleright^\kappa \).

Proof sketch To see that the notion of smallness is closed under dependent product and sum one uses the fact that \( \iota^* \) is an LCC functor (Theorem 3.8) and the fact that a universe is closed under dependent products if a particular generic map is small and this generic map can be constructed using only the LCC structure, hence it is preserved by \( \iota^* \). The same approach works for dependent sums.

Closure under \( \triangleright \) follows by first showing that the universes \( V^\Delta \) have codes \( \triangleright^\kappa \) for \( \triangleright^\kappa \) and then deriving codes for \( U^\Delta_\Delta \) from these using Proposition 3.14. Closure under \( \forall \kappa \) is also shown first for universes \( V^\Delta \) and then using the Beck-Chevalley condition (Proposition 3.17) for \( U^\Delta_\Delta \). See Mogelberg [16] for more details. \( \square \)

Finally, these additional universes are preserved by clock substitution in the appropriate way.

Proposition 3.25 Let \( f : \Delta_1 \rightarrow \Delta_2 \) be a function between clock contexts \( \Delta_1 \) and \( \Delta_2 \). Let \( \Delta' \subseteq \Delta_1 \) be another clock context and \( (u^\Delta_1, \mathcal{E}^\Delta_1, U^\Delta_1) \) the universe from Definition 3.23. There exist two natural isomorphisms \( c^f_\Delta \) and \( c^f_\triangleright \) such that the diagram

\[
\begin{array}{ccc}
 f^* \mathcal{E}^\Delta_1 & \xrightarrow{c^f_\mathcal{E}} & \mathcal{E}^\Delta_2 \\
 f^*(u^\Delta_1) & \Downarrow & f^*(u^\Delta_2) \\
 f^* U^\Delta_1 & \xrightarrow{c^f_\triangleright} & U^\Delta_2
\end{array}
\]

commutes. In particular, \( f^*(U^\Delta_1) \cong U^\Delta_2 \).

229
Proof Let $\iota_1$ be the inclusion of $\Delta'$ into $\Delta_1$ and $\iota_2$ the inclusion of $f[\Delta']$ into $\Delta_2$. Let $s : \Delta' \to f[\Delta']$ be the restriction of $f$. By definition $s$ is surjective and $f \circ \iota_1 = \iota_2 \circ s$ and so $f^* \circ \iota_1^* = \iota_2^* \circ s^*$. Lemma 3.21 gives natural isomorphisms $c^V$ and $c^E$ such that the diagram on the left

\[
\begin{array}{c}
s^* \left( E^V_{\Delta'} \right) \xrightarrow{c^E} E^V_{f[\Delta']}
s^* \left( V^\Delta \right) \xrightarrow{c^V} V^f[\Delta']
s^* \left( u^\Delta \right) \downarrow \downarrow u^f[\Delta']
\end{array}
\]

\[
\begin{array}{c}
\iota_2^* \left( s^* \left( E^V_{\Delta'} \right) \right) \xrightarrow{\iota_2^*(c^E)} \iota_2^* \left( E^V_{f[\Delta']} \right) \\
\iota_2^* \left( s^* \left( V^\Delta \right) \right) \xrightarrow{\iota_2^*(c^V)} \iota_2^* \left( V^f[\Delta'] \right) \\
\iota_2^* \left( u^\Delta \right) \downarrow \downarrow u^f[\Delta']
\end{array}
\]

commutes. Hence the diagram on the right commutes and the vertical morphisms are isomorphisms. But notice that, e.g. $\iota_2^* \left( s^* \left( V^\Delta \right) \right) = f^* \left( \iota_1^* \left( V^\Delta \right) \right) = f^* \left( \mathcal{U}_{\Delta_1}^\Delta \right)$ and also by definition $\iota_2^* \left( V^f[\Delta'] \right) = \mathcal{U}_{f[\Delta'_1]}^\Delta$. This concludes the proof. $\square$

4 Conclusions and future work

We have sketched (up to solving the coherence problem) that allowing clock synchronisation retains soundness by constructing a model which validates it. With regards to the coherence problem, we can certainly solve it in each $\mathfrak{SR}(\Delta)$, so that substitution of terms into types and terms behaves correctly. However we also need to interpret clock substitution, which we do using the functors $\mathfrak{SR}(f)$ for functions $f : \Delta_1 \to \Delta_2$ between clock contexts. And in order to validate equalities such as $\llbracket \Delta_2 \vdash f(\Gamma) \rrbracket = \mathfrak{SR}(f) \left( \llbracket \Delta_1 \vdash \Gamma \rrbracket \right)$ we would require $\mathfrak{SR}(f)$ to preserve our choice of interpretation of all the constructs on the nose, but it only does so up to canonical isomorphism.

We believe this is a technical, rather than essential, problem with the particular presentation. In particular, without universes, we do have a solution to the coherence problem by replacing the categories $\mathfrak{SR}(\Delta)$ by equivalent ones obtained by the Bénabou construction [4] (see also [10, Corollary 5.2.5]). This then allows us to make choices of structure that are preserved on the nose by functors interpreting clock substitution. However doing this breaks type equalities like $\text{El}(\text{in}(t)) \simeq \text{El}(t)$ where in is a universe inclusion from $\mathcal{U}_{\Delta_1}^\Delta$ to $\mathcal{U}_{\Delta_2}^\Delta$, for instance. The types on the left and right are only interpreted as isomorphic objects, not equal.

We are working on giving computational meaning to various type isomorphisms validated by the model and required for working with coinductive types via guarded recursive types. Removing the “freshness” requirements in clock instantiation rule considerably simplifies the syntactic theory.

Acknowledgement

We thank Lars Birkedal for helpful discussions. Aleš Bizjak is supported in part by a Microsoft Research PhD grant.
References


Stateful runners of effectful computations

Tarmo Uustalu

Institute of Cybernetics, Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia

Abstract

What structure is required of a set so that computations in a given notion of computation can be run statefully with this set as the state set? For running nondeterministic computations statefully, a resolver structure is needed; for interactive I/O computations, a “responder-listener” structure is necessary; to be able to serve stateful computations, the set must carry the structure of a lens. We show that, in general, to be a stateful runner of computations for a monad corresponding to a Lawvere theory (defined as a set equipped with a monad morphism between the given monad and the state monad for this set) is the same as to be a comodel of the theory, i.e., a coalgebra of the corresponding comonad. We work out a number of instances of this observation and also compare runners to handlers.

Keywords: effects, monads, Lawvere theories, comodels, state monads, handlers

1 Introduction

This paper is about Moggi’s monad-based and Plotkin and Power’s Lawvere theories based approaches to effectful computation [8,10].

Given a monad \((T, \eta, \mu)\), a computation of a value in \(X\) is an element of \(TX\). Computations are there to compute values, so we consider it natural to wish to extract these values, to run computations. Ideally, we might want to have at our disposal a polymorphic function \(\theta : \forall X. TX \to X\) for extracting values from computations, but this is generally too much to ask (although it is possible, e.g., for writer monads).

However we can often produce a value, if we are allowed to rely on some input—think of it as an initial state—drawn from some set \(C\) with suitable structure. For example, if we have a finitely nondeterministic computation in the sense of a binary wellfounded leaf tree, a bitstream can be used to identify a leaf. As running should reasonably be compositional in the sense that running the sequence of two computations should be the same as composing two runs, a run should not only depend on an initial state, but also return a final state (that can serve as the initial state for another run). In the case of nondeterminism and bitstreams, the final state
could be the remainder of the bitstream provided as the initial state. So in general we might want to look for a polymorphic function $\theta : \forall X. T X \to T^C X$ where $(T^C, \eta^C, \mu^C)$ is the state monad for $C$ as the state set. The compositionality we want amounts to $\theta$ being not just a natural transformation, but a monad morphism.

In this paper, we answer the question of when a set $C$ can be used to run computations in a monad $(T, \eta, \mu)$ statefully, assuming that the monad corresponds to a Lawvere theory. The answer is: $C$ has to carry a comodel of the Lawvere theory (i.e., a coalgebra of the corresponding comonad). We spell out a number of instances of this generality, for nondeterminism, interactive I/O and stateful computations. This is an easy exercise, but the results are quite instructive, we find. For some versions of nondeterminism, for instance, runners can only recover a part of the information in a given computation; other versions of nondeterminism admit only trivial runners that reveal nothing about the computation. So some variations of nondeterminism are inherently more operational than others.

Runners are somewhat similar to handlers, but one bigger difference is that runners are polymorphic in the value set. For example, handling allows one to extract a value from a nondeterministic computation (a binary wellfounded leaf tree) over a specific value set that carries a binary operation by folding this operation over the leaf labels. (If for us a nondeterministic computation is a nonempty list of values, this operation must be associative.) Running does not allow such things. In our view, the pragmatics of handlers and runners are different: handlers are a programming language construct, but runners are compilation schemes.

The paper is organized as follows. In Section 2, we review the few basic facts about Lawvere theories, models and comodels that we need. In Section 3, we show that stateful runners for a monad corresponding to a Lawvere theory are in a bijection with comodels of the theory (coalgebras of the corresponding comonad). We also compare this observation to a fact about monad morphisms to continuation monads—a different type of runners. In Section 4, we work out the instances for nondeterminism, interactive I/O and stateful computation. Just before concluding, in Section 5, we compare runners to handlers.

2 Lawvere theories, models, comodels

We begin by reviewing the most basic definitions and facts about finitary Lawvere theories and models (for a proper exposition, see, e.g., [6]) as well as Power’s comodels [13,11]. Countable Lawvere theories are analogous.

Lawvere theories

A (finitary) Lawvere theory is given by a small category $L$ with finite products and a functor $L : F^{op} \to L$ that is identity on objects and strictly preserves the finite products of $F^{op}$.

Here $F$ is the category of finite cardinals, i.e., the skeleton of the category of finite sets. It is a strict monoidal category wrt. finite coproducts, in fact it is the free such category on the one-object category.

A theory can (non-uniquely) be specified by a presentation, i.e., by some subset of the maps $O_P^f : I^f \to O^f$ of $L$ (operations) from which all other maps are definable.
together with some subset of the commuting diagrams \( \text{LHS}_k = \text{RHS}_k \) of \( \text{L} \) (equations) from which all other commuting diagrams follow.

Notice that one can always do with operations of arities \( I \to 1 \) only. As \( O = \prod_{o \in O} 1 \), any operation \( \text{OP} : I \to O \) can be replaced with \( O \) many operations \( \text{OP}^o : I \to 1 \) via \( \text{OP}^o = \text{in}_o \circ \text{OP} \) and \( \text{OP} = (\prod_{o \in O} \text{OP}^o) \circ \nabla \).\(^2\)

**Models**

A model of a theory \((\mathbb{L}, \mathbb{L})\) is given by a functor \( \mathbb{J}^{\mathbb{K}} : \mathbb{L} \to \text{Set} \) that preserves the finite products of \( \mathbb{L} \) (non-strictly).

To give a model (up to isomorphism) of \((\mathbb{L}, \mathbb{L})\), it suffices to specify a set \( A = \mathbb{[1]} \) since, for any other object \( Y \), we have \( \mathbb{[Y]} = \prod_{y \in Y} 1 \cong \prod_{y \in \mathbb{Y}[1]} = [1] \) \( \Leftarrow Y \)\(^3\) together with functions

\[
\text{op}_j = \Lambda^{-1} [\text{OP}_j] : O_j \times ([1] \Leftarrow I_j) \to [1]
\]

since, for any other map \( f, [f] \) is then uniquely determined by functoriality and preservation of finite products. Moreover, any set \( A \) with functions \( \text{op}_j : O_j \times (A \Leftarrow I_j) \) defines a model, provided that the equations \( \text{lhs}_k = \text{rhs}_k \) hold for the derived functions \( \text{lhs}_k, \text{rhs}_k \).

**Theories and monads**

A theory defines a unique monad whose algebras are essentially the same as its models. The underlying functor is

\[
T X = T_0 X/\sim_X
\]

where \( T_0 X \) is the set defined inductively by

\[
\begin{align*}
\var x : X & \quad \text{var} : T_0 X \quad o : O_j 
\end{align*}
\]

(\( f : T_0 X \Leftarrow I_j \)) and \( \sim_X \) is a binary relation on \( T_0 X \) defined inductively by

\[
\forall i : I_i, f i \sim_X f' i \quad \text{op}_j (o, f) \sim_X \text{op}_j (o, f') \quad \text{lhs}_k (o, f) \sim_X \text{rhs}_k (o, f)
\]

The unit \( \eta \) is \( \text{var} \) and the multiplication \( \mu \) is defined recursively by \( \mu (\text{var} t) = t \) and \( \mu (\text{op}_j (o, f)) = \text{op}_j (o, \lambda i. \mu (f i)) \).

\(^2\) We write maps of \( \mathbb{L} \) in terms of the operations of the presentation, maps of \( \mathbb{F} \), the product bifunctor of \( \mathbb{L} \), which we denote \( + \) (sic!) to agree with the notation for the coproducts of \( \mathbb{F} \), and composition of \( \mathbb{L} \). Note that maps of \( \mathbb{F} \) have their directions reversed in \( \mathbb{L} \), so \( \text{in}_o : 1 \to O \) in \( \mathbb{F} \), but \( \text{in}_o : O \to 1 \) in \( \mathbb{L} \), etc.

\(^3\) To avoid the need to explicitly use the symmetry of \( \times \) in the examples, we will use two exponential functors \( \Leftarrow \) and \( \Rightarrow \); think of \( \Leftarrow Y \) as the right adjoint of \( Y \times - \) and \( Y \Rightarrow - \) as the right adjoint of \( - \times Y \).
Algebras of \((T, \eta, \mu)\) are essentially the same as models. A model \((A, (\text{op}_j : O_j \times (A \Leftarrow I_j) \to A))\) and an algebra \((A, \alpha : TA \to A)\) are interdefinable recursively by \(\alpha(\text{var } x) = x\) and \(\alpha(\text{op}_j (o, f)) = \text{op}_j (o, \lambda i. \alpha(f i))\) and by \(\text{op}_j (o, f) = \alpha(\text{op}_j (o, \lambda i. \text{var } (f i)))\).

The monad corresponding to a theory is finitary. And every finitary monad corresponds to exactly one theory in this fashion.

**Comodels**

We are now prepared to discuss Power’s notion of comodels.

A comodel of a theory \((L, L : \mathbb{F}^{op} \to \mathbb{L})\) is given by a functor \(\llangle - \rrangle : \mathbb{L}^{op} \to \text{Set}\) that preserves the finite coproducts of \(\mathbb{L}^{op}\) (recall that a model was a functor from \(\mathbb{L}\) preserving its finite products).

To give a comodel (up to isomorphism), it suffices to specify a set \(C = \llangle 1 \rrangle\) since \(\llangle X \rrangle = \llangle \coprod_{x \in X} 1 \rrangle = \llangle 1 \rrangle \times X\), together with functions \(\text{op}_j = \llangle \text{op}_j \rrangle : \llangle 1 \rrangle \times O_j \to \llangle 1 \rrangle \times I_j\) where we often split \(\text{op}_j\) as \(\langle \text{op}_n, \text{op}_s \rangle\) (with \(n\) and \(s\) mnemonic for “next” and “show”). Also, any set \(C\) with functions \(\text{op}_j : C \times O_j \to C \times I_j\) defines a comodel, provided that the equations \(\text{lhs}_k = \text{rhs}_k\) hold for the derived functions \(\text{lhs}_k, \text{rhs}_k\).

**Theories and comonads**

Besides defining a (finitary) monad whose algebras are the same as models, a theory also defines a comonad with the property that comodels of the theory are the same as coalgebras of the comonad.

The comonad corresponding to a theory \((\mathbb{L}, L)\) is a subcomonad of a cofree comonad. The underlying functor is defined by

\[
DX = D_0 X \mid \text{ok}_X
\]

where \(D_0 X\) is a set defined coinductively by

\[
\begin{align*}
\text{var} d : D_0 X & \quad d : D_0 X \quad o : O_j \\
\text{op}_j (d, o) : D_0 X & \quad \text{op}_j (d, o) : I_j
\end{align*}
\]

(so that, in a more compact notation, \(D_0 X \cong \nu Z. X \times \prod_j (O_j \Rightarrow Z \times I_j)\)) and \(\text{ok}_X\) is a predicate on \(D_0 X\) defined coinductively by

\[
\begin{align*}
\text{ok}_X d & \quad \text{ok}_X (\text{op}_j (d, o)) = \text{ok}_X d = \text{ok}_X d \\
\text{ok}_X (\text{op}_j (d, o)) & \quad \text{ok}_X d = \text{ok}_X d
\end{align*}
\]

The counit \(\varepsilon\) is \(\text{var}\). The comultiplication \(\delta\) is defined corecursively by \(\text{var} (\delta d) = d\), \(\text{op}_j (\delta d, o) = \delta (\text{op}_j (d, o))\), \(\text{op}_j (\delta d, o) = \text{op}_j (d, o)\).

---

4 We write coinductive definitions in a destructor-based fashion (as opposed to the constructor-based style commonly used in proof assistants), as this works more smoothly.
A comodel \((C, \overline{op}_{j} : C \times O_{j} \to C \times I_{j})_{j}\) is interdefinable with a coalgebra \((C, \gamma : C \to DC)\) corecursively by \(\text{var} (\gamma c) = c, \overline{opn}_{j}(\gamma c, o) = \gamma \overline{opn}_{j}(c, o), \overline{ops}_{j}(\gamma c, o) = \overline{ops}_{j}(c, o), \) and by \(\overline{op}_{j}(c, o) = \varnothing \overline{op}_{j}(\gamma c, o), \overline{ops}_{j}(c, o) = \overline{ops}_{j}(\gamma c, o).\)

The comonad corresponding to a theory in the above fashion is in all but the simplest cases non-finitary. Also, one comonad can correspond to many theories. (E.g., all theories with at least one nullary operation \((op : 0 \to O\) with \(O \neq 0\) have the initial comonad \((DX = 0)\) as the corresponding comonad.)

3 Stateful runners = comodels

We are prepared to prove the theorem that this paper revolves around. We prove that a stateful runner is the same as a comodel.

**Proposition 3.1** Given a Lawvere theory \((L, L)\), let \((T, \eta, \mu)\) be the monad corresponding to the theory. Given a set \(C\), let \((T^{C}, \eta^{C}, \mu^{C})\) be the state monad for \(C\). There is a bijection between \(\text{comonad morphisms from } (T, \eta, \mu) \text{ to } (T^{C}, \eta^{C}, \mu^{C})\) and \(\text{comodels on } C\) (i.e., coalgebras of the comonad \((D, \varepsilon, \delta)\) corresponding to \((L, L)\)).

**Proof.** Let the Lawvere theory \((L, L)\) be given by operations \(OP_{j} : I_{j} \to O_{j}\) and equations \(\text{LHS}_{k} = \text{RHS}_{k}\). The corresponding monad \((T, \eta, \mu)\) is then constructed as described in the previous section.

Given a comodel \((\overline{op}_{j} : C \times O_{j} \to C \times I_{j})_{j}\), the monad morphism \(\theta_{X} : \forall X. T X \to (C \times X) \Leftarrow C\) is defined by recursion on \(t : T_{0} X\) by

\[
\theta_{X} (\text{var} x) = \lambda c. (c, x) \\
\theta_{X} (\text{op}_{j} (o, f)) = \lambda c. \theta_{X} (f (\overline{op}_{j} (c, o))) (\overline{op}_{j} (c, o))
\]

For this definition to be legitimate, \(\theta_{X}\) must send \(\sim_{X}\)-related computations in \(T_{0} X\) to equal computations in \(T^{C} X\). This is proved by induction on the proof of \(t \sim_{X} t'\) from \(\text{LHS}_{k} = \text{RHS}_{k}\) and \(\text{LHS}_{k} = \text{RHS}_{k}\).

The unit preservation law of a monad morphism holds trivially. The multiplication preservation law is a “substitution lemma” that is proved by induction on \(t : T_{0} (T_{0} X)\).

In the converse direction, given a monad morphism \(\theta\), we define the comodel \((\overline{op}_{j})_{j}\) by

\[
\overline{op}_{j} (c, o) = \theta_{t_{j}} (\text{op}_{j} (o, \lambda i. \text{var} i)) c
\]

The equations \(\text{LHS}_{k} = \text{RHS}_{k}\) and \(\text{LHS}_{k} = \text{RHS}_{k}\) follow from \(\theta_{X}\) sending \(\sim_{X}\)-related computations in \(T_{0} X\) to equal computations in \(T^{C} X\).

The roundtrip from a comodel to a monad morphism and back is straightforwardly identity. The other roundtrip is identity thanks to naturality of \(\theta\).

We compare this theorem to the following well known theorem (see, e.g., [5]) about running with continuations.

**Proposition 3.2** Given a monad \((T, \eta, \mu)\) and a set \(A\). Let \((T^{A}, \eta^{A}, \mu^{A})\) be the continuation monad for \(A\). Monad morphisms \(\theta\) from \((T, \eta, \mu)\) to \((T^{A}, \eta^{A}, \mu^{A})\) are in a bijection with \((T, \eta, \mu)\)-algebra structures \(\alpha\) on \(A\).
Proof. Given an algebra structure $\alpha : TA \to A$, the monad morphism $\theta : \forall X. TX \to (A \leftarrow X) \Rightarrow A$ is defined by $\theta_X t = \lambda f. \alpha (T f t)$; the laws of a monad morphism follow from the laws of an algebra.

In the converse direction, given a monad morphism $\theta$, the corresponding algebra structure $\alpha$ is defined by $\alpha_t = \theta_A t \id_A$; the laws of an algebra follow from the laws of a monad morphism.

The roundtrip from an algebra to a monad morphism and back is straightforwardly identity. The proof of the other roundtrip being identity relies on the naturality of $\theta$. 

Notice that in the first theorem we need a Lawvere theory to relate a monad and a comonad. In the second theorem, there is no need to involve a Lawvere theory.

Another major difference is this. Given any monad, for any value set $X$, one can find a runner with continuations $(A, \theta)$ such that $\theta_X$ is mono, i.e., all information about a computation over a given value set $X$ in the given monad is retained in its counterpart in the continuation monad. Indeed, for any $X$, invoking the free algebra $(TX, \mu_X)$, we have $\theta_X t \eta_X = t$.

In the case of stateful running, it is much more difficult to achieve $\theta_X$ being mono. As we will see shortly, it is easy to construct examples where no state set $C$ is sufficient to recover computations over a given value set $X$.

Both theorems can be strengthened to isomorphisms of categories; we skip the details here.

4 Instances

4.1 Nondeterminism

Let us see the above proposition at work on the example of various finite nondeterminism monads as well as the partiality monad.

Finite nondeterminism

First we consider theories given by the following operation and some of the following equations:

\[
\begin{align*}
1 + 1 \quad & (c) \quad 1 + (1 + 1) + 1 + (1 + 1) \\
1 \quad & (d) \quad 1 + 1 + 1 + 1 \\
1 \quad & (e) \quad 1 + 1 + 1 + 1
\end{align*}
\]

A model of each such theory is given by a set $A$ carrying the following function and satisfying the correct equations from among the following:

\[
\begin{align*}
A \times A \quad & (c) \quad (A \times A) \times A \rightarrow (A \times A) \\
A \quad & (d) \quad A \times A \\
A \quad & (e) \quad A \times A
\end{align*}
\]

We see that models of these theories are exactly what we expect: without any equations we get magmas, with (c) semigroups, with (c), (d) commutative semigroups, with (c), (d), (e) semilattices, with (d) alone commutative magmas.
The corresponding monads are those of binary leaf trees (free magmas, $TX \cong \mu Z. X + Z \times Z$), nonempty lists (free semigroups), nonempty finite multisets (free commutative semigroups), nonempty finite sets (free semilattices), binary unordered leaf trees (free commutative magmas). All of them are quotients of the first monad. They all model nondeterminism to some level of granularity; which monad to use depends on what exactly one wants to track as the nondeterminism effect (and in fact on whether one wants to be able to resolve nondeterminism, as we will see shortly). The theory view of these monads tells us that the single operation $\text{ch}$ is complete for programming finitely nondeterministic functions.

A comodel (a runner for nondeterministic computations) is a set $C$ with the following function satisfying the intended equations from among the following:

$$C + C \quad \overset{(c)}{\longrightarrow} \quad C + (C + C) \quad \overset{\pi + \pi}{\longrightarrow} \quad C + C \quad \overset{\pi}{\longrightarrow} \quad C$$

Reformulating minimally, a comodel is a set $C$ with a function $\overline{\text{ch}} = (\overline{\text{chn}}, \overline{\text{chs}}) : C \rightarrow C \times 2$ satisfying the appropriate equations. For the theory without equations, it is nothing but a resolver (scheduler) for the finest notion of nondeterminism that remembers the order that binary choices are made, the order of options in binary choices etc. It is a machine that can make a binary choice on request and go to a new state. Given a nondeterministic computation, which is binary leaf-labelled tree in this case, it can thus choose a path from the root to some leaf.

Equation (e) forces that $\overline{\text{chn}} x = x$. Equation (c) requires $\overline{\text{chs}} (\overline{\text{chn}} x) = \overline{\text{chs}} x$ and $\overline{\text{chn}} (\overline{\text{chn}} x) = \overline{\text{chn}} x$, which are trivially fulfilled when $\overline{\text{chn}}$ is identity. So a resolver for the version of nondeterminism where order of binary choices is considered irrelevant and a binary choice between two equal options is considered the same as no choice at all (so computations are nonempty square-free lists of values, i.e., lists with no sublist occurring twice in a row) amounts to a set $C$ with an unconstrained function $\overline{\text{chs}} : C \rightarrow 2$, This is a machine that always makes its choice without changing its state, so, given a leaf tree (identified with other leaf trees that flatten into the same nonempty list), it walks down from the root by always turning to the left or always turning to the right, eventually reaching the leftmost or rightmost leaf (the first or last position of the list). Note that the other leaves (inner positions of the list) are unreachable for a runner—they are not addressable “crisply” enough.  

In a comodel for any theory containing equation (d), it must be that $\not\overline{\text{chs}} x = \text{chs} x$, which can only hold when $C \cong \mathbb{0}$. This says that, as soon as the order of the options in a choice is considered immaterial, resolving nondeterminism is impossible, apart from the uninteresting degenerate case.

The comonad for the equationless theory is that of streams of states and bits, $DX \cong \nu Z. X \times (2 \times Z)$. The comonads for more specific theories are subcomonads. In particular, the comonad for the theory with (c) alone has $DX \cong X \times (2 \times X)$.

---

[5] These two elements of $T\overline{3}$ are equal: $t_0 = \text{ch} (\text{ch} (\text{var} 0, \text{var} 1), \text{var} 2)$ and $t_1 = \text{ch} (\text{var} 0, \text{ch} (\text{var} 1, \text{var} 2))$. But $t_0 = \mu t_0$ and $t_1 = \mu t_1$ for $t_0 = \text{ch} (\text{var} (\text{ch} (\text{var} 0, \text{var} 1)), \text{var} (\text{var} 2))$ and $t_1 = \text{ch} (\text{var} (\text{var} 0), \text{var} (\text{ch} (\text{var} 1, \text{var} 2)))$ in $T(T\overline{3})$. Both $t_0$ and $t_1$ are of the form $\text{ch} (\text{var} (\ldots), \text{var} (\ldots))$. A runner that is able to extract from $t_0$ the value 1, must process $t_0$ by first going to the left, but then it must do the same to $t_1$ in which case it cannot extract the value 1 from $t_1$, which was equal to $t_0$. 238
the theory with both (c) and (e) has $DX \cong X \times 2$. The comonads for theories containing (d) have $DX \cong 0$.

**Finite nondeterminism with failure**

Let us consider extending the theories considered with the following operation and, possibly, the following equations:

\[
\begin{array}{c}
0 \xrightarrow{\text{die}} 1 \\
1 + 1 \xrightarrow{\text{ch}} 1
\end{array}
\]

\[
\begin{array}{c}
0 + 1 \xrightarrow{\text{die} + 1} 1 + 1 \\
1 + 1 \xrightarrow{\text{ch} + 1} 1
\end{array}
\]

A model is now a set $A$ supporting the following function satisfying the intended equations from among following:

\[
\begin{array}{c}
1 \xrightarrow{\text{die}} A \\
A \times A \xleftarrow{\text{ch}} A
\end{array}
\]

Models are pointed magmas, monoids, commutative semilattices with a bottom, commutative pointed magmas. The monads are the monad of nullary-binary leaf trees (free pointed magmas, $TX \cong \mu Z. X + 1 + Z \times Z$) and its different quotients—the monads of lists (free monoids), finite multisets (free commutative monoids), finite sets (free commutative semilattices with bottom), nullary-binary unordered leaf trees (free pointed commutative magmas). They all model nondeterminism with the “no-option” option.

A comodel is a set $C$ with the following function satisfying the intended ones of the following equations:

\[
\begin{array}{c}
0 \xrightarrow{\text{ch}} C \\
C + C \xrightarrow{\text{ch}} C
\end{array}
\]

\[
\begin{array}{c}
C + C \xleftarrow{\text{ch}} C
\end{array}
\]

It is immediate that there are no interesting comodels: the carrier of a comodel must be empty even in the case of no equations. The comonads are all constant 0 ($DX \cong 0$). It is impossible (except for the uninteresting degenerate case of an impossible initial state) to resolve a nondeterministic computation that may fail.

Observe that the same happens with any theory with one or more nullary operations ($\text{op} : 0 \rightarrow O$ where $O \neq 0$) or, in terms of monads, with any monad such that $T0 \neq 0$.

**Partiality**

Finally we could also skip $\text{ch}$ and consider the theory with just $\text{die}$ and no equations.

Models of this theory are pointed sets. The monad is the maybe monad (of sets with an added point, free pointed sets, $TX \cong 1 + X$), commonly used for modelling partiality. We learn the unsurprising fact that $\text{die}$ is the sole operation needed for programming partial functions. We know already that there are no interesting comodels.
4.2 Interactive input/output

We move on to consider examples of other types. For the start, take two sets $I$, $O$, and consider the very simple theory with the following two operations and no equations:

\[
\begin{array}{ccl}
I & \xrightarrow{\text{get}} & 1 \\
1 & \xrightarrow{\text{put}} & O \\
\end{array}
\]

A model is a set $A$ endowed with functions

\[
\begin{array}{ccl}
A & \leftarrow I & A \\
A & \leftarrow O & A \\
\end{array}
\]

The monad is the free monad defined by $TX = \mu Z.X + (Z \leftarrow I) + O \times Z$. We recognize in it the monad for interactive input/output with $I$ and $O$ as the input and output alphabets.

A comodel is a set $C$ with two functions

\[
\begin{array}{ccl}
C \times I & \xrightarrow{\text{get}} & C \\
C & \leftarrow C \times O & C \\
\end{array}
\]

It is a runner for interactive input/output, a machine that can provide input and consume output, changing its state. The comonad is the cofree comonad defined by $DX = \nu Z.X \times (Z \times I) \times (O \Rightarrow Z)$.

The case of interactive input only is covered by the special case $O = 0$ when we could just as well drop the operation put as forced and void of information.

Allowing interactive output only corresponds to dropping the get operation. This is different from the case $I = 0$ (input from an empty alphabet, leads to partiality), as well as from the case $I = 1$ (input from a singleton alphabet). But via $P = O^*$ (the free monoid on $O$) it is an instance of writing considered in the next section.

4.3 Stateful computation

We proceed to stateful computation. We first look at reading only and writing only, to then continue with reading and overwriting (modelled by state monads). In Appendix 6 we also look at reading and general updating (modelled by what we call update monads).

Reading

We begin with reading. Take a set $S$ (of states). We look at this theory:

\[
\begin{array}{ccl}
S & \xrightarrow{\text{get}} & S \\
S \times S & \xrightarrow{\Delta} & S \\
1 \times S & \xrightarrow{\text{put}} & S \\
1 & \xrightarrow{\text{put}} & 1 \\
\end{array}
\]
A model is a set $A$ with

$$
\begin{array}{c}
A \Leftarrow S \\
A \Leftarrow A \Leftarrow S \\
A \Leftarrow A \Leftarrow S \\
A \Leftarrow S \Leftarrow A \\
A \Leftarrow S \Leftarrow A \\
A \Leftarrow S \Leftarrow A \\
A \Leftarrow S \Leftarrow A \\
A \Leftarrow S \Leftarrow A \\
A \Leftarrow S \Leftarrow A
\end{array}
$$

By the general construction, the monad for this theory is given by $TX = T_0 X/\sim_X$ with $T_0 X$ and $\sim_X$ defined inductively by

$$
\begin{array}{c}
x : X \\
\var x : T_0 X \\
\ll p f : T_0 X
\end{array}
\quad
\begin{array}{c}
f : T_0 X \Leftarrow S \\
\ll p f : T_0 X
\end{array}
$$

(so that $T_0 X \cong \mu Z. X + (Z \Leftarrow S)$) and

$$
\begin{array}{c}
∀ s. f.s \sim_X f'.s \\
\ll p f \sim_X \ll p f' \\
c \sim_X \ll p (\lambda s. c) \\
\ll p (\lambda s'. \ll p (\lambda s. f.s')) \sim_X \ll p (\lambda s. f.s)
\end{array}
$$

It is easy to verify that every element of $TX$ can be presented in the normal form $\ll p (\lambda s. \var (f.s))$ for a unique $f : X \Leftarrow S$. It follows that the monad can alternatively be defined without quotienting by $TX = X \Leftarrow S$ and $\eta x = \lambda s. x$, $\mu f = \lambda s. f.s$. This is the reader monad for $S$ as the state set.

A comodel is a set $C$ with

$$
\begin{array}{c}
C \times S \\
C \Leftarrow S \\
(C \times S) \times S
\end{array}
\quad
\begin{array}{c}
C \times S \\
C \Leftarrow S \\
(C \times S) \times S
\end{array}
$$

or equivalently with

$$
\begin{array}{c}
C \ll p n \\
S \ll p s \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n
\end{array}
\quad
\begin{array}{c}
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n \\
C \ll p n
\end{array}
$$

In the latter description, the 1st equation explicitly asks that $\ll p n = \text{id}_C$, making $\ll p m$ redundant, and the 2nd and 3rd follow, so we are left with a function $\ll p s : C \to S$ and no equations. A runner for reading amounts thus to a machine that is happy to serve a lookup request with an external state (drawn from set $S$) and continue then in the same internal state (from set $C$)—so next time it will provide the same external state again.

It follows that the comonad can be defined by $DX = X \times S$ and $\varepsilon (x,s) = x$, $\delta (x,s) = ((x,s),s)$. This is the cofree comonad on the constant $S$ functor.

**Writing**

For writing, we take a monoid $(P, \circ, \oplus)$ (of updates) and consider the following theory:

$$
\begin{array}{c}
1 \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d
\end{array}
\quad
\begin{array}{c}
1 \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d
\end{array}
\quad
\begin{array}{c}
1 \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d
\end{array}
\quad
\begin{array}{c}
1 \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d \\
P \ll p d
\end{array}
$$
A model of this theory is a set $A$ with

$A \xleftarrow{\text{upd}} A \xrightarrow{\text{upd}} A \xrightarrow{\text{upd}} A \xrightarrow{\text{upd}} A$

or, alternatively, in uncurried form,

$P \times A \xrightarrow{\xleftarrow{\text{upd}}} P \times A \xrightarrow{\text{upd}} P \times A \xrightarrow{\text{upd}} A$

which is exactly what it means to be a left action of $(P, \circ, \oplus)$.

The general construction tells us that the corresponding monad is given by

$T X = T_0 X / \sim_X$ where $T_0 X$ and $\sim_X$ are defined inductively by

$x : X \xrightarrow{\text{var}} x : T_0 X \xrightarrow{\text{upd}} (p : P) \xrightarrow{c : T_0 X} (p, c) : T_0 X$

(so that $T_0 X = \mu Z. X + P \times Z$) and

$\text{upd} (p, c) \sim_X \text{upd} (p', c') \quad \text{upd} (p, \text{upd} (p', c)) \sim_X \text{upd} (p \oplus p', c)$

We can witness that every element of $T X$ can be cast in the form $\text{upd} (p, \text{var} x)$ for a unique pair $(p, x) : P \times X$. As a consequence, the monad is alternatively definable by $T X = P \times X$ and $\eta x = (\circ, x)$, $\mu (p, (p', x)) = (p \oplus p', x)$. This is the familiar writer monad for $(P, \circ, \oplus)$ as the monoid of updates.

A comodel for the theory of writing (a runner for writing computations) is a set $C$ with

$C \xleftarrow{\text{upd}} C \times P \xrightarrow{\text{c} \circ \text{p}} C \times P \xrightarrow{\text{c} \oplus \text{p}} C \times P \xrightarrow{\text{c} \oplus \text{p}} C \times P$

i.e., a right action of the monoid. We think of it as a machine that listens to updates and changes its state.

The comonad is constructed by taking $D X = D_0 X \mid \text{ok}_X$ where $D_0 X$ and $\text{ok}_X$ are defined coinductively by

$c : D_0 X \xrightarrow{\text{var}} c : X \xrightarrow{\text{upd}} c : D_0 X \xrightarrow{\text{p} : P}$

(so that $D_0 X = \nu Z. X \times (P \Rightarrow Z)$) and

$\text{ok}_X c \xrightarrow{\text{ok}_X (\text{upd} (c, p))} \text{ok}_X c \xrightarrow{\text{ok}_X (\text{upd} (c, o))} \text{ok}_X c \xrightarrow{\text{upd} (\text{c, p})} \text{ok}_X c \xrightarrow{\text{upd} (\text{c, p} \oplus p')}$

Here it is the case that everything that can be learned at all about an element $[]$ of $D X$ (i.e., an ok element of $D_0 X$) is summarized in the function $\lambda p. \text{var} (\text{upd} ([], p)) : P \Rightarrow X$. It is a universal way of observing elements of $D X$ in the sense that, for
any set \( Y \), any function \( f : DX \to Y \) is expressible as \( \lambda c. g (\lambda p. \text{var} (\text{upd} (c, p))) \) for a unique \( g : (P \Rightarrow X) \to Y \). Therefore, the comonad is more succinctly (without carving a subset) defined by \( DX = \{ P \Rightarrow X, \varepsilon v = v \circ \delta v = \lambda p. \lambda p', v (p \oplus p') \} \).

**Reading and overwriting**

We can now proceed reading and overwriting a state.

Given a set \( S \) (of states), the theory of reading and overwriting is given by two operations \( \text{lkp} \) and \( \text{upd} \)

\[
\begin{array}{c}
\downarrow \text{lkp} & \downarrow \text{upd} \\
1 & S \\
S & S
\end{array}
\]

A model is a set \( A \) with functions \( \text{lkp} \) and \( \text{upd} \) such that

\[
\begin{array}{c}
\downarrow \text{lkp} & \downarrow \text{upd} \\
A & A \ll S \\
A & A \ll S
\end{array}
\]

or, alternatively, in uncurried form,

\[
\begin{array}{c}
\downarrow \text{lkp} & \downarrow \text{upd} & \downarrow \text{lkp} \\
A & A \ll S & A \ll S \\
A & A \ll S & A \ll S
\end{array}
\]

The corresponding monad is \( TX = T_0 X / \sim_X \) where \( T_0 X \) and \( \sim_X \) are defined inductively by

\[
x: X \quad \var x: T_0 X \\
f : T_0 X \ll S \\
\text{lkp} f : T_0 X \\
s : S \\
c : T_0 X \\
\text{upd} (s, c) : T_0 X
\]

(so that \( T_0 X = \mu Z. X + (Z \ll S) + S \times Z \)) and

\[
\begin{array}{c}
\forall s. f s \sim_X f' s \\
\text{lkp} f \sim_X \text{lkp} f' \\
c \sim_X c' \\
\text{lkp} \sim_X \text{lkp} f \\
\text{upd} (s, \text{lkp} f) \sim_X \text{upd} (s, f s)
\end{array}
\]

As every element of \( TX \) can be uniquely presented in the normal form \( \text{lkp} (\lambda s. \text{upd} (g s, \text{var} (h s))) \) for some \( \langle g, h \rangle : (S \times X) \ll S \), we have that \( TX \cong (S \times X) \ll S \) whereby \( \eta x = \lambda s. (s, x) \), \( \mu f = \lambda s. \text{let} (s', g) \leftarrow f s \text{ in } g s' \) — the state monad for \( S \).
A comodel is a set $C$ together with functions $\text{lkp}$ and $\text{upd}$ such that

$$
\begin{align*}
\begin{array}{c}
C \times S & \xleftarrow{\text{lkp}} C & C \times S & \xrightarrow{\text{upd}} C \\
C & \xleftarrow{\text{lkp}} C & C \times S & \xrightarrow{\text{upd}} C \\
\end{array}
\end{align*}
$$

In that context, the weaker (less well behaved) structures do not deserve the name 'lens'. More precisely, the bidirectional transformations term would be 'very well-behaved lens', but from our perspective the weaker (less well behaved) structures do not deserve the name 'lens'.

$$
\begin{align*}
\begin{array}{c}
C \times S & \xleftarrow{\text{lkp}} C & C \times S \\
\text{upd} \times S & \xrightarrow{\text{upd}} (C \times S) \times S & C \times (S \times S) \\
\end{array}
\end{align*}
$$

Here the 1st and 3rd equation together give that $\text{lkpn} = \text{id}_C$ making $\text{lkpn}$ redundant. The 1st equation then simplifies to $\text{upd} \circ (\text{id}_C, \text{lkps}) = \text{id}_C$ and the 3rd equation becomes tautological. We see that a runner for a stateful computation is a machine responding to lookups (without changing its state) and listening to overwrites. This structure is known in bidirectional transformations [4] as a lens between $C$ and $S$.

The comonad is $D X = D_0 X \mid ok_X$ where $D_0 X$ and $ok_X$ are defined coinductively by

$$
\begin{align*}
c : D_0 X & \xrightarrow{\text{var}} X \\
\text{lc} : D_0 X & \xrightarrow{\text{lkps}} C \\
\text{upd} : C \times S & \xrightarrow{\text{upd}} (C \times S) \times S \\
\end{align*}
$$

(s0 that $D_0 X = \nu Z. X \times S \times (S \Rightarrow Z)$) and

$$
\begin{align*}
ok_X c \xrightarrow{\text{lkps} (c, s)} & ok_X c \xrightarrow{\text{upd} (c, \text{lkps} c)} ok_X c \\
\text{upd} (c, ok_X c) & = \text{upd} (c, ok_X c) \\
\text{var} (\text{lkps} (c, s)) & = s
\end{align*}
$$

All that can be known about an element $[\ ]$ of $D X$ can be summarized in the universal observation ($\text{lkps} [\ ]$, $\lambda s. \text{var} (\text{lkps} ([\ ]), s) : S \times (S \Rightarrow X)$. Hence the comonad can also be defined by $D X = S \times (S \Rightarrow X)$ and $\varepsilon (s, v) = vs$ and $\delta (s, v) = (s, \lambda s'. (s', v))$. This comonad is known as the costate (or array) comonad $[13,9]$.  

4.4 Continuations

Continuation monads have no rank, so our analysis in terms of Lawvere theories does not apply. However it is easy to check directly that they cannot have non-trivial runners (i.e., runners with a non-empty carrier).

---

6 The computation has $S$ as its state set; the runner’s state set is $C$. The final comodel has $C = S$, $\text{lkps} = \text{id}_S$ and $\text{upd} = \text{snd}$, but it is not the only comodel. It is the case however that, for any comodel, $C \cong S \times C'$ for some set $C'$. The $C'$ projection of the runner’s state cannot be looked up, cannot be overwritten by computations.

7 More precisely, the bidirectional transformations term would be 'very well-behaved lens', but from our perspective the weaker (less well behaved) structures do not deserve the name 'lens'.

8 In that context, $S$ is called the view state set and $C$ the source state set.
Indeed, fix a nonempty set $R$. A runner structure on a set $C$ would be a monad morphism between the continuation monad for $R$ and the state monad for $C$, so a family of maps $\theta : \forall X. (R \sqsubseteq X) \Rightarrow R \Rightarrow (C \times X) \sqsubseteq C$. Consider $\theta_0$. The set $(0 \Rightarrow R) \Rightarrow R$ is obviously inhabited (for any element $r$ of $R$, it contains $\lambda f. r$), but for the set $(C \times 0) \sqsubseteq C$ to be inhabited we need a function $C \rightarrow 0$, which can only exist if $C = 0$.

5 Running vs. handling

Runners bear some similarity to Plotkin and Pretnar’s handlers [12], but they are not the same. Let us spell out the exact relationship.

Broadly speaking, both are about specifying ways to extract a value from a computation.

Handling is based on the fact that, for any set $A$, $(TA, \mu_A : T(TA) \rightarrow TA)$ is the free $(T, \eta, \mu)$-algebra on $A$, with $\eta_A : A \rightarrow TA$ as the associated injection.

Spelled out, this means that, given two sets $A, B$, a map $f : A \rightarrow B$ and a $(T, \eta, \mu)$-algebra structure $g : TB \rightarrow B$, we have a unique map $h : TA \rightarrow B$ making the diagrams commute.

Running, at the same time, is based on the observation that any coalgebra $(C, g : C \rightarrow DC)$ of a suitable comonad $(D, \varepsilon, \delta)$ induces a unique monad morphism between the given monad $(T, \eta, \mu)$ and the state monad $(TC, \eta_C, \mu_C)$. This is to say that we have a unique natural transformation $\theta$ satisfying

$$
\begin{array}{c}
X \xrightarrow{\eta_X} TX \xleftarrow{\mu_X} T(TX) \\
\downarrow \quad \downarrow \quad \downarrow \\
TCX \xrightarrow{\eta_C} TC(TCX) \xleftarrow{\mu_C} T(TCX) \\
\downarrow \quad \downarrow \quad \downarrow \\
T CX \xrightarrow{\theta_{TCX}} T(TCX)
\end{array}
$$

The important differences are these. First, handlers are monomorphic and any codomain is possible: a handler defines map from $TA$ to $B$ for some fixed sets $A, B$. Runners are polymorphic, but the codomain is restricted to a specific form: a runner gives a family of functions from $TX$ to $TCX$ for a fixed set $C$. Second, the data inducing handlers and runners are different.

Runners are an instance of handlers, but not in a very useful way: as a circular unique existence property rather than a direct one. Indeed, in the diagram above we have rendered the pentagon stating the condition that $\theta_X$ sends $\mu_X$ to $\mu_C$ in a layout suggesting that the composite map $\xi_X = \mu_C \circ \theta_{TCX}$ might be a $(T, \eta, \mu)$-algebra structure on $TCX$, and it is easily verified to be so. But $\xi_X$ is defined in terms of $\theta_{TCX}$, i.e., another component of $\theta$. So we cannot say that the algebra morphism $\theta_X$ is induced by an independently given algebra structure $\xi_X$.

Modulo this reservation, due to their polymorphic nature, runners are actually more than just handlers, they are uniform handlers. A general definition of a uniform handler proceeds from natural transformations as monad-algebra structures. Say that a set functor $F$ with a natural transformation $\sigma : T \cdot F \rightarrow F$ is a
(\(T, \eta, \mu\))-algebra, if it also meets the conditions

\[
\begin{array}{c}
\begin{array}{c}
\xymatrix{FX \ar[r]^-{\eta_X} \ar[d]_-{\sigma X} & T(FX) \ar[d]^-{\sigma X} \\
T(FX) \ar[r]_-{\epsilon X} & FX}
\end{array}
\end{array}
\]

It is now easy to check that the free \((T, \eta, \mu)\)-algebra on \(F\) is \((T \cdot F, \mu \cdot F : T \cdot T \cdot F \to T \cdot F)\), with \(\eta \cdot F : F \to T \cdot F\) as the associating injection. Accordingly, for any functor \(G\), a natural transformation \(\phi : F \to G\) and a \((T, \eta, \mu)\)-algebra structure \(\psi : T \cdot G \to G\), we have a unique natural transformation \(\chi : T \cdot F \to G\) such that

\[
\begin{array}{c}
\begin{array}{c}
\xymatrix{FX \ar[r]^-{\eta_X} \ar[d]_-{\sigma X} & T(FX) \ar[r]^-{\mu_X} \ar[d]^-{\psi_X} & T(T(FX)) \ar[d]^-{\phi_X} \\
T(FX) \ar[r]_-{\epsilon X} & FX \ar[r]_-{\phi_X} & G(X)}
\end{array}
\end{array}
\]

A runner \(\theta\) is now a uniform handler for \(F = \text{Id}\), \(G = T^C\), \(\phi = \eta^C\), \(\psi = \mu^C \circ (\theta \cdot T^C)\).

All of these considerations apply of course to any monad morphism from \((T, \eta, \mu)\), e.g., the morphism to the continuation monad on the the carrier of a \((T, \eta, \mu)\)-algebra discussed in Section 3—a monad morphism is always a monad algebra morphism.

6 Conclusion and future work

We showed when a computation in a given monad can be mapped into a stateful computation: it is when the state set carries a comodel of the corresponding Lawvere theory (a coalgebra of a suitable comonad). We find this to be a nice small new application of comodels (cf. the discussion by Behrisch et al. [3] on whether comodels are the “correct” dual of models). We also believe that it gives some new insight into the mechanics of different monadic/algebraic notions of effects, especially in regard to the impact the degree of abstractness (i.e., how much detail of effects we want to observe). For nondeterminism, for example, we saw that some approaches that make perfect sense for denotational semantics are not operational at all and there are simple mathematical reasons why this has to be so. We find it curious that state monads turn out to have a special role in connecting models and and comodels. Something similar appears in Møgelberg and Staton’s work [7] on every monad being a linear state monad (under a certain viewpoint).

In future work, we intend to study sufficient conditions for a monad morphism to a state monad to be mono (so the stateful computation can capture all information in a given computation). We also plan to consider other target monads, in particular, combinations of state monads with other monads such as exception monads, and to relate this work and Plotkin and Power’s [11] tensor of a comodel and a model.

Acknowledgments

I am most grateful to Gordon Plotkin, Alex Simpson, Tom Schrijvers and Nicolas Wu for the comments they made in response to my talks on this material as well as to the MFPS reviewers for their feedback. This research was supported by the ERDF funded projects EXCS and Coinduction, the Estonian Ministry of Education.
and Research institutional research grant no. IUT33-13 and the Estonian Science Foundation grant no. 9475.

References


More on stateful computation

Reading and general writing

Let us also consider reading and general writing (as opposed to just overwriting) in combination.

Given a set \( S \) (of states), a monoid \((P, \circ, \oplus)\) (of updates) and a right action \( \downarrow: S \times P \rightarrow S \) (describing application of updates to states), we are interested in the theory given by the following operations and equations:

\[
\begin{align*}
\forall s, f s \sim_X f' s & \quad \vdash_{X} c \sim_{X} c' \quad \text{upd} (p, c) \sim_{X} \text{upd} (p', c) \quad c \sim_{X} \text{lkp} (\lambda s. c) \quad \text{lkp} (\lambda s'. \text{lkp} (\lambda s. f s s')) \sim_{X} \text{lkp} (\lambda s. f s s) \\
\text{lkp} f \sim_{X} \text{lkp} f' & \quad \text{lkp} \text{upd} (p, c) \sim_{X} \text{lkp} \text{upd} (p', c) \quad \text{lkp} (p, \text{lkp} f) \sim_{X} \text{lkp} (p, \text{lkp} f (s \downarrow p))
\end{align*}
\]

Since the equations allow us to present any element of \( TX \) uniquely in the normal form \( \text{lkp} (\lambda s. \text{upd} (g s, \text{var} (h s))) \) for some \( \langle g, h \rangle : (P \times X) \leftarrow S \), we get that

\footnote{This is not the minimal presentation, but the simplest one. The minimal one has the same operations, but three (more involved) equations.}
We have called this monad the update monad for \( S, (P \circ, \oplus), \downarrow \) [2]. Update monads are exactly the compatible compositions of reader and writer monads—distributive laws between them are in bijections with right actions.

A comodel is a set \( C \) with functions \( \text{lkp} \) and \( \text{upd} \) such that

\[
\begin{align*}
C \times S & \xrightarrow{\text{upd}} C \\
C & \xrightarrow{\text{lkp}} C \times 1 \\
C \times P & \xrightarrow{\text{lkp} \times P} C \times (C \times S) \\
C \times (C \times S) & \xrightarrow{\Delta} C \times (P \times P) \\
C \times (P \times P) & \xrightarrow{\text{upd} \times P} C \times S \\
(C \times S) \times P & \xrightarrow{(\text{upd}, \delta)} (C \times S) 	imes P \\
(C \times S) \times P & \xrightarrow{\text{upd} \times P} C \times S \\
(C \times S) \times P & \xrightarrow{\text{upd} \times P} C \times S \\
S & \xrightarrow{\text{upd} \times P} S \times P \\
S & \xrightarrow{\text{upd} \times P} S \times P
\end{align*}
\]

Splitting \( \text{lkp} = (\text{lkp}_n, \text{lkp}_p) \), we see that the 1st equation just says \( \text{lkp}_n = \text{id}_C \), making \( \text{lkp}_p \) redundant. This makes the 2nd equation tautological and simplifies the 5th to

\[
\begin{align*}
S & \xrightarrow{\text{upd} \times P} S \times P
\end{align*}
\]

We have previously christened these structures update lenses [1], they are a refinement of state-based lenses. An update lens is a machine that responds to lookups (without changing its state) and listens to updates.

The corresponding comonad is \( DX = D_0 X \mid ok_X \) where \( D_0 X \) and \( ok_X \) are defined coinductively by

\[
\begin{align*}
c : D_0 X & \quad e : D_0 X \\
\text{var} e : X & \quad e : D_0 X \quad p : P \\
\text{lkp}_e : S & \quad \text{upd} (e, c) : D_0 X
\end{align*}
\]

(so that \( D_0 X = \nu Z. X \times S \times (P \Rightarrow Z) \)) and

\[
\begin{align*}
\text{ok}_X (\text{var} (c, e)) & = \text{update} (e, c) \\
\text{ok}_X (\text{update} (e, c, p')) & = \text{update} (e, c, p' + p) \\
\text{ok}_X (\text{lkp}_p) & = \text{lkp}_p
\end{align*}
\]

All information available about an element \([ \ ]\) of \( DX \) is captured in the universal observation \( (\text{lkp} [ \ ], \lambda p. \text{var} (\text{update} (\text{lkp} [ \ ], p))) : S \times (P \Rightarrow X) \), which tells us that \( DX \cong S \times (P \Rightarrow X) \) (the coupdate comonad) whereby

\[
\begin{align*}
\varepsilon_X (s, v) & = v \\
\delta_X (s, v) & = (s, \lambda p. (s \downarrow p, v (p \oplus p')))
\end{align*}
\]
Unguarded Recursion on Coinductive Resumptions

Sergey Goncharov  Christoph Rauch  Lutz Schröder

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

We study a model of side-effecting processes obtained by adjoining free operations to a monad modelling base effects by means of a cofree coalgebra construction; one thus arrives at what one may think of as types of non-wellfounded side-effecting trees, generalizing the infinite resumption monad. Types of this kind have received some attention in the recent literature; in particular, it has been shown that they admit guarded iteration. Here, we show that they also admit unguarded iteration, i.e. form complete Elgot monads, provided that the underlying base effect supports unguarded iteration.

Keywords: Recursion, coalgebra, coinduction, complete Elgot monad, resumptions.

1 Introduction

Following seminal work by Moggi [17], monads are widely used to represent computational effects in program semantics, and in fact in actual programming languages [28]. Their main attraction lies in the fact that they provide an interface to a generic notion of side-effect at the right level of abstraction: they subsume a wide variety of side-effects such as state, non-determinism, random, and I/O, and at the same time retain enough internal structure to support a substantial amount of generic meta-theory and programming, the latter witnessed, for example, by the monad class implemented in the Haskell basic libraries [19].

In the current work, we study a particular construction on monads motivated partly by the goal of modelling generic side-effects in the semantics of reactive processes. Specifically, given a base monad $T$ and objects (types) $a$, $b$, we have, assuming enough structure on $T$ and the base category, a family of final coalgebras

$$T^b_a X = \nu \gamma. T(X + a \times \gamma^b)$$

for each object $X$. These final coalgebras can be seen as arising in two ways: on the one hand, one may start from reactive processes sending messages of type $a$ and...
receiving messages of type $b$ (possibly terminating with results of type $X$), modelled as non-wellfounded $a$-labelled $b$-branching trees (with leaves labelled in $X$), i.e. inhabitants of $\nu \gamma.(X + a \times \gamma^b)$, and then add generic side-effects encapsulated by $T$ to the model (e.g. non-determinism or access to a global shared memory). On the other hand, one may see $a$ and $b$ as the types of an uninterpreted side-effect $f : a \to b$ added to the base monad $T$, e.g. an I/O-operation (in fact, the interactive input and output monads originally considered as examples by Moggi [17] can be seen as generated by uninterpreted effects of this kind); if one wishes to model non-terminating programs that use $f$ as well as side-effects from $T$, one obtains infinite trees of exactly the kind given by $T^b_a X$. The construction of $T^b_a X$ from $T$ is an infinite version of the generalized resumption transformer introduced by Cienciarelli and Moggi [9]. It has been termed the coalgebraic generalized resumption transformer by Piróg and Gibbons [20] (later generalized further [21]), who show that on the Kleisli category of $T$, $T^b_a$ is the free completely iterative monad generated by $T(a \times _b b)$. The result that $T^b_a$ is a completely iterative monad brings us to the contribution of the current paper. Recall that complete iterativity of $T^b_a$ means that for every morphism $e : X \to T^b_a(Y + X)$, read as an equation defining the inhabitants of $X$, thought of as variables, as terms over the defined variables (from $X$) and parameters from $Y$, has a unique solution $e^\dagger : X \to T^b_a Y$ in the evident sense, provided that $e$ is guarded. The latter concept is defined in terms of additional structure of $T^b_a$ as an idealized monad, which essentially allows distinguishing terms beginning with an operation from mere variables. Guardedness of $e$ then means essentially that recursive calls can happen only under a free operation. Similar results on guarded recursion abound in the literature; for example, the fact that $T^b_a$ admits guarded recursive definitions can also be deduced from more general results on guarded recursion by Uustalu on parametrized monads [27].

The central result of the current paper is to remove the guardedness restriction in the above setup. That is, we show that a solution $e^\dagger : X \to T^b_a Y$ exists for every morphism $e : X \to T^b_a(X + Y)$. Of course, the solution is then no longer unique (for example, we admit definitions of the form $x = x$); moreover, we clearly need to make additional assumptions about $T$. Our result states, more precisely, that $T^b_a$ allows for a principled choice of solutions $e^\dagger$ satisfying standard equational laws for recursion [25], thus making $T^b_a$ into a complete Elgot monad [3] \footnote{We vary the original definition of Elgot monads, which requires the object $X$ of variables to be a finitely presentable object in an lfp category, by admitting unrestricted objects of variables; this change is explicitly not an important part of our contribution, and presumably not central to the technical development although we have not checked details in the finitary case.}. The assumption on $T$ that we need to enable this result is that $T$ itself is an Elgot monad (e.g. partiality, nondeterminism, or combinations of these with state), i.e. we show that the class of Elgot monads is stable under the coinductive generalized resumption transformer. We show moreover that the structure of $T^b_a$ as an Elgot monad is uniquely determined as extending that of $T$. 251
The motivation for these results is, well, to free non-wellfounded recursive definitions from the standard guardedness constraint. Note for example that in [20], it was necessary to assume guards in all loop iterations when interpreting a while-language with actions originally proposed by Rutten [24] over a completely iterative monad. Contrastingly, given that $T^b_a$ is a (complete) Elgot monad, one can now just write unrestricted while loops. We elaborate this example in Section 5, and recall a standard example of unguarded recursion in process algebra in Section 6.

2 Preliminaries

According to Moggi [17], a notion of computation can be formalized as a strong monad $T$ over a Cartesian category $C$ (i.e. a category with finite products). In order to support the constructions occurring in the main object of study, here we work in a distributive category $C$, i.e. a category with finite products and coproducts (including a final and an initial object) and such that the natural transformation

$$X \times Y + X \times Z \cong X \times (Y + Z)$$

is an isomorphism [10], whose inverse we denote $\text{dist}_{X,Y,Z}$. Here we denote injections into binary coproducts by $\text{inl}: A \to A + B$, $\text{inr}: B \to A + B$. The projections from binary products are denoted $\text{fst}: A \times B \to A$, $\text{snd}: A \times B \to B$; pairing is denoted by $\langle \_ , \_ \rangle$, and copairing of $f: A \to C$, $g: B \to C$ by $[f,g]: A + B \to C$. Unique morphisms $A \to 1$ into the terminal object are written $!_A$, or just $!$. We write $|C|$ for the class of objects of $C$. Distributivity essentially allows for using context variables in case expressions, i.e. in copairing.

We shall also require existence of certain exponentials, i.e. objects $X^a$ adjoint to Cartesian products $a \times X$, which means: for any $X$ and $Y$, there is an isomorphism

$$\text{curry}_{X,Y}: \text{Hom}_C(X \times a, Y) \cong \text{Hom}_C(X, Y^a)$$

natural in $X$ and $Y$. We write $\text{uncurry}_{X,Y}$ for the inverse map $\text{curry}_{X,Y}^{-1}$. Then evaluation morphism $\text{ev}_X: X^a \times a \to X$ (natural in $X$) is obtained as $\text{uncurry}_{X^a,X}(\text{id}_{X^a})$. We commonly omit indices at natural transformations if it improves readability unless confusion arises.

Remark 2.1 The role of exponentials in $X^a$ is to capture a notion of arity of algebraic operations generating effects, e.g. $a = 2$ would correspond to binary operations such as nondeterministic choice. A more general setup would involve categories enriched over a symmetric monoidal closed category $V$ whose objects are then treated as arities (and coarities, i.e. objects used for indexing families of operations) [13,12]. Instead of assuming existence of exponentials $X^a$ one assumes existence of tensors $a \times X$ and cotensors $X^b$ with $a, b \in |V|$. Cotensors are adjoint to tensors in the same way as exponentials are adjoint to products with a constant object. We expect that our main results extend to this setting.

Recall that a monad $T$ over $C$ can be given by a Kleisli triple $(T, \eta, \_ \_^*)$ where $T$ is an endomap of $|C|$ (in the following, we always denote Kleisli triples and their functor parts by the same letter, with the former in blackboard bold), the unit $\eta$ is
a family of morphisms $\eta_X : X \to TX$, and the Kleisli lifting $\_^*$ maps $f : X \to TY$ to $f^* : TX \to TY$, subject to the equations

$$\eta^* = \text{id} \quad f^* \circ \eta = f \quad (f^* \circ g)^* = f^* \circ g^*.$$ 

This is equivalent to the presentation in terms of an endofunctor $T$ with natural transformations unit and multiplication. A monad is strong if it is equipped with a natural transformation $\tau_{X,Y} : X \times TY \to T(X \times Y)$ called strength, subject to a number of coherence conditions (e.g. [17]). Strength enables interpreting programs over more than one variable, and allows for internalization of the Kleisli lifting, thus legitimating expressions like $\lambda x. (f(x))^* : X \to (TY \to TZ)$ for $f : X \to (Y \to TZ)$, which essentially encodes $\text{curry}(\text{uncurry}(f)^* \circ \tau)$. Strength is equivalent to the monad being enriched over $\mathcal{C}$ [14]; in particular, every monad on $\textbf{Set}$ is strong. Henceforth we shall use the term ‘monad’ to mean ‘strong monad’ unless explicitly stated otherwise.

The standard intuition for a monad $T$ is to think of $TX$ as the set of terms in some algebraic theory, with variables taken from $X$. In this view, the unit converts variables into terms, and a Kleisli lifting $f^*$ applies a substitution $f : X \to TY$ to terms over $X$. In our setting, the ‘terms’ featuring here are often infinite; nevertheless, we sometimes call them algebraic terms for distinction from the terms in our metalanguage.

The Kleisli category $\mathcal{C}_T$ of a monad $T$ has the same objects as $\mathcal{C}$, and $\mathcal{C}$-morphisms $X \to TY$ as morphisms $X \to Y$. The identity on $X$ in $\mathcal{C}_T$ is $\eta_X$; and the Kleisli composite of $f : X \to TY$ and $g : Y \to TZ$ is $g^* \circ f$. A monad $T$ has rank $\kappa$ if it preserves $\kappa$-filtered colimits. On $\textbf{Set}$ this condition intuitively means that $T$ is determined by its values on sets whose cardinality is smaller than $\kappa$.

### 3 Complete Elgot Monads

As indicated in the introduction, we will be interested in recursive definitions over a monad $T$; abstractly, these are morphisms

$$f : X \to T(Y + X)$$

thought of as associating to each variable $x : X$ a definition $f(x)$ in the shape of an algebraic term from $T(Y + X)$, which thus employs parameters from $Y$ as well as the defined variables from $X$. The latter amount to recursive calls of the definition. This notion is agnostic to what happens in the case of non-terminating recursion. For example, $T$ might identify all non-terminating sequences of recursive calls into a single value $\perp$ signifying non-termination; at the other extreme, $T$ might be a type of infinite trees that just records the tree of recursive calls explicitly.

To a recursive definition $f$ as above, we wish to associate a solution

$$f^\dagger : X \to TY,$$

which amounts to a non-recursive definition of the elements of $X$ as terms over $Y$ only. As we do not assume any form of guardedness, this solution will in general
We call the derived operator \( T \) operations, i.e. via the coinductive generalized resumption transformer.

**Definition 3.1 (Complete Elgot monads)** A complete Elgot monad is a strong monad \( T \) equipped with an operator \(_{\dagger}\), called iteration, that sends any \( f : X \to T(Y + X) \) to \( f^\dagger : X \to TY \) satisfying the following conditions:

- unfolding: \([\eta, f]^* \circ f = f^\dagger;\)
- naturality: \( g^* \circ f^\dagger = ([T \text{inl} \circ g \circ \text{inr}]^* \circ f)^\dagger \) for any \( g : Y \to TZ;\)
- dinaturality: \( ([\eta \circ \text{inl}, h]^* \circ g)^\dagger = [\eta, (\eta \circ \text{inl}, g]^* \circ h])^\dagger \circ g \) for any \( f : X \to T(Y + Z) \) and \( h : Z \to T(Y + X);\)
- codiagonal: \( T[\text{id}, \text{inr}] \circ g)^\dagger = (g)^\dagger \) for any \( g : X \to T((Y + X) + X);\)
- uniformity: \( f \circ h = T(\text{id} + h) \circ g \) implies \( f^\dagger \circ h = g^\dagger \) for any \( f : Z \to T(Y + Z) \) and \( h : Z \to X.\)

Additionally, iteration must be compatible with strength in the following sense: for any \( f : X \to T(Y + X) \), \( \tau \circ (\text{id} \times f^\dagger) = (T \text{dist} \circ \tau \circ (\text{id} \times f))^\dagger.\)

**Remark 3.2** The above definition is inspired by the axioms of parametrized uniform iterativity [25], which goes back to Bloom and Ėsik [8]. Adámek et al. [3] define Elgot monads by means of a slightly different system of axioms: the co-diagonal and dinaturality axioms are replaced with the Bekič identity. Both axiomatizations are however equivalent, which is essentially a result about iteration theories [8, Section 6.8]. Moreover, the iteration operator in [3] is defined only for \( f : X \to T(Y + X) \) with finitely presentable \( X \), under the assumption that \( \mathcal{C} \) is locally finitely presentable; hence our use of the term ‘complete Elgot monad’ instead of ‘Elgot monad’. We have the impression that this difference is not technically essential but have not checked details for the finitary variant of our results.

In the further development, examples of complete Elgot monads will arise either as so-called \( \omega \)-continuous monads (Definition 3.3) or as extensions thereof with free operations, i.e. via the coinductive generalized resumption transformer.

If \( T \) supports an iteration operator \(_{\dagger}\), then it is always possible to parametrize it with an additional argument to be carried over the recursion loop, i.e. we derive an operator \(_{\dagger}\) sending \( f : Z \times X \to T(Y + X) \) to \( f^\dagger : Z \times X \to TY \) by

\[
f^\dagger = (T(\text{snd} + \text{id}) \circ (T \text{dist} \circ (\tau_Z, Y, X) \circ (\text{fst}, f))^\dagger.\tag{1}\]

We call the derived operator \(_{\dagger}\) **strong iteration**.

As indicated above, an important class of examples of complete Elgot monads arises via a suitable order-enrichment of the Kleisli category.

**Definition 3.3 (\( \omega \)-continuous monad)** An \( \omega \)-continuous monad consists of a monad \( T \) and an enrichment of the Kleisli category \( \mathcal{C}_T \) of \( \mathcal{T} \) over the category \( \omega \text{Cppo} \) of \( \omega \)-complete partial orders with bottom and (nonstrict) continuous maps, satisfying the following conditions:

- strength is \( \omega \)-continuous: \( \tau \circ (\text{id} \times \bigcup_i f_i) = \bigcup_i (\tau \circ (\text{id} \times f_i));\)
• copairing in $C_T$ is $\omega$-continuous in both arguments: $[\bigsqcup_i f_i, \bigsqcup_i g_i] = \bigsqcup_i [f_i, g_i]$;

• bottom elements are preserved by strength and by postcomposition in $C_T$: $\tau \circ (\text{id} \times \bot) = \bot, f^* \circ \bot = \bot$.

**Example 3.4** Many of the standard computational monads on $\text{Set}$ [17] are $\omega$-continuous, including nontermination ($TX = X + 1$), nondeterminism ($TX = \mathcal{P}(X)$), and the nondeterministic state monad ($TX = \mathcal{P}(X \times S)^S$ for a set $S$ of states). On $\omega\text{Cpo}$, lifting ($TX = X_{\bot}$) and the various power domain monads are $\omega$-continuous.

**Remark 3.5** As observed by Kock [14], monad strength is equivalent to enrichment over the base category. One consequence of this fundamental fact is that if $C$ is enriched over the category $\omega\text{Cpo}$ of bottomless $\omega$-complete partial orders and $\omega$-continuous maps (i.e. $C$ is an $\mathbf{O}$-category in the sense of Wand [29] and Smyth and Plotkin [26]), with the bicartesian closed structure enriched in the obvious sense, then $C_T$ is also enriched over $\omega\text{Cpo}$, since $T$, underlying a strong monad, is an $\omega\text{Cpo}$-functor (aka locally continuous functor [26]). Then $T$ is $\omega$-continuous in the sense of Definition 3.3 iff each $\text{Hom}(X, TY)$ has a bottom element preserved by strength and postcomposition in $C_T$. This allows for incorporating numerous domain-theoretic examples by taking $C$ to be a suitable category of predomains, and $T$, in the simplest case, the lifting monad $TX = X_{\bot}$ (from which one builds more complex examples by the construction explored next).

If $T$ is an $\omega$-continuous monad, then the endomap

$$h \mapsto [\eta, h]^* \circ f$$

on the hom-set $\text{Hom}_C(A, TB)$ is continuous because copairing and Kleisli composition in $T$ are continuous, and hence has a least fixpoint by Kleene’s fixpoint theorem. We can define an iteration operator by taking $f^\dagger$ to be this fixpoint; in other words, $f^\dagger$ is defined to be the smallest solution of the unfolding equation as per Definition 3.1. The verification of the remaining identities is tedious but straightforward; in summary,

**Theorem 3.6** On every $\omega$-continuous monad, defining iteration by taking least fixpoints determines a complete Elgot monad structure.

This result is unsurprising in the light of analogous facts known for so-called $\omega$-continuous theories [8, Theorem 8.2.15, Exercise 8.2.17].

**Remark 3.7** Every complete Elgot monad $T$ can express unproductive divergence as the generic effect

$$\left( X \xrightarrow{\eta \circ \text{inr}} T(Y + X) \right)^\dagger.$$ 

This computation never produces any effects, i.e. behaves like a deadlock. If $T$ is $\omega$-continuous, then unproductive divergence coincides with the least element of $\text{Hom}(X, TY)$, for which reason we use the same symbol $\bot$ for the above morphism, but in general, there is no ordering in which unproductive divergence could be a least element.
4 The Coinductive Generalized Resumption Transformer

We proceed to recall the definition of the coinductive generalized resumption transformer [20], of which for simplicity we consider a version with only one family of free operations (rather than a whole signature or, even more generally, an arbitrary endofunctor on the base category). We then prove our main result, stability of the class of complete Elgot monads under this construction (Theorem 4.5).

Given $a, b \in |C|$ such that exponentials of the form $X^b$ exist and a monad $T\tau$, put

$$\begin{align*}
\lambda^b_{\tau} a & = a \times \lambda^b \quad \text{and} \quad T^b a \tau X = \nu \gamma T(X + \lambda^b_{\gamma a});
\end{align*}$$

i.e. $T^b a \tau X$ is the final coalgebra of $T(X + \lambda^b_{\gamma a})$, which we assume to exist. The assignment $\lambda^b_{\tau} a$ is clearly a functor, i.e. applies also to morphisms. Intuitively, $T^b a \tau X$ is a type of possibly non-terminating computation trees, with each node consisting of a computation with side-effects specified by $T\tau$ that either returns a value in $X$ or continues with one of $a$-many free operations each combining $b$-many subsequent computations. Let

$$\begin{align*}
\text{out}_a : T^b a \tau X & \rightarrow T(X + (T^b a \tau X)^b)
\end{align*}$$

be the final coalgebra structure, and let $\coit(g) : Y \rightarrow T^b a \tau X$ denote the final morphism induced by a coalgebra $g : Y \rightarrow T(X + Y^b) :

$$
\begin{array}{ccc}
Y & \xrightarrow{\coit(g)} & T^b a \tau X \\
\downarrow g & & \downarrow \text{out}_a \\
T(X + Y^b) & \xrightarrow{T(X + (\coit(g))^b)} & T(X + (T^b a \tau X)^b).
\end{array}
$$

Intuitively, $\coit(g)$ encapsulates (in $T^b a \tau X$) a computation tree that begins by executing $g$, terminates in a leaf of type $X$ if $g$ does, and otherwise (co-)recursively continues to execute $g$, forming a new tree node for each recursive call. It is easy to verify that $\text{out}_a$ is natural in $X$. By Lambek’s lemma, out is a natural isomorphism. Thus, $T\tau$ maps into $T^b a \tau$ via

$$
\begin{align*}
\text{ext} = T & \xrightarrow{T \text{inl}} T(\text{Id} + (T^b a \tau^b) \rightarrow \text{out}^{-1} \rightarrow T^b a \tau.
\end{align*}
$$

We record explicitly that $T^b a \tau$ is a strong monad:

**Lemma 4.1** Given a monad $T\tau$ and $a, b \in |C|$, $T^b a \tau$ is the functorial part of a monad $T^b a \tau$, with the monad structure characterized by the following properties.

(i) The unit $\eta^\tau : X \rightarrow T^b a \tau X$ is defined by $\text{out} \circ \eta^\tau = \eta \circ \text{inl}$ (i.e. $\eta^\tau = \text{out}^{-1} \circ \eta \circ \text{inl}$).

(ii) Given $f : X \rightarrow T^b a \tau Y$, the Kleisli lifting $f^\tau : T^b a \tau X \rightarrow T^b a \tau Y$ is the unique solution of the equation $\text{out} \circ f^\tau = [\text{out} \circ f, \eta \circ \text{inr}(f^\tau)^b] \circ \text{out}$.

(iii) Given $f : X \rightarrow T^b a \tau Y$, let $g = [f, \eta^\tau] : X + Y \rightarrow T^b a \tau Y$; then $g^\tau$ is a final morphism of coalgebras, namely $g^\tau = \coit([T(\text{id} + (T^b a \text{inr})^b) \circ \text{out} \circ g, \eta \circ \text{inr}]^\tau \circ \text{out}$.

(iv) The strength $\tau^\tau : X \times T^b a \tau Y \rightarrow T^b a \tau (X \times Y)$ is the unique solution of $\text{out} \circ \tau^\tau = T(\text{id} + (\tau^\tau)^b) \circ \text{out} \circ \delta \circ \tau \circ (\text{id} \times \text{out})$ where $\delta : X \times (Y + (T^b a \tau Y)^b) \rightarrow (X \times Y) + (X \times Y)$. 

256
Let Example 4.4 algebra [7], illustrated in simplified form as follows.

\[
\begin{align*}
T^b_a Y^b & \text{ is the obvious distributivity transformation:} \\
X \times T^b_a Y & \xrightarrow{(T \delta)(id \times out)} T(X \times Y + (X \times T^b_a Y)^b_a) \\
\tau^+ & \downarrow \quad \quad \quad \quad \downarrow \tau(id + (\tau^+)^b_{a}) \\
T^b_a(X \times Y) & \xrightarrow{out} T(X \times Y + T^b_a(X \times Y)^b_a).
\end{align*}
\]

The proof of Lemma 4.1 is facilitated by the fact that \( T(X + (_-)^b_a) \) can be shown to be a parametrized monad [27, Theorems 3.7 and 3.9]. Alternatively, the fact that \( T^b_a \) is a monad can be read off directly from the results of [20]. What is new here is that we show that \( T^b_a \) is, in fact, strong, and hence supports an interpretation of the standard computational metalanguage [17]. This amounts to showing that the strength defined in the last item satisfies the requisite laws [17]. One fact of potentially independent interest used in the (quite involved) proof of these laws is

**Lemma 4.2** For any functor \( G : \mathcal{B} \to \mathcal{C} \), \( out_G : T^b_a G \to T(G + (T^b_a G)^b_a) \) is the final \( T(G + Id^b_a) \)-coalgebra in \([\mathcal{B}, \mathcal{C}]\).

Following Uustalu [27] (and other work [20,1]), we next introduce a notion of guardedness.

**Definition 4.3** (Guardedness) A morphism \( f : X \to T^b_a(Y + Z) \) is guarded if there is \( u : X \to T(Y + T^b_a(Y + Z)^b_a) \) such that \( out \circ f = T(inl + id) \circ u \):

\[
\begin{array}{ccc}
\text{X} & \xrightarrow{f} & T^b_a(Y + Z) \\
\uparrow & & \downarrow \text{out} \\
T(Y + T^b_a(Y + Z)^b_a) & \xrightarrow{T(inl + id)} & T((Y + Z) + (T^b_a(Y + Z))^b_a).
\end{array}
\]

Guardedness of \( f : X \to T^b_a(Y + Z) \) intuitively means that any call to a computation of type \( Z \) in \( f \) occurs only under a free operation, i.e. via the right summand in \( T((Y + Z) + (T^b_a(Y + Z))^b_a) \). A familiar instance of this notion occurs in process algebra [7], illustrated in simplified form as follows.

**Example 4.4** Let \( T \) be the countable powerset monad over a suitable category, i.e. \( TX = \mathcal{P}_{\omega_1}X = \{ Y \subseteq X \mid |Y| \leq \omega \} \). The object \( T_1^\gamma X = \nu \gamma. \mathcal{P}_{\omega_1}(X + A \times \gamma) \) can be considered as the domain of possibly infinite countably nondeterministic processes over actions from \( A \) with final results in \( X \). A morphism \( n \to T_1^\gamma(X + n) \) can be seen as a system of \( n \) mutually recursive process definitions; the latter is guarded in the sense of Definition 4.3 iff every recursive call of a process is preceded by an action, which coincides with the standard notion of guardedness from process algebra. We recall an example of an unguarded definition in this setting in Section 6.

The following result is the main technical contribution of the paper; it states essentially that iteration operators, i.e. Elgot monad structures, propagate uniquely along extensions \( T \to T^b_a \).

257
Theorem 4.5 Let \( T \) be a complete Elgot monad. Given \( a, b \in |C| \), let \( T^b_a \) be the monad identified in Lemma 4.1.

(i) There is a unique iteration operator making \( T^b_a \) a complete Elgot monad that extends iteration in \( T \) in the sense that for \( f : X \to T^b_a(Y + X) \) and \( g : X \to T(Y + X) \), if

\[
out \circ f = (T \text{inl}) \circ g
\]

(i.e. \( f = \text{out}^{-1} \circ (T \text{inl}) \circ g \)) then

\[
out \circ f^\dagger = (T \text{inl}) \circ g^\dagger.
\]

(ii) For any guarded morphism \( f : X \to T^b_a(Y + X) \), \( f^\dagger \) is the unique morphism satisfying the unfolding property \([\eta^\nu, f^\dagger] \circ f = f^\dagger\).

Proof. (Sketch) Uustalu already proves that guarded morphisms \( f \) have unique iterates \( f^\dagger \) [27, Theorem 3.11]. The key step is then to define \( f^\dagger \) for unrestricted \( f \) in a consistent manner. For \( f : X \to T^b_a(Y + X) \), let \( f^\dagger : X \to T^b_a(Y + X) \) be the composite

\[
\begin{align*}
X & \xrightarrow{w^\dagger} T(Y + T^b_a(Y + X)_{a}) \\
& \xrightarrow{T(\text{inl} + \text{id})} T(((Y + X) + T^b_a(Y + X)_{a}) \\
& \xrightarrow{\text{out}^{-1}} T^b_a(Y + X)
\end{align*}
\]

(guarded by definition), where \( w \) is the composite

\[
\begin{align*}
X & \xrightarrow{f} T^b_a(Y + X) \\
& \xrightarrow{\text{out}} T(((Y + X) + T^b_a(Y + X)_{a}) \\
& \xrightarrow{T\pi} T(((Y + T^b_a(Y + X)_{a}) + X)
\end{align*}
\]

with \( \pi = [\text{inl} + \text{id}, \text{inl} \text{inr}] \). That is, \( \triangleright f \) makes \( f \) guarded by iterating

\[
out \circ f : X \to T(((Y + X) + T^b_a(Y + X)_{a})
\]

(in the complete Elgot monad \( T \)) over the middle summand of the result. It is easy to check that \( \triangleright f = f \) when \( f \) is guarded. We hence can define

\[
f^\dagger = (\triangleright f)^\dagger
\]

(in \( T^b_a \)). Further (nontrivial) calculations show that this definition indeed satisfies the axioms of complete Elgot monads.

To establish uniqueness, we first show that any morphism \( f : X \to T^b_a(Y + X) \) can be decomposed into two morphisms \( g : X \to T^b_a(Z + X) \) and \( h : Z \to T^b_a(Y + X) \), where \( Z = Y + T^b_a(Y + X)_{a} \), as

\[
f = [h, \eta^\nu \circ \text{inr}] \circ g
\]

with \( g \) completely unguarded, i.e. \( \text{out} \circ g = (T \text{inl}) \circ g' \) for some \( g' \); that is, we split \( f \) into a guarded part and a completely unguarded one, with iteration on the latter
part being determined by the requirement that iteration on $T_T^b$ extend iteration on $T$. Next we show that for any choice of Elgot monad structure $\_^\dagger$ on $T_T^b$, 

$$f^\dagger = (h^\dagger \circ g^\dagger)^\dagger$$

and that

$$h^\dagger \circ g^\dagger = \triangleright f.$$ 

In summary, we then obtain that $f^\dagger = (h^\dagger \circ g^\dagger)^\dagger = (\triangleright f)^\dagger$, i.e. our previous definition of $f^\dagger$ is the only possible one with the given properties, as $\triangleright f$ is guarded and therefore $(\triangleright f)^\dagger$ is determined uniquely already by the unfolding property. □

The following results characterize $T_T^b$ within the (overlarge) category $\text{CElg}(C)$ of complete Elgot monads over $C$ and (strong) monad morphisms [16] preserving iteration in the evident sense:

**Definition 4.6** A complete Elgot monad morphism $\xi : R \to S$ between complete Elgot monads $R$, $S$ is a morphism $\xi$ between the underlying strong monads (i.e. $\xi \circ \eta = \eta$, $\xi \circ f^* = (\xi \circ f)^* \circ \xi$ for $f : X \to RY$, and $\xi \circ \tau = \tau \circ (\text{id} \times \xi)$) additionally satisfying

$$(\xi \circ g)^\dagger = \xi \circ g^\dagger$$

for $g : X \to R(Y + X)$.

**Lemma 4.7** The natural transformation $\text{ext} : T \to T_T^b$ is a complete Elgot monad morphism.

**Theorem 4.8** Suppose that $\text{CElg}(C)$ has an initial object $L$. Then

(i) $L_T^b$ is the free complete Elgot monad over the signature functor $\bigwedge_T^b : C \to C$;

(ii) For any complete Elgot monad $T$, $T_T^b$ is the coproduct of $T$ and $L_T^b$ in $\text{CElg}(C)$, with left injection $\text{ext} : T \to T_T^b$ (in particular, $\text{ext}$ is a morphism in $\text{CElg}(C)$).

The crucial step in proving Theorem 4.8 is the following statement, which is interesting in its own right.

**Lemma 4.9** Let $a, b \in |C|$ and let $T$, $S$ be two complete Elgot monads. Given a complete Elgot monad morphism $\rho : T \to S$ and a Kleisli morphism $u : a \to Sb$, the transformation $\zeta^\dagger : T_T^b \to S$ with $\zeta$ defined componentwise as the composite

$$T_T^b X \xrightarrow{\text{out}} T(X + a \times (T_T^b X)^b) \xrightarrow{[\eta \circ \text{id} \times \lambda(x,f), S(\text{inr } u(x)) \circ \rho]} S(X + T_T^b X)$$

extends to a complete Elgot monad morphism. Conversely, any $\xi : T_T^b \to S$ induces $\xi \text{ ext} : T \to S$ and

$$a \xrightarrow{\text{out}' \circ \eta \circ \text{inr} \circ (\text{id} \times \lambda)} T_T^b \xrightarrow{\xi} Sb.$$

These two passages are mutually inverse and thus witness a bijection between complete Elgot monad morphisms $T_T^b \to S$ and pairs consisting of Kleisli morphisms $a \to Sb$ and complete Elgot monad morphisms $T \to S$.

The existence and the exact shape of the initial complete Elgot monad $L$ mentioned in Theorem 4.8 depend on the properties of $C$. Recall that $C$ is hyperextensive [2]
if it has countable coproducts that are disjoint and universal (i.e. stable under pullbacks), and coproduct injections are, as subobjects, closed under countable disjoint unions. Examples include Set, \( \omega \text{Cpo} \), and complete metric spaces as well as all presheaf categories.

**Theorem 4.10**  Let \( C \) be hyperextensive. Then the monad \( L \) given by \( LX = X + 1 \) is \( \omega \)-continuous. Equipped with the arising complete Elgot monad structure according to Theorem 3.6, \( L \) is the initial complete Elgot monad over \( C \).

**Proof.** The base category \( C \) is, a fortiori, extensive; in any extensive category, \( L \) is the partial map classifier for partial morphisms whose domains are coproduct injections. Thus, the Kleisli category of \( L \) inherits orderings on its hom-sets from the extension ordering on partial functions; the fact that coproduct injections are closed under unions in \( C \) then guarantees that these orderings are \( \omega \)-complete (note that any ascending chain of coproduct injections qua subobjects can, using universality of coproducts, be transformed into a disjoint union of coproduct injections). Using the properties of hyperextensive categories, one can show that this induces an \( \omega \text{Cpo} \)-enrichment of \( C_L \) that satisfies all additional conditions imposed in Definition 3.3.

To see initiality, note that any complete Elgot monad \( T \) for any \( X \in |C| \) possesses a global element \( \bot_X = \delta_X : 1 \to TX \) where \( \delta_X = \eta \circ \text{inr} : 1 \to T(X + 1) \). It follows by naturality of iteration that \( \bot_X \) is actually natural in \( X \). Moreover, \( \bot \) is preserved by complete Elgot monad morphisms. It is easy to see that \( \xi_X = [\eta, \bot_X] \) yields a complete Elgot monad morphism \( \xi : L \to T \). On the other hand it is the only such because for any other complete Elgot monad morphism \( \theta : \text{L} \to T \) one would have \( \theta \circ \text{inl} = \theta \circ \eta = \eta = \xi \circ \text{inl} \) and \( \theta \circ \text{inr} = \theta \circ \bot = \bot = \xi \circ \text{inr} \) implying \( \theta = \xi \). \( \Box \)

5  Example: Unrestricted While Loops

We use a simple while-language with actions proposed by Rutten, given by the grammar

\[
P, Q ::= A \mid P; Q \mid \text{if } b \text{ then } P \text{ else } Q \mid \text{while } b \text{ do } P
\]

and, following Piróg and Gibbons [20], interpreted in the Kleisli category of a monad \( \mathbb{M} \). Here, \( A \) ranges over atomic actions interpreted as Kleisli morphisms \( \llbracket A \rrbracket : n \to Mn \) for some fixed object \( n \), and \( b \) over atomic predicates, interpreted as Kleisli morphisms \( \llbracket b \rrbracket : n \to M(1 + 1) \) (with the left-hand summand read as ‘false’). We say that \( A \) is of output type if \( \llbracket A \rrbracket \) has the form \( (M \text{fst}) \circ \tau \circ \langle \text{id}_n, p \rangle \) for some \( p : n \to M1 \), and of input type if \( \llbracket A \rrbracket \) factors through \( ! : n \to 1 \). Sequential composition \( P; Q \) is interpreted as Kleisli composition \( \llbracket Q \rrbracket \circ \llbracket P \rrbracket \), and

\[
\llbracket \text{if } b \text{ then } P \text{ else } Q \rrbracket = \llbracket Q \rrbracket \circ \text{fst} \circ \llbracket P \rrbracket \circ \text{fst}^* \circ M \text{ dist} \circ \tau \circ \langle \text{id}, \llbracket b \rrbracket \rangle.
\]

The key point, of course, is the interpretation of the while loop, given in the presence of iteration \( \bot \) by

\[
\llbracket \text{while } b \text{ do } P \rrbracket = \left( (M \text{ inl}) \circ \eta \circ \text{fst} \circ (M \text{ inr}) \circ \llbracket P \rrbracket \circ \text{fst}^* \circ M \text{ dist} \circ \tau \circ \langle \text{id}, \llbracket b \rrbracket \rangle \right)^\dagger.
\]

It has been observed by Piróg and Gibbons that if one instantiates \( \mathbb{M} \) with a completely iterative monad, one needs to guard every iteration of the while loop, i.e.
change the semantics of while to be
\[
\llbracket \text{while } b \text{ do } P \rrbracket = \bigl( (M \text{ inl}) \circ \eta \circ \text{fst}, (M \text{ inr}) \circ \text{fst} \bigr) \circ \left[ M \text{ dist} \circ \tau \circ \langle \text{id}, \llbracket b \rrbracket \rangle \circ \gamma \right]^* 
\]
where \( \gamma : n \to Mn \) is guarded, as otherwise the iteration may fail to be defined. If we instantiate \( M \) with an Elgot monad, such as \( T^b_a \) for an Elgot monad \( T \), then the guard is unnecessary, i.e. we can stick to the original semantics (2). As an example, consider a simple-minded form of processes that input and output symbols from \( n \) and have side effects specified by \( T \); i.e. we work in \( M = (T^1_n)^n \) meaning to use the adjoined free effects \( 1 \to n \) to capture output and those of type \( n \to 1 \) to capture input. We assume an atomic action \( \text{write} \) that outputs a symbol from \( n \), and an atomic action \( \text{read} \) that inputs a symbol. We interpret \( \text{write} \) as being of output type, i.e. by \( \llbracket \text{write} \rrbracket = (M \text{ fst}) \circ \tau \circ \langle \text{id}, w \rangle \) where
\[
w = \text{ext} \circ \text{out}^{-1} \circ \eta \circ \text{inr} \circ \langle \text{id}_n, \eta^\prime \circ \text{snd} \rangle : n \to (T^1_n)^n(1)
\]
(\( \eta^\prime \) being the unit of \( T^1_n \)), while \( \text{read} \) is of input type, i.e. \( \llbracket \text{read} \rrbracket = r \circ !_n \) where
\[
r = \text{out}^{-1} \circ \eta^\prime \circ \text{inr} \circ \langle \text{id}_1, r_0 \rangle : 1 \to (T^1_n)^n
\]
and \( r_0 : 1 \to (T^1_n)^n(n) \) is obtained by currying \( \eta^M \circ \text{snd} : 1 \times n \to (T^1_n)^n(n) \) (\( \eta^M \) being the unit of \( M \)). Moreover, assume a basic predicate \( b \) whose interpretation is largely irrelevant to the example as long as it may take both truth values; for example, \( b \) might just pick a truth value non-deterministically or at random, depending on the nature of the base monad \( T \). Consider the program

\[
\text{read; while true do if } b \text{ then skip else write}
\]

where \( \text{skip} \) is an atomic action interpreted as \( \llbracket \text{skip} \rrbracket = \eta^M : n \to Mn \). It is possible for the loop to not perform any write operations, as \( b \) might happen to always pick the left-hand branch; that is, the loop body fails to be guarded. Since \( M \) is an Elgot monad and not just completely iterative, the semantics of the loop is defined (by (2)) nonetheless.

6 Example: Simple Process Algebra

It is shown in [5, Theorem 5.7.3] that a simple process algebra BSP featuring finite choice and action prefixing can express all countable transition systems if unguarded recursion is allowed. The idea of the proof is to introduce variables \( X_{ik} \) for \( i, k \in \mathbb{N} \) representing the \( k \)-th transition of the \( i \)-th state, with \( X_{i0} \) representing the \( i \)-th state itself, and (unguarded) recursive equations

\[
X_{ik} = b_{ik} \cdot X_{j(i,k),0} + X_{i,k+1}
\]

where the \( k \)-th transition of the \( i \)-th state performs action \( b_{ik} \) and reaches the \( j(i,k) \)-th state. (It is then stated explicitly that the use of unguarded recursion is essential.) To model this phenomenon using the coinductive generalized
resumption transformer, we take $T = \mathcal{P}$, the powerset monad on $\textbf{Set}$ (an Elgot monad by Theorem 3.6) and an operation $\text{act}$ with interpretation $[[\text{act}]] = \text{out}^{-1} \circ \eta \circ \text{inr} \circ (\text{id}_a, \eta \nu!_a) : a \to T^1_a$, where $a$ is the type of actions. That is, we regard (unbounded) nondeterminism as part of the base effect, and add action prefixing via coinductive generalized resumptions. Then the definition (3) is represented by the map $g = \text{out}^{-1} \circ f : \mathbb{N} \times \mathbb{N} \to T^1_a((\mathbb{N} \times \mathbb{N}) + a \times T^1_a((\mathbb{N} \times \mathbb{N}) + b)) \simeq T^1_a(0 + \mathbb{N} \times \mathbb{N})$ with $f : \mathbb{N} \times \mathbb{N} \to T((\mathbb{N} \times \mathbb{N}) + a \times T^1_a(\mathbb{N} \times \mathbb{N}) + b)$. Eliding the exponent 1 given by $f(i, k) = \{ \text{inr}(b_{ik}, \eta \nu(j(i, k), 0)), \text{inl}(i, k + 1) \}$.

Again, our result that $T^1_a$ is an Elgot monad guarantees that this equation has a solution $g^\dagger$; the choice $\dagger$ of solutions in $T^1_a$ is uniquely determined as extending the usual structure of $T = \mathcal{P}$ as an Elgot monad via taking least fixed points.

7 Related Work

The above results benefit from extensive previous work on monad-based axiomatic iteration. In particular we draw on the concept of Elgot monad studied by Adámek et al. [3]; the construction of the free Elgot monad over a functor [4] is strongly related to Theorem 4.8.i, and we do not claim this result as a contribution of this paper. There is extensive literature on solutions of (co)recursive program schemes [6,1,15,11,20,21], from which our present work differs primarily in that we do not restrict to guarded systems of equations. In particular, as mentioned in the introduction, Pirog and Gibbons [20] actually work with the same monad transformer, the coinductive generalized resumption transformer. Pirog and Gibbons [21, Corollary 4.6] also prove a characterization of the coinductive generalized resumption transformer as taking coproducts of monads similar to our Theorem 4.8.ii; but again, this takes place in a different category, that is, in completely iterative monads (admitting guarded recursive definitions) rather than complete Elgot monads (admitting unrestricted corecursive definitions). One consequence of this is that the second summand in our coproduct result is a free Elgot monad and not a free completely iterative monad over $a \times b$, and hence has a built-in notion of divergence. Technically, results on $T^a_b$ being a completely iterative monad are incomparable to our result on $T^b_a$ being a complete Elgot monad – we prove a stronger recursion scheme for $T^a_b$ but need to assume that $T$ is an Elgot monad, while $T^b_a$ is completely iterative without any assumptions on $T$.

We construct solutions of unguarded recursive equations from solutions of guarded recursive equations, for the latter relying crucially on results by Uustalu on guarded recursion over parametrized monads [27], which in particular has allowed us to make do without idealized monads.

The axiomatic treatment of iteration via Elgot monads is essentially dual to the axiomatic treatment of recursion by Simpson and Plotkin [25], who work in a category $\mathbf{D}$ with a parametrized uniform recursion operator $\text{Hom}_\mathbf{D}(Y \times X, X) \to \text{Hom}_\mathbf{D}(Y, X)$ and a subcategory $\mathbf{S}$ of strict functions in $\mathbf{D}$. Given a distributive category $\mathbf{C}$ equipped with a complete Elgot monad, we can take $\mathbf{S} = \mathbf{C}_\text{op}$ and $\mathbf{D} = \mathbf{C}_\text{op}$. Then the iteration operator over $\mathbf{C}_T$ sending $f : X \to T(Y + X)$ to $f^\dagger : X \to TY$ induces precisely a parametrized uniform recursion operator for the
pair \((D, S)\) in the sense of Simpson and Plotkin.

8 Conclusions and Future Work

We have developed semantic foundations for non-wellfounded side-effecting recursive definitions, in the form of iteration, specifically for recursive definitions over the so-called coinductive generalized resumption transformer that constructs from a base monad \(T\) the monad \(\nu \gamma. T(\_ + a \times \gamma^b)\). While previous work on the same monad transformer was focussed on guarded corecursive definitions, in the framework of completely iterative monads, we work in the setting of (complete) Elgot monads, which admit unrestricted recursive definitions. As the core results, we have established that

- \(T^b_a\) is a complete Elgot monad if \(T\) is a complete Elgot monad;
- the structure of \(T^b_a\) as a complete Elgot monad is uniquely determined as extending the one of \(T\);
- if the underlying category \(C\) admits an initial complete Elgot monad \(L\) (typically \(L = \_ + 1\)) then \(T^b_a \cong T + L^b_a\) in the category of complete Elgot monads on \(C\).

In particular this requires proving the equational laws of complete Elgot monads for the solution operator that we construct on \(T^b_a\). Ongoing work is concerned with a formal verification of our results, which are technically quite involved, in the Coq proof assistant; a preliminary version can be found at https://git8.cs.fau.de/redmine/projects/corque.

Besides the fact that applying the coinductive resumption monad transformer to a complete Elgot monad again yields a complete Elgot monad \(T^b_a\), the resulting object obviously has a richer structure provided by the adjoined free operations. One topic for further investigation is to identify (and possibly axiomatize) this structure. Future work is concerned with using this structure for programming definitions of free operations as morphisms \(T^b_a X \to TX\) in a similar spirit as in the paradigm of handling algebraic effects [23]. In conjunction with iteration this actually produces a recursion operator more expressive than iteration. This however requires going beyond the first-order setting of this paper (which was sufficient for iteration), as call-by-value recursion is known to be an inherently higher-order concept.

Acknowledgements The authors wish to thank Stefan Milius and Paul Blain Levy for useful discussions.

References


Abstract

This paper is a tutorial on algebraic effects and handlers. In it, we explain what algebraic effects are, give ample examples to explain how handlers work, define an operational semantics and a type & effect system, show how one can reason about effects, and give pointers for further reading.

Keywords: algebraic effects, handlers, effect system, semantics, logic, tutorial

Algebraic effects are an approach to computational effects based on a premise that impure behaviour arises from a set of operations such as get & set for mutable store, read & print for interactive input & output, or raise for exceptions \[16,18\]. This naturally gives rise to handlers not only of exceptions, but of any other effect, yielding a novel concept that, amongst others, can capture stream redirection, backtracking, co-operative multi-threading, and delimited continuations \[21,22,5\].

I keep hearing from people that they are interested in algebraic effects and handlers, but do not know where to start. This is what this tutorial hopes to fix. We will look at how to program with algebraic effects and handlers, how to model them, and how to reason about them. The tutorial requires no special background knowledge except for a basic familiarity with the theory of programming languages (a good introduction can be found in \[8,15\]).

1 Language

Before we dive into examples of handlers, we need to fix a language in which to work. As the order of evaluation is important when dealing with effects, we split language terms (Figure 1) into inert values and potentially effectful computations,
following an approach called fine-grain call-by-value [13]. There are a few things worth mentioning:

**Sequencing** In do $x ← c_1$ in $c_2$, we first evaluate $c_1$, and once this returns a value, we bind it to $x$ and proceed by $c_2$. If $x$ does not appear in $c_2$, we abbreviate the sequencing to $c_1; c_2$.

**Operation calls** The call $\text{op}(v; y, c)$ passes a parameter value $v$ (e.g. the memory location to be read) to the operation $\text{op}$, and after $\text{op}$ performs the effect, its result value (e.g. the contents of the memory location) is bound to $y$ and the evaluation of $c$, called a continuation, resumes. However, note that encompassing handlers may override this behaviour.

**Generic effects** Having an explicit continuation in the call is convenient for the semantics, but less so for a programmer, who just wants to get back the result of an operation. So, instead of a full-blown operation call, we define a function, called a generic effect [18], also labelled as $\text{op}$, which takes a parameter and passes it to an operation call with the trivial continuation:

$$\text{op} \stackrel{\text{def}}{=} \text{fun } x \mapsto \text{op}(x; y, \text{return } y)$$

Though simpler to use, generic effects are just as expressive because we can recover the operation call $\text{op}(v; y, c)$ by evaluating do $y ← \text{op } v$ in $c$.

**Language extensions** To focus on new constructs, we shall keep our language small, but for examples, we are going to extend its values with integers, primitive arithmetic functions, strings, recursive functions rec fun $f x \mapsto c$, the unit () and pairs $(v_1, v_2)$. Furthermore, we allow patterns in binding constructs (functions, handler clauses, operation calls, and sequencing). In particular, we use the pattern _ to denote ignored parameters, and a pair pattern $(x_1, x_2)$ to extract components from a pair. For example, we bind 7 to $x$ and ignore 8 in the application (fun $(x, _) \mapsto 6 + x$) $(7, 8)$.

**Separation of values & computations** We were a bit lax about the separation of values and computations when writing the last example. Since the addition $6 + x$ is in fact a double application ($(+) 6) x$, the first application $(+) 6$ is already

---

**Fig. 1. Syntax of terms.**

| value $v ::= x$ | variable |
| true | false | boolean constants |
| fun $x \mapsto c$ | function |
| $h$ | handler |

**handler $h ::= handler \{ \text{return } x \mapsto c, \text{op}_1(x; k) \mapsto c_1, \ldots, \text{op}_n(x; k) \mapsto c_n \}$**

| computation $c ::= \text{return } v$ | return |
| \text{op}(v; y, c) | operation call |
| \text{do } x ← c_1 \text{ in } c_2 | sequencing |
| \text{if } v \text{ then } c_1 \text{ else } c_2 | conditional |
| $v_1 v_2$ | application |
| with $v$ handle $c$ | handling |
a computation. Thus, it cannot be applied to $x$ because both subterms of an application must be values. Instead, we need to use sequencing and write the example in our restricted syntax as:

$$(\text{fun } (x, \_)) \mapsto \text{do } f \leftarrow (+) 6 \text{ in } f \, x \, (7, 8)$$

However, this longer form adds little value and makes examples hard to read, so while keeping it in mind, we are going to use the shorter form from now on.

Conversely, we shall implicitly insert $\text{return}$ whenever we use a value where a computation is expected. For example, we shall write $\text{fun } x \mapsto \text{fun } y \mapsto (x, y)$ instead of $\text{fun } x \mapsto \text{return } (\text{fun } y \mapsto \text{return } (x, y))$.

**Semantics** Observe that each operation call creates a branching point in the evaluation, with as many branches as there are possible results that can be yielded to the continuation. For example, $\text{decide}$ will have two branches, $\text{print}$ just one, and $\text{read}$ will have infinite many branches: one for each possible input. Thus, we can imagine computations as trees, whose leaves are returned values and branching points are called operations. For an example, see Figure 2.

![Fig. 2. A computation and a corresponding tree.](image)

In the presence of recursion, some of the leaves of the tree may also be labelled by $\perp$ to indicate a divergent computation that does not call any operations. A divergent computation that repeatedly calls operations is represented by a non-well-founded tree. Denotational semantics is further discussed in Section 6.3.

## 2 Examples

We now informally describe the behaviour of handlers through examples. You may also prefer to first take a look at the operational semantics given in Section 3.

### 2.1 Input & output

Let us start with input & output as it is a very simple algebraic effect, but one which exposes almost all important aspects of handlers. It can be described by two operations: $\text{print}$, which takes a message to be printed and yields the unit value (), and $\text{read}$, which takes a unit value and yields a string that was read. For example, a computation that asks the user for his forename and surname and prints out his
full name, is written as:

\[
\text{printFullName} \equiv \text{print} \text{"What is your forename?";} \\
\text{do forename} \leftarrow \text{read()} \text{ in} \\
\text{print} \text{"What is your surname?";} \\
\text{do surname} \leftarrow \text{read()} \text{ in} \\
\text{print}(\text{join forename surname})
\]

where \text{join} is a function that takes two strings and joins them with a space in the middle.

2.1.1 Constant input
A simple example of a handler is:

\[
\text{handler} \{ \text{read(}_; k \text{)} \mapsto k \text{“Bob"} \}
\]

which provides a constant input string “Bob” each time \text{read} is called. We can, of course, generalise it to a function that takes a string \text{s} and returns a handler that feeds it to \text{read}:

\[
\text{alwaysRead} \equiv \text{fun } s \mapsto \text{handler} \{ \text{read(}_; k \text{)} \mapsto k s \}
\]

This handler works as follows: whenever \text{read} is called, we ignore its unit parameter and capture its continuation in a function \text{k} that expects the resulting string and resumes the evaluation when applied. Next, instead of calling \text{read}, we evaluate the computation in the handling clause: we resume the continuation \text{k}, but instead of reading the string from interactive input, we yield the constant string \text{s}. The handler implicitly continues to handle the continuation, so any \text{read} in the handled computation again yields \text{s}. If the handled computation calls any operation other than \text{read}, the call escapes the handler, but the handler again wraps itself around the continuation so that it may handle any further \text{read} calls. For example, evaluating

\[
\text{with (alwaysRead "Bob") handle printFullName}
\]

first prints out “What is your name?” as \text{print} is unhandled. Then, \text{read} is handled so “Bob” gets bound to \text{forename}. Similarly, the second \text{print} is unhandled, and in the second \text{read}, “Bob” gets bound to \text{surname} as well and finally “Bob Bob” is printed out.

It is not obvious whether handlers should continue handling operations in the continuation, or handle just the first call. Experience with exception handlers offer us no guidance here, because raised exceptions have no continuation, and so the two choices are equivalent. As it turns out, the first choice, which we are settling on in this paper, has nicer denotational semantics, is what one usually desires in practice, and is perhaps also more intuitive because \text{with h handle c} suggests that the whole \text{c} should be handled by \text{h}. The second choice leads to shallow handlers [10], which are more convenient for certain uses, and can be considered a more elementary approach as they can express the usual handlers through recursion.
2.1.2 Reversed output

We can use handlers to not only change what is fed to the continuation, but also to change the way the continuation is used. For example, to reverse the order of printouts, we use:

\[
\text{reverse} \overset{\text{def}}{=} \text{handler} \{ \text{print}(s;k) \mapsto k(); \text{print} s \}
\]

Here, we handle a \text{print} by first calling the continuation, and only after this is finished, print out \(s\). Since the handler wraps itself around \(k\), the same rule applies for the continuation and so all printouts are reversed. So, if we define

\[
abc \overset{\text{def}}{=} \text{print } \text{"A"}; \text{print } \text{"B"}; \text{print } \text{"C"}
\]

then \textbf{with} reverse handle \(abc\) prints out first \text{"C"}, then \text{"B"}, and finally \text{"A"}.

2.1.3 Collecting output

A more useful handler is one that collects all printouts into one big string and returns it together with the final value:

\[
\text{collect} \overset{\text{def}}{=} \text{handler} \{ \text{return } x \mapsto \text{return } (x, \text{"\"})
\]

\[
\text{print}(s;k) \mapsto
\]

\[
d(x,acc) \leftarrow k() \text{ in}
\]

\[
\text{return } (x, \text{join } s \text{ acc})\}
\]

If the handled computation does not print anything and just returns some value \(x\), we need to handle it by returning an empty string in addition to \(x\). But if a computation prints some string \(s\), we resume the continuation. Since this is handled in the same way, it returns the accumulated string \(acc\) in addition to the final value \(x\). Now, we only need to join \(s\) with \(acc\) and return it together with \(x\). If we handle \(abc\) with \text{collect}, we get a pair \(((), \text{"A B C"})\), where () is the unit result of the last \text{print}.

We can also nest handlers, and

\[
\textbf{with} \text{ collect handle} (\textbf{with} \text{ reverse handle} \ abc)
\]

evaluates to \(((), \text{"C B A"})\). The order in which we nest the handlers is significant as it is the innermost handler that determines how to first handle the call. If we switch the handlers in the above example, we get \(((), \text{"A B C"})\) because \text{collect} handles all \text{print} calls, and so none reach the \text{reverse} handler, which then does nothing.

Alternatively, we could implement the same handler using a technique called parameter-passing [22], where we transform the handled computation into a function that passes around a parameter, in our case the accumulated string:

\[
\text{collect'} \overset{\text{def}}{=} \text{handler} \{ \text{return } x \mapsto \text{fun} \ acc \mapsto \text{return } (x,acc)
\]

\[
\text{print}(s;k) \mapsto
\]

\[
\text{fun} \ acc \mapsto (k()) \text{ join } acc \ s\}
\]
When a computation returns a value $x$, there will be no further printouts, so we can return the given accumulator $acc$ in addition to $x$. But if $\text{print}$ is called, we resume the continuation by yielding it the expected unit result. Since the continuation is further handled into a function, we need to pass $k(\cdot)$ the new accumulator, which is $acc$ extended with $s$. To obtain the collected output of a computation $c$, we apply the resulting function to the empty accumulator as:

\[
\text{(with collect' handle c) ""}
\]

In Section 5, we show that $\text{collect}$ and $\text{collect'}$ indeed exhibit equivalent behaviour. Using parameter-passing, we can also implement a converse handler that feeds words from a given string to the input.

### 2.2 Exceptions

Exception handlers are, of course, a special instance of handlers. We represent exceptions with an operation $\text{raise}$ that takes an exception argument (e.g. error message) and yields nothing to the continuation (for more details on how this can be enforced, see Example 4.1).

In practice, exception handlers are rarely reused, but an example of a more general exception handler is:

\[
def \text{default} \equiv \text{fun } x \mapsto \text{handler \{raise(\_; \_)} \mapsto \text{return } x\}
\]

which returns a default value $x$ in case the handled computation raises an exception.

### 2.3 Non-determinism

Handlers can be used not only to override existing effectful behaviour, but to define new one as well. To implement non-determinism, we take a single operation $\text{decide}$, which takes a unit parameter, and non-deterministically yields a boolean. Then, a binary choice can be implemented as a function

\[
def \text{choose} \equiv \text{fun } (x, y) \mapsto \\
\text{do } b \leftarrow \text{decide (\_)} \text{ in} \\
\text{if } b \text{ then (return } x\text{) else (return } y\text{)}
\]

However, unlike $\text{print}$, we assume no intrinsic behaviour for $\text{decide}$, and we must use handlers to determine whether to return a fixed result, a random result, an optimal result, or all results. Without an encompassing handler, an application $\text{choose}(3, 4)$ is stuck when it encounters the $\text{decide}$ call. The simplest handler for $\text{decide}$ is

\[
def \text{pickTrue} \equiv \text{handler \{decide(\_; k) \mapsto k true\}}
\]
which makes each decide yield true to the continuation, so choose always chooses the left argument. So, if we define

\[
\text{chooseDiff } \overset{\text{def}}{=} \text{do } x_1 \leftarrow \text{choose}(15, 30) \text{ in } \\
\text{do } x_2 \leftarrow \text{choose}(5, 10) \text{ in } \\
\text{return } (x_1 - x_2)
\]

then with pickTrue handle chooseDiff will choose 15 for \(x_1\) and 5 for \(x_2\), and will thus evaluate to return 10.

**2.3.1 Maximal result**

With handlers, we can also traverse all possible branches to select the maximal result:

\[
\text{pickMax } \overset{\text{def}}{=} \text{handler } \{ \text{decide}(\_; k) \mapsto \\
\text{do } x_t \leftarrow k \text{ true in } \\
\text{do } x_f \leftarrow k \text{ false in } \\
\text{return } \max(x_t, x_f) \}
\]

In this case, evaluating with pickTrue handle chooseDiff will make the choices needed to get the maximal possible difference 25, even if this means choosing the smaller argument of choose (in particular, we pick 30 for \(x_1\) and 5 for \(x_2\)).

If we included lists in our language, we could adapt pickMax to a handler pickAll that select all possible results [5]. To do so, the return clause would return a singleton list containing the returned value, while the decide clause would concatenate the lists \(x_t\) and \(x_f\) that result from yielding both possible results to the handled continuation.

**2.3.2 Backtracking**

To implement backtracking, where we employ non-deterministic search for a given solution, we add an operation fail to signify that no solution exists. Then, for example:

\[
\text{rec fun chooseInt } (m, n) \mapsto \\
\text{if } m > n \text{ then fail () else } \\
\text{do } b \leftarrow \text{decide () in } \\
\text{if } b \text{ then (return } m) \text{ else chooseInt } (m + 1, n)
\]

is a function that non-deterministically chooses an integer in the interval \([m, n]\), or fails if this interval is empty, while:

\[
\text{pythagorean } \overset{\text{def}}{=} \text{fun } (m, n) \mapsto \\
\text{do } a \leftarrow \text{chooseInt } (m, n - 1) \text{ in } \\
\text{do } b \leftarrow \text{chooseInt } (a + 1, n) \text{ in } \\
\text{if isSquare } (a^2 + b^2) \text{ then (return } (a, b, \sqrt{a^2 + b^2})) \text{ else fail ()}
\]
Pretnar

is a function that searches for an integer Pythagorean triple \((a, b, c)\) such that \(m \leq a < b \leq n\). We perform backtracking by handling each `decide` by first trying to yield `true`, and if this fails, yield `false`:

\[
\text{backtrack} \overset{\text{def}}{=} \text{handler} \{ \text{decide}(-; k) \mapsto \\
\quad \text{with} \\
\quad \text{handler} \{ \text{fail}(-; _) \mapsto k \text{ false} \} \\
\quad \text{handle} \\
\quad k \text{ true} \}
\]

Then, with `backtrack` handle `pythagorean` \((m, n)\) finds \((5, 12, 13)\) for \((m, n) = (4, 15)\) but fails for \((m, n) = (7, 10)\). The exact triple found depends on the implementation of the handler. If, instead, we first tried yielding `false`, the resulting triple for \((m, n) = (4, 15)\) would be \((9, 12, 15)\). To get a list of all possible triples, we can use the handler `pickAll` from Section 2.3.1, but extended with a clause that handles `fail` with an empty list.

### 2.4 State

We represent state with operations `set` for setting the state contents, and `get` for reading them. For simplicity, we assume a single memory location that holds an integer. So, `set` takes an integer, stores it, and returns a unit result, while `get` takes a unit parameter, reads the stored integer, and returns it.

We can use handlers to temporarily alter the stored value or to log all updates. But we can also use them to implement stateful behaviour even if we do not assume a built-in one. Like in Section 2.1.3, we use a parameter-passing handler to pass around the current state:

\[
\text{state} \overset{\text{def}}{=} \text{handler} \{ \text{get}(-; k) \mapsto \text{fun} \ s \mapsto (k \ s) \ s \\
\quad \text{set}(s; k) \mapsto \text{fun} \ _ \mapsto (k ()) \ s \\
\quad \text{return} \ x \mapsto \text{fun} \ _ \mapsto \text{return} \ x \}
\]

We handle `get` with a function that takes the current state \(s\) and passes it first as a result of `get` to the continuation, and then again as the unchanged state. Conversely, we handle `set` by first yielding the unit result, and then applying the handled continuation to the new state \(s\) as given in the parameter of `get`.

The return clause of `state` ignores the final state, but if we want to inspect it, we can return it together with the final value by changing the return clause to:

\[
\text{return} \ x \mapsto \text{fun} \ s \mapsto \text{return} \ (s, x)
\]

#### 2.4.1 Transactions

In a similar way, we can implement transactional memory, where we commit the changed state only after the handled computation successfully terminated with a
value, so in case an exception is raised, the memory contents remain unchanged:

\[
\text{transaction} \triangleq \text{handler} \begin{array}{l}
\text{get}(\_, k) \mapsto \text{fun} \ s \mapsto (k \ s) \ s \\
\text{set}(s; k) \mapsto \text{fun} \ \_ \mapsto (k \ () \ s) \\
\text{return} \ x \mapsto \text{fun} \ s \mapsto \text{set} \ s; \text{return} \ x
\end{array}
\]

3 Operational semantics

To make the intuition about the behaviour of computations concrete, we now give an operational semantics. The idea behind it is that operation calls do not perform actual effects (e.g. printing to an output device), but behave as signals that propagate outwards until they reach a handler with a matching clause. For simplicity, any operation call that escapes all handlers will be treated as a terminating computation, i.e. one that does not further reduce. We can assume that actual effectful behaviour is simulated by an outermost handler, or consider one of the approaches listed in Section 6.5.

In the following rules, we set \(h = \text{handler} \{ \text{return} \ x \mapsto c_r, \text{op}_1(x; k) \mapsto c_1, \ldots, \text{op}_n(x; k) \mapsto c_n \} \):

\[
c \sim c' \quad \text{with} \ h \ \text{handle} \ c \sim \text{with} \ h \ \text{handle} \ c''
\]

\[
\text{if false then} \ c_1 \ \text{else} \ c_2 \sim c_1
\]

\[
\text{if true then} \ c_1 \ \text{else} \ c_2 \sim c_2
\]

\[
(\text{fun} \ x \mapsto c) \ v \sim c[v/x]
\]

\[
\text{with} \ h \ \text{handle} \ \text{return} \ v \ \text{in} \ c \sim c[v/x]
\]

\[
\text{with} \ h \ \text{handle} \ \text{op}_1(v; y. c) \sim c_1[v/x, (\text{fun} \ y \mapsto \text{with} \ h \ \text{handle} \ c)/k] \quad (1 \leq i \leq n)
\]

\[
\text{with} \ h \ \text{handle} \ \text{op}(v; y. \text{with} \ h \ \text{handle} \ c) \quad (\text{op} \notin \{\text{op}_1, \ldots, \text{op}_n\})
\]

Fig. 3. Step relation.

Small-step operational semantics is given using a relation \(c \sim c'\), defined in Figure 3. Observe that there is no such relation for values, as these are inert. The rules for conditionals and function application are standard. For sequencing \(\text{do} \ x \leftarrow c_1 \ \text{in} \ c_2\), we start by evaluating \(c_1\). If this returns some value \(v\), we bind it to \(x\) and evaluate \(c_2\). But if \(c_1\) calls an operation, we propagate the call outwards and defer further evaluation to the continuation of the call, as shown in Figure 4.

Fig. 4. The call of \(\text{op}\) in the innermost sequencing propagates outwards until it reaches the top.

273
For handling with \texttt{with handle} \texttt{c}, the behaviour is similar. We start by evaluating \texttt{c}, and if it returns a value, we continue by evaluating the return clause of \texttt{h}. If \texttt{c} calls an operation \texttt{op}, there are two options: if \texttt{h} has a matching clause for \texttt{op}, we start evaluating that, passing in the parameter and the \texttt{handled} continuation; if not, we propagate the call outwards and defer further handling to the continuation, just like in sequencing.

4 Type system

To ensure that the evaluation goes smoothly, we introduce a type and effect system along the lines presented in [4,10]. Just as we split terms into values and computations, we split types into \textit{value types} and \textit{computation types}, given in Figure 5.

\begin{align*}
\text{value type } A, B & ::= \text{ bool } \\
& | A \rightarrow C \\
& | C \Rightarrow D \\
\text{computation type } C, D & ::= A!\{\text{op}_1, \ldots, \text{op}_n\}
\end{align*}

Fig. 5. Syntax of types.

The value type \( A \rightarrow C \) is given to functions that take a value of type \( A \) and perform a computation of type \( C \), while the handler type \( C \Rightarrow D \) is given to handlers that transform computations of type \( C \) into ones of type \( D \). Every computation type has the form \( A!\Delta \), where \( A \) is the type of values the computation returns, and \( \Delta \) is the set of operations it possibly calls, i.e. the set \( \Delta \) is an over-approximation of the operations that are actually called. Also note that \( \Delta \) contains no information about the number of occurrences, passed parameters, or order of operations.

Typing information about operations is given in a \textit{signature} \( \Sigma \) of the form

\[ \{ \text{op}_1 : A_1 \rightarrow B_1, \ldots, \text{op}_n : A_n \rightarrow B_n \} \]

which assigns a parameter value type \( A_i \) and a result value type \( B_i \) to each operation \( \text{op}_i \).

\textbf{Example 4.1} Assuming that value types are extended with types \texttt{int} of integers, \texttt{str} of strings, \texttt{unit}, which is given to the unit value (\( ) \), and the empty type \texttt{void}, the operations we have seen in Section 2 can be assigned the following types:

\begin{align*}
\text{print} : \text{str} \rightarrow \text{unit} \\
\text{read} : \text{unit} \rightarrow \text{str} \\
\text{raise} : \text{str} \rightarrow \text{void} \\
\text{decide} : \text{unit} \rightarrow \text{bool} \\
\text{fail} : \text{unit} \rightarrow \text{void} \\
\text{get} : \text{unit} \rightarrow \text{int} \\
\text{set} : \text{int} \rightarrow \text{unit}
\end{align*}
Since there are no values of the void type, a call to raise or fail effectively aborts the continuation, because there are no handlers that could resume it by yielding a suitable value.

In Figure 6 we define two typing judgements: \( \Gamma \vdash v : A \) for values and \( \Gamma \vdash c : C \) for computations. In both, the context \( \Gamma \) is an assignment of value types to variables.

\[
\begin{array}{c}
(x : A) \in \Gamma \\
\Gamma \vdash x : A \\
\Gamma \vdash \text{true} : \text{bool} \\
\Gamma \vdash \text{false} : \text{bool} \\
\Gamma, x : A \vdash c : C \\
\Gamma \vdash \text{fun } x \mapsto c : A \rightarrow C \\
\Delta \vdash x : A! \Delta' \\
[\{ \text{op}_i : A_i \rightarrow B_i \} \in \Sigma] \\
\Gamma, x : A_i, k : B_i \vdash B! \Delta' \vdash c_i : B! \Delta'_{1 \leq i \leq n} \\
\Delta \setminus \{ \text{op}_i \}_{1 \leq i \leq n} \subseteq \Delta' \\
\Gamma \vdash \text{handler} \{ \text{return } x \mapsto c, \text{op}_1(x;k) \mapsto c_1, \ldots, \text{op}_n(x;k) \mapsto c_n : A! \Delta \Rightarrow B! \Delta' \}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash v : A \\
\Gamma \vdash \text{return } v : A! \Delta \\
(\text{op} : A_{\text{op}} \rightarrow B_{\text{op}}) \in \Sigma \\
\Gamma \vdash v : A_{\text{op}} \\
\Gamma, y : B_{\text{op}} \vdash c : A! \Delta \\
\text{op} \in \Sigma \\
\Gamma \vdash \text{op}(v, y, c) : A! \Delta \\
\Delta \vdash c_i : A! \Delta \\
\Gamma, x : A \vdash c_2 : B! \Delta \\
\Gamma \vdash \text{do } x \leftarrow c_1 \text{ in } c_2 : B! \Delta \\
\Gamma \vdash \text{if } v \text{ then } c_1 \text{ else } c_2 : C \\
\Gamma \vdash v_1 : C \Rightarrow D \\
\Gamma \vdash v_2 : A \\
\Gamma \vdash c : C \\
\Gamma \vdash \text{with } v \text{ handle } c : D
\end{array}
\]

Fig. 6. Typing judgements.

Typing rules hold no surprises except for:

**Return** You might expect the conclusion to be \( \Gamma \vdash \text{return } v : A!\emptyset \) as that is the most precise type one can assign. However, we give all the rules in a form that allows coarser types because this loses no generality (e.g. in this particular rule, we can set \( \Delta = \emptyset \)), is sufficient for our purposes and leads to a simpler type system. See [23] for an algorithm that produces a more precise type.

**Operation call** Here similarly, we can assume that although \( \Delta \) contains \( \text{op} \), it can be assigned to the continuation \( c \) even when \( c \) does not call \( \text{op} \).

**Handling** According to the above interpretation that \( C \Rightarrow D \) is given to handlers that take computations of type \( C \) to ones of type \( D \); it is not surprising that handling behaves like an application of a function.

**Handler** To give handler a type \( A! \Delta \Rightarrow B! \Delta' \), we need to check that it correctly handles returned values and operations both with and without a matching operation clause. For return values, it is simple: given a value of type \( A \), the return clause must be a computation of type \( B! \Delta' \).

Next, for each handled operation \( \text{op}_i : A_i \rightarrow B_i \), the handling clause again needs to be a computation of type \( B! \Delta' \). Here, the parameter is expected to have the type \( A_i \) as determined by \( \Sigma \). Similarly, the captured continuation is a function that takes a result of type \( B_i \) and performs a computation of type \( B! \Delta' \). Notice that even though the handled computation has type \( A! \Delta \), the continuation has a different type because it is further handled.

Finally, we want to handle computations that call operations without a matching operation clause in the handler. For this case, we allow \( \Delta \) to contain operations not in \( \{ \text{op}_i \}_{1 \leq i \leq n} \), but any such operation must also appear in \( \Delta' \) as it
may also be called in the handled computation (and thus also in continuations of
handled operations).

The given typing system then ensures that well-typed computations do not get
stuck [4].

**Theorem 4.2 (Safety)** If \( \vdash c : A!\Delta \) holds, then either:

- \( c = \text{return } v \) for some \( \vdash v : A \), or
- \( c = \text{op}(v; y, c') \) for some \( \text{op} \in \Delta \), or
- \( c \leadsto c' \) for some \( \vdash c' : A!\Delta \).

## 5 Reasoning

Recall that two terms are \textit{observationally equivalent} [8] if we may exchange any
occurrence of the first with the second without affecting the observable properties
of the surrounding program. Due to the separation in the syntax, we define obser-
vational equivalence of both computations (\( c \equiv c' \)) and values (\( v \equiv v' \)). We can
show [4] that \( \equiv \) is a congruence and that it satisfies a collection of basic equivalences
given in Figure 7.

\[
\begin{align*}
do x &\leftarrow \text{return } v \text{ in } c \equiv c[v/x] \quad &(1) \\
do x &\leftarrow \text{op}(v; y, c_1) \text{ in } c_2 \equiv \text{op}(v; y, \text{do } x \leftarrow c_1 \text{ in } c_2) \quad &(2) \\
do x &\leftarrow c \text{ in return } x \equiv c \quad &(3) \\
do x_2 &\leftarrow (\text{do } x_1 \leftarrow c_1 \text{ in } c_2) \text{ in } c_3 \equiv \text{do } x_1 \leftarrow c_1 \text{ in } (\text{do } x_2 \leftarrow c_2 \text{ in } c_3) \quad &(4) \\
\text{if true then } c_1 \text{ else } c_2 &\equiv c_1 \quad &(5) \\
\text{if false then } c_1 \text{ else } c_2 &\equiv c_2 \quad &(6) \\
\text{if } v \text{ then } c[\text{true}/x] \text{ else } c[\text{false}/x] &\equiv c[v/x] \quad &(7) \\
(\text{fun } x \mapsto c) v &\equiv c[v/x] \quad &(8) \\
\text{fun } x \mapsto v x &\equiv v \quad &(9)
\end{align*}
\]

In the following rules, we have \( h = \text{handler } \{ \text{return } x \mapsto c_r, \text{op}_1(x; k) \mapsto c_1, \ldots, \text{op}_n(x; k) \mapsto c_n \} \):

\[
\begin{align*}
\text{with } h \text{ handle } (\text{return } v) &\equiv c_r[v/x] \quad &(10) \\
\text{with } h \text{ handle } (\text{op}_i(v; y, c)) &\equiv c_i[v/x, (\text{fun } y \mapsto \text{with } h \text{ handle } c)/k] \quad (1 \leq i \leq n) \quad &(11) \\
\text{with } h \text{ handle } (\text{op}(v; y, c)) &\equiv \text{op}(v; y, \text{with } h \text{ handle } c) \quad (\text{op} \notin \{\text{op}_i\}_{1 \leq i \leq n}) \quad &(12) \\
\text{with } (\text{handler } \{ \text{return } x \mapsto c_2 \}) \text{ handle } c_1 &\equiv \text{do } x \leftarrow c_1 \text{ in } c_2 \quad &(13)
\end{align*}
\]

Fig. 7. Basic equivalences.

The main new tool we can use for reasoning about algebraic effects is the \textit{induction
principle} [20,4], which states that for a given predicate \( \phi \) on computations, \( \phi(c) \)
holds for all computations \( c \) if:

(i) \( \phi(\text{return } v) \) holds for all values \( v \), and

(ii) \( \phi(\text{op}(v; y, c')) \) holds for all operations \( \text{op} \) and parameters \( v \), if we assume that
\( \phi(c') \) holds for all possible results \( y \).

We can use the induction principle to derive equivalences (3), (4), and (13), but
for a more interesting example, let us show that handlers \textit{collect} and \textit{collect'} from
Section 2.1.3 exhibit equivalent behaviour, in particular:

\[
\text{with collect handle } c \equiv \text{ do } g \leftarrow (\text{with collect' handle } c) \text{ in } g "\text{"}
\]

To succeed with induction, we need to prove a stronger statement that for any string \( s_0 \), we have

\[
\text{do } (x_1, s_1) \leftarrow (\text{with collect handle } c) \text{ in return } (x_1, \text{join } s_0 s_1) \equiv \\
\text{do } g \leftarrow (\text{with collect' handle } c) \text{ in } g s_0
\]

We recover the desired goal by setting \( s_0 = "\" \). The induction on \( c \) goes as follows:

(i) The base case is trivial: if \( c = \text{return } v \), both sides are equal to \( \text{return } (v, s_0) \).

(ii) For the induction step when \( c = \text{op}(v; y, c') \), we have two possibilities: either \( \text{op} \neq \text{print} \), which is again trivial, or \( \text{op} = \text{print} \), where we show:

\[
\text{do } (x_1, s_1) \leftarrow (\text{with collect handle } \text{print}(s_2; \_c')) \text{ in return } (x_1, \text{join } s_0 s_1) \\
\equiv (11) \& (8)
\]

\[
\text{do } (x_1, s_1) \leftarrow (\text{with collect handle } c') \text{ in return } (x_1, \text{join } s_2 \text{acc}) \\
\equiv (4)
\]

\[
\text{do } (x, \text{acc}) \leftarrow (\text{with collect handle } c') \text{ in return } (x, \text{join } s_0 s_1) \\
\equiv (1)
\]

\[
\text{do } (x, \text{acc}) \leftarrow (\text{with collect handle } c') \text{ in return } (x, \text{join } s_0 (s_2 \text{acc})) \\
\equiv \text{ (associativity of join)}
\]

\[
\text{do } (x, \text{acc}) \leftarrow (\text{with collect handle } c') \text{ in return } (x, \text{join } (s_0 s_2) \text{acc}) \\
\equiv \text{ (induction hypothesis)}
\]

\[
\text{do } f \leftarrow (\text{with collect' handle } c') \text{ in } f \text{ (join } s_0 s_2) \\
\equiv (1) \& (8)
\]

\[
\text{do } g \leftarrow \text{return } ( \text{fun acc} \mapsto \text{do } f \leftarrow (\text{with collect' handle } c') \text{ in } f \text{ (join acc } s_2) \\
) \text{ in } g s_0 \\
\equiv (11) \& (8)
\]

\[
\text{do } g \leftarrow (\text{with collect' handle } \text{print}(s_2; \_c')) \text{ in } g s_0
\]

6 Further reading

6.1 Call-by-push-value

Call-by-push-value [12] is an evolved version of the fine-grain call-by-value approach. Though the latter was used in this tutorial as it is closer to the more familiar call-by-value, a significant part of the recent work on algebraic effects uses the former.
To compare given operational semantics and effect system to ones done in a call-by-push-value setting, see [10], while for denotational semantics and reasoning, see [22].

6.2 Programming with handlers

The list of examples in Section 2 is by no means exhaustive. For more involved examples that include multi-threading, delimited continuations, selection functionals, text processing, resource management, efficient backtracking, or logic programming, see [5,10,6,25]. A number of implementations of handlers has also sprung up, either as independent languages [3,14], or as libraries in existing languages [10,6,25]. More recently, a multicore [2] branch of OCaml [1] has started adopting handlers as a way of implementing concurrency primitives.

6.3 Denotational semantics

In the naive setting where operations return only first-order values and there is no recursion, we can interpret each value type \( A \) with a set \( [A] \), while a computation type \([A!\Delta]\) is interpreted as the set of trees (like ones described in Section 1) with leaves in \([A]\) and nodes corresponding to operations in \( \Delta \). Handlers are interpreted as functions between trees, and are defined by structural recursion on the tree of the handled computation, while handling is interpreted by application of such functions.

More abstractly, we define a model of \( \Delta \) to be a set \( M \) together with a map \( \text{op}_M: [A] \times M^{[\Delta]} \to M \) for each operation \( \text{op}: A \to B \in \Delta \), while a homomorphism between models \( M \) and \( N \) is defined to be a map \( h: M \to N \) such that \((h \circ \text{op}_M)(x,k) = \text{op}_N(x,h \circ k)\). It turns out that \([A!\Delta]\) is exactly the free model of \( \Delta \) over \([A]\), i.e. a model characterized with the following universal property: given any model \( M \) of \( \Delta \) and any map \( f: [A] \to M \), there exists a unique homomorphism \( h: [A!\Delta] \to M \) that agrees with \( f \) on leaves. We can use this universal property to interpret handlers: operation clauses define a model of operations, and the return clause provides a function \( f \) that can be extended to a homomorphism.

For more detail, see [22]. In the general setting with recursion and higher-order results, we need to switch from sets to domains, but the general idea is the same [4].

6.4 Algebraic theories

Traditionally, algebraic effects were described not only by a set of operations, but also by an equational theory that captures their properties. For example, nondeterminism can be represented with a binary operation \( \text{decide} \) and equations stating its idempotency, commutativity, and associativity [18,9,17]. The benefit of equations is that they validate certain program optimizations [11] and better capture the effectful behaviour of operations. With various extensions of such theories, one can also describe complicated effects such as control-flow jumps [7] even in the absence of handlers, or quantum computation [24].

However, a lot of computationally interesting handlers (for example \text{backtrack} from Section 2.3.2) do not respect these equations and thus cannot receive a homomorphic interpretation described above [22]. For this reason, current research on handlers assumes no such equations, but connections exists in both directions:
on one hand, we can still apply previous results by assuming a trivial equational theory, and on the other hand, we can use reasoning techniques to recover equations from the behaviour of handlers [4].

6.5 Modelling actual effects

One can model “real-world” effects with a comodel, which is a set $W$ representing the possible world states together with a map $op^W : W \times \llbracket A \rrbracket \to W \times \llbracket B \rrbracket$ for each operation $op : A \to B \in \Sigma$. Thus, when an operation call $op(v; y, c)$ escapes all handlers, we pass the current state $w \in W$ and the parameter $v$ to $op^W$ and get back the new state and a result, which we assign to $y$ and continue evaluating $c$. For more details, see [5, Section 4.1], which is based on a more abstract treatment in [19], where the duality between models and comodels is explained in more detail.

Acknowledgement

I want to thank Andrej Bauer and Alex Simpson for their truly helpful feedback.

References

  URL http://ocaml.org

  URL https://github.com/ocamllabs/ocaml-multicore

  URL http://www.eff-lang.org

  URL http://dx.doi.org/10.2168/LMCS-10(4:9)2014

  URL http://dx.doi.org/10.1016/j.jlamp.2014.02.001

  URL http://dx.doi.org/10.1145/2500368.2500381

  URL http://dx.doi.org/10.1145/2603088.2603163


  URL http://dx.doi.org/10.1016/j.tcs.2006.03.013

  URL http://dx.doi.org/10.1145/2500368.2500590

  URL http://doi.acm.org/10.1145/2103656.2103698


Dirichlet is natural

Vincent Danos¹
Département d’Informatique
École normale supérieure
Paris, France

Ilias Garnier²
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom

Abstract
The categorical treatment of probabilities due to Giry (and Lawvere), and G, the Giry probabilistic monad, offer an elegant and under-exploited treatment of higher-order probabilities. There are opportunities for the development of an approximation theory that would parallel the one for Markov processes, promote a novel perspective on Bayesian learning and its computational underpinnings, and provide a stepping stone of a theory of uncertain Markov processes (see Introduction).

To begin to do this, we need to find a computational handle on higher-order probabilities. It turns out that such an object already exists and is known as the Dirichlet process. This is a family of higher-order probabilities indexed by measures, which is at the core of nonparametric Bayesian learning. The specific goal of this paper is to reconstruct this object in Giry style.

Our contributions are as follows. Given a Polish space X, we build a measurable family indexed by non-zero finite measures over X of higher-order probabilities in G(G(X)). The construction relies on two fundamental ingredients. First, we develop a decomposition/recomposition method whereby we map any zero-dimensional Polish space X to a projective system of finite approximations, the limit of which is the (universal in zero-dimensional Polish spaces) compactification of X. Second, we use a refined (functorial) version of Bochner’s probability extension theorem on Polish spaces, where consistent systems of probabilities over a projective system give rise to an actual probability on the limit. The above, once combined with known combinatorial results on Dirichlet processes on finite spaces (also known as Dirichlet distributions), allows one to obtain the Dirichlet family D_X on X as a natural transformation from the monad of non-zero finite measures to G o G. When composed with the monad multiplication, Dirichlet gives the “normalisation” of (finite non-zero) measures. In addition, for any Polish space X, we find that D_X is continuous. Finally, our construction generalises to Dirichlet-like processes generated by infinitely divisible distributions on the positive reals.

Keywords: probability, topology, category theory, monads

1 Introduction
Bisimulation metrics arise naturally from exact bisimulations in situations where one does not know exactly the transition probabilities of a Markovian model. The

¹ vincent.danos@ens.fr
² ilias.garnier@ens.fr
³ Even though the existence of symmetries in physical systems can sometimes lead to exact bisimulations which depend only on structure and not on the actual values of transition probabilities [26]. There are at-

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
model needs to be taken with a grain of salt, and uncertainties lead one to use approximate notions of equivalence as a way of comparing processes. Here, we take a new look at this central issue of uncertainty in the model and explore a novel and richer framework to deal with it. The idea is to impute uncertainty not only to the means of comparison (ie the Kantorovich metrics and its lifting to MCs) but also to the process itself (after all, in the first option we say that the model is uncertain but do not do anything to quantify that). We wish to build a clean and tractable framework to quantify uncertainty in the description of Markov chains. Specifically, we propose to explore “uncertain Markov chains” as elements of type $X \rightarrow G^2(X)$, where $X$ is an object of $\textbf{Pol}$, the category of Polish spaces (separable and completely metrisable spaces) and $G$ is the Giry probability functor. This is to say that the chain takes values in random probabilities (ie a probability of probabilities). As we will see below, it will be key to find good parametric families of random probabilities. This natural treatment of behavioural uncertainty in probabilistic models will allow one to formulate a notion of (Bayesian) learning and therefore to obtain notions of 1) models which can learn under observations and 2) of behavioural comparisons which incorporate data and reductions in uncertainty. Beyond the proximal goal of building flexible data-assimilative behavioural metrics, there is also a distal reward in building at the interface with machine learning, a topic of increasing importance in the domain of formal methods.

One needs to set up a sufficiently general framework for learning under observation within the coalgebraic approach. Learning a probability in a Bayesian framework is naturally described as a (stochastic) process of type $G^2(X) \rightarrow G^2(X)$ (so $G^2(X) \rightarrow G^3(X)$ really!) driven by observations. For finite $X$s this setup poses no difficulty, but for more general spaces, one needs to construct a computational handle on $G^2(X)$ - the space of uncertain or higher-order probabilities. This is what we do in this paper.

To this effect, we build a theory of Dirichlet-like processes in $\textbf{Pol}$. Dirichlet processes [14,1] form a family of elements in $G^2(X)$ indexed by finite measures over $X$ [1, p.17] and which is closed under Bayesian learning. Integral to our construction is a method of “decomposition/recomposition” which allows us to build higher-probabilities via finite approximations (the limit of which lead to a compactification of the original space). In order to lift finite higher-probabilities we use an extension theorem of the Kolmogorov-Bochner type in $\textbf{Pol}$ (Sec. 2.3). Kolmogorov consistent assignments of probabilities on finite partitions of measurable spaces (or finite joint distributions of stochastic processes) can be seen systematically as points in the image under $G$ of projective (countable co-directed) diagrams in $\textbf{Pol}$.

Using the above we show that Dirichlet-like processes in $\textbf{Pol}$ can be seen as natural transformations from $M^*$ (the monad of non-zero finite measures on $\textbf{Pol}$) to $G^2$ built up from finite discrete spaces. The finite version of naturality goes under the name of “aggregation laws” in the statistical literature and can be traced back to the “infinite divisibility” of the one building block, namely the $\Gamma$ distribution. (This opens up the possibility of an axiomatic version of the construction presented here, see conclusion.)

\footnote{\textit{tempus}, parallel to bisimulation metrics, at defining robustly the satisfaction of a temporal logic formula [12]}

\footnote{Eg as for Poisson point processes.}

282
2 Notations & basic facts

A useful reference on general topology is [10]. Weak convergence of probability measures is treated in [5, 25].

2.1 Finite measures on Polish spaces and the Giry monad

Weak topology

A measure $P$ on a topological space $X$ is a positive countably additive set function defined on the Borel $\sigma$-algebra $B(X)$ verifying $P(\emptyset) = 0$. We will only consider finite measures on Polish spaces, i.e. $P(X) < \infty$. When $P(X) = 1$, $P$ is a probability measure. We write $G(X)$ for the space of all probability measures over $X$ with the weak topology [5, 25], the initial topology for the family of evaluation maps $EV_f = P \mapsto \int_X f dP$ where $f$ ranges in $C_b(X)$ and where $(C_b(X), \|\cdot\|_\infty)$ is the Banach space of real-valued continuous bounded function over $X$ with the sup norm. A neighbourhood base for a measure $P \in G(X)$ is given by the sets

$$N_P(f_1, \ldots, f_n, \epsilon_1, \ldots, \epsilon_n) = \left\{ Q \left| \left| \int f_i dP - \int f_i dQ \right| < \epsilon_i, 1 \leq i \leq n \right\}$$

where $f_i \in C_b(X), \epsilon_i > 0$. One can restrict w.l.o.g. to the subset of real-valued bounded uniformly continuous functions, noted $U_b(X)$. Importantly, if $X$ is Polish the weak topology on $G(X)$ is also Polish (see e.g. Parthasarathy, [25] Chap. 2.6) and metrisable by the Wasserstein-Monge-Kantorovich distance [27]. We denote the convergence of a sequence $(P_n \in G(X))_{n \in \mathbb{N}}$ to $P \in G(X)$ in the weak sense by $P_n \rightharpoonup P$. The support of a probability $P \in G(X)$ is noted $supp(P)$ and is defined as the smallest closed set such that $P(supp(P)) = 1$. For $X,Y$ Polish and $P \in G(X), Q \in G(Y)$, we write $P \otimes Q \in G(X \times Y)$ the product probability, so that $(P \otimes Q)(B_X \times B_Y) = P(B_X)Q(B_Y)$.

Giry monad

The operation $G$ can be extended to a functor $G : Pol \rightarrow Pol$ compatible with the Giry monad structure $(G, \eta, \mu)$ [17]. For any continuous map $f : X \rightarrow Y$ we set $G(f)(P) = B \in B(Y) \mapsto P(f^{-1}(B))$, i.e. $G(f)(P)$ is the pushforward measure. For a given $X$, $\eta_X : X \rightarrow G(X)$ is defined as the Dirac delta $\eta_X(x) = \delta_x$ while $\mu_X : G^2(X) \rightarrow G(X)$ is defined as averaging: $\mu_X(P) = B \in B(X) \mapsto \int_{G(X)} \int_B \int_B \mu_X(B) \mapsto \int_B \int_B \mu_X(B)$ where $EV_B = Q \in G(X) \mapsto Q(B)$ evaluates a probability on the Borel set $B$. We have the “change of variables” formula: for all $P \in G(X), f : X \rightarrow Y$ and $g : Y \rightarrow \mathbb{R}$ bounded measurable, $\int_Y gdG(f)(P) = \int_X g \circ f dP$. Finally, $G$ preserves surjectivity, injectivity and openness:

**Lemma 2.1** (i) $f : X \rightarrow Y$ is injective if and only if $G(f)$ is injective;

(ii) $f$ is surjective if and only if $G(f)$ is surjective.

(iii) If $f$ is an embedding, so is $G(f)$. 283
Finite measures

The set of all finite non-negative Borel measures on a Polish space, noted \( M(X) \), is a Polish space when endowed with the weak topology ([7] Theorem 8.9.4). \( M : \text{Pol} \to \text{Pol} \) is a functor extending \( G \), mapping continuous functions to the corresponding pushforward morphism. The monad multiplication \( \mu_X \) can be conservatively extended to a morphism from \( M^2(X) \) to \( M(X) \) by defining \( \mu_X(P) = B \in B(X) \mapsto \int_{M(X)} EVdP \). The everywhere zero measure, noted \( 0 \), is an element of \( M(X) \) that we might want to exclude: \( M(X) \) being Hausdorff implies that the set of nonzero measures \( M^+(X) \triangleq M(X) \setminus \{0\} \) is open, hence \( G_\delta \), hence Polish as a subspace of \( M(X) \). A measure \( Q \in M(X) \) is strictly positive if for all nonempty open sets \( U \subseteq X \), \( Q(U) > 0 \). Equivalently, \( Q \) is strictly positive if and only if \( \text{supp}(Q) = X \).

**Lemma 2.2** Strictly positive finite measures on a Polish space \( X \) form (when they exist) a Polish subspace of \( M(X) \). We denote this subspace by \( M^+(X) \).

Summing up, we have for \( X \) Polish the following inclusions of Polish spaces of finite measures:

\[
M^+(X) \subseteq M^*(X) \subseteq M(X)
\]

Note also that \( M \) and \( M^* \) are endofunctors on \( \text{Pol} \) but \( M^+ \) is not, unless one restricts to the subcategory of epimorphisms.

Normalisation of measures

We note \( \nu_X : M^*(X) \to G(X) \) the continuous map taking any measure \( Q \in M^*(X) \) to its normalisation \( \nu_X(Q) \triangleq B \in B(X) \mapsto Q(B)/|Q| \), where \( |Q| \triangleq Q(X) \) is the total mass of the measure. \( \nu_X \) verifies an useful property:

**Lemma 2.3** For every continuous map \( f : X \to Y \), \( G(f) \circ \nu_X = \nu_Y \circ M^+(f) \).

Finitely supported measures

When \( X \) is a finite, discrete space such that \( X = \{x_1, \ldots, x_n\} \), \( G(X) \) is in bijection with the simplex \( \Delta_n \subseteq \mathbb{R}^n \), where \( \Delta_n = \{(p_1, \ldots, p_n) \subseteq \mathbb{R}^n \mid p_i \geq 0, \sum p_i = 1\} \). Notice that \( \Delta_n \) is an \( n-1 \) dimensional space. \( M(X) \) corresponds to the positive orthant, noted \( \mathbb{R}^n_{\geq 0} \). Since for \( X \) finite \( G(X) \) is (topologically) a subspace of a finite dimensional vector space, it is homeomorphic to \( \Delta_n \cap \mathbb{R}^n \) while the topology of \( M(X) \) corresponds to that of \( \mathbb{R}^n_{\geq 0} \cap \mathbb{R}^n \). If we note \( n \) the \( n \)-element set, we in particular have the trivial identities \( M(n) = \mathbb{R}^n_{\geq 0} \) and \( M(m) \times M(n) = M(m+n) \).

2.2 Projective limits of topological spaces

Many of our theorems will deal with spaces obtained as projective limits (also known as inverse limits or cofiltered limits) of topological spaces. These topological projective limits are defined as adequate topologisations of projective limits in \( \text{Set} \), the usual category of sets and functions.

Let \( (I, \leq) \) be a directed partially ordered set seen as a category and let \( D : I^{\text{op}} \to \text{Set} \) be a cofiltered \( \text{Set} \) diagram. The projective limit of \( D \) is a terminal cone...
(lim $D, \pi_i$) over $D$ where $\lim D$ is the set

$$\lim D \triangleq \{ x \mid D(i \leq j)(\pi_j(x)) = \pi_i(x) \} \subseteq \prod_i D(i)$$

and the $\pi_i : \prod_j D(j) \to D(i)$ are the canonical projections. Notice that $D$ is contravariant from $I$ to $\text{Set}$. As emphasised in the definition, $\lim D$ is the subset of the cartesian product $\prod_i D(i)$ containing all sequences of elements that respect the constraints imposed by the diagram $D$. The elements of $\lim D$ are called threads and the maps $D(i \leq j) : D(j) \to D(i)$ are the bonding maps. Of course, $\lim D$ can be empty (see [30] for a short example). A sufficient condition to ensure non-emptiness of the limit is to consider functors $D$ where $I$ is countable and the bonding maps are surjective [4]. As a convenience, we will note those bonding maps as $\pi_{ij} \triangleq D(i \leq j)$, and we write countable cofiltered surjective diagrams $ccd$ for short.

Writing $U : \text{Top} \to \text{Set}$ for the underlying set functor, cofiltered limits in $\text{Top}$ for diagrams $D : I^{\text{op}} \to \text{Top}$ are obtained by endowing the $\text{Set}$ limit of $U \circ D$ with the initial topology for the canonical projections $\{ \pi_i \}_{i \in I}$. The following useful additional fact follows by considering $\lim D$ as the intersection of the (closed) subsets of $\prod_i D(i)$ satisfying $D(i \leq j)(\pi_j(x)) = \pi_i(x)$ for all pairs $(i, j)$ s.t. $i \leq j$.

Lemma 2.4 ([10], Ch. 1, §8.2, Corollaire 2) $\lim D$ is a closed subset of $\prod_i D(i)$.

2.3 The Bochner extension theorem

The construction of a stochastic process given a system of consistent finite-dimensional marginals is an important tool in probability theory, a classical example being the construction of the Brownian motion using the Kolmogorov extension theorem [23]. Besides Kolmogorov’s there are many other variants, collectively called Bochner extension theorem [22]. They differ in the amount of structure of the space over which probabilities are considered (measurable, topological or vector spaces) – and we will make crucial use of the Bochner extension theorem for Polish spaces, which admits a particularly elegant presentation:

Theorem 2.5 For all $D$ a ccd in $\text{Pol}$, $G(\lim D) \cong \lim G \circ D$. We denote by $bcn : \lim G \circ D \to G(\lim D)$ this homeomorphism.

In words, the Bochner extension theorem states that any projective family of probabilities that satisfy the diagram constraints (elements of $\lim G \circ D$) can be uniquely lifted to a probability over the limit space (elements of $G(\lim D)$) – and what’s more, this extension is a homeomorphism! This presentation of the Bochner extension seems not to be well-known: a similar statement is given in Metivier ([22], Theorem 5.5) in the case of locally compact spaces, which intersects but does not include Polish spaces; Fedorchuk proves the continuity of $G$ on the class of compact Hausdorff spaces in [13] while more recently Banakh [3] provides an extension theorem in the more general setting of Tychonoff spaces, using properties of the Stone-Čech compactification.

285
3 Finitely supported approximants of Polish measures

It is natural in applications to consider finitary approximations of stochastic processes. Accordingly, the correctness of such approximations should correspond to some kind of limiting argument, stating that increasingly finer approximations yield in some suitable sense the original object. In view of the Bochner extension theorem, it suffices to consider as input a projective family of probabilities supported by the finitary approximants of the underlying space. However, the very same theorem tells us that we can only obtain by this means probabilities on a projective limit of finite spaces (also called profinite spaces), a rather restrictive class:

**Proposition 3.1** A space is a countable projective limit of finite discrete spaces if and only if it is a compact, zero-dimensional Polish space.

The proof can be found under a slightly different terminology in Borceux & Janelidze [8], where it is shown that these spaces correspond to Stone spaces—indeed, profinite spaces are exactly the spaces homeomorphic to the Stone dual of their Boolean algebra of clopen sets! As the proof of this proposition is quite enlightening for the developments to come, we provide it here.

**Proof.** Let $D : I^{op} \to \text{Pol}_{fin}$ be a ccd of finite spaces. Polishness of $\lim D$ comes from the closure of $\text{Pol}$ under countable limits. Finite spaces are compact and by Tychonoff’s theorem so is $\prod_i D(i)$. Lemma 2.4 asserts that $\lim D$ is closed in this compact product, hence $\lim D$ is itself compact. Recall that $\lim D$ has the initial topology for the canonical projections maps $\pi_i : \lim D \to D(i)$, therefore a base of $\lim D$ is constituted of finite intersections of prebase opens $\pi_i^{-1}(X_i)$, for $X_i \subseteq D(i)$. Since the $D(i)$ are discrete, any of their subsets is clopen and so are the prebase opens; we conclude by noticing that a finite intersection of clopen sets is again clopen.

Conversely, let $X$ be a compact zero-dimensional Polish space. As $X$ is zero-dimensional Polish, its topology is generated by a countable base of clopen sets. Since $X$ is compact, each clopen can be written as a finite union of base clopens. Therefore its algebra of clopens $\mathcal{C}(X)$ is also generated by the same countable base, and is itself countable. Note that $\mathcal{C}(X)$ does not depend on the choice of the base! Let us consider the set $\mathcal{I}(X)$ of all finite discrete quotients of $X$. For any $i \in \mathcal{I}(X)$, there exist by assumption a continuous surjective quotient map $f_i : X \to i$. Since $i$ is discrete, the fibres of $f_i$ are clopen. Conversely, any finite partition of $X$ by clopens induces an element of $\mathcal{I}(X)$, therefore $\lim D$ is also countable. $\mathcal{I}(X)$ is partially ordered by partition refinement: for all $i, j \in \mathcal{I}(X)$, we write $i \leq j$ if there exists a surjective map $f_{ji} : j \to i$ such that $f_{ji} \circ f_j = f_i$ (any such map, if it exists, is unique). $\mathcal{I}(X)$ is also directed by considering pairwise intersections of the cells of any two partitions. The system of finite discrete quotients of $X$ together with the bonding maps $f_{ji}$ clearly defines a ccd that we write $D : \mathcal{I}(X)^{op} \to \text{Pol}_{fin}$, mapping each element of $\mathcal{I}(X)$ to itself and the partial order of bonding maps to the opposite order. Therefore, there exists a limit cone $(\lim D, \pi_i)$. By universality of this cone, there exists a unique continuous map $e_X : X \to \lim D$ s.t. $f_i = \pi_i \circ e_X$. Let us show that $e_X$ is an homeomorphism. As $\lim D$ and $X$ are both compact, it is enough to show that $e_X$ is a bijection. Recall that $\mathcal{C}(X)$ separates points (it contains a base for
a Hausdorff topology) therefore for any \( x \neq y \in X \) we can exhibit two clopen cells separating them, implying that \( e_X \) is injective. Surjectivity of \( e_X \) is a consequence of that of the quotient and bonding maps.

We see that the projective limit space of the finite quotients of any Polish space \( X \) is in general not the original space, but something quite different! As a consequence, the limit space of the finitary approximations of \( G(X) \) will also not be what we expected. This prompts the search for additional conditions under which such a finitary approximation can be properly carried out.

As is known from the theory of Stone duality, \( \lim D \) as constructed in the proof of Prop. 3.1 corresponds to the set of ultrafilters on the Boolean algebra of clopens \( C(X) \) \cite{19}, for \( X \) compact zero-dimensional Polish. This is a (trivial) instance of another famous construction in topology: the Stone-Čech compactification \( \beta X \) \cite{29}.

This insight yields a route for the finitary approximation of measures on a more general (non-compact) Polish space \( X \): obtain from \( X \) a Polish zero-dimensional compactification (which in our case will not be Stone-Čech), homeomorphic by the previous discussion to a limit of finite quotients, and define the approximants on the finite quotients. The last and hardest step is to prove that the projective limit measure on the compactification suitably restricts to the starting space. We start by reviewing the properties of Polish zero-dimensional compactifications.

### 3.1 Zero-dimensional Polish compactification

Since zero-dimensionality is hereditary, we can only hope to obtain a zero-dimensional compactification if we start from a zero-dimensional space. The proof of Prop. 3.1 shows how to reconstruct a zero-dimensional compact Polish space as a limit of its finite clopen quotients. Let us reiterate this construction on a zero-dimensional, non-necessarily compact Polish space \( X \). In opposition to the compact case, the Boolean algebra \( C(X) \) of clopens of \( X \) is not necessarily countably generated: we therefore consider partitions of \( X \) taken in some sub-algebra \( C \subseteq C(X) \) such that \( C \) is generated by a countable base \( \mathcal{F} \triangleq \{ C_n \}_{n \in \mathbb{N}} \) of clopens of \( X \), which is equivalent (\( \mathcal{F} \) being closed under finite intersections) to consider partitions taken directly in \( \mathcal{F} \). We will show that the choice of a particular base \( \mathcal{F} \) does not matter, up to isomorphism. We define:

\[
\mathfrak{F}(X) \triangleq \{ \mathcal{F} \mid \mathcal{F} \text{ is a countable clopen base for } X \}
\]

We write \( \mathcal{I}_\mathcal{F}(X) \) for the directed partial order of clopen partitions of \( X \) taken in \( \mathcal{F} \in \mathfrak{F}(X) \). Since \( \mathcal{F} \) is countable, so is \( \mathcal{I}_\mathcal{F}(X) \). We recall that the construction of \( \mathcal{I}_\mathcal{F}(X) \) is described in the proof of Prop. 3.1.

**Proposition 3.2** For \( \mathcal{F} \in \mathfrak{F}(X) \), let \( D_\mathcal{F} : \mathcal{I}_\mathcal{F}(X) \to \text{Pol}_{\text{fin}} \) be the diagram of finite clopen quotients of \( X \), then \( \lim D_\mathcal{F} \) is a zero-dimensional compactification of \( X \).

**Proof.** Existence of \( \lim D_\mathcal{F} \) stems from the countability of \( \mathcal{F} \), \( \lim D_\mathcal{F} \) is trivially compact, so it is enough to prove that \( X \) densely embeds in \( \lim D_\mathcal{F} \). We affirm that the universal mediating map \( e_X : X \to \lim D_\mathcal{F} \) is the following: \( e_X = x \mapsto (i \mapsto f_i(x)) \), where \( f_i : X \to i \) is the quotient surjection. This is well-defined: for
each partition \( i, x \) belongs to exactly one clopen cell, \( f_i(x) \). For any \( i \in \mathcal{I}_F(X) \) we also clearly have \( f_i = \pi_i \circ e_X \). Injectivity of \( e_X \) and density of \( e_X(X) \) in \( \lim D_F \) are routine. Consider a point \( x \in X \) and a clopen neighbourhood \( f_i(x) \) of \( x \). Clearly, \( e_X(f_i(x)) = \pi_i^{-1}(f_i(x)) \cap e_X(X) \), which is open by definition, therefore \( e_X \) is an homeomorphism on its image. By the same reasoning, note that \( e_X \) is also closed. Hence \( \lim D_F \) is a zero-dimensional compactification of \( X \).

The following asserts that the choice of countable clopen base does not matter.

**Proposition 3.3** Let \( \mathcal{F}_1, \mathcal{F}_2 \in \mathfrak{F}(X) \) be two countable clopen bases. Then there exists an unique isomorphism \( h_{\mathcal{F}_1,\mathcal{F}_2} : \lim D_{\mathcal{F}_1} \to \lim D_{\mathcal{F}_2} \).

**Proof.** Countable clopen bases are trivially closed under union. Let \( \mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2 \). For any clopen \( C \) in \( \mathcal{F} \) there exists a finer clopen \( C_1 \subseteq C \) in \( \mathcal{F}_1 \). Accordingly, any partition \( i \in \mathcal{I}_F(X) \) is refinable by a partition \( j \in \mathcal{I}_{\mathcal{F}_1}(X) \). We deduce that \( \mathcal{I}_{\mathcal{F}_1} \) is cofinal in \( \mathcal{I}_F \). Prop. 3., §7, Chapter 3 of [9] yields that there exists a canonical (set) bijection \( h_{\mathcal{F}_1} : \lim D_{\mathcal{F}_1} \to \lim D_{\mathcal{F}_2} \). As both spaces are compact, openness, which is trivial, entails that \( h_{\mathcal{F}_1} \) is a homeomorphism. The same holds for \( \mathcal{F}_2 \), which allows to conclude.

This compactification can be presented slightly differently, as the space of ultrafilters over a countable Boolean algebra of clopens containing a base of clopens. This type of compactification is sometimes designated as being of **Wallman type** ([29], 1.19). Together with Prop. 3.3, this justifies the following notation:

**Definition 3.4** For \( X \) Polish zero-dimensional, we note \( wX \) the compactification corresponding to Prop. 3.2.

As \( wX \) is always a profinite space, Prop. 3.1 ensures there always exists a cofiltered diagram \( D \) in \( \text{Pol}_{\text{fin}} \) such that \( \lim D \cong wX \). We will switch from one point of view to the other freely. We should insist on the fact that our compactification is not the Stone-Čech compactification, as these are in general not metrisable (except when compactifying an already metrisable compact space, obviously). Take for instance the discrete (hence zero-dimensional) Polish space \( \mathbb{N} \): \( \beta\mathbb{N} \) has cardinality \( 2^{\aleph_0} \) ([29], Theorem 3.2) while Polish spaces have cardinality at most \( 2^{\aleph_0} \). However, \( wX \) enjoys an analogous property to that of Stone-Čech:

**Proposition 3.5** Let \( X \) be a Polish zero-dimensional space embedding through \( e_X : X \to wX \) into its zero-dimensional Polish compactification. Then for each continuous map \( f : X \to K \) to a compact zero-dimensional Polish space \( K \), there exists a continuous extension \( w f : wX \to K \) such that \( w f \circ e_X = f \).

**Proof.** Prop. 3.1 entails that there exists a ccd \( D_K : I^{op} \to \text{Pol}_{\text{fin}} \) s.t. \( K \cong \lim D_K \), with limit cone \( (\lim D_K, \{ \pi_i : \lim D_k \to D_k(i) \}_{i \in I}) \). Note that by continuity of \( \pi_i \circ f \), each finite quotient of \( K \) is a finite quotient of \( X \) (inducing a clopen partition of the latter). By choosing a good base of clopens, we can exhibit a compactification \( wX' \cong wX \) (Prop. 3.3) with an associated cone \( \langle wX', \{ \lambda_i : wX' \to D_K(i) \} \rangle \) and therefore an unique map \( w f : wX' \to K \) such that \( w f \circ e_X = f \) (up to the isomorphism \( wX \cong wX' \)).
Corollary 3.6 For any continuous \( f : X \to Y \) between zero-dimensional spaces, there exists a map \( w f : w X \to w Y \) such that \( w f \circ e_X = e_Y \circ f \), i.e. \( w \) acts functorially on zero-dimensional spaces.

3.2 Projective limit measures on zero-dimensional compactification

For \( X \) Polish zero-dimensional, the developments of Sec. 3.1 allow us to map any measure in \( G(X) \) to \( G(w X) \) through \( G(e_X) \). Crucially, thanks to Lemma 2.1 this is a faithful operation. Therefore any measure on \( X \) can be obtained, up to isomorphism, as a projective limit of finitely supported measures. However, as pointed out before, the converse operation is the difficult one. Let \( D \) be a diagram such that \( w X \cong \lim D \) and \( \{P_i\}_i \in \lim G \circ D \) a projective family of finitely supported probabilities. There is in general no way to assert that the corresponding projective limit probability \( P \in G(w X) \) obtained through the Bochner extension theorem restricts to \( G(X) \). We delineate the most general conditions under which a probability can be restricted to a subspace and propose a simplification of previous arguments [24], based on the properties of the Giry monad. Note that the results to follow are not specific to zero-dimensional spaces. Polish subspaces of Polish spaces are always \( G_\delta \) sets (and conversely, see [21], 3.11), hence Borel sets. This allows for a simple restriction criterion.

Proposition 3.7 Let \( P \in M(Y) \) be a finite measure on a Polish space \( Y \) and let \( X \subseteq Y \) be a Polish subspace (hence a \( G_\delta \) in \( Y \)). The restriction of \( P \) to \( X \), defined as the set function \( P|_X \triangleq (B \in \mathcal{B}(Y) \cap X) \mapsto P(B) \), verifies \( P|_X \in G(X) \) if and only if \( P(X) = 1 \).

This criterion lifts to “higher-order” probabilities, that is probabilities over spaces of probabilities, thanks to the multiplication of the Giry monad. The following theorem states that such a higher order probability measure restricts to a subspace if and only if it restricts in the mean. This is essentially Theorem 1.1 in [24].

Theorem 3.8 For all \( X \subseteq Y \) Polish spaces and all \( P \in G^2(Y) \) we have \( P|_{G(X)} \in G^2(X) \) if and only if \( (\mu_Y(P))|_X \in G(X) \).

3.3 Zero-dimensional refinements of Polish spaces

The previous discussion leaves aside the adequacy of zero-dimensional Polish spaces for doing probability theory. It turns out that for any Polish space, one can find a refinement of its topology that is Polish zero-dimensional. There is not necessarily a unique such refinement though! However, we will show that any Polish space has the final topology for a particular family of zero-dimensional refinements, that we will construct. We recall the following two lemmas, taken verbatim from Kechris [21], Sec. 13:

Lemma 3.9 For any Polish space \((X, T_X)\) and any closed set \( A \), there exists a Polish topology \( T_{X_A} \) so that \( T_X \subseteq T_{X_A} \), \( A \) is clopen in \( T_{X_A} \) and \( \mathcal{B}(T_X) = \mathcal{B}(T_{X_A}) \). \( T_X \cup \{O \cap A \mid O \in T_X\} \) is a base of \( T_{X_A} \).
Lemma 3.10  Let $(X, \mathcal{T}_X)$ be Polish and let $\{\mathcal{T}_{X_n}\}_{n \in \mathbb{N}}$ be a family of Polish topologies on $X$, then the topology $\mathcal{T}_{X_\infty}$ generated by $\bigcup_n \mathcal{T}_{X_n}$ is Polish. Moreover if $\forall n, \mathcal{T}_{X_n} \subseteq \mathcal{B}(\mathcal{T}_X)$, then $\mathcal{B}(\mathcal{T}_{X_\infty}) = \mathcal{B}(\mathcal{T}_X)$.

Let $(X, \mathcal{T}_X)$ be Polish with countable base $\mathcal{F} = \{O_n\}_{n \in \mathbb{N}}$. Let $\mathcal{F}^c_\delta \triangleq \{X \setminus O_n \mid O_n \in \mathcal{F}\}_\delta$ be the closure under finite intersections of the complements of elements of $\mathcal{F}$. Let us note $\mathcal{T}_X | \mathcal{F}^c_\delta \triangleq \{O \cap D \mid O \in \mathcal{T}_X, D \in \mathcal{F}^c_\delta\}$.

Proposition 3.11  The topology generated by the base $\mathcal{T}_X \cup \mathcal{T}_X | \mathcal{F}^c_\delta$ is Polish and zero-dimensional. We write the corresponding topological space $Z_F(X)$. Moreover, we have $\mathcal{B}(\mathcal{T}_X) = \mathcal{B}(\mathcal{T}_{Z_F(X)})$.

Proof. Consider the family of Polish topologies $\{\mathcal{T}_{X_{O_n}}\}_{n \in \mathbb{N}}$, as obtained using Lemma 3.9. Closure under finite intersections of $\bigcup_n \mathcal{T}_{X_{O_n}}$ yield that the topology generated by $\bigcup_n \mathcal{T}_{X_{O_n}}$ has base $\mathcal{T}_X \cup \mathcal{T}_X | \mathcal{F}^c_\delta$. Lemma 3.10 entails that the resulting space is indeed Polish. An equivalent base to $\mathcal{T}_X \cup \mathcal{T}_X | \mathcal{F}^c_\delta$ is $\mathcal{F} \cup \mathcal{F}^c_\delta$ and the elements of this base are clopen, hence the resulting space is also zero-dimensional. □

To the best of our knowledge, we can’t do away with the dependency on $\mathcal{F}$: one can exhibit a Polish space $X$ with two distinct bases $\mathcal{F}, \mathcal{G}$ such that $Z_{\mathcal{F}}(X) \neq Z_{\mathcal{G}}(X)$. Despite this apparent lack of canonicity, any Polish topology is entirely determined by its collection of zero-dimensional refinements:

Theorem 3.12  Any Polish space $X$ has the final topology for the family $\{id_F : Z_F(X) \to X\}_F$ of all the (continuous) identity maps from its zero-dimensionalisations, where $\mathcal{F}$ ranges over all the countable bases of $X$.

The proof of this theorem relies on the following lemma.

Lemma 3.13  Let $X$ be a Polish space and $(x_n)_{n \in \mathbb{N}} \to x$ a convergent sequence in $X$. Let $\mathcal{F}$ be a countable base for $X$. $(x_n)_{n \in \mathbb{N}}$ converges to $x$ in $Z_F(X)$ if $x \not\in \bigcup_{O \in \mathcal{F}} \partial O$.

Proof. Recall that a countable base of $Z_F(X)$ is $\mathcal{F} \cup \mathcal{F} | \mathcal{F}^c_\delta$. Assume $x$ is not in the boundary of any element $O \in \mathcal{F}$. Let $U$ be a basic open neighbourhood of $x$ in $Z_F(X)$. If $U \in \mathcal{F}$ then it is trivial to exhibit the convergence property by referring to the topology of $X$ only. If not, we have that $U = O \cap D$, where $D = \cap_{i=1}^n X \setminus O_i$, $O_i \in \mathcal{F}$; in other terms $x \in (X \setminus \bigcup_{i=1}^n O_i) \cap O$. Observe that since the $O_i$ are open, $\overline{O}_i = O_i \cup \partial O_i$ – therefore, using the initial assumption, we have $x \in (X \setminus \bigcup_{i=1}^n \overline{O}_i) \cap O$, which is an open set in $X$. The result follows. □

Proof. (Proof of Theorem 3.12) It suffices to prove that for all topological space $Y$, a function $f : X \to Y$ is continuous if and only if $f \circ id_F : Z_F(X) \to Y$ is continuous for all countable base $\mathcal{F}$. The forward implication is trivial. Assume that for all countable base $\mathcal{F}$, $f \circ id_B : Z_F(X) \to Y$ is continuous. Consider a converging sequence $(x_n)_{n \in \mathbb{N}} \to x$ in $X$. It is sufficient to exhibit one space $Z_F(X)$ where this sequence also converges. Lemma 3.13 gives as a sufficient criterion that $x$ does not belong to $\partial O$ for any $O \in \mathcal{F}$. Let us build such a base. Consider a dense set $\mathcal{D}$ of $X$. Let $d : X^2 \to [0, 1]$ be some metric that completely metrises $X$. Without loss of generality, assume $x \in \mathcal{D}$. Write $r_n \triangleq d(x, d_n)$ for $d_n \in \mathcal{D} \setminus \{x\}$. For all $n$, take the family of open balls centred on each $d_n$ with rational radii strictly below $r_n$, e.g.
Since $\text{diam}(B(d_n, r_n/3)) = \text{diam}(B(d_n, r_n/3))$, $x \notin \partial B(d_n, r)$ for $r < r_n/3$. This family still constitutes a neighbourhood base. The countable union of countable sets is countable, therefore it constitutes a countable base of $X$. 

Notice that as a consequence of Lemma 3.9, the identity function $id : Z_F(X) \to X$ is continuous. Functoriality of $G$ then yields a continuous map $G(id) : G(Z_F(X)) \to G(X)$. However, observe that the topologies of $G(X)$ and $G(Z_F(X))$ might be different, and there is in general no continuous map from $G(X)$ to $G(Z_F(X))$. It should also be emphasised that the “zero-dimensionalisation” of a Polish space is not an innocent operation: for instance if $X$ is compact and non-zero-dimensional then $Z_F(X)$ will never be compact! Since $G$ preserves compactness, this lifts to spaces of probabilities.

4 The Dirichlet process

The Dirichlet process stands out among other Bayesian methods in that the prior and posterior distributions are second order probabilities, that is elements of $G^2(X)$. Learning becomes an operation of type $X \to G^2(X) \to G^2(X)$, mapping some evidence in $X$ and a prior in $G^2(X)$ to a posterior in $G^2(X)$, and it can be proved that the second-order stochastic process induced by sampling from identically and independently distributed random variable will converge (in Kullback-Leibler divergence, hence in the weak topology [16]) to a singular distribution over the law of the target.

4.1 The Dirichlet distribution

For a fixed finite discrete space $X$, Dirichlet is a function $\mathcal{D}_X : M^*(X) \to G^2(X)$, the parameter in $M^*(X)$ representing the initial prior as well as the degree of certainty about this prior (encoded in its total mass). As we highlight below, $\mathcal{D}_X$ is continuous and verifies other properties, among which naturality and normalisation:

**Proposition 4.1 (Naturality)** $\mathcal{D} \triangleq X \mapsto \mathcal{D}_X$ is a natural transformation $\mathcal{D} : M^* \Rightarrow G^2$ when $M^*$ and $G^2$ are restricted to $\text{Pol}_{\text{fin}}$.

**Proposition 4.2 (Normalisation)** $\mu_X \circ \mathcal{D}_X = \nu_X$

4.2 Extension to zero-dimensional Polish spaces

The finite support case is instructive but lacks generality. We proceed to the extension of finitely supported Dirichlet distributions to Dirichlet processes supported by arbitrary zero-dimensional Polish spaces. Our construction preserves both naturality and continuity – in fact, it can be framed as the extension of the natural transformation $\mathcal{D}$ from $\text{Pol}_{\text{fin}}$ to $\text{Pol}_{\z}$, the full subcategory of zero-dimensional Polish spaces and continuous maps. In what follows, we denote by $F|_C : C \to \text{Pol}$ the restriction of the domain of some endofunctor $F : \text{Pol} \to \text{Pol}$ to a subcategory $C$ of $\text{Pol}$. When unambiguous, we drop this notation.

**Theorem 4.3** There exists a unique (up to isomorphism) natural transformation $\tilde{\mathcal{D}} : M|_{\text{Pol}_{\z}} \Rightarrow G^2|_{\text{Pol}_{\z}}$ such that $\tilde{\mathcal{D}}$ coincides with $\mathcal{D}$ on $\text{Pol}_{\text{fin}}$. 

291
versality of $L^\otimes M$ so by Prop. 3.1 there exists a ping cone $C(G)$.

Applying the finitary Dirichlet space $X$.

Existence. Let $\varepsilon_X : X \to wX$ be the embedding of a Polish zero-dimensional space $X$ into its compactification $wX$ (Def. 3.4). $wX$ is compact zero-dimensional so by Prop. 3.1 there exists a ccd of finite spaces $D$ such that $wX \cong \lim D$. Let us construct $\hat{\mathcal{D}}_{\lim D}$, the extension of Dirichlet to $\lim D$ (see Fig. 1). Applying the functor $M^\ast$ yields a cone $C_{\lim D} = (M^\ast(\lim D), \{M^\ast(\pi_i) : M^\ast(\lim D) \to (M^\ast \circ D)(i)\}_i)$. Applying the finitary Dirichlet $\mathcal{D}$ on the base of this cone yields a ccd in $G^2 \circ D$, of which we take the limit, obtaining a terminal cone $L = (\lim G^2 \circ D, \{\rho_i : \lim G^2 \circ D \to G^2(D(i))\}_i)$. By naturality of $\mathcal{D}$, the cone $C_{\lim D}$ extends to a cone $C'_{\lim D} = (M^\ast(\lim D), \{\mathcal{D}(\lim D) \circ M^\ast(\pi_i) : M^\ast(\lim D) \to (G^2 \circ D)(i)\}_i)$. By universality of $L$, there exists a unique morphism $u : M^\ast(\lim D) \to G^2(\lim D)$ mapping $C'_{\lim D}$ to $L$. The Bochner extension theorem (Thm 2.5) yields an isomorphism $G(\varepsilon_X) : \lim G^2 \circ D \to G^2(\lim D)$ (the fact that $G(\varepsilon_X)$ is an isomorphism is a consequence of Lemma. 2.1). This yields a morphism

$$\hat{\mathcal{D}}_{\lim D} : M^\ast(\lim D) \to G^2(\lim D)$$

$$\hat{\mathcal{D}}_{\lim D} = u \circ G(\varepsilon_X) \circ \beta_n$$

that trivially coincides with $\mathcal{D}$ when $\lim D$ happens to be finite. In order to conclude the existence part of the extension, we need to show that $\hat{\mathcal{D}}_{\lim D} \circ M^\ast(\varepsilon_X) : M^\ast(X) \to$
$G^2(\lim D)$ actually ranges in $G^2(e_X(X)) \subseteq G^2(\lim D)$, after which we can set $\widehat{\mathcal{D}}_X \triangleq (\widehat{\mathcal{D}}_{\lim D} \circ M^*(e_X)).$ By Theorem 3.8, it suffices to check that for any $Q \in M^*(X)$,

\[
(\mu_{\lim D} \circ \widehat{\mathcal{D}}_{\lim D} \circ M^*(e_X))(Q) \bigg|_{e_X(X)} \in G(e_X(X))
\]

which by Prop. 3.7 amounts to checking that this measure attributes full measure to $e_X(X)$. We take advantage of the normalisation property (Prop. 4.2) of $\mathcal{D}$. Thanks to this property and to the naturality of $\mu$, the diagram in Fig. 2 commutes. The Bochner extension theorem entails the universality of the cone $(G(\lim D), \{G(\pi_i)\}_i)$ at the top of the diagram, therefore commutation of the diagram in Fig. 2 entails the existence of a unique morphism from the cone $(M^*(\lim D), \{\nu^i_{D(i)\circ M^*(\pi_i)}\}_i)$ to $(G(\lim D), \{G(\pi_i)\}_i)$ (morphism represented as a dashed line in Fig. 2). This morphism is no other than the normalisation $\nu_{\lim D} : M^*(\lim D) \to G(\lim D)$. Therefore,

\[
(\mu_{\lim D} \circ \widehat{\mathcal{D}}_{\lim D} \circ M^*(e_X)) (Q) = (\mu_{\lim D} \circ M^*(e_X)) (Q)
\]

Trivially, $M^*(e_X)(Q)(X \setminus e_X(X)) = 0$, therefore $(\mu_{\lim D} \circ M^*(e_X))(Q)$ is concentrated on $e_X(X)$. Hence, up to isomorphism, $\mathcal{D}_X$ restricts to a morphism $\mathcal{D}_X : M^*(X) \to G^2(X)$. This concludes the proof of existence.

**Naturality.** For any map $f : X \to Y$ between zero-dimensional Polish spaces, we must prove $\widehat{\mathcal{D}}_Y \circ M^*(f) = G^2(f) \circ \widehat{\mathcal{D}}_X$. By Corollary 3.6, we can reduce the task to the case of a morphism $w : wX \to wY$ between compact zero-dimensional spaces (see Fig. 3b). It remains to prove $\widehat{\mathcal{D}}_{wY} \circ M^*(w) = G^2(w) \circ \widehat{\mathcal{D}}_{wX}$. By Prop. 3.1, $wX \cong \lim D_X$ and $wY \cong \lim D_Y$ where $D_X$ and $D_Y$ are their respective finite discrete quotient ccds. Let us write $(wY, \{\pi_i : wY \to D_Y(i)\}_i)$ the terminal cone corresponding to $D_Y$. The universal property of this limit cone allows to reduce the goal to the commutation of the diagram in Fig. 3b:

\[
\widehat{\mathcal{D}}_Y \circ M^*(f) = G^2(f) \circ \widehat{\mathcal{D}}_X \iff \forall i, G^2(\pi_i) \circ \widehat{\mathcal{D}}_Y \circ M^*(f) = G^2(\pi_i) \circ G^2(f) \circ \widehat{\mathcal{D}}_X
\]

\[
\iff \forall i, \mathcal{D}_{D_Y(i)} \circ M^*(\pi_i) \circ M^*(f) = G^2(\pi_i) \circ G^2(f) \circ \widehat{\mathcal{D}}_X
\]

As already argued in the proof of Prop. 3.5, any finite discrete quotient of $wY$ is a finite discrete quotient of $wX$ since the two spaces are related by a continuous function $w : wX \to wY$, therefore the diagram in Fig. 3b commutes.

**Uniqueness.** Assume there exists two distinct natural transformations $\widehat{\mathcal{D}}, \widehat{\mathcal{D}}' : M^*(X) \to G^2(X)$ that coincide with $\mathcal{D}$ on finite spaces. It is clear that it is enough to exhibit a contradiction in the case of $X$ compact zero-dimensional Polish. We refer to Fig. 1 for the notations. Let $D$ be a ccd of finite spaces such that $X \cong \lim D$, with canonical projections $\pi_i : \lim D \to D(i)$. By assumption, there must exist a measure $Q \in M^*(X)$ such that $\widehat{\mathcal{D}}(Q) \neq \widehat{\mathcal{D}}'(Q)$. But both $\widehat{\mathcal{D}}$ and $\widehat{\mathcal{D}}'$ verify (by assumption of naturality and consistency with the finitary case) the equalities $G^2(\pi_i) \circ \widehat{\mathcal{D}}_X = \mathcal{D}_{D(i)} \circ M^*(\pi_i) = G^2(\pi_i) \circ \widehat{\mathcal{D}}_X$ for all $i$. Therefore, $Q$ induces through $\widehat{\mathcal{D}}$ and $\widehat{\mathcal{D}}'$ the same projective family of finite-dimensional Dirichlet distributions $\{\mathcal{D}_{D(i)} \circ M^*(\pi_i)(Q)\}_i$, which yields (by unicity of extensions, see Theorem 2.5) a contradiction. \(\square\)
The set of countable bases of


Proof.

Lemma 4.4

Let \( \mathcal{X} \) be Polish and \( \mathcal{Z}_F(X), \mathcal{Z}_G(X) \) be two zero-dimensional refinements as constructed in Prop. 3.11. Then \( \mathcal{D}_{\mathcal{Z}_F(X)} \) and \( \mathcal{D}_{\mathcal{Z}_G(X)} \) are equal in Set.

Proof. The set of countable bases of \( \mathcal{X} \) is directed by union. Let us write \( \mathcal{H} \equiv \mathcal{F} \cup \mathcal{G} \).

The (continuous) identity functions \( id_F : \mathcal{Z}_H(X) \rightarrow \mathcal{Z}_F \) and \( id_G : \mathcal{Z}_H(X) \rightarrow \mathcal{Z}_G \) lift to identity functions \( G^2(id_F), G^2(id_G) \), and similarly for the functor \( M^* \). Therefore, the commutation relation \( G^2(id_F) \circ \mathcal{D}_{\mathcal{Z}_H(X)} = \mathcal{D}_{\mathcal{Z}_F(X)} \circ M^*(id_F) \) boils down in Set to the equality of \( \mathcal{D}_{\mathcal{Z}_F(X)} \) and \( \mathcal{D}_{\mathcal{Z}_G(X)} \).

Theorem 4.5

There exists a unique (up to isomorphism) natural transformation \( \mathcal{T} : M^* \Rightarrow G^2 \) such that \( \mathcal{T} \) coincides with \( \mathcal{T} \) on \( \text{Pol} \).

Proof. Let \( \mathcal{X} \) be a Polish space. Consider the family \( \{ \mathcal{Z}_F(X) \} \) of its zero-dimensional refinements, as constructed in Prop. 3.11. For each \( \mathcal{Z}_F(X) \), Theorem 4.3 asserts the existence of a continuous Dirichlet map \( \mathcal{D}_{\mathcal{Z}_F(X)} : M^*(\mathcal{Z}_F(X)) \rightarrow G^2(\mathcal{Z}_F(X)) \), which extends by continuity of the identity and functoriality to a continuous map

\[
G^2(id_F) \circ \mathcal{D}_{\mathcal{Z}_F(X)} : M^*(\mathcal{Z}_F(X)) \rightarrow G^2(X)
\]

(1)

By Lemma 4.4, all these maps coincide in Set. Let us prove that this Set map, that we will note \( \mathcal{D}_X \), is continuous from \( M^*(\mathcal{X}) \) to \( G^2(\mathcal{X}) \). Let \( (Q_n)_{n \in \mathbb{N}} \rightarrow Q \) be a weakly converging sequence of measures in \( M^*(\mathcal{X}) \). By definition, this is equivalent to having \( \int_X f dQ_n \rightarrow \int_X f dQ \) for all \( f \in C_b(X) \). By finality of \( \mathcal{X} \) w.r.t. the \( \{ \mathcal{Z}_F(X) \} _{\mathcal{X}} \), this is in turn equivalent to having \( (Q_n)_{n \in \mathbb{N}} \rightarrow Q \) in \( M^*(\mathcal{Z}_F(X)) \) for all countable bases \( \mathcal{F} \). Therefore, \( \mathcal{D}_X(Q_n) \rightarrow \mathcal{D}_X(X) \) in \( G^2(\mathcal{X}) \) if and only if \( \mathcal{D}_X \circ M^*(id_F)(Q_n) \rightarrow \mathcal{D}_X \circ M^*(id_F)(Q) \) for all countable base \( \mathcal{F} \). This is in turn equivalent to \( G^2(id_F) \circ \mathcal{D}_{\mathcal{Z}_F(X)}(Q_n) \rightarrow G^2(id_F) \circ \mathcal{D}_{\mathcal{Z}_F(X)}(Q) \) for all \( \mathcal{F} \), which holds by continuity of \( 1 \). Therefore \( \mathcal{D}_X \) is continuous. The other stated properties follow from Theorem 4.3. \( \square \)
Our construction of the Dirichlet process in categorical style subsumes existing ones \cite{15,24} while establishing continuity and naturality. However, further work, which we intend to pursue right away, is required to consolidate our understanding of the finitary approximation framework we have built for higher-order probabilities. Our construction relies heavily on the properties of Polish spaces: for instance we use the fact that zero-dimensional Polish spaces are Borel sets of their compactifications (Prop. 3.7); the measurable selection theorem used in Lemma 2.1 also requires the spaces considered to be Polish. The process by which we rebuild Dirichlet relies on some simple properties of \( \Gamma \) distributions. Naturality is a consequence of closure of \( \Gamma \) under convolution (a particular case of *infinite divisibility* also exhibited by e.g. normal distributions), and the fact that Dirichlet restricts to \( X \), which is only a subset of its compactification \( \mathcal{w}X \), follows from the normalisation property (see §4.1). By axiomatising these properties, we can generalise our main result. However, it remains to be seen whether other interesting distributions on \( \mathbb{R}_{>0} \) fit the conditions and generate Dirichlet-like processes.

Beyond the immediate questions above, we can return to the less immediate goals expounded on in the introduction, namely higher-order learning using uncertain chains of Dirichlet type. Any uncertain Markov chain \( \tau \), meaning a morphism \( X \to G^2(X) \) in \( \textbf{Pol} \), can be post-composed with the multiplication of \( G \) to obtain the “mean” Markov chain of type \( X \to G(X) \). We will investigate the case where \( \tau \) takes values in Dirichlet processes -focusing on the tractable “uncertain chains of Dirichlet type”. Such chains can be decomposed as \( \alpha : X \to G(X) \) followed by the Dirichlet natural transformation \( D_X : M^*(X) \to G^2(X) \). The first component \( \alpha \) is \( \tau \)'s parameterising chain. As \( \mu \circ D_X \circ \alpha \) is the normalised version of \( \alpha \), \( \alpha \) is again up to normalisation the mean chain of the uncertain \( \tau \). Our construction ensures that \( \tau \) is continuous by construction. At this stage, it is already possible given \( \tau : X \to G^2(X) \) to quantify the uncertainty at each point by considering moments of the “Kantorovich” random variable

\[
K_X \triangleq (P \in G(X)) \mapsto d_K((\mu \circ \tau)(x), P),
\]

where \( K_X \) is defined over the probability space \( (G(X), \tau(x)) \) and \( d_K \) is metrises \( G(X) \). The next step is to adapt the Bayesian learning scheme which in the discrete case maps the prior \( D_X(Q) \) to the posterior \( D_X(Q+s) \), for \( Q \) in \( M^*(X) \) the current parameter, given \( s \) a multiset of observed values in \( X \) (seen as a counting measure). Via the projective limit construction, learning can be led at the level of behavioural approximants \cite{11} and a subsidiary goal is to understand how the two levels relate. The second goal consists of in extending the probabilistic Kantorovich metric to uncertain chains (of this specific type) and understand its evolution under learning. Until now we assumed that the state of the system is fully observable, but the above questions should be developed as well in a broader context where the state is only partially and noisily so.

**Acknowledgement.** We warmly thank Laurent Duffloux for his valuable insights, and the anonymous reviewers for their thorough work and helpful remarks.
References


296


Complete positivity and natural representation of quantum computations

Mathys Rennela

Institute for Computing and Information Sciences
Radboud University

Sam Staton

Department of Computer Science
Oxford University

Abstract

We propose a new ‘quantum domain theory’ in which Scott-continuous functions are replaced by Scott-continuous natural transformations. Completely positive maps are widely accepted as a model of first-order quantum computation. We begin by establishing a categorical characterization of completely positive maps as natural families of positive maps. We explore this categorical characterization by building various representations of quantum computation based on different structures: affine maps between cones of positive elements, morphisms of algebras of effects, and affine maps of convex sets of states. By focusing on convex dcpos, we develop a quantum domain theory and show that it supports some important constructions such as tensor products by quantum data, and lifting.

Introduction

This paper is about semantic models of quantum computation. In common with other approaches to programming language semantics, the general idea is to interpret a type $A$ as a space $[A]$ of observations about $A$. One interprets a computation $x : A \vdash t : B$, that produces something $t$ of type $B$ but depends on something $x$ of type $A$, as a predicate transformer $[B] \to [A]$, which maps a predicate on $B$ to its weakest precondition. (See e.g. [DP06, Ren13, Cho14].)

In more detail, one interprets a type $A$ as a C*-algebra of operators $[A]$, and the computations describe maps that are in particular positive: it is actually only the positive elements of the algebra that describe the observables, and these must be preserved. Moreover the maps should be completely positive. Informally this means that it makes sense to run the computation on a subsystem of a bigger system; for example, we could adjoin an extra qubit to the system and still run the computation. More formally it means that not only does the map $[t] : [B] \to [A]$
preserve positive elements, but also \( \text{id}_{[\text{qubit}] \otimes [r]} : [\text{qubit}] \otimes [B] \to [\text{qubit}] \otimes [A] \) preserves positive elements.

The first contribution of this paper (Section 2) is a technique for building representations of quantum computation in terms of completely positive maps. In the second half of the paper (Section 3) we demonstrate our technique by making some first steps in the development of a ‘quantum domain theory’.

**A technique for building representations**

Here, a representation is a full and faithful functor \( F : C \to R \), that is, a functor for which each function \( F_{A,B} : C(A, B) \to R(F(A), F(B)) \) is a bijection.

From a programming language perspective, where objects interpret types and morphisms interpret programs, a representation result gives two things. Firstly, it gives a way of interpreting types as different mathematical structures, which can be illuminating or convenient, while retaining essentially the same range of interpretable programs. Secondly, since \( R \) may be bigger than \( C \), it gives the chance to interpret more types without altering the interpretation of programs at existing types.

There are several existing representation results which allow us to understand and analyze quantum computations in terms of different structures, such as convex sets (e.g. [JWW15]), domains (e.g. [Ren13]), partial monoids and effect algebras (e.g. [Jac12]). However, many of these representation results are only valid for positive maps, and so they do not fully capture quantum computation. Our contribution is a general method for extending these results to completely positive maps. Roughly, the method allows us to convert a full and faithful functor

\[
(\text{positive maps}) \to R
\]

(where \( R \) is an arbitrary category) into a full and faithful functor

\[
(\text{completely positive maps}) \to [N, R]
\]

into a functor category, where \( N \) is a category whose objects are natural numbers.

**Towards a quantum domain theory**

In the second part of the paper we demonstrate our technique by making some first steps in the development of a ‘quantum domain theory’. The ultimate goal in this line of work is to analyze all kinds of quantum programming by solving domain equations involving qubits. For example, one should expect a solution to the equation

\[
A = (\text{qubit} \otimes A)_{\bot}
\]

which would be a type of infinite streams of qubits. In this paper we exhibit (for the first time) a domain theory that supports qubits and lifting.

In brief, we begin from the observation that taking states of a \( W^* \)-algebra yields a representation of positive maps in terms of affine maps between convex sets. We use this to build a representation

\[
(W^*\text{-algebras and completely positive maps}) \to [N, (\text{convex sets and affine maps})]
\]
We can now extend the representation with domain theoretic structure, by replacing convex sets with directed complete convex sets. Thus ‘quantum domains’ are defined to be functors

\[ \mathbb{N} \to (\text{convex dcpos and affine continuous maps}) \]

and quantum computations are interpreted as affine Scott-continuous natural transformations between quantum domains. We show that this class of quantum domains supports various constructions, including tensor with quantum data and lifting.

1 Preliminaries

1.1 Linear maps of C*-algebras

The basic idea of matrix mechanics is that the observables for a quantum system are elements of a C*-algebra. Recall that a (unital) C*-algebra is a vector space over the field of complex numbers that also has multiplication, a unit and an involution, satisfying associativity laws for multiplication, involution laws (e.g. \( x^{**} = x \), \((xy)^* = y^*x^*\), \((\alpha x)^* = \bar{\alpha}(x^*)\)) and such that the spectral radius provides a norm making it a Banach space.

A key source of examples of C*-algebras are the algebras \( M_k \) of \( k \times k \) complex matrices, with matrix addition and multiplication, and where involution is conjugate transpose. In particular the set \( M_1 = \mathbb{C} \) of complex numbers has a C*-algebra structure, and the \( 2 \times 2 \) matrices, \( M_2 \), contain the observables of qubits.

If \( A \) is a C*-algebra then the \( k \times k \) matrices valued in \( A \) also form a C*-algebra, \( M_k(A) \). For instance \( M_k(\mathbb{C}) = M_k \), and \( M_k(M_l) \cong M_{k \times l} \). Informally, we can think of the C*-algebra \( M_k(A) \) as representing \( k \) entangled copies of \( A \). This can be thought of as a kind of tensor product: as a vector space \( M_k(A) \) is a tensor product \( M_k(\mathbb{C}) \otimes A \). There are various ways to extend this to define a tensor product on arbitrary C*-algebras, but we will not need tensor products other than \( M_k(A) \) in this paper.

The ‘direct sum’ \( X \oplus Y \) of C*-algebras is given by the cartesian product of the underlying sets. It has the universal property of the categorical product. The C*-algebra \( \mathbb{C} \oplus \mathbb{C} \) represents classical bits.

An element \( x \in A \) is positive if it can be written in the form \( x = y^*y \) for \( y \in A \). We denote by \( A^+ \) the set of positive elements of a C*-algebra \( A \) and define the following partial order on the elements of \( A \): \( x \leq y \) if and only if \( (y-x) \in A^+ \).

We consider the following kinds of map of C*-algebras. Let \( f : A \to B \) be a linear map between the underlying vector spaces.

P The map \( f \) is positive if it preserves positive elements and therefore restricts to a function \( A^+ \to B^+ \). A positive map \( A \to \mathbb{C} \) will be called a state on \( A \).

U The map \( f \) is unital if it preserves the unit, i.e. \( f(1_A) = 1_B \);

SU The map \( f \) is sub-unital if \( f(1_A) \leq 1_B \);

CP The map \( f \) is completely positive if it is \( n \)-positive for every \( n \in \mathbb{N} \), i.e. the map \( M_n(f) : M_n(A) \to M_n(B) \) defined for every matrix \([x_{i,j}]_{i,j \leq n} \in M_n(A)\) by \( M_n(f)([x_{i,j}]_{i,j \leq n}) = [f(x_{i,j})]_{i,j \leq n} \) is positive for every \( n \in \mathbb{N} \).
As a matter of convenience, we will denote through this paper different classes of maps by the first letters of the names of the properties they follow. In particular, the term (C)P((S)U)-map will refer respectively to a (completely) positive (sub-) unital map. We write \( C^*\text{-Alg}_{CPSU} \) for the category of \( C^* \)-algebras and completely positive sub-unital maps, and so on.

We refer the interested reader to [Sak71,Tak02] for a complete introduction to \( C^* \)-algebras.

1.2 Representation of quantum computations

For the reader familiar with semantics of programming languages, we recall basic ideas for the semantics of quantum programming languages in \( C^* \)-algebras. A type \( A \) is interpreted as a \( C^* \)-algebra \([A] \). A terminating computation-in-context \( x_1 : A_1, \ldots, x_n : A_n \vdash t : B \) is interpreted as a CPU-map \( B \to \bigotimes_i A_i \), transforming observations about the result type to requisite observations about the free variables. We let \([qubit] = M_2\), and the empty tensor is \( C \), and so a computation \( \vdash t : qubit \) that generates a qubit with no free variables is interpreted as a CPU-map \( M_2 \to C \). In the theory of operator algebras, CPU-maps into \( C \) are called states.

1.3 Isometries and pure states

An important class of CP-maps comes from multiplication by matrices. Let \( A \) be a \( C^* \)-algebra. Any \( m \times n \) complex matrix \( F \) induces a CP-map \( F_*F : M_m(A) \to M_n(A) \) given by \((F_*F)(x) = F_*xF\), where \( F_* \) is the conjugate transpose of \( F \). This is a CPU-map if \( F \) is an isometry, i.e. \( F_*F = I \). In particular, putting \( n = 1 \) and \( A = C \), any vector \( v \in C^2 \) with \( v^*v = 1 \) induces a state \( v^*v : M_2 \to C \), called a ‘pure state’.

In what follows we will consider the vectors

\[
\begin{align*}
|0\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
|1\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
|+\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
|\rho\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}
\end{align*}
\]

which all induce CPU-maps \( M_2 \to C \). One often writes \( \langle v \) for the conjugate transpose \( |v\rangle^* \), so the induced CPU-map can be written \( \langle v | - | v \rangle : M_2(A) \to A \). In particular:

\[
\begin{align*}
\langle 0 | \begin{pmatrix} a & b \\ c & d \end{pmatrix} | 0 \rangle &= a \\
\langle + | \begin{pmatrix} a & b \\ c & d \end{pmatrix} | + \rangle &= \frac{1}{2} (a + b + c + d) \\
\langle 1 | \begin{pmatrix} a & b \\ c & d \end{pmatrix} | 1 \rangle &= d \\
\langle \rho | \begin{pmatrix} a & b \\ c & d \end{pmatrix} | \rho \rangle &= \frac{1}{2} (a - ib + ic + d)
\end{align*}
\]

2 Naturality and representations of complete positivity

In this section, we will provide a new categorical characterization of completely positive maps as natural families of positive maps (§2.1–2.3). This gives a technique for building representations of completely positive maps (§2.3), which we demonstrate with several examples: positive cones (§2.4–2.5), effects (§2.6), and states (§2.7).
2.1 Complete positivity as naturality

In Section 1.3 we considered how a matrix \( F \in \mathbb{C}^{m \times n} \) induces a completely positive map \( F^*: F : M_m \to M_n \). This construction is functorial. To make this precise, we introduce the category \( \mathbb{N}_{\text{Mat}} \) of complex matrices: the objects are non-zero natural numbers seen as dimensions, and the morphisms \( m \to n \) are \( m \times n \) complex matrices. Composition is matrix multiplication. (We remark that the category \( \mathbb{N}_{\text{Mat}} \) is equivalent to the category of finite-dimensional complex vector spaces and linear maps, since every finite-dimensional vector space is isomorphic to \( \mathbb{C}^n \). It is also equivalent to the category of finite-dimensional Hilbert spaces and linear maps, since every such space has a canonical inner product.)

The construction of matrices of elements of a C*-algebra can be made into a functor \( \mathbb{C}^*-\text{Alg}_{\text{CP}} \times \mathbb{N}_{\text{Mat}} \to \mathbb{C}^*-\text{Alg}_{\text{P}} \). It takes a C*-algebra \( A \) and a pair of morphisms \( (f,F) : (A,m) \to (B,n) \) to the positive map \( F^*(f_\ast)F : M_m(A) \to M_n(B) \).

We will consider this functor in curried form, \( M : \mathbb{C}^*-\text{Alg}_{\text{CP}} \to [\mathbb{N}_{\text{Mat}}, \mathbb{C}^*-\text{Alg}_{\text{P}}] \). It takes a C*-algebra \( A \) to a functor, i.e. an indexed family of C*-algebras, \( M(A) = \{M_n(A)\}_n \). A completely positive map \( f : A \to B \) is taken to the corresponding family of positive maps \( M(f) = \{M_n(f) : M_n(A) \to M_n(B)\}_n \). This gives our main result: the completely positive maps are in natural bijection with families of positive maps.

**Theorem 2.1** The functor \( M : \mathbb{C}^*-\text{Alg}_{\text{CP}} \to [\mathbb{N}_{\text{Mat}}, \mathbb{C}^*-\text{Alg}_{\text{P}}] \) is full and faithful.

The rest of this section is dedicated to the proof of Theorem 2.1. Faithfulness is obvious, since for any CP-map \( f : A \to B \) we have \( M(f)_1 = f \). To show fullness we begin with the following lemma.

**Lemma 2.2** Consider two positive maps \( f_2 : M_2(B) \to M_2(A) \) and \( f_1 : B \to A \) of C*-algebras. The following conditions are equivalent:

1. \( \forall y \in M_2(B), v \in \mathbb{C}^2. \ v^*(f_2(y))v = f_1(v^*yv) \)
2. \( f_2 = M_2(f_1) \).

**Proof.** We can show that (ii) implies (i) with the following argument:

For every 2-by-2 matrix \( \begin{bmatrix} a & b \\ c & d \end{bmatrix} \in M_2(B) \):

\[
v^*(M_2(f_1) \begin{bmatrix} a & b \\ c & d \end{bmatrix})v = v^* \begin{bmatrix} f_1(a) & f_1(b) \\ f_1(c) & f_1(d) \end{bmatrix} v = f_1(v^* \begin{bmatrix} a & b \\ c & d \end{bmatrix} v)
\]

since \( v^*v \) maps a 2-by-2 matrix to a linear combination of its entries, which will be preserved by the linear map \( f_1 \).

We will now focus on the proof that (i) \( \implies \) (ii).

Consider \( \begin{bmatrix} a & b \\ c & d \end{bmatrix} \in M_2(B) \), let \( \begin{bmatrix} a' & b' \\ c' & d' \end{bmatrix} = f_2 \begin{bmatrix} a & b \\ c & d \end{bmatrix} \) and suppose that (i) holds.
We use the assumption (i) with the vectors $|0\rangle$, $|1\rangle$, $|+\rangle$ and $|\rho\rangle$, to obtain $a' = f_1(a)$, $d' = f_1(d)$, $a' + b' + c' + d' = f_1(a) + f_1(b) + f_1(c) + f_1(d)$ and $a' - i b' + i c' + d = f_1(a) - i f_1(b) + i f_1(c) + f_1(d)$. We can combine these four facts to also deduce that $b' = f_1(b)$ and $c' = f_1(c)$.

And thus finally, we observe that $f_2 \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} f_1(a) & f_1(b) \\ f_1(c) & f_1(d) \end{bmatrix} = M_2(f_1) \begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

This concludes our proof of Lemma 2.2.

We use Lemma 2.2 to establish fullness in our proof of Theorem 2.1. Consider a subcategory $M$ of $V$. Then define $F$ with the same objects but where the morphisms are isometries ($N = M(1)$). The proof of Theorem 2.1 is quite flexible and can accommodate some variation in the index category and the base category.

For a first variation, we change the index category so that we can focus on unit-preserving completely positive maps. We consider the subcategory $N_{\text{isom}}$ of $N_{\text{Mat}}$ with the same objects but where the morphisms are isometries ($F^*F = I$).

We will be quite general about the base category. Consider a subcategory $V$ of $\text{C}^*\text{-Alg}_p$ that is closed under matrix algebras, i.e.

$$\mathbb{C} \in V \quad \text{and} \quad A \in V \implies M_n(A) \in V.$$  

Then define $V_C$ to be the closure of $V$ under matrices of morphisms: the objects of $V_C$ are the same as the objects of $V$, and a function $f : A \to B$ is in $V_C$ if $M_n(f) : M_n(A) \to M_n(B)$ is in $V$ for all $n$. For instance, $(\text{C}^*\text{-Alg}_p)_C = \text{C}^*\text{-Alg}_{\text{CP}}$.

**Theorem 2.3** Consider a subcategory $V$ of $\text{C}^*\text{-Alg}_p$ satisfying (1) and such that
the matrices functor
\[ V \times N_{\text{Isom}} \to \text{C}^*\text{-Alg}_P \]
factors through \( V \). It induces a full and faithful functor \( V \to [N_{\text{Isom}}, V] \).

There are other variations on the result, by changing the index category to a different subcategory of \( \text{C}^*\text{-Alg}_{CP} \). We focus on two examples which are particularly relevant in the enriched setting (see §2.4):

- Let \( N_{CP} \) be the category whose objects are natural numbers and where a morphism \( m \to n \) is a completely positive map \( M_m \to M_n \). In the literature, this category is often called CPMs [MSS13,Cho14], W [Sel04] or CPM[FdHilb] [Sel05].
- Let \( N_{CPU} \) be the category whose objects are natural numbers and where a morphism \( m \to n \) is a completely positive unital map \( M_m \to M_n \). The dual of this category can be thought of as comprising the trace-preserving completely positive maps between density matrices (e.g. \( Q_\cdot \) in [MSS13, Def. 2.9]).

The matrices functors
\[ \text{C}^*\text{-Alg}_{CP} \times N_{\text{Mat}} \to \text{C}^*\text{-Alg}_P \quad \text{C}^*\text{-Alg}_{CPU} \times N_{\text{Isom}} \to \text{C}^*\text{-Alg}_{PU} \]
extend to functors
\[ \text{C}^*\text{-Alg}_{CP} \times N_{CP} \to \text{C}^*\text{-Alg}_P \quad \text{C}^*\text{-Alg}_{CPU} \times N_{CPU} \to \text{C}^*\text{-Alg}_{PU} \]
using the idea that \( M_n(A) = M_n \otimes A \), and if \( f : M_m \to M_n \) is completely positive then so too is \( f \otimes A : M_m(A) \to M_n(A) \).

**Theorem 2.4** Consider a subcategory \( V \) of \( \text{C}^*\text{-Alg}_{CP} \) that is closed under matrix algebras (1) and such that the matrices functor
\[ V \times N_{CP(U)} \to \text{C}^*\text{-Alg}_P \]
factors through \( V \). It induces a full and faithful functor \( V \to [N_{CP(U)}, V] \).

### 2.3 Representations of quantum computation

Our intention is to use Theorem 2.1 to build representation results for completely positive maps out of representation results for positive maps. For instance, the following corollary is immediate:

**Corollary 2.5** Every full and faithful functor \( F : \text{C}^*\text{-Alg}_P \to \mathbb{R} \) induces a full and faithful functor \( \text{C}^*\text{-Alg}_{CP} \to [N_{\text{Mat}}, \text{C}^*\text{-Alg}_P] \to [N_{\text{Mat}}, \mathbb{R}] \).

For the remainder of this section we illustrate our technique by building representation theorems for CP-maps.

### 2.4 Example: positive cones of C*-algebras

We show how to build a representation for CP-maps out of affine maps between cones. We begin by recalling basic definitions. For any \( i \leq m \) let \( \delta^{i,m}_m \) be the Kronecker vector with \( \delta^{i,m}_i = 1 \) and \( \delta^{i,m}_j = 0 \) for \( i \neq j \).
Definition 2.6 A cone is a set $X$ together with an $m$-ary function $(\vec{r})_X : X^m \to X$ for each vector $\vec{r} = (r_1 \ldots r_m)$ of non-negative real numbers, often written infix as $\sum_i r_i x_i$, such that for each $i$, $\delta^{i,m}_{i,m}(x_1 \ldots x_m) = x_i$, and for each $m \times n$ matrix $(s_{i,j})_{i,j}$ of non-negative real numbers, $\sum_i r_i (\sum_j (s_{i,j} x_j)) = \sum_j ((\sum_i (r_i s_{i,j}))) x_j$.

A homomorphism of cones is a function that preserves the algebraic structure. Homomorphisms are often called affine maps. The category Cone is the category of cones together with affine maps.

There are different ways to formulate this definition. A subset of a real vector space forms a cone if it is closed under addition and multiplication by positive real scalars, and conversely every cone arises in this way. This motivates the terminology ‘affine map’.

Alternatively, the abstract definition of cones can be reformulated in terms of scalar multiplication and binary addition, and all the $m$-ary operations can be built from these operations.

Representations

For any C*-algebra the positive elements form a cone.

Lemma 2.7 Taking the cone of positive elements yields a full and faithful functor $(-)^+ : C^*-\text{Alg}_P \to \text{Cone}$.

Proof. [notes] Any positive map $f : X \to Y$ is completely defined by its action on $X^+$: an arbitrary element $x \in X$ can be written uniquely as linear sum of four positive elements $x = x_1 + ix_2 - x_3 - ix_4$, for $x_i$ all positive [FJ13, Lemma 2.2], determining $f(x)$. □

Example 2.8 The functor $C^*-\text{Alg}_{CP} \to [\mathbb{N}_{\text{Mat}}, \text{Cone}]$ taking $A$ to $n \mapsto (M_n(A))^+$ is full and faithful.

This appears to be a new categorical way to formulate the theory of matrix ordered spaces (e.g. [Pau03, Ch. 13], [ER00]).

Enrichment

Example 2.9 The functor $C^*-\text{Alg}_{CP} \to [\mathbb{N}_{\text{CP}}, \text{Cone}]$ taking $A$ to $n \mapsto (M_n(A))^+$ is full and faithful.

The category $\mathbb{N}_{\text{CP}}$ is enriched in Cone: one can scale completely positive maps and add them too. This leads us to focus on locally affine functors $F : \mathbb{N}_{\text{CP}} \to \text{Cone}$, i.e., functors that preserve the cone structure of the hom-sets, i.e., enriched presheaves [Day70].

The category of locally affine functors $\mathbb{N}_{\text{CP}} \to \text{Cone}$ is the free colimit completion of $\mathbb{N}_{\text{CP}}$ as a Cone-enriched category. This draws a comparison with other models of quantum computation, which partly inspired the current work. Firstly there are models based around biproduct completions of $\mathbb{N}_{\text{CP}}$ (e.g. [Sel04] and [PVS14]): this is relevant since a biproduct completion is a free coproduct completion of $\mathbb{N}_{\text{CP}}$ as a Cone-enriched category. Secondly there are models based around (non-enriched) colimit completions of categories such as $\mathbb{N}_{\text{CPU}}$ [MSS13].
2.5 Example: Directed complete cones and $W^*$-algebras

Directed-completeness

Recall that a directed complete partial order is a partial order in which every directed set has a least upper bound. A bounded dcpo (bdcpo) is a partial order in which every directed set that has an upper bound has a least upper bound.

Definition 2.10 [e.g. [KP09]] A conic bdcpo (or d-cone) is a cone $X$ (Def. 2.6) equipped with a bdcpo structure such that the operations $(\vec{r})_X : X^m \to X$ are all Scott-continuous functions from the product bdcpo. This yields a category $d\text{Cone}$ of conic bdcpo and affine Scott-continuous maps between them.

Definition 2.11 A $C^*$-algebra $A$ is called monotone complete if the cone $A^+$ of positive elements is a conic bdcpo. A positive map between monotone complete $C^*$-algebras $A \to B$ is called normal if its restriction to the positive cone preserves joins of bounded directed sets.

We will focus on $W^*$-algebras, which are monotone complete $C^*$-algebras such that for every non-zero positive element $x \in A^+$ there is a normal positive map $f : A \to \mathbb{C}$ such that $f(x) \neq 0$ (e.g. [Tak02, III.3.16]). $W^*$-algebras encompass all finite dimensional $C^*$-algebras, and also the algebras of bounded operators on any Hilbert spaces, the function space $L^\infty(X)$ for some standard measure space $X$, and the space $ell^\infty(\mathbb{N})$ of bounded sequences.

We write $W^*-\text{Alg}_P$ for the category of $W^*$-algebras and normal positive maps, and $W^*-\text{Alg}_{CP}$ for the category of $W^*$-algebras and normal completely positive maps, and so on. Essentially by definition we have a full and faithful functor $(-)^+ : W^*-\text{Alg}_P \to d\text{Cone}$. In consequence:

Example 2.12 The functor $W^*-\text{Alg}_{CP} \to [\mathbb{N}_{\text{Mat}}, d\text{Cone}]$ taking $A$ to $n \mapsto (M_n(A))^+$ is full and faithful.

2.6 Examples: Effects

We briefly discuss examples based on the theory of effects of $C^*$-algebras, although we will not elaborate on this any further in this article.

An effect of a $C^*$-algebra is a positive element that is less than 1. Informally, an effect is a kind of ‘unsharp’ predicate. The effects $[0,1]_A$ of a $C^*$-algebra $A$ form an algebraic structure called an ‘effect module’: they have a partial monoid structure given by addition, a top element, and they admit multiplication by scalars in the unit interval $[0,1]$.

Taking effects actually yields a full and faithful functor $C^*-\text{Alg}_{PU} \to \text{EMod}$ (see e.g. [FJ13]), giving us another illustration of our framework:

Example 2.13 The functor $C^*-\text{Alg}_{CPU} \to [\mathbb{N}_{\text{Isom}}, \text{EMod}]$, taking $A$ to $n \mapsto [0,1]_{M_n(A)}$, is full and faithful.

There are some interesting variations on this example.

Example 2.14 • A ‘generalized effect module’ is an effect module without a top element. By ignoring the top element we obtain a full and faithful fun-
tor $\mathbf{C^*-Alg}_{\text{PSU}} \rightarrow \text{GEMod}$ [FJ13] and hence a full and faithful functor $\mathbf{C^*-Alg}_{\text{CPSU}} \rightarrow [\mathbb{N}_{\text{Isom}}, \text{GEMod}]$.

- In a $W^*$-algebra, the effects form a directed complete effect module; this gives a full and faithful functor $W^* - \mathbf{Alg}_{\text{PU}} \rightarrow \text{dEMod}$ [Ren13] and hence we obtain a new full and faithful functor $W^* - \mathbf{Alg}_{\text{CPU}} \rightarrow [\mathbb{N}_{\text{Isom}}, \text{dEMod}]$.

- Similarly, from a full and faithful functor $W^* - \mathbf{Alg}_{\text{PSU}} \rightarrow \text{dGEMod}$ [Ren13] we obtain a full and faithful functor $W^* - \mathbf{Alg}_{\text{CPSU}} \rightarrow [\mathbb{N}_{\text{Isom}}, \text{dGEMod}]$.

2.7 Examples: States

Convex sets

Definition 2.15 A convex set is a set $X$ together with an $m$-ary function $(\vec{r})_X : X^m \rightarrow X$ for each vector $\vec{r} = (r_1 \ldots r_m)$ of non-negative real numbers with $\sum r_i = 1$, such that for each $i$, $\delta^m_X(x_1, \ldots, x_m) = x_i$, and for each $m \times n$ matrix $(s_{i,j})_{i,j}$ of non-negative real numbers such that $\sum_j s_{i,j} = 1$, we have $\sum_i r_i (\sum_j (s_{i,j} \cdot x_j)) = \sum_j (\sum_i (r_i s_{i,j})) \cdot x_j$.

A homomorphism of convex sets is a function that preserves the algebraic structure. Homomorphisms are often called affine maps.

As with cones, there are different ways to formulate this definition. A subset of a real vector space is convex if it is closed under convex sums, and conversely every convex set arises in this way. The definition of convex sets can be reformulated in terms of a weighted binary addition (e.g. [Kei09]).

For a $W^*$-algebra $A$, consider the normal state space $\mathcal{NS}(A) = W^* - \mathbf{Alg}_{\text{PU}}(A, \mathbb{C})$. The hom-sets of the category $W^* - \mathbf{Alg}_{\text{PU}}$ can be given a convex structure, considered as a subset of the vector space of all linear maps. The mapping $\mathcal{NS}(\cdot)$ can thus be turned into a contravariant functor to the category of convex sets, which acts as follows on positive unital maps: $\mathcal{NS}(A \xrightarrow{f} B) = (\cdot) \circ f : \mathcal{NS}(B) \rightarrow \mathcal{NS}(A)$.

Theorem 2.16 ([Sak71],[AS01],[Fur15]) The functor $\mathcal{NS}(\cdot) : W^* - \mathbf{Alg}_{\text{PU}}^{\text{op}} \rightarrow \text{Conv}$ is full and faithful.

(The normal states functor is not faithful when restricted to completely positive maps ($W^* - \mathbf{Alg}_{\text{CPU}}^{\text{op}}$): the transpose map is positive but not completely positive, and it yields an isomorphism of convex sets.)

Example 2.17 The functor $W^* - \mathbf{Alg}_{\text{CPU}}^{\text{op}} \rightarrow [\mathbb{N}_{\text{Isom}}^{\text{op}}, \text{Conv}]$, taking $A$ to $n \mapsto \mathcal{NS}(M_n(A))$, is full and faithful.

Example 2.18 The functor $W^* - \mathbf{Alg}_{\text{CPU}}^{\text{op}} \rightarrow [\mathbb{N}_{\text{CPU}}^{\text{op}}, \text{Conv}]$, taking $A$ to $n \mapsto \mathcal{NS}(M_n(A))$, is full and faithful.

3 Quantum domain theory

In this section, we will use the techniques in Section 2 to begin to build a ‘quantum domain theory’: a new categorical model for quantum computations based on order-valued functors.
We proceed by analogy with classical domain theory. Recall that in classical domain theory there are two categories that play important roles: firstly a category Predom of dcpos and Scott-continuous functions, and secondly a category Dom of pointed dcpos (dcpos with a bottom element) and strict Scott-continuous functions (functions that preserve the bottom element). Lifting (freely adding a bottom element) is left adjoint to the evident forgetful functor (e.g. [AJ94]).

3.1 Preliminaries on convex dcpos

**Definition 3.1** [e.g. [JP89, §9]] A **convex dcpo** is a convex set (Def. 2.15) equipped with a dcpo structure such that the functions that constitute its convex structure are Scott-continuous. This yields a category dConv of convex dcpos and affine Scott-continuous maps between them.

A simple example of a convex dcpo is the unit interval of the reals.

3.1.1 Sums of convex dcpos

Recall that the sum of two convex sets, $A$ and $B$, can be described as the set $A \uplus B \uplus (A \times B \times (0, 1))$, where $(0, 1)$ is the open unit interval. Its elements either come directly from $A$, or from $B$, or are a non-trivial formal convex combination of elements from $A$ and $B$. With a slightly informal notation, we write $(a, -, 0)$ instead of $a$, and $(-, b, 1)$ instead of $b$. Then define the convex structure as follows

$$
\sum_i r_i (a_i, b_i, \lambda_i) \overset{\text{def}}{=} \left( \sum_i r_i \frac{1 - \lambda_i}{1 - \sum_i r_i \lambda_i}, a_i, \sum_i r_i \lambda_i b_i, (\sum_i r_i \lambda_i) \right)
$$

taking the obvious convention where $(\sum_i r_i \lambda_i)$ is 0 or 1. This has the universal property of the coproduct in the category of convex sets.

3.1.2 Skew sums

There is a variation on the sum that will be useful in what follows. To motivate, observe that if $A$ and $B$ are partial orders then we can form a new partial order $A \uplus_+ B$ whose carrier is $A \uplus B$ but with the partial order generated by $a \leq_{A \uplus_+ B} a'$ whenever $a \leq_{A} a'$, and $b \leq_{A \uplus B} b'$ whenever $b \leq_{B} b'$, and $a \leq_{A \uplus_+ B} b$ whenever $a \in A$ and $b \in B$. We call this the skew sum. It gives a universal square

$$
\begin{array}{c}
A \times B \\
\downarrow \quad \downarrow \\
A \uplus_+ B
\end{array}
$$

If $A$ and $B$ are convex dcpos then we define a skew sum $A \uplus_+ B$ as the coproduct of convex sets, but with the partial order $(a, b, \lambda) \leq (a', b', \mu)$ if $a \leq_{A} a'$ and $b \leq_{B} b'$ and $\lambda \leq \mu$. This has a universal property like a coproduct except with an additional requirement that $a \leq b$ for $a \in A, b \in B$.

For example, we can freely add a bottom element to a convex dcpo $A$ by taking the skew sum $(1 \uplus A)$. 

308
3.2 Abstract definitions of quantum domains

Our definition of quantum domain is inspired by Example 2.18. Recall that \( \mathbb{N}_{\text{CPU}} \) is the category of natural numbers and where a morphism \( m \to n \) is a CPU-map (§ 2.3). A functor \( D : \mathbb{N}_{\text{CPU}}^{\text{op}} \to \text{dConv} \) is ‘locally affine’ if \( D \) preserves the convex structure of morphisms, i.e. \( D(r.f + s.g) = r.D(f) + s.D(g) \) whenever \( r + s = 1 \).

**Definition 3.2** A quantum predomain is a locally affine functor \( D : \mathbb{N}_{\text{CPU}}^{\text{op}} \to \text{dConv} \). A quantum domain is a locally affine functor \( D : \mathbb{N}_{\text{CPU}}^{\text{op}} \to \text{dConv} \) such that the convex dcpo \( D(1) \) has a least element.

A morphism of quantum predomains, which will be called a QD-map, is a natural transformation \( \phi : D \Rightarrow E \) between quantum domains, i.e. a family of continuous affine maps \( \{ \phi_n : D(n) \to E(n) \}_{n \in \mathbb{N}} \) such that, for every map \( f : n \to m \) in \( \mathbb{N}_{\text{CPU}} \), the following diagram commutes:

\[
\begin{array}{ccc}
D(n) & \xrightarrow{\phi_n} & E(n) \\
\downarrow f & & \downarrow E(f) \\
D(m) & \xrightarrow{\phi_m} & E(m)
\end{array}
\]

If \( D \) is a quantum domain, i.e. \( D(1) \) has a least element, then we say that a QD-map \( \phi : D \to E \) is strict if \( \phi_1(\bot_{D(1)}) \) is a least element in \( E(1) \).

We define \( \text{QDom} \) to be the full subcategory of \( \text{QPredom} \) comprising quantum domains, and \( \text{QDom} \) to be the subcategory of \( \text{QPredom} \) comprising quantum domains and strict QD-maps. Those three categories are enriched over the category \( \text{Dcpo} \) of dcpos together with Scott-continuous maps.

For a motivating example, recall (Ex. 2.18) that every W*-algebra \( A \) induces a functor \( \mathcal{N}S(A) : \mathbb{N}_{\text{CPU}}^{\text{op}} \to \text{Conv} \) with \( \mathcal{N}S(A)(n) = \text{W}^*-\text{Alg}_{\text{CPU}}(M_n(A), \mathbb{C}) \). This can be understood as a quantum predomain, where each \( \mathcal{N}S(A)(n) \) is considered with a discrete order. In particular \( \mathcal{N}S(\mathbb{C}) \cong \mathbb{N}_{\text{CPU}}(-, 1) \). This gives an embedding \( \mathcal{N}S : \text{W}^*-\text{Alg}_{\text{CPU}}^{\text{op}} \to \text{QPredom} \).

3.3 Construction of quantum domains

3.3.1 Sums of quantum predomains

The coproduct of quantum predomains is defined pointwise. We define the sum of two quantum predomains \( D \) and \( E \) pointwise: let \( (D + E)(n) = D(n) + E(n) \) for \( n \in \mathbb{N} \). This has the universal property of the coproduct in \( \text{QPredom} \).

The embedding \( \text{W}^*-\text{Alg}_{\text{CPU}}^{\text{op}} \to \text{QPredom} \) preserves sums. This follows from two facts: first, \( \text{W}^*-\text{Alg}_{\text{CPU}}^{\text{op}}(A \oplus B, \mathbb{C}) \cong \text{W}^*-\text{Alg}_{\text{CPU}}^{\text{op}}(A, \mathbb{C}) + \text{W}^*-\text{Alg}_{\text{CPU}}^{\text{op}}(B, \mathbb{C}) \) (e.g. [JWW15, Prop. 16]) and second, \( M_n(A \oplus B) \cong M_n(A) \oplus M_n(B) \), for all W*-algebras \( A \) and \( B \).

3.3.2 Tensor with quantum data, aka copower

For every quantum predomain \( D \), one can define a quantum predomain \( (n \odot D) \) by \( (n \odot D)(m) = D(nm) \) for every natural number \( n \in \mathbb{N} \) and \( (n \odot D)(f) = D(M_n(f)) \)
for \( f : m \to p \) in \( \mathcal{N}_{\text{CPU}}^{\text{op}} \). In particular \((\mathbb{2} \odot \mathcal{D})\) can be thought of informally as a predomain of entangled pairs \((x,d)\) where \( x \) is a qubit and \( d \) is from \( \mathcal{D} \).

We can build a functor
\[
\mathbb{1} : \mathcal{N}_{\text{CPU}}^{\text{op}} \times \text{QPredom} \to \text{QPredom}.
\]
This has the universal property of ‘copower by representables’ (see e.g. [JK01]).

### 3.3.3 Lifting via skew sums

We can also define a pointwise skew-sum of quantum predomains. For quantum predomains \( \mathcal{D} \) and \( \mathcal{E} \), as a quantum predomain \( \mathcal{D} + \mathcal{E} \) with \((\mathcal{D} + \mathcal{E})(n) = \mathcal{D}(n) + E(n)\) for \( n \in \mathbb{N} \). This has the universal property of the skew coproduct (§3.1.2) in \( \text{QPredom} \).

We use the skew coproduct to define a way of lifting quantum predomains to quantum domains. Let \( \mathcal{D} \perp = \mathcal{NS}(\mathbb{C}) \oint \mathcal{D} \). In more detail, \( \mathcal{D} \perp(n) = W^\ast\text{-Alg}_{\text{CPU}}(M_n, \mathbb{C}) \oint \mathcal{D}(n) \). Since \((\mathcal{NS}(\mathbb{C}))(1) = 1\) we know that \( \mathcal{D} \perp(1) \) has a least element for any quantum predomain \( \mathcal{D} \).

**Proposition 3.3** The construction \((-) \perp : \text{QPredom} \to \text{QDom}!\) is left adjoint to the forgetful functor \( \text{QDom}! \to \text{QPredom} \).

Moreover the adjunction is \( \text{Dcpo} \)-enriched.

**Proof.** Consider \( \delta : \mathcal{D} \Rightarrow \mathcal{E} \) and \( \eta : \mathcal{D} \perp \Rightarrow \mathcal{E} \).

Firstly, one can define a strict QD-map \( \delta^\ast : \mathcal{D} \perp \Rightarrow \mathcal{E} \) as a natural family of maps \( \delta^\ast(n) : \mathcal{D} \perp(n) = W^\ast\text{-Alg}_{\text{CPU}}(M_n, \mathbb{C}) \oint \mathcal{D}(n) \to \mathcal{E}(n) \), where \( \delta^\ast(n) : (\varphi, x, \lambda) \mapsto (\varphi, \delta(n)(x), \lambda) \) for every \( n \in \mathbb{N} \).

Secondly, one can define a QD-map \( \eta^\ast : \mathcal{D} \Rightarrow \mathcal{E} \) as a natural family of maps \( \eta^\ast(n) : \mathcal{D}(n) \to \mathcal{E}(n) \), where \( \eta^\ast(n) : x \mapsto \eta(-, x, 1) \) for every \( n \in \mathbb{N} \).

The Scott-continuous maps of hom-sets, \((-) \perp^\ast : \text{QPredom}(\mathcal{D}, U(\mathcal{E})) \to \text{QDom}_!(\mathcal{D} \perp, \mathcal{E})\) and \((-) \perp^\ast : \text{QDom}_!(\mathcal{D} \perp, \mathcal{E}) \to \text{QPredom}(\mathcal{D} \perp, U(\mathcal{E}))\), are inverse of each other. \( \square \)

### 3.4 Relationship with the L"owner order and earlier work

We conclude by relating these steps in quantum domain theory with earlier work on using operator algebra to model quantum computation.

To make an analogy, we recall the basic adjunction between the category \( \text{Set} \) of sets and functions and the category \( \text{Pfn} \) of sets and partial functions.

\[
\text{Set} \xrightarrow{\text{identity on objects}} \text{Pfn}
\]

Partial functions can be thought of as first-order computations, and indeed each hom-set forms a dcpo. However, the adjunction is not enriched in dcpos. Thus,
although there is a notion of lifting it does not properly capture partiality. The set $A + 1$ captures the notion of ‘programs that either return something of type $A$ or diverge’, but the order structure associated with partiality is not captured in this set. To remedy this, one embeds this adjunction into one involving domains

\[
\begin{array}{ccc}
\text{Set} & \xrightarrow{\text{identity on objects}} & \text{Pfn} \\
\text{Predom} & \xrightarrow{\downarrow \text{flat predomain}} & \text{Dom}_{\text{f}} \\
\text{Predom} & \xrightarrow{\downarrow \text{forgetful}} & \text{Dom}_{\text{f}} \\
\end{array}
\]

where the lower adjunction and the right-hand embedding are dcpo-enriched.

Now, we also have an adjunction between $W^{*-}\text{Alg}_{\text{CPSU}}$ and $W^{*-}\text{Alg}_{\text{CPU}}$:

\[
\begin{array}{ccc}
W^{*-}\text{Alg}_{\text{CPU}} & \xleftarrow{\text{identity on objects}} & W^{*-}\text{Alg}_{\text{CPSU}} \\
\mathcal{NS} & \xrightarrow{\downarrow \text{NS}} & \mathcal{QDom}_{\text{f}} \\
\mathcal{QPredom} & \xrightarrow{\downarrow \text{forgetful}} & \mathcal{QDom}_{\text{f}} \\
\end{array}
\]

which has been proposed for studying quantum computation. The hom-sets of $W^{*-}\text{Alg}_{\text{CPSU}}$ are dcpos, under the Löwner order: $f \leq g$ if $g - f$ is completely positive. Again, however, the adjunction is not dcpo-enriched. This time, the $W^{*-}$ algebra $A \oplus \mathbb{C}$ captures the notion of ‘programs that either return something of type $A$ or diverge’, but again the order structure associated with partiality is not captured in this algebra. Our proposal for quantum domains resolves this since there is an embedding

where the right-hand embedding is dcpo-enriched and so is the lower adjunction.

### 3.5 Next steps for quantum domain theory

We have demonstrated that our representation techniques can be used to build a quantum domain theory that supports lifting and tensor products by quantum data. We conclude by mentioning some next steps in this direction.

Each quantum predomain of the form $\mathcal{NS}(A)$ has some extra structure. For example, the block diagonal function $A \oplus A \rightarrow M_2(A)$ between $W^{*-}$-algebras gives a QD-map $2 \circ \mathcal{NS}(A) \rightarrow \mathcal{NS}(A) \oplus \mathcal{NS}(A)$. This can be thought of as measuring a qubit in the standard basis, returning a classical bit. There is also unique CPU-map $\mathbb{C} \rightarrow M_2(A)$, which means that there is a unique QD-map $\mathcal{NS}(A) \rightarrow \mathcal{NS}(\mathbb{C})$. This algebraic structure has been axiomatized in [Sta15]. It seems likely that it would be helpful to require this structure on all quantum domains.
The order on a quantum domain will allow us to understand recursion at higher types, but we would also expect a quantum domain to have a physical realization. One candidate is this: a ‘realizable quantum domain’ is a quantum domain $D$ together with a Hilbert space $H$ and a natural epimorphism $\mathcal{N}S(B(H)) \to D$.

In earlier work [Ren14] the first author has shown that the category $W^*\text{-Alg}_{\text{CPSU}}$ is algebraically complete, and so supports the solution of recursive domain equations. An important next step is to investigate whether this result extends to quantum domains.

It would also be interesting to investigate connections to other models of higher-order quantum computation [HH11].

Summary

In the first part of the paper, we have characterized the notion of complete positivity in a natural way. This abstract setting can be used as a way of extending the logical and semantical properties of positive maps to completely positive maps.

In the other half of the paper, we have shown that $W^*$-algebras are order-valued presheaves on a category of qubits, and normal completely positive maps are natural transformations between $W^*$-algebras, seen as order-valued presheaves. We have exposed some of the categorical properties of those presheaves. We argue that our presheaves are a suitable generalization of $W^*$-algebras when it comes to denotational semantics of quantum programs.

Acknowledgments

We would like to thank Kenta Cho, Robert Furber, Tobias Fritz, Bart Jacobs and Phil Scott for helpful discussions, and the anonymous referees for their suggestions. This research has been financially supported by the European Research Council (ERC) under the QCLS grant (Quantum Computation, Logic & Security) and a Royal Society University Research Fellowship.

References

Rennela, Staton


Focused Linear Logic and the $\lambda$-calculus

Taus Brock-Nannestad$^{1,3}$

\textit{INRIA Saclay \& LIX, École Polytechnique}

Nicolas Guenot$^{2,4}$

\textit{IT University of Copenhagen}

Abstract

Linear logic enjoys strong symmetries inherited from classical logic while providing a constructive framework comparable to intuitionistic logic. However, the computational interpretation of sequent calculus presentations of linear logic remains problematic, mostly because of the many rule permutations allowed in the sequent calculus. We address this problem by providing a simple interpretation of focused proofs, a complete subclass of linear sequent proofs known to have a much stronger structure than the standard sequent calculus for linear logic. Despite the classical setting, the interpretation relates proofs to a refined linear $\lambda$-calculus, and we investigate its properties and relation to other calculi, such as the usual $\lambda$-calculus, the $\lambda\mu$-calculus, and their variants based on sequent calculi.

Keywords: Linear Logic, Focusing, Lambda-calculus, Curry-Howard Correspondence

1 Introduction

The idea of “proofs as programs” found in the original Curry-Howard correspondence between intuitionistic natural deduction and the simply-typed $\lambda$-calculus has proved to be a powerful narrative, leading to the development of functional programming, with its expressive type systems and strong guarantees on the behaviour of programs. A number of variants and extensions have been proposed, such as the $\lambda\mu$-calculus [20], based on classical logic. In the midst of many developments around the computational interpretation of logical systems, linear logic [11] has taken an important place in this field by providing a refined view of logics and their computational meaning. However, it is mostly seen as a tool, the instrument of a methodology [3,14], rather than the object of a computational interpretation, at least in its standard form — its intuitionistic variant has been used to describe a linear $\lambda$-calculus [7]. The problem

---

1 Email: taus.brock-nannestad@inria.fr
2 Email: ngue@itu.dk
3 Supported by the ERC Advanced Grant ProofCert.
4 Supported by grant 10-092309 from the Danish Council for Strategic Research to the Demtech project.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
lies in the framework of the sequent calculus in which it is described, which has a structure too lax to be conveniently represented through any kind of λ-terms. This fundamental problem has been tackled from the beginning using the formalism of proof-nets, described as natural deduction for linear logic [11], but this is a radical departure from traditional proof syntaxes, implying a use of graphs as representation, one way or another, to write programs. Keeping the standard syntax of proof trees requires changing the kind of programs considered, and interpretations based on pattern-matching [22] or processes [1,6,23] have been proposed.

In another setting, focusing [4] has been developed to improve proof search in linear logic by identifying a complete subset of proofs endowed with a strong structure. As this normal form was studied in details, it became apparent that the notion of polarity and the associated permutability properties leading to focused systems was an essential aspect of linear proof theory [15]. However, the focused sequent calculus for linear logic, just as its unfocused variant, has not appeared as a framework of choice for a direct computational interpretation in the Curry-Howard tradition. In this paper, we propose a simple interpretation of the most standard focused presentation of MELL, showing how the structure of focused proofs can be described by refined linear λ-terms as found in an intuitionistic setting, despite the “classical” nature of this logic — and in particular, despite the presence of the ⊗ connective.

The key to this interpretation is the use of an explicitly polarised syntax, where the negative formulas type computations while positive formulas type values, as done in the call-by-push-value framework [17]. Moreover, we consider a strongly focused system, where inversion is performed maximally and in an ordered fashion, thus yielding normal forms where no two inference rule instances can be permuted, although entire focused phases can still be permuted. The extracted calculus, that we call λπ, has no explicit control operator, but its type system allows the encoding of calculi with control, such as λµ, through a relatively simple translation. We present in Section 2 our focused proof system for MELL along with a term assignment, and discuss its basic properties.

The two fundamental results, cut elimination and focalisation, are proved in Section 3, and we discuss the computational interpretation of these two theorems, when viewing their proofs as transformations of terms in the λπ-calculus. The reduction of cuts, done in big steps, corresponds to the expected notion of reduction, based on substitution and the decomposition of pairs. Focusing corresponds to a reorganisation of terms that simplifies the structure of a term by rearranging the position of its subterms, merging values previously kept separated by unrelated phases of computation.

Finally, we discuss in Section 4 the expressivity of the λπ-calculus by considering fragments and encodings of known calculi into these fragments. Of particular interest are the sequent calculus variants of the simply-typed λ-calculus and of the λµ-calculus, which are closely related to their originals stemming from natural deduction. Note that λπ has a rich structure: it is a sequent-based variant of λµ with a notion of linearity, in which terms can be applied to trees rather than just lists of arguments. We conclude in Section 5 and discuss further investigations, from generalisations to richer logics to practical applicability.
Related work. As mentioned, the computational meaning of the standard system for “classical” linear logic has not been investigated as much as the interpretations given for intuitionistic logic or classical logic. A detailed description of focusing in intuitionistic logic, with proof terms, can for example be found in [21]. In the classical setting, the study of computation in the sequent calculus [9] has lead to a system called L providing a syntax extending the λµ-calculus in a symmetric way, and this system has been studied in the linear setting [19]. However, this system is not focused in general and cuts, performing the selection of a formula to focus on, cannot all be eliminated. Also related is the work on polarised linear logic [15], and in particular the encoding the λµ-calculus in polarised proof-nets [16]. We discuss this connection in Section 4.

2 Focused Proofs and Linear λ-terms

Focusing can be seen as a way of structuring proofs. Take for instance the standard rules of the multiplicative fragment of linear logic:

\[
\begin{align*}
\vdash a, \pi & \quad \vdash \Gamma, A, B \\
\vdash \Gamma, A & \quad \vdash \Gamma, A \otimes B \\
\vdash \Gamma, B & \quad \vdash \Gamma, A \otimes B
\end{align*}
\]

Proofs within this fragment have very little structure beyond that enforced by the subformula relation. Focusing allows us to enforce further structure in two ways. Firstly, the rule introducing the ≈ connective is invertible (i.e. the conclusion implies the premise), hence we can assume that this inversion property is always applied maximally within a proof. In other words, no ⊗ is decomposed if there is a ≈ in the remaining context. Secondly, and perhaps most importantly, the decomposition of ⊗ can be rearranged into maximal chains of ≈-decompositions. Note that the ⊗ rule is not invertible, as it requires the linear context to be split into two parts, and this split may not be known beforehand. Thus, decomposing, say, A ⊗ (B ⊗ C) results in subderivations containing A and B ⊗ C respectively. In the focusing discipline, we would require that B ⊗ C was decomposed immediately as well.

We enforce the maximality of the inversion (or asynchronous) and chaining (or synchronous) phases by dividing the sequent into two parts. During inversion, we maintain a list of potentially invertible formulas, and always decompose the first element of this list. If the top connective is a ≈, we put both subformulas back into the list, otherwise we move the formula into the other part of the sequent. In this way, we ensure that every ≈ formula in the context gets decomposed, if possible. During the chaining phase, we maintain a stoup containing a single formula, called the “focus”. When decomposing a ⊗, the two subformulas are put in this stoup in the premises, thus ensuring that these formulas are decomposed in turn, if possible.

We consider the multiplicative-exponential fragment of linear logic [11] with the purely linear connectives ≈ and ⊗, the exponential modalities ? and ! as well as the explicit polarity shifts ↑ and ↓ — as linear mediation between positives and negatives [15]. We also assume given a countable set A of atoms partitioned such that any atom a has a uniquely defined negative counterpart ⃗a in A. The grammar of formulas in this polarised variant of MELL is divided in two classes:

\[
P, Q ::= \downarrow\pi \mid \downarrow N \mid P \otimes Q \mid !N \\
N, M ::= \uparrow a \mid \uparrow P \mid N \otimes M \mid ?P
\]
where $P$ and $N$ denote positive and negative formulas respectively. The notion of duality ($\perp$) extending the relation between $a$ and $\overline{a}$ to all formulas is defined as usual in linear logic. Notice that in our syntax, atoms always appear immediately under a polarity shift: this is an explication of the bias, the choice of the polarity of a given atom. A polarised formula therefore contains all the required polarity information. In the following, we will write $\uparrow A$ to denote either $\uparrow a$ or $\uparrow P$, and $\downarrow A$ for either $\downarrow a$ or $\downarrow N$. The sequent calculus shown above in Figure 1 is the standard triadic form of the focused system [4] for MELL, made more precise by the use of polarity shifts. It operates on two kinds of sequents:

$$\begin{align*}
\vdash \Psi; \Gamma \uparrow S & : \text{asynchronous sequent, where } S \text{ is a sequence of negatives} \\
\vdash \Psi; \Gamma \downarrow P & : \text{synchronous sequent, where } P \text{ is a single positive}
\end{align*}$$

where $\Psi$ and $\Gamma$ are multisets containing named formulas, written $x : {?P}$ and $x : \uparrow A$ respectively. As usual, we assume that the variables affixed to these formulas are distinct. The list $S$ can be empty, and represents the part of the context treated in the inversion phase, while $P$ in the other sequent is the positive decomposed in a focus phase — more details on this system can be found in [4]. Note that formulas in $\Psi$ are subject to weakenings and contractions, since they are exponential.

The sequent calculus in Figure 1 comes with a term assignment: it is a variant of the linear $\lambda$-calculus separating terms from values. This distinction is not surprising given the known impact of focusing on computation [10], but the direct typing of $\lambda$-terms by proofs of the “classical” flavour of linear logic is. Indeed, although intuitionistic linear logic has been interpreted through linear $\lambda$-terms [7], the standard linear logic using $\otimes$ in a sequent calculus has been connected to processes [6,23] or pattern-matching [22] only. This is an illustration of the ability of focalisation to shed light on the computational meaning of proofs in the sequent calculus.

We call the language used to represent focused MELL proofs the $\lambda\pi$-calculus: it can be viewed as a linear variation on the $\lambda$-calculus representing LJT proofs.
Identity expansion. As usual in linear logic, the more general form of the identity axiom applicable on a compound formula is admissible in this system. However, due to the polarised setting, this requires a precise definition of the expansion of a negative formula $N$, denoted by $N^*$ and defined as follows:

$$(\uparrow A)^* = \uparrow A \quad (N \otimes M)^* = N^* \otimes M^* \quad (?P)^* = ?P$$

so that we can relate any $N$ to the pair of a persistent and a linear context $(\Psi, \Gamma)$ through expansion. In the following, we write $[\Gamma]$ for the set of formulas contained in $\Gamma$. We can now prove in a mutual induction the following two lemmas justifying the generalisation of the identity axiom.

**Lemma 2.1** The sequents $\vdash \Psi; x : \uparrow P \downarrow P^\perp$ and $\vdash \Psi; x : ?P; \cdot \downarrow P^\perp$ are provable.

**Proof.** In the first case, we can decompose $P^\perp$ eagerly until we obtain a premise of the shape $\vdash \Psi; \Omega; \Gamma, x : \uparrow P \downarrow \cdot$ and focus on $P$, so that we conclude by Lemma 2.2. The other case is treated the same way, but uses the exponential focus rule. $\square$

**Lemma 2.2 (Identity expansion)** If $N^* = [\Psi, \Gamma]$ then we have $\vdash \Psi, \Omega; \Gamma \downarrow N^\perp$.

**Proof.** We proceed by induction on the formula $N$. In the base case, $N$ is of the shape $\uparrow a$ and we conclude using the axiom rule. In the general case, if $N$ is of the shape $M \otimes L$ we apply the induction hypothesis on $M$ and $L$ separately and compose the proofs obtained using the $\otimes$ rule. If $N$ is of the shape $\uparrow P$ or $?P$ then we conclude using Lemma 2.1 on $P^\perp$ and either the $\downarrow$ rule or the $!$ rule. $\square$

An immediate consequence of this last lemma is that the identity axiom can now be generalised into the following two rules:

$$\begin{align*}
\frac{\vdash \Psi; x : \uparrow P \downarrow P^\perp}{\vdash \Psi, x : \uparrow P \downarrow P^\perp} & \quad \frac{\vdash \Psi, x : ?P; \cdot \downarrow !P^\perp}{\vdash \Psi, x : ?P; \cdot \downarrow !P^\perp} 
\end{align*}$$

(1)

and if such rules are added, we can type terms that are essentially not $\eta$-long. Moreover, the proof of identity expansion provides us with a transformation of terms comparable to $\eta$-expansion, where a variable value is replaced with a thunk of a more complex term depending on its type in the unordered contexts. The rules are:

\[
\begin{align*}
x & \rightarrow \eta \ [\lambda z. (x z)] & \text{if } x : \uparrow N & \quad !x & \rightarrow \eta \ [\lambda z. (x z)] & \text{if } x : ?\downarrow N \\
x & \rightarrow \eta \ [\pi \lambda y. \lambda z. x (y, z)] & \text{if } x : \uparrow (P \otimes Q) & \quad !x & \rightarrow \eta \ [\pi \lambda y. \lambda z. x (y, z)] & \text{if } x : ?(P \otimes Q) \\
x & \rightarrow \eta \ [\lambda ! z. (x ! z)] & \text{if } x : \uparrow ! N & \quad !x & \rightarrow \eta \ [\lambda ! z. (x ! z)] & \text{if } x : ?! N
\end{align*}
\]
when decomposing the result of Lemma 2.2 into small steps. In the following, we will only consider terms in \( \eta \)-long normal form — that is, such that none of the rules above apply and therefore corresponding to a proof with atomic identity axioms.

3 Cut Elimination and Focusing as Transformations

We now turn to the dynamics of the system, by considering proof transformations and their effect on corresponding terms. As usual, cut elimination is interpreted as a rewrite system implementing computation, and in our system, the focalisation result is interpreted as well, corresponding to a reorganisation that simplifies \( \lambda \pi \)-terms.

Cut elimination in focused sequent calculi is usually presented as the admissibility of a collection of cut rules. A step of cut reduction consists in having the cut interact with the rules appearing directly above it, either permuting the cuts above said rules, or decomposing the cut into smaller instances. This presentation has several drawbacks, however. Firstly, the “commutative cuts” have no computational meaning, and serve only to permute inference rules around until the next step of computation can take place. Moreover, this small-step nature of the reduction almost invariably leads to a failure of strong normalisation for the associated system of reductions.

Since we are interested in the dynamics of cut elimination, we will opt for a very conservative view. We propose that there is only one relevant instance of the cut rule, the one shown in Figure 1:

\[
\Gamma \vdash \Delta \Rightarrow N, S \quad \vdash \Delta \Rightarrow N^\perp \\
\vdash \Gamma, \Delta \Rightarrow S
\]

Note that because the calculus enforces maximal synchronous and asynchronous phases, the formulas \( N \) and \( N^\perp \) are both principal in the premises. In other words, this cut rule represents exactly the principal cases of the cut-elimination argument. To handle the cases where the principal formula is not decomposed, we see the derivation with the non-principal formula as the composition of a context and a subderivation that is again principal. For the sake of readability, we elide the proof terms for the time being: later in this section, we will show the set of reductions that form the computational content of the following theorems. To ease the notation, we use \( \vdash \Gamma; \Gamma \Rightarrow \Sigma \) as a shorthand for either \( \vdash \Gamma; \Gamma \Rightarrow P \) or \( \vdash \Gamma; \Gamma \Rightarrow S \).

The first step is to prove a series of decomposition lemmas based on a case analysis of the structure of derivations. We omit the proof of the most basic lemma, concerning formulas of the shape \( \uparrow a \).

**Lemma 3.1 (Atomic decomposition)** For any proof \( \vdash \Psi; \Gamma, x : \uparrow a \Rightarrow \Sigma \) there exists an open derivation \( \mathcal{G} \) from \( \vdash \Psi; x : \uparrow a \Rightarrow \pi \) to \( \vdash \Psi; \Gamma, x : \uparrow a \Rightarrow \Sigma \).

**Lemma 3.2 (Linear decomposition)** Any derivation \( \mathcal{E} :: \vdash \Psi; \Gamma, x : \uparrow P \Rightarrow \Sigma \) can be decomposed into a derivation \( \mathcal{F} :: \vdash \Psi, \Omega; \Delta \Rightarrow P \) and an open derivation \( \mathcal{G} \) from \( \vdash \Psi, \Omega; \Delta \Rightarrow \pi \) to \( \vdash \Psi, \Gamma \Rightarrow \Sigma \), and this open derivation does not contain an instance of the \( \uparrow \) rule between the open hypothesis and the conclusion.

**Proof.** The proof proceeds by induction on the derivation \( \mathcal{E} \). If \( \mathcal{E} \) ends with a rule that does not focus on \( x : \uparrow P \) in its premise, we place this rule on top of the open derivation \( \mathcal{G} \). If the rule focuses on \( x : \uparrow P \), then this is the desired derivation \( \mathcal{F} \), and
we are done. At no point do we encounter the ! rule in \( \varepsilon \), as this rule requires an empty linear context, which is impossible due to the presence of \( x : \uparrow P \).

The requirement that \( \not \exists \) does not contain instances of the ! rule is important, as it justifies extending the linear context uniformly across the open derivation: If \( \not \exists \) is an open derivation from \( \vdash \Psi; \Omega; \Delta \uparrow \cdot \to \vdash \Psi; \Gamma \vdash \Sigma \) satisfying the requirement, then it is also an open derivation from \( \vdash \Psi; \Omega; \Delta, \Phi \uparrow \cdot \to \vdash \Psi; \Gamma, \Phi \vdash \Sigma \).

When the formula is of the shape \( ?P \) and appears under the name \( x \) in the persistent context, we define the multiplicity \( \|x\|_{\varepsilon} \) of \( x \) in a derivation \( \varepsilon \) as the number of times \( x : ?P \) is principal in a focus rule in \( \varepsilon \) — how many focus phases are started on \( x \). This requires considering proofs modulo the usual notion of \( \alpha \)-equivalence.

**Lemma 3.3 (Persistent decomposition)** For any proof \( \varepsilon :: \vdash \Psi; x : ?P; \Gamma \vdash \Sigma \), either we have \( \varepsilon :: \vdash \Psi; \Gamma \vdash \Sigma \) or there exists a derivation \( \mathcal{F} :: \vdash \Psi; \Omega; \Delta \vdash P \) and an open derivation \( \not \exists \) from \( \vdash \Psi; \Omega; \Delta \vdash \cdot \to \vdash \Psi, x : ?P; \Gamma \vdash \Sigma \), such that \( \|x\|_{\not \exists} < \|x\|_{\varepsilon} \).

**Proof.** Again, we prove this by induction on the given derivation \( \varepsilon \). Intuitively, we pick a topmost focus rule using \( x : ?P \), yielding the derivation \( \mathcal{F} \). The remaining derivation, in which \( x \) necessarily has a lower multiplicity, then becomes \( \not \exists \).

On the basis of these decomposition lemmas, we can complete our cut elimination argument. We state it here simply as a weak normalisation argument, although we expect the calculus to admit a strong normalisation argument as well, given its similarity to the linear substitution calculus [2].

**Theorem 3.4 (Cut elimination)** The following two cut principles hold in focused MELL, assuming the derivations \( \mathcal{D} \) and \( \varepsilon \) are cut-free:

(i) if \( \mathcal{D} :: \vdash \Psi; \Gamma \vdash P \perp, S \) and \( \varepsilon :: \vdash \Psi; \Delta \vdash P \), there exists \( \mathcal{F} :: \vdash \Psi; \Gamma, \Delta \vdash S \),

(ii) if \( \mathcal{D} :: \vdash \Psi, x : ?P; \Gamma \vdash S \) and \( \varepsilon :: \vdash \Psi; \cdot \vdash P \perp \), there exists \( \mathcal{F} :: \vdash \Psi; \Gamma \vdash S \).

**Proof.** The first statement is established by induction on the structure of \( P \). Note that by design, both \( P \) and \( P \perp \) must be principal in the assumptions, hence the structure of the formula also forces what the final rule must be in the \( \mathcal{D} \) and \( \varepsilon \) derivations.

- If \( P = Q \otimes R \):

\[
\begin{align*}
\mathcal{D} &:: \vdash \Psi; \Gamma \vdash P \perp, S \\
\mathcal{E}_1 &:: \vdash \Psi; \Delta_1 \vdash Q \perp \\
\mathcal{E}_2 &:: \vdash \Psi; \Delta_2 \vdash R
\end{align*}
\]

we need to construct a derivation \( \mathcal{F} :: \vdash \Psi; \Gamma, \Delta_1, \Delta_2 \vdash S \). We proceed by applying the induction hypothesis on \( \mathcal{D} \) and \( \mathcal{E}_1 \) to obtain \( \mathcal{D}' :: \vdash \Psi; \Gamma, \Delta_1 \vdash R \perp, S \), and then apply the induction hypothesis again on \( \mathcal{D}' \) and \( \mathcal{E}_2 \) to produce the desired derivation for \( \vdash \Psi; \Gamma, \Delta_1, \Delta_2 \vdash S \).

- If \( P = \downarrow \pi \):

\[
\begin{align*}
\mathcal{D} &:: \vdash \Psi; \Gamma, x : \uparrow a \vdash S \\
\mathcal{E} &:: \vdash \Psi; \gamma : \uparrow \pi
\end{align*}
\]
where the second derivation is forced since the grammar disallows atoms without a polarity shift. We need to construct a derivation $\mathcal{F} :: \vdash \Psi; \Gamma, y : \uparrow a \uparrow S$: if we were only interested in provability, we could simply reuse $\mathcal{D}$. In order to get the term reductions to behave as expected, we instead reason as follows: by applying Lemma 3.1 on $\mathcal{D}$ we obtain an open derivation $\mathcal{G}$ from $\vdash \Psi, \Omega; x : \uparrow a \downarrow \pi$ to $\vdash \Psi; \Gamma, x : \uparrow a \uparrow S$ that we compose with $\mathcal{E}$ to produce the desired result.

- If $P = \downarrow Q^\perp$:

$$\begin{align*}
\mathcal{D} &:: \vdash \Psi; \Gamma, x : \uparrow Q \uparrow S \\
\vdash \Psi; \Gamma \uparrow \uparrow Q, S \\
\vdash \Psi; \Delta \downarrow \downarrow Q^\perp
\end{align*}$$

we need to construct a derivation $\mathcal{F} :: \vdash \Psi; \Gamma, \Delta \uparrow S$. By applying Lemma 3.2 on $\mathcal{D}$, we obtain a derivation $\mathcal{D}' :: \vdash \Psi, \Omega; \Gamma' \downarrow Q$ and an open derivation $\mathcal{G}$ from $\vdash \Psi, \Omega, \Gamma' \uparrow \cdot$ to $\vdash \Psi; \Gamma \uparrow S$. Then by the induction hypothesis on $\mathcal{D}'$ and $\mathcal{E}$ we produce a derivation of $\vdash \Psi, \Omega; \Gamma', \Delta \uparrow \cdot$ and compose it with $\mathcal{G}$ to obtain a derivation of $\vdash \Psi; \Gamma, \Delta \uparrow S$. Notice that the linear context has changed from $\Gamma'$ to $\Gamma, \Delta$, but this extension of the linear context is done uniformly across the open derivation, and hence is not a problem.

- If $P = !Q^\perp$:

$$\begin{align*}
\mathcal{D} &:: \vdash \Psi, x : ?Q; \Gamma \uparrow S \\
\vdash \Psi; \Gamma \uparrow ?Q, S \\
\vdash \Psi; \Delta \downarrow !Q^\perp
\end{align*}$$

we can directly apply the second induction hypothesis on $\mathcal{D}$ and $\mathcal{E}$.

For the second statement, we proceed by induction on the multiplicity $|x|_D$ of $x : ?P$ in $\mathcal{D}$. Given the following two derivations:

$$\begin{align*}
\mathcal{D} &:: \vdash \Psi, x : ?Q; \Gamma \uparrow S \\
\vdash \Psi; \Gamma \uparrow ?Q, S \\
\vdash \Psi; \cdot \uparrow Q^\perp
\end{align*}$$

we need to construct a derivation $\mathcal{F} :: \vdash \Psi; \Gamma \uparrow S$. By applying Lemma 3.3 on $\mathcal{D}$ we can obtain a derivation $\mathcal{D}' :: \vdash \Psi, \Omega; \Delta \downarrow Q$ and an open derivation $\mathcal{G}$ from $\vdash \Psi, \Omega, \Delta \uparrow \cdot$ to $\vdash \Psi, x : ?Q; \Gamma \uparrow S$. By the first induction hypothesis on $\mathcal{D}'$ and $\mathcal{E}$, we produce a derivation of $\vdash \Psi, \Omega; \Delta \uparrow \cdot$ and compose it with $\mathcal{G}$ to obtain a derivation of $\vdash \Psi, x : ?Q; \Gamma \uparrow S$ in which $x$ has a lower multiplicity, and finally by applying the second induction hypothesis, we get the desired derivation.

In order to describe the computational behaviour of the lemmas and of the procedure specified by the proof of cut elimination, within the language of $\lambda\pi$, we introduce the following notations for term contexts and value contexts respectively:

$$T ::= - | \lambda x.T | x V | T p | t V | \pi.T | \lambda!x.T | !x V$$

$$V ::= \sim | [T] | (V, q) | (q, V) | !T$$

We use this definition to give two kinds of contexts, depending on whether the “hole” is $-$ or $\sim$. We use $T(t)$ and $T[p]$ for the result of substituting $t$ and $p$ for $-$ and $\sim$ respectively, where the difference in brackets indicates which kind of hole is intended. The decomposition lemmas can now be stated in terms of the contexts:
Corollary 3.5 (Term decomposition) Well-typed $\lambda\pi$-terms are such that:

(i) a term $t$ containing a free, linearly occurring variable $x$ can be decomposed either into a context $T(\cdot)$ and a value $p$ such that $t = T(x\ p)$, or into a context $T(\sim)$, such that $t = T[x]$.

(ii) for a term $t$ and a free exponential variable $x$, either $x$ does not occur in $t$ or $t$ can be decomposed into a context $T(\cdot)$ and a value $p$ such that $t = T(!x\ p)$.

When reducing the term $(\lambda x. t)\ [u]$, we first decompose $t$ into $T(x\ p)$, then we create the inner cut $(u\ p)$, and finally we plug it inside the context again, yielding $T(u\ p)$ — note that this amounts to substituting $u$ for $x$ inside $t$. Written using contexts, the full set of reduction rules for $\lambda\pi$ is thus:

\[
\begin{align*}
(\lambda x. T[x])\ y & \rightarrow T[y] \\
(\lambda x. T(x\ p))\ [u] & \rightarrow T(u\ p) \\
(\pi. t)\ (p, q) & \rightarrow (t\ p)\ q \\
(\lambda!x. T(!x\ p))\ !u & \rightarrow (\lambda!x. T(u\ p))\ !u \\
(\lambda!x. t)\ !u & \rightarrow t \text{ if } x \notin \text{fv}(t)
\end{align*}
\]

where $\text{fv}(t)$ denotes, as usual, the set of all free variables of $t$. The rewrite system obtained bears a striking resemblance to the linear substitution calculus defined by Accattoli et al. [2], which is itself based on ideas from linear logic and proof-nets. We leave the investigation of this connection as future work: for the purposes of this presentation, it suffices to note that the above reductions for $\lambda$ and $\lambda!$ implement the usual notion of capture-avoiding substitution. We can therefore summarise reduction in $\lambda\pi$ using implicit substitution, which yields a system containing the following four rules:

\[
\begin{align*}
(\lambda x. t)\ y & \rightarrow t\{y/x\} \\
(\lambda x. t)\ [u] & \rightarrow t\{u/x\} \\
(\pi. t)\ (p, q) & \rightarrow (t\ p)\ q \\
(\lambda!x. t)\ !u & \rightarrow t\{u/x\}
\end{align*}
\]

and we observe that the first two correspond to the basic polarised subsystem of MELL, while the two others match the multiplicative and exponential subsystems respectively. Note that we consider here the typed fragment of the language — in the untyped case, more terms would be accepted and a thunk $[u]$ could be plugged inside a value.

Beyond cut elimination, a focused and explicitly polarised system such as ours can be given another form of dynamics, which does not represent computation as $\beta$-reduction does, but rather corresponds to a form of simplification of terms. Indeed, explicit polarity shifts can be used to introduce delays, compounds formed by a pair of opposite shifts, that can break a focusing phase or prevent maximal inversion. This can be interpreted as placing a piece of computation across a value, splitting this value in two: removing a delay therefore contributes to the production of a simpler, more compact term, with a different computational behaviour.

From the opposite viewpoint, it is clear that any unfocused proof can be mapped to a delayed, focused proof. With this, we can now restate the focalisation result in terms of the admissibility of delay elimination [8,24], rather than state it in
terms of completeness with respect to an unfocused reference system. In stating
the result, we use the notion of formula context, written $\xi\{\neg\}$. The formula $\xi\{P\}$
should be interpreted as a formula (positive or negative) with a linear occurrence of
the subformula $P$. The outer context $\xi\{\neg\}$ then represents the path through $\xi\{P\}$
on which this subformula appears.

**Lemma 3.6 (Positive delay elimination)** For any $\mathcal{D} :: |- \Psi; \Gamma, x : \uparrow \xi\{\uparrow P\} \uparrow S$
cut-free, there exists a derivation $\mathcal{E} :: |- \Psi; \Gamma, x : \uparrow \xi\{P\} \uparrow S$.

**Proof.** Before we prove this statement, we will make some simplifying assumptions.
First, note that the nature of the formula in the context $\xi\{\neg\}$ only becomes relevant
when the formula becomes focused. We will therefore assume that $\xi\{\uparrow P\}$ contains
only positive formulas between the topmost connective and the location of the
subformula $\downarrow P$. When $\xi\{\uparrow P\}$ is selected as the focus — which we may assume
appears as final rule in $\mathcal{D}$ — we can thus decompose the derivation $\mathcal{D}$ into
a subderivation $\mathcal{D}' :: |- \Psi; \Gamma' \downarrow \downarrow P$ and an open derivation $\mathcal{G}$ from $|- \Psi; \Gamma' \downarrow \downarrow P$
to $|- \Psi; \Gamma, \uparrow \xi\{\uparrow P\} \uparrow \cdot$. Note that as $\mathcal{G}$ can only split the context, we must have $\Gamma = \Gamma', \Delta$
for some context $\Delta$. Furthermore, composing any proof $|- \Psi; \Gamma'' \downarrow Q$ with this open
derivation will yield a proof of $|- \Psi; \Gamma'', \Delta \downarrow \xi\{Q\}$.

By appealing to inversion twice on $\mathcal{D}'$, we get a derivation $\mathcal{D}'' :: |- \Psi; \Gamma', \uparrow P \uparrow \cdot$
and by applying the linear decomposition lemma, this can be decomposed into a derivation
$\mathcal{F} :: |- \Psi, \Psi'; \Gamma'' \downarrow P$ and an open derivation $\mathcal{G}'$ from $|- \Psi, \Psi'; \Gamma'' \uparrow \cdot$ to $|- \Psi, \Psi'; \Gamma' \uparrow \cdot$
so that we can now string together these derivations. From $\mathcal{F}$ and $\mathcal{G}$, we get a derivation
of $|- \Psi, \Psi'; \Gamma'', \Delta \downarrow \xi\{P\}$, and hence a derivation of $|- \Psi, \Psi'; \Gamma'', \Delta, \uparrow \xi\{P\} \uparrow \cdot$ that we
compose with $\mathcal{G}'$ to get $|- \Psi, \Gamma, \uparrow \xi\{P\} \uparrow \cdot$, where we use the equality $\Gamma = \Gamma', \Delta$.  

**Lemma 3.7 (Negative delay elimination)** For any $\mathcal{D} :: |- \Psi; \Gamma \uparrow S_1, \xi\{\uparrow N\}, S_2$
cut-free, there exists a derivation $\mathcal{E} :: |- \Psi; \Gamma \uparrow S_1, \xi\{N\}, S_2$.

**Proof.** The proof proceeds in a similar manner, and we only sketch the argument.
First, we may assume that $\xi\{\uparrow N\}$ is the first formula in the inversion context,
and that only negative formulas occur on the path to $\uparrow N$. Then we do a decomposition
until $\uparrow N$ is placed in the unordered context, and find the subderivation where it is
focused again. This subderivation must immediately decompose $N$, and hence we can
transport this decomposition down, and compose it with the context $\xi\{\neg\}$ to
get the desired proof.

The focalisation result is obtained by iterated application of the previous lemmas:
in particular, removing negative delays forces $\otimes$ connectives to be decomposed eagerly,
and removing positive delays groups instances of the $\otimes$ rule. On a computational
level, the procedures specified by the proofs of these two lemmas correspond to some
reorganisation of a $\lambda \pi$-term that removes an unnecessary intermediate name.

In order to precisely described the effect of delay elimination on terms, we need
to refine our language of contexts to consider monopolar contexts, that can only
contain one pole, which is a layer of contiguous negative or positive connectives:

$$
A ::= - | (A, p) | (p, A) \\
B ::= - | \lambda x. B | \lambda !x. B | \pi.B
$$
Using these notions of contexts, we can precisely describe the effect of positive and negative delay elimination respectively:

\[
y A(\lambda x. T(x p)) \rightarrow T(y A(p))
\]

\[
\lambda x. T(x [B(y p)]) \rightarrow B(T(y p))
\]

embodied by rewrite rules that can apply anywhere inside a T context. Observe that in the basic case, where A is empty, the first rule is just \(y [\lambda x. T(x p)] \rightarrow T(y p)\), which corresponds to the start of a focus phase immediately aborted.

4 Fragments of Linear Logic and their \(\lambda\)-calculi

We have seen in the previous section how the focused sequent calculus for polarised MELL corresponds to a refined linear \(\lambda\)-calculus. However, it may be easier to grasp the expressivity of the \(\lambda\pi\)-calculus by considering subcalculi generated by specific subsets of valid formulas and sequents, or restrictions on rules.

**Purely linear fragment.** The simplest fragment that can be considered is obtained by ignoring the exponentials and the persistent context. This yields a subset of \(\lambda\pi\) that can be related to the purely linear \(\lambda\)-calculus — that is, the fragment of the \(\lambda\)-calculus where variables must be used exactly once. More precisely, there is a relatively simple encoding of the linear version of the \(\lambda\pi\)-calculus proposed by Herbelin as an interpretation of the LJT focused intuitionistic sequent calculus [12].

This calculus has lists of arguments in applications, and is based on the following syntax:

\[
t, u ::= x k \mid \lambda x. t \mid t k
\]

\[
k, m ::= \varepsilon \mid t :: k
\]

where \(\varepsilon\) represents an empty list of arguments, needed to recover the simple variable \(x\), encoded here as \(x \varepsilon\), and :: is the list constructor. The type system for this calculus, in its linear form, is given by the linear variant of LJT that we call IMLLT, where two kinds of sequents are distinguished: we write \(\Gamma \vdash N\) for an unfocused sequent and \(\Gamma, [N] \vdash a\) for a sequent focusing on the left on \(N\), where the right-hand side is limited to an atom because we consider \(\eta\)-long terms. The encoding is based on a simple translation of unpolarised intuitionistic linear formulas, that we denote by \(A\) or \(B\):

\[
\llbracket a \rrbracket = \uparrow a \quad \llbracket A \multimap B \rrbracket = \uparrow \llbracket A \rrbracket \otimes \llbracket B \rrbracket
\]

which can be extended pointwise to contexts. The most important part of the translation relates sequents of IMLLT to sequents of our system where the persistent context is always empty and thus omitted:

\[
\Gamma \vdash A \leadsto_{JT} \vdash \llbracket \Gamma \rrbracket \uparrow \llbracket A \rrbracket \quad \Gamma, [B] \vdash a \leadsto_{JT} \vdash \llbracket \Gamma \rrbracket \uparrow c : \uparrow a \downarrow \llbracket B \rrbracket \uparrow
\]

where by convention we choose \(c\) as the name of the atomic right-hand side of any focused IMLLT sequent — this variable will represent the empty list. The translation of the typing derivations is then based on these translations, as follows:

\[
[a] \vdash \varepsilon : a \quad \leadsto_{JT} \quad c \vdash c : \uparrow a \downarrow \llbracket \varepsilon \rrbracket
\]
\( \Gamma, [B] \vdash k : a \)
\[ \Gamma, x : B \vdash x : k : a \]
\[ \sim \rightarrow \text{JT} \]
\( p \vdash \downarrow \uparrow \Gamma \uparrow, c : \downarrow \uparrow \Gamma \uparrow \]
\[ x \vdash \downarrow \uparrow \Gamma \uparrow, x : \uparrow \Gamma \uparrow, c : \uparrow \Gamma \uparrow \]
\[ \lambda c. (x \ p) \vdash \downarrow \uparrow \Gamma \uparrow, x : \uparrow \Gamma \uparrow, c : \uparrow \Gamma \uparrow \]
\[ \Gamma, x : A \vdash t : B \]
\[ \Gamma \vdash \lambda x. t : A \rightarrow B \]
\[ \sim \rightarrow \text{JT} \]
\[ \Gamma \vdash t : B \quad \Delta, [C] \models k : a \]
\[ \Gamma, \Delta, [B \rightarrow C] \models t : k : a \]
\[ \sim \rightarrow \text{JT} \]
\[ u \vdash \downarrow \uparrow \Gamma \uparrow, B \]
\[ \Gamma \vdash \Delta, \ [N] \models k : a \]
\[ \Gamma, \Delta \vdash t \ k : a \]
\[ \sim \rightarrow \text{JT} \]
\[ u \ p \vdash \downarrow \uparrow \Gamma \uparrow, \Delta \]
\[ p \vdash \downarrow \uparrow \Gamma \uparrow, c : \downarrow \uparrow \Gamma \uparrow \]
\[ \lambda c. (x \ p) \vdash \downarrow \uparrow \Gamma \uparrow, c : \downarrow \uparrow \Gamma \uparrow \]
\[ \pi. \lambda x. u \vdash \downarrow \uparrow \Gamma \uparrow, \Delta \]
\[ x \vdash \downarrow \uparrow \Gamma \uparrow, x : \uparrow \Gamma \uparrow, c : \uparrow \Gamma \uparrow \]
\[ \Gamma \vdash t : N \quad \Delta, [N] \models k : a \]
\[ \Gamma, \Delta \vdash t \ k : a \]
\[ \sim \rightarrow \text{JT} \]
\[ (\lambda x. t) \ (u :: k) \rightarrow \text{JT} \quad t \{u/x\} \ k \]
\[ (\lambda x. t) \ v \rightarrow \text{JT} \quad t \]
\[ (t \ k) \ m \rightarrow \text{JT} \quad t \ (k \ o \ m) \]

where \( p \) is the translation of \( k \), and \( u \) is the translation of \( t \). The syntax \( \uparrow \Gamma \) denotes a context \( \Gamma \) where formulas are annotated with a negative shift. This translation turns lists of arguments into right-associated pairs of thunks, and translating abstractions requires the \( \pi \) operator. There is therefore a matching between these constructs, and we can compose such terms in the expected way with the cut rule, corresponding to the main head cut of \( \text{LJT} \) in its linear form:

\[ \Gamma \vdash t : N \quad \Delta, [N] \models k : a \]
\[ \Gamma, \Delta \vdash t \ k : a \]
\[ \sim \rightarrow \text{JT} \]
\[ u \vdash \downarrow \uparrow \Gamma \uparrow, \Delta \]
\[ p \vdash \downarrow \uparrow \Gamma \uparrow, c : \downarrow \uparrow \Gamma \uparrow \]
\[ \lambda c. (x \ p) \vdash \downarrow \uparrow \Gamma \uparrow, c : \downarrow \uparrow \Gamma \uparrow \]

so that reduction in this linear \( \lambda \pi \)-calculus is simulated by reduction in our focused \( \text{MELL} \) system. More specifically, the reduction system \( \rightarrow \text{JT} \) that we use for \( \text{IMLLT} \) is based on implicit substitution rather than the original explicit ones in \( \text{LJT} \) [12]:

\[ (\lambda x. t) \ (u :: k) \rightarrow \text{JT} \quad t \{u/x\} \ k \]
\[ (\lambda x. t) \ v \rightarrow \text{JT} \quad t \]
\[ (t \ k) \ m \rightarrow \text{JT} \quad t \ (k \ o \ m) \]

where \( \emptyset \) denotes the concatenation of lists. In the \( \lambda \pi \)-calculus, we consider reduction rules in their compact form, as shown in (2). As a result, we obtain the simple simulation of this linear \( \lambda \pi \)-calculus in \( \lambda \pi \) described by the theorem below.

**Theorem 4.1** If \( t \rightarrow \text{JT} \ u \) and \( u \rightarrow \text{JT} \ v \) there is a \( w \) such that \( v \rightarrow \text{JT} \ w \) and \( u \rightarrow^* \ w \).

**Proof.** We proceed by structural induction on the term \( t \), extending the statement to handle the translation of lists as \( \lambda \pi \)-terms. In the base case, it is the empty list and none of the reduction rules apply, so we are done. In general, all cases except for the redex \( t \ k \) directly rely on the induction hypothesis. In this last case, we consider possible reductions, so that the compound reductions:

\[ \lambda c. ((\pi. \lambda x. t) \ (\downarrow \uparrow \Delta \uparrow, c : \downarrow \uparrow \Delta \uparrow)) \rightarrow^* \lambda c. (t \{u/x\} \ q) \]
\[ \lambda c. ((\lambda x. t) \ c) \rightarrow^* \lambda c. (t \ c \ x) \]
\[ \lambda c. ((\lambda d. u) \ q) \rightarrow^* \lambda c. (u \ p \ q \ d) \]

325
are just the translations of the reduction rules shown in (3). The first reduction is simple and relies on the decomposition of the pair encoding the :: constructor, and a substitution. The second reduction just performs a substitution, but one should note that after reduction, c is no longer a right-hand side marker, but simply a renaming of x. Finally, the last reduction relies on the encoding of lists as right-associated pairs, so that \( p\{q/d\} \) is exactly the encoding of \( k \otimes m \) in \( \lambda\pi \).

Notice that this encoding translates primitive constructs in \( \lambda\pi \) into compound constructs of the \( \lambda\pi \)-calculus. Some terms in our calculus have no equivalent in the interpretation of IMLLT: we have captured here only an intuitionistic fragment of MLL. This can be seen quite clearly in our translation, as it corresponds to the presence of the single variable \( c \) with a type of the shape \( \uparrow a \) in the context, which represents the unique right-hand side of a sequent. We can capture a larger, more classical fragment of the calculus, but makes the embedding slightly more complex. Although we could study encodings of a linear variant of the \( \lambda\mu \)-calculus, we choose to use exponentials to represent the full sequent-based variant of \( \lambda\mu \).

**Exponential fragment.** Beyond the purely linear fragment discussed above, an obvious subsystem of interest is the one where no polarity shifts appear, related to LLP [15], and one might want to push this further and try to eliminate the need for a linear context altogether. This is problematic since the axiom rule applies only when the linear context is not empty, but it is possible to work around this problem by considering the exponential axiom rule from (1) obtained from the identity expansion result:

\[
\frac{}{!x \vdash \Psi, x : ?P \Downarrow !P^\perp}
\]

However, it is not possible to avoid entirely polarity shifts in our system, since atoms are not handled without them. We consider the \( \overline{\lambda}\mu \)-calculus of Herbelin [13], which is an interpretation of LKT, and can be obtained by adding the control rules from \( \lambda\mu \) into the LJT system and term assignment. Due to the polarised setting in MELL, we use an explicitly polarised version of this calculus, where the \( \mu \) and naming rules are reflected on types by shifts — that we write \( \downarrow \) and \( \uparrow \) to distinguish them from the shifts of MELL. The translation of formulas is then defined as:

\[
[a] = ?\overline{a} \quad [P \supset N] = ?[P]^\perp \otimes [N] \quad \uparrow N = ![[N]] \quad \downarrow P = ?[P]
\]

The LKT system has two kinds of sequents, just as LJT, which are obtained by adding the context \( \Delta \) of other right-hand sides. Since we used polarised formulas, the context \( \Gamma \) in the left-hand side always contains formulas of the shape \( \downarrow N \), just as \( \Delta \). Sequents are then encoded using the translation of formulas as follows:

\[
\frac{}{\Gamma \vdash N \mid \Delta \rightsquigarrow_{\text{KT}} \vdash [\Gamma]^\perp, ?[\Delta] \uparrow [N]}
\]

\[
\Gamma, [N] \vdash a \mid \Delta \rightsquigarrow_{\text{KT}} \vdash [\Gamma]^\perp, ?[\Delta], c : ?\pi \Downarrow [N]^\perp
\]

where focused MELL sequents are written without a linear context, since it will always be empty in this translation. Indeed, polarity shifts, needed to introduce formulas in the linear context, are used only on atoms and therefore they cannot be treated out of the axiom rule. The control rules of the calculus, concerning \( \mu \) and
naming, are translated as follows in our system:

\[
\begin{align*}
\Gamma \vdash t : N & \quad \alpha : \Delta & \quad \Leftrightarrow \quad u \vdash [\Gamma]^{+}, ?[\Delta], c : ?! [N] \uparrow. \\
\Gamma \vdash \mu \alpha.t : N & \quad \alpha : \Delta & \quad \Leftrightarrow \quad \lambda !c.u \vdash [\Gamma]^{+}, ?[\Delta], \uparrow ? ![N] \\
\Gamma \vdash t : N & \quad \alpha : \Delta & \quad \Leftrightarrow \quad u \vdash [\Gamma]^{+}, ?[\Delta], c : ?! [N] \uparrow [N] \\
\Gamma \vdash [\alpha] \ t : \alpha : \Delta & \quad \Leftrightarrow \quad !u \vdash [\Gamma]^{+}, ?[\Delta], c : ?! [N] \downarrow [N] \\
\end{align*}
\]

where \(c\) is the name given to the marker labelled \(\alpha\) in \(\mathbf{LKT}\), and \(u\) is the translation of \(t\). The other rules are translated in a way very similar to the encoding for \(\mathbf{LJT}\), but they all have the generalised treatment of the right-hand side context \(\Delta\):

\[
\begin{align*}
\Gamma, [\alpha] \vdash \varepsilon : a & \quad \Leftrightarrow \quad \Leftrightarrow \quad \lambda c !. [\Gamma]^{+}, ?[\Delta], c : ?! \pi \downarrow \uparrow a \\
\Gamma, [N] \vdash k : a & \quad \Delta & \quad \Leftrightarrow \quad \Leftrightarrow \quad \pi x p \vdash [\Gamma]^{+}, x : ?[N]^{+}, ?[\Delta], c : ?! \pi \downarrow \uparrow [N] \\
\Gamma, x : [N] \vdash x \ k : a & \quad \Delta & \quad \Leftrightarrow \quad \Leftrightarrow \quad \lambda c !. (\lambda x p) \vdash [\Gamma]^{+}, x : ?[N]^{+}, ?[\Delta] \uparrow \uparrow \pi \\
\Gamma, x : [N] \vdash t : M & \quad \Delta & \quad \Leftrightarrow \quad \Leftrightarrow \quad \pi ! x . u \vdash [\Gamma]^{+}, ?[\Delta] \uparrow ? ![N]^{+} \uparrow ![M] \\
\Gamma \vdash \lambda x. t : [N \supset M] & \quad \Delta & \quad \Leftrightarrow \quad \Leftrightarrow \quad \pi ! x . u \vdash [\Gamma]^{+}, ?[\Delta] \uparrow ? ![N]^{+} \uparrow \uparrow ![M] \\
\pi \lambda x. u \vdash [\Gamma]^{+}, ?[\Delta] \uparrow ? ![N]^{+} \uparrow ![M] \\
\pi ! u \vdash [\Gamma]^{+}, ?[\Delta], c : ?! \pi \downarrow ![N] \\
\end{align*}
\]

Observe that the left implication rule is compound in our presentation of this calculus, but so is the translation, in the same way. The key idea here is that more than a single right-hand side marker can be used in a single sequent, due to the classical setting. But in the \(\lambda \pi\)-calculus itself, control does not need to use these tools: it is the consequence of the shape of typing rules, and in particular of the continuation behaviour of the focus rule. Indeed, if a variable of type \(\uparrow P\) is available, it can be applied to a value typed by \(P\) itself in a focused phase. The continuation behaviour of variable application is well illustrated by the a special case of positive delay elimination:

\[
y [\lambda x. T(x \ p)] \rightarrow T(y \ p)
\]

which can be read as \(y [\lambda x.t] \rightarrow t[y/x]\), so that we see \(t\) using the name of the variable to which it was given as argument.
Finally, the question of simulating the reduction of the $\lambda\mu$-calculus using this encoding is more complex than in the purely linear case. Indeed, the reduction rule for $\mu$ is performing a relatively complex operation:

$$(\mu\alpha.t) \ k \rightarrow_{KT} \ \mu\alpha.t\{[\alpha](u\ k)/[\alpha]\ u}\$$

that can be observed in $\lambda\pi$ if the proper cut and shift rules are introduced, but these yield problems concerning the preservation of types of different subterms during the reduction process.

5 Conclusion and Future Work

We presented a Curry-Howard interpretation of focused MELL, with a novel proof of cut elimination based on reducing cuts at a distance. The variant of the $\lambda$-calculus obtained is more similar to the usual $\lambda$-calculus than other calculi based on sequent calculi and it offers a simple syntax for MELL proofs. Moreover, this system has connections to some well-known variations of the $\lambda$-calculus, and its reduction simulates in one step the usual notion of $\beta$-reduction. The investigation of the computational meaning of a focused cut elimination is important, as it relates to the question of evaluation strategies and has a nice proof-theoretic behaviour.

As mentioned, the system of reductions we present bears a striking resemblance to that of the linear substitution calculus, which employs a similar notion of reduction at a distance. It would be interesting to observe how deep this similarity is, and whether the LSC can be generalised based on the proof-theoretic approach presented here. In a similar vein, it would be interesting to have a computational interpretation of focused proofs in the presence of the remaining connectives of linear logic, in particular the additives, which are known to introduce a notion of case analysis. Richer logics could also be considered, such as $\mu$MALL [5], where induction and coinduction are supported directly in an elegant, proof-theoretic way. The use of fixpoints as alternative to persistent variables yields many questions, concerning for example the use of this system as a programming language. In terms of language it would also, of course, be interesting — and surely straightforward — to extend this interpretation to second-order quantifiers, reaching the expressivity of System $F$.

Finally, it is well-known that using delays, many other calculi can be represented as appropriately polarised fragments of a strongly focused sequent calculus. This is seen for example in [18] where LJT and LJQ are both shown to be representable as fragments of LJF. Given the generality of our calculus, it is possible that it could serve as a lingua franca for the large variety of classical $\lambda$-calculi found in the literature. Exploring such a possibility will pinpoint the general position of linear logic in the field, and highlight how it provides a crucial tool in the understanding of computational phenomena.

Acknowledgement

The authors would like to thank the anonymous reviewers for their helpful and thorough comments on a previous draft.
References


The expressiveness of CSP with priority

A.W. Roscoe

Oxford University Department of Computer Science

Abstract

The author previously [16,15] defined CSP-like operational semantics whose main restrictions were the automatic promotion of most \( \tau \) actions, no cloning of running processes, and no negative premises in operational semantic rules. He showed that every operator with such an operational semantics can be translated into CSP and therefore has a semantics in every model of CSP. In this paper we demonstrate that a similar result holds for CSP extended by the priority operator described in Chapter 20 of [15], with the restriction on negative premises removed.

Keywords: CSP, operational semantics, priority

1 Introduction

As well as its denotational semantics in models such as traces \( T \) and failures-divergences \( N \), CSP [11] has a well-established operational semantics first described in SOS in [5,6], and congruence with that is perhaps the main criterion for the acceptability of any new semantic model.

The author previously created a class of CSP-like operational semantic definitions that automatically have semantics over every CSP model. In addition to a number of other restrictions on the full generality of Structured Operational Semantic (SOS) definitions, CSP-like ones are not permitted any negative premises: thus there can be no rule in which some action can fire only if one of its arguments can not perform some (either one or more) action(s).

There have been a number of proposals for adding priority to CSP. A straightforward one, because it does not involve building special semantic models or types of LTSs, was proposed in [15]. \( \text{Pri}_{\leq} (P) \), for a partial order on the events that processes perform, permits \( P \) an event \( x \) only when no higher priority event is possible. With restrictions on how the invisible event \( \tau \) fits into \( \leq \), this adds very usefully to CSP, for example by permitting the accurate description of real-time systems.

\( \text{Pri}_{\leq} (\cdot) \) is not CSP-like since it requires negative premises. Indeed it does not have a semantics in most CSP models. This raises the question of whether we can...
capture a notion of Pri-CSP-like operational semantics which includes this operator, where all Pri-CSP-like operators can be expressed in terms of CSP plus \( \text{Pri} \leq (\cdot) \). Establishing such a notion is the job of the present paper.

In the next section, we remind ourselves about CSP and its operational semantics. We then recall CSP-like operational semantics and outline their expressiveness result. Finally we recall the definitions of \( \text{Pri} \leq (\cdot) \) in terms of operational semantics and over \( \mathcal{FL} \), the finite linear or ready traces model that can record an acceptance set before each event. In Section 3 we generalise the definition of CSP-like to achieve the goal set out above. The main result of this paper then follows, in which we show that any operator (or class of operators) with such Pri-CSP-like operational semantics can be simulated precisely in augmented CSP. The precision obtained by this simulation depends on whether or not the language involves the CSP concept of termination, represented \( \checkmark \). However, for brevity this paper does not include the role of \( \checkmark \) in CSP semantics: it is fully covered in the extended version [18].

As with [16], the primary motivation of this paper is to characterise what operators and languages can be translated into CSP (in this paper extended by \( \text{Pri} \leq (\cdot) \)) to identify which of these can be handled on the model checker FDR [9], which itself now supports this operator \(^2\). We give some examples of what is now representable in Section 5.

2 Background

2.1 The operational semantics of CSP

The SOS operational semantics [5,6] of CSP came along after its well-known denotational semantics. For CSP (without \( \checkmark \) and sequential composition), the action labels come from \( \Sigma \cup \{\tau\} \), where \( \Sigma \) is the alphabet, the actions that are visible to and controllable by the external observer, and \( \tau \) is an invisible and uncontrollable event such that whenever it is enabled and another event does not happen quickly, it will. Given the process \( P \), \( \alpha P \) means its own set of \( \Sigma \) actions, which is usually just the visible events it uses.

In SOS style [13] we need rules to infer every action that each process can perform. The conditions that enable actions can be of three sorts:

- **Positive**: Some other process can perform a specific action. This other process is determined from the syntax of the process \( P \) whose transitions we are calculating. In our setting these other processes are, except in the case of recursion, arguments of the operator whose semantics we are defining.

- **Negative**: The same except the other process cannot perform a given action.

- **Side conditions** on the actions etc that appear.

A rule has a set of actions/alphabets etc parameters, and some positive and/or negative premises. A rule with free parameters other than processes is a rule schema denoting a separate formal rule for each permitted value of these.

\(^2\) FDR3 supports two priority operators: \texttt{prioritisepo} is directly equivalent to the one used in this paper, while \texttt{prioritise} is a restricted case that does not require the programmer to construct an explicit partial order.
CSP has a few constant processes, a number of operators which can be applied to argument processes, and recursive constructions. The operational semantics of constants simply describe their actions directly. STOP, which has no actions, has no operational rules. Other constants are RUNA, which performs any sequence of events from A ⊆ Σ and never refuses one, ChaosA, which is the most nondeterministic non-divergent process on the events A, and div, which simply diverges: performs an infinite series of τs.

There are two approaches to the operational semantics of recursion:

\[
\begin{align*}
\mu p. P & \xrightarrow{\tau} P[\mu p. P/p] \\
\mu p. P & \xrightarrow{\tau} Q \quad (B)
\end{align*}
\]

where p is a process identifier and P a process term where p may be free. Rule (A) introduces a τ every time a recursion is unwound, and Rule (B) does not. Thanks to the CSP principle that the process τ P (in CCS notation: one that performs a τ before becoming P) is equivalent in all but operational semantics to P, there is no observable difference between the results of these two rules, provided (B) is well defined. For a clean analysis of operational semantics, (A) is better as the τ guards eliminate problems caused by under-defined recursions (of which the simplest example is µ p. p), which become more severe in the presence of negative premises.

Without such an undefined recursion (one where the first-step actions of a recursive body P[Q/p] are not independent of those of Q, or where the derivation of actions in an infinite mutual recursion is not well founded, as with the recursion \( P_i = P_{i+1} \square a \rightarrow STOP \)), such problems do not arise and (B) gives a more efficient LTS (i.e. less states and transitions). In this paper, for simplicity (not only with negative premises) we generally assume approach (A) in any case where it cannot be determined simply that every recursive call is guarded by at least one action (which can be τ), and the more efficient (B) otherwise.

2.2 The transition rules of CSP operators

Communications are introduced via prefixing e → P. It has rule

\[
e \rightarrow P \xrightarrow{a} subs(a, e, P) \quad (a \in comms(e))
\]

Here e may represent a range of possible communications and bind one or more identifiers in P, as in the examples ?x : A → P, c!?x?y → P and c!?xe → P. We assume the existence of functions

- \( comms(e) \) is the set of communications described by e. For example, \( d.3 \) represents \{d.3\} and c?x:A?y represents \{c.a.b | a.b ∈ type(c), a ∈ A\}.
- If a ∈ \( comms(e) \), \( subs(a, e, P) \) substitutes part of a for each identifier bound by e. So \( subs(c.1,2, c!x?y, d!x → P(x, y)) = d!1 → P(1, 2) \).

3 A formulation of ChaosA valid in all CSP models has τ transitions to ?x : B → ChaosA for every B ⊆ A. This can be simplified when only the most common semantic models are in use to have only two states: one which can do a τ to STOP (the other state) or any member of A to itself.
Nondeterministic choice picks an argument to act like:

\[
P \sqcap Q \xrightarrow{\tau} P \\
P \sqcap Q \xrightarrow{\tau} Q
\]

The initial actions of prefixing and \( P \sqcap Q \) do not depend on those of process arguments. All the other operators have rules that allow us to deduce what actions a process of the given form has from the actions of the sub-processes. Operators may have some arguments ‘active’ and some ‘inactive’. The former are those whose actions are immediately relevant, the latter the ones which are not needed to deduce the first actions of the combination.

Both arguments of external choice (\( \Box \)) are active. When an argument is active, it must be allowed to perform any \( \tau \) action it is capable of, since the argument’s environment (in this case the operator) is incapable of stopping them. Such \( \tau \) actions are invisible to the operator, so there are always rules like the following for active arguments:

\[
\begin{align*}
P \xrightarrow{\tau} P' \\
P \Box Q \xrightarrow{\tau} P' \Box Q
\end{align*}
\]

which simply allow the \( \tau \) to happen without otherwise affecting the process state. These promote the \( \tau \) actions of the arguments to \( \tau \) actions of the whole process. \( \Box \) can use visible actions, here resolving the choice.

\[
\begin{align*}
P \xrightarrow{a} P' \quad & (a \neq \tau) \\
P \Box Q \xrightarrow{a} P' \quad & (a \neq \tau)
\end{align*}
\]

It is important that the argument \( P \) of \( e \to P \) is inactive. If not, it would be allowed to perform \( \tau \)s so \( a \to P \) might diverge without performing \( a \).

The rules for hiding and renaming both allow all the actions of the underlying process but change some of the names of the events. Renaming applies a relation to visible ones; hiding turns selected actions into \( \tau \)s.

\[
\begin{align*}
P \xrightarrow{x} P' \quad & (x \notin B) \\
P \setminus B \xrightarrow{x} P' \setminus B
\end{align*}
\]

\[
\begin{align*}
P \xrightarrow{\tau} P' \\
P \lbrack \Box R \rbrack \xrightarrow{\tau} P' \lbrack \Box R \rbrack
\end{align*}
\]

We give the semantics of just one parallel operator. Others can be deduced from it: \( P \parallel [X] \parallel Q \) synchronises \( P \) and \( Q \) on all actions in \( X \), and lets them communicate freely on other events. Both arguments are active

\[
\begin{align*}
P \xrightarrow{\tau} P' \\
P \parallel [X] \parallel Q \xrightarrow{\tau} P' \parallel [X] \parallel Q
\end{align*}
\]

There are three rules for visible events: two symmetric ones for \( a \notin X \)

\[
\begin{align*}
P \xrightarrow{a} P' \quad & (a \in \Sigma \setminus X) \\
P \parallel [X] \parallel Q \xrightarrow{a} P' \parallel [X] \parallel Q
\end{align*}
\]

\[
\begin{align*}
P \parallel [X] \parallel Q \xrightarrow{a} P' \parallel [X] \parallel Q' \quad & (a \in \Sigma \setminus X)
\end{align*}
\]
and one to show $a \in X$ requiring both participants to synchronize

\[
\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P || X || Q \xrightarrow{a \cap X} P'||X||Q'}\quad (a \in X)
\]

Other forms of CSP parallel are *interleaving* $P || Q$, equivalent to $P || [\emptyset] || Q$, and *alphabetised parallel* $P || [A || B] || Q$, which forces $P$ to communicate all events in $A$, and $Q$ in $B$. Provided that $P$ and $Q$ do not attempt to communicate outside $A$ and $B$ respectively it is equivalent to $P || [A \cap B] || Q$.

CSP provides two ways of getting one process to take over from another without the first one terminating: interrupt $P \triangledown Q$ allows $P$ to run, but at any time offers the initial events of $Q$. If one of the latter happens then $Q$ takes over. Both arguments are initially active.

\[
\frac{P \xrightarrow{\tau} P'}{P \triangle Q \xrightarrow{\tau} P' \triangle Q}
\]

\[
\frac{Q \xrightarrow{\tau} Q'}{P \triangle Q \xrightarrow{\tau} P \triangle Q'}
\]

If $P$ performs $a \in \Sigma$, then the result is interruptable, whereas if $Q$ performs $a \in \Sigma$, then it takes over.

\[
\frac{P \xrightarrow{a} P'}{P \triangle Q \xrightarrow{a} P' \triangle Q}\quad (a \in \Sigma)
\]

\[
\frac{Q \xrightarrow{a} Q'}{P \triangle Q \xrightarrow{a} P \triangle Q'}\quad (a \in \Sigma)
\]

The other operator allows any event from $P$ in the set $A$ to close it down and hand over to $Q$: the *throw* operator $P \Theta_A Q$. $P$ is active: it is allowed to perform $\tau$ or $a \notin A$ and carry on whereas $a \in A$ hands control to $Q$:

\[
\frac{P \xrightarrow{\tau} P'}{P \Theta_A Q \xrightarrow{\tau} P' \Theta_A Q}\quad (x \notin A)
\]

\[
\frac{P \xrightarrow{a} P'}{P \Theta_A Q \xrightarrow{a} Q}\quad (a \in A)
\]

The operational semantics of $\text{Pri}_<\cdot\cdot\cdot$ can be found in Section 2.4.

### 2.3 CSP-like operational semantics

All premises above are positive. The rules also have properties

- If an argument process performs an action $P \xrightarrow{x} P'$ in the premises, and remains after the derived action then $P$ has become $P'$ in the result.
- If an argument process does not act in the premises, then if it remains after the action it stays in its initial state.
- If $P$ appears in any of the premises of the operator $F(P,\ldots)$ (i.e., the initial actions of $F(P,\ldots)$ depend on those of $P$), and $P \xrightarrow{\tau} P'$, then $F(P,\ldots) \xrightarrow{\tau} F(P',\ldots)$. There are no other rules with $\tau$ as a premise.
- No argument process ever appears more than once in the result of any actions. This is the *no cloning* property. In fact CSP can clone inactive arguments via recursion, but never active ones.

In [16,15], the author codified all of the above conditions together, including the banning of negative premises, and described an operational semantics all of whose
operators obey these principles as CSP-like. The clearest way of doing this was creating a new notation for operational semantics, so constrained that it can only express CSP-like operators.

CSP-like operational semantics bears close comparison with simply WB cool rules as defined in [10]. This is a restriction on SOS that ensures that operators respect weak bisimulation (hence WB). We adopt some of the nomenclature of [10], though this is different from that in [15,16]. This includes the terms active, and inactive otherwise. [15,16] termed these on and off respectively. The rules which simply promote a \( \tau \) action are called patience rules.

In giving a combinator semantics for the operator \( F(P_1, \ldots, P_n) \), the first thing we need to identify is which of the \( P_i \) are initially active: which of them appear in the premises of \( F \)'s SOS operational rules. The notation we will use for an operator with active arguments \( P \) and inactive ones \( Q \) in defining its combinator semantics will take the form \( F^Q(P) \), emphasising that the active ones are those immediately relevant. We allow an infinite number of components to \( Q \). This case does arise in CSP, both thanks to taking the nondeterministic choice of an infinite number of processes and, in the case where the alphabet \( \Sigma \) is infinite, prefix constructs (such as \( e?x \rightarrow \cdot \) when the type of \( e \) is infinite). We only allow finitely many active arguments: not only does the infinite case not arise in CSP, but it is theoretically problematic.

As with SOS, a combinator operational semantics consists of rule schemas, with events, sets of events etc varying under side conditions to create sets of rules for individual operators. An individual rule takes the form of a triple, sometimes abbreviated to a pair.

- The first component is a tuple with one component for each active argument. The members of this \( m \)-tuple \( (x_1, \ldots, x_m) \) are taken from \( \Sigma \cup \{\cdot\} \). The meaning of this tuple is that all active arguments whose component is not “·” perform the relevant action, in a synchronised fashion, for the rule to fire. (We will put quotes around “·” in text to help distinguish it.) Note that in some CSP operators \( m = 0 \), which simply says that all of the operator’s actions are unconditional on arguments’ actions. In these cases we write the now null premises as \( \rightarrow \). Note that \( \tau \) is not permitted in these tuples: we will discuss this below.

- The second component is an action \( y \) in \( \Sigma \cup \{\cdot, \tau\} \) which represents the result action of the rule: the one that the operator performs when the active arguments perform the components of the first. Hiding gives a case where a visible action is turned into \( \tau \), hence the possibility of \( y \) being \( \tau \).

- The third component represents the successor process after the action. There are two possibilities here:
  - (i) The result of the action does not change the process’s shape: it is still the same operator applied to the same arguments, the only change being that those active arguments that have participated in the action have moved forward according to respective component actions. This is a common case, and applies to all actions of parallel, hiding and renaming operators, and combinations of these. The third component is then omitted, so the combinator becomes a pair. Such combinators are homogeneous.
(ii) In any other case we do need to record the state that the process moves into. This will always be a piece of syntax with place-holders for the active and inactive arguments. The form of this syntax has to be restricted so as to prevent either the cloning or suspension of the active arguments of the original operator. The syntax can, however, do what it likes with the inactive arguments, and discard any argument it wishes.

The way combinators build the syntax of successor processes can be defined by specifying that they must treat active arguments, if they are retained at all, in a way that keeps them active and follows the principles of distributivity, common to all non-recursive CSP operators. This is a piece of syntax $T$ in which each argument (active and inactive) is represented by some standardised identifier. For us these are bold-face indices drawn from $\{1, \ldots, m\} \cup I$, so $1$ represents the first active argument, and so on. The result state is now $T$ with the substitutions:

- An index $i \in \{1, \ldots, m\}$ is replaced by $P_i$ or $P'_i$ such that $P_i \xrightarrow{x_i} P'_i$ depending on whether $x_i = \cdot$ or $x_i \in \Sigma$.
- An index $i \in I$ is replaced by $Q_i$.

To follow the principles above we have to impose conditions on $T$:

- No active index $i \in \{1, \ldots, m\}$ can appear more than once in $T$.
- Such active indexes only appear at immediately distributive (ID) places in $T$ (i.e., where the operational semantics we can derive for $T$ makes a process argument placed here initially active). This is easy to define by structural recursion:
  - The appearance of $i$ in the simple term $i$ is ID.
  - If $i$ appears ID in the term $T$, then it appears ID in $\bigoplus(...)T(...)$, where the place $T$ is an active argument of the CSP-like operator $\bigoplus$.
  - No other appearance of $i$, including any in a recursive definition, is ID.

The pieces of syntax $T$ above can contain arbitrary closed CSP processes at any point without restriction.

Hiding $P \ \not\! X$ has rules $(a, a)[a \not\in X]$ and $(a, \tau)[a \in X]$, using the convention that for operators with a single active argument, we write $a$ rather than $(a)$ for the first component. The result of $P \ \not\! X$ processing an action $P \xrightarrow{a} P'$ is always $P' \ \not\! X$, so its combinators are homogeneous. On the other hand, the resolution of $P \square Q$ does change the process structure, so its rules are $((a, \cdot), a, 1)$ and $(\cdot, a, a, 2)$: either side can perform any action in $\Sigma$, resolving the choice. We do not write down patience rules since they always apply.

**Definition 2.1**

An operator (language) is CSP-like if and only if it (all its operators) can be given a combinator operational semantics.

**Theorem 2.2** Every CSP-like operator $F$ has a translation to CSP which we write $F_{\text{CSP}}$ such that, for any collection of arguments $(P, Q)$, the operational semantics of $F^Q(P)$ and $F^Q_{\text{CSP}}(P)$ are strongly bisimilar.

Therefore any CSP-like operator has a fully compositional semantics over any model of CSP.
The proof can be found in [16,15] and is indicated in that of the main theorem of this paper.

2.4 Priority

While there have been a number of versions of CSP with priority, for example [12,7], the one we use in this paper is that introduced in [15]. This is conceptually simple, as it does not require any re-interpretation of LTS’s or CSP models as entities where one action has priority over another. Instead $\text{Pri}_\leq(\cdot)$ inputs an ordinary LTS and the result is another ordinary one. $\leq$ is a partial order on events $\Sigma \cup \{\tau\}$ which is subject to several conditions stated below. The SOS operational semantics are

$P \xrightarrow{x} P' \land \forall y. y > x \Rightarrow \neg P \xrightarrow{y}$

$\text{Pri}_\leq(P) \xrightarrow{\cdot} \text{Pri}_\leq(P')$

$P$ performs actions that are not strictly lower under $\leq$ than another action that $P$ can perform from the same state. In the above, $x$ and $y$ range over the whole of $\Sigma \cup \{\tau\}$. In order to make this consistent with the tenets of CSP we need to respect the idea that $\tau$ is not controllable and that every process is equivalent to the one where a single $\tau$ precedes it:

- $\tau$ is maximal in $\leq$: it is not dominated by any other event.
- If $a < b$ for any actions $a$ and $b$, then $a < \tau$

Only the richest CSP models make $\text{Pri}_\leq(\cdot)$ compositional. Of those discussed in Chapters 10, 11 and 12 of [15], the only ones compositional for the full range of permitted $\leq$ are the $\mathcal{FL}$ class of models, recording traces extended by one of the following before each event and after the last:

- $\bullet$ meaning that the state from which the next event happened, or at the end of the behaviour, has not been observed to be stable (i.e., a state where no $\tau$ is possible).
- Where stability is observed, the exact set of events that the state offers.

Thus a typical behaviour looks like $(A_0, a_1, A_1, \ldots, A_{n-1}, b_n, A_n)$ with the $b_i$ being drawn from $\Sigma$, and the $A_i$ being drawn from $\{\bullet\} \cup \mathcal{P}(\Sigma)$

The semantics of $\text{Pri}_\leq(P)$ over $\mathcal{FL}$ are as follows, quoted from [17].

$$\{ (A_0, b_1, A_1, \ldots, A_{n-1}, b_n, A_n) \mid (Z_0, b_1, Z_1, \ldots, Z_{n-1}, b_n, Z_n) \in P \}$$

where for each $i$ one of the following holds:

- $b_i$ is maximal under $\leq$ and $A_{i-1} = \bullet$ (so there is no condition on $Z_{i-1}$ except that it exists).
- $b_i$ is not maximal under $\leq$ and $A_{i-1} = \bullet$ and $Z_{i-1}$ is not $\bullet$ and neither does $Z_{i-1}$ contain any $c > b_i$.
- Neither $A_i$ nor $Z_i$ is $\bullet$, and $A_i = \{a \in Z_i \mid \neg \exists b \in Z_i. b > a \}$.

---

4 Tom Gibson-Robinson [8] implemented the constructions of [16], thereby providing a translation of arbitrary CSP-like operators into CSP for use on FDR [9].
• In each case where $A_{i-1} \neq \bullet$, $b_i \in A_{i-1}$.

Priority is not CSP-like, so we name the extended language $Pri$-$CSP$.

3 What can we express in $Pri$-$CSP$?

$Pri_\leq(\cdot)$ has some of the qualities of CSP-like operators, for example it has the patience property, and never clones its argument. The only one it obviously fails is the ban on negative premises.

To grasp what can be expressed in $Pri$-$CSP$ we change the expressive power of combinators. Recall that the first component of a combinator is a tuple of actions from the active processes. We can extend this by turning the components of this tuple into pairs. The first component is either an action in $\Sigma$ that the corresponding process should perform or “·” if it does not perform one in the action. The second is a set of events, which if non-empty contains $\tau$ (if not written down it is assumed implicitly), that the process must not be able to perform if the rule is to fire. We annotate such negative premises with the negation symbol $\neg$. A negative premise can only be satisfied in a stable state of its argument.

We will be liberal with the way we write down such pairs: where one or other component is trivial (i.e., · or $\emptyset$ (rather than $\{\tau\}$)) we will just write the other, and if both are trivial we will just write “·”.

There is no difference in the second component of combinators. However, problems discussed fully in [18] make us more restrictive in the syntax of the allowed third component syntax $T$. Specifically we restrict the third component to be any of

• One of the argument processes by itself (a common case in CSP): this can be active or inactive in the original state.
• Any constant CSP process (one that does not refer to any argument).
• Any $Pri$-$CSP$ operator application where each active argument of the original operator, if it appears at all, appears in exactly one place amongst the active arguments of the new operator.

We again assume a patience rule for each active argument. A homogeneous $n$-combinator is one where the third component is omitted because the result has the same structure as the initial process.

Any combinator with a negative premise is termed an $n$-combinator, and an $n$-combinator operational semantics is one in terms of these and ordinary combinators. A positive combinator semantics is one with only ordinary combinators.

$Pri_\leq(\cdot)$’s operational semantics can itself be expressed in these extended combinators. It has the implicit patience rule and, for each $a \in \Sigma$ maximal in $\leq$ the simple combinator $(a, a)$. For non-maximal $a$ it has the $n$-combinator $( (a, \neg \{ x \in \Sigma \cup \{\tau\} \mid a < x \} ), a )$, where we note that the set of negative premises always includes $\tau$ due to the restrictions placed on $\leq$ in the definition of the priority operator.

Recall that the operator $P \Theta_A Q$ starts $Q$ whenever $P$ communicates an element in $A$. We can think of this as $P$ throwing an exception. With $n$-combinators we could build an operator $P \Delta \Theta Q$ in which any deadlock in $P$ was caught and starts
$Q$: with the active argument $P$ it would simply need the combinators $(a, a)$ for $a \in \alpha P$ plus the n-combinator $(\neg \alpha P, \tau, \mathbf{q})$, where $\mathbf{q}$ points to the inactive argument $Q$. Once we have discussed the implementation of general negative premises later, we will show how to implement this.

**Definition 3.1** An operator has Pri-CSP-like operational semantics if its operational semantics can be given according to the above conventions in terms of combinators and n-combinators.

### 4 Expressibility theorem

**Theorem 4.1** Suppose the operator $F^{Q}(P)$ is Pri-CSP-like together with all other operators reachable (transitively) through the $T$ third components of its combinators. Then for any arguments $P$ and $Q$, $F^{Q}(P)$ is expressible in Pri-CSP in the sense that the simulation is strongly bisimilar to $F^{Q}(P)$.

This implies that such operators have a compositional semantics over $\mathcal{FL}$.

As in [16,15], our proof is to construct the (Pri-)CSP implementation. This is even more complex than the one without negative premises. For the issues in common with the earlier result, the constructions we use have a lot in common, though we do find several simplifications.

(i) First we consider the case of homogeneous combinators (no negative premises). Thus we consider operators whose combinators are all of the form $(p, a)$, with $p$ having no negative aspect.

(ii) Next we consider how to add similarly restricted n-combinators. This is the heart of the extension to the original construction.

(iii) The next step is to allow actions to throw away active arguments.

(iv) We then allow non-homogeneous combinators, but only ones that use the existing active arguments rather than inactive ones.

(v) The final stage is to show how to use inactive arguments.

In this version of the paper we concentrate on the first two stages. The rest are given in detail in the extended version. At each stage the simulation we build takes the form of the parallel composition of processes representing each argument that is active, plus additional parallel components to regulate behaviour, and “zombie” processes representing those that have been inactivated.

#### 4.1 Homogeneous positive combinators

In this case (in a simplification from the construction in [16,15]), the simulation will take the form

\[
(((\|_{i=1}^{n}(A_{i},P_{i}[R_{i}])) \| [\cup A \| C] \| RUN_{C}\| CR]) \| \{ Tau\})
\]

where $P_{1}, \ldots, P_{n}$ are the (all active) arguments of some operator $F$. $Tau$ is a member of $\Sigma$ we introduce to model a combinator generating a $\tau$ action. $\cup A$ is the union of the $A_{i}$.
Let $C$ be the set of combinators for $F$. We add $C$ into the alphabet and can construct the renamings as follows.

- $R_i$ maps each event $a$ of $P_i$ to each combinator which requires the $i$th argument to perform $a$.
- $CR$ maps the combinator $(p, a)$ to $a$ if $a \in \Sigma$, and to $\text{Tau}$ if $a = \tau$.
- $A_i$ consists of all combinators $c$ which have a proper premise (i.e., not “.”) in position $i$.
- The $\text{RUN}$ process provides a way in which combinators with no active arguments can happen. It is later replaced by more elaborate regulator processes.
- Any $P_i$ that can perform a $\tau$ can perform it in the simulation, with the simulation state progressing exactly as we require in the patience rule that $F$ must have for its $i$th argument.
- The event representing the combinator $c$ can occur precisely when the premises of $c$ are met (i.e., each non-“.” component performs the appropriate event). The renamed $P_i$ can then synchronise to perform $c$, which $CR$ and the hiding of $\text{Tau}$ combine to turn into its own second component. Again the successor state (with just the $P_i$ that contribute to $c$ progressing) is exactly the one that simulates the state that the combinator semantics will have reached under the same action.
- Every state reachable from $F(P_1, \ldots, P_n)$ in our restricted circumstances is of the form $F(P'_1, \ldots, P'_n)$ for $P'_i$ some state of $P_i$, and by the above observations this state is strongly bisimilar – indeed isomorphic in the sense of transition systems – to the following simulation state.

$$((\prod_{i=1}^n (A_i, P'_i[|R_i|])) \mid (\cup A \parallel C \parallel \text{RUN}_C)\parallel CR) \setminus \{\text{Tau}\}$$

We have therefore completed the construction in this first case.

### 4.2 Adding negation

Suppose for the moment that no combinator has both positive and negative premises for the same argument. Then we can get the argument process if necessary to contribute one or other to the firing of the combinator. We know how to achieve this for positive ones. For negative premises we can use priority to deliver an event just when some set of actions is not possible.

For the $S \subseteq \Sigma$ that might (each together with $\{\tau\}$) be negative premises for argument process $P$, let $\neg S$ be a new event that will represent $P$’s inability to perform any of them. Let the set of such $\neg S$ for $P$ (in the context it is placed) be $\text{negs}(P)$. Then $\text{Negate}_0(A, P) = \text{Pri}_{\leq P}(P \parallel \text{RUN}_{\text{negs}(P)})$ where $\neg S <_P a$ if and only if $a \in S$, can perform $\neg S$ when $P$ is in a stable state that cannot perform any member of $S$. We can check a negative premise on $P$ by getting $\text{Negate}_0(P)$ to perform an event as part of a combinator synchronisation whenever that is appropriate. The first component of a combinator now becomes a tuple with components that are either a positive event $a$, a $\neg S$ or the absence “.” of that process’s involvement. The renamings $R_i$ on the components are extended so the $\neg S$ is renamed to each combinator $c$ that has $\neg S$ as a component at the given process’s place.
There are cases with positive and negative premises on the same argument, such as the semantics of the priority operator itself: $\textbf{Pri}_{\leq}(P)$ can only perform non-maximal $a$ if $P$ itself can, but cannot perform any higher priority event. To handle this we use further events: $(a, \neg S)$ (with $a \notin S$) means that the process can perform $a$ while in a stable state where no member of $S$ can happen. The above definition is extended to $\text{Negate}(P) = \textbf{Pri}_{\leq_P}(P\|[\text{NegR}]\| \text{RUN}_{\text{Neg}(P)})$ where $\text{NegR}$ maps each event $a$ in $P$’s alphabet to both itself and the $(a, \neg S)$ we introduced above, and $\leq_P$ is extended so that $(a, \neg S)$ is given the same priority as $\neg S$. Thus $(a, \neg S)$ can happen just in those stable states where $a$ can be performed by $P$ but no member of $S$ can be. To handle this the renaming $R_i$ is extended so that $(a, \neg S)$ is mapped to every combinator $c$ which has this particular pair of premises for its $i$th argument.

4.3 Further stages

The rest of this proof follows similar lines to the one in [16] without priority. This is set out in detail in the extended paper [18]. To handle processes being discarded (as can happen to either argument of $P \triangle Q$ and the first arguments of $P \Theta_A Q$) we place each $\text{Negate}(P_i)$ in a harness, where $P_i$ can be switched off by $\Theta_A$ whether or not $P_i$ itself participates in the (necessarily non-homogeneous) combinator that causes this effect. A strong sense of how this is done is given by the deadlock-exception catching operator $P \Delta \Theta Q$ we described earlier: it can be written $((\textbf{Pri}_{\leq}(P \|[\delta \rightarrow \text{STOP}]))\Theta_{\delta} Q) \setminus \{\delta\}$ for $\delta$ a new event, the only ordering by $\leq$ being to place $\delta$ below all others.

The other aspects of non-homogeneous combinators that need to be handled are (i) allowing the sorts of continuation permitted by the third component syntax $T$ and (ii) activating inactive arguments to participate in the operations of such $T$. The first is achieved by extending the alphabet to include labelled (n-)combinators for every form that the system might evolve to as the simulation progresses, together with a regulator process which understands what the present format of the system is and how the current active argument processes map onto the format’s active arguments.

There are two ways of handling the activation of inactive arguments: one each is described in [16] and [18]. The first dynamically generates new argument processes each time one is activated. The second is possible where the overall number of active arguments has an upper bound, and works by recycling them: letting the zombies created by turning arguments off be reborn in a new form.

All of this can be be done in such a way as to obtain strong bisimulation.

5 Examples

We have seen how to create an operator that allows deadlock in one process to cause a second process to start. The following transformation provides the basis for many similar constructions.

Suppose $P$ has alphabet $\Sigma_0$, and that we have added as follows to the overall alphabet: $\Sigma_1 = \{\neg a \mid a \in \Sigma_0\}$ and $\Sigma_2 = \{\neg\neg a \mid a \in \Sigma_0\}$ (all these new labelled events being different to each other and members of $\Sigma_0$).
Now define two partial orders on $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$: $\neg a <_1 a$ and $\neg \neg a <_2 \neg a$ for each $a \in \Sigma_0$, with no other pairs ordered except for those required to make $\tau$ maximal. Let us now think through the behaviour of the process

$$Probe(P) = \text{Pri}_{\leq_2}(\text{Pri}_{\leq_1}(P \parallel RUN_{\Sigma_1 \cup \Sigma_2}))$$

(i) The process inside the priority operators can perform any action of $P$, and also always perform any action in $\Sigma_1 \cup \Sigma_2$ without changing state.

(ii) The result of the inner priority operator can still perform any action of $P$ and any member of $\Sigma_2$, but can now only perform $\neg a \in \Sigma_1$ when $P$ itself is in a stable state than cannot perform $a$.

(iii) $Probe(P)$ can perform the same members of $\Sigma_0 \cup \Sigma_1$ as at stage 2, but can now only perform $\neg a \in \Sigma_2$ when $P$ is in a stable state which can perform $a$. When this process performs either $\neg a$ or $\neg \neg a$, its state does not change. This gives the observer, by viewing events alone, the ability to “probe” what events the current state of $P$ can perform.

Building on this, we can for example create a stronger version of the angelic choice operator $\boxplus$ of [15]: $P \boxplus_N Q$ behaves like $P \square Q$ except that when $P$ and $Q$ offer the same visible event $a$ the choice between them is delayed rather than forced when $a$ occurs. The operational semantics of $P \boxplus_N Q$ can only perform a visible event $a$ if either both $P$ and $Q$ perform it in parallel, or if one of them does perform it and the other one cannot.

This is implemented by running $\hat{P} \Theta_{\Sigma_1} RUN_{\Sigma_0}$ alongside $\hat{Q} \Theta_{\Sigma_1} RUN_{\Sigma_0}$. Here, $\hat{P}$ is $Probe(P)$ without the $\neg \neg a$ events, and in the combination $a \in \Sigma_0$ can synchronise with either itself or $\neg a$, in each case creating the external event $a$. For full details see the the extended version.

6 Comparisons

The good comparator for CSP-like operational semantics is van Glabbeek’s [10] concept of simply WB-cool operational semantics. This is more liberal than CSP-like because it permits cloning and because it explicitly allows arbitrary probing of active arguments: it allows multiple premises of the form $P \xrightarrow{a} P'_a$ for different $a$s, and we can choose to use either the results $P'_a$ or the original $P$ in the result term. Van Glabbeek also allows active arguments to become inactive under limited circumstances. The restrictions there are all expressed in the language of SOS. Van Glabbeek shows that such semantics ensure congruence under weak bisimulation. [10] also introduces some variants on WB-cool which have congruence properties for different forms of bisimulation.

Both CSP-like and Pri-CSP-like operational semantics (like strictly WB cool and similar classes) come firmly within the GSOS class of operational semantics defined in [3]. This is well studied, and implies, for example [4], that the use of negative premises causes no problems with the well-definedness of operational semantics.

---

5 Van Glabbeek’s work was itself closely related to work by Bloom and others [1].
A very detailed survey of restrictions on SOS semantics which are intended to preserve various forms of congruence is provided in [2]. This identifies full probing – namely the ability to test the complete acceptance/ready set as a condition for actions – with the natural notion of operational semantics which coincides with the $\mathcal{FL}$ style of model, there termed ready-trace. The main construction in Section 5, elaborated on in the extended version of this paper, shows that the same can be done in Pri-CSP. Whereas CSP-like operators are a closed world in the sense that any composition of CSP-like operators is also describable in combinators, the same is not true of Pri-CSP-like and (n)-combinators, helping to explain why one can go beyond direct expressibility in terms of these by composing operators that are. This is why the continuation syntax $T$ is more restricted when negative premises are allowed.

7 Conclusions

One of the most interesting features of this work is the great expressive power of $\text{Pri}_{\leq}(\cdot)$ in conjunction with ordinary CSP.

In a future paper by the author and others, we will show how refinement checking over a wide variety of CSP models can be reduced, using priority, to trace refinement.

It is reasonable to ask how crucial the choice of priority is for an extra operator to achieve the degree of expressiveness seen here. Clearly such an operator cannot be CSP-like, and must have the property that $\text{Pri}_{\leq}(\cdot)$ is expressible using it and the rest of CSP. It cannot have a semantics in any CSP model where $\text{Pri}_{\leq}(\cdot)$ does not, for it must be able to express priority. We pose this as a question for further work.

Acknowledgements

This work was done under funding from the DARPA HACMS program. It has benefited hugely from discussions with Rob van Glabbeek, Tom Gibson-Robinson, Augusto Sampaio and David Mestel.

References


Open maps in concrete categories and branching bisimulation for prefix orders

H. Beohar¹,²

No affiliation

P.J.L. Cuijpers³

Dep. of Mathematics and Computer Science
Technische Universiteit Eindhoven
Eindhoven, The Netherlands

Abstract

Open maps, as introduced in concurrency theory by Joyal, Nielsen and Winskel, provide an abstract way to define functional bisimulations across a wide variety of models of computation (like labelled transition systems, event structures, etcetera). Furthermore, the existence of a span of open maps characterises the well-known relational definition of bisimulations found in the literature associated with these models of computation. However, in our working category of prefix orders (in which the objects represent the sets of executions generated by arbitrary dynamical systems) the open maps do not immediately result in functional bisimulations and the existence of a span of open maps does not result in an equivalence. This is rather surprising, since prefix orders are mere generalizations of (discrete) execution trees, for which the open map approach is known to work. After taking a closer look at the definition of open map, we show in this paper that the issue can be remedied by considering prefix orders as a concrete category and reinterpreting the definition of open-map in this light. As a bonus, the choice of a path-category on which the notion of open-map relies becomes a natural one, namely the subcategory of embeddings. While the existence of spans still does not result in an equivalence, it is shown that the existence of cospans does. In fact, we present a characterisation of the notion of branching bisimulation by van Glabbeek and Weijland which, to the best of our knowledge, was not studied in the framework of open maps before.

Keywords: Open maps, Prefix orders, Branching bisimulation, Concrete categories.

1 Introduction

Since van Glabbeek’s work [20] on comparative concurrency semantics, we are aware of the many ways in which different states of a labelled transition system can be considered behaviourally equivalent (resulting in the well-known van Glabbeek spectrum). In their seminal paper [14], Joyal, Nielsen and Winskel proposed an abstract definition of strong bisimulation using the language of category theory and thus

¹ Thanks goes to Paul Taylor for providing his macro to neatly draw commutative diagrams in L\(\text{TEX.}\)
² Email: harshbeohar@gmail.com
³ Email: p.j.l.cuijpers@tue.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Beohar and Cuijpers embarked a way to capture behavioural equivalences in a uniform framework. In particular, bisimilarity through spans (cospans) of open maps is defined as the existence of a span (cospan) of open maps between two objects, where a map \( o \) of \( \mathcal{M} \) is open (denoted \( \dashv \) ) whenever, for any map \( p \) in a subcategory \( \mathcal{P} \) (denoted \( \subset \) ) and maps \( s \) and \( m \) from \( \mathcal{M} \) such that the outer square commutes (i.e., \( s \cdot p = k \cdot m \)), there exists a map \( k \) in \( \mathcal{M} \) (existence emphasized by dashing the arrow) making the two inner triangles commute (i.e., \( k \cdot p = m \) and \( o \cdot k = s \)). These arrow-notations will be overloaded later in an obvious way, when discussing the concrete categorical variant of openness.

Taking \( \mathcal{M} \) as the category of labeled transition systems with transition preserving maps between them, and \( \mathcal{P} \) as the category of path-extensions (containing all transition preserving maps between chains of transitions) Joyal et al. showed that bisimilarity through spans of open maps coincides with the familiar notion of strong bisimulation from concurrency theory [15]. Subsequently, bisimilarity through spans or cospans of open maps has been shown to coincide with useful notions of bisimulation in many alternative models of behavior as well (see, e.g., [3,9,11,12]).

Despite the generality offered by the open map framework [13], it suffers from two limitations. Firstly, there is as yet no uniform treatment of weak equivalences from the van Glabbeek spectrum (see [17]). Most work on weak equivalences deals with the notion of weak bisimulation (e.g. [3,9]) and seems to rely on first saturating (merging) the so-called invisible steps of the transition systems under study and then instantiating the strong bisimilarity on the saturated versions. As it is well known from [19], such a saturation method of the invisible steps is not sound with respect to branching bisimulation equivalence; thus, the techniques developed in [3,9] fall short in characterising branching bisimulation equivalence. Secondly, in order for bisimilarity through spans of open maps to result in an equivalence, the category \( \mathcal{M} \) must have pullbacks, which can be a difficult condition to obtain (see [7,16]).

Surprisingly, in our own research [4,5] on describing behavioural systems as prefix ordered sets of executions, the definitions of branching bisimulation arose naturally via a different path, but we had trouble to apply the open map framework of [14] even for strong bisimulation. To be precise, there was no suitable choice of the subcategory of paths such that open maps would result in the usual notion of functional bisimulation (cf. [14, Proposition 1]).

In our attempts to remedy this, we discovered that reinterpreting the above diagram in the context of concrete categories leads to many new insights. Firstly, it allows us to define functional bisimulations in the concrete category of prefix orders with partial orders as a base category. Secondly, the natural choice for the subcategory \( \mathcal{P} \) of paths turns out to be the subcategory of embeddings. And thirdly,
the existence of cospans of these embedding-open maps turns out to coincide with branching bisimulation when the prefix orders are the sets of runs generated by a labelled transition system. Finally, the reinterpretation comes with a flavour of syntax and semantics, in which we consider morphisms in the concrete (base) category to be implementations (observations).

The approach we take in this paper towards re-evaluating an existing categorical notion is not uncommon. In many categories, for example, the notion of monomorphism is an effective way to capture the idea of a ‘subobject’, but in concrete categories it often turns out that the notion of ‘embedding’ (a special kind of monomorphism) is to be preferred. In a similar vein, we hope that our adaptation of the notion of open map will result in a more widely applicable notion of ‘reflecting extensions’. The fact that we obtain embeddings as a natural choice for the subcategory of path-extensions is a hint that this may indeed be the case.

In the next section, we re-introduce the notion of open maps in the context of concrete categories. We re-emphasize the fact that preservation of open maps by pull-backs or push-outs is sufficient to guarantee that bisimilarity through spans or cospans, respectively, is an equivalence, and we show that in case there are ‘enough’ open maps, there is an alternative characterization of open map that may appeal to the reader’s intuitions on bisimulation. Sadly, despite some effort, we did not find a nice categorical characterization under which pull-backs and push-outs preserve open maps, so this will still remain to be proven for each category separately. In particular, the result of [14] that the existence of pullbacks suffices to guarantee equivalence does not carry over to the concrete setting straightforwardly. In section 4, we recall the category of prefix orders from [5,4,8] and prove that embedding-open maps are functional bisimulations and that pushouts preserve open maps in this category so that bisimulation through cospans is an equivalence. Finally, in section 5 we prove that the existence of cospans in fact coincides with branching bisimulation if the executions in the prefix order are generated by labeled transition systems. In section 6 we give some concluding remarks and suggestions for future.

2 Open maps in concrete categories

Let us start by adapting the notion of open map from [14] in the setting of concrete categories [1]. For this, we begin by recalling the following fundamental definition.

**Definition 2.1** A category $\mathcal{P}$ is a subcategory of $\mathcal{M}$ if every morphism or object of $\mathcal{P}$ is a morphism or object of $\mathcal{M}$, respectively. A category $\mathcal{M}$ is concrete over a base category $\mathcal{S}$ if there exists a faithful functor $\mathcal{M} \overset{\cong}{\rightarrow} \mathcal{S}$, i.e., for any two morphisms $f, g : \mathcal{A} \rightarrow \mathcal{B}$ from $\mathcal{M}$ we find that $|f| = |g|$ implies $f = g$.

In this paper, by ‘a concrete category with faithful functor $\mathcal{M} \overset{\cong}{\rightarrow} \mathcal{S}$’, we tend to think of the objects of $\mathcal{M}$ as concrete models of behavior and the objects of $\mathcal{S}$ as semantic models of behavior. A morphism $\mathcal{A} \overset{f}{\rightarrow} \mathcal{B}$ of $\mathcal{M}$ represents a way of implementing the behavior of $\mathcal{B}$ (the specification) as a behavior $\mathcal{A}$ (the implementation). On the semantic level, a morphism $\mathcal{C} \overset{\varphi}{\rightarrow} \mathcal{D}$ describes how the behavior of $\mathcal{C}$ can be observed as a part of the behavior of $\mathcal{D}$.
Furthermore, in [14], the objects of the subcategory $\mathcal{P}$ of $\mathcal{M}$ represented a collection of path-extensions which needed to be preserved, but in this paper we simply think of them as arbitrary behavioral extensions, achieved through implementation. We emphasize that extensions are concrete models, meaning intuitively that we are only interested in preserving behavior that can be described as objects of $\mathcal{M}$ and not necessarily in all semantic behavior that may occur in $\mathcal{S}$. Informally, a $\mathcal{P}$-open map $X \xrightarrow{o} Y$ says that if any observed behavior of $X$ has a concrete extension of interest (i.e. from $\mathcal{P}$) observed in $Y$, then this extension can be observed in $X$ as well (although it is not necessarily part of the implementation of $X$). Thus, in short, $o$ reflects all the concrete extensions from $\mathcal{P}$.

![Diagram of bisimulation through spans and cospans of open maps](image)

**Definition 2.2** [Open map] Given a concrete category with faithful functor $\mathcal{M} \xrightarrow{\mathcal{S}} \mathcal{S}$ and subcategory $\mathcal{P}$ of $\mathcal{M}$, a map $o$ from $\mathcal{M}$ is $\mathcal{P}$-open (denoted $\xrightarrow{\mathcal{P}}$) if for every $p$ from $\mathcal{P}$ and maps $m, s$ from $\mathcal{S}$ such that $s \cdot |p| = |o| \cdot m$ (i.e. the outer square in diagram 1a commutes in $\mathcal{S}$) there exists a map $k$ from $\mathcal{S}$ such that $k \cdot |p| = m$ and $|o| \cdot k = s$ (i.e. the inner triangles in diagram 1a commutes as well). Two objects $A$ and $B$ from $\mathcal{M}$ are $\mathcal{P}$-bisimilar through spans, denoted $A \equiv^P B$, if there exists a span of $\mathcal{P}$-open maps between them, i.e. if there exists an object $C$ in $\mathcal{M}$ and $\mathcal{P}$-open maps $C \xleftarrow{f} A$ and $C \xrightarrow{g} B$, as in Figure 1b. They are $\mathcal{P}$-bisimilar through cospans, denoted $A \equiv^c P B$, if there exists a cospan of $\mathcal{P}$-open maps between them, i.e. if there exists an object $C$ in $\mathcal{M}$ and $\mathcal{P}$-open maps $A \xleftarrow{f} C$ and $B \xrightarrow{g} C$, as in Figure 1c.

Obviously, our concrete definition coincides with the one in [14] when $\mathcal{M} = \mathcal{S}$ and the faithful functor is the identity. This means that the only ‘new’ part in our definition is the distinction between the use of concrete and base maps. Just like in [14], open maps form a subcategory of $\mathcal{M}$.
Theorem 2.3 Given a concrete category with faithful functor $\mathcal{M} \rightarrow \mathcal{S}$ and subcategory $\mathcal{P}$ of $\mathcal{M}$, every identity in $\mathcal{M}$ is $\mathcal{P}$-open, and if $A \circ_{\mathcal{P}} B$ and $B \circ_{\mathcal{P}} C$ are $\mathcal{P}$-open maps, then so is their composition $q \cdot p$.

Moreover, if pullbacks (pushouts) preserve open maps than bisimulation through spans (cospans) is an equivalence, which follows easily from diagram chasing.

Definition 2.4 Given any category $\mathcal{M}$ and a cospan $X \xleftarrow{f} Z \xrightarrow{g} Y$, a span $X \xleftarrow{h} P \xrightarrow{k} Y$ is a pullback if $f \cdot h = g \cdot k$ and for any span $X \xleftarrow{h'} Q \xrightarrow{k'} Y$ with $f \cdot h' = g \cdot k'$ there exists a unique map $Q \xrightarrow{u} P$ such that $h \cdot u = h'$ and $g \cdot u = g'$. Dually, given a span $X \xrightarrow{h} P \xrightarrow{k} Y$, a cospan $X \xrightarrow{h} Z \xrightarrow{g} Y$ is a pushout if $h \cdot f = k \cdot g$ and for any span $X \xrightarrow{h'} Q \xrightarrow{k'} Y$ with $h' \cdot f = k' \cdot g$ there exists a unique map $P \xrightarrow{u} Q$ such that $u \cdot h = h'$ and $u \cdot g = g'$.

Theorem 2.5 (Bisimulation equivalence through spans or cospans) In a concrete category with faithful functor $\mathcal{M} \rightarrow \mathcal{S}$ bisimulation through spans is an equivalence if every cospan of open maps $X \xleftarrow{f} Z \xrightarrow{g} Y$ has a pullback $X \xleftarrow{h} P \xrightarrow{k} Y$ consisting of open maps. Dually, bisimulation through cospans is an equivalence if every span of open maps has a pushout of open maps.

Joyal et al. showed that in a category where all cospans have pullbacks also all open cospans have open pullbacks [14]. Therefore, in the original definition, bisimilarity through spans is an equivalence in all categories that have pullbacks. Conversely, existence of pushouts is not sufficient to guarantee that bisimilarity through cospans is an equivalence.

In our new interpretation over concrete categories, this result is not so easily repeated. We have found some conditions under which bisimilarity through spans is an equivalence, but they involve the existence of retracts in the base category and were not very ‘elegant’. Furthermore, it turns out that our working category of choice, prefix orders with partial orders as a base category, does not satisfy those properties. Therefore we left those results out of the current presentation. In fact, in Section 4 we show that bisimilarity through spans turns out not to be an equivalence at all, even though pullbacks do exist in this category. On the other hand, bisimilarity through cospans does turn out to be an equivalence, even though we do not have a fully category theoretic proof for this, yet.

3 Embeddings as path extensions

In concrete categories, the natural notion of ‘extension’ is usually considered to be that of an embedding. And as it turns out, the subcategory of embeddings is a natural choice for the category $\mathcal{P}$.

Definition 3.1 Given a concrete category $\mathcal{M} \rightarrow \mathcal{S}$, a map $A \xleftarrow{f} B$ from $\mathcal{M}$ is an embedding, and $B$ is called an extension of $A$ if:

- (underlying monos) for all maps $g, h$ of $\mathcal{S}$ with $|f| \cdot g = |f| \cdot h$ we find $g = h$;
• (initial) for any \( g \) of \( S \) and \( h \) of \( M \) with \(|h| = |f| \cdot g\) there is a \( \hat{g} \) of \( M \) with \(|\hat{g}| = g\).

We will denote embedding-bisimilarity through spans simply by \( \equiv_s \) and embedding-bisimilarity through cospans by \( \equiv_c \).

In [1], embeddings where also called extensions, which lead us to research the possibility of using embeddings as the subcategory of path-extensions. But we were convinced of being on the right track when we discovered the similarity between the definition of \( \mathcal{P} \)-open map and that of \( \mathcal{P} \)-injective objects defined in [1]. In particular, the observation in that an object is an absolute retract if and only if it is an injective object in any category that has enough injective objects (see Proposition 9.10 in [1] for the original definitions) turned out to have a nice translation to the setting of open maps as well.

The following definitions and theorems are direct adaptations of those in [1], and the proofs are completely analogous.

**Definition 3.2** Given a concrete category \( \mathcal{M} \Rightarrow S \) and subcategory \( \mathcal{P} \) of \( \mathcal{M} \), we say that \( \mathcal{M} \) has enough \( \mathcal{P} \)-open maps if for every map \( X \xrightarrow{f} Z \) there exists an extension \( X \xleftarrow{p} Y \) from \( \mathcal{P} \) and a \( \mathcal{P} \)-open map \( Y \xrightarrow{o} Z \) such that \( o \cdot p = f \).

**Definition 3.3** Given a concrete category \( \mathcal{M} \Rightarrow S \) and subcategory \( \mathcal{P} \) of \( \mathcal{M} \), a map \( X \xrightarrow{f} Y \) has absolute \( \mathcal{P} \)-retractability if for any extension \( X \xleftarrow{p} Z \) and map \( |Z| \xrightarrow{g} |Y| \) of \( S \) with \( g \cdot |p| = |f| \), there is a map \( |Z| \xrightarrow{r} |X| \) of \( S \) such that \( r \cdot |p| = id \) and \( |f| \cdot r = g \).

**Theorem 3.4** In a concrete category \( \mathcal{M} \Rightarrow S \) and subcategory \( \mathcal{P} \) of \( \mathcal{M} \), if \( \mathcal{M} \) has enough \( \mathcal{P} \)-open maps then a map \( X \xrightarrow{f} Y \) is \( \mathcal{P} \)-open iff it is absolute \( \mathcal{P} \)-retractable.

Interpreting this theorem using the view on concrete categories that concrete morphisms represent implementations and base morphisms represent observations, and especially considering \( \mathcal{P} \) as the subcategory of embeddings, we see that in a category with enough open maps, a map \( A \xrightarrow{f} B \) is open if and only if every extension of the behavior of \( A \) that is observable within \( B \) is already observable within \( A \). In other words, the behavior of \( A \) is a complete representation of all observable behavior in \( B \).
Another nice result on our concrete definition of open map, is that all open maps are epimorphisms under the assumption that the faithful functor preserves epimorphisms and the initial object of a category. In fact, all open maps are retracts in the base category.

**Definition 3.5** Given a category $\mathcal{M}$, a map $e$ is an epimorphism (denoted $\xrightarrow{e}$) if for any pair of maps $g \xrightarrow{f} C$ with $f \cdot e = g \cdot e$ we find $f = g$.

**Definition 3.6** Given a category $\mathcal{M}$, an object $0$ is called initial if for any object $X$ there is a unique map $0 \xrightarrow{} X$. A concrete category with faithful functor $\mathcal{M} \xrightarrow{} \mathcal{S}$ preserves the initial object of $\mathcal{M}$ if $|0|$ is also initial for $\mathcal{S}$.

**Definition 3.7** Given a concrete category $\mathcal{M} \xrightarrow{} \mathcal{S}$ a map $X \xrightarrow{f} Y$ in $\mathcal{M}$ is a base retract if there exists a map $|Y| \xrightarrow{k} |X|$ in $\mathcal{S}$ such that $|f| \cdot f^\leftarrow = id$.

**Theorem 3.8** (Embedding-open maps are base retracts) Given a concrete category $\mathcal{M} \xrightarrow{} \mathcal{S}$, if $\mathcal{M}$ has an initial object $0$ and the faithful functor preserves it, then every embedding-open map is a base retract and an epimorphism.

**Proof.** For any embedding-open map $X \xrightarrow{f} Y$ consider $|0| \xrightarrow{n} |X|$, $m = id$ and $|0| \xrightarrow{p} |Y|$. Since $|0|$ is initial it has no incoming arrows except isomorphisms, hence $p$ is an embedding. This gives us a map $|Y| \xrightarrow{k} |X|$ that makes the diagram in figure 1a commute, hence $k$ is a base retract for $f$ in $\mathcal{S}$, and because any retract is an epimorphism, $f$ is an epimorphism in $\mathcal{S}$. Finally, for any $Y \xrightarrow{h} Z$ with $g \cdot f = h \cdot f$ we find $|g| \cdot |f| = |g \cdot f| = |h \cdot f| = |h| \cdot |f|$, so $|h| = |g|$ and by faithfulness of $|.|$ we have $h = g$. Therefore $f$ is an epimorphism. 

## 4 Prefix orders

In [4,5] we argued that any dynamical system, be it discrete, continuous, or hybrid, can be considered as a set of executions under their natural prefix ordering, and we showed how notions like refinement, bisimulation, and asynchronous product arise naturally as history preserving maps, cospans of history and future preserving maps, and the categorical product on history preserving maps. We decided to choose history preserving maps as the natural notion of morphism for this category because of these results.

**Definition 4.1** A partial order $(\mathcal{U}, \leq)$ is a set $\mathcal{U}$ equipped with a reflexive, transitive and anti-symmetric relation $\leq \subseteq \mathcal{U} \times \mathcal{U}$. A prefix order is a partial order satisfying:

- **downward total:** $\forall x, y, z \in \mathcal{U}$ ($x \leq z$ \& $y \leq z$) $\Rightarrow$ ($x \leq y$ \& $y \leq x$).

A function $f : \mathcal{U} \rightarrow \mathcal{V}$ between two partial orders is order preserving if for all $x \leq y$ we find $f(x) \leq f(y)$, and we denote the category of partial orders with order preserving maps between them by $\textbf{Pos}$. Furthermore, we write $x^- = \{y \mid y \leq x\}$ for the downward closure, or history, of a point $x \in \mathcal{U}$ in a partial or prefix order, and
write \( f(A) = \{ f(a) \mid a \in A \} \) to lift a function to sets. Then, a function \( f : U \to V \) between prefix orders \( U, V \) is said to be

- **history preserving** if \( \forall x \in U \ f(x)^- = f(x^-) \).

We write \( \textsf{Pfx} \) for the category of prefix orders and history preserving functions between them.

In order to define functional bisimulation on prefix orders without the use of category theory, we look at paths and their continuations explicitly.

**Definition 4.2** A subset \( P \subseteq U \) is a **path** in a prefix order \( U \) if

- \( P \) is a **chain**, i.e., \( \forall_{x,y \in P} \ x \preceq y \lor y \preceq x \), and
- \( P \) is **prefix closed**, i.e., \( \forall_{x \in P} x^- \subseteq P \)

Furthermore, a map \( f : U \to V \) between prefix orders is a **functional bisimulation** if

- \( f \) is history preserving, and
- for any path \( P \subseteq U \) and \( v \in V \) with \( f(P) \subseteq v^- \), there exists a \( u \in U \) with \( P \subseteq u^- \) and \( f(u) = v \).

Note that in case of transfinite executions, this definition also relates Zeno-points and other limit behavior, generalizing the solution of [6] to a problem widespread in the study of e.g. timed and hybrid systems [18,10]. Also note, that this definition slightly differs from the one in [5] in which Zeno-choices were not explicitly taken into account yet. However, for executions of the more usual models of computations, such as labeled transition systems, they coincide.

**Counterexample bisimulation in \( \textsf{Pfx} \):**

One of the concerns that were left in the categorical treatment of prefix orders was that functional bisimulations could not (yet) be described in terms of history preserving maps alone. The reason for this, turns out to be that the diagonal \( k \) in diagram 1a exists, but is merely order preserving and not history preserving. A simple example of this is the map \( o : \{0, 1, 2\} \to \{0, 1\} \) shown in figure 3. This is a functional bisimulation according the definition above, but also according to most other definitions of functional bisimulation that have been proposed in literature. It is also history preserving, hence a morphism in \( \textsf{Pfx} \), and taking \( B = \{0, 1\} \) and \( A = \{0\} \) gives us a commuting square of history preserving maps. Nevertheless, no history preserving diagonal \( k \) can exist for this square. The fact that there does exists an order preserving \( k \) in this case lead us to investigate the possibility of a concrete category theoretic approach.

![Fig. 3. A history preserving functional bisimulation \( o \) without history preserving diagonal \( k \).](image-url)
Counterexample equivalence through spans:

A second concern was that using spans of functional bisimulations to define bisimulation relations between prefix orders does not lead to an equivalence. An example of that was already mentioned in [5], based on the set $\neg\Omega$ of all ordinal numbers upto (but not including) the first uncountable ordinal, ordered in the opposite direction, and the set $\neg\mathbb{N}$ of all natural numbers ordered in the opposite direction. The maps $\neg\mathbb{N} \to 1$ and $\neg\Omega \to 1$ are both functional bisimulations, thus witnessing that $\neg\mathbb{N}$ is bisimilar to 1 and $\neg\Omega$ is bisimilar to 1 (the witnessing spans are those maps and the identities). Nevertheless, the pullback of these two maps, which is supposed to be a witness for bisimilarity of $\neg\mathbb{N}$ and $\neg\Omega$, turns out to be the empty set $\emptyset$. There is no other prefix order possible that has order preserving surjections into both sets, simply because there is no order preserving surjection from $\neg\Omega$ to $\neg\mathbb{N}$. Indeed, the pullback does exist, but the maps $\emptyset \to \neg\Omega$ and $\emptyset \to \neg\mathbb{N}$ are certainly not functional bisimulations (they are not even surjective). We conclude that a span of functional bisimulations between $\neg\mathbb{N}$ and $\neg\Omega$ cannot exist, hence they are not bisimilar through spans.

In the previous section, we have paved a way of defining bisimulation through cospans categorically. What is left to show, is that the embedding-open maps indeed give the notion of bisimulation that we would like to have. In order to do this, we consider prefix orders as a concrete category over the category of partial orders.

In the following theorem, we compile a list of interesting facts pertaining to the categories $\text{Pfx}$ and $\text{Pos}$. The proof of each items are either standard or trivial.

**Theorem 4.3**

(i) Every history preserving function is order preserving, hence the identity functor $\text{Pfx} \xrightarrow{\text{id}} \text{Pos}$ which maps every prefix order and history preserving map to itself is a faithful functor.

(ii) The embeddings in the concrete category $\text{Pfx} \xrightarrow{\text{id}} \text{Pos}$ are precisely the injective history preserving functions.

(iii) The empty set $\emptyset$ is the initial object of $\text{Pfx}$ and $\text{Pos}$.

(iv) The surjective history preserving functions are precisely the epimorphisms of $\text{Pfx}$, and are epimorphisms in $\text{Pos}$ as well.

(v) All pushouts and pullbacks exist in the category $\text{Pfx}$.

Next, we show that the open maps in $\text{Pfx} \xrightarrow{\text{id}} \text{Pos}$ are precisely the functional bisimulations.

**Theorem 4.4** A map $o : \mathbb{U} \to \mathbb{V}$ of $\text{Pfx}$ in the concrete category $\text{Pfx} \xrightarrow{\text{id}} \text{Pos}$ is embedding-open if and only if it is a functional bisimulation.

**Proof.** That every open map is a functional bisimulation is straightforward to prove, by taking any path $P$ from the definition of functional bisimulation as the object $A$ and $P \cup \{v\}$ as $B$ in diagram 1a. The resulting map $k$ will point out $u = k(v)$, and order preservation of $k$ gives $P \subseteq u^-$ and $o(u) = v$. The reverse direction, however, is less trivial and requires the set-theoretic axiom of choice.

Let $o$ be a map of $\text{Pfx}$ such that for any path $P$, any $v \in \mathbb{V}$ with $o(P) \subseteq v^-$ there is a $u \in \mathbb{U}$ with $P \subseteq u^-$ and $o(u) = v$. Then, it is not hard to verify (using order
This concludes the proof that \( \forall v \ o(P) \prec v' \preceq v \implies \exists u' \ P \prec u' \preceq u \wedge o(u') = v' \). \hfill (*)

Here, \( P \preceq u' \iff \forall u \in P \ u \preceq u' \). Now, consider the commutative square of figure 1a.

We need to find an order preservation function \(|B| \xrightarrow{k} |U|\) such that \( k \cdot |p| = m \) and \(|o| \cdot k = s\). This we do by induction over prefix closed subsets of \( \mathbb{B} \). We prove that such a function exists for \( p(\mathbb{A}) \subseteq \mathbb{B} \). And subsequently we will prove for any subset \( p(\mathbb{A}) \subseteq X \subseteq \mathbb{B} \) with \( X^- \subseteq X \) and any point \( b \in \mathbb{B} \), that if there exists an order preserving function \( k_X : X \to U \) satisfying \( k_X \cdot p = m \) and \(|o| \cdot k_X = s\), then there exists a set \( p(\mathbb{A}) \subseteq X \subseteq Y \subseteq \mathbb{B} \) with \( Y^- = Y \) and \( b \in Y \) for which a similar function \( k_Y \) exists and furthermore \( k_X(x) = k_Y(x) \) for \( x \in X \). It is a standard category theoretic result that the limit of an infinite series of such order preserving maps results in such an order preserving map again, and therefore proving the hypothesis above suffices to conclude that \(|B| \xrightarrow{k} |U|\) exists.

**Base case** Pick \( X = p(\mathbb{A}) \) and define \( k_X = m \cdot p^{-1} \). Note that \( p^{-1} \) is an order preserving function because \( p \) is an embedding, and by construction \(|o| \cdot k_X = \hat{s}_X \).

**Inductive case** Pick \( p(\mathbb{A}) \subseteq X \subseteq \mathbb{B} \) with \( X^- = X \) and any point \( b \in \mathbb{B} \), and let \( k_X \) be satisfying \( k_X \cdot p = m \) and \(|o| \cdot k_X = s_X \). If \( b \notin X \), we pick \( Y = X \) and we are done. So let \( b \notin X \) and construct the path \( P = k_X(b^- \cap X^-) \subseteq U \) to find \( o(P) \preceq s(b) \). Condition (*) gives a point \( u \) such that \( P \preceq \{u\} \), \( o(u) = s(b) \), and

\[ \forall u' \ o(P) \prec s(b') \preceq s(b) \implies \exists u' \ P \prec u' \preceq u \wedge o(u') = s(b'). \]

Note that, for any \( b' \) satisfying the above antecedent, there may be more than one \( u' \) satisfying the above condition, but applying the set theoretic axiom of choice we may still construct a function \( g : b^- \to u^- \) such that \( g(b) = u \) and \( o(g(b')) = s(b') \). As a second stage, we use a quotient construction to construct a function \( \hat{g}_X : b^- \to u^- \) such that for every \( b, b' \in \mathbb{B} \) with \( s(b) = s(b') \) we have \( \hat{g}_X(b) = \hat{g}_X(b') \) and there exists a \( b'' \) such that \( s(b) = s(b'') \) and \( \hat{g}_X(b'') = g(b'') \). Finally, we may define \( Y = X \cup b^- \) and merge \( k_X \) and \( \hat{g}_X \) to find a function \( k_Y \) such that \( k_Y(y) = k_X(y) \) if \( y \in X \) and \( k_Y(y) = \hat{g}_X(y) \) if \( P \prec y \preceq b \). Note that \( Y \) is prefix closed and \( k_Y \cdot p = m \) and \(|o| \cdot k_Y = s_Y \) satisfies by construction; so it remains to show \( k_Y \) is order preserving.

For this, pick \( y, y' \in Y \) with \( y \preceq y' \). We distinguish the following cases.

- **\( y, y' \in X \)**: trivial since \( k_X \) is order preserving;
- **\( y \notin X \) and \( y' \in X \)**: leads to a contradiction since \( X = X^- \);
- **\( y \in X \) and \( y \notin X \)**: because \( y \preceq y' \) we find \( k_Y(y) = k_X(y) \in P \) and by construction \( P \prec \hat{g}(y') = k_Y(y') \);
- **\( y, y' \notin X \)**: Then \( k_Y(y), k_Y(y') \preceq k_X(b) \) and by downward totality we have either \( k_Y(y) \preceq k_Y(y') \) (in which case we are done) or \( k_Y(y') \preceq k_Y(y) \). In the latter case we find \( s(y) \preceq s(y') = o(k_Y(y')) \preceq o(k_Y(y)) = s(y) \) hence \( s(y) = s(y') \) hence \( \hat{g}_X(y) = \hat{g}_X(y') \) hence \( k_Y(y) = k_Y(y') \).

This concludes the proof that \( o \) is an open map. \( \Box \)

Finally, we already have seen that bisimulation through spans will not result in an equivalence, with the pullback of \( -\Omega \to 1 \leftarrow -\mathbb{N} \) as an example of a
pullback that does not preserve open maps. In order to prove that pushouts do preserve open maps, we first need to recall a few notions and theorems regarding the construction of pushouts.

**Definition 4.5** Given a prefix order $U$, an equivalence relation $\sim \subseteq U \times U$ is

- **Order contracting** if: $\forall u,v,w,x \in U \; u \preceq v \land v \sim w \land w \preceq x \land x \sim u \Rightarrow u \sim v$.

**Theorem 4.6** Every morphism $f: U \to V$ defines an order contracting equivalence on $U$ given by $u \sim u' \iff f(u) = f(u')$. Furthermore, the equivalence classes $U/ \sim$ are prefix ordered by defining $[u]_\sim \subseteq [u']_\sim$ iff $\exists w, w' u \sim w \preceq w' \sim u'$, and the projection $[.]_\sim: U \to U/ \sim$ is an epimorphism.

**Theorem 4.7** Given two prefix orders $Y$ and $Z$, their coproduct is the disjoint union $Y \uplus Z$ of the underlying sets, prefix ordered by the relation $\subseteq$ such that $a \subseteq b$ if and only if $a \preceq b$ and $a, b \in Y$, or $a \preceq b$ and $a, b \in Z$.

**Theorem 4.8** Given two history preserving maps $X \xrightarrow{f} Y$ and $X \xrightarrow{g} Z$, their pushout is the set $(Y \uplus Z)/ \sim$ where $\sim$ is the smallest order contracting equivalence such that $\forall x \in X f(x) \sim g(x)$, and the natural projections are given by $[.]_\sim: Y \to (Y \uplus Z)/ \sim$ and $[.]_\sim: Z \to (Y \uplus Z)/ \sim$.

**Theorem 4.9** In the concrete category $\text{Pfx} \xrightarrow{|.|} \text{Pos}$ pushouts preserve open maps (hence bisimulation through cospan of open maps is an equivalence).

**Proof.** Given two open maps $X \xrightarrow{f} Y$ and $X \xrightarrow{g} Z$, we know their pushout is the set $(Y \uplus Z)/ \sim$ where $\sim$ is the smallest order contracting equivalence such that $\forall x \in X f(x) \sim g(x)$. To see that the map $[.]_\sim: Y \to (Y \uplus Z)/ \sim$ is open (and similarly for $Z$), pick $P \subseteq Y$ a path in $Y$ and $[P] \subseteq v^-$ for some $v \in (Y \uplus Z)/ \sim$. So there exists either a $y \in Y$ or $z \in Z$ with $[y] = v$ or $[z] = v$ respectively. Furthermore, if $z$ exists we use the fact that $g$ is open hence has a base retract to find $y = f(g^+(z)) \sim z$ hence $[y] = v$. Because taking equivalence classes is order preserving, either $P \subseteq y^-$ (in which case we are done) or there exists a $p \in P$ that is unrelated to $y$, yet $[p] \preceq [y]$. By history preservation of $[.]$ we find a $p' \preceq y$ with $p \sim p'$. Next, we will show that the latter leads to a contradiction by induction on the construction of $\sim$.

By construction $p \sim p'$ may be obtained in six ways:

- **(reflexivity)** $p = p'$, but then $p \preceq y$; A contradiction.
- **(symmetry)** $p' \sim p$, in which case we find that all the other arguments actually are symmetric;
- **(transitivity)** $p' \sim p'' \sim p$, in which case we find by induction a $y' \sim y$ such that $p'' \preceq y'$, and by using induction once more a $y'' \sim y'$ with $p \preceq y''$. A contradiction.
- **(base equivalence)** there exists $x, x'$ such that $p = f(x), g(x) = g(x')$, and $p' = f(x')$. But in this case we use that $f$ and $g$ are open maps hence functional bisimulations to find a point $y' \sim y$ with $p \preceq y'$ hence $[p] \preceq [y'] = [y]$. A contradiction;
- **(order contraction)** there exist $q, q'$ such that $p \preceq p' \sim q \preceq q' \sim p$. But then $p \preceq p' \preceq y$. A contradiction;
• (order contraction) there exist \( q, q' \) such that \( p' \preceq q \preceq q' \preceq p' \). But then by induction there is a successor \( y'' \) with \( q' \preceq y'' \) and by induction once more a \( y''' \) with \( p \preceq y''' \) and \( y''' \preceq y' \preceq y \). A contradiction.

From this we conclude that \([\cdot]\) is a functional bisimulation, hence an open map. \( \square \)

As a side remark, there are enough open maps in our concrete category, allowing us an alternative route in reasoning about bisimilarity using theorem 2a.

**Theorem 4.10** In \( \text{Pfx} \xrightarrow{\dashv} \text{Pos} \), there are enough open maps.

**Proof.** Given a map \( \mathcal{X} \xrightarrow{f} \mathcal{Z} \) as in figure 2a, simply start by extending all paths in \( \mathcal{X} \) with the futures they will obtain after mapping to \( \mathcal{Z} \). This gives an embedding \( p \) into \( \mathcal{Y} \), and since no new behavior is added while mapping \( \mathcal{Y} \) to \( \mathcal{Z} \) the resulting map \( o \) is a functional bisimulation, hence open. \( \square \)

## 5 A characterisation of branching bisimulation

In this final section, we show that the so-obtained notion of bisimulation through cospans on prefix orders coincides with branching bisimulation when studying the executions from labelled transition systems. In order to do this, we consider the executions over labeled transition systems as objects in the *slice category* \( \text{Pfx}/A^* \), i.e. the category in which the objects are history preserving maps of the form \( f : U \longrightarrow A^* \) from some arbitrary prefix order \( U \) into the fixed prefix order \( A^* \) of strings over the alphabet \( A \), and in which a morphism between \( f : U \longrightarrow A^* \) and \( g : \mathcal{V} \longrightarrow A^* \) is a history preserving map \( h : U \rightarrow \mathcal{V} \) such that \( f = g \cdot h \).

Note, that without the use of a slice category, prefix orders represent fully observable executions of a system, but in a slice category only the result of the map \( f : U \longrightarrow A^* \) is observable. In a slice category, a system is therefore an observation as a function of ‘branching time’.

**Definition 5.1** A *labelled transition system* is a tuple \( \langle \mathcal{P}, A_\tau, \rightarrow \rangle \) with \( \mathcal{P} \) being a set of states and \( \rightarrow \subseteq \mathcal{P} \times A_\tau \times \mathcal{P} \) is the so called transition relations over the alphabet \( A_\tau \), with \( \tau \notin A \) denoting an invisible action. As usual, we write \( p \xrightarrow{a} q \) for \( (p, a, q) \in \rightarrow \).

**Definition 5.2** A run \( u \) starting from a state \( p \) is a function \( \sigma^- \rightarrow \mathcal{P} \) (from the history of some word \( \sigma \in A_\tau^* \)) satisfying the following conditions: \( u(\varepsilon) = p \) and \( u(\sigma') \xrightarrow{a} u(\sigma'a) \), for every \( \sigma' \preceq \sigma \). We write \( \text{Runs}(p) \) to denote the set of all runs starting from the state \( p \), ordered by: \( u \preceq u' \iff \text{dom}(u) \subseteq \text{dom}(u') \land \forall \sigma \in \text{dom}(u) u'(\sigma) = u(\sigma) \). Furthermore, \( \text{obs}_p : \text{Runs}(p) \rightarrow A^* \) denotes the *observation function* which simply forgets the occurrence of \( \tau \)'s in a word \( \sigma \in A_\tau^* \). In other words, the invisible steps correspond to no change in observation.

**Theorem 5.3** The sets of all runs starting from a state \( p \) in \( \langle \mathcal{P}, A_\tau, \rightarrow \rangle \) is a prefix order. Moreover, the observation function \( \text{obs}_p \) is a history preserving function.

**Definition 5.4** A binary relation \( R \subseteq \mathcal{P} \times \mathcal{P} \) is a branching *bisimulation* relation if and only if the following transfer properties are satisfied.

\[ \begin{align*}
(i) & \text{ If } pRq' \text{ and } p \xrightarrow{a} q, \text{ then either } a = \tau \land qRq', \text{ or there are } p'', q' \in \mathcal{P} \text{ such that } p' \xrightarrow{a} p'' \xrightarrow{a} q', \text{ pRq''}, \text{ and qRq'}.
\end{align*} \]
(ii) If \( p \mathcal{R} p' \) and \( p' \xrightarrow{a} q \), then either \( a = \tau \land p \mathcal{R} q \), or there are \( p'', q' \in \mathbb{P} \) such that \( p \xrightarrow{\varepsilon} p'' \xrightarrow{a} q', p'' \mathcal{R} p' \), and \( q' \mathcal{R} q \).

Here, \( \varepsilon \) denotes zero or more \( \tau \) steps. Two processes \( p, q \in \mathbb{P} \) are branching bisimilar if there exists a branching bisimulation relation \( \mathcal{R} \) such that \( p \mathcal{R} q \).

**Theorem 5.5** Two processes are branching bisimilar iff their corresponding prefix orders are bisimilar through cospans of open maps in the slice category \( \mathbf{Pfx}/A^* \).

**Proof.** Let \( \text{Runs}(p), \text{Runs}(q) \) be bisimilar through cospans of open maps in the slice category \( \mathbf{Pfx}/A^* \). I.e., there exists a prefix order \( \mathcal{R} \) and embedding-open maps \( \text{Runs}(p) \xleftarrow{f} \mathbb{P} \xrightarrow{g} \text{Runs}(q) \) and a history preserving function \( \text{obs} : \mathbb{P} \to A^* \) with \( \text{obs}_p = f \circ \text{obs} \) and \( \text{obs}_q = g \circ \text{obs} \). Define a relation \( \mathcal{R} \subseteq \mathbb{P} \times \mathbb{P} \) by relating \( \pi(u) \mathcal{R} \pi(v) \) whenever \( f(u) = g(v) \), where \( \pi(u) \) returns the last state visited by \( u \). It is easy to verify that this relation is indeed a branching bisimulation relation (see e.g. [5]).

Reversely, suppose \( \mathcal{R} \) is the largest branching bisimulation relation, which is well known to exist between runs of this type and to be an equivalence relation (see [19]). Moreover, from [2], we know that there is a surjection \( f : \mathbb{P} \to \mathbb{P}' \) onto the reduced labelled transition system satisfying:

(i) \( \forall_{p,q \in \mathbb{P}} \ (p \xrightarrow{a} q \land a \in A) \implies f(p) \xrightarrow{a} f(q) \).

(ii) \( \forall_{p,q \in \mathbb{P}} \ p \xrightarrow{\tau} q \implies f(p) = f(q) \lor f(p) \xrightarrow{\tau} f(q) \).

(iii) \( \forall_{p \in \mathbb{P}, q \in \mathbb{P}'} \ f(p) \xrightarrow{a} \bar{q} \implies \exists_{p', q \in \mathbb{P}} p \xrightarrow{\varepsilon} p' \xrightarrow{a} q \land f(p) = f(p') \land f(q) = \bar{q} \).

Next, one may verify using induction on the length of runs that a function \( f : \mathbb{P} \to \mathbb{P}' \) satisfying conditions (i) and (ii) induces a history preserving function \( f_p : \text{Runs}(p) \to \text{Runs}(R(p)) \) (for any \( p \in \mathbb{P} \)), where \( R(p) \) denotes the equivalence class of \( p \). Moreover, Condition (iii) guarantees that \( f_p \) is surjective and it satisfies the characterising set-theoretic condition of an embedding-open map of Theorem 4.4. Now suppose \( q \) is branching bisimilar to \( p \). Like \( f_p \), there is an embedding-open map \( f_q : \text{Runs}(q) \to \text{Runs}(R(q)) \) (note \( R(p) = R(q) \)); thus, giving us a co-span of embedding-open maps \( f_p, f_q \) as required. \( \square \)

6 Discussion and conclusive remarks

In this paper we have shown that a concrete category theoretic interpretation of the definition of open map makes this notion applicable to a wider range of models of behavior and to a wider range of behavioral equivalences. In particular, we have shown that it is able to characterize the notion of branching bisimulation (something that was not possible before) in the setting of prefix orders (in which the original definition of open map did not produce satisfactory results). Furthermore, we have shown that in the category of prefix orders, the path-category that usually parameterizes the notion of open map can be chosen to be subcategory of embeddings, thus giving it a natural and fully category theoretic characterization.

A careful look at the characterisation of bisimulation which we obtained in theorem 4.4, reveals that the paths that are being extended do not necessarily have a maximum. This means that also the continuations of paths that are transfinitely long are taken into account, which occur for example in the literature on hybrid
systems in the context of Zeno behavior [18,10,6]. Thus we expect that open maps can be used to preserve these phenomena in a uniform manner.

From a philosophical point of view, using a concrete category for behavioral models means that we are to distinguish implementations from observations in such a way that we assume all implementations to be observable. Interpreting the result in Theorem 3.4 along these lines, we see that \( X \) is bisimilar to \( Y \) iff any conceivable extension of \( X \) that implements additional behavior of \( Y \) is already observable in \( X \) (although may have been implemented differently). Still, it depends on the concrete category which implementations are actually extensions, i.e. which are the embeddings that need to be preserved.

In the quest for a common approach to modeling computations and other dynamic behavior, the next logical step seems to be to study split faithful functors over \( \text{Pfx} \). This may give insight in which embeddings are and are not to be taken into account. In a sense we already have studied such a split in section 5 by looking at the runs of labeled transition systems rather than at the transition systems themselves. But more general theory may be found here. For example, take any category of syntactic computational models \( M \). We expect the executions of a model in this category will form a prefix order, meaning that implementations in \( M \) can be mapped to history preserving maps in \( \text{Pfx} \) and order preserving maps in \( \text{Pos} \). From the philosophical point of view, the syntactic constructs of \( M \) are the actual implementations and the order preserving maps of \( \text{Pos} \) would serve as observations. The definition of bisimulation for \( M \) would then come from open maps in the concrete category of \( M \) over \( \text{Pos} \), but the expectation that this forgetful functor splits over \( \text{Pfx} \) can be exploited when studying bisimilarity of models in \( M \). For sure, in such a split there are less embeddings in \( M \) than there are in \( \text{Pfx} \), therefore the notion of bisimulation can only have become weaker (there is less to preserve). This means that two models in \( M \) are bisimilar whenever they are bisimilar in \( \text{Pfx} \). The reverse does not necessarily hold, for example because open maps in \( \text{Pfx} \) check for Zeno behavior, while there may not be any Zeno-embeddings in \( M \).

Naturally, splits over \( \text{Pfx} \) alone will usually not be enough. Even in the case of labeled transition systems, we actually used a split over the slice category \( \text{Pfx}/A^* \). The reason for this, is that prefix orders only model the actual behavior and how it is implemented, while the order preserving maps in the base category only relate the moments of observation. If the actual observations need to be preserved, something more (like a slice category) is needed. The advantage of using \( A^* \) as an observation space is that it is itself prefix ordered, but in continuous systems or probabilistic systems this will not be the case. Future research will be aimed at finding a method to deal with such observations of different kinds in a uniform manner.

Finally, this paper stirs up the discussion on whether bisimulation through spans or cospans of open maps is to be preferred as a general theory of behavior. If we follow our own intuition on the meaning of the open map diagram in \( \text{Pfx} \), we see that an open map \( X \leftarrowarrow Y \) represents a relation between an implementation \( X \) and a specification \( Y \) such that all specified behavior is actually implemented. The existence of a span of such maps then means that two specifications ‘agree’ in the sense that they have a common implementation, while the existence of a cospan means that two implementations share the same specification. Apparently
the latter indeed results in an equivalence, while the former does generally not. Indeed, if specification $A$ and $B$ have a common implementation $D$ and specification $C$ and $B$ have a common implementation $E$, then this does not guarantee that specification $A$ and $C$ are not conflicting in some sense. So if our interpretation of the maps is any good, then using cospans is indeed the more reasonable approach towards obtaining an equivalence, but the meaning of spans may still be useful where multiple specifications need to be combined (for instance, in the context of model based design and system architecture).

References


Steps in Modular Specifications for Concurrent Modules
(Invited Tutorial Paper)

Pedro da Rocha Pinto $^{1,3}$

Department of Computing
Imperial College London
London, United Kingdom

Thomas Dinsdale-Young $^{2,4}$

Department of Computer Science
Aarhus University
Aarhus, Denmark

Philippa Gardner $^{1,5}$

Department of Computing
Imperial College London
London, United Kingdom

Abstract
The specification of a concurrent program module is a difficult problem. The specifications must be strong enough to enable reasoning about the intended clients without reference to the underlying module implementation. We survey a range of verification techniques for specifying concurrent modules, in particular highlighting four key concepts: auxiliary state, interference abstraction, resource ownership and atomicity. We show how these concepts combine to provide powerful approaches to specifying concurrent modules.

Keywords: Concurrency, specification, program verification.

1 Introduction
The specification of a concurrent program module is a difficult problem. When concurrent threads work with shared data, the resulting behaviour can be complex.

This research was supported in part by the EPSRC Programme Grants EP/H008373/1 and EP/K008528/1.

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

Email: pmd09@doc.ic.ac.uk
Email: tyoung@cs.au.dk
Email: pg@doc.ic.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Consequently, the specification of such modules requires effective abstractions for describing such complex behaviour. The amount of progress that has been made since the 1970s has been substantial. In this paper, we describe some of the key concepts that have emerged over the last few decades. We restrict our exposition to those concepts which we find most important: auxiliary state, interference abstraction, resource ownership and atomicity.

We use a counter module to highlight the challenges of specifying a concurrent module. We require a specification to be expressive enough for verifying the intended clients of the module, such as a ticket lock. We also require that the specification to be opaque, in that the implementation details do not leak into the specification. Using the counter as illustration, we look at a range of historical verification techniques for concurrency:

- Owicki-Gries introduces auxiliary state to abstract internal state of threads;
- rely/guarantee introduces interference abstraction to abstract the interactions between different threads;
- concurrent separation logic introduces resource ownership to encode interference abstraction as auxiliary state;
- linearisability introduces atomicity as a way to abstract the effects of an operation.

We show how recent developments enable us to combine these techniques to provide expressive ways for specifying concurrent modules such as the counter.

## 2 A Concurrent Counter

We use a concurrent counter as a running example throughout this paper.

### 2.1 Implementation

Consider the following implementation of a concurrent counter:

```plaintext
function read(x) {
    r := [x];
    return r;
}

function incr(x) {
    do {
        r := [x];
        b := CAS(x, r, r + 1);
    } while (b = 0);
    return r;
}

function wkincr(x) {
    r := [x];
    [x] := r + 1;
    return r;
}
```

A specification should describe how each operation affects the value of the counter. Here, the read operation returns the value of the counter, the incr operation increments the value and returns the old value, and the incr just increments the value of the counter.

A specification should require the counter to exist as a precondition for each operation, since operations will not work unless the memory holding the counter

---

6 We assume that the primitive read, write and compare-and-swap (CAS) memory operations are atomic.
function lock(x) { 
    t := incr(x.next);
    do {
        v := read(x.owner)
    } while (v ≠ t);
}

function unlock(x) {
    wkincr(x.owner);
}

Fig. 1. A ticket lock implementation using the counter module.

is allocated. In this paper, we use the abstract predicate \( C(x, n) \) to denote the existence of a counter at memory location \( x \) with the value \( n \).

A specification should also describe the permitted interference from the context of concurrent operations. Intuitively, the \texttt{read} and \texttt{incr} operations are robust with respect to concurrent operations that change the value of the counter. By contrast, the (potentially faster\(^7\)) \texttt{wkincr} requires that there is no concurrent operation which changes the value of the counter between the read and increment of the value.

2.2 Ticket Lock Client

The ticket lock algorithm [16] uses the counter module to provide synchronisation. The code for the lock is given in Fig. 1. The lock uses two counters, \texttt{next} and \texttt{owner}, which both initially have value 0. A thread acquires the lock by calling the \texttt{lock} operation. This operation increments the \texttt{next} counter to obtain a notional ticket. When the value of the \texttt{owner} counter agrees with this ticket, the thread has acquired the lock. It can then use whatever resources are protected by the lock without interference from other threads. Control of these resources is relinquished by calling the \texttt{unlock} operation. This increments the \texttt{owner} counter, passing the lock on to the next waiting thread. Intuitively, the use of \texttt{incr} for the \texttt{lock} operation is necessary, since it needs to be robust with respect to concurrent threads taking tickets. The use of \texttt{wkincr} for the \texttt{unlock} operation is possible since only the thread holding the lock should release it.

The challenge is to develop a concurrent specification of a counter module that is strong enough to reason about the ticket lock. This example requires a precise description of how each operation affects the value of the counter, and a detailed account of interference to capture the intuitive distinction between \texttt{incr} and \texttt{wkincr}.

The counter and its ticket lock client are realistic examples that illustrate key difficulties in specifying and reasoning about concurrent modules.

3 Sequential Specification

We can give a sequential specification for the counter module using Hoare triples [11]:

\[
\{C(x, n)\} \texttt{read}(x) \{C(x, n) \land \text{ret} = n\} \\
\{C(x, n)\} \texttt{incr}(x) \{C(x, n + 1) \land \text{ret} = n\} \\
\{C(x, n)\} \texttt{wkincr}(x) \{C(x, n + 1)\}
\]

\(^{7}\) In a quick and dirty experiment, \texttt{wkincr} was around 60\% faster than \texttt{incr}.
With standard Hoare logic, we can use this specification to verify sequential clients that call the counter operations. However, this specification gives no information about the behaviour of the operations in a concurrent setting.

4 Auxiliary State

Owicki and Gries [19] developed the first tractable proof technique for concurrent programs, identifying the importance of reasoning about interference between threads and of using auxiliary state. With the Owicki-Gries method, each thread is given a sequential proof. When the threads are composed, we must check that they do not interfere with each others’ proofs. This is achieved by extending standard Hoare logic with the Owicki-Gries rule for parallel composition:

\[
\begin{align*}
\{P_1\} C_1 \{Q_1\} \quad &\quad \{P_2\} C_2 \{Q_2\} \quad \text{non-interference} \\
\{P_1 \land P_2\} C_1 \parallel C_2 \{Q_1 \land Q_2\}
\end{align*}
\]

The non-interference side condition constrains the proof derivations for $C_1$ and $C_2$. It requires that every intermediate assertion between atomic actions in the proof of $C_1$ must be preserved by every atomic action in the proof of $C_2$, and vice-versa.

An abstract specification for the counter needs to be robust with respect to the non-interference condition. However, in general, the condition will vary depending on the concurrent context. Let us assume that the client may invoke any of the counter operations concurrently, but will not directly interact with the state of the counter. That is, we will only consider interference caused by the counter operations themselves. To this end, we can use an invariant — an assertion that is preserved by each atomic action in the module. For the counter, the invariant $\exists n. C(x, n)$ asserts that the counter at $x$ is allocated and has some value.

We can give the following specification for the counter module:

\[
\begin{align*}
\{\exists n. C(x, n)\} \text{read}(x) \{\exists n, m. C(x, n) \land \text{ret} = m\} \\
\{\exists n. C(x, n)\} \text{incr}(x) \{\exists n, m. C(x, n) \land \text{ret} = m\} \\
\{\exists n. C(x, n)\} \text{wk incr}(x) \{\exists n. C(x, n)\}
\end{align*}
\]

However, these specifications are too weak to specify such clients as the ticket lock. They lose all information about the value of the counter, and give no information about how the operations change this value. In fact, the read operation could change the value of the counter and it would still satisfy the specification! Unfortunately, assertions that describe the precise value of the counter are not invariant.

The Owicki-Gries method is able to provide stronger specifications by using auxiliary state, which records extra information about the execution history via auxiliary variables. The code is instrumented with auxiliary code, which updates the auxiliary variables. Since the auxiliary code only updates auxiliary variables, it has no effect on the program behaviour, and so can be erased — it is not required when the program is run.

By way of example, consider two threads that both increment a counter, as in Fig. 2. The auxiliary variables $y$ and $z$, with initial values 0, are used to record the contribution (i.e. the number of increments) of each thread. For each thread, the
Fig. 2. Reasoning about concurrent increments using auxiliary state.

code of the incr operation is instrumented with code that updates the auxiliary variables when the CAS operations succeed. The auxiliary variables must be updated at the same instant as the counter, so that the counter always holds the sum of the two contributions — our invariant. This is expressed angle brackets, the ⟨⟩, which indicate that the CAS and auxiliary code should be executed in a single atomic step.

The resulting specification of the two-increment program is strong, with precise information about the initial and final value of the counter. However, it comes at the price of modularity.

Firstly, each use of the incr operation requires the underlying implementation to be extended with auxiliary code to increment the appropriate auxiliary variable. A modular proof would not modify the module code for each use by the client.

Secondly, the incr operations require different specifications depending on the client’s use: in our example, the counter predicate \( C(x, y + z) \) uses auxiliary variables \( y \) and \( z \); with three threads, the specification requires three auxiliary variables. A modular proof would give a specification for the module that captures all use cases.

Thirdly and more subtly, the Owicki-Gries method requires the global non-interference condition. To meet this, we made the implicit assumption that the client only interacts with the state of the counter through the counter operations. A modular proof would be explicit about such assumptions about the behaviour of the client.

Thesis.

The concept of auxiliary state, introduced in the Owicki-Gries method, is important in specifying concurrent modules. Auxiliary state abstracts the internal state of threads. It is more convenient to reason using auxiliary variables than to consider the program counter and local variables of each thread in describing invariants. This abstraction is a step towards compositional reasoning. As we shall see, various subsequent approaches have taken a more modular approach to auxiliary state than auxiliary variables provide in the Owicki-Gries method.
5 Interference Abstraction

Jones [13] introduced interference abstraction, providing the rely/guarantee method as a way to improve the compositionality of the Owicki-Gries approach. To avoid the global non-interference condition, specifications explicitly constrain the interference from the concurrent context, and describe the interference that a thread may cause. To this end, each specification incorporates two relations — the rely and guarantee relations — that abstract the interference between threads. The rely relation abstracts the actions of other threads; each assertion in the derivation must be stable under all of these actions. The guarantee relation abstracts the actions in the derivation; each atomic update by the thread must be described by the guarantee.

Rely/guarantee specifications have the form $R, G \vdash \{P\} \Box \{Q\}$, where $R$ and $G$ are the rely and guarantee relations respectively. When composing concurrent threads, the guarantee of each thread must be included in the rely of the other. The parallel composition rule is therefore adapted to:

$$
R \cup G_2, G_1 \vdash \{P_1\} \Box_1 \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\} \Box_2 \{Q_2\}
$$

$$
R, G_1 \cup G_2 \vdash \{P_1 \land P_2\} \Box_1 \Box_2 \{Q_1 \land Q_2\}
$$

The rely/guarantee specifications for the read and incr operations are:

$$
A, \emptyset \vdash \{\exists n. C(x,n)\} \text{read}(x) \{\exists n. C(x,n) \land \text{ret} \leq n\}
$$

$$
A, A \vdash \{\exists n. C(x,n)\} \text{incr}(x) \{\exists n. C(x,n) \land \text{ret} \leq n\}
$$

where $A = \{C(x,n) \rightsquigarrow C(x,n+1) \mid n \in \mathbb{N}\}$. The read specification has an empty guarantee relation indicating that nothing is changed by the read. It has rely relation $A$ stating that the other threads can only increment the counter, although they can do so as many times as they like. The incr relation has the same rely relation. Its guarantee relation is also $A$, stating that the increment can increase the value of the counter. The guarantee must be defined for all values $n$, because the context can change the counter value. This means that we cannot express that the incr operation only does a single increment.

The rely/guarantee specification for the wkincr operation is subtle. Recall that, intuitively, the wkincr operation is intended to be used when no other threads are concurrently updating the counter. As a first try, we can give a simple specification with a rely condition that enforces this constraint:

$$
\emptyset, G \vdash \{C(x,n)\} \text{wkincr}(x) \{C(x,n+1)\}
$$

where $G \triangleq \{C(x,n) \rightsquigarrow C(x,n+1)\}$. The rely relation is empty, so this specification cannot be used in a context where concurrent updates may occur. This means that the guarantee relation can be very precise, consisting of a single action. Effectively, the increment will appear as a single atomic operation.

Although this specification captures some of the intended behaviour of wkincr, it is insufficient to reason about the ticket lock. With the ticket lock, it is possible for two invocations of the wkincr operation to be executing concurrently. Only one thread can call unlock at any one time, because only one thread can have the
lock. However, suppose one thread calling `unlock` has executed the body of `wkincr`. Then, a second thread may correctly conclude that it now has the lock and release it, before the call of the first thread has returned. This results in a concurrent invocation of `wkincr`. By ruling out all concurrent updates to the counter with an empty rely relation, the above specification does not allow this concurrent behaviour.

By changing the rely, it is possible to allow such concurrent updates, but at the expense of weakening the specification:

$$R, G \vdash \{C(x,n)\} \text{wkincr}(x) \{\exists n' \geq n + 1. C(x,n')\}$$

where $R = \{C(x,m) \leadsto C(x,m + 1) \mid m > n\}$ and $G$ is as before. Notice that the rely states that concurrent increments can only happen when the value of the counter is above $n$. Also notice that, in weakening the rely, we must weaken the postcondition to make it stable.

In summary, this specification is again too weak to reason about the ticket lock. It is possible to instrument the code with auxiliary variables, as with the Owicki-Gries method, but this would again lead to a loss of modularity.

**Thesis.**

The concept of interference abstraction, introduced in the rely/guarantee method, is important in specifying concurrent modules. By abstracting the interactions between different threads, specifications can express constraints on their concurrent contexts. This abstraction leads to more compositional reasoning: since the interference is part of the specification, we do not need to examine proofs in order to justify parallel composition. While they may specify it differently, some form of interference abstraction is generally present in subsequent concurrency verification approaches.

### 6 Resource Ownership

In the Owicki-Gries and rely/guarantee approaches, auxiliary variables provide a mechanism for reasoning about which threads can do what and when. For instance, auxiliary variables can be used to reason about the contribution of individual threads to the counter, as we demonstrated in §4, or that one thread can increment a counter after another. O’Hearn introduced a style of reasoning based on resource ownership, developing concurrent separation logic [18] which provides an alternative, more modular approach to such reasoning.

Concurrent separation logic is a Hoare logic, with assertions describing data (such as heap cells or counter objects) treated as resources. Each operation acts on specific resources, with the precondition conferring ownership of the resources it represents. When threads operate on disjoint resources, they do not interfere and so their effects can be simply combined. This principle is embodied in the disjoint parallel composition rule:

$$\{P_1\} C_1 \{Q_1\} \quad \{P_2\} C_2 \{Q_2\} \\
\{P_1 \parallel P_2\} C_1 \parallel C_2 \{Q_1 \parallel Q_2\}$$
where the separating conjunction $P_1 \ast P_2$ describes the disjoint combination of the resources of $P_1$ and $P_2$.

We can think of ownership as embodying specialised notions of auxiliary state and interference abstraction. Ownership is a form of auxiliary state: the program does not explicitly record which threads own what resources, it is an abstraction that we use for reasoning. Ownership implements a simple interference abstraction: threads may update the resources that they own, and disjointness of ownership enforces that they cannot interfere with other threads’ resources.

In the original concurrent separation logic, it was only possible to reason about shared resource that had been transferred between threads through synchronisation. Subsequent approaches [23,7,6] added support for reasoning about fine-grained concurrency by incorporating various styles of rely/guarantee reasoning over shared resources. Building on this work, the concurrent abstract predicates (CAP) [5] approach introduces abstractions over these shared resources that may be split, effectively allowing concurrent manipulation at the abstract level.

Treating the abstract predicate $C(x, n)$ as a resource, we could use the original sequential specification as a concurrent one. However, for multiple threads to use the counter, they would have to transfer the resource between each other using some form of synchronisation. Such a specification effectively enforces sequential access to the counter. This is because the client has no mechanism for dividing the resource: in particular,

$$C(x, n) \implies C(x, n) \ast C(x, n)$$

does not hold.

Following Boyland [2], Bornat et al. [1] introduced permission accounting to separation logic. This allows shared resources to be divided by associating with them a fraction in the interval $(0, 1]$. Shared resources may be subdivided by splitting this fraction. For instance, we may associate fractions with our counter resource and declare the logical axiom:

$$C(x, n, \pi_1 + \pi_2) \iff C(x, n, \pi_1) \ast C(x, n, \pi_2)$$

for $\pi_1 + \pi_2 \leq 1$. We can now modify our counter specification to give concurrent read access:

$$\{C(x, n, \pi)\} \text{read}(x) \{C(x, n, \pi) \land \text{ret} = n\}$$
$$\{C(x, n, 1)\} \text{incr}(x) \{C(x, n + 1, 1) \land \text{ret} = n\}$$
$$\{C(x, n, 1)\} \text{wkincr}(x) \{C(x, n + 1, 1)\}$$

Notice that we require full permission (the 1) in order to perform either increment operation. This means that only concurrent reads are permitted; concurrent updates must be synchronised with all other concurrent accesses (both increments and reads). If only partial permission were necessary, then the specification for $\text{read}$ would be incorrect, since it could no longer guarantee that the value being read matched the resource it had.

---

8 In [18], conditional critical regions provide the synchronisation mechanism.
It is possible to specify concurrent increments, by changing how we interpret the counter predicate $C(x, n, \pi)$. Now the resource $C(x, n, \pi)$ no longer asserts that the value of the counter is $n$, except if $\pi = 1$. Instead, it asserts that the thread is contributing $n$ to the value of the counter; other threads may also have contributions. We can split this counter resource by declaring the logical axiom:

$$C(x, n_1 + n_2, \pi_1 + \pi_2) \iff C(x, n_1, \pi_1) \ast C(x, n_2, \pi_2)$$

for $n_1, n_2 \in \mathbb{N}$ and $\pi_1, \pi_2 \in (0, 1]$. We then specify our counter operations as:

- $\{C(x, n, \pi)\} \text{read}(x) \{C(x, n, \pi) \land \text{ret} \geq n\}$
- $\{C(x, n, 1)\} \text{read}(x) \{C(x, n, 1) \land \text{ret} = n\}$
- $\{C(x, n, \pi)\} \text{incr}(x) \{C(x, n + 1, \pi) \land \text{ret} = n\}$
- $\{C(x, n, 1)\} \text{wkincr}(x) \{C(x, n + 1, 1)\}$

At last, we have a specification that allows concurrent reads and increments.

Fig. 3 shows how it can be used to verify the example of two concurrent increments. Whereas in Fig. 2 each thread was instrumented with different auxiliary code, here the code has not been changed. Rather than each thread having an auxiliary variable to record its contribution to the counter, the contribution is recorded in auxiliary resources that are owned by the thread and encapsulated in the $C(x, n, \pi)$ predicate. This idea of subjective auxiliary state is at the core of subjective concurrent separation logic (SCSL) [15] (and the subsequent FCSL [17]).

This specification still has some weaknesses. The $\text{wkincr}$ operation must still be synchronised with the other operations. Also, sequenced reads will never see decreasing values of the counter (since the contribution is not changed and only provides the lower bound). It is possible to describe a more elaborate permission system that allows $\text{wkincr}$ in the presence of reads, and to extend the predicate to record the last known value as a lower bound for reads. This would give us a more useful, if somewhat cumbersome, specification. However, it would still not handle the ticket lock.

While a ticket lock has been verified using CAP [5], the proof depends on the atomicity of the underlying counter operations in order to synchronise access to shared resources. The proof does work with any of our abstract specifications, since they simply do not embody the necessary atomicity.
Thesis.

The concept of resource ownership, developed by concurrent separation logic and its successors, is important in specifying concurrent modules. The idiom of ownership can be seen as a form of auxiliary state, which critically embodies a notion of disjointness and interference abstraction. Various approaches have explored the power of ownership for reasoning about concurrency [5, 15, 17, 3, 14]. While it is an effective tool, and can be used to give elegant specifications, something more is required to provide the strong specifications we are seeking.

7 Atomicity

Atomicity is the abstraction that an operation takes effect at a single, discrete instant in time. The concurrent behaviour of such atomic operations is equivalent to a sequential interleaving of the operations. A well-known correctness condition for atomicity, which identifies when the operations of a concurrent module appear to behave atomically, is linearisability [10]. A client can use such operations as if they were simple atomic operations.

Using the linearisability approach, each operation is given a sequential specification. The operations are then proved to behave atomically with respect to each other. One way of seeing this is that there is an instant during the invocation of each operation at which it appears to take effect. This instant is referred to as the linearisation point. With linearisability, the interference of every operation is tolerated at all times by any of the other operations. Consequently, the interference abstraction is deemed to be the module boundary.

Given our sequential specification for the counter in §3, is our implementation linearisable? If we only consider the read and incr operations, then yes, it is. However, the addition of the wkincr operation breaks linearisability. The problem with wkincr is that, for instance, two concurrent calls can result in the counter only being incremented once. This is not consistent with atomic behaviour.

The essence of the problem is that we only envisage calling wkincr in a concurrent context where there are no other increments. In such a case, it would appear to behave atomically. By itself, the sequential specification cannot express this constraint. We need an interference abstraction that constrains the concurrent context.

Linearisability is related to the notion of contextual refinement. With contextual refinement, the behaviour of program code is described by (more abstract) specification code.\footnote{In general, the specification code need not be directly executable, although it does have a semantics.} Contextual refinement asserts that the specification code can be replaced by the program code in any context, without introducing new observable behaviours; we say that the program code contextually refines the specification code. Filipović et al. [8] have shown that, under certain assumptions about a programming language, linearisability implies contextual refinement for that language. For a linearisable module, each operation contextually refines the operation itself executed atomically. For instance, incr(x) contextually refines ⟨incr(x)⟩.

CaReSL [22] is a logic for proving contextual refinement of concurrent programs. CaReSL makes use of auxiliary state, interference abstraction and ownership in its
proof technique. However, these concepts are not exposed in their specifications. This means that it is not obvious what a suitable specification of \texttt{wkincr} in CaReSL should be.

**Thesis.**

The concept of \textit{atomicity}, put forward by linearisability, is important in specifying concurrent modules. Atomicity can be seen as a form of interference abstraction: it effectively guarantees that the only observable interference from an operation will occur at a single instant in its execution. This is a powerful abstraction, since a client need not consider intermediate states of an atomic operation (which, for non-atomic operations, might violate invariants) but only the overall transformation it performs.

8 Synthesis

We now examine several approaches that bring together the ideas we have so far discussed to provide expressive specifications for concurrent modules.

8.1 A Higher-Order Approach

One way of overcoming the non-modularity of the Owicki-Gries method was introduced by Jacobs and Piessens \cite{JacobsPiessens}. Their key idea is to give higher-order specifications for operations, which are parametrised by auxiliary code that is performed when the abstract atomic operation appears to take effect (the linearisation point). Where previously we instrumented the code of the \texttt{incr} operation differently for different call sites, here it is instrumented uniformly; the auxiliary code is a parameter that is determined at the call site.

Applying this idea to the \texttt{incr} operation we have the following code:

```c
function incr(x, \rho) {
  do {
    r := x;
    \langle b := \text{CAS}(x, r, r + 1);
    if (b) \rho; \rangle
  } while (b = 0);
  return r;
}
```

Note that \(\rho\) is an auxiliary code parameter of the operation. When the atomic update to the counter occurs, the auxiliary code is run and can update auxiliary variables of the client.

The specification of \texttt{incr} is parametrised by the specification of the auxiliary code. Written as a proof rule, the specification is as follows:

\[
I(x) \ast S \Leftrightarrow \exists n. C(x, n) \ast R(n) \quad \forall n. \{R(n) \ast P\} \quad \rho \{R(n + 1) \ast Q(n)\}
\]

\[
I(x) \vdash \{S \ast P\} \quad \text{incr}(x, \rho) \quad \{S \ast Q(\text{ret})\}
\]
In the conclusion of this rule, \( I(x) \) is an invariant; it is disjoint from the pre- and postcondition, and must be preserved by atomic updates of all threads. At the point where the counter is atomically incremented, the following steps conceptually take place:

(i) The equivalence from the premiss is used to convert the combination of the invariant \( I(x) \) and the portion of the precondition \( S \) into the counter predicate \( C(x, n) \) and \( R(n) \) for some value of \( n \).

(ii) The module performs the increment, updating \( C(x, n) \) to \( C(x, n + 1) \).

(iii) The auxiliary code \( \rho \) is run, updating \( R(n) \) to \( R(n + 1) \).

(iv) Together, \( C(x, n + 1) \) and \( R(n + 1) \) are converted back to recover the invariant \( I(x) \) and \( S \).

This specification now allows us to exploit the expressivity of auxiliary variables in a modular way. Fig. 4 shows how this technique can be used to prove two concurrent increments. The proof is very similar to the one shown in Fig. 2. The new specification allows us to abstract the atomic update performed by the \( \text{incr} \) and use the same module implementation for both threads.

In Fig. 4, the invariant \( I(x) \) is instantiated as \( C(x, y + z) \). The predicate \( R(n) \) is instantiated as \( n = y + z \). The predicate \( S \) is \text{True}, while \( P \) and \( Q \) are instantiated with the pre- and postconditions of \( \text{incr} \) at each call site.

The \text{read} operation can be specified as:

\[
I(x) \land S \iff \exists n. C(x, n) \land R(n) \land \forall n. (R(n) \land P) \land (R(n) \land Q(n))
\]

\[
I(x) \vdash \{ S \land P \} \text{read}(x, \rho) \{ S \land Q(\text{ret}) \}
\]

Finally, recall that the \text{wkincr} operation is intended to be used when there are no updates from the environment. This can be specified as:

\[
I(x) \land P \iff C(x, n) \land R \land C(x, n + 1) \land R \land I(x) \land Q
\]

\[
I(x) \vdash \{ P \} \text{wkincr}(x, \rho) \{ Q \}
\]

A key difference with the \text{wkincr} specification is that \( n \) is not quantified in each of the premisses. This is because the value of the counter must be preserved by other threads before the update.
Note that although these specifications are written in the form of proof rules, they are actually implications. If a client establishes the premises then it can use the conclusion. The implementation must show that the conclusion follows from the premises. The predicates $I$, $P$, $Q$, $R$ and $S$, as well as the ghost code $\rho$, are universally quantified: the client can instantiate them as necessary.

This higher-order specification approach has been adopted in other higher-order logics such as HOCAP [21], iCAP [20] and Iris [14]. In these logics, auxiliary state is not manipulated by auxiliary code, but by view shifts [4]. These view shifts serve essentially the same purpose — they can update auxiliary state, but have no effect on the concrete state — but do not involve instrumenting the code.

### 8.2 A First-Order Approach

An alternative way of providing specifications for concurrent modules was introduced in the program logic TaDA [3] using atomic triples. Rather than treating atomic specifications as a higher-order construct, atomic triples build such specifications in to TaDA as a first-order construct. An atomic triple has the following form:

$$\forall x \in X. (P(x)) \ C \ (Q(x))$$

Superficially, this can be read as “$C$ atomically updates $P(x)$ to $Q(x)$ (for arbitrary $x \in X$)”. What it actually means is a bit more subtle.

An implementation of the specification may assume that the assertion $P(x_0)$ holds initially (for some $x_0 \in X$). It must tolerate interference from the environment updating $P(x)$ to $P(x')$ (for any $x, x' \in X$). It is at liberty to update the state, providing that it preserves $P(x)$ (for the current value of $x$), until it updates it to $Q(x)$. After this update $Q(x)$ is no longer available to the implementation (another thread may be using it). Finally, the implementation cannot terminate without having updated $P(x)$ to $Q(x)$ at some point.

Using the atomic triple, we can specify the counter as:

\[
\forall n. (C(x, n)) \ \text{read}(x) \ (C(x, n) \land \text{ret} = n) \\
\forall n. (C(x, n)) \ \text{incr}(x) \ (C(x, n + 1) \land \text{ret} = n) \\
(C(x, n)) \ \text{wkincr}(x) \ (C(x, n + 1))
\]

Intuitively, the first two specifications state the value of the counter will be read and incremented atomically, even in the presence of concurrent updates by the environment that change the value of the counter — since the value $n$ is bound by $\forall$. However, the environment must preserve the counter, e.g. it cannot deallocate it. The last specification means that $\text{wkincr}(x)$ will atomically update the counter from $n$ to $n + 1$, as long as the environment guarantees that the shared counter will not change value before the atomic update — since the value of $n$ is not bound by $\forall$.

Atomic triples specify operations with respect to an abstraction (e.g. $C(x, n)$), which means that each operation can be verified independently. This makes it possible to extend modules with new operations without having to verify the existing operations again. Linearisability, by contrast, is a whole module property: adding new operations (e.g. $\text{wkincr}$) can break the linearisability.
In [3], we introduce a generalised version of the atomic triple that can combine atomicity with resource transfer. For example, we can specify an operation that reads the value of the counter into a buffer; the read happens atomically, but the write to the buffer does not, and so ownership of the buffer is transferred between the client and implementation. This is not possible with traditional linearisability, although Gotsman and Yang [9] have proposed an extension of linearisability that supports ownership transfer.

Evaluation.

The counter specifications shown in this section are strong: a client can derive the abstract disjoint specifications from them. Moreover, they are strong enough to support synchronisation: the correctness of the ticket lock can be justified from the counter specifications. These approaches to specification are expressive enough to enforce obligations on both the client and the implementation. By contrast, CAP specifications tend to unduly restrict the client (e.g. a counter specification cannot be used for synchronisation), while linearisability specifications tend to unduly restrict the implementation (e.g. a counter cannot provide a \texttt{w} \texttt{k} \texttt{i} \texttt{n} \texttt{c} \texttt{r} operation).

Atomic triples have been encoded in Iris [14] by interpreting them as specifications in the Jacobs-Piessens style. This captures the intensional meaning behind atomic triples — that is, what they can be used for — which in TaDA is expressed through the proof rules for using atomic triples.

9 Conclusions

We have considered a number of proof methods for verifying concurrent programs: Owicki-Gries, rely/guarantee, concurrent separation logics and linearisability. In each method, we have identified a particularly valuable contribution towards specifying concurrent modules. Finally, we have demonstrated how these ideas can be brought together to produce specifications that are both expressive and modular.

References


