Basic Research in Computer Science

BRICS NS-01-2 Brookes & Mislove (eds.): MFPS '01 Preliminary Proceedings

Preliminary Proceedings of the 17th Annual Conference on

Mathematical Foundations of Programming Semantics

MFPS '01

Aarhus, Denmark, May 24–27, 2001

Stephen Brookes
Michael Mislove
(editors)

BRICS Notes Series **NS-01-2**

ISSN 0909-3206 **May 2001**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/01/2/`

Electronic Notes in Theoretical Computer Science

Volume 45

Mathematical Foundations of Programming Semantics
Seventeenth Annual Conference

Aarhus University
Aarhus, Denmark
May 23 – 26, 2001

Guest Editors:
S. Brookes M. Mislove

Preliminary Proceedings
Final Proceedings will be available at
http://www.elsevier.nl/locate/entcs/volume45.html

Table of Contents

Foreword

These are the preliminary proceedings of the Seventeenth Conference on the Mathematical Foundations of Programming Semantics. The meeting consists of seven invited talks, given by the following:

Olivier Danvy	Joshua Guttman
BRICS	*Mitre*
Neil Jones	Kim Larsen
DIKU	*Aalborg*
Prakash Panangaden	Jan Rutten
McGill	*CWI*

Glynn Winskel
Cambridge

There also are three special sessions, whose topics are:

- A session honoring Neil Jones, organized by Olivier Danvy and David Schmidt. This session begins with an invited address by Professor Danvy, and includes talks by Radhia Cousot, John Hannan, John Hughes, David Schmidt and Peter Sestoft.
- A session on model checking organized by Gavin Lowe. This commences with an invited talk by Kim Larsen, and includes talks by José Desharnais, Michael Huth, Henrik Jensen, Marta Kwiatkowska, and Gavin Lowe,
- A session on security, organized by Catherine Meadows. This commences with an invited talk by Joshua Guttman, and includes talks by Andrew Gordon and Alan Jeffrey, Gavin Lowe, Thomas Jensen, Catherine Meadows, and Andre Scedrov.

The remainder of the program is made up of papers selected by the Program Committee from those selected from the submission in response to the Call for Papers. The Program Committee was co-chaired by Stephen Brookes and Michael Mislove, and included

Lars Birkedal	Rance Cleaveland
ITU	*SUNY, Stony Brook*
Marcelo Fiore	Matthew Hennessy
Cambridge	*Sussex*
Alan Jeffrey	Achim Jung
DePaul	*Birmingham*
Gavin Lowe	Catherine Meadows
Oxford	*NRL*
Peter O'Hearn	Susan Older
Queen Mary & Westfield	*Syracuse*

Dusco Pavlovic
Kestrel

Uday Reddy
Birmingham

Giuseppe Rosolini
Genoa

Davide Sangiorgi
INRIA

Andre Scedrov
Pennsylvania

This year's meeting is being hosted by Aarhus University, with the local arrangements being carried out by Professors Olivier Danvy and Andrzej Filinski. We are grateful to these colleagues for their having so efficiently overseen the local arrangements. The Organizers also express their appreciation to Karen Kjær Møller, the chief secretary at BRICS for her help with the meeting.

The meeting is being supported by BRICS and by the U. S. Office of Naval Research. We are grateful to both organizations for making the meeting possible, and we especially thank Dr. R. F. Wachter at ONR who has provided continued support for the MFPS series.

Stephen Brookes *Michael Mislove*
Conference Co-chairs

Dedication

The Organizers of the MFPS series dedicate these Proceedings to Neil Jones for his continuing inspiration to researchers in theoretical computer science. Neil has been a regular participant in the MFPS series, having been one of the invited speakers at the 1987 meeting, and having regularly participated in the series. MFPS appreciates the continued inspiration that his research results have provided, and that his talks at MFPS have so clearly elucidated.

MFPS 17 Preliminary Version

A Relationship between Equilogical Spaces and Type Two Effectivity

Andrej Bauer [1]

Institut Mittag-Leffler
The Royal Swedish Academy of Sciences

Abstract

In this paper I compare two well studied approaches to topological semantics—the domain-theoretic approach, exemplified by the category of countably based equilogical spaces, Equ, and Type Two Effectivity, exemplified by the category of Baire space representations, $\mathsf{Rep}(\mathbb{B})$. These two categories are both locally cartesian closed extensions of countably based T_0-spaces. A natural question to ask is how they are related.

First, we show that $\mathsf{Rep}(\mathbb{B})$ is equivalent to a full coreflective subcategory of Equ, consisting of the so-called 0-equilogical spaces. This establishes a pair of adjoint functors between $\mathsf{Rep}(\mathbb{B})$ and Equ. The inclusion $\mathsf{Rep}(\mathbb{B}) \to \mathsf{Equ}$ and its coreflection have many desirable properties, but they do not preserve exponentials in general. This means that the cartesian closed structures of $\mathsf{Rep}(\mathbb{B})$ and Equ are essentially different. However, in a second comparison we show that $\mathsf{Rep}(\mathbb{B})$ and Equ do share a common cartesian closed subcategory that contains all countably based T_0-spaces. Therefore, the domain-theoretic approach and TTE yield equivalent topological semantics of computation for all higher-order types over countably based T_0-spaces. We consider several examples involving the natural numbers and the real numbers to demonstrate how these comparisons make it possible to transfer results from one setting to another.

1 Introduction

In this paper I compare two approaches to topological semantics—the domain-theoretic approach, exemplified by the category of countably based *equilogical spaces* [6,23], Equ, and *Type Two Effectivity* (TTE) [27,26,25,14], exemplified by the category of *Baire space representations*, $\mathsf{Rep}(\mathbb{B})$. These frameworks have been extensively studied, albeit by two somewhat separate research communities. The present paper relates the two approaches and helps transfer results between them.

[1] E-mail: Andrej.Bauer@andrej.com, URL: http://andrej.com

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Domain-theoretic models of computation arise from the idea that the result of a (possibly infinite) computation is *approximated* by the *finite* stages of the computation. As the computation progresses, the finite stages approximate the final result ever so better. This leads to a formulation of partially ordered spaces, called *domains*, in which every element is the supremum of the distinguished "finite" elements that are below it. We recommend [1] and [24] for an introduction to domain theory.

The TTE framework arises from the study of (possibly infinite) computations performed by Turing machines that read infinite input tapes and write results on infinite output tapes. If we view input and output tapes as a sequences of natural numbers, then Turing machines correspond to computable partial operators on the Baire space $\mathbb{B} = \mathbb{N}^{\mathbb{N}}$. We obtain a purely topological model of computation by considering all *continuous* partial operators on $\mathbb{B}$, not just the computable ones. We recommend [27] for an introduction to TTE.

The use of equilogical spaces as an exemplification of the domain-theoretic approach to topological semantics needs an explanation. Already in the original manuscript [23] Scott showed that equilogical spaces are equivalent to partial equivalence relations (PERs) on algebraic lattices. He also proved that the category of algebraic domains is a cartesian closed subcategory of equilogical spaces, and it is not hard to see that the same holds for continuous lattices. In [6,5] we showed that equilogical spaces are a generalization of domain theory with totality [9,8,7,20,21]. The crucial observation needed for those results is that equilogical spaces are equivalent to the category of *dense* PERs on algebraic domains (a PER on a domain is said to be dense if its extension is a dense subset of the domain). The equivalence remains if we take dense PERs on continuous domains instead. In this sense, it is fair to say that equilogical spaces generalize several domain-theoretic frameworks and contain a number of important categories of domains that have been studied, but of course not all of them. In this paper we focus solely on the countably based equilogical spaces, and call them simply "equilogical spaces".

As the ambient category of TTE we take the category of Baire space representations, $\mathsf{Rep}(\mathbb{B})$, which is defined in Section 3. Contemporary formulations of TTE often use the Cantor space in place of the Baire space, but since we are not concerned with computational complexity here, it does not matter which one we use because they yield in equivalent categories. We call Baire space representations just "representations".

Equilogical spaces and representations both form locally cartesian closed extensions of the category of countably based T_0-spaces, $\omega\mathsf{Top}_0$. Thus they are both appealing models of computation on topological spaces. This is why it is important from the programming semantics point of view to understand precisely how they are related.

The general framework within which we carry out the comparison is realizability theory, since Equ and $\mathsf{PER}(\mathbb{B})$ are just realizability models; the former is equivalent to the PER model on the Scott-Plotkin graph model $\mathcal{P}\mathbb{N}$, whereas

the latter is equivalent to the PER model on the Second Kleene Algebra $\mathbb{B}$. We can then use Longley's theory of applicative morphisms between partial combinatory algebras (PCAs) to compare the two PER models [17]. While this may be the most general and elegant technique that could be used to compare other semantic frameworks as well, it has a distinctly anti-topological flavor. But we can translate all the results from realizability back into the language of topology, which is precisely what we do. This immediately gives us the first result: a simple topological description of $\mathsf{Rep}(\mathbb{B})$, without any mention of the partial combinatory structure of the Second Kleene Algebra.

From the topological description of $\mathsf{Rep}(\mathbb{B})$ so obtained, it is apparent that $\mathsf{Rep}(\mathbb{B})$ is equivalent to a full subcategory of Equ. This subcategory is denoted by $\mathsf{0Equ}$ and consists of all the 0-*equilogical spaces*, which are those equilogical spaces whose underlying topological spaces are 0-dimensional. The inclusion $I\colon \mathsf{0Equ} \to \mathsf{Equ}$ has a coreflection $D\colon \mathsf{Equ} \to \mathsf{0Equ}$. These two functors have many desirable properties, but they do *not* preserve the function spaces in general.

We compare Equ and $\mathsf{Rep}(\mathbb{B})$ in another way, by demonstrating that they share a common cartesian closed subcategory that contains all countably based T_0-spaces. This subcategory was discovered by Menni and Simpson [19,18] as the category of *ω-projecting T_0-quotients*, and by Schröder [22] as the category of *sequential T_0-spaces with admissible representations*. We prove that these two categories coincide. Therefore, the domain-theoretic approach and TTE yield equivalent topological semantics of computation for all higher-order types over countably based T_0-spaces.

Finally, we discuss various consequences and the potential for transfer of results between the two settings, in particular with respect to the natural numbers, the real numbers, and their higher-order function spaces.

The paper is organized as follows. In Section 1 we review the basic definitions and facts about equilogical spaces and ω-projecting quotients. In Section 3 we review Baire space representations and admissible representations. Sections 4 and 5 contain the two comparisons of Equ and $\mathsf{Rep}(\mathbb{B})$. In Section 6 we obtain various transfer results between the two settings.

The material presented here is part of my Ph.D. dissertation [4], written under the supervision of Dana Scott. The omitted proofs can be found in the dissertation.

I gratefully acknowledge helpful discussions about this topic with Steven Awodey, Lars Birkedal, Peter Lietz, Alex Simpson, Matthias Schröder, and Dana Scott. Peter and I found the equivalence of 0-equilogical spaces and Baire space representations together. I could have never proved the coincidence of ω-projecting quotients and admissible representations without talking to Matthias and Alex. I also thank the knowledgeable anonymous referee for helpful suggestions on how to better present the material.

2 Equilogical Spaces and ω-projecting Quotients

An *equilogical space* was defined by Scott [23,6] to be a T_0-space with an equivalence relation. Here we are only interested in *countably based equilogical spaces*, which are countably based T_0-spaces with equivalence relations. We denote the category of countably based T_0-spaces and continuous maps by $\omega\mathsf{Top}_0$. We omit the qualifier "countably based" from now on, unless we are explicitly dealing with spaces that are not countably based.

More precisely, an equilogical space is a pair $X = (|X|, \equiv_X)$ where $|X| \in \omega\mathsf{Top}_0$ and $\equiv_X$ is an equivalence relation on the underlying set of $|X|$. The *associated quotient* of an equilogical space X is the topological quotient $\|X\| = |X|/\equiv_X$. The canonical quotient map $|X| \to \|X\|$ is denoted by q_X. Note that $\|X\|$ need not be T_0 or countably based. A morphism $f\colon X \to Y$ between equilogical spaces X and Y is a continuous map $f\colon \|X\| \to \|Y\|$ that is *tracked* by some (not necessarily unique) continuous map $g\colon |X| \to |Y|$, which means that the following diagram commutes:

$$\begin{array}{ccc} |X| & \xrightarrow{\;g\;} & |Y| \\ {\scriptstyle q_X}\downarrow & & \downarrow{\scriptstyle q_Y} \\ \|X\| & \xrightarrow[\;f\;]{} & \|Y\| \end{array}$$

Any map g that appears in the top row of such a diagram is *equivariant*, or *extensional*, meaning that, for all $x, y \in |X|$, $x \equiv_X y$ implies $gx \equiv_Y gy$.[2] The category of equilogical spaces and morphisms between them is denoted by Equ.

An *exponential* of X and Y is an object $E = Y^X$ with a morphism $e\colon E \times X \to Y$, called the *evaluation map*, such that, for all Z and $f\colon Z \times X \to Y$, there exists a unique map $\widetilde{f}\colon Z \to E$, called the *transpose* of f, such that the following diagram commutes:

$$\begin{array}{ccc} E \times X & & \\ {\scriptstyle \widetilde{f} \times 1_X}\uparrow & \searrow^{e} & \\ Z \times X & \xrightarrow[\;f\;]{} & Y \end{array}$$

A *weak exponential* is defined in the same way but without the uniqueness requirement for $\widetilde{f}$. A category is said to be *cartesian closed* when it has the terminal object, finite products, and all exponentials. It is *locally cartesian closed* when every slice is cartesian closed.

[2] We could define morphisms between equilogical spaces to be equivalence classes of equivariant maps, which is the original definition from [23].

The category Equ is equivalent to the PER model $\mathsf{PER}(\mathcal{P}\mathbb{N})$ [4, Theorem 4.1.3], which is a regular locally cartesian closed category. This equivalence gives us a description of exponentials in Equ, though a very impractical one. A somewhat better description can be obtained as follows. Suppose X and Y are equilogical spaces, and (W, e) is a weak exponential of $|X|$ and $|Y|$ in $\omega\mathsf{Top}_0$. Define a relation $\equiv_E$ on W by

$$f \equiv_E g \iff \forall\, x, y \in |X|\,.\,(x \equiv_X y \Longrightarrow e(f,x) \equiv_Y e(g,y)) \ .$$

Let $E = (|E|, \equiv_E)$ be the equilogical space whose underlying space is

$$|E| = \left\{ f \in W \;\middle|\; f \equiv_E f \right\} \subseteq W \ .$$

It is easy to check that E with the morphism induced by the evaluation map $e\colon |E| \times |X| \to |Y|$ is the exponential of X and Y [4, Proposition 4.1.7]. The category $\omega\mathsf{Top}_0$ has weak exponentials, thus the following construction shows that Equ has exponentials. It would be desirable to have a good theory of weak exponentials of topological spaces, as that would give us better descriptions of exponentials in Equ. In certain cases (weak) exponentials have good descriptions. For example, if $|X|$ is locally compact and Hausdorff, then the space of continuous maps $W = \mathcal{C}(|X|, |Y|)$ with the compact-open topology together with the usual evaluation map is an exponential of $|X|$ and $|Y|$ in $\omega\mathsf{Top}_0$.

Every countably based T_0-space X can be viewed as an equilogical space $(X, =_X)$ where $=_X$ is equality on X. This defines a full and faithful inclusion functor $I\colon \omega\mathsf{Top}_0 \to \mathsf{Equ}$. The inclusion preserves finite limits, coproducts, and all exponentials that already exist in $\omega\mathsf{Top}_0$. Preservation of exponentials follows directly from the above description of exponentials in Equ.

There is the *associated quotient* functor $Q\colon \mathsf{Equ} \to \mathsf{Top}$ that maps an equilogical space X to the associated quotient $QX = \|X\|$ and a morphism $f\colon X \to Y$ to the continuous map $Qf = f\colon \|X\| \to \|Y\|$. Here Top is the category of *all* topological spaces and continuous maps, because the associated quotient need not be countably based or T_0. Clearly, Q is a faithful functor, and it is not hard too see that it is not full. Menni and Simpson [19,18] showed that there is a largest subcategory $\mathcal{C}$ of Equ such that Q restricted to $\mathcal{C}$ is full. They worked with equilogical spaces built from all countably based topological spaces, as opposed to just T_0-spaces, but their results hold when we restrict them to T_0-spaces. We are restricting to T_0-spaces because Schröder proved his results for T_0-spaces. Below we summarize the relevant findings from [19,18].

Definition 2.1 A subset $S \subseteq X$ of a topological space X is *sequentially open* when every sequence with limit in S is eventually in S. A topological space X is a *sequential space* when every sequentially open set $V \subseteq X$ is open in X. The category of sequential spaces and continuous maps between them is denoted by Seq.

Theorem 2.2 *Sequential spaces form a cartesian closed category that contains* $\omega\mathsf{Top}_0$*. The inclusion* $\omega\mathsf{Top}_0 \to \mathsf{Seq}$ *preserves finite limits and all exponentials that already exist in* $\omega\mathsf{Top}_0$*.*

Proof. This is well known and follows from the fact that Seq is a reflective subcategory of the cartesian-closed category Lim of *limit spaces* [15], and the reflection preserves products. □

Definition 2.3 Let $X \in \omega\mathsf{Top}_0$ and $q\colon X \to Y$ be a continuous map. Then q is said to be *ω-projecting* when for every $Z \in \omega\mathsf{Top}_0$ and every continuous map $f\colon Z \to Y$ there exists a lifting $g\colon Z \to X$ such that $f = q \circ g$.

An equilogical space X is *ω-projecting* when the canonical quotient map $q_X\colon |X| \to \|X\|$ is ω-projecting. The full subcategory of Equ on the ω-projecting equilogical spaces is denoted by EPQ_0. Let PQ_0 be the category of those T_0-spaces Y for which there exists an ω-projecting map $q\colon X \to Y$.

The name PQ_0 stands for "ω-projecting quotient", and EPQ_0 stands for "equilogical ω-projecting quotient".

Theorem 2.4 (Menni & Simpson [19]) *The category* PQ_0 *is a cartesian closed subcategory of* Seq*,* EPQ_0 *is a cartesian closed subcategory of* Equ*, and the categories* PQ_0 *and* EPQ_0 *are equivalent via the restriction of the associated quotient functor* $Q\colon \mathsf{EPQ}_0 \to \mathsf{PQ}_0$*.*

Proof. See [19]. In fact, Menni and Simpson prove that PQ_0 is the largest common subcategory $\mathcal{C}$ of Equ and Top such that Q restricted to $\mathcal{C}$ is full. □

3 Type Two Effectivity

In this section we review the basic setup of Type Two Effectivity. The Baire space $\mathbb{B} = \mathbb{N}^{\mathbb{N}}$ is the set of all infinite sequences of natural numbers, equipped with the product topology. Let $\mathbb{N}^*$ be the set of all finite sequences of natural numbers. The length of a finite sequence a is denoted by $|a|$. If $a, b \in \mathbb{N}^*$ we write $a \sqsubseteq b$ when a is a prefix of b. Similarly, we write $a \sqsubseteq \alpha$ when a is a prefix of an infinite sequence $\alpha \in \mathbb{B}$. A countable topological base for $\mathbb{B}$ consists of the basic open sets, for $a \in \mathbb{N}^*$,

$$a{::}\mathbb{B} = \big\{a{::}\beta \;\big|\; \beta \in \mathbb{B}\big\} = \big\{\alpha \in \mathbb{B} \;\big|\; a \sqsubseteq \alpha\big\} \ .$$

The expression $a{::}\beta$ denotes the concatenation of the finite sequence $a \in \mathbb{N}^*$ with the infinite sequence $\beta \in \mathbb{B}$. We write $n{::}\beta$ instead of $[n]{::}\beta$ for $n \in \mathbb{N}$ and $\beta \in \mathbb{B}$. The base $\big\{a{::}\mathbb{B} \;\big|\; a \in \mathbb{N}^*\big\}$ is a clopen countable base for the topology of $\mathbb{B}$, which means that $\mathbb{B}$ is a countably based 0-dimensional T_0-space. Recall that a space is *0-dimensional* when its clopen subsets form a base for its topology. A 0-dimensional T_0-space is always Hausdorff.

In order to obtain a simple topological description of Baire space representations, we need to characterize subspaces of $\mathbb{B}$ and those partial continuous

maps $\mathbb{B} \rightharpoonup \mathbb{B}$ that can be encoded as elements of $\mathbb{B}$. This is accomplished by the Embedding and Extension Theorems for $\mathbb{B}$, which we prove next.

Theorem 3.1 (Embedding Theorem for $\mathbb{B}$) *A topological space is a 0-dimensional countably based T_0-space if, and only if, it embeds into $\mathbb{B}$.*

Proof. Clearly, every subspace of $\mathbb{B}$ is a countably based 0-dimensional T_0-space. Suppose X is a countably based 0-dimensional T_0-space with a countable base $\{U_k \mid k \in \mathbb{N}\}$ of clopen sets. Define the map $e\colon X \to \mathbb{B}$ by

$$ex = \lambda n \in \mathbb{N}.\,(\texttt{if } x \in U_n \texttt{ then } 1 \texttt{ else } 0)\ .$$

It is easy to check that e is a topological embedding. □

For topological spaces X and Y, a partial map $f\colon X \rightharpoonup Y$ is said to be *continuous* when the restriction to its domain $f\colon \mathsf{dom}(f) \to Y$ is a continuous (total) map, where $\mathsf{dom}(f)$ is equipped with the subspace topology inherited from X. There is no requirement that $\mathsf{dom}(f)$ be an open subset of X. We consider partial continuous maps $\mathbb{B} \rightharpoonup \mathbb{B}$ and characterize those that can be encoded as elements of $\mathbb{B}$.

Given a finite sequence of numbers $a = [a_0, \ldots, a_{k-1}]$, let $\mathsf{seq}\, a$ be the encoding of a as a natural number, for example

$$\mathsf{seq}\,[a_0, \ldots, a_{k-1}] = \prod_{i=0}^{k-1} {p_i}^{1+a_i}\ ,$$

where p_i is the i-th prime number. For $\alpha \in \mathbb{B}$ let $\overline{\alpha}n = \mathsf{seq}\,[\alpha 0, \ldots, \alpha(n-1)]$. For $\alpha, \beta \in \mathbb{B}$, define $\alpha \star \beta$ by

$$\alpha \star \beta = n \iff \exists\, m \in \mathbb{N}.\,\big(\alpha(\overline{\beta}m) = n + 1 \wedge \forall\, k < m\,.\,\alpha(\overline{\beta}k) = 0\big)\ .$$

If there is no $m \in \mathbb{N}$ that satisfies the above condition, then $\alpha \star \beta$ is undefined. Thus, $\star$ is a partial operation $\mathbb{B} \times \mathbb{B} \rightharpoonup \mathbb{N}$. It is continuous because the value of $\alpha \star \beta$ depends only on finite prefixes of α and β. The *continuous function application* $\Box \mid \Box\colon \mathbb{B} \times \mathbb{B} \to \mathbb{N} \rightharpoonup \mathbb{N}$ is defined by

$$(\alpha \mid \beta)n = \alpha \star (n{::}\beta)\ .$$

The Baire space $\mathbb{B}$ together with $\mid$ is a partial combinatory algebra, where $\alpha \mid \beta$ is considered to be undefined when $\alpha \mid \beta$ is not a total function, see [13] for details. Every $\alpha \in \mathbb{B}$ represents a partial function $\boldsymbol{\eta}_\alpha\colon \mathbb{B} \rightharpoonup \mathbb{B}$ defined by

$$\boldsymbol{\eta}_\alpha \beta = \alpha \mid \beta\ .$$

We say that a partial map $f\colon \mathbb{B} \rightharpoonup \mathbb{B}$ is *realized* when there exists $\alpha \in \mathbb{B}$ such that $f = \boldsymbol{\eta}_\alpha$. Such an α is called a *realizer* for f. Because $\mid$ is a continuous operation, a realized map is always continuous, although not every partial

continuous map is realized. Recall that a G_δ-set is a set that is equal to a countable intersection of open sets.

Proposition 3.2 *If $U \subseteq \mathbb{B}$ is a G_δ-set then the function $u\colon \mathbb{B} \rightharpoonup \mathbb{B}$ defined by*

$$u\alpha = \begin{cases} \lambda n\colon \mathbb{N}.\, 1 & \alpha \in U\ , \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

is realized.

Proof. The set U is a countable intersection of countable unions of basic open sets, $U = \bigcap_{i\in\mathbb{N}} \bigcup_{j\in\mathbb{N}} a_{i,j}{::}\mathbb{B}$. Define a sequence $\upsilon \in \mathbb{B}$ for all $i, j \in \mathbb{N}$ by $\upsilon(\mathsf{seq}\,(i{::}a_{i,j})) = 2$, and set $\upsilon n = 0$ for all other arguments n. Clearly, if $\boldsymbol{\eta}_\upsilon\alpha$ is total then its value is $\lambda n.\, 1$, so we only need to verify that $\mathsf{dom}(\boldsymbol{\eta}_\upsilon) = U$. If $\alpha \in \mathsf{dom}(\boldsymbol{\eta}_\upsilon)$ then $\upsilon \star (i{::}\alpha)$ is defined for every $i \in \mathbb{N}$, therefore there exists $ci \in \mathbb{N}$ such that $\upsilon(\mathsf{seq}\,(i{::}[\alpha 0, \ldots, \alpha(ci)])) = 2$, which implies that $\alpha \in a_{i,ci}$. Hence $\alpha \in \bigcap_{i\in\mathbb{N}} a_{i,ci}{::}\mathbb{B} \subseteq U$. Conversely, if $\alpha \in U$ then for every $i \in \mathbb{N}$ there exists some $ci \in \mathbb{N}$ such that $\alpha \in a_{i,ci}$. For every $i \in \mathbb{N}$, $\upsilon(\mathsf{seq}\,(i{::}[\alpha 0, \ldots, \alpha(ci)])) = 2$, therefore $(\boldsymbol{\eta}_\upsilon\alpha)i = \upsilon \star (i{::}\alpha) = 1$. Hence $\alpha \in \mathsf{dom}(\boldsymbol{\eta}_\upsilon)$. □

Corollary 3.3 *Suppose $\alpha \in \mathbb{B}$ and $U \subseteq \mathbb{B}$ is a G_δ-set. Then there exists $\beta \in \mathbb{B}$ such that $\boldsymbol{\eta}_\alpha\gamma = \boldsymbol{\eta}_\beta\gamma$ for all $\gamma \in \mathsf{dom}(\boldsymbol{\eta}_\alpha) \cap U$ and $\mathsf{dom}(\boldsymbol{\eta}_\beta) = U \cap \mathsf{dom}(\boldsymbol{\eta}_\alpha)$.*

Proof. By Proposition 3.2 there exists $\upsilon \in \mathbb{B}$ such that for all $\beta \in \mathbb{B}$

$$\boldsymbol{\eta}_\upsilon\beta = \begin{cases} \lambda n\colon \mathbb{N}.\, 1 & \beta \in U\ , \\ \text{undefined} & \text{otherwise}\ . \end{cases}$$

It suffices to show that the function $f\colon \mathbb{B} \rightharpoonup \mathbb{B}$ defined by

$$(f\beta)n = ((\eta_\upsilon\beta)n) \cdot ((\eta_\alpha\beta)n)$$

is realized. This is so because coordinate-wise multiplication of sequences is realized, and so are pairing and composition. □

Theorem 3.4 (Extension Theorem for $\mathbb{B}$) *(a) Every partial continuous map $\mathbb{B} \rightharpoonup \mathbb{B}$ can be extended to a realized one. (b) The realized partial maps $\mathbb{B} \rightharpoonup \mathbb{B}$ are precisely those continuous partial maps whose domains are G_δ-sets.*

Proof. (a) Suppose $f\colon \mathbb{B} \rightharpoonup \mathbb{B}$ is a partial continuous map. Consider the set $A \subseteq \mathbb{N}^* \times \mathbb{N}^2$ defined by

$$A = \big\{ \langle a, i, j\rangle \in \mathbb{N}^* \times \mathbb{N}^2 \;\big|\; a{::}\mathbb{B} \cap \mathsf{dom}(f) \neq \emptyset \text{ and } \forall\, \alpha \in (a{::}\mathbb{B} \cap \mathsf{dom}(f))\,.\,((f\alpha)i = j) \big\}\ .$$

If $\langle a,i,j\rangle \in A$, $\langle a',i,j'\rangle \in A$ and $a \sqsubseteq a'$ then $j = j'$ because there exists $\alpha \in a'::\mathbb{B} \cap \mathsf{dom}(f) \subseteq a::\mathbb{B} \cap \mathsf{dom}(f)$ such that $j = (f\alpha)i = j'$. We define a sequence $\phi \in \mathbb{B}$ as follows. For every $\langle a,i,j\rangle \in A$ let $\phi(\mathsf{seq}\,(i::a)) = j+1$, and for all other arguments let $\phi n = 0$. Suppose that $\phi(\mathsf{seq}\,(i::a)) = j+1$ for some $i,j \in \mathbb{N}$ and $a \in \mathbb{N}^*$. Then for every prefix $a' \sqsubseteq a$, $\phi(\mathsf{seq}\,(i::a')) = 0$ or $\phi(\mathsf{seq}\,(i::a')) = j+1$. Thus, if $\langle a,i,j\rangle \in A$ and $a \sqsubseteq \alpha$ then $\phi \star (i::\alpha) = j$. We show that $(\boldsymbol{\eta}_\phi\alpha)i = (f\alpha)i$ for all $\alpha \in \mathsf{dom}(f)$ and all $i \in \mathbb{N}$. Because f is continuous, for all $\alpha \in \mathsf{dom}(f)$ and $i \in \mathbb{N}$ there exists $\langle a,i,j\rangle \in A$ such that $a \sqsubseteq \alpha$ and $(f\alpha)i = j$. Now we get $(\boldsymbol{\eta}_\phi\alpha)i = (\phi \mid \alpha)i = \phi \star (i::\alpha) = j = (f\alpha)i$.

(b) First we show that $\boldsymbol{\eta}_\alpha$ is a continuous map whose domain is a G_δ-set. It is continuous because the value of $(\boldsymbol{\eta}_\alpha\beta)n$ depends only on n and finite prefixes of α and β. The domain of $\boldsymbol{\eta}_\alpha$ is the G_δ-set

$$\begin{aligned}\mathsf{dom}(\boldsymbol{\eta}_\alpha) &= \big\{\beta \in \mathbb{B} \;\big|\; \forall\, n \in \mathbb{N}.\,((\alpha \mid \beta)n \text{ defined})\big\} \\ &= \bigcap_{n\in\mathbb{N}} \big\{\beta \in \mathbb{B} \;\big|\; (\alpha \mid \beta)n \text{ defined}\big\} = \bigcap_{n\in\mathbb{N}} \bigcup_{m\in\mathbb{N}} \big\{\beta \in \mathbb{B} \;\big|\; \alpha \star (n::\beta) = m\big\}\ .\end{aligned}$$

Each of the sets $\big\{\beta \in \mathbb{B} \;\big|\; \alpha \star (n::\beta) = m\big\}$ is open because $\star$ and $::$ are continuous operations. Now let $f : \mathbb{B} \rightharpoonup \mathbb{B}$ be a partial continuous function whose domain is a G_δ-set. By part (a) of this theorem there exists $\phi \in \mathbb{B}$ such that $f\alpha = \boldsymbol{\eta}_\phi\alpha$ for all $\alpha \in \mathsf{dom}(f)$. By Corollary 3.3 there exists $\psi \in \mathbb{B}$ such that $\mathsf{dom}(\boldsymbol{\eta}_\psi) = \mathsf{dom}(f)$ and $\boldsymbol{\eta}_\psi\alpha = \boldsymbol{\eta}_\phi\alpha$ for every $\alpha \in \mathsf{dom}(f)$. □

A *Baire space representation*, or simply a *representation*, is a partial surjection $\delta_S : \mathbb{B} \rightharpoonup S$, where S is a set. A representation $\delta_S : \mathbb{B} \rightharpoonup S$ of a set S induces a quotient topology on S, defined by

$$U \subseteq S \text{ open} \iff \delta_S^{-1}(U) \text{ open in } \mathsf{dom}(\delta_S)\ .$$

We denote by $\|S\|$ the topological space S with the quotient topology induced by δ_S. A *realized map* $f : (S,\delta_S) \to (T,\delta_T)$ is a function $f : S \to T$ such that there exists a partial continuous map $g : \mathbb{B} \rightharpoonup \mathbb{B}$ which tracks f, meaning that $\mathsf{dom}(f) \subseteq \mathsf{dom}(g)$ and that, for every $\alpha \in \mathsf{dom}(f)$, $f(\delta_S\alpha) = \delta_T(g\alpha)$. A realized map f is always continuous as map $f : \|S\| \to \|T\|$. The category of Baire space representations and realized maps is denoted by $\mathsf{Rep}(\mathbb{B})$.

The category $\mathsf{Rep}(\mathbb{B})$ is equivalent to the PER model $\mathsf{PER}(\mathbb{B})$ where $\mathbb{B}$ is equipped with the structure of the Second Kleene Algebra. The objects of $\mathsf{PER}(\mathbb{B})$ are partial equivalence relations on $\mathbb{B}$. If A is a PER on $\mathbb{B}$ we denote it by A when we think of it as an object and by $=_A$ when we think of it as a binary relation. For $A,B \in \mathsf{PER}(\mathbb{B})$, we say that $\alpha \in \mathbb{B}$ *realizes* a morphism $[\alpha] : A \to B$ when, for all $\beta,\gamma \in \mathbb{B}$, if $\beta =_A \gamma$, then $\alpha \mid \beta$ and $\alpha \mid \gamma$ are defined, and $\alpha \mid \beta =_B \alpha \mid \gamma$. Here α and α' realize the same morphism, $[\alpha] = [\alpha']$, when, for all $\beta,\gamma \in \mathbb{B}$, $\beta =_A \gamma$ implies $\alpha \mid \beta =_B \alpha' \mid \gamma$. The equivalence of $\mathsf{Rep}(\mathbb{B})$ and

$\mathsf{PER}(\mathbb{B})$ assigns to each representation $\delta_S : \mathbb{B} \rightharpoonup S$ the PER $=_S$ defined by

$$\alpha =_S \beta \iff \delta_S(\alpha) = \delta_S(\beta) \ .$$

If $f : (S, \delta_S) \to (T, \delta_T)$ is a realized map in $\mathsf{Rep}(\mathbb{B})$, tracked by $g : \mathbb{B} \rightharpoonup \mathbb{B}$, then by Extension Theorem 3.4 there exists $\alpha \in \mathbb{B}$ such that $\boldsymbol{\eta}_\alpha$ is a continuous extension of g. Under the equivalence $\mathsf{Rep}(\mathbb{B}) \simeq \mathsf{PER}(\mathbb{B})$, the morphism f corresponds to the morphism $[\boldsymbol{\eta}_\alpha]$. The most relevant consequence of this equivalence is that $\mathsf{Rep}(\mathbb{B})$ is a regular locally cartesian closed category, since every PER model on a PCA is such a category [4]. For example, the exponential B^A of PERs $A, B \in \mathsf{PER}(\mathbb{B})$ is defined by

$$\alpha =_{B^A} \alpha' \iff \forall \beta, \gamma \in \mathbb{B} \,.\, (\beta =_A \gamma \Longrightarrow (\alpha \mid \beta)\downarrow =_B (\alpha' \mid \gamma)\downarrow) \ .$$

Unfortunately, this description of exponentials in not very helpful in particular cases, and it completely obscures the topological properties of exponentials. In many important cases better descriptions are available, cf. Theorem 4.5.

In TTE we are typically interested in representations of topological spaces, rather than arbitrary sets. For this reason it is important to represent a topological space X with a representation (X, δ_X) which has a reasonable relation to the topology of X. An obvious requirement is that the original topology of X should coincide with the quotient topology of $\|X\|$. However, as is well known by the school of TTE, this requirement is too weak because it allows ill-behaved representations. A desirable condition on representations of topological spaces is that all continuous maps between them be realized. Thus, we are led to further restricting the allowable representations of topological spaces as follows.

Definition 3.5 An *admissible representation* of a topological space X is a partial continuous quotient map $\delta : \mathbb{B} \rightharpoonup X$ such that every partial continuous map $f : \mathbb{B} \rightharpoonup X$ can be factored through δ. This means that there exists $g : \mathbb{B} \rightharpoonup \mathbb{B}$ such that $f\alpha = \delta(g\alpha)$ for all $\alpha \in \mathsf{dom}(f)$.

The main effect of this definition is that if $\delta_X : \mathbb{B} \rightharpoonup X$ and $\delta_Y : \mathbb{B} \rightharpoonup Y$ are admissible representations, then every continuous map $f : X \to Y$ is realized, and conversely, every realizer that respects δ_X and δ_Y induces a continuous map $X \to Y$.

The requirement that and admissible representation $\delta : \mathbb{B} \rightharpoonup X$ be a quotient map implies that X is a sequential space, since it is a quotient of the sequential space $\mathsf{dom}(\delta)$. It is easy to show that any two admissible representations are isomorphic in $\mathsf{Rep}(\mathbb{B})$. An obvious question to ask is which sequential spaces have admissible representations.

Definition 3.6 Let AdmSeq be the full subcategory of Seq on those sequential T_0-spaces that have admissible representations.[3]

[3] It is believed that the T_0 requirement is inessential for the results proved here, but that

Schröder [22] has characterized AdmSeq as follows.

Definition 3.7 [Schröder [22]] A *pseudobase* for a space X is a family $\mathcal{B}$ of subsets of X such that whenever $\langle x_n \rangle_{n\in\mathbb{N}} \to_{\mathcal{O}(X)} x_\infty$ and $x_\infty \in U \in \mathcal{O}(X)$ then there exists $B \in \mathcal{B}$ such that $x_\infty \in B \subseteq U$ and $\langle x_n \rangle_{n\in\mathbb{N}}$ is eventually in B.

Theorem 3.8 (Schröder [22]) *A sequential T_0-space has an admissible representation if, and only if, it has a countable pseudobase.*

From Schröder's proof of Theorem 3.8 we get a specific admissible representation δ for a T_0-space X with a countable pseudobase $\left\{B_k \;\middle|\; k \in \mathbb{N}\right\}$, defined by

$$\begin{aligned}\delta(\alpha) = x \iff{}& \\ & \forall\, k \in \mathbb{N}\,.\,(x \in B_{\alpha k}) \wedge \forall\, U \in \mathcal{O}(X)\,.\,(x \in U \implies \exists\, k \in \mathbb{N}\,.\,B_{\alpha k} \subseteq U)\ .\end{aligned}$$

The above formula says that α is a δ-representation of x when α enumerates (indices of) a sequence of pseudobasic open neighborhoods of x that get arbitrarily small. In case X is a T_0-space with a countable base $\left\{U_k \;\middle|\; k \in \mathbb{N}\right\}$, we may use an equivalent but simpler admissible representation δ', defined by

$$\delta'(\alpha) = x \iff \left\{U_{\alpha k} \;\middle|\; k \in \mathbb{N}\right\} = \left\{U_n \;\middle|\; n \in \mathbb{N} \wedge x \in U_n\right\}\ .$$

The above formula says that α is a δ'-representation of x when it enumerates the basic open neighborhoods of x.

If $X \in \mathsf{AdmSeq}$ then its admissible representation is determined up to isomorphism in $\mathsf{Rep}(\mathbb{B})$. Therefore, AdmSeq is equivalent to the full subcategory of $\mathsf{Rep}(\mathbb{B})$ on the admissible representations, so that AdmSeq can be thought of as a subcategory of $\mathsf{Rep}(\mathbb{B})$. The following result by Schröder [22] tells us that the inclusion of AdmSeq into $\mathsf{Rep}(\mathbb{B})$ preserves the cartesian closed structure.

Theorem 3.9 (Schröder [22]) *Let (X, δ_X) and (Y, δ_Y) be admissible representations for sequential T_0-spaces X and Y. Then the product $(X, \delta_X) \times (Y, \delta_Y)$ formed in $\mathsf{Rep}(\mathbb{B})$ is an admissible representation of the product $X \times Y$ formed in Seq, and similarly the exponential $(Y, \delta_Y)^{(X,\delta_X)}$ formed in $\mathsf{Rep}(\mathbb{B})$ is an admissible representation for the exponential Y^X formed in Seq.*

4 $\mathsf{Rep}(\mathbb{B})$ as a subcategory of Equ

In this section we describe $\mathsf{Rep}(\mathbb{B})$ as a full subcategory of equilogical spaces. We then study the properties of the inclusion $\mathsf{Rep}(\mathbb{B}) \to \mathsf{Equ}$.

Definition 4.1 A 0-*equilogical space* is an equilogical space whose underlying topological space is 0-dimensional. The category $\mathsf{0Equ}$ is the full subcategory of Equ on 0-equilogical spaces.

has not been checked yet.

Thus $\mathsf{0Equ}$ is formed just like Equ, where we use $\mathsf{0Dim}$ instead of $\omega\mathsf{Top}_0$.

Theorem 4.2 *The categories* $\mathsf{0Equ}$, $\mathsf{Rep}(\mathbb{B})$, *and* $\mathsf{PER}(\mathbb{B})$ *are equivalent.*

Proof. We show that $\mathsf{0Equ}$ and $\mathsf{PER}(\mathbb{B})$ are equivalent, since we already know that $\mathsf{PER}(\mathbb{B})$ and $\mathsf{Rep}(\mathbb{B})$ are equivalent. By Embedding Theorem 3.1 for $\mathbb{B}$, a countably based T_0-space is 0-dimensional if, and only if, it embeds in $\mathbb{B}$. Thus every 0-equilogical space is isomorphic to one whose underlying topological space is a subspace of $\mathbb{B}$. This make it clear that equivalence relations on 0-dimensional countably based T_0-spaces correspond to partial equivalence relations on $\mathbb{B}$. Morphisms work out, too, since by the Extension Theorem for $\mathbb{B}$ 3.4 every partial continuous map on $\mathbb{B}$ can be extended to a realized one. □

The inclusion functor $I\colon \mathsf{0Equ} \to \mathsf{Equ}$ has a right adjoint $D\colon \mathsf{Equ} \to \mathsf{0Equ}$, which is defined as follows. For every countably based T_0-space X there exists an admissible representation $\delta_X\colon \mathbb{B} \rightharpoonup X$. The subspace $X_0 = \mathsf{dom}(\delta) \subseteq \mathbb{B}$ is a countably based 0-dimensional Hausdorff space. Now if $X = (|X|, \equiv_X)$ is an equilogical space, let $DX = (X_0, \equiv_{DX})$ where $a \equiv_{DX} b$ if, and only if, $\delta_X a \equiv_X \delta_X b$. If $f\colon X \to Y$ is a morphism in Equ, tracked by $g\colon |X| \to |Y|$, then Df is the morphism tracked by a continuous map $h\colon X_0 \to Y_0$ that tracks $g\colon X \to Y$, as shown in the following commutative diagram:

$$\begin{array}{ccc} X_0 & \xrightarrow{\;h\;} & Y_0 \\ {\scriptstyle\delta_X}\downarrow & & \downarrow{\scriptstyle\delta_Y} \\ X & \xrightarrow[\;g\;]{} & Y \end{array}$$

Such a map h exists because δ_X and δ_Y were chosen to be admissible representations. The main properties of the adjoints $I \dashv D$ are summarized in the following theorem.

Theorem 4.3

(i) *Functors I and D are a section and a retraction, i.e., $D \circ I$ is naturally equivalent to 1_{0Equ}.*

(ii) *I is full and faithful and preserves countable colimits and limits (which are precisely all the limits and colimits that exist in Equ).*

(iii) *D is faithful and preserves countable limits and colimits (which are precisely all the limits and colimits that exist in $\mathsf{0Equ}$).*

(iv) *D is not full, but its restriction to EPQ_0 is full.*

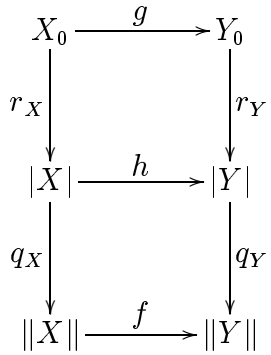
Proof. (i) This follows by a general category-theoretic argument from the fact that I is full and faithful, cf. the dual of [11, Proposition 3.4.1].

(ii) It is obvious that I is full and faithful since it is just the inclusion functor of a full subcategory. It preserves colimits because it is a left adjoint,

and it preserves limits because the inclusion $\mathsf{0Dim} \to \omega\mathsf{Top}_0$ does.

(iii) It is obvious that D is faithful, and it preserves limits because it is a right adjoint. That D preserves finite colimits can be verified explicitly, and it also follows from [17, Proposition 2.5.11]. That D preserves countable coproducts holds because a countable coproduct of admissible representations is again an admissible representation.

(iv) If D were full then by [11, Proposition 3.4.3] it would follow that the counit of the adjunction $\eta\colon I \circ D \to 1_{\mathsf{Equ}}$ is a natural isomorphism, which obviously is not the case. For example, $\eta_{\mathbb{R}}$ is not a natural isomorphism, where $\mathbb{R}$ are the real numbers equipped with the Euclidean topology, because every morphism $\mathbb{R} \to I(D\mathbb{R})$ is constant, as it must be tracked by a continuous map from $\mathbb{R}$ into the 0-dimensional Hausdorff space $|I(D\mathbb{R})|$. However, when D is restricted to EPQ_0 then we can show that it is full as follows. Suppose $X, Y \in \mathsf{EPQ}_0$, and let $r_X\colon X_0 \to |X|$ and $r_Y\colon Y_0 \to |Y|$ be admissible representations. Suppose $f\colon DX \to DY$ is a morphism tracked by a continuous map $g\colon X_0 \to Y_0$. The situation is shown in the following diagram:



Because q_Y is ω-projecting, f is tracked by an arrow $h\colon |X| \to |Y|$ so that the lower square commutes. Therefore f is a morphism in Equ, hence $Df = f$. □

Remark 4.4 Since I and D both preserve all limits and colimits that exist, one wonders whether they have any further adjoints.[4] This does not seem to be the case. One might try embedding the categories Equ and $\mathsf{Rep}(\mathbb{B})$ into larger categories and extending I and D, in hope that the "missing" adjoint can be obtained that way. This idea was worked out in [2] for a general applicative retraction $I \dashv D$ between PER models. The PER models were embedded into suitable toposes of sheaves over PCAs. The adjunction $I \dashv D$ then extends to an adjunction at the level of toposes, with a further right adjoint. This makes it possible to apply the logical transfer principle from [3] to show that a certain class of first-order sentences is valid in the internal logic of Equ if, and only if, it is valid in the internal logic of $\mathsf{Rep}(\mathbb{B})$.

The next question to ask is whether I and D preserve any exponentials.

[4] Note that Equ and $\mathsf{0Equ}$ are only *countably* complete and cocomplete so that we cannot directly apply the Adjoint Functor Theorem.

Theorem 4.5

(i) *Functor D restricted to EPQ_0 preserves exponentials.*

(ii) *If $X, Y \in \mathsf{0Equ}$ and there exists in $\omega\mathsf{Top}_0$ a 0-dimensional weak exponential of $|X|$ and $|Y|$, then I preserves the exponential Y^X.*

(iii) *Functor I preserves the natural numbers object $\mathbb{N}$, the exponentials $\mathbb{N}^{\mathbb{N}}$ and $2^{\mathbb{N}}$, and the object $\mathbb{R}_c$ of Cauchy reals.*

(iv) *Functor I does* not *preserve exponentials in general. In particular, it does not preserve $\mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$.*

Proof. (i) This follows from results obtained in Section 5, and so we postpone the proof until then. It can be found on page 16.

(ii) If $W \in \mathsf{0Dim}$ is a weak exponential of X and Y in $\omega\mathsf{Top}_0$, then it is also a weak exponential of X and Y in $\mathsf{0Dim}$. Therefore, the construction of Y^X from W in Equ, as described in Section 2 coincides with the one in $\mathsf{0Equ}$.

(iii) The Baire space $\mathbb{N}^{\mathbb{N}}$ and the Cantor space $2^{\mathbb{N}}$ both satisfy the condition from (ii). The real numbers object $\mathbb{R}_c$ is a regular quotient of $\mathbb{N} \times 2^{\mathbb{N}}$ [4, Proposition 5.5.3], and the left adjoint I preserves it because it preserves $\mathbb{N}$, $2^{\mathbb{N}}$, products, and coequalizers.

(iv) Let $X = \mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$ in $\mathsf{0Equ}$, and let $Y = \mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$ in Equ. The space $|X|$ is a Hausdorff space. The space $|Y|$ is the subspace of the total elements of the Scott domain $D_Y = [{\mathbb{N}_\perp}^\omega \to \mathbb{N}_\perp]$. The equivalence relation on $|Y|$ is the consistency relation of D_Y restricted to $|Y|$. Suppose $f\colon |Y| \to |X|$ represented an isomorphism, and let $g\colon |X| \to |Y|$ represent its inverse. Because f is monotone in the specialization order and $|X|$ has a trivial specialization order, $a \equiv_Y b$ implies $fx = fy$. Therefore, $g \circ f\colon |Y| \to |Y|$ is an equivariant retraction. By [4, Proposition 4.1.8], Y is a topological object. By [4, Corollary 4.1.9], this would mean that the topological quotient $\|Y\|$ is countably based, but it is not, as is well known. Another way to see that Y cannot be topological is to observe that Y is an exponential of the Baire space, but the Baire space is not exponentiable in $\omega\mathsf{Top}_0$, and in particular $\mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$ is not a topological object in Equ. □

Remark 4.6 In [2] we used a logical transfer principle between Equ and $\mathsf{Rep}(\mathbb{B})$ to prove that I does not preserve ${\mathbb{R}_c}^{\mathbb{R}_c}$ either.

As already mentioned in the introduction, we could obtain the results of this section by applying Longley's theory of applicative adjunctions between applicative morphisms of partial combinatory algebras [17]. Lietz [16] used this approach to compare the realizability toposes $\mathsf{RT}(\mathcal{P}\mathbb{N})$ and $\mathsf{RT}(\mathbb{B})$.

5 A Common Subcategory of Equ and Rep($\mathbb{B}$)

In Sections 2 and 3 we saw that sequential spaces contain cartesian closed subcategories PQ_0 and AdmSeq which are also cartesian closed subcategories

of Equ and Rep($\mathbb{B}$), respectively. In this section we prove that PQ_0 and AdmSeq are the same category.

Lemma 5.1 *Suppose $\mathcal{B} = \{B_i \mid i \in \mathbb{N}\}$ is a countable pseudobase for a countably based T_0-space Y. Let X be a first-countable space and $f\colon X \to Y$ a continuous map. For every $x \in X$ and every neighborhood V of fx there exists a neighborhood U of x and $i \in \mathbb{N}$ such that $fx \in f(U) \subseteq B_i \subseteq V$.*

Proof. Note that the elements of the pseudobase do not have to be open sets, so this is not just a trivial consequence of continuity of f. We prove the lemma by contradiction. Suppose there were $x \in X$ and a neighborhood V of fx such that for every neighborhood U of x and for every $i \in \mathbb{N}$, if $B_i \subseteq V$ then $f_*(U) \not\subseteq B_i$. Let $U_0 \supseteq U_1 \supseteq \cdots$ be a descending countable neighborhood system for x. Let $p\colon \mathbb{N} \to \mathbb{N}$ be a surjective map that attains each value infinitely often, that is for all $k, j \in \mathbb{N}$ there exists $i \geq k$ such that $pi = j$. For every $i \in \mathbb{N}$, if $B_{pi} \subseteq V$ then $f_*(U_i) \not\subseteq B_{pi}$. Therefore, for every $i \in \mathbb{N}$ there exists $x_i \in U_i$ such that if $B_{pi} \subseteq V$ then $fx_i \notin B_{pi}$. The sequence $\langle x_n \rangle_{n\in\mathbb{N}}$ converges to x, hence $\langle fx_n \rangle_{n\in\mathbb{N}}$ converges to fx. Because $\mathcal{B}$ is a pseudobase there exists $j \in \mathbb{N}$ such that $B_j \subseteq V$ and $\langle fx_n \rangle_{n\in\mathbb{N}}$ is eventually in B_j, say from the k-th term onwards. There exists $i \geq k$ such that $pi = j$. Now we get $fx_i \in B_{pi} \subseteq V$, which is a contradiction. □

Theorem 5.2 PQ_0 *and* AdmSeq *are the same category.*

Proof. It was independently observed by Schröder that PQ_0 is a full subcategory of AdmSeq, which is the easier of the two inclusions. The proof goes as follows. Suppose $q\colon X \to Y$ is an ω-projecting quotient map. We need to show that Y is a sequential space with an admissible representation. It is sequential because it is a quotient of a sequential space. There exists an admissible representation $\delta_X\colon \mathbb{B} \rightharpoonup X$. Let $\delta_Y = q \circ \delta_X$. Suppose $f\colon \mathbb{B} \rightharpoonup Y$ is a continuous partial map. Because q is ω-projecting f lifts though X, and because δ_X is an admissible representation, it further lifts through $\mathbb{B}$.

It remains to prove the converse, namely that if a sequential T_0-space X has an admissible representation then there exists an ω-projecting quotient $q\colon Y \to X$. Since X has an admissible representation it has a countable pseudobase $\mathcal{B} = \{B_i \mid i \in \mathbb{N}\}$, by Theorem 3.8. The powerset $\mathcal{P}\mathbb{N}$ ordered by inclusion is an algebraic lattice. We equip it with the Scott topology, which is generated by the subbasic open sets $\uparrow n = \{a \in \mathcal{P}\mathbb{N} \mid n \in a\}$, $n \in \mathbb{N}$. Let $q\colon \mathcal{P}\mathbb{N} \rightharpoonup X$ be a partial map defined by

$$qa = x \iff (\forall\, n \in a\,.\, x \in B_n) \wedge \forall\, U \in \mathcal{O}(X)\,.\,(x \in U \implies \exists\, n \in a\,.\, B_n \subseteq U)\ .$$

The map q is well defined because $qa = x$ and $qa = y$ implies that x and y share the same neighborhoods, so they are the same point of the T_0-space X. Furthermore, q is surjective because $\mathcal{B}$ is a pseudobase. To see that p is

continuous, suppose $pa = x$ and $x \in U \in \mathcal{O}(X)$. There exists $n \in \mathbb{N}$ such that $x \in B_n \subseteq U$. If $n \in b \in \mathsf{dom}(p)$ then $pb \in B_n \subseteq U$. Therefore, $a \in \uparrow n$ and $p_*(\uparrow n) \subseteq B_n \subseteq U$, which means that p is continuous. Let $Y = \mathsf{dom}(p)$.

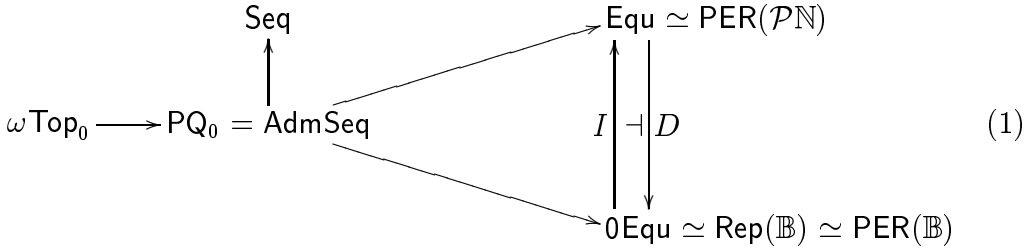
Let us show that $q\colon Y \to X$ is ω-projecting. Suppose $f\colon Z \to X$ is a continuous map and $Z \in \omega\mathsf{Top}_0$. Define a map $g\colon Z \to \mathcal{P}\mathbb{N}$ by

$$gz = \left\{ n \in \mathbb{N} \;\middle|\; \exists U \in \mathcal{O}(Z)\,.\,(z \in U \wedge f_*(U) \subseteq B_n) \right\} .$$

The map g is continuous almost by definition. Indeed, if $gz \in \uparrow n$ then there exists a neighborhood U of z such that $f_*(U) \subseteq B_n$, but then $g_*(U) \in \uparrow n$. To finish the proof we need to show that $fz = p(gz)$ for all $z \in Z$. If $n \in gz$ then $fz \in B_n$ because there exists $U \in \mathcal{O}(Z)$ such that $z \in U$ and $f_*(U) \subseteq B_n$. If $fz \in V \in \mathcal{O}(X)$ then by Lemma 5.1 there exists $U \in \mathcal{O}(Z)$ and $n \in \mathbb{N}$ such that $z \in U$ and $f_*(U) \subseteq B_n \subseteq U$. Hence, $n \in gz$. This proves that $fz = p(gz)$. □

Remark 5.3 Matthias Schröder has showed recently that if a sequential T_0-space X arises as a topological quotient of a subspace of $\mathbb{B}$, then X has an admissible representation. This result implies Theorem 5.2, and also gives a very nice characterization of EPQ_0: it is precisely the category of all T_0-spaces that are topological quotients of countably based T_0-spaces.

The relationships between the categories are summarized by the following diagram:



(1)

The unlabeled arrows are full and faithful inclusions, preserve countable limits, and countable coproducts. The inclusion $\omega\mathsf{Top}_0 \to \mathsf{PQ}_0$ preserves all exponentials that happen to exist in $\omega\mathsf{Top}_0$, and the other three unlabeled inclusions preserve cartesian closed structure. The right-hand triangle involving the two inclusions and the coreflection D commutes up to natural isomorphism (and the one involving the inclusion I does not).

We still owe the proof of Theorem 4.5(i), namely, that D restricted to EPQ_0 preserves exponentials. But this is now obvious, since the right-hand triangle involving D commutes.

6 Transfer Results between Equ and Rep(B)

The correspondence (1) explains why domain-theoretic computational models agree so well with computational models studied by TTE—as long as we

only build spaces by taking products, coproducts, exponentials, and regular subspaces, starting from countably based T_0-spaces, we remain in PQ_0, the common cartesian closed core of equilogical spaces and TTE.

As a first example of a transfer result, we translate a characterization of Kleene-Kreisel countable functionals [12] from Equ to $\mathsf{Rep}(\mathbb{B})$. In [6] we proved that the iterated exponentials $\mathbb{N}$, $\mathbb{N}^{\mathbb{N}}$, $\mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$, ... of the natural numbers object $\mathbb{N}$ in Equ are precisely the Kleene-Kreisel countable functionals. Because $\mathbb{N}$ is the natural numbers object in $\mathsf{Rep}(\mathbb{B})$ as well, and it belongs to PQ_0, the same hierarchy appears in $\mathsf{Rep}(\mathbb{B})$.

Proposition 6.1 *In* $\mathsf{Rep}(\mathbb{B})$*, the hierarchy of exponentials* $\mathbb{N}$, $\mathbb{N}^{\mathbb{N}}$, $\mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$, ... *, built from the natural numbers object* $\mathbb{N}$*, corresponds to the Kleene-Kreisel countable functionals.*

As a second example, we consider transfer between the *internal logics* of Equ and $\mathsf{Rep}(\mathbb{B})$. Because Equ and $\mathsf{Rep}(\mathbb{B})$ are equivalent to realizability models $\mathsf{PER}(\mathcal{P}\mathbb{N})$ and $\mathsf{PER}(\mathbb{B})$, respectively, they admit a realizability interpretation of first-order intuitionistic logic. This has been worked out in detail in [4]. It is often advantageous to work in the internal logic, because it lets us argue abstractly and conceptually about objects and morphisms. We never have to mention explicitly the realizers of morphisms or the underlying topological spaces, which makes arguments more perspicuous. Every map that can be defined in the internal logic is automatically realized (and computable, if we work with the computable versions of the realizability models).

Suppose we want to use internal logic to construct a particular map $f\colon X \to Y$ where $X, Y \in \mathsf{PQ}_0$. For example, we might want to define the definite integration operator $I\colon \mathbb{R}^{[0,1]} \to \mathbb{R}$,

$$If = \int_0^1 f(x)\,dx\ .$$

It may happen that X and Y are much more amenable to the internal logic of $\mathsf{Rep}(\mathbb{B})$ than to the internal logic of Equ, or vice versa. In such a case we can pick whichever internal logic is better and work in it, because if a map $f\colon X \to Y$ is definable in one internal logic, then it exists as a morphism in both Equ and $\mathsf{Rep}(\mathbb{B})$.

Let us see how this applies in the case of definite integration. The real numbers $\mathbb{R}$ are much better behaved in $\mathsf{Rep}(\mathbb{B})$ than in Equ, because $\mathbb{R}$ can be characterized in the internal logic of $\mathsf{Rep}(\mathbb{B})$ as *the Cauchy complete Archimedean field*, which gives us all the properties of $\mathbb{R}$ we could wish for. On the other hand, in the internal logic of Equ, $\mathbb{R}$ does not seem to be characterizable at all, and it does not even satisfy the Archimedean axiom

$$\forall\, x \in \mathbb{R}\,.\,\exists\, n \in \mathbb{N}\,.\,x < n\ ,$$

because in Equ there is no *continuous* choice map $c\colon \mathbb{R} \to \mathbb{N}$ that would

satisfy $x < cx$ for all $x \in \mathbb{R}$.[5] This makes it impractical to argue about $\mathbb{R}$ in the internal logic of Equ. The situation with the space $\mathbb{R}^{[0,1]}$ of continuous real function on the unit interval is similar—it is much better behaved in the internal logic of $\mathsf{Rep}(\mathbb{B})$ than in the internal logic of Equ. In particular, in $\mathsf{Rep}(\mathbb{B})$ the statement "every map $f\colon [0,1] \to \mathbb{R}$ is uniformly continuous" is valid, whereas it is not valid in the internal logic of Equ. This makes it clear that the internal logic of $\mathsf{Rep}(\mathbb{B})$ is the better choice. Indeed, in the internal logic of $\mathsf{Rep}(\mathbb{B})$ definite integral may be defined in the usual way as a limit of Riemann sums. The convergence of Riemann sums can then be proved constructively because $\mathsf{Rep}(\mathbb{B})$ "believes" that all maps from $[0,1]$ to $\mathbb{R}$ are uniformly continuous. Once we have constructed the definite integral operator $I\colon \mathbb{R}^{[0,1]} \to \mathbb{R}$ in $\mathsf{Rep}(\mathbb{B})$, we can transfer it to Equ via PQ_0.

7 Conclusion

Let me conclude by commenting on the following comparison of domain theory and TTE from Weihrauch's recently published book on computable analysis [27, Section 9.8, p. 267]:

> "The domain approach developed so far is consistent with TTE. Roughly speaking, a domain (for the real numbers) contains approximate objects as well as precise objects which are treated in separate sets in TTE. A computable domain function must map also all approximate objects reasonably. In many cases, constructing a domain which corresponds to given representation still is a difficult task. Concepts for handling multi-valued functions and for computational complexity have not yet been developed for the domain approach. The elegant handling of higher type functions in domain theory can be simulated in TTE by means of function space representations $[\delta \to \delta']$ (Definition 3.3.13). To date, there seems to be no convincing reason to learn domain theory as a prerequisite for computable analysis."

The present paper provides a precise mathematical comparison of TTE and the domain approach, as exemplified by equilogical spaces. The correspondence (1) gives us a clear picture about the relationships between the domain approach and TTE. Overall, it supports the claim that these two approaches are consistent, at least as far as computability on PQ_0 is concerned.

Indeed, domains are built from the approximate as well as the precise objects, and I join Weihrauch in pointing out that it is a good idea to distinguish the precise objects from the approximate ones. In domain theory this is most easily done by taking seriously domains *with totality*, or more generally PERs on domains, which leads to the notion of equilogical spaces and domain representations, which were studied by Blanck [10].

[5] The Archimedean axiom is valid in $\mathsf{Rep}(\mathbb{B})$ because there is a continuous choice map $|D\mathbb{R}| \to \mathbb{N}$ such that $[a] < ca$ for all $a \in |D\mathbb{R}|$, where $[a]$ the real number represented by the realizer a. The point is that ca may depend on the realizer a.

I hope that the adjoint functors I and D between Equ and $\mathsf{Rep}(\mathbb{B})$ will ease the task of constructing a domain which corresponds to a given representation.

Power-domains are the domain-theoretic models of non-deterministic computation, and I believe they could be used to model multi-valued functions.

In this paper we did not consider the computational complexity or even computability in Equ and $\mathsf{Rep}(\mathbb{B})$. In [4] the inclusion $\mathsf{Rep}(\mathbb{B}) \to \mathsf{Equ}$ and its coreflection are constructed for the computable versions of equilogical spaces and TTE, from which we may conclude that computability in domain theory is essentially the same as in TTE.

By Theorem 4.5, the higher type function spaces in equilogical spaces do not generally agree with the corresponding function space representations in TTE. However, the two approaches to higher types do agree on an important class of spaces, namely the category PQ_0, which contains all countably based T_0-spaces, therefore also all countably based continuous and algebraic domains. Higher types seem not to catch a lot of interest in the TTE community. This may be because the descriptions of higher types in terms of representations can get quite unwieldy and are hard to work with. The theory of cartesian closed categories and the internal logic of $\mathsf{Rep}(\mathbb{B})$ ought to be helpful here, as they allows us to talk about the higher types abstractly, without having to refer to their representations all the time. After all, higher types cannot be ignored in computable analysis: real numbers are a quotient of type 1, integration and differentiation operators have type 2, solving a differential equation is a type 3 process, and still higher types are reached when we study spaces of distributions and operators on Hilbert spaces.

Finally, is there a convincing reason to learn domain theory as a prerequisite for computable analysis? By Theorem 4.2, $\mathsf{Rep}(\mathbb{B})$ is a full subcategory of Equ. This may suggest the view that the domain approach is more general than TTE. At any rate, they are *not* competing approaches. They fit with each other very well, and each has its advantages: domain theory handles higher types more elegantly and is more general than TTE, whereas TTE provides a more convenient internal logic and handles questions about computational complexity better. So why not learn both, and a bit of category theory, realizability, and constructive logic on top?

References

[1] Amadio, R. and P.-L. Curien, "Domains and Lambda-Calculi," Cambridge Tracts in Theoretical Computer Science **46**, Cambridge University Press, 1998.

[2] Awodey, S. and A. Bauer, *Sheaf toposes for realizability* (2000), available at `http://andrej.com/papers`.

[3] Awodey, S., L. Birkedal and D. Scott, *Local realizability toposes and a modal logic for computability*, in: L. Birkedal, J. van Oosten, G. Rosolini and D. Scott,

editors, *Tutorial Workshop on Realizability Semantics, FLoC'99, Trento, Italy, 1999*, Electronic Notes in Theoretical Computer Science **23** (1999).

[4] Bauer, A., "The Realizability Approach to Computable Analysis and Topology," Ph.D. thesis, Carnegie Mellon University (2000), available as CMU technical report `CMU-CS-00-164` and at `http://andrej.com/thesis`.

[5] Bauer, A. and L. Birkedal, *Continuous functionals of dependent types and equilogical spaces*, in: *Computer Science Logic 2000*, 2000, available at `http://andrej.com/papers`.

[6] Bauer, A., L. Birkedal and D. Scott, *Equilogical spaces*, Preprint submitted to Elsevier (1998).

[7] Berger, U., *Total sets and objects in domain theory*, Annals of Pure and Applied Logic **60** (1993), pp. 91–117, available at `http://www.mathematik.uni-muenchen.de/~berger/articles/apal/diss.dvi.Z`.

[8] Berger, U., "Continuous Functionals of Dependent and Transitive Types," Habilitationsschrift, Ludwig-Maximilians-Universität München (1997).

[9] Berger, U., *Effectivity and density in domains: A survey*, , **23** (2000).

[10] Blanck, J., "Computability on Topological Spaces by Effective Domain Representations," Ph.D. thesis, Department of Mathematics, Uppsala University (1997).

[11] Borceux, F., "Handbook of Categorical Algebra I. Basic Category Theory," Encyclopedia of Mathematics and Its Applications **51**, Cambridge University Press, 1994.

[12] Kleene, S., *Countable functionals*, in: *Constructivity in Mathematics*, 1959, pp. 81–100.

[13] Kleene, S. and R. Vesley, "The Foundations of Intuitionistic Mathematics, especially in relation to recursive functions," North-Holland Publishing Company, 1965.

[14] Kreitz, C. and K. Weihrauch, *Theory of representations*, Theoretical Computer Science **38** (1985), pp. 35–53.

[15] Kuratowski, C., "Topologie," Warszawa, 1952.

[16] Lietz, P., *Comparing realizability over $\mathcal{P}\omega$ and K_2* (1999), available at `http://www.mathematik.tu-darmstadt.de/~lietz/comp.ps.gz`.

[17] Longley, J., "Realizability Toposes and Language Semantics," Ph.D. thesis, University of Edinburgh (1994).

[18] Menni, M. and A. Simpson, *The largest topological subcategory of countably-based equilogical spaces*, in: *Preliminary Proceedings of MFPS XV*, 1999, available at `http://www.dcs.ed.ac.uk/home/als/Research/`.

[19] Menni, M. and A. Simpson, *Topological and limit-space subcategories of countably-based equilogical spaces* (2000), submitted to Math. Struct. in Comp. Science.

[20] Normann, D., *Categories of domains with totality* (1998), available at `http://www.math.uio.no/~dnormann/`.

[21] Normann, D., *The continuous functionals of finite types over the reals*, Preprint Series 19, University of Oslo (1998).

[22] Schröder, M., *Admissible representations of limit spaces*, in: J. Blanck, V. Brattka, P. Hertling and K. Weihrauch, editors, *Computability and Complexity in Analysis*, Informatik Berichte **272** (2000), pp. 369–388, cCA2000 Workshop, Swansea, Wales, September 17–19, 2000.

[23] Scott, D., *A new category?* (1996), unpublished Manuscript. Available at `http://www.cs.cmu.edu/Groups/LTC/`.

[24] Stoltenberg-Hansen, V., I. Lindström and E. Griffor, "Mathematical Theory of Domains," Number 22 in Cambridge Tracts in Computer Science, Cambridge University Press, 1994.

[25] Weihrauch, K., *Type 2 recursion theory*, Theoretical Computer Science **38** (1985), pp. 17–33.

[26] Weihrauch, K., "Computability," EATCS Monographs on Theoretical Computer Science **9**, Springer, Berlin, 1987.

[27] Weihrauch, K., "Computable Analysis," Springer-Verlag, 2000.

MFPS 17 Preliminary Version

Transfer Principles for Reasoning About Concurrent Programs

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

In previous work we have developed a *transition trace* semantic framework, suitable for shared-memory parallel programs and asynchronously communicating processes, and abstract enough to support compositional reasoning about safety and liveness properties. We now use this framework to formalize and generalize some techniques used in the literature to facilitate such reasoning. We identify a *sequential-to-parallel transfer theorem* which, when applicable, allows us to replace a piece of a parallel program with another code fragment which is *sequentially* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected. Two code fragments are said to be sequentially equivalent if they satisfy the same partial and total correctness properties. We also specify both coarse-grained and fine-grained version of trace semantics, assuming different degrees of atomicity, and we provide a *coarse-to-fine-grained transfer theorem* which, when applicable, allows replacement of a code fragment by another fragment which is *coarsely* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected even if we assume fine-grained atomicity. Both of these results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system which would normally require the use of a more sophisticated semantic model.

1 Introduction

It is well known that syntax-directed reasoning about behavioral properties of parallel programs tends to be complicated by the combinatorial explosion

[1] This research is sponsored in part by the National Science Foundation (NSF) under Grant No. CCR-9988551. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the official policies, either expressed or implied, of the NSF or the U.S. government.

[2] Email: brookes@cs.cmu.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

inherent in keeping track of dynamic interactions between code fragments. Simple proof methodologies based on state-transformation semantics, such as Hoare-style logic, do not adapt easily to the parallel setting, because they abstract away from interaction and only retain information about the initial and final states observed in a computation. A more sophisticated semantic model is required, in which an accurate account can be given of interaction.

Trace semantics provides a mathematical framework in which such reasoning may be carried out [2,3,4,5]. The trace set of a program describes all possible patterns of interaction between the program and its "environment", assuming fair execution [9]. One can define both a *coarse-grained* trace semantics, in which assignment and boolean expression evaluation are assumed to be executed atomically, and a *fine-grained* trace semantics, in which reads and writes (to shared variables) are assumed to be atomic. Trace semantics can be defined denotationally, and is *fully abstract* with respect to a notion of program behavior which subsumes partial correctness, total correctness, safety properties, and liveness properties [2].

To some extent program proofs may be facilitated by a number of laws of program equivalence, validated by trace semantics, which allow us to deduce properties of a program by analyzing instead a semantically equivalent program with simpler structure. The use of a succinct and compact notation for trace sets (based on extended regular expressions) can also help streamline program analysis. Yet the problem remains that in general the trace set of a program can be difficult to manipulate and hard to use to establish correctness properties. Trace sets tend to be rather complex mathematical objects, since a trace set describes *all* possible interactions between the program and *any* potential environment. For the same reason, both the coarse- and the fine-grained trace semantics induce a rather discriminating notion of semantic equivalence, and few laws of equivalence familiar from the sequential setting also hold in all parallel contexts. It can therefore be difficult to establish trace equivalence of programs merely by direct manipulation of the semantic definitions, or by using trace-theoretic laws of program equivalence in a syntax-directed manner.

In practice, parallel systems ought to be designed carefully to ensure that the interactions between component processes are highly disciplined and constrained. Moreover, when analyzing the properties of code to be run in tightly controlled contexts, we ought to be able to work within a simpler semantic model (or, at least, within a reduced subset of the trace semantics) whose simplicity reflects this discipline. Correspondingly, whenever we know that a program fragment will be used in a limited form of context, we would like to be able to employ forms of reasoning which take advantage of the limitations.

For example, we might know that a piece of code is going to be used "sequentially" inside a parallel program (in a manner to be made precise soon) and want to use Hoare-style reasoning about this code in establishing safety and liveness properties of the whole program. It is not generally safe

to do so, since laws of program equivalence that hold in the sequential setting cease to be valid in parallel languages because of the potential for interference between concurrently executing code. Yet *local variables* can only be accessed by processes occurring within a syntactically prescribed scope, and cannot be changed by any other processes running concurrently, so we ought to be able to take advantage of this non-interference property to simplify reasoning about code which only affects local variables. In particular, when local variables are only ever used sequentially, in a context whose syntactic structure guarantees that no more than one process ever gains concurrent access, we should be able to employ Hoare-style reasoning familiar from the sequential setting. We would like to know the extent to which this idea can be made precise, and when this technique is applicable.

In a similar vein, it is usually regarded as realistic to assume fine-grained atomicity when trying to reason about program behavior, but more convenient to make the less realistic but simplifying assumption of coarse granularity, since this assumption may help to reduce the combinatorial explosion. We would like to be able to identify conditions under which it is safe to do so.

A number of *ad hoc* techniques have been proposed along these lines in the literature, usually without detailed consideration of semantic foundations [1]. Their common aim is to facilitate concurrent program analysis by allowing replacement of a code fragment by another piece of code with "simpler" behavioral properties that permit an easier correctness proof.

In this paper we use the trace-theoretic framework to formalize and generalize some of these techniques. By paying careful attention to the underlying semantic framework we are able to recast these techniques in a more precise manner and we can be more explicit about the (syntactic and semantic) assumptions upon which their validity rests. Since these techniques allow us to deduce program equivalence properties based on one semantic model by means of reasoning carried out on top of a different semantic model, we refer to our results as *transfer principles*. We provide transfer principles specifically designed to address the two example scenarios used for motivation above: a *sequential-to-parallel* transfer principle allowing use of Hoare-style reasoning, and a *coarse-to-fine* transfer principle governing the use of coarse semantics in fine-grained proofs of correctness.

Our work can be seen as further progress towards a theory of *context-sensitive development of parallel programs*, building on earlier work of Cliff Jones [8] and spurred on by the recent Ph. D. thesis of Jüergen Dingel [7]. We focus our attention initially on some methodological ideas presented in Greg Andrews's book on concurrent programming [1]. Later we intend to explore more fully the potential of our framework as a basis for further generalization and to extend our results to cover some of the contextual refinement ideas introduced by Dingel.

In this preliminary version of the paper we omit explicit details of the underlying trace semantics, which the reader can find in [2], and we omit

most of the proofs, which require detailed use of the semantic definitions.

2 Syntax

2.1 The programming language

Our parallel programming language is described by the following abstract grammar for commands c, in which b ranges over boolean-valued expressions, e over integer-valued expressions, x over identifiers, a over atomic commands (finite sequences of assignments), and d over declarations. The syntax for expressions is conventional and is assumed to include the usual primitives for arithmetic and boolean operations.

$$\begin{aligned} c ::= {} & \mathbf{skip} \mid x{:=}e \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \\ & \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{local}\ d\ \mathbf{in}\ c \mid \\ & \mathbf{await}\ b\ \mathbf{then}\ a \mid\ c_1 \| c_2 \\ d ::= {} & x = e \mid d_1; d_2 \\ a ::= {} & \mathbf{skip} \mid x{:=}e \mid a_1; a_2 \end{aligned}$$

A command of form **await** b **then** a is a *conditional atomic action*, and causes the execution of a without interruption when executed in a state satisfying the test expression b; when executed in a state in which b is false the command idles. .

A *sequential program* is just a command containing no **await** and no parallel composition.

Assume given the standard definitions of $\mathtt{free}(e)$ and $\mathtt{free}(b)$, the set of identifiers occurring free in an expression. We will use the standard definitions of $\mathtt{free}(c)$ and $\mathtt{free}(d)$ for the sets of identifiers occurring free in a command or a declaration, and $\mathtt{dec}(d)$, the set of identifiers declared by d.

2.2 Parallel, atomic, and sequential contexts

A *context* is a command which may contain a syntactic "hole" (denoted $[-]$) suitable for insertion of another command. Formally, the set of (parallel) contexts, ranged over by C, is described by the following abstract grammar, in which c_1, c_2 again range over commands:

$$\begin{aligned} C ::= {} & [-] \mid \mathbf{skip} \mid x{:=}e \mid C; c_2 \mid c_1; C \mid \\ & \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C \mid \\ & \mathbf{while}\ b\ \mathbf{do}\ C \mid \mathbf{local}\ d\ \mathbf{in}\ C \mid \\ & \mathbf{await}\ b\ \mathbf{then}\ a \mid \\ & C \| c_2 \mid c_1 \| C \end{aligned}$$

Note that our abstract grammar for contexts only allows at most one hole to appear in any particular context. It would be straightforward to adopt a more general notion of multi-holed context, but the technical details would become more involved and in any case there is no significant loss of generality.

We also introduce the notion of an *atomic context*, i.e. a parallel context whose hole occurs inside the body of an **await** command. We will use A to range over atomic contexts.

A *sequential context* is a limited form of context in which the hole never appears in parallel. We can characterize the set of sequential contexts, ranged over by S, as follows:

$$\begin{aligned} S ::= {} & [-] \mid \mathbf{skip} \mid x{:=}e \mid S; c_2 \mid c_1; S \mid \\ & \mathbf{if}\ b\ \mathbf{then}\ S\ \mathbf{else}\ c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ S \mid \\ & \mathbf{while}\ b\ \mathbf{do}\ S \mid \mathbf{local}\ d\ \mathbf{in}\ S \mid \\ & \mathbf{await}\ b\ \mathbf{then}\ a \mid \ c_1 \| c_2 \end{aligned}$$

The important point in this definition is that $c_1 \| S$ is not a sequential context even when S is sequential, but we do allow "harmless" uses of parallelism inside sequential contexts, as for example in $(c_1 \| c_2); [-]$. The key feature is that sequentiality of S ensures that when we fill the hole with a command we have the guarantee that the command will not be executed concurrently with any of the rest of the code in S.

We write $C[c]$ for the command obtained by inserting c into the hole of C. We use similar notation $A[a]$ for the result of inserting an atomic command a into an atomic context A, and $S[c]$ for the result of inserting a (parallel) command c into a sequential context S.

It is easy to define the set $\mathtt{free}(C)$ of identifiers occurring free in a context C, as usual by structural induction. Similarly we let $\mathtt{free}(S)$ and $\mathtt{free}(A)$ be the sets of identifiers occurring free in sequential context S and in atomic context A.

Contexts may also have a *binding* effect, since the hole in a context may occur inside the scope of one or more (nested) declarations, and free occurrences of identifiers in a code fragment may become bound after insertion into the hole. For example, the context

$$\mathbf{local}\ y = 0\ \mathbf{in}\ ([-] \| y{:=}z + 1)$$

binds y, but not z. On the other hand, the context

$$(\mathbf{local}\ y = 0\ \mathbf{in}\ c_1) \| ([-]; c_2)$$

does not bind any identifier, since the hole does not occur inside a subcommand of **local** form.

To be precise about this possibility we make the following definition. We also make use of analogous notions for sequential contexts and for atomic

contexts, which may be defined in the obvious analogous way. Although we will not prove this here, it follows from the definition that (except for the case of a degenerate context with no hole) for all contexts C and commands c, $\mathtt{free}(C[c]) = \mathtt{free}(C) \cup (\mathtt{free}(c) - \mathtt{bound}(C))$.

Definition 2.1 For a context C, let $\mathtt{bound}(C)$ be the set of identifiers for which there is a binding declaration enclosing the hole in C, defined as follows:

$\mathtt{bound}([-]) = \emptyset$

$\mathtt{bound}(x{:=}e) = \emptyset$

$\mathtt{bound}(C; c_2) = \mathtt{bound}(c_1; C) = \mathtt{bound}(C)$

$\mathtt{bound}(\textbf{if } b \textbf{ then } C \textbf{ else } c_2) = \mathtt{bound}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } C) = \mathtt{bound}(C)$

$\mathtt{bound}(\textbf{while } b \textbf{ do } C) = \mathtt{bound}(C)$

$\mathtt{bound}(\textbf{await } b \textbf{ then } a) = \emptyset$

$\mathtt{bound}(C \| c_2) = \mathtt{bound}(c_1 \| C) = \mathtt{bound}(C)$

$\mathtt{bound}(\textbf{local } d \textbf{ in } C) = \mathtt{bound}(C) \cup \mathtt{dec}(d)$

3 Semantics

3.1 Operational semantics

We assume conventional coarse-grained and fine-grained operational semantics for expressions and commands [2]. In both cases command configurations have the form $\langle c, s\rangle$, where c is a command and s is a state. A state s determines a (finite, partial) function from identifiers to variables, and a "store" mapping variables to their "current" integer values. A transition of form

$$\langle c, s\rangle \to \langle c', s'\rangle$$

represents the effect of c performing an *atomic step* enabled in state s, resulting in a change of state to s', with c' remaining to be executed. A terminal configuration, in which all parallel component commands have terminated, is represented by a (final) state s. In a fine-grained semantics reads and writes to variables are atomic, but assignments and boolean condition evaluations need not be. In a coarse-grained semantics, assignments and boolean expressions are atomic.

A *computation* of a command c is a finite sequence of transitions, ending in a terminal configuration, or an infinite sequence of transitions that is *fair* to all parallel component commands of c. (We may also refer to a *fine-grained computation* or a *coarse-grained computation*, when we need to be precise about which granularity assumption is relevant.) We write $\langle c, s\rangle \to^* \langle c', s'\rangle$ to indicate a finite, possibly empty, sequence of transitions; and $\langle c, s\rangle \to^\omega$ to indicate the existence of a (weakly) fair infinite computation starting from a

given configuration. An *interactive computation* is a finite or infinite sequence of transitions in which the state may be changed between steps, representing the effect of other commands executing in parallel. There is an analogous notion of fairness for interactive computations. A computation is just an interference-free interactive computation, that is, an interactive computation in which no external changes occur.

3.2 State-transformation semantics and sequential equivalence

Definition 3.1 The standard state-transformation semantics for programs, denoted $\mathcal{M}$, is characterized operationally by:

$$\mathcal{M}(c) = \{(s, s') \mid \langle c, s\rangle \rightarrow^* s'\} \ \cup \ \{(s, \perp) \mid \langle c, s\rangle \rightarrow^\omega\}.$$

Definition 3.2 Two programs c_1 and c_2 are *sequentially equivalent*, written $c_1 \equiv_{\mathcal{M}} c_2$, if and only if $\mathcal{M}(c_1) = \mathcal{M}(c_2)$.

As is well known, sequential equivalence is a congruence with respect to the sequential subset of our programming language. In fact, for all parallel programs c_1 and c_2, and all *sequential* contexts S,

$$c_1 \equiv_{\mathcal{M}} c_2 \ \Leftrightarrow \ S[c_1] \equiv_{\mathcal{M}} S[c_2].$$

However, the analogous property fails to hold for *parallel* contexts, because, for example, we have:

$$x{:=}x + 2 \equiv_{\mathcal{M}} x{:=}x + 1;\ x{:=}x + 1$$

but

$$x{:=}x + 2 \| y{:=}x \not\equiv_{\mathcal{M}} (x{:=}x + 1;\ x{:=}x + 1) \| y{:=}x.$$

3.3 Trace semantics

A transition trace of a program c is a finite or infinite sequence of steps, each step being a pair of states that represents the effect of a finite sequence of atomic actions performed by the program. A particular trace $(s_0, s_0')(s_1, s_1') \ldots (s_n, s_n') \ldots$ of c represents a possible fair interactive computation of c in which the inter-step state changes (from s_0' to s_1, and so on) are assumed to be caused by processes executing concurrently to c. Traces are "complete", representing an entire interactive computation, rather than "partial" or "incomplete". A trace is *interference-free* if the state never changes between successive steps along the trace, i.e. in the notation used above when we have $s_i' = s_{i+1}$ for all i. An interference-free trace represents a sequence of snapshots of the state taken during an interference-free fair computation.

Again we obtain both a coarse-grained notion of trace, based on the coarse interpretation of atomicity and the coarse-grained operational semantics, and a fine-grained notion of trace, based on the fine interpretation of atomicity and the fine-grained operational semantics. Both coarse- and fine-grained trace

semantics interpret conditional atomic actions **await** b **then** a as atomic. The coarse-grained trace semantics, which we will denote $\mathcal{T}_{coarse}$, also assumes that assignments and boolean expression evaluations are atomic. The fine-grained semantics, denoted $\mathcal{T}_{fine}$, assumes only that reads and writes to simple variables are atomic. In the rest of this paper, when stating a result which holds for both fine- and coarse-grained semantics, we may use $\mathcal{T}$ to stand for either version of the trace semantic function.

Trace semantics can be defined compositionally, and we note in particular that the traces of $c_1 \| c_2$ are obtained by forming fair merges of a trace of c_1 with a trace of c_2, and the traces of $c_1; c_2$ are obtained by concatenating a trace of c_1 with a trace of c_2, closing up under stuttering and mumbling as required. The traces of **local** $x = e$ **in** c do not change the value of (the "global" version of) x, and are obtained by projection from traces of c in which the value of (the "local" version of) x is never altered between steps.

A parallel program denotes a trace set closed under two natural conditions termed *stuttering* and *mumbling*, which correspond to our use of a step to represent finite sequences of actions: idle or stuttering steps of form (s, s) may be inserted into traces, and whenever two adjacent steps $(s, s')(s', s'')$ share the same intermediate state they can be combined to produce a mumbled trace which instead contains the step (s, s''). The closure properties ensure that trace semantics is *fully abstract* with respect to a notion of behavior which assumes that we can observe the state during execution. As a result trace semantics supports compositional reasoning about safety and liveness properties. Safety properties typically assert that no "bad" state ever occurs when a process is executed, without interference, from an initial state satisfying some pre-condition. A liveness property typically asserts that some "good" state eventually occurs. When two processes have the same trace sets it follows that they satisfy identical sets of safety and liveness properties, in all parallel contexts.

3.4 Fine- and coarse-grained semantic equivalences

When using coarse-grained semantics one can safely use algebraic laws of arithmetic to simplify reasoning about program behavior. For instance, in coarse-grained trace semantics the assignments $x{:=}x + x$ and $x{:=}2 \times x$ are equivalent. This feature can be used to considerable advantage in program analysis. However, coarse granularity is in general an unrealistic assumption since implementations of parallel programming languages do not generally guarantee that assignments are indeed executed indivisibly.

The fine-grained trace semantics is closer in practice to conventional implementations, but less convenient in program analysis. When using fine-grained semantics one cannot assume with impunity that algebraic laws of expression equivalence remain valid. For instance, the assignments $x{:=}x{+}x$ and $x{:=}2{\times}x$ are not equivalent in fine-grained trace semantics; this reflects the fact that

the former reads x twice, so that if x is changed during execution (say from 0 to 1), the value assigned may be $0, 1$ or 2, whereas the latter assignment (under the same circumstances) would assign either 0 or 2.

It should be clear from the above discussion, even without seeing all of the semantic definitions, that despite the connotations suggested by our use of "fine" *vs.* "coarse", these two trace semantic variants induce *incomparable* notions of semantic equivalence. Let us write

$$c_1 \equiv_{coarse} c_2 \;\Leftrightarrow\; \mathcal{T}_{coarse}(c_1) = \mathcal{T}_{coarse}(c_2)$$
$$c_1 \equiv_{fine} c_2 \;\Leftrightarrow\; \mathcal{T}_{fine}(c_1) = \mathcal{T}_{fine}(c_2)$$

For instance, we have already seen a pair of programs which are equivalent in coarse-grained semantics but not in fine-grained:

$$x{:=}x + x \equiv_{coarse} x{:=}2 \times x, \quad x{:=}x + x \not\equiv_{fine} x{:=}2 \times x,$$

so that $c_1 \equiv_{coarse} c_2$ does not always imply $c_1 \equiv_{fine} c_2$.

The converse implication also fails, as shown by the programs $x{:=}x + x$ and

$$\textbf{local } y = 0; z = 0 \textbf{ in } (y{:=}x; z{:=}x; x{:=}y + z)$$

These are equivalent in fine-grained but not in coarse-grained semantics.

Despite the incomparability of fine-grained equivalence and coarse-grained equivalence, for any particular program c the coarse-grained trace set will be a subset of its fine-grained traces:

$$\mathcal{T}_{coarse}(c) \subseteq \mathcal{T}_{fine}(c),$$

so that it is reasonable to refer to the coarse-grained semantics as "simpler".

We also remark that the state-transformation semantics of a parallel program is determined by its trace semantics, in fact by its *interference-free* traces, since $(s, s') \in \mathcal{M}(c)$ if and only if $(s, s') \in \mathcal{T}_{fine}(c)$, and $(s, \perp) \in \mathcal{M}(c)$ if and only if there is an infinite interference-free trace in $\mathcal{T}_{fine}(c)$ beginning from state s. (Here we adopt the usual pun of viewing (s, s') simultaneously as a *pair* belonging to $\mathcal{M}(c)$ and as a *trace* of length 1 belonging to $\mathcal{T}(c)$. Such a trace is trivially interference-free.)

Each trace equivalence is a congruence for the entire parallel language, so that for all contexts C and parallel commands c_1 and c_2 we have:

$$c_1 \equiv_{coarse} c_2 \;\Leftrightarrow\; C[c_1] \equiv_{coarse} C[c_2]$$
$$c_1 \equiv_{fine} c_2 \;\Leftrightarrow\; C[c_1] \equiv_{fine} C[c_2]$$

Moreover, $c_1 \equiv_{fine} c_2$ implies $c_1 \equiv_{\mathcal{M}} c_2$, but the converse implication is not generally valid.

4 Reads and writes of a command

To prepare the ground for our transfer principles, we first need to define for each parallel program c the *multiset* $\texttt{reads}(c)$ of identifier occurrences which

appear free in non-atomic sub-expressions of c. It is vital here, as suggested by the terminology, to keep track of how many references the program makes, to each identifier. We need only be concerned with non-atomic subphrases, since these are the only ones whose execution may be affected by concurrent activity.

We also need to refer to the analogous notions for expressions and for declarations; since we have not provided a full grammar for expressions we will give details only for a few key cases, which suffice for understanding all of the examples which follow and which convey the general ideas.

For precise mathematical purposes, we may think of a multiset as a set of identifiers equipped with a non-negative multiplicity count. In the empty multiset every identifier has multiplicity 0. When M_1 and M_2 are multisets, we let $M_1 \cup_+ M_2$ be the multiset union in which multiplicities are added, and $M_1 \cup_{max} M_2$ be the multiset union in which multiplicities are combined using max. That is, an identifier x which occurs n_1 times in M_1 and n_2 times in M_2 will occur $n_1 + n_2$ times in $M_1 \cup_+ M_2$ and $max(n_1, n_2)$ times in $M_1 \cup_{max} M_2$.

We write $\{\!|x|\!\}$ for the singleton multiset containing a single occurrence of x. We also write $\{\!|\ |\!\}$ for the empty multiset. The cardinality of a multiset M is denoted $|M|$.

Each version of union is symmetric and associative:

$$M_1 \cup_+ M_2 = M_2 \cup_+ M_1$$

$$M_1 \cup_+ (M_2 \cup_+ M_3) = (M_1 \cup_+ M_2) \cup_+ M_3$$

$$M_1 \cup_{max} M_2 = M_2 \cup_{max} M_1$$

$$M_1 \cup_{max} (M_2 \cup_{max} M_3) = (M_1 \cup_{max} M_2) \cup_{max} M_3$$

In addition, $\cup_{max}$ is idempotent:

$$M \cup_{max} M = M$$

Obviously $\cup_+$ is not idempotent.

The empty multiset is a unit for both forms of union, since

$$M \cup_+ \{\!|\ |\!\} = M \cup_{max} \{\!|\ |\!\} = M.$$

Given a multiset M and a set X of identifiers, we define $M - X$ to be the multiset obtained from M by removing all occurrences of identifiers in X, and we let $M \cap X$ be the multiset consisting of those members of M which are also in X, with the same multiplicities as they have in M.

We are now ready to define the read multiset of an expression. Again we include only a few representative cases. Note that we will use the additive form of multiset union for an expression of form $e_1 + e_2$ (and also, in general, for expressions built with binary operators), because we want to count the number of times an identifier needs to be read during the evaluation of an expression.

Definition 4.1 The multiset $\mathtt{reads}(e)$ of free identifier occurrences in an expression e is given inductively by:

$$\mathtt{reads}(n) = \{\!\!\{\ \}\!\!\}$$

$$\mathtt{reads}(x) = \{\!\!\{x\}\!\!\}$$

$$\mathtt{reads}(e_1 + e_2) = \mathtt{reads}(e_1) \cup_+ \mathtt{reads}(e_2)$$

A similar definition can be given for boolean expressions.

Definition 4.2 The multiset $\mathtt{reads}(d)$ of free identifier occurrences in a declaration d is given inductively by:

$$\mathtt{reads}(x = e) = \mathtt{reads}(e)$$

$$\mathtt{reads}(d_1; d_2) = \mathtt{reads}(d_1) \cup_{max} (\mathtt{reads}(d_2) - \mathtt{dec}(d_1))$$

Here we combine using maximum since d may require the separate evaluation of several sub-expressions.

Now we can provide the definition for commands:

Definition 4.3 The multiset $\mathtt{reads}(c)$ of free identifier occurrences read by command c is given inductively by:

$$\mathtt{reads}(\textbf{skip}) = \{\!\!\{\ \}\!\!\}$$

$$\mathtt{reads}(x{:=}e) = \mathtt{reads}(e)$$

$$\mathtt{reads}(c_1; c_2) = \mathtt{reads}(c_1 \| c_2) = \mathtt{reads}(c_1) \cup_{max} \mathtt{reads}(c_2)$$

$$\mathtt{reads}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = \mathtt{reads}(b) \cup_{max} (\mathtt{reads}(c_1) \cup_{max} \mathtt{reads}(c_2))$$

$$\mathtt{reads}(\textbf{while } b \textbf{ do } c) = \mathtt{reads}(b) \cup_{max} \mathtt{reads}(c)$$

$$\mathtt{reads}(\textbf{await } b \textbf{ then } a) = \{\!\!\{\ \}\!\!\}$$

$$\mathtt{reads}(\textbf{local } d \textbf{ in } c) = \mathtt{reads}(d) \cup_{max} (\mathtt{reads}(c) - \mathtt{dec}(d))$$

Again we use the maximum-forming union operation to combine the counts from all sub-expression evaluations. Notice that we regard an **await** command as having *no* reads, because it will be executed atomically and its effect will therefore be immune from concurrent interference.

Next we define the *set* $\mathtt{writes}(c)$ of identifier occurrences which occur free in c as targets of assignments. It will turn out that we do not need an accurate count of how many times an individual identifier is assigned, just the knowledge of whether or not each identifier is assigned to: even once is bad enough. Our definition ensures that $x \in \mathtt{writes}(c)$ if and only if there is at least one free occurrence of x in c in a sub-command of the form $x{:=}e$.

Definition 4.4 The set $\mathtt{writes}(c)$ of identifiers occurring freely as targets of

assignment in c is given by:

$$\begin{aligned}
&\mathtt{writes}(\mathbf{skip}) = \emptyset \\
&\mathtt{writes}(x{:=}e) = \{x\} \\
&\mathtt{writes}(c_1; c_2) = \mathtt{writes}(c_1 \| c_2) = \mathtt{writes}(c_1) \cup \mathtt{writes}(c_2) \\
&\mathtt{writes}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \mathtt{writes}(c_1) \cup \mathtt{writes}(c_2) \\
&\mathtt{writes}(\mathbf{while}\ b\ \mathbf{do}\ c) = \mathtt{writes}(c) \\
&\mathtt{writes}(\mathbf{await}\ b\ \mathbf{then}\ a) = \mathtt{writes}(a) \\
&\mathtt{writes}(\mathbf{local}\ d\ \mathbf{in}\ c) = \mathtt{writes}(c) - \mathtt{dec}(d)
\end{aligned}$$

5 Concurrent reads and writes of a context

Next we define, for each parallel context C, the pair $\mathtt{crw}(C) = (R, W)$, where R is the set of identifiers which occur free in evaluation contexts concurrent to a hole of C, and W is the set of identifiers occurring free in assigning contexts concurrent to a hole. As usual the definition is inductive. It suffices to work with sets here rather than multisets, since what matters for our present purposes is whether or not the context may change an identifier's value concurrently while whatever command occupies the hole is running, not how many times the context may do this; even once is bad enough.

Definition 5.1 The concurrent-reads-and-writes of a context C are given by:

$$\begin{aligned}
&\mathtt{crw}([-]) = \mathtt{crw}(\mathbf{skip}) = \mathtt{crw}(x{:=}e) = (\emptyset, \emptyset) \\
&\mathtt{crw}(C; c_2) = \mathtt{crw}(c_1; C) = \mathtt{crw}(C) \\
&\mathtt{crw}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C) = \mathtt{crw}(\mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2) = \mathtt{crw}(C) \\
&\mathtt{crw}(\mathbf{while}\ b\ \mathbf{do}\ C) = \mathtt{crw}(C) \\
&\mathtt{crw}(\mathbf{await}\ b\ \mathbf{then}\ a) = (\emptyset, \emptyset) \\
&\mathtt{crw}(\mathbf{local}\ d\ \mathbf{in}\ C) = \mathtt{crw}(C) \\
&\mathtt{crw}(c \| C) = \mathtt{crw}(C \| c) = (R \cup \mathtt{reads}(c), W \cup \mathtt{writes}(c)), \\
&\qquad \text{where } (R, W) = \mathtt{crw}(C)
\end{aligned}$$

Note that the clause for **local** d **in** C may include in the concurrent reads and writes some of the identifiers declared by d; when code is inserted into the context occurrences of these identifiers become bound, but we still need to know if and how the code uses these identifiers concurrently.

6 Transfer principles

We now state some fundamental properties of trace semantics, which formalize the sense in which the behavior of a parallel program depends only on the values of its free identifiers. We say that two states s and s' *agree* on a set X of identifiers if they map each identifier in this set to (variables which have) the same integer value. These properties are analogues in the parallel setting of "Agreement" properties familiar from the sequential setting. Their proofs are straightforward structural inductions based on the trace semantic definitions.

Theorem 6.1 *Let α be a trace of c and (s, s') be a step of α. Then s agrees with s' on all identifiers not in* $\mathtt{writes}(c)$. □

Theorem 6.2 *Let $(s_0, s_0')(s_1, s_1') \ldots (s_n, s_n') \ldots$ be a trace of c. Then for every sequence of states $t_0, t_1, \ldots, t_n, \ldots$ such that for all $i \geq 0$, t_i agrees with s_i on $X \supseteq \mathtt{reads}(c)$, there is a trace*

$$(t_0, t_0')(t_1, t_1') \ldots (t_n, t_n') \ldots$$

of c such that for all $i \geq 0$, t_i' agrees with t_i on $X \cup \mathtt{writes}(c)$. □

Having set up the relevant background definitions and this key agreement lemma we can now present the transfer principles to which we have been leading.

6.1 A transfer principle for atomic contexts

The first one is almost too obvious to include: it suffices to use sequential reasoning about any code used in a syntactically atomic context. This holds in both coarse- and fine-grained semantics, so we will use $\equiv_{\mathcal{T}}$ to stand for either form of trace equivalence.

Theorem 6.3 *If A is an atomic context and $a_1 \equiv_{\mathcal{M}} a_2$, then $A[a_1] \equiv_{\mathcal{T}} A[a_2]$.*

Proof. The traces of **await** b **then** a depend only on the "atomic" traces of a, i.e. on the traces of a which represent uninterrupted complete executions; and (s, s') is an atomic trace of a iff $(s, s') \in \mathcal{M}(a)$. □

6.2 A sequential transfer principle

The next transfer principle identifies conditions under which sequential equivalence of code fragments can safely be relied upon to establish trace equivalence of parallel programs.

Theorem 6.4 *If $\mathtt{free}(c_1) \cup \mathtt{free}(c_2) \subseteq \mathtt{bound}(C)$, and $(R, W) = \mathtt{crw}(C)$, and*

$$|\mathtt{reads}(c_i) \cap W| + |\mathtt{writes}(c_i) \cap R| = 0, \ i = 1, 2$$

then

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow C[c_1] \equiv_{\mathcal{T}} C[c_2].$$ □

It is worth noting that the provisos built into this theorem are essential. If we omit the local declaration around the context the result becomes invalid, since the assumption that c_1 and c_2 are sequentially equivalent is not strong enough to imply that c_1 and c_2 are trace equivalent. And if we try to use the code fragments in a context with which it interacts non-trivially again the result fails: when c_1 and c_2 are sequentially equivalent it does not follow that **local** d **in** $(c\|c_1)$ and **local** d **in** $(c\|c_2)$ are trace equivalent for all c, even if d declares all of the free identifiers of c_1 and c_2. A specific counterexample is obtained by considering the commands

$$c_1: \quad x{:=}x+1;\ x{:=}x+1$$

$$c_2: \quad x{:=}x$$

We have $\mathtt{reads}(c_i) = \{\!|x|\!\}$, $\mathtt{writes}(c_i) = \{x\}$. Let C be the context

$$\textbf{local } x = 0 \textbf{ in } (([-]\|x{:=}2);\ y{:=}x).$$

Then $\mathtt{bound}(C) = \{x\}$ and $\mathtt{crw}(C) = (\emptyset, \{x\})$. Using the notation of the theorem, we have

$$|\mathtt{reads}(c_i) \cap W| = 1, \quad |\mathtt{writes}(c_i) \cap R| = 0$$

so that the assumption is violated. And it is easy to see that $c_1 \equiv_{\mathcal{M}} c_2$, but

$$C[c_1] \equiv_{\mathcal{T}} y{:=}0 \textbf{ or } y{:=}1 \textbf{ or } y{:=}2$$

$$C[c_2] \equiv_{\mathcal{T}} y{:=}0 \textbf{ or } y{:=}2$$

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.[3]

Another example shows that the other half of the assumption cannot be relaxed. Consider

$$c_1: \quad x{:=}1;\ \textbf{while true do skip}$$

$$c_2: \quad x{:=}2;\ \textbf{while true do skip}$$

Let C be the context

$$\textbf{local } x = 0 \textbf{ in } ([-]\|y{:=}x).$$

Then $\mathtt{bound}(C) = \{x\}$, $\mathtt{free}(c_i) = \mathtt{writes}(c_i) = \{x\}$, and $\mathtt{reads}(c_i) = \{\!|\,|\!\}$. Moreover $c_1 \equiv_{\mathcal{M}} c_2$, since $\mathcal{M}(c_i) = \{(s, \perp) \mid s \in \mathbf{S}\}$ $(i = 1, 2)$. We have

$$|\mathtt{reads}(c_i) \cap W| = 0, \quad |\mathtt{writes}(c_i) \cap R| = 1$$

so that the assumption is violated again. And we also have

$$C[c_1] \equiv_{\mathcal{T}} (y{:=}0 \textbf{ or } y{:=}1); \textbf{while true do skip}$$

$$C[c_2] \equiv_{\mathcal{T}} (y{:=}0 \textbf{ or } y{:=}2); \textbf{while true do skip}$$

[3] Although our programming language did not include a non-deterministic choice operator c_1 **or** c_2 it is convenient to use it as here, to specify a command that behaves like c_1 or like c_2; in terms of trace sets we have $\mathcal{T}(c_1 \textbf{ or } c_2) = \mathcal{T}(c_1) \cup \mathcal{T}(c_2)$, a similar equation holiding in coarse- and in fine-grained versions.

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.

The above theorem is always applicable in the special case where the context is sequential. We therefore state the following:

Corollary 6.5 *If S is a sequential context, and* $\mathtt{free}(c_1) \cup \mathtt{free}(c_2) \subseteq \mathtt{bound}(S)$, *then*

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow S[c_1] \equiv_{\mathcal{T}} S[c_2].$$

Proof. When S is sequential we can show, by induction on the structure of S, that $\mathtt{crw}(S) = (\emptyset, \emptyset)$. □

To illustrate the benefits of these results, note that many simple laws of sequential equivalence are well known, and tend to be taken for granted when reasoning about sequential programs. Note in particular the following instances of de Bakker's laws of (sequential) equivalence [6], which can be used to simplify sequences of assignments:

$$\begin{aligned} x{:=}x &\equiv_{\mathcal{M}} \mathbf{skip} \\ x{:=}e_1; x{:=}e_2 &\equiv_{\mathcal{M}} x{:=}[e_1/x]e_2 \\ x_1{:=}e_1; x_2{:=}e_2 &\equiv_{\mathcal{M}} x_2{:=}e_2; x_1{:=}e_1, \\ &\qquad \text{if } x_1 \notin \mathtt{free}(e_2) \;\&\; x_2 \notin \mathtt{free}(e_1) \;\&\; x_1 \neq x_2 \end{aligned}$$

These laws fail to hold in the parallel setting, and become unsound when $\equiv_{\mathcal{M}}$ is replaced by $\equiv_{\mathit{fine}}$ or $\equiv_{\mathit{coarse}}$. Our result shows the extent to which such laws may safely be used when reasoning about the safety and liveness properties of *parallel* programs, pointing out sufficient conditions under which sequential analysis of key code fragments is enough to ensure correctness of a parallel program.

6.3 A coarse- to fine-grained transfer principle

Finally, we now consider what requirements must be satisfied in order to safely employ coarse-grained trace-based reasoning in establishing fine-grained equivalences. This may be beneficial, as remarked earlier, since for a given code fragment the coarse-grained trace set forms a (usually proper) subset of the fine-grained trace set and may therefore permit a streamlined analysis. This is especially important for code which may be executed concurrently, since it may help minimize the combinatorial analysis. Indeed, Andrews [1] supplies a series of examples of protocols in which a "fine-grained" solution to a parallel programming problem (such as mutual exclusion) is derived by syntactic transformation from a "coarse-grained" solution whose correctness is viewed as easier to establish. Common to all of these examples is the desire to appeal to coarse-grained reasoning when trying to establish correctness in the fine-grained setting.

We begin with a so-called "at-most-once" property that Andrews uses informally to facilitate the analysis and development of a collection of mutual

exclusion protocol designs. The relevant definitions from Andrews, adapted to our setting, are as follows:

- An expression b (or e) has the at-most-once property if it refers to at most one identifier that might be changed by another process while the expression is being evaluated, and it refers to this identifier at most once.
- An assignment $x{:=}e$ has the at-most-once property if either e has the at-most-once property and x is not read by another process, or if e does not refer to any identifier that may be changed by another process.
- A command c has the at-most-once property if every assignment and boolean test occurring non-atomically in c has the at-most-once property.

An occurrence is atomic if it is inside a subcommand of form **await** b **then** a.

Andrews's methodology is based on the idea that if a command has the at-most-once property then it suffices to assume coarse-grained execution when reasoning about its behavior, since there will be no discernible difference with fine-grained execution. However, the above characterization of an at-most-once property is only informal and slightly imprecise, in particular in relying on implicit analysis of the context in which code is to be executed. We will couch our transfer principle in slightly more specific but general terms based on a precise reformulation of this property, referring to the `crw` definition from above.

Theorem 6.6 *If* $\mathtt{free}(c_1) \cup \mathtt{free}(c_2) \subseteq \mathtt{bound}(C)$, *and* $(R, W) = \mathtt{crw}(C)$, *and*

$$\begin{aligned} \textit{either} \quad & |\mathtt{reads}(c_i) \cap W| = 0 \\ \textit{or} \quad & |\mathtt{reads}(c_i) \cap W| = 1 \ \&\ |\mathtt{writes}(c_i) \cap (R \cup W)| = 0, \ i = 1, 2 \end{aligned}$$

then

$$c_1 \equiv_{\mathit{coarse}} c_2 \Rightarrow C[c_1] \equiv_{\mathit{fine}} C[c_2]. \qquad \square$$

Thus our formal version of the at-most-once property can be read as requiring that the command reads at most one occurrence of an identifier written concurrently by the context, and if it reads one then none of its writes affect any identifier which is either read or written concurrently by the context. Our insistence in the above theorem that the code being analyzed (c_1 and c_2) only affects local variables, i.e. identifiers which become bound when the code is inserted into the context, is reflected in Andrews's setting by an assumption that all processes have local registers.

Again we show that the built-in provisos imposing locality and the at-most-once property cannot be dropped.

Firstly, every program has the at-most-once property, trivially, for the context $[-]$. But the assumption that $c_1 \equiv_{\mathit{coarse}} c_2$ is insufficient to ensure that $c_1 \equiv_{\mathit{fine}} c_2$. Thus the result becomes invalid if we omit the localization around the context.

To illustrate the need for the at-most-once assumption, let the programs c_1 and c_2 be $y{:=}x + x$ and $y{:=}2 \times x$. These programs are clearly coarsely equivalent. Let C be the context

$$\textbf{local } x = 0; y = 0 \textbf{ in } (([-]\|x{:=}1);\ z{:=}y).$$

Of course c_1 refers twice to x, which is assigned to by the context concurrently; c_1 does not satisfy the at-most-once property for C. Moreover we can see that

$$\textbf{local } x = 0; y = 0 \textbf{ in } ((y{:=}x + x\|x{:=}1); z{:=}y) \equiv_{\mathit{fine}} z{:=}0 \textbf{ or } z{:=}1 \textbf{ or } z{:=}2$$

$$\textbf{local } x = 0; y = 0 \textbf{ in } ((y{:=}2 \times x\|x{:=}1); z{:=}y) \equiv_{\mathit{fine}} z{:=}0 \textbf{ or } z{:=}2$$

so that $C[c_1] \not\equiv_{\mathit{fine}} C[c_2]$.

Also note that the other way for the assumption to fail is when c_1 (say) both reads and writes to a concurrently accessed identifier. For instance, let c_1 be $x{:=}x$ and c_2 be **await true then** $x{:=}x$. Let C be the context

$$\textbf{local } x = 0 \textbf{ in } (([-]\|x{:=}1); y{:=}x)$$

Then we have $|\mathtt{reads}(c_i) \cap W| = 1$ and $|\mathtt{writes}(c_i) \cap (R \cup W)| > 0$. And $c_1 \equiv_{\mathit{coarse}} c_2$. But $C[c_1] \equiv_{\mathit{fine}} y{:=}0$ **or** $y{:=}1$, and $C[c_2] \equiv_{\mathit{fine}} y{:=}1$.

It is also worth remarking that the above principle cannot be strengthened by replacing the assumption that c_1 and c_2 are *coarsely equivalent* with the weaker assumption that c_1 and c_2 are *sequentially equivalent*. For example, let c_1 and c_2 be

$$y{:=}1;\ \textbf{while true do skip}$$

and

$$y{:=}2;\ \textbf{while true do skip}.$$

Let C be the context **local** $y = 0$ **in** $([-]\|z{:=}y)$. Then we have $\mathtt{reads}(c_i) = \emptyset$, $\mathtt{writes}(c_i) = \{y\}$, $\mathtt{crw}(C) = (\{y\}, \{z\})$, $\mathtt{bound}(C) = \{y\}$. Moreover, $c_1 \equiv_{\mathcal{M}} c_2$ holds, since $\mathcal{M}(c_i) = \{(s, \bot) \mid s \in \mathbf{S}\}$, $i = 1, 2$. However,

$$C[c_1] \equiv_{\mathit{fine}} (z{:=}0 \textbf{ or } z{:=}1)$$

and

$$C[c_2] \equiv_{\mathit{fine}} (z{:=}0 \textbf{ or } z{:=}2),$$

so that $C[c_1] \not\equiv_{\mathit{fine}} C[c_2]$.

The coarse- to fine-grained transfer theorem given above generalizes some more *ad hoc* arguments based on occurrence-counting in Andrews's book, resulting in a single general principle in which the crucial underlying provisos are made explicit. To make the connection with Andrews's examples more precise, note the following special cases of our theorem, which appear in paraphrase in Andrews:

- If b refers at most once to identifiers written concurrently (by the context), then **await** b **then skip** can be replaced by **while** $\neg b$ **do skip** (throughout

the program). This rule may be used to justify replacement of a conditional atomic action with a (non-atomic) busy-wait loop.

- If $x{:=}e$ has the at-most-once property (for the context) then the assignment $x{:=}e$ can be replaced by its atomic version **await true then** $x{:=}e$ (throughout the program). This rule may be used to simplify reasoning about the potential for interaction between processes.

7 Conclusions and future work

We have identified conditions under which it is safe to employ "sequential" reasoning about code fragments while trying to establish "parallel" correctness properties such as safety and liveness. We have also identified conditions governing the safe use of coarse-grained reasoning in proving fine-grained properties.

These transfer principles can be seen as supplying a semantic foundation for some of the ideas behind Andrews's protocol analysis, and a potential basis for further generalization and the systematic development of techniques to permit easier design and analysis of parallel programs. We plan to extend our ideas and results to cover a wider variety of examples, including some of the protocols discussed by Dingel. It would also be interesting to explore the relationship between our approach and Dingels' notion of context-sensitive approximation.

These results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system. It would be interesting to investigate the possible utility of transfer principles in improving the efficiency of model-checking for finite-state concurrent systems.

8 Acknowledgements

The anonymous referees made a number of helpful suggestions. The author would also like to thank his former Ph.D. student, Jürgen Dingel, whose thesis research provides a stimulus for the work reported here.

References

[1] Andrews, G., *Concurrent Programming: Principles and Practice.* Benjamin/ Cummings (1991).

[2] Brookes, S., *Full abstraction for a shared-variable parallel language.* Information and Computation **127**(2), 145–163 (June 1996).

[3] Brookes, S., *The essence of Parallel Algol.* Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 164–173 (1996). To appear in *Information and Computation.*

[4] Brookes, S., *Idealized CSP: Combining Procedures with Communicating Processes.* Mathematical Foundations of Programming Semantics, 13^{th} Conference, March 1997. Electronic Notes in Theoretical Computer Science 6, Elsevier Science (1997). URL: `http://www.elsevier.nl/ locate/ entcs/volume6.html`.

[5] Brookes, S., *Communicating Parallel Processes.* In: *Millenium Perspectives in Computer Science*, Proceedings of the Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare, edited by Jim Davies, Bill Roscoe, and Jim Woodcock, Palgrave Publishers (2000).

[6] de Bakker, J., *Axiom systems for simple assignment statements.* In *Symposium on Semantics of Algorithmic Languages*, edited by E. Engeler. Springer-Verlag LNCS vol. 181, 1–22 (1971).

[7] Dingel, J., *Systematic parallel programming.* Ph. D. thesis, Carnegie Mellon University, Department of Computer Science (May 2000).

[8] Jones, C. B., *Tentative steps towards a development method for interfering programs*, ACM Transactions on Programming Languages and Systems, 5(4):576–619 (1983).

[9] Park, D., *On the semantics of fair parallelism.* In *Abstract Software Specifications*, edited by D. Bjørner, Springer-Verlag LNCS vol. 86, 504–526 (1979).

[10] Park, D., *Concurrency and automata on infinite sequences.* Springer LNCS vol. 104 (1981).

MFPS 17 Preliminary Version

Time Stamps for Fixed-Point Approximation

Daniel Damian

BRICS[1]

Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: damian@brics.dk

Abstract

Time stamps were introduced in Shivers's PhD thesis for approximating the result of a control-flow analysis. We show them to be suitable for computing program analyses where the space of results (e.g., control-flow graphs) is large. We formalize time-stamping as a top-down, fixed-point approximation algorithm which maintains a single copy of intermediate results. We then prove the correctness of this algorithm.

1 Introduction

1.1 Abstract interpretation and fixed-point computation

Abstract interpretation [6,10] is a framework for systematic derivation of program analyses. In this framework, the standard semantics of a program is approximated by an abstract semantics. The abstract semantics simulates the standard semantics and is used to extract properties of the actual run-time behavior of the program.

Abstract interpretation often yields program analyses specified by a set of recursive equations. Formally, the analysis is defined as the least fixed point of a functional over a specific lattice. Analyzing a program then amounts to computing such a least fixed point. The design and analysis of algorithms for computing least fixed points has thus become a classic research topic.

This article presents a top-down algorithm that computes an approximate solution for a specific class of program analyses. This class includes analyses of programs with dynamic control-flow, namely programs whose control-flow is determined by the run-time values of program variables. Such programs are common, for instance, in higher-order and object-oriented languages.

[1] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The common problem of analyzing programs with dynamic control flow is to compute a static approximation of the dynamic control flow graph. The flow information is usually represented as a table mapping each program point to the set of points that form possible outgoing edges from that point. The analysis may compute flow information either as a separate phase, or as an integral component of the abstract interpretation. In any case, flow information is itself computed as a least fixed point of a functional.

An algorithm for computing a solution is met with a difficult practical constraint: due to the potential size of the control-flow graph embedded in the result of the analysis, one cannot afford to maintain multiple intermediate results. The time-stamps based algorithm considered here only needs to maintain a single intermediate analysis result throughout the computation.

1.2 The time-stamping technique

The time-stamping technique has been previously introduced in Shivers's PhD thesis [19] on control-flow analysis for Scheme, based on ideas from Hudak and Young's "memoized pending analysis" [20]. Using time stamps Shivers implements a top-down algorithm which computes an approximation of the original analysis but does not maintain multiple intermediate results. The algorithm ensures termination by relying on the required monotonicity of the abstract semantics and by using time stamps on abstract environments. It obtains an approximation by using increasingly approximate environments on the sequential analysis of program paths.

To our knowledge, Shivers's thesis contains the only description of the time-stamping technique. The thesis provides a formal account of some of the transformations performed on the abstract control-flow semantics in order to obtain an efficient implementation (as, for instance, the "aggressive cutoff" approach). The introduction of time stamps, however, remains only informally described. In particular, his account of the time-stamps algorithm [19, Chapter 6] relies on the property that the recursion sets computed by the modified algorithm are included in the recursion sets computed by the basic algorithm. Such property relies on the monotonicity of the original semantics, and the relationship with the algorithm modified to use a single-threaded environment remains unclear.

Our work:

We formalize the time-stamps based approximation algorithm as a generic fixed-point approximation algorithm, and we prove its correctness.

1.3 Overview

The rest of the article is organized as follows: In Section 2 we describe the time-stamps based approximation algorithm. In Section 2.1 we define the class of recursive equations on which the algorithm is applicable. In Section 2.2 we

describe the intuition behind the time stamps. We proceed in Section 3 to formalize the time-stamps based algorithm (Section 3.1) and prove its correctness (Section 3.2). In Section 3.3 we estimate the complexity of the algorithm. In Section 4 we show how to extend the algorithm to a wider class of analyses. In Section 5 we review related work and in Section 6 we conclude.

2 The time-stamps based approximation algorithm

2.1 A class of recursive equations

We consider a class of recursive equations which model abstract interpretations that gather information about a program by simulating its execution. We can abstract the program as a set of nodes in a graph. We consider that a given program p induces a finite set of program points Lab. Transitions from a program point to another are described as directed edges in the graph. The abstract semantics collects information as an element $\widehat{\rho}$ of a complete lattice A (we assume that A has finite height). Typically, such analysis information is in the form of a cache which collects information of interest on program points and variables.

In our setting, at a program point $\ell \in Lab$, with intermediate analysis information $\widehat{\rho}$, the result of the analysis is computed from some local analysis information and from the union of results obtained by following all possible outgoing paths. For instance, at a branching statement with an unknown boolean condition, we analyze both branches and merge the results. In higher-order languages, at a function call $(e_0\ e_1)$, we analyze as many outgoing paths as the number of functions the expression e_0 can evaluate to.

The choice of a specific outgoing path determines a specific update of the analysis information. For instance, by choosing one of the functions that may be called at an application point, one updates the information associated to the formal parameter with the information associated to the actual parameter.

We consider therefore that local analysis information is defined as a monotone function $B : (Lab \times A) \to A$, and that the analysis information associated with an edge is given by a monotone function $V : (Lab \times Lab \times A) \to A$. Such functions V correspond, for instance, to Sagiv, Reps and Horowitz's environment transformers [17], but they can also model monotone frameworks [12,13]. In any case, we are modeling a form of collecting analyses [6,18], as we merge the execution information to the already computed information when following an edge.

To model dynamic control flow, we assume that, at a specific node ℓ and in the presence of already computed analysis information $\widehat{\rho}$, the set of possible outgoing edges is described by a monotone function $R : (Lab \times A) \to \mathcal{P}(Lab)$: edges are formed from the current node ℓ and the elements of $R(\ell, \widehat{\rho})$. A generic analysis function $F : (Lab \times A) \to A$ may therefore be defined by the

following recursive equation:

$$F(\ell, \widehat{\rho}) = B(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell, \widehat{\rho})} F(\ell', \widehat{\rho} \sqcup V(\ell, \ell', \widehat{\rho})), \qquad (*)$$

If the functions B, R and V are monotone on $\widehat{\rho}$ (Lab is essentially a flat domain), it can be easily shown that Equation $(*)$ has solutions. Given the starting point of the program ℓ_0 and some initial (possibly empty) analysis information $\widehat{\rho}_0$, we are interested in computing a value $F(\ell_0, \widehat{\rho}_0)$, where F is the least solution of Equation $(*)$.

It is usually more expensive to compute the entire function F as the least solution of Equation $(*)$. Naturally, we want to implement a program that computes the value of F on a specific pair $(\ell, \widehat{\rho})$. In order to compute the value $F(\ell, \widehat{\rho})$, one needs to control termination (repeating sequences of pairs $(\ell, \widehat{\rho})$ might appear) and one also needs to save intermediate copies of the current analysis information $\widehat{\rho}$ when the current node ℓ has multiple outgoing edges.

Memoization may be an easy solution for controlling termination. When the space of analysis results is large, however, the cost of maintaining the memoization table, coupled with the cost of saving intermediate results, leads to a prohibitively expensive implementation. We can use Shivers's time-stamping technique [19] to solve these two problems, as long as we are satisfied with an approximation of $F(\ell_0, \widehat{\rho}_0)$.

2.2 The intuition behind time stamps

We present a pseudo-code formulation of the algorithm in order to informally describe the time-stamping technique. We will properly formalize the algorithm and prove its correctness in Section 3. We assume that we can compute the functions B, R and V which define a instance of the analysis.

The pseudo-code of the time-stamps based approximation algorithm is given in Figure 1. The time-stamps based algorithm uses a time counter t (initialized with 0) and a table τ which associates to each program point ℓ a time stamp $\tau[\ell]$, initialized with 0. We compute the result of the analysis into a global variable $\widehat{\rho}$, which is initialized with $\widehat{\rho}_0$. Otherwise said, we lift the $\widehat{\rho}$ parameter out of the F function.

The time counter t and the time-stamps table τ (modeled as an array of integers) are also global variables. The function U updates the global analysis with fresh information: if new results are computed, the time counter is incremented before they are added in the global analysis. The function F implements the time-stamps based approximation. To compute the value of $F(\ell)$, we first compute the local information $B(\ell_0, \widehat{\rho})$ and add the result into the global analysis. We then compute the set of outgoing nodes $R(\ell_0, \widehat{\rho})$. For each outgoing node $\ell' \in R(\ell_0, \widehat{\rho})$, *sequentially*, we compute the execution information $V(\ell, \ell', \widehat{\rho})$ along the edge (ℓ, ℓ'), we add its result to $\widehat{\rho}$ and we then call $F(\ell')$. Because all the calls to F on the second or later branches are made

```
global ρ̂ : A, t : N, τ : N array
fun U(ρ̂_1) = if ρ̂_1 ⋢ ρ̂ then t := t + 1; ρ̂ := ρ̂ ⊔ ρ̂_1
fun F(ℓ) = if τ[ℓ] ≠ t then
                τ[ℓ] := t;
                U(B(ℓ, ρ̂));
                foreach ℓ' in R(ℓ, ρ̂)
                    U(V(ℓ, ℓ', ρ̂)); F(ℓ')
```

Fig. 1. Time-stamps based approximation algorithm

with a possibly larger $\widehat{\rho}$, an approximation may occur.

Each time $\widehat{\rho}$ is increased by addition of new information, we increment the time counter. Each time we call F on a program point ℓ, we record the current value of the time counter in the time-stamps table at ℓ's slot, i.e., $\tau[\ell] := t$. We use the time-stamps table to control the termination. If the function F is called on a point ℓ such that $\tau[\ell] = t$, then there has already been a call to F on ℓ, and the environment has not been updated since. Therefore, no fresh information is going to be added to the environment by this call, and we can simply return without performing any computation.

Such correctness argument is only informal, though. In his thesis, Shivers [19] makes a detailed description of the time-stamps technique in the context of a control-flow analysis for Scheme. He proves that memoization (the so-called "aggressive cutoff" method) preserves the results of the analysis. The introduction of time-stamps and the approximation obtained by collecting results in a global variable remain only informally justified. In the next section we provide a formal description of the time-stamps based approximation algorithm and we prove its correctness.

3 A formalization of the time-stamps based algorithm

3.1 State-passing recursive equations

We formalize the algorithm and the time-stamping technique as a new set of recursive equations. The equations describe precisely the computational steps of the algorithm. They are designed such that their solution can be immediately related with the semantics of an implementation of the algorithm from Figure 1 in a standard programming language. In the same time, they define an approximate solution of Equation $(*)$ on the page before. We prove that the solution of the new equations is indeed an approximation of the original form.

The equations are modeling a state-passing computation. The global state of the computation contains the analysis information $\widehat{\rho}$, the time-stamps table τ and the time counter t. The time-stamps table is modeled by a function

$$
\begin{aligned}
F'(\ell,(\widehat{\rho},\tau,t)) = {} & \textbf{if } \tau(\ell)=t \textbf{ then } (\widehat{\rho},\tau,t) \\
& \textbf{else let} \\
& \quad \{\ell_1,\ldots,\ell_n\} = R(\ell,\widehat{\rho}) \\
& \quad (\widehat{\rho}_0,\tau_0,t_0) = U(B(\ell,\widehat{\rho}),(\widehat{\rho},\tau[\ell\mapsto t],t)) \\
& \quad (\widehat{\rho}_1,\tau_1,t_1) = F'(\ell_1,U(V(\ell,\ell_1,\widehat{\rho}_0),(\widehat{\rho}_0,\tau_0,t_0))) \\
& \quad \vdots \\
& \quad (\widehat{\rho}_n,\tau_n,t_n) = F'(\ell_n,U(V(\ell,\ell_n,\widehat{\rho}_{n-1}),(\widehat{\rho}_{n-1},\tau_{n-1},t_{n-1}))) \\
& \quad \textbf{in } (\widehat{\rho}_n,\tau_n,t_n) \\
U(\widehat{\rho}_1,(\widehat{\rho},\tau,t)) = {} & \textbf{if } \widehat{\rho}_1 \not\sqsubseteq \widehat{\rho} \textbf{ then } (\widehat{\rho}\sqcup\widehat{\rho}_1,\tau,t+1) \textbf{ else } (\widehat{\rho},\tau,t)
\end{aligned}
$$

Fig. 2. Time-stamps based approximation equation

$\tau \in Lab \to \mathbf{N}$:

$$(\widehat{\rho},\tau,t) \in States = (A \times (Lab \to \mathbf{N}) \times \mathbf{N})$$

Unlike in the standard denotational semantics, we consider $\mathbf{N}$ with the usual ordering on natural numbers. Therefore *States* is an infinite domain containing infinite ascending chains. To limit the height of ascending chains, we restrict the space to reflect more precisely the set of possible states in the computation:

$$States = \{(\widehat{\rho},\tau,t) \in (A \times (Lab \to \mathbf{N}) \times \mathbf{N}) | t \le h(\widehat{\rho}) \wedge \forall \ell \in Lab . \tau(\ell) \le t\}$$

Here the function $h(\widehat{\rho})$ defines "the length of the longest chain of elements of A below $\widehat{\rho}$".

Informally, the restriction accounts for the fact that we increment t each time we add information into $\widehat{\rho}$. Starting from $\widehat{\rho} = \bot$ and $t = 0$, t is always smaller than the longest ascending path from bottom to $\widehat{\rho}$ in A. The second condition accounts for the fact that the time-stamps table records time stamps smaller than or equal to the value of the time counter.

The recursive equations that define the time-stamps approximation are stated in Figure 2. They define a function $F' : (Lab \times States) \to States$ that models a state-passing computation. It is easy to show that $U : (A \times States) \to States$ is well-defined (on the restricted space of states). The existence of solutions for the equations from Figure 2 can then be easily established, also due to the monotonicity of B, V and R.

Note also that the order in which the elements of the set of outgoing nodes $R(\ell,\widehat{\rho})$ are processed remains unspecified. This aspect does not affect our further development, while leaving room for improving the evaluation strategy.

The main reason for the restriction on the states and for the non-standard

semantics is that we restrict the definition of the function to the strictly-terminating instances. It is easy to show that F' terminates on any initial program point and initial state. In fact, such initial configuration determines a trace of states which we use to show that the function F' computes a safe approximation of the analysis.

3.2 Correctness

The correctness of the time-stamps based algorithm, i.e., the fact that it computes an approximation of the function defined by Equation $(*)$ on page 46, is established by the following theorem.

Theorem 3.1 *For any $\ell \in Lab$ and $\widehat{\rho} \in A$:*

$$F(\ell, \widehat{\rho}) \sqsubseteq \pi_1(F'(\ell, (\widehat{\rho}, \lambda\ell.0, 1)))$$

The theorem is proven in two steps. First, we show that using time stamps to control recursion does not change the result of the analysis. In this sense, we consider an intermediate equation defining a function $F'' : (Lab \times States) \to A$.

$$F''(\ell, (\widehat{\rho}, \tau, t)) = \textbf{if } \tau(\ell) = t \textbf{ then } \bot \\ \textbf{else } B(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell, \widehat{\rho})} F''(\ell', U(V(\ell, \ell', \widehat{\rho}), (\widehat{\rho}, \tau[\ell \mapsto t], t)))$$

We show that the function F'' computes the same analysis as the function defined by Equation $(*)$.

Lemma 3.2 *Let $\ell \in Lab$ be a program point and $(\widehat{\rho}, \tau, t) \in States$. Let $S = \{\ell' \in Lab | \tau(\ell') = t\}$. Then we have:*

$$F''(\ell, (\widehat{\rho}, \tau, t)) \sqcup \bigsqcup_{\ell' \in S} F(\ell', \widehat{\rho}) = F(\ell, \widehat{\rho}) \sqcup \bigsqcup_{\ell' \in S} F(\ell', \widehat{\rho})$$

Lemma 3.2 is proved using a well-founded induction on states: essentially, $F''(\ell, (\widehat{\rho}, \tau, t))$ makes recursive calls on F'' on states strictly above $(\widehat{\rho}, \tau, t)$. As an instance of the Lemma 3.2 we obtain:

Corollary 3.3

$$\forall \ell \in Lab, \widehat{\rho} \in A. F(\ell, \widehat{\rho}) = F''(\ell, (\widehat{\rho}, \lambda\ell.0, 1))$$

We show that the time-stamps algorithm computes an approximation of the function F''.

Lemma 3.4

$$\forall (\widehat{\rho}, \tau, t) \in States, \ell \in Lab. F''(\ell, (\widehat{\rho}, \tau, t)) \sqsubseteq \pi_1(F'(\ell, (\widehat{\rho}, \tau, t)))$$

The proof of Lemma 3.4 relates the recursion tree from the definition of function F'' and the trace of states in the computation of F'. In essence, the

value of $F''(\ell, (\widehat{\rho}, \tau, t))$ is composed from the union of a tree of values of the from $B(\ell_i, \widehat{\rho}_i)$. We show by induction on the depth of the tree that each of these values is accumulated in the final result at some point on the trace of states in the computation of F'.

Combining the two lemmas we obtain the statement of Theorem 3.1.

Even if we have used non-standard ordering and domains when defining the solutions of the equations in Figure 2, showing that the function F' agrees on the starting configuration with a standard semantic definition of the algorithm in Figure 1 is trivial and is not part of the current presentation.

3.3 Complexity estimates

Let us assume that computing the function U takes m time units, and that B, R and V can be computed in constant time (one time unit). The time-counter can be incremented at most $h(A)$ times, where the function h defines the height (given by the longest ascending chain) of the lattice A. In the worst case, between two increments, each edge in the graph may be explored, and for each edge we might spend m time units in computing the U function. Thus, computing $F'(\ell, (\widehat{\rho}, \lambda\ell.0, 1))$ has a worst-case complexity of $O(|Lab|^2 \cdot m \cdot h(A))$.

Space-wise, it is immediate to see that at most two elements of A are in memory at any given time: the global value $\widehat{\rho}$ and one temporary value created at each call of B or V. The temporary value is not of a concern though: in most usual cases, the size of the results of B or V is one order of magnitude smaller than the size of $\widehat{\rho}$.

The worst-case space complexity is also driven by the exploration of edges. It is immediate to see that each edge might put aside between two updates of the global environment. Denoting with $S(A)$ the size of an element in A, the worst-case space complexity might be $O(S(A) + |Lab|^2 \cdot h(A))$. It seems apparent however that many of the edges put aside are redundant. We are currently exploring possibilities of removing some of these redundancies.

4 An extension

The time-stamps method has originally been presented in the setting of flow analysis of Scheme programs in continuation-passing style (CPS) [19]. Indeed, the formulation of the equations facilitate the analysis of a computation that "never returns". In their paper on CPS versus direct style in program analysis [16], Sabry and Felleisen also use a memoization technique to compute the result of their constant propagation for a higher-order language in direct style.

We can apply the time-stamps based technique to Sabry and Felleisen's analysis in order to compute a more efficiently an approximate solution. In order to do so, we extend the applicability of time-stamps based algorithm to functions which, after following a set of edges, return to the current point and

restart an analysis over a newer set of edges. We consider equations of the following form:

$$F(\ell,\widehat{\rho}) = \textbf{let } \widehat{\rho}_1 = B(\ell,\widehat{\rho}) \sqcup \bigsqcup_{\ell' \in R(\ell,\widehat{\rho})} F(\ell', \widehat{\rho} \sqcup V(\ell,\ell',\widehat{\rho}))$$
$$\textbf{in } B'(\ell,\widehat{\rho}_1) \sqcup \bigsqcup_{\ell' \in R'(\ell,\widehat{\rho}_1)} F(\ell', \widehat{\rho}_1 \sqcup V'(\ell,\ell',\widehat{\rho}_1))$$

Indeed, in order to model the analysis of a term like **let** $x = V_1\ V_2$ **in** M, Sabry and Felleisen's analysis explores all possible functions that can be called in the header of the let, joins the results and, afterwards, analyzes the term M.

The algorithm is straightforwardly extended to account for the second call with another iteration over $R'(\ell,\widehat{\rho}_1)$. The proof of correctness extends as well. It is remarkable that despite the more complicated formulation of the equations, the complexity of the algorithm remains the same, due to the bounds imposed by the time-stamps.

The benefit of applying the time-stamps based algorithm to Sabry and Felleisen's analysis is that it yields a more efficient algorithm than their proposed memoization-based implementation (for the reasons outlined in Section 2.1). The approximation obtained is still precise enough. In particular, the time-stamps based analysis is able to distinguish returns. Consider for instance the following example (also due to Sabry and Felleisen):

$$\begin{array}{ll}\textbf{let} & f = \lambda x.x \\ & x_1 = f\ 1 \\ & x_2 = f\ 2 \\ \textbf{in} & x_1\end{array}$$

The time-stamps based analysis computes a solution in which x_1 (and, therefore, the result of the entire expression) is bound to 1, and x_2 is bound to $\top$. In contrast, a constraint-based data-flow analysis [13] is only able to compute a solution in which both x_1 and x_2 are bound to $\top$.

Formally, it is relatively easy to show that the time-stamps based constant propagation is always computing a result at least as good as a standard constraint-based data-flow analysis. The details are omitted from this article. Note that the improvement in the quality over the constraint-based analysis comes at a price in the worst-case time and space complexity.

5 Related work

A number of authors describe algorithms for computing least fixed points as solutions to program analyses using chaotic iteration, which are also adapted to compute approximation by using widenings or narrowings [6]. O'Keefe's bottom-up algorithm [14] has inspired a significant number of articles, where the convergence speed is improved using refined strategies on choosing the next iteration, or exploiting locality properties of the specifications [1,11,15].

Such algorithms have also been applied to languages with dynamic control flow. Chen, Harrison and Yi [4] developed advanced techniques such as "waiting for all successors", "leading edge first", "suspend evaluation", which improve the behavior of the bottom-up algorithm when applied to such languages. In a subsequent work [3], the authors use reachability information to implement a technique called "context projection" which reduces the amount of abstract information associated to each program point. In contrast, time stamps approximate the solution, by maintaining only one global context common to all program points.

Other algorithms that address languages with dynamic flow have been developed in the context of strictness analysis. Clack and Peyton-Jones [5] have introduced the frontier-based algorithm. The algorithm reduces space usage by representing the solution only with a subset of relevant values. The technique has been developed for binary lattices. Hunt's PhD thesis [9] contains a generalization to distributive lattices.

The top-down vs. bottom-up aspects of fixed-point algorithms for abstract interpretation of logic programs have been investigated by Le Charlier and Van Hentenryck. The two authors have developed a generic top-down algorithm fixed-point algorithm [2], and have compared it with the alternative bottom-up strategy. The evaluation strategy of their algorithm is similar to the time-stamps based one in this article. In contrast, however, since their algorithm precisely computes the least fixed point, it also maintains multiple values from the lattice of results.

Fecht and Seidl [7] design the time-stamps solver "WRT" which combines the benefits of both the top-down and bottom-up approaches. The algorithm also uses time stamps, in a different manner though: the time stamps are used to interpret the algorithm's worklist as a priority queue. Our technique uses time stamps simply to control the termination of the computation. In a sequel paper [8], the authors derive a fixed-point algorithm for distributive constraint systems and use it, for instance, to compute a flow graph expressed as a set of constraints.

6 Conclusion

We have presented a polynomial-time algorithm for approximating the least fixed point of a certain class of recursive equations. The algorithm uses time stamps to control recursion and avoids duplication of analysis information over program branches by reusing intermediate results. The time-stamping technique has originally been introduced by Shivers in his PhD thesis [19]. To the best of our knowledge, the idea has not been pursued. We have presented a formalization of the technique and we have proven its correctness.

Several issues regarding the time-stamps based algorithm might be worth further study. For instance, it is noticeable that the order in which the outgoing edges are processed at a certain node might affect the result of the analysis.

Designing an improved strategy for selection of nodes is worth investigating. Also, as we observed in Section 3.3, an edge might be processed several times independently, each time with a larger analysis information. This suggests that some of the processing might be redundant. We are currently investigating such a possible improvement of the algorithm, and its correctness proof.

7 Acknowledgments

I am grateful to Olivier Danvy, David Toman and Zhe Yang for comments and discussion on this article. Thanks are also due to the anonymous referees for their comments.

References

[1] Bourdoncle, F., *Efficient chaotic iteration strategies with widenings*, in: *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, Lecture Notes in Computer Science **735** (1993), pp. 128–141.

[2] Charlier, B. L. and P. V. Hentenryck, *A universal top-down fixpoint algorithm*, Technical Report CS–92–25, Brown University, Providence, Rhode Island (1992).

[3] Chen, L.-L. and W. L. Harrison, *An efficient approach to computing fixpoints for complex program analysis*, in: *Proceedings of the 8th ACM International Conference on Supercomputing* (1994), pp. 98–106.

[4] Chen, L.-L., W. L. Harrison and K. Yi, *Efficient computation of fixpoints that arise in complex program analysis*, Journal of Programming Languages **3** (1995), pp. 31–68.

[5] Clack, C. and S. L. Peyton Jones, *Strictness analysis—a practical approach*, in: J.-P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **201** (1985), pp. 35–49.

[6] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: R. Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages* (1977), pp. 238–252.

[7] Fecht, C. and H. Seidl, *An even faster solver for general systems of equations*, in: R. Cousot and D. A. Schmidt, editors, *Proceedings of 3rd Static Analysis Symposium*, Lecture Notes in Computer Science **1145** (1996), pp. 189–204.

[8] Fecht, C. and H. Seidl, *Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems*, in: C. Hankin, editor, *Proceedings of the 7th European Symposium on Programming*, Lecture Notes in Computer Science **1381** (1998), pp. 90–104.

[9] Hunt, S., "Abstract Interpretation of Functional Languages: From Theory to Practice," Ph.D. thesis, Department of Computing, Imperial College of Science Technology and Medicine, London, UK (1991).

[10] Jones, N. D. and F. Nielson, *Abstract interpretation: A semantics-based tool for program analysis*, in: S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Semantic Modelling*, The Handbook of Logic in Computer Science **4** (1995), pp. 527–636.

[11] Jørgensen, N., *Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration*, in: B. L. Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science (1994), pp. 329–345.

[12] Kam, J. B. and J. D. Ullman, *Monotone data flow analysis frameworks*, Acta Informatica **7** (1977), pp. 305–317.

[13] Nielson, F., H. R. Nielson and C. Hankin, "Principles of Program Analysis," Springer-Verlag, 1999.

[14] O'Keefe, R. A., *Finite fixed-point problems*, in: J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference* (1987), pp. 729–743.

[15] Rosendahl, M., *Higher-order chaotic iteration sequences*, in: M. Bruynooghe and J. Penjam, editors, *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, number 714 in Lecture Notes in Computer Science (1993), pp. 332–345.

[16] Sabry, A. and M. Felleisen, *Is continuation-passing useful for data flow analysis?*, in: V. Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 29, No 6 (1994), pp. 1–12.

[17] Sagiv, S., T. W. Reps and S. Horwitz, *Precise interprocedural dataflow analysis with applications to constant propagation*, Theoretical Computer Science **167** (1996), pp. 131–170.

[18] Schmidt, D. A., *Natural-semantics-based abstract interpretation*, in: A. Mycroft, editor, *Static Analysis*, number 983 in Lecture Notes in Computer Science (1995), pp. 1–18.

[19] Shivers, O., "Control-Flow Analysis of Higher-Order Languages," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania (1991), Technical Report CMU-CS-91-145.

[20] Young, J. and P. Hudak, *Finding fixpoints on function spaces*, Technical Report YALEEU/DCS/RR-505, Yale University, New Haven, CT (1986).

MFPS 17 Preliminary Version

A New Approach to Quantitative Domain Theory

Lei Fan [1,2]

Department of Mathematics
Capital Normal University
Beijing 100037, P.R.China

Abstract

This paper introduces a new approach to the theory of Ω-categories enriched by a frame. The approach combines ideas from various areas such as generalized ultrametric domains, Ω-categories, constructive analysis, and fuzzy mathematics. As the basic framework, we use the Wagner's Ω-category [18,19] with a frame instead of a quantale with unit. The objects and morphisms in the category will be called L-Fuzzy posets and L-Fuzzy monotone mappings, respectively. Moreover, we introduce concepts of adjoints and a kind of convergence in an L-Fuzzy poset that makes the theory "constructive" or "computable".

1 Introduction

Quantitative Domain Theory has attracted much attention [4], [15], [17], and [18]. Amongst these developments, K.Wagner's theory of Ω-categories is most general, and J.J.M.M.Rutten's theory of generalized ultrametric domains is closest to the standard domain theory. So it is natural to think that some of the properties about the latter, especially those that closely connected with the operational and topological properties of the unit interval [0,1], may not be generalized to the theory of Ω-categories without restricted conditions on the valued quantale. Of course this is right in general, but it is not always true as K.Wagner's work shows. In this paper we provide more examples to further support this observation.

In section 2, we review some materials essential for this paper. As the basis we use Wagner's Ω-category [18] with a frame instead of a commutative quantale with unit. However, the method used in this paper applies to the

[1] This work is supported by China National Natural Science Foundations
[2] Email: fanlei63@hotmail.com

general case. The objects and morphisms in the category will be called *L-Fuzzy* posets and *L-Fuzzy* monotone mappings respectively because we hope to stress the fuzzy view that this paper takes. We then prove a representation theorem which shows that every L-Fuzzy preordered set can be represented by a family of preorders on that set properly glued together. In the end of the section, we propose a theory of adjoint pairs on L-Fuzzy monotone mappings which is a generalization of Rutten's theory of metric adjoint pairs. In section 3, we introduce a theory of convergence in L-Fuzzy posets. The theory is based on a simple idea from constructive analysis, that is, replacing the arbitrary $\epsilon > 0$ with a proper "computable" sequence such as $\{1/n\}$. So our work can be seen as a constructive version of Wagner's theory of liminf convergence. In the final section, we develop a theory for recursive domain equations in the category of L-Fuzzy posets and L-Fuzzy adjoint pairs, following the methods of J.J.M.M.Rutten [15].

2 LF-posets and LF-monotone mappings

First, we review some basic concepts from the theory of Ω-categories in a slightly different form, see [19] for details. Note that we use a frame instead of a commutative quantale with unit.

In what follows, $(L, \leq)$ will denote a fixed nontrivial frame (or complete Heyting algebra) with maximal element 1 and minimal element 0. For $a, b \in L$, the meet, union and implication in L will be denoted by $a \wedge b$, $a \vee b$ and $a \to b$ respectively.

Definition 2.1 Let X be a non-empty set, $e : X \times X \longrightarrow L$ a mapping. e is called an *L-Fuzzy preorder* on X if it satisfies the following conditions:

1. for all $x \in X, e(x, x) = 1$,
2. for all $x, y, z \in X, e(x, z) \wedge e(x, y) \leq e(y, z)$.
   The pair (X, e) or X is called an *L-Fuzzy preordered set*. If e satisfies the additional condition
3. for all $x, y \in X, e(x, y) = e(y, x) = 1 \Rightarrow x = y$,
   then it is called an *L-Fuzzy partial order on* X and (X, e) is called an *L-Fuzzy partial ordered set* (abbreviated as *L-Fuzzy poset* or *LF-poset*).
4. Let (X, e_X) and (Y, e_Y) be L-Fuzzy preordered sets, $f : X \longrightarrow Y$ a mapping. f is called an *L-Fuzzy monotone mapping* if for all $x, y \in X$,

$$e_Y(f(x), f(y)) \geq e_X(x, y).$$

The category of LF-preordered sets (LF-posets) and LF-monotone mappings will be denoted by **LF-Pre** (**LF-POS**).

Remark 2.2 (1) If $L = \{0, 1\}$, then the category **LF-Pre** (**LF-POS**) can be identified with the category **Pre** (**POS**) of ordinary preordered sets

(partially ordered sets) and monotone mappings.

(2) If $L = [0,1]$, then the category **LF-Pre** (**LF-POS**) can be identified with the category **Gums** (**Qums**) of Rutten's generalized ultrametric spaces (quasi ultrametric spaces) and non-expansive mappings through the relation defined below:

$$e(x,y) = 1 - d(x,y), x, y \in X.$$

Intuitively, $e(x,y)$ is interpreted as the degree of $x \leq y$. This partially justifies the term *L-Fuzzy*. Of course, there are other reasons for that. See the following Example 2.3(2),[10], [11], and [13] for more information.

Example 2.3 (1) Let $(X, \leq)$ be a preordered set. For $x, y \in X$, let

$$e_{\leq}(x,y) = 1 \Longleftrightarrow x \leq y.$$

Then $(X, e_{\leq})$ is an L-Fuzzy preordered set. Moreover, $(X, e_{\leq})$ is an L-Fuzzy poset when $\leq$ is a partial order on X.

(2) Let $A : X \longrightarrow L$ be an L-Fuzzy set on X. For $x, y \in X$, let

$$e_A(x,y) = A(x) \to A(y).$$

Then (X, e_A) is an L-Fuzzy preordered set. In particular, every frame L can be seen as an L-Fuzzy preordered set by letting $X = L$ and $A = id_L$.

Let (X, e_X) and (Y, e_Y) be L-Fuzzy preordered sets, and

$$Y^X = [X \to Y] = \{f \mid f : X \longrightarrow Y \text{is } L\text{-monotone}\}.$$

We can make Y^X as an L-Fuzzy preordered set by defining

$$E_{Y^X}(f,g) = \bigwedge\{e_Y(f(x), g(x)) \mid x \in X\}, f, g \in Y^X.$$

Moreover, we define the *noise* between f and g as

$$\delta\langle f, g\rangle = E_{X^X}(id_X, g \circ f) \wedge E_{Y^Y}(f \circ g, id_Y).$$

Let (X, e) be an L-Fuzzy preordered set and $x, y \in X$, $a \in L$. Define a relation $\sqsubseteq_a$ on X as follows: $x \sqsubseteq_a y \Longleftrightarrow e(x,y) \geq a$. Then it is easy to check that $\sqsubseteq_a$ is a preorder on X for all $a \in L$. In fact we have:

Theorem 2.4 *(The decomposition theorem) Let (X, e) be an L-Fuzzy preordered set. Then*

(1) *If $a \leq b$, then $\sqsubseteq_b \subseteq \sqsubseteq_a$.*

(2) *For all $S \subseteq L$, if $a = \bigvee S$, then $\sqsubseteq_a = \bigcap\{\leq_s \mid s \in S\}$.*

(3) *For all $x, y \in X$, $e(x,y) = \bigvee\{a \in L \mid x \sqsubseteq_a y\}$.*

Moreover, if $f : X \longrightarrow Y$ is a mapping between L-Fuzzy preordered sets, then f is L-monotone if and only if for all $a \in L$, $f : (X, \sqsubseteq_a) \longrightarrow (Y, \sqsubseteq_a)$ is monotone, that is, $x \sqsubseteq_a y \Longrightarrow f(x) \sqsubseteq_a f(y)$. □

Theorem 2.5 *(The representation theorem) Let X be a set and $\mathcal{F} = \{R_a \mid a \in L\}$ a family of preorders on X with the following properties:*

(1) *if $a \leq b$, then $R_b \subseteq R_a$;*

(2) *for all $S \subseteq L$, $R_a = \bigcap\{R_s \mid s \in S\}$ when $a = \bigvee S$.*

Then $(X, e_{\mathcal{F}})$ is an L-Fuzzy preordered set, where

$$e_{\mathcal{F}}(x, y) = \bigvee\{a \in L \mid (x, y) \in R_a\}, x, y \in X.$$

Moreover, suppose that X, Y are sets with $\mathcal{F} = \{R_a \mid a \in L\}$, $\mathcal{G} = \{T_a \mid a \in L\}$ satisfying properties (1) and (2) above, and $f : X \longrightarrow Y$ a mapping such that for all $a \in L$, $f : (X, R_a) \longrightarrow (Y, T_a)$ is monotone. Then $f : (X, e_{\mathcal{F}}) \longrightarrow (Y, e_{\mathcal{G}})$ is an L-monotone mapping. □

The proof of above Theorems are routine.

It is interesting to note that Theorem 2.4 and Theorem 2.5 can be rephrased in the language of (pre-)sheaves as follows. Recall that a *presheaf* on L is a contravariant functor $F : L \longrightarrow \mathbf{Set}$ from L (seen as a category) to the category **Set** of sets and mappings. One obtains a $\mathcal{C}$ -presheaf if one replaces **Set** with a more general category $\mathcal{C}$ with proper structures.

Let $PO(X)$ denote the poset (so a category) of all preorders on set X with subset inclusion as the order. Then it is easy to see that condition (1) in Theorem 2.5 is equivalent to saying that $\mathcal{F} = \{R_a \mid a \in L\}$ is a $PO(X)$-presheaf on L and condition (2) is exactly the sheaf condition.

It is well know that the theory of adjoint pairs plays an essential role in domain theory. J.J.M.M.Rutten [15] and F.Alesi et al. [2] established a truly quantitative version of the classical theory of adjoints. We will now set up a theory of adjoints about LF- monotone mappings that is a generalization to Rutten's.

For $a, b, \eta \in L$, set $a * b = (a \to b) \wedge (b \to a)$ and $a \approx_\eta b \Leftrightarrow a * b \geq \eta$. In informal fuzzy logic terms, $a * b$ is the "degree" of equivalence of propositions a and b, whereas $a \approx_\eta b$ means that a and b are equivalent "up to degree η" at least.

Definition 2.6 Let (X, e_X) and (Y, e_Y) be LF-preordered sets, $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$ LF-monotone mappings and $\eta \in L$. If for all $x \in X$, $y \in Y$,

$$e_Y(f(x), y) \approx_\eta e_X(x, g(y)),$$

then f, g is called an η-adjoint pair, and denoted by $f \dashv_\eta g$.

Theorem 2.7 *Let (X, e_X) and (Y, e_Y) be LF-preordered sets, $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$ LF-monotone mappings and $\eta \in L$. Then the following*

conditions are equivalent:

(1) $f \dashv_\eta g$;

(2) $\delta\langle f, g\rangle \approx_\eta 1$;

(3) *for all* $x \in X$, $y \in Y$, $\epsilon \leq \eta$, $f(x) \leq_\epsilon y \Leftrightarrow x \leq_\epsilon g(y)$;

(4) $id_X \sqsubseteq_\eta g \circ f$, $f \circ g \sqsubseteq_\eta id_Y$. □

The essential part of the proof is a simple result from frame theory as below.

Lemma 2.8 *Let L be a frame and $a, b, \eta \in L$. The the following conditions are equivalent:*

(1) $a \approx_\eta b$;

(2) $a \wedge \eta = b \wedge \eta$;

(3) $a \to \eta = b \to \eta$;

(4) *for all* $\epsilon \in L, \epsilon \leq \eta, \epsilon \leq a \Leftrightarrow \epsilon \leq b$.

3 A Theory of Convergence in LF-posets

In this section, we introduce a theory of convergence in LF-posets. It is based on a very simple and intuitive idea from constructive analysis, that is, we replace arbitrary $\epsilon > 0$ with a computable sequence decreasing to 0 (such as $\{1/n\}$) for all practical purposes, see [3] for example. We generalize the idea to LF-posets. In fact, the resulting theory is a special case of Wagner's liminf theory of convergence.

Definition 3.1 Let $\eta = (\eta_n)_{n\in\omega}$ be an increasing sequence in L and $\bigvee\{\eta_n \mid n \in \omega\} = 1$. Then η is called a *testing sequence*.

Example 3.2 (1) Let $L = \{0, 1\}$ and for all $n \in \omega$, $\eta_n = 1$. Then $\eta = (\eta_n)$ is a testing sequence in L. This corresponds to the classical theory based on preordered sets.

(2) Let $L = [0, 1]$ and for all $n \in \omega$, $\eta_n = 1 - (1/n)$. Then $\eta = (\eta_n)$ is a testing sequence in L. This corresponds to Rutten's generalized ultrametric theory.

(3) Let $L = \omega \cup \{\omega\}$ and for all $n \in \omega$, $\eta_n = n$. Then $\eta = (\eta_n)$ is a testing sequence in L. This corresponds to Monteiro's theory of **sfe** (sets with families of equivalence), see [14] for the details.

Definition 3.3 Let (X, e) be a non-empty LF-poset, $(x_n)_{n\in\omega}$ a sequence in X.

(1) (x_n) is said to be converging to x *with respect to* η (η-converges to x, briefly) and denoted by $x = \eta\text{-}\lim x_n$ if there exists an $x \in X$ such that

for every $N \in \omega$ and $a \in X$,

$$\bigwedge_{n \geq N} e(x_n, a) \approx_{\eta_N} e(x, a).$$

(2) (x_n) is called a (*forward*) *Cauchy sequence with respect to* η (η-*Cauchy sequence*, briefly) if for every $N \in \omega$ and $m \geq n \geq N$, $e(x_n, x_m) \geq \eta_N$, or equivalently, $e(x_n, x_{n+1}) \geq \eta_N$ for all $n \geq N$.

(3) (X, e) is called η-*complete* if every η-Cauchy sequence in X converges.

The category of η-complete LF-posets and LF-monotone mappings will be denote by η-**CPO**.

Remark 3.4 An anonymous referee points out to the author that the convergence w.r.t η is a special instance of the notion of weighted-(co)limit from enriched category theory, see [5]. For the case of metric spaces see [16].

Example 3.5 Let $L = \{0, 1\}$, and η is the testing sequence in Example 3.2(1). Then a sequence (x_n) in X has the limit x w.r.t η if and only if that x is the least upper bound of the set $\{x_n \mid n \in \omega\}$. Moreover, (x_n) is η-Cauchy if and only if it is an increasing sequence in X. So we have:

Theorem 3.6 *Let X be a poset seen as an LF-poset as in Example 2.3(1) and η be the testing sequence defined in Example 3.2(1). Then X is η-complete if and only if it is an ω-dcpo.* □

Theorem 3.7 *Let (X, e) be an LF-poset, (x_n) a sequence in X and $x \in X$. Then $x = \eta\text{-}\lim x_n$ if and only if the following conditions hold:*

(1) $\bigwedge_{n \geq N} e(x_n, x) \geq \eta_N, N \in \omega$;

(2) $\bigwedge_{n \geq N} e(x_n, a) \leq e(x, a), N \in \omega, a \in X$. □

Corollary 3.8 *Let (x_n) be a sequence in X and $x \in X$. If $x = \eta\text{-}\lim x_n$ then:*

(1) $n \geq N, e(x_n, x) \geq \eta_N, N \in \omega$;

(2) *If $x' \in X$ such that the condition (1) holds then $e(x, x') = 1$.* □

The conditions (1) and (2) in Corollary 3.8 can be interpreted in order-theoretic terms as follows:

(1') for all $N \in \omega$, $n \geq N$, $x_n \sqsubseteq_{\eta_N} x$,

(2') If $x' \in X$ such that the condition (1') holds, then $x \sqsubseteq_{\eta_N} x'$.

In other words, x is the least upper bound of set $\{x_n \mid n \in \omega, n \geq N\}$ at the level η_N for all $N \in \omega$.

Theorem 3.9 *Let L be a frame seen as an LF-poset as in Example 2.3(2) and let η be a testing sequence in L. If (x_n) is an η-Cauchy sequence in L, then*

$$\eta\text{-}\lim x_n = \bigvee \bigwedge \{x_n \mid N \in \omega, n \geq N\}.$$

In particular, L is η-complete as an LF-poset. □

Definition 3.10 Let (X, e_X), (Y, e_Y) be LF-posets and $f : (X, e_X) \longrightarrow (Y, e_Y)$ be an LF-monotone mapping.

(1) f is called *η-continuous* if for every convergent sequence (x_n) in X, $(f(x_n))$ is a convergent sequence in Y, and

$$f(\eta\text{-}\lim x_n) = \eta\text{-}\lim f(x_n).$$

The set $C(X, Y)$ of all η-continuous mappings from X to Y is an LF-poset too when it is seen as a subset of $Y^X = [X \to Y]$.

(2) f is called *η-approximate* if for all $x, y \in X$, $N \in \omega$,

$$e(x, y) \geq \eta_N \Longrightarrow e(f(x), f(y)) \geq \eta_{N+1}.$$

The term "approximate" was coined by L.Monteiro in [14]. It is a constructive form of contraction mapping in the theory of metric spaces.

Remark 3.11 It is well know that every contraction mapping is continuous in the standard metric space. But it is not true in the present case. In fact, η-continuous and η-approximate mappings are incomparable.

Theorem 3.12 *Suppose X, Y are LF-posets and Y is η-complete. Then $C(X, Y)$ is also η-complete.* □

Theorem 3.13 (Fixed Point Theorem) *Let (X, e) be an η-complete LF-poset and $f : X \longrightarrow X$ an LF-monotone mapping.*

(1) *If f is η-continuous and there exists an $x \in X$ such that $e(x, f(x)) = 1$, then f has a fixed point.*

(2) *If f is η-continuous and η-approximate and there exists an $x \in X$ such that $e(x, f(x)) \geq \eta_0$, then f has a fixed point.* □

The proof of Theorem 3.13 is similar to the corresponding result of generalized ultrametric spaces, see Theorem 6.3 in [15].

4 Domain Equations in the category η-CPO

In this section, we develop a theory for solving domain equations in the category of η-complete LF-posets and LF-adjoint pairs following the methods of J.M.Rutten [15]. Proofs of results in this section are similar to the cases of generalized ultrametric spaces, see [6] for details.

As basic framework we use the category η-$\mathbf{CPO}^P$ (P stand for pairs) of η-complete LF-posets and η-continuous LF-adjoint pairs, that is, objects in η-$\mathbf{CPO}^P$ are η-complete LF-posets and morphisms in η-$\mathbf{CPO}^P$ are pairs $\langle f, g\rangle : X \longrightarrow Y$, where $f : X \longrightarrow Y$ and $g : Y \longrightarrow X$ are η-continuous mappings. The composition of morphisms is defined as usual: if $\langle f, g\rangle : X \longrightarrow Y$, $\langle h, k\rangle : Y \longrightarrow Z$ are morphisms in η-$\mathbf{CPO}^P$, then $\langle f, g\rangle \circ \langle h, k\rangle = \langle h \circ f, g \circ k\rangle$.

Definition 4.1 (1) A sequence

$$X_0 \stackrel{\langle f_0,g_0\rangle}{\longrightarrow} X_1 \stackrel{\langle f_1,g_1\rangle}{\longrightarrow} \cdots$$

in η-$\mathbf{CPO}^P$ is called an *η-Cauchy chain* if for every $N \in \omega$ and $n \geq N$, $f_n \dashv_{\eta_N} g_n$, or equivalently, $\delta\langle f_n, g_n\rangle \approx_{\eta_N} 1$.

(2) Let

$$X_0 \stackrel{\langle f_0,g_0\rangle}{\longrightarrow} X_1 \stackrel{\langle f_1,g_1\rangle}{\longrightarrow} \cdots$$

be an η-Cauchy chain in η-$\mathbf{CPO}^P$. A cone of the chain is a sequence $\{\langle \alpha_k, \beta_k\rangle : X_k \to X\}$ of morphisms in η-$\mathbf{CPO}^P$ such that

$$\langle \alpha_k, \beta_k\rangle = \langle \alpha_{k+1}, \beta_{k+1}\rangle \circ \langle f_k, g_k\rangle$$

for every $k \in \omega$.

(3) A cone $\{\langle \alpha_k, \beta_k\rangle : X_k \to X\}$ is a colimit if it is initial, that is, for every other cone $\{\langle \alpha'_k, \beta'_k\rangle : X_k \to X'\}$, there exists an unique morphism $\langle f, g\rangle : X \longrightarrow X'$ such that

$$\langle \alpha'_k, \beta'_k\rangle = \langle \alpha_k, \beta_k\rangle \circ \langle f, g\rangle$$

for every $k \in \omega$.

We will use the following conventions. For all $k, l \in \omega$, $k < l$,

$$f_{kl} = f_{l-1} \circ \cdots f_{k+1} \circ f_k, g_{kl} = g_k \circ g_{k+1} \cdots g_{l_1}.$$

Note that $f_{k,k+1} = f_k$, $g_{k,k+1} = g_k$.

Theorem 4.2 *Let*

$$X_0 \stackrel{\langle f_0,g_0\rangle}{\longrightarrow} X_1 \stackrel{\langle f_1,g_1\rangle}{\longrightarrow} \cdots$$

be an η-Cauchy chain in η-$\mathbf{CPO}^P$ and $\{\langle \alpha_k, \beta_k\rangle : X_k \longrightarrow X\}$ a cone of the chain. Then $\{\langle \alpha_k, \beta_k\rangle\}$ is a colimit if and only if the following conditions hold:

(1) *$\beta_k \circ \alpha_k = \eta\text{-}\lim_{l>k}(g_{kl} \circ f_{kl})$ for every $k \in \omega$.*

(2) *$\eta\text{-}\lim(\alpha_k \circ \beta_k) = id_X$.* □

Theorem 4.3 *Every η-Cauchy chain in η-$\mathbf{CPO}^P$ has an unique colimit cone.* □

Definition 4.4 Suppose $F : \mathbf{LF-POS} \longrightarrow \mathbf{LF-POS}$ be a functor and

$$F_{XY} : Y^X \longrightarrow F(Y)^{F(X)}$$

denote the mapping $f \mapsto F(f)$ for LF-posets X, Y.

(1) F is said to be local *LF-monotone* if F_{XY} is LF-monotone for any LF-posets X, Y.

(2) F is said to be local *η-continuous* if F_{XY} is η-continuous for any LF-posets X, Y.

(3) F is said to be local *η-approximate* if F_{XY} is η-approximate for any LF-posets X, Y.

Every functor $F : \mathbf{LF-POS} \longrightarrow \mathbf{LF-POS}$ can be extended to a functor $F^P : \eta\text{-}\mathbf{CPO}^P \longrightarrow \eta\text{-}\mathbf{CPO}^P$ as follows: $F^P(X) = F(X)$ for every object X in $\eta\text{-}\mathbf{CPO}^P$ and $F^P(\langle f, g\rangle) = \langle F(f), F(g)\rangle$ for every morphism $\langle f, g\rangle$ in $\eta\text{-}\mathbf{CPO}^P$. The functor $F^P : \eta\text{-}\mathbf{CPO}^P \longrightarrow \eta\text{-}\mathbf{CPO}^P$ is said to be *local LF-monotone* (*local η-continuous, local η-approximate*, respectively) if the corresponding functor F is.

Theorem 4.5 *Let $F^P : \eta\text{-}\mathbf{CPO}^P \longrightarrow \eta\text{-}\mathbf{CPO}^P$ be the functor defined as above. Then:*

(1) *If F is local LF-monotone, then*

$$\delta(F^P(\langle f, g\rangle)) = \delta\langle F(f), F(g)\rangle \geq \delta\langle f, g\rangle$$

for every morphism $\langle f, g\rangle$ in $\eta\text{-}\mathbf{CPO}^P$.

(2) *If F is local η-approximate, then*

$$\delta\langle f, g\rangle \geq \eta_N \Longrightarrow \delta\langle F(f), F(g)\rangle \geq \eta_{N+1}$$

for every morphism $\delta\langle f, g\rangle$ in $\eta\text{-}\mathbf{CPO}^P$ and $N \in \omega$. □

As the case of generalized ultrametric spaces, we can now give a categorical version of the Theorem 3.13.

Theorem 4.6 *(The fixed point theroem, categorical version) Let $F^P : \eta\text{-}\mathbf{CPO}^P \longrightarrow \eta\text{-}\mathbf{CPO}^P$ be a functor.*

(1) *If F is local η-continuous and there exists an object X and a morphism $\langle f, g\rangle : X \longrightarrow F(X)$ of $\eta\text{-}\mathbf{CPO}^P$ such that $f \dashv g$. Then there exists an object Y of $\eta\text{-}\mathbf{CPO}^P$ satisfying that $F(Y) \cong Y$.*

(2) *If F is local η-continuous and η-approximate and there exists an object X and a morphism $\langle f, g\rangle : X \longrightarrow F(X)$ of $\eta\text{-}\mathbf{CPO}^P$ such that $f \dashv_{\eta_0} g$. Then there exists an object Y of $\eta\text{-}\mathbf{CPO}^P$ satisfying that $F(Y) \cong Y$.* □

Acknowledgement

The author is grateful to Professors M. Mislove, G.-Q. Zhang, and referees for their invaluable help to correct numerous errors, improve the presentation and make comments.

References

[1] Abramsky, S., A. Jung, Domain theory, in S.Abramsky, D.Gabbay, T.Maibaum, editors, "Handbook of Logic in Computer Science", vol.**3**, pp.1-168, Oxford University Press, 1995.

[2] Alesi, F., P. Baldan, G. Belle and J.J.M.M. Rutten, Solutions of functorial and non-functorial metric domain equations, Electronic Notes in Theoretical Computer Science **1** (1995). URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[3] Bishop, E., D. Bridges, "Constructive Analysis", Springer-Verlag 1985.

[4] Bonsangue, M.M., F.van Breugel, J.J.M.M.Rutten, Generalized Ultrametric spaces: completion, topology, and powerdomains via the Yoneda embedding, Theoretical Computer Science **193** (1998), pp.1-51.

[5] Borceux, F., G. M. Kelly, A notion of limit for enriched categories, Bull. Austral. Math. Soc. **12** (1975), pp.49-72.

[6] Fan, L., Some Questions in Domain Theory, Ph.D Thesis, Capital Normal University 2001. (In Chinese)

[7] Flagg, B., R. Kopperman, Continuity Spaces: Reconciling Domains and Metric Spaces, Theoretical Computer Science **177** (1997), pp.111-138.

[8] Flagg, B., R. Kopperman, Fixed points and reflexive domain equations in categories of continuity spaces, Electronic Notes in Theoretical Computer Science **1** (1995). URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[9] Flagg, B., P. Sünderhauf and K. Wagner. A Logical Approach to Quantitative Domain Theory. Preprint Submitted to Elsevier, 1996.

[10] Grayson, R. J., Heyting-valued semantics, in G.Lotti et al., editors, Logic Colloquium' 82, pp.181-208, Elservier Science 1983.

[11] Höhle, U., Fuzzy sets and subobjects, in A. Jones et al., editors, "Fuzzy Sets and Applications", pp.69-76, D.Reidel Publishing Company 1986.

[12] Lawvere, F. W., Metric Spaces, Generalized Logic and Closed Categories, Rend. Sem. Mat. e. Fisico di Milano **43** (1973), pp.135-166.

[13] Lowen, R., Mathematics and Fuzziness, in A.Jones et al., editors, "Fuzzy Sets and Applications", pp.3-38, D.Reidel Publishing Company 1986.

[14] Monteiro, L., Semantic domains based on sets with families of equivalences, Electronic Notes in Theoretical Computer Science **11**(1996). URL: `http://www.elsevier.nl/locate/entcs/volume11.html`.

[15] Rutten, J.J.M.M., Elements of generalized ultrametric domain theory, Theoreotical Computer Science, 170(1996), pp.349-381.

[16] Rutten, J.J.M.M., Weighted colimits and formal balls in generalized metric spaces, Topology and its Applications **89** (1998), pp.179-202.

[17] Smyth, M. B., Quasi-Uniformities: Reconciling Domains with Metric Space, Mathematical Foundations of Programming Language Semantics, Lecture Notes in Computer Science, Springer-Verlag 1987.

[18] Wagner, K., Solving Recursive Domain Equations With Enriched Categories, Ph.D Thesis, Carnegie Mellon University 1994.

[19] Wagner, K., Liminf convergence in Ω-categories, Theoretical Computer Science, to appear.

[20] Zheng, Chongyou, L. Fan, H. B. Cui, "Frame and Continuous Lattices" (2nd edition), Capital Normal University Press, Beijing 2000. (In Chinese)

MFPS 17 Preliminary Version

A Concurrent Graph Semantics For Mobile Ambients

Fabio Gadducci, Ugo Montanari [1]

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, Pisa, Italy
Email: {gadducci,ugo}@di.unipi.it

Abstract

We present an encoding for finite processes of the mobile ambients calculus into term graphs, proving its soundness and completeness with respect to the original, interleaving operational semantics. With respect to most of the other approaches for the graphical implementation of calculi with name mobility, our term graphs are unstructured (that is, non hierarchical), thus avoiding any "encapsulation" of processes. The implication is twofold. First of all, it allows for the reuse of standard graph rewriting theory and tools for simulating the reduction semantics. More importantly, it allows for the simultaneous execution of independent reductions, which are nested inside ambients, thus offering a concurrent semantics for the calculus.

Key words: concurrent graph rewriting, graphical encoding of process calculi, mobile ambients, reduction semantics.

1 Introduction

After the development of so-called optimal implementation of λ-calculus, many authors proposed graphical presentation for calculi with name mobility, in particular for the π-calculus [24]. These proposals usually introduce a syntactical notation for graphs, then they map processes into graphs *via* that notation. With a few exceptions [13,27], the resulting graphical structures are eminently hierarchical (that is, roughly, each node/edge/label is itself a structured entity, and possibly a graph), thus forcing the development of ad-hoc mechanisms for graph rewriting, in order to simulate process reduction.

[1] Research partly supported by the EC TMR Network *General Theory of Graph Transformation Systems* (GETGRATS); by the EC Esprit WG *Applications of Graph Transformations* (APPLIGRAPH); and by the Italian MURST Project *Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi* (TOSCA).

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In this paper we present instead a general proposal for mapping processes of calculi with name mobility into unstructured, non-hierarchical graphs. As the main example we chose mobile ambients [6], partly for its rising popularity in the community, while still lacking an analysis of its concurrency features; and partly because the complex name handling presented by its reduction rules highlights the power of our framework.

In fact, we believe that the intuitive appeal of non-hierarchical graphs, and the local nature of the associated rewriting mechanism, may help cast some light on the distributed features of the calculus. To this end, our first step is to prove the soundness and correctness of our encoding of processes into graphs, in the sense that two processes are structurally equivalent if and only if the corresponding graphs are isomorphic. Our second step is to prove that the encoding is faithful with respect to the reduction semantics, in the sense that standard graph rewriting techniques may now be used to simulate reduction steps on processes by sequences of rewrites on their encodings.

One of the additional advantages of formulating the reduction semantics of mobile ambients in terms of graph rewriting is the existence of a well-developed concurrent semantics [1], which extends the concurrent semantics of Petri nets and which allows to derive graph processes, event structures and prime algebraic domains from graph transformation systems. A concurrent semantics puts an upper limit to the amount of parallelism that is intrinsic in the reductions, and moreover it allows to derive causality links between reduction steps, which can be useful in better understanding the behaviour of a process, e.g. with respect to security and non-interference.

The paper has the following structure: In Section 2 we recall the mobile ambients calculus, and we discuss two alternative reduction semantics. In Section 3 we introduce a set-theoretical presentation for (ranked term) graphs, and we define two operations on them, namely *sequential* and *parallel composition* [7,8]. These operations are used in Section 4 to formulate our encoding for processes of the mobile ambient calculus, which is then proved to be sound and complete with respect to structural congruence. Finally, in Section 5 we recall the basic tools of graph rewriting, according to the DPO approach, and we show how four simple graph rewriting rules allow for simulating the reduction semantics of the mobile ambients calculus. We then argue how the information on causal dependencies between rewriting steps offered by the concurrent semantics of graph rewriting may be used for detecting *interferences* among process reductions, according to the taxonomy proposed in [22]. We close the paper with a few remarks, concerning the relevance of mapping processes into unstructured graphs from the point of view of parallelism; the generality of the approach, and its relationship with ongoing work on the graphical presentation of algebraic formalisms; and finally, the way to extend our results, in order to handle recursive processes.

$$P = Q \quad \text{for } P, Q \ \alpha\text{-convertible};$$
$$P \mid Q = Q \mid P, \qquad P \mid (Q \mid R) = (P \mid Q) \mid R, \qquad P \mid \mathtt{0} = P;$$
$$(\nu n)(\nu m)P = (\nu m)(\nu n)P \qquad (\nu n)(P \mid Q) = P \mid (\nu n)Q \quad \text{for } n \notin fn(P).$$
$$(\nu n)m[P] = m[(\nu n)P] \quad \text{for } n \neq m$$

Fig. 1. The set of axioms without deadlock detection

$$(\nu n)\mathtt{0} = \mathtt{0}$$

Fig. 2. The additional axiom for deadlock detection

2 Structural congruences for mobile ambients

This section shortly introduces the finite, communication-free fragment of the mobile ambients calculus, its structural equivalence and the associated reduction semantics. In addition, we describe two alternative structural equivalences for the calculus, proving that the associated reduction semantics are in fact "coincident", in a way to be made precise later on, to the original semantics.

2.1 The original calculus

Definition 2.1 (processes) *Let $\mathcal{N}$ be a set of atomic* names*, ranged over by $m, n, o, \ldots$. A* process *is a term generated by the following syntax*

$$P ::= \mathtt{0},\ n[P],\ M.P,\ (\nu n)P,\ P_1 \mid P_2$$

for the set of capabilities

$$M ::= in\, n,\ out\, n,\ open\, n.$$

We let $P, Q, R, \ldots$ range over the set Proc of processes.

We assume the standard definitions for the set of free names of a process P, denoted by $fn(P)$. Similarly for α-convertibility, with respect to the *restriction* operators (νn). Using these definitions, the dynamic behaviour of a process P is described as a relation over *abstract processes*, i.e., a relation obtained by closing a set of basic rules under structural congruence.

Definition 2.2 (reduction semantics) *The* reduction relation *for processes is the relation $R_m \subseteq Proc \times Proc$, closed under the structural congruence $\cong$ induced by the set of axioms in Figure 1 and Figure 2, inductively generated by the following set of axioms and inference rules*

$$\frac{}{m[n[out\, m.P \mid Q] \mid R] \to n[P \mid Q] \mid m[R]}$$

$$\frac{}{n[in\, m.P \mid Q] \mid m[R] \to m[n[P \mid Q] \mid R]} \qquad \frac{}{open\, n.P \mid n[Q] \to P \mid Q}$$

$$\frac{P \to Q}{(\nu n)P \to (\nu n)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \to Q}{n[P] \to n[Q]}$$

where $P \to Q$ means that $\langle P, Q \rangle \in R_m$.

$$(\nu n)M.P = M.(\nu n)P \quad \text{for } n \notin fn(M)$$

Fig. 3. The additional axiom for capability floating

2.2 Two alternative structural congruences

An important novelty in calculi with name mobility is the use of structural congruence for presenting the reduction semantics. This is intuitively appealing, since *abstract* processes allows for a simple representation (that is, modulo a suitable equivalence) of the spatial distribution of a system. Many equivalences, though, may be taken into account. Let us denote respectively as $P \rightarrow_d Q$ the reduction relation obtained by closing the inference rules presented in Definition 2.2 with respect to the structural congruence, denoted by $\cong_d$, induced by the set of axioms in Figure 1; and by $P \rightarrow_f Q$ the reduction relation obtained by closing the inference rules presented in Definition 2.2 with respect to the structural congruence, denoted by $\cong_f$, induced by the set of axioms in Figure 1 and Figure 3.

The first equivalence $\cong_d$ is finer than $\cong$, since it just forbids the identification of the deadlocked processes $\mathtt{0}$ and $(\nu n)\mathtt{0}$. Nevertheless, the mapping from abstract processes according to $\cong_d$, into abstract processes according to $\cong$, faithfully preserves the reduction semantics, as stated by next theorem.

Proposition 2.3 (deadlock and reductions) *Let P, Q be processes.* (1) *If $P \rightarrow_d Q$, then $P \rightarrow Q$. Vice versa,* (2) *if $P \rightarrow Q$, then there exists a process R such that $P \rightarrow_d R$ and $Q \cong R$.*

In other terms, the mapping does not add reductions. Sometimes, these kinds of mapping are also called *transition preserving morphisms* [11], a special form of the general notion of *open map* [18]. A similar property is satisfied by the mapping from abstract processes according to $\cong_d$, into abstract processes according to $\cong_f$, adding the distributivity of restriction with respect to capability (that is, letting the restrictions float to the top of a term).

Proposition 2.4 (distributivity and reductions) *Let P, Q be processes.* (1) *If $P \rightarrow_d Q$, then $P \rightarrow_f Q$. Vice versa,* (2) *if $P \rightarrow_f Q$, then there exists a process R such that $P \rightarrow_d R$ and $Q \cong_f R$.*

Our main theorem will present an alternative characterization of the relation $\rightarrow_f$ by means of graph rewriting techniques.

3 Graphs and term graphs

We open the section recalling the definition of (ranked) term graphs: We refer to [5,7] for a detailed introduction, as well as for a comparison with standard definitions such as [3]. In particular, we assume in the following a chosen signature (Σ, S), for Σ a set of operators, and S a set of sorts, such that the *arity* of an operator in Σ is a pair (ω_s, ω_t), for ω_s, ω_t strings in S^*.

Definition 3.1 (graphs) *A* labelled graph *d (over (Σ, S)) is a five tuple $d = \langle N, E, l, s, t\rangle$, where N, E are the sets of* nodes *and* edges*; l is the pair of* labeling *functions $l_e : E \to \Sigma$, $l_n : N \to S$; $s, t : E \to N^*$ are the* source *and* target *functions; and such that for each edge $e \in dom(l)$, the arity of $l_e(e)$ is $(l_n^*(s(e)), l_n^*(t(e)))$, i.e., each edge preserves the arity of its label.*

With an abuse of notation, in the definition above we let l_n^* denote the extension of the function l_n from nodes to strings of nodes. Moreover, we denote the components of a graph d by N_d, E_d, l_d, s_d and t_d.

Definition 3.2 (graph morphisms) *Let d, d' be graphs. A* (graph) morphism *$f : d \to d'$ is a pair of functions $f_n : N_d \to N_{d'}$, $f_e : E_d \to E_{d'}$ that preserves the labeling, source and target functions.*

In order to inductively define an encoding for processes, we need to define some operations over graphs. The first step is to equip them with suitable "handles" for interacting with an environment, built out of other graphs.

Definition 3.3 ((ranked) term graphs) *Let d_r, d_v be graphs with no edges. A (d_r, d_v)-*ranked graph *(a* graph of rank *(d_r, d_v)) is a triple $g = \langle r, d, v\rangle$, for d a graph and $r : d_r \to d$, $v : d_v \to d$ the injective* root *and* variable *morphisms.*

Let g, g' be ranked graphs of the same rank. A ranked graph morphism *$f : g \to g'$ is a graph morphism $f_d : d \to d'$ between the underlying graphs that preserves the root and variable morphisms.*

Two graphs $g = \langle r, d, v\rangle$ and $g' = \langle r', d', v'\rangle$ of the same rank are isomorphic *if there exists a* ranked graph isomorphism *$\phi : g \to g'$. A (d_r, d_v)-ranked term graph G is an isomorphism class of (d_r, d_v)-ranked graphs.*

With an abuse of notation, we sometimes refer to the nodes in the image of the variable (root) morphism as variables (roots, respectively). Moreover, we often use the same symbols of ranked graphs to denote term graphs, so that e.g. $G_{d_v}^{d_r}$ denotes a term graph of rank (d_r, d_v).

Definition 3.4 (sequential and parallel composition) *Let $G_{d_v}^{d_i}$, $H_{d_i}^{d_r}$ be term graphs. Their* sequential composition *is the term graph $G_{d_v}^{d_i}; H_{d_i}^{d_r}$ of rank (d_r, d_v) obtained by first the disjoint union of the graphs underlying G and H, and second the gluing of the roots of G with the corresponding variables of H.*

Let $G_{d_v}^{d_r}$, $H_{d'_v}^{d'_r}$ be term graphs, such that $d_v \cap d'_v = \emptyset$. Their parallel composition *is the term graph $G_{d_v}^{d_r} \otimes H_{d'_v}^{d'_r}$ of rank $(d_r \cup d'_r, d_v \cup d'_v)$ obtained by first the disjoint union of the graphs underlying G and H, and second the gluing of the roots of G with the corresponding roots of H.* [2]

[2] Let $G_{d_v}^{d_i} = \langle r, d, v\rangle$ and $H_{d_i}^{d_r} = \langle r', d', v'\rangle$ be term graphs. Then, $G; H = \langle r'', d'', v''\rangle$, for d'' the disjoint union of d and d', modulo the equivalence on nodes induced by $r(x) = v'(x)$ for all $x \in N_{d_i}$, and $r'' : d_r \to d''$, $v'' : d_v \to d''$ the uniquely induced arrows. Let now $G_{d_v}^{d_r} = \langle r, d, v\rangle$ and $H_{d'_v}^{d'_r} = \langle r', d', v'\rangle$ be term graphs. Then, $G \otimes H = \langle r'', d'', v''\rangle$, for d'' the disjoint union of d and d', modulo the equivalence on nodes induced by $r(x) = r'(x)$ for all $x \in N_{d_r} \cap N_{d'_r}$, and $r'' : d_r \cup d'_r \to d''$, $v'' : d_v \cup d'_v \to d''$ the uniquely induced arrows.

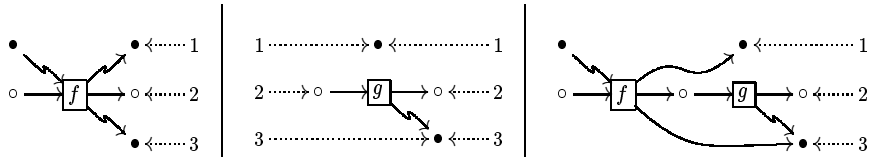


Fig. 4. Two term graphs, and their sequential composition

Note that the two operations are defined on "concrete" graphs. Nevertheless, the result is clearly independent of the choice of the representative, and it implies that both parallel and sequential composition are associative.

Example 3.5 (sequential composition) *Let us consider the signature* (Σ_e, S_e)*, for* $S_e = \{s_1, s_2\}$ *and* $\Sigma_e = \{f : s_1 s_2 \to s_1 s_2 s_1, g : s_2 \to s_2 s_1\}$*. Two term graphs, built out of the signature* (Σ_e, S_e)*, are shown in Figure 4. The nodes in the domain of the root (variable) morphism are depicted as a vertical sequence on the right (left, respectively); edges are represented by their label, from where arrows pointing to the target nodes leave, and to where the arrows from the source node arrive. The root and variable morphisms are represented by dotted arrows, directed from right-to-left and left-to-right, respectively.*

The term graph on the left has rank $(\{1,2,3\}, \emptyset)$*, five nodes and one edge (labelled by* f*); the term graph on the middle has rank* $(\{1,2,3\}, \{1,2,3\})$*, four nodes and one edge (labelled by* g*). For graphical convenience, in the underlying graph the nodes of sort* s_1 *are denoted by* $\bullet$*, those of sort* s_2 *by* $\circ$*.*

Sequential composition of term graphs is performed by matching the roots of the first graph with the variables of the second one, as shown by the term graph on the right: It has rank $(\{1,2,3)\}, \emptyset)$*, six nodes and two edges, and it is obtained by sequentially composing the other two.*

A *(term graph) expression* is a term over the signature containing all ranked term graphs as constants, and parallel and sequential composition as binary operators. An expression is *well-formed* if all occurrences of both parallel and sequential composition are defined for the rank of the argument subexpressions, according to Definition 3.4; the *rank* of an expression is then computed inductively from the rank of the term graphs appearing in it, and its *value* is the term graph obtained by evaluating all operators in it.

4 Channels as wires: from processes to term graphs

The first step in our implementation is to encode processes into term graphs, built out of a suitable signature (Σ_m, S_m), and proving that the encoding preserves structural convertibility. Then, standard graph rewriting techniques are used for simulating the reduction mechanism.

The set of sorts S_m contains the elements s_p and s_a. The first symbol is reminiscent of the word *process*, since the elements of sort s_p can be considered as processes reached by a transition. The second sort, s_a, is reminiscent of *ambient*, and the elements of this sort correspond to names of the calculus.

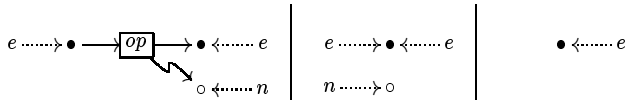


Fig. 5. Term graphs op_n (for $op \in \{amb, in, open, out\}$), ν_n and $\mathtt{0}$.

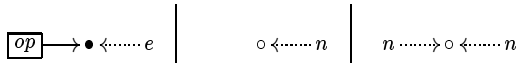


Fig. 6. Term graphs op (for $op \in \{go, idle\}$), new_n e id_n.

The operators are $\{in : s_p \to s_p s_a, out : s_p \to s_p s_a, open : s_p \to s_p s_a\} \cup \{amb : s_p \to s_p s_a\} \cup \{go : \lambda \to s_p, idle : \lambda \to s_p\}$. The elements of the first set simulate the capabilities of the calculus; the *amb* operator simulates ambients. Note that there is no operator for simulating name restriction; instead, the operators *go* and *idle* are syntactical devices for detecting the status of those nodes in the source of an edge labeled *amb*, thus avoiding to perform any reduction below the outermost capability operator, as shown in Section 5.

The second step is the characterization of a class of graphs, such that all processes can be encoded into an expression containing only those graphs as constants, and parallel and sequential composition as binary operators. Thus, let us consider a name $e \notin \mathcal{N}$: Our choice is depicted in Figure 5 and Figure 6.

Definition 4.1 (encoding for processes) *Let P be a process, and let Γ be a set of names, such that $fn(P) \subseteq \Gamma$. The* encoding $[\![P]\!]^{go}_{\Gamma}$ *maps a process P into a term graph, as defined below by structural induction,*

$$
\begin{aligned}
[\![P]\!]^{go}_{\Gamma} &= [\![P]\!]_{\Gamma} \otimes go \\
[\![\mathtt{0}]\!]_{\Gamma} &= \mathtt{0} \otimes (\textstyle\bigotimes_{o \in \Gamma} new_o) \\
[\![n[P]]\!]_{\Gamma} &= ([\![P]\!]_{\Gamma} \otimes idle); (amb_n \otimes (\textstyle\bigotimes_{o \in \Gamma} id_o)) \\
[\![M.P]\!]_{\Gamma} &= [\![P]\!]_{\Gamma}; (M_n \otimes (\textstyle\bigotimes_{o \in \Gamma} id_o)) \text{ for } M \text{ capability with } fn(M) = \{n\} \\
[\![(\nu n)P]\!]_{\Gamma} &= [\![P\{^m/_n\}]\!]_{\{m\} \cup \Gamma}; (\nu_m \otimes (\textstyle\bigotimes_{o \in \Gamma} id_o)) \text{ for name } m \notin \Gamma \\
[\![P \mid Q]\!]_{\Gamma} &= [\![P]\!]_{\Gamma} \otimes [\![Q]\!]_{\Gamma}
\end{aligned}
$$

where we assume the standard definition for name substitution.

Thus, the mapping prefixes the term graph $[\![P]\!]_{\Gamma}$ with the occurrence of a "ready" tag, the *go* operator: It will denote an activating point for reduction.

The mapping is well-defined, in the sense that the result is independent of the choice of the name m in the last rule; moreover, given a set of names Γ, the encoding $[\![P]\!]^{go}_{\Gamma}$ of a process P is a term graph of rank $(\{e\} \cup \Gamma, \emptyset)$.

Example 4.2 (a graphical view of firewalls) *We present the implementation of a firewall access, as proposed by Cardelli and Gordon [6]. First, some graphical conventions. The encoding of a process P is a term graph $G = [\![P]\!]_{\{k\}}$*

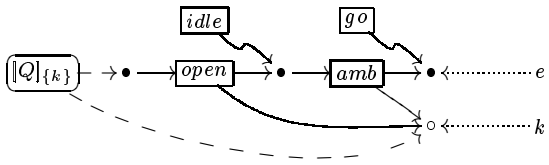


Fig. 7. Term graph for $Agent(Q) = k[open\, k.Q]$.

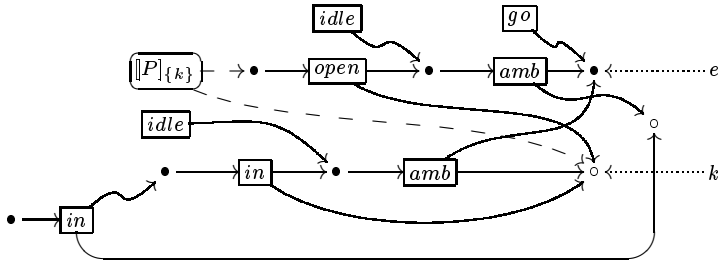


Fig. 8. Term graph for $Firewall(P) = (\nu w)(w[open\, k.P] \mid k[in\, k.in\, w.\mathsf{0}])$.

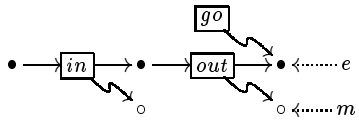


Fig. 9. Term graph encoding for both $(\nu n)out\, m.in\, n.\mathsf{0}$ and $out\, m.(\nu n)in\, n.\mathsf{0}$.

of rank $(\{e, k\}, \emptyset)$*: We represent it by circling the expression, from where two dashed arrows leave, directed to the roots of* G *(hence, to the nodes of* G *pointed by* e *and* k*, respectively). The term graph* $[\![k[open\, k.Q]]\!]^{go}_{\{k\}}$ *is shown in Figure 7.*

The process $(\nu w)(w[open\, k.P] \mid k[in\, k.in\, w.\mathsf{0}])$*, simulating a firewall, is instead implemented by the ranked term graph in Figure 8.*

The mapping $[\![-]\!]^{go}_{\Gamma}$ is not surjective, because there are term graphs of rank $(\{e\} \cup \Gamma, \emptyset)$ that are not the image of any process; nevertheless, our encoding is sound and complete, as stated by the proposition below.

Proposition 4.3 *Let* P*,* Q *be processes, and let* Γ *be a set of names, such that* $fn(P) \cup fn(Q) \subseteq \Gamma$*. Then,* $P \cong_f Q$ *if and only if* $[\![P]\!]^{go}_{\Gamma} = [\![Q]\!]^{go}_{\Gamma}$*.*

Our encoding is thus sound and complete with respect to equivalence $\cong_f$. It is easy to see e.g. that the processes $(\nu n)out\, m.in\, n.\mathsf{0}$ and $out\, m.(\nu n)in\, n.\mathsf{0}$, for $n \neq m$, are mapped to the same term graph, represented in Figure 9.

5 Reductions as graph rewrites

We open the section recalling the basic tools of the double-pushout (DPO) approach to graph rewriting, as presented in [9,10], and introducing a mild generalization of its well-understood *process semantics* [1]. We then provide a graph rewriting system $\mathcal{R}_m$ for modeling the reduction semantics of mobile ambients. Finally, we discuss the concurrent features of the rewriting system $\mathcal{R}_m$, as captured by the process semantics, arguing that they enhance the analysis of the causal dependencies among the possible reductions performed by a

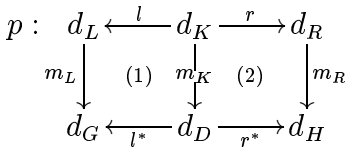


Fig. 10. A DPO direct derivation

mobile ambient process, with respect to the original interleaving semantics.

5.1 Tools of DPO graph rewriting

Definition 5.1 (graph production and derivation) *A* graph production $p : \sigma$ *is composed of a* production name p *and of a span of graph morphisms* $\sigma = (d_L \xleftarrow{l} d_K \xrightarrow{r} d_R)$. *A* graph transformation system *(or* GTS*)* $\mathcal{G}$ *is a set of productions, all with different names. Thus, when appropriate, we denote a production* $p : \sigma$ *using only its name* p.

A graph production $p : (d_L \xleftarrow{l} d_K \xrightarrow{r} d_R)$ *is* injective *if* l *is injective. A graph transformation system* $\mathcal{G}$ *is* injective *if all its productions are so.*

A double-pushout *diagram is like the diagram depicted in Figure 10, where top and bottom are spans and (1) and (2) are pushout squares in the category* $\mathbf{G}_{\Sigma,S}$ *of graphs and graph morphisms (over the signature* (Σ, S)*). Given a production* $p : (d_L \xleftarrow{l} d_K \xrightarrow{r} d_R)$*, a* direct derivation *from* d_G *to* d_H *via production* p *and triple* $m = \langle m_L, m_K, m_R \rangle$ *is denoted by* $d_G \overset{p/m}{\Longrightarrow} d_H$.
A derivation *(of length* n*)* ρ *in a* GTS $\mathcal{G}$ *is a finite sequence of direct derivations* $d_{G_0} \overset{p_1/m_1}{\Longrightarrow} \ldots \overset{p_n/m_n}{\Longrightarrow} d_{G_n}$ *where* $p_1, \ldots, p_n$ *are productions of* $\mathcal{G}$.

Operationally, the application of a production p to a graph d_G consists of three steps. First, the *match* $m_L : d_L \to d_G$ is chosen, providing an occurrence of d_L in d_G. Then, all objects of G matched by $d_L - l(d_K)$ are removed, leading to the *context graph* d_D. Finally, the objects of $d_R - r(d_K)$ are added to d_D, obtaining the derived graph d_H.

The role of the *interface* graph d_K in a rule is to characterize the elements of the graph to be rewritten that are read but not consumed by a direct derivation. Such a distinction is important when considering *concurrent* derivations, possibly defined as an equivalence class of concrete derivations up-to so-called *shift equivalence* [9], identifying (as for the analogous, better-known *permutation equivalence* of λ-calculus) those derivations which differ only for the scheduling of independent steps. Roughly, the equivalence states the interchangeability of two direct derivations $d_1 \Longrightarrow d_2 \Longrightarrow d_3$ if they act either on disjoint parts of d_1, or on parts that are in the image of the interface graphs.

A more concrete, yet equivalent notion of abstract derivation for a GTS is obtained by means of the so-called *process semantics*. As for the similar notion on Petri nets [15], a graph process represents a description for a derivation that abstracts from the ordering of causally unrelated steps (as it is the case for shift

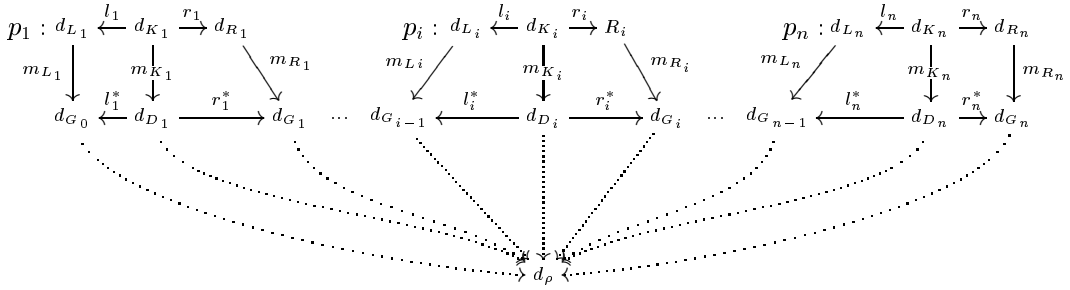


Fig. 11. Colimit construction for derivation $\rho = d_{G_0} \overset{p_1/m_1}{\Longrightarrow} \ldots \overset{p_n/m_n}{\Longrightarrow} d_{G_n}$

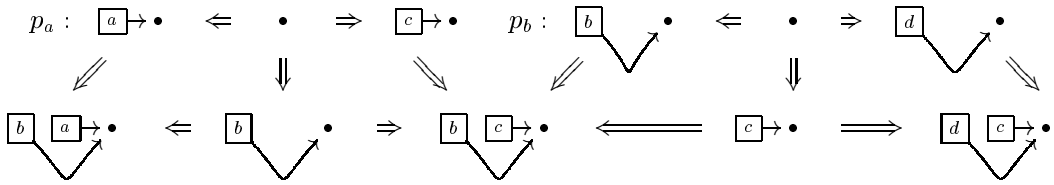


Fig. 12. The derivation $\rho_{ex} = d_{G_0} \overset{p_a/m_a}{\Longrightarrow} d_{G_a} \overset{p_b/m_b}{\Longrightarrow} d_{G_b}$

equivalence), and that offers at the same time a concrete representative for a class of equivalent derivations. The definition below slightly generalizes [1].

Definition 5.2 (graph processes) *Let $\mathcal{G}$ be an injective* GTS*, and let ρ be a derivation $d_{G_0} \overset{p_1/m_1}{\Longrightarrow} \ldots \overset{p_n/m_n}{\Longrightarrow} d_{G_n}$ of length n (upper part of Figure 11). The (graph) process $\Pi(\rho)$ associated to the derivation ρ is the $n+1$-tuple $\langle t_{G_0}, \langle p_1, \pi_1 \rangle, \ldots, \langle p_n, \pi_n \rangle\rangle$: Each π_i is a triple $\langle t_{L_i}, t_{K_i}, t_{R_i} \rangle$, and the graph morphisms $t_{x_i} : d_{x_i} \to d_\rho$, for $x_i \in \{L_i, K_i, R_i\}$ and $i = 1, \ldots, n$, are those uniquely induced by the colimit construction shown in Figure 11.*

Let ρ, ρ' be two derivations of length n, both originating from graph d_{G_0}. They are process equivalent *if the associated graph processes are* isomorphic*, i.e., if there exists a graph isomorphism $\gamma_\pi : d_\rho \to d_{\rho'}$ and a bijective function $\gamma_p : \{1, \ldots, n\} \to \{1, \ldots, n\}$, such that productions p_i and $p'_{\gamma_p(i)}$ coincide for all $i = 1, \ldots, n$, and all the involved diagrams commute.* [3]

A graph process associated to a derivation ρ thus includes, by means of the colimit construction and of the morphisms t_{x_i}, the action of each single production p_i on the graph d_ρ. ¿From the image of each d_{x_i} is then possible to recover a suitable partial order among the direct derivations in ρ, which faithfully mirrors the causal relationship among them. For example, let (Σ_{ex}, S_{ex}) be the one-sorted signature containing just four constants, namely $\{a, b, c, d\}$; and let $\mathcal{G}_{ex}$ be the GTS containing two rules, roughly rewriting a into c and b into d. The derivation ρ_{ex} is represented in Figure 12, where, for the sake of readability, graph morphisms are simply depicted as thick arrows.

[3] Explicitly, $\gamma_\pi \circ t_{G_0} = t'_{G_0}$, and $\gamma_\pi \circ t_{x_i} = t'_{x_{\gamma_p(i)}}$ for $x_i \in \{L_i, K_i, R_i\}$ and $i = 1, \ldots, n$.

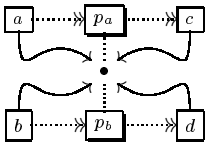


Fig. 13. Compact representation for the process $\Pi(\rho_{ex})$

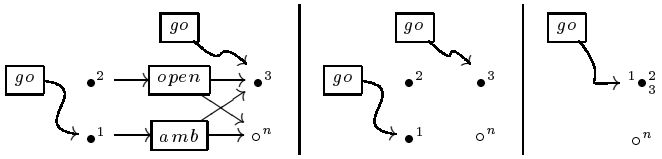


Fig. 14. The rewriting rule for $open\, n.P \mid n[Q] \to P \mid Q$

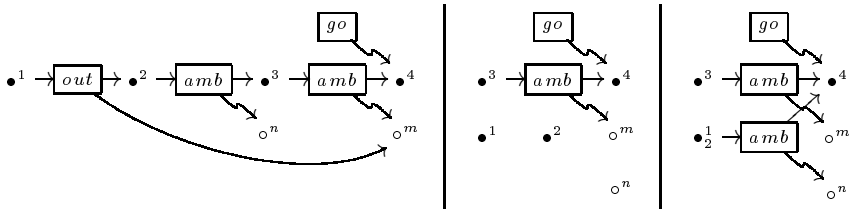


Fig. 15. The rewriting rule for $m[n[out\, m.P \mid Q] \mid R] \to m[R] \mid n[P \mid Q]$

The process $\Pi(\rho_{ex})$ can be described as in Figure 13, extending the graph $d_{\rho_{ex}}$ with two shaded boxes: They are labelled p_a and p_b, in order to make explicit the mappings t_{x_i} (hence, the action of the rules on the initial graph). Thus, (the application of) the production p_a consumes the a edge (it is in the image of t_{L_a}, but not in the image of t_{K_a}), and this is denoted by the dotted arrow from a into p_a; it then reads the only node (which is indeed in the image of t_{K_a}), denoted by the dotted arrow with no head; and finally, it creates the c edge, denoted by the dotted arrow into c. Similarly, (the application of) the production p_b consumes the b edge, reads the node and creates the d edge.

We feel confident that our example underlines the connection between the process semantics for graphs, and the standard process semantics for Petri nets. This compact representation is further argued upon on Section 5.3.

5.2 A graph rewriting system for ambients

We finally introduce in this section the graph rewriting system $\mathcal{R}_m$. We first discuss informally its set of productions, then stating more precisely how its rewrites simulate the operational behaviour of processes.

The rule $p_{open} : (d_{Lo} \xleftarrow{l_o} d_{Ko} \xrightarrow{r_o} d_{Ro})$ for synchronizing an *open* edge with a relevant ambient occurrence is presented in Figure 14: the graph on the left-hand side (center, right-hand side) is d_{Lo} (d_{Ko} and d_{Ro}, respectively); the action of the rule (that is, the span of graph morphisms) is intuitively described by the node identifiers. Both *amb* and *open* edges disappear after

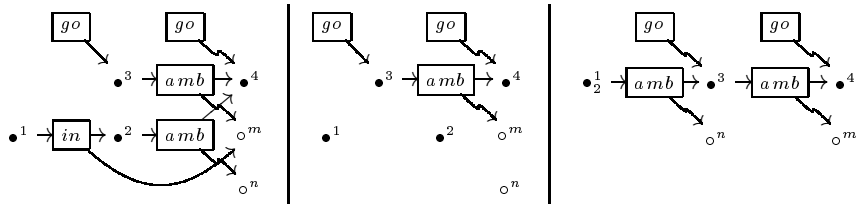


Fig. 16. The rewriting rule for $m[P] \mid n[in\, m.Q \mid R] \to m[n[Q \mid R] \mid P]$

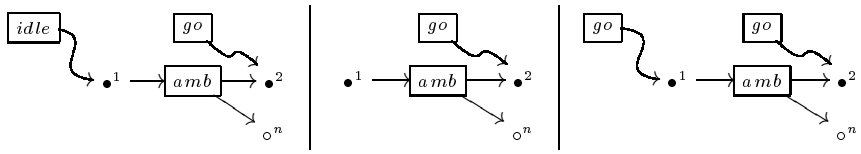


Fig. 17. The rewriting rule for *broadcasting*

reduction, and all the connected nodes are coalesced. Notice that the reduction cannot happen unless both the node shared in the synchronization and the node under the *amb* prefix are activated, i.e., are labelled by the *go* mark. After reduction, also the node under the *open* prefix becomes activated. The occurrence of the nodes in the interface graph allows for applying the rule in every possible context. Similarly, the occurrence of the *go* operators allows for the simultaneous execution of other derivations using these "tags", since the "read" politics for edges in the interface implies that e.g. more than one pair of distinct resources may synchronize at the top level.

Let us consider now the rules p_{out} and p_{in}, for simulating the *out* and *in* reductions of the calculus, presented in Figure 15 and Figure 16. As for the p_{open} rule, the action of the two productions is described by the node identifiers. It is relevant that the ambients linked with identifier n are first consumed and then re-created by the rules, as they do not belong to the interface graphs. On the contrary, the ambients linked with identifier m are just read, and this implies that e.g. more than one reduction may act simultaneously on that ambient: This fact will be further confirmed when discussing the process semantics for the GTS $\mathcal{R}_m$ in Section 5.3.

Finally, let p_{broad} be the rule in Figure 17. It has no correspondence in the reduction semantics, and its purpose is broadcasting the activation mark to a tree of ambients, whenever its root becomes activated. An occurrence of the *go* operator, denoting an activating point for the process reduction, permeates into the external ambient, reaching the internal node labelled by identifier 1. Of course, the propagation cannot proceed when a capability prefix is reached.

Let the expression $d_G \Longrightarrow_b^* d_H$ denote that d_H is obtained by a finite number of applications of the broadcasting rule p_{broad} to d_G. We can finally state the main theorems of the paper, concerning the soundness and completeness of our encoding with respect to the reduction semantics.

Theorem 5.3 (encoding preserves reductions) *Let P, Q be processes,*

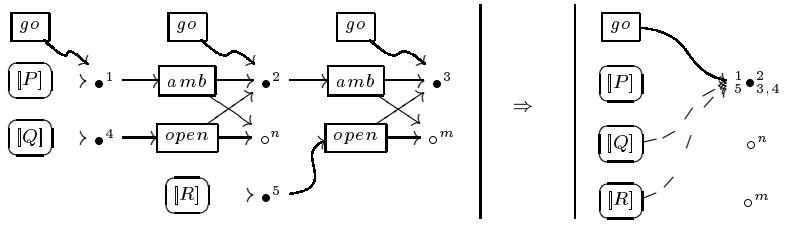


Fig. 18. Simultaneous application of nested, yet causally unrelated reductions

and let Γ *be a set of names such that* $fn(P) \subseteq \Gamma$. *If the reduction* $P \to_f Q$ *is entailed, then* $\mathcal{R}_m$ *entails a derivation* $\{\!|P|\!\}_\Gamma \Longrightarrow_b^* d_G \Longrightarrow d_H$, *such that* $\{\!|Q|\!\}_\Gamma \Longrightarrow_b^* d_H$.

Intuitively, process reduction is simulated by first applying a sequence of broadcasting rules, thus enabling (by the propagation of the *go* operator) those events whose activating point is nested inside one or more ambients, and then simulating the actual reduction step. The mapping $\{\!|P|\!\}_\Gamma$ introduced in the statement of the theorem denotes the graph (that is, a representative of the equivalence class of isomorphic graphs) underlying the term graph $[\![P]\!]_\Gamma^{go}$.

Theorem 5.4 (encoding does not add reductions) *Let* P *be a process, and let* Γ *be a set of names such that* $fn(P) \subseteq \Gamma$. *If* $\mathcal{R}_m$ *entails a derivation* $\{\!|P|\!\}_\Gamma \Longrightarrow_b^* d_G \Longrightarrow d_H$, *then there exists a process* Q *such that* $P \to_f Q$ *is entailed and* $\{\!|Q|\!\}_\Gamma \Longrightarrow_b^* d_H$.

5.3 On causal dependency and simultaneous execution

We argued in the Introduction that the concurrent semantics of GTS's may shed some light in the understanding of process behaviour for mobile ambients.

It is in fact an obvious consideration that by our encoding we can equip mobile ambients with a concurrent semantics, simply considering for each process P of the calculus the classes of process equivalent derivations associated to the graph $\{\!|P|\!\}_{fn(P)}$. This is intuitively confirmed by the analysis of a rather simple process, namely, $S = m[n[P] \mid open\, n.Q] \mid open\, m.R$. The process S may obviously perform two reductions, opening either the ambient m, or the ambient n: These reductions should be considered as independent, since they act on nested, yet causally unrelated occurrences of an ambient. This independence becomes explicit in the graph d_S, obtained by applying twice the broadcasting rule to $\{\!|S|\!\}_{\{m,n\}}$, and depicted on the left-hand-side of Figure 18 (forgetting for the sake of clarity the subscripts and the dashed arrows leaving from the graphs underlying $[\![P]\!]_{\{m,n\}}$ and $[\![Q]\!]_{\{m,n\}}$ and directed to either m or n). Production p_{open} may now be applied twice, reducing either those edges linked with the node n, or those linked with the node m, thus simulating the reductions originating from S. These rewrites may be executed in any order, resulting in two different derivations, which are nevertheless process equivalent. The resulting graph is depicted on the right-hand side of Figure 18.

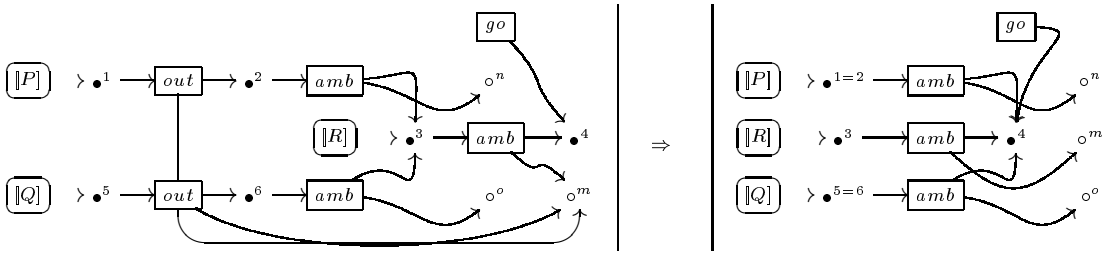


Fig. 19. Simultaneous application of nested reductions sharing an ambient

Let us consider now a more complex example, and let T be the process $m[n[out\, m.P] \mid o[out\, m.Q] \mid R]$, which can be reduced into $n[P] \mid m[R] \mid o[Q]$ by applying twice the *out* reduction on ambient m, and depicted in Figure 19. The two rules may be applied simultaneously, since the occurrence of the *amb* operator, linked to the node with identifier m, is shared. The process resulting from the colimit construction of Figure 11, if represented as in Figure 13, contains two events: The first one consumes the *out* edge linked with nodes 1, 2 and m, and the *amb* edge linked with nodes 2, 3 and n; reads the *amb* edge linked with nodes 3, 4 and m (and all the related nodes); and creates the *amb* edge linked with nodes $1 = 2$, 4 and n. Symmetrically, the other consumes the *out* edge linked with nodes 5, 6 and m, and the *amb* edge linked with nodes 6, 3 and o; reads the *amb* edge linked with nodes 3, 4 and m (and all the related nodes); and creates the *amb* edge linked with nodes $5 = 6$, 4 and o.

Let U be the process $m[n[out\, m.P] \mid open\, n.R]$. This is listed by Levi and Sangiorgi [22] as an example of *grave interference*, representing a situation in the calculus that should be deprecated, and actually "should be regarded as a programming error". The execution of the internal *out* reduction on the ambient m destroys the possibility to perform the execution of the external *open* reduction on the ambient n, and vice versa. This is confirmed by the analysis of the graph in the middle of Figure 20, obtained by applying twice the broadcasting rule to $\{\!|U|\!\}_\Gamma$. The two derivations originating from that graph, and simulating the execution of the two reductions, are represented on the right-hand-side (the internal *out*) and on the left-hand-side (the external *open*). These derivations can not be extended with additional steps, in order to become process equivalent. This situation is usually described by saying that the two derivations denote a *symmetric conflict* of events.

More interestingly, let us consider an apparently similar instance of grave interference, represented by the process $V = m[n[out\, m.P] \mid Q] \mid open\, m.R$. The external *open* reduction on ambient m destroys the possibility to perform the internal *out* reduction on the same ambient, but *the vice versa does not hold*. After the execution of the internal *out* reduction, an external *open* may be performed, and the two applications of p_{open} represent *the same event*. Since the occurrence of the *amb* operator is only read by p_{out} of Figure 15, the same operator is available after the rewriting step. We are thus facing an *asymmetric conflict*, lifting the notion from a recent extension of the event

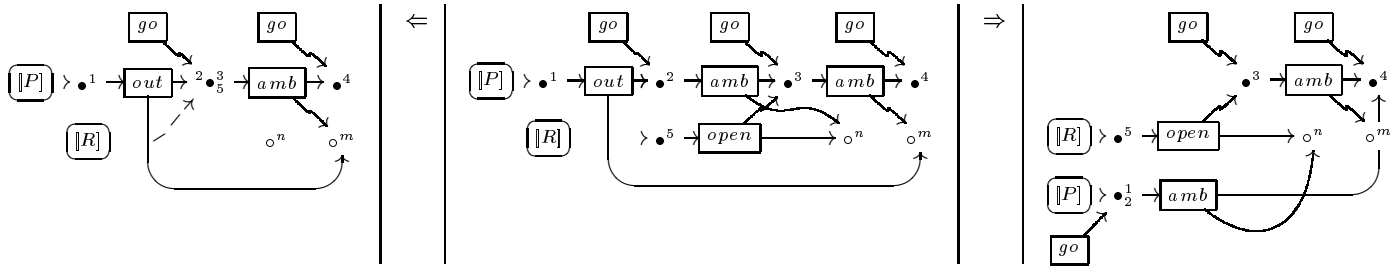


Fig. 20. Grave interference as symmetric conflict

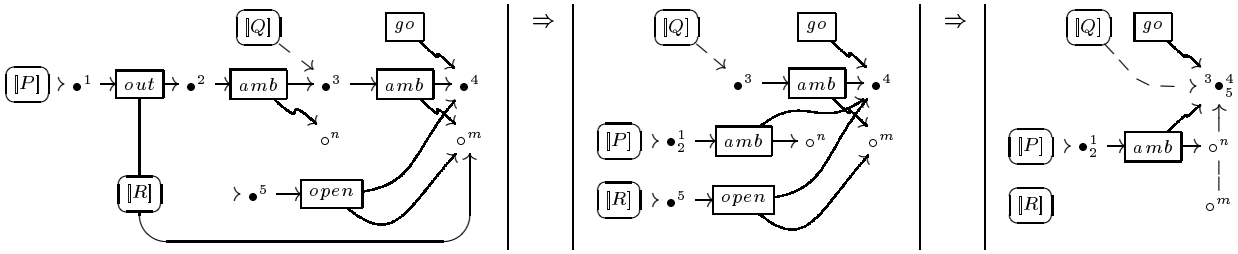


Fig. 21. Grave interference as asymmetric conflict

structures formalism [2]. The graph $\{\!|V|\!\}_{\{m,n\}}$ is represented on the left-hand side of Figure 21; the graphs obtained by first the application of p_{out}, and then of p_{open}, are represented on the center and on the right-hand side of the figure.

6 Conclusions and Further Works

We presented an encoding for finite, communication-free processes of the mobile ambients calculus into term graphs, proving its soundness and completeness with respect to the original, interleaving operational semantics.

With respect to most of the other approaches for the graphical implementation of calculi with name mobility (see e.g. Milner's π-*nets* [23], Parrow 's *interaction diagrams* [26], Gardner's *process frameworks* [14], Hasegawa's *sharing graphs* [16], Montanari and Pistore's presentation of π-calculus by DPO rules [25] or König SPIDER *calculus* [21]; an exception are Yoshida's *concurrent combinators* [27]), we considered unstructured (that is, non hierarchical) graphs, thus avoiding any "encapsulation" of processes. The implication is twofold. First of all, it allows the reuse of standard graph rewriting theory and tools for simulating the reduction semantics, such as e.g. the DPO formalism and the HOPS programming system [20]. More importantly, it allows for the simultaneous execution of independent reductions, which are nested inside ambients, and possibly share some resource. While this feature is less relevant for e.g. the π-calculus, where each process can be considered just a soup of disjoint sequential agents (much in the spirit of Berry's and Boudol's CHAM approach [4]), it is relevant in the present context, where ambients are nested, and yet can be "permeated" by a reduction. A first, rough analysis is per-

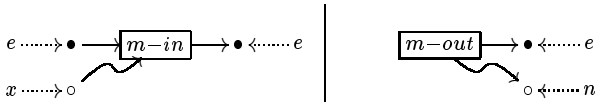


Fig. 22. Term graphs for input (x) and asynchronous output $\langle n \rangle$ actions.

formed in Section 5.3, and we plan to extend our preliminary considerations to a non-deterministic concurrent semantics for mobile ambients, much in the spirit of the event structure semantics developed in [1].

Our encoding can be extended to recover the communication primitives, as long as we restrict communication to name passing: The graphs for encoding input and asynchronous output actions are depicted in Figure 22. In fact, we feel confident that any calculus with name mobility may find a presentation within our formalism, along the line of the encoding for mobile ambients. The calculus should of course contain a parallel operator which is associative, commutative and with an identity; moreover, its operational semantics should be reduction-like (i.e., expressed by unlabelled transitions), and the rules should never substitute a free name for another, so that name substitution can be handled by node coalescing (with a mechanism reminiscent of *name fusion*).

It should be noted that any monoidal category with a suitable enrichment (namely, where each object a is equipped with two monoidal transformations $a \to a \times a$ and $1 \to a$, making it a *monoid*) could be used as a sound model for the encoding. The relevant thing is that, among this class of models, (a suitable sub-category of) the category $\mathbf{RG}_{\Sigma,S}$ of graphs as objects, and ranked graphs as morphisms, is the initial one [5,7], so that Proposition 4.3 is just a corollary of this general result. Our work is thus tightly linked with ongoing research on the graphical presentations for categorical formalisms, as e.g. on *premonoidal* [17] and *traced monoidal* [19] categories. More importantly, also graph processes may be equipped with an algebraic structure [8,12], thus providing a formalism for denoting also reductions in mobile ambients.

As for the finiteness conditions, it is a different matter. In fact, it is a difficult task to recover the behaviour of processes including a *replication* operator, since replication is a global operation, involving the duplication of necessarily unspecified sub-processes, and it is hence hard to model *via* graph rewriting, which is an eminently local process. Nevertheless, our framework allows for the modeling of *recursive* processes, that is, defined using constant invocation, so that a process is a family of judgments of the kind $A = P$. Thus, each process is compiled into a different graph transformation system, adding to the four basic rewriting rules a new production p_A for each constant A, intuitively simulating the unfolding step $\{\!|A|\!\}_\Gamma \Rightarrow \{\!|P|\!\}_\Gamma$, for a suitable Γ.

References

[1] P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In H. Ehrig, H.-

J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3, pages 107–187. World Scientific, 1999.

[2] P. Baldan, A. Corradini, and U. Montanari. An event structure semantics for P/T contextual nets: Asymmetric event structures. In M. Nivat, editor, *Foundations of Software Science and Computation Structures*, Lect. Notes in Comp. Science, pages 63–80. Springer Verlag, 1998. Revised version to appear in *Information and Computation*.

[3] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph reduction. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Parallel Architectures and Languages Europe*, volume 259 of *Lect. Notes in Comp. Science*, pages 141–158. Springer Verlag, 1987.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoret. Comput. Sci.*, 96:217–248, 1992.

[5] R. Bruni, F. Gadducci, and U. Montanari. Normal forms for algebras of connections. *Theoret. Comput. Sci.*, 2001. To appear. Available at `http://www.di.unipi.it/~ugo/tiles.html`.

[6] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lect. Notes in Comp. Science*, pages 140–155. Springer Verlag, 1998.

[7] A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7:299–331, 1999.

[8] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence between the operational and the categorical description. *Informatique Théorique et Applications/Theoretical Informatics and Applications*, 33:467–493, 1999.

[9] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

[10] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

[11] G. Ferrari and U. Montanari. Towards the unification of models for concurrency. In A. Arnold, editor, *Trees in Algebra and Programming*, volume 431 of *Lect. Notes in Comp. Science*, pages 162–176. Springer Verlag, 1990.

[12] F. Gadducci, R. Heckel, and M. Llabrés. A bi-categorical axiomatisation of concurrent graph rewriting. In M. Hofmann, D. Pavlovič, and G. Rosolini, editors, *Category Theory and Computer Science*, volume 29 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences, 1999. Available at `http://www.elsevier.nl/locate/entcs/volume29.html/`.

[13] F. Gadducci and U. Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoret. Comput. Sci.*, 2001. To appear. Available at `http://www.di.unipi.it/~ugo/tiles.html`.

[14] Ph. Gardner. From process calculi to process frameworks. In C. Palamidessi, editor, *Concurrency Theory*, volume 1877 of *Lect. Notes in Comp. Science*, pages 69–88. Springer Verlag, 2000.

[15] U. Golz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.

[16] M. Hasegawa. *Models of Sharing Graphs*. PhD thesis, University of Edinburgh, Department of Computer Science, 1997.

[17] A. Jeffrey. Premonoidal categories and a graphical view of programs. Technical report, School of Cognitive and Computing Sciences, University of Sussex, 1997. Available at `http://www.cogs.susx.ac.uk/users/alanje/premon/`.

[18] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation from open maps. *Information and Computation*, 127:164–185, 1996.

[19] A. Joyal, R.H. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:425–446, 1996.

[20] W. Kahl. The term graph programming system HOPS. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 136–149. Springer Verlag, 1999. Available at `http://ist.unibw-muenchen.de/kahl/HOPS/`.

[21] B. König. *Description and Verification of Mobile Processes with Graph Rewriting Techniques*. PhD thesis, Technische Universität München, 1999.

[22] F. Levi and D. Sangiorgi. Controlling interference in ambients. In T. Reps, editor, *Principles of Programming Languages*, pages 352–364. ACM Press, 2000.

[23] R. Milner. Pi-nets: A graphical formalism. In D. Sannella, editor, *European Symposium on Programming*, volume 788 of *Lect. Notes in Comp. Science*, pages 26–42. Springer Verlag, 1995.

[24] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Part I and II. *Information and Computation*, 100:1–77, 1992.

[25] U. Montanari and M. Pistore. Concurrent semantics for the π-calculus. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Computer Science*. Elsevier, 1995.

[26] J. Parrow. Interaction diagrams. *Nordic Journal of Computing*, 2:407–443, 1995.

[27] N. Yoshida. Graph notation for concurrent combinators. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming*, volume 907 of *Lect. Notes in Comp. Science*, pages 393–412. Springer Verlag, 1994.

MFPS 17 Preliminary Version

Regular-Language Semantics for a Call-by-Value Programming Language

Dan R. Ghica [1,2]

Department of Computing and Information Science,
Queen's University, Kingston,
Ontario, Canada K7L 3N6

Abstract

We explain how game semantics can be used to reason about term equivalence in a finitary imperative first order language with arrays. For this language, the game-semantic interpretation of types and terms is fully characterized by their sets of complete plays. Because these sets are regular over the alphabet of moves, they are representable by (extended) regular expressions. The formal apparatus of game semantics is greatly simplified but the good theoretical properties of the model are preserved. The principal advantage of this approach is that it is mathematically elementary, while fully formalized. Since language equivalence for regular languages is decidable, this method of proving term equivalence is suitable for automation.

1 Introduction

In the last decade the use of game semantics in the analysis of programming languages has yielded numerous remarkable theoretical results. Most importantly, this innovative approach led to fully abstract models for languages that have been under semantic scrutiny for decades, such as PCF [10,2] and idealized Algol [4,6]. The theoretical success of game semantics is well complemented by an elegantly articulated and intuitive conceptual association between key language features (such as mutable state or control) and neat combinatorial constraints on strategies (such as *innocence* or *bracketing*)[3].

There is, however, a frustrating aspect of game semantics. While the models proposed are fully abstract, which means that in principle they correctly validate all program equivalences and inequivalences, they are at the same time so intricate that applying them to that end is often a Gordian task. What is needed is an adequate notation that would allow a *calculus* of games,

[1] This author acknowledges the support of a NSERC PGSB grant.

[2] Email: ghica@cs.queensu.ca

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to make proofs less prolix and more formal. While a wieldy and accessible calculus that captures the full power of games may be unattainable, calculi for restricted yet non-trivial subsets of game-based models are very handy. They illustrate the game semantics in an applied setting, making the subject more accessible to those who find its abstractness daunting. But, more importantly, such calculi can actually serve as a foundation for a new and practical approach to program analysis, predicated on solid theoretical results. A similar avenue of research, but as applied to static analysis, is explored by Hankin and Malacaria [8,9].

In a previous paper [7] we showed how a greatly simplified games model of idealized Algol can be used to give elementary proofs to semantically relevant putative equivalences that have been an important part of the study of the language [12]. This paper follows a similar approach, but focuses on a different language, an imperative call-by-value first order language with arrays. This language is important from a practical point of view; it is the idiom in which many common programs, for example for searching or sorting, are written. For this language we present what we believe to be a practicable semantic calculus which can be used for validating term (subprogram) equivalences. Since equivalent subprograms can be replaced in any context, without restrictions, the technique presented here can be applied to both program development through refinement and to program maintenance. We are optimistic about the practical application of such a technique because it is mathematically elementary and calculational, both features considered essential requirements for a "popular semantics" [15]. Moreover, this calculus is fully formal. Because it is based on regular expressions, language equivalence is decidable, which makes it suitable for automation.

2 Foil: a first order imperative language

In this paper we are concerned with a simple, prototypical, programming language that can be found at the core of most of today's imperative languages. It combines the features of the simple imperative language (mutable local variables, control structures) with a recursion-free first-order procedure mechanism based on the simply typed call-by-value lambda calculus, and an elementary data structure facility (arrays). The data sets of Foil are finite, as is the case with realistic programming languages. The decision to set aside higher order procedures and recursion is dictated by the need to confine the formalism to regular expressions only; they are not expressive enough to represent these more powerful features. Another restriction is to allow only uncurried functions, but this is only for the sake of simplicity of presentation. Curried functions can be readily added and explicated within the bounds of the same formalism.

Foil has three kinds of types: return types, argument types and function types. The return types are the most elementary, and can be returned by

functions. They are the "values" of the language: booleans, naturals and a type of commands, **void**, similar to that in C or Java.

$$\tau ::= \mathbf{bool} \mid \mathbf{nat} \mid \mathbf{void}.$$

The argument types include the return types plus all the other types that can be passed as arguments to functions, which are variables, arrays and tuples:

$$\sigma ::= \tau \mid \mathbf{var} \mid \mathbf{array}[\mathbf{n}] \mid \sigma \times \sigma, \quad \mathbf{n} \in \mathbb{N}.$$

The types above and function types, from argument to return types, form the type system of the language:

$$\theta ::= \sigma \mid \sigma \to \tau.$$

The terms of the language are associated with typing judgments of the form:

$$\iota_1 : \theta_1, \iota_2 : \theta_2, \ldots, \iota_k : \theta_k \vdash M : \theta,$$

where ι_i are free variables.

The terms of the language are constants, operators, free variables, control structures, command composition, variable declaration, array declaration, array element selection, assignment, dereferencing and function declaration and application (Figure 1).

3 Game semantics of Foil

The reader is not expected to be familiar with game semantics in order to understand this article. Also, it is not possible to condense such a rich topic in a few pages, but good and comprehensive introductory material is available [3]. The specific games model used to interpret Foil and on which we base the present regular language model is the one developed by Abramsky and McCusker [5,1]. In this section we will only introduce some of the key intuitive concepts of game semantics, especially as applied to call-by-value games.

The concept of *game* employed by game semantics is a broad one: "an activity conducted according to prescribed rules." Computation is represented as a dialogical game between two protagonists: *Player* (P) represents the program and *Opponent* (O) represents the environment, or the context, in which the program is run. The interaction between O and P consists of a sequence of moves, governed by rules. For example, O and P need to take turns and every move needs to be *justified* by a preceding move. The moves are of two kinds, *questions* and *answers*; one of the rules constraining the interplay is that every answer must correspond to the last unanswered question (*bracketing*).

To every type in the language corresponds a *game*; that is, the set of all possible sequences of moves, together with the way in which they are

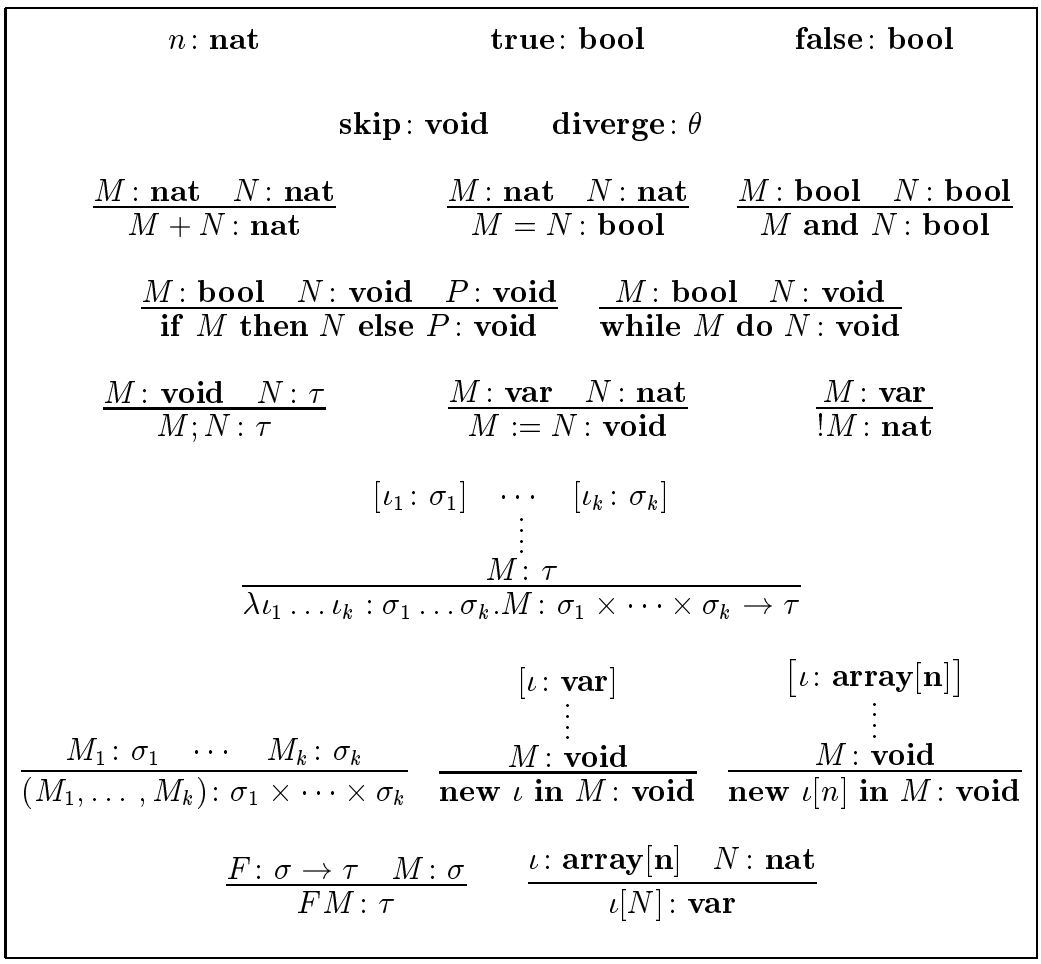


Fig. 1. Terms and typing judgments

justified within the sequence. A program is represented as a set of sequences of moves in the appropriate game, more precisely as a *strategy* for that game: a predetermined way for P to respond to O's moves. The semantic models which provide full abstraction for call-by-name languages are developed within this general games framework.

In an influential paper, Moggi showed that call-by-value languages are interpreted in a Cartesian Closed Category (CCC) with coproducts and a *strong monad* [13]. If a CCC has infinite co-products then its free completion under co-product produces the required monadic structure. The games framework forms indeed a CCC, with games as objects and strategies as morphisms, and McCusker showed how co-product games can be added [11]. These two ideas are incorporated in [5] to create a category of games suitable for interpreting call-by-value languages. Arriving at the concrete call-by-value games presented here is only a matter of carrying out in enough detail the categorical construction.

The resulting games are, however, interesting in their own right because they offer some basic insight into call-by-value computation. A type is not

represented by a game, but by a *family of games*. A strategy interpreting a term has two distinct stages, a *protocol* stage in which one of the members of the family is selected, followed by a play in the selected game. Intuitively, this mirrors the fact that, in call-by-value, all arguments are evaluated exactly once before the body of the function is evaluated. Accordingly, free identifiers can have only one value throughout the evaluation of a term. In contrast, call-by-name allows identifiers to correspond to different values at various points in the evaluation.

This point is illustrated by the following example: $\mathrm{f}:\mathbf{nat}\to\mathbf{nat},\mathrm{x}:\mathbf{nat}\vdash \mathrm{f(x)}:\mathbf{nat}$. For call-by-name a typical play is:

$\mathrm{f}:\mathbf{nat}\to\mathbf{nat}$	$\mathrm{x}:\mathbf{nat}$	$\vdash$	$\mathrm{f(x)}:\mathbf{nat}$
			q
	q		
q			
		q	
		n	
n			
q			
		q	
		n'	
n'			
	m		
			m

O asks for the value of f(x); P asks for the value returned by f; O asks for the argument of f; P asks for the value of x; O answers n; P relays that answer back to O; O asks again for the argument of f and the same cycle repeats; O answers with m to the value returned by f; P relays that answer back to O, answering the initial question. Notice that in the course of the play the value of x can be requested several times, and since the answer is given by O it may change. Now let us look at the same term evaluated under call-by-value:

$\mathrm{f}:\mathbf{nat}\to\mathbf{nat}$	$\mathrm{x}:\mathbf{nat}$	$\vdash$	$\mathrm{f(x)}:\mathbf{nat}$
			?
		?	
		n	
		$?(n)$	
	$?(n)$		
	m		
			m

The moves under the $\vdash$ symbol are not part of the play, but they are some of the concealed activities that are part of the protocol. The play is: O asks for the value of f(x); as a result of the protocol, P asks for the value of f *in component* n; O answers m; P relays the answer back, answering the initial question. Only part of the protocol is shown: P asks in what component should

the play proceed; O answers with a component index n for x; P requests that the game should continue in component n.

In [5] it was shown that this games framework gives a fully abstract model for call-by-value. Moreover, by relaxing one of the constraints on strategies (*innocence*) the same article shows how a fully abstract model for an imperative language with ML-style data references can be defined, using a *good-variable* non-innocent strategy to model mutable store. These ideas are further expanded in [1] to show that dropping another constraint on strategies (*visibility*) gives rise to a fully abstract model for general references (references to data, procedures, higher order functions, other references).

4 Regular language semantics of types

A game for a type, or a strategy for a term, is fully characterized by its set of plays together with the way moves are justified. But if the language is sufficiently restricted then there is only one way in which moves can be justified within a play sequence—FOIL is such a restricted language. This means that such languages can be fully characterized by plays taken to be sequences of moves only. Moreover, the sequences of moves are regular sets, so they can be denoted by (extended) regular expressions. This is an approach we took before, in dealing with ALGOL [7]. For the restricted language, extended regular expressions give a convenient, compact, fully formal calculus, quite handy in defining the semantics of actual programs. The regular-language semantics arises out of the model in [5], by working out the details of the categorical construction.

Definition 4.1 *The set $\mathcal{R}_{\mathcal{A}}$ of extended regular expressions R over a finite alphabet $\mathcal{A}$ is defined as:*

$$
\begin{array}{lll}
R ::= \emptyset \mid \epsilon \mid a, \quad a \in \mathcal{A} & & \textit{Constants},\\
R ::= R \cdot R \mid R + R \mid R \cap R & & \textit{Operators},\\
R ::= R^* & & \textit{Iteration},\\
R ::= R|_{\mathcal{A}'}, \quad \mathcal{A}' \subseteq \mathcal{A} & & \textit{Hiding},\\
R ::= R\langle v \rangle & & \textit{Indexing}.
\end{array}
$$

Most of the above are standard regular expression constructs, to which we add intersection and two new operations, *hiding* and *indexing*. The latter are operations on regular languages that can be carried out directly at the level of regular expressions. Hiding represents a restriction of a regular expression to a subset of its alphabet by removing all the occurrences of symbols in the restricted alphabet; its language is the set of sequences of the original languages with all the elements of $\mathcal{A}'$ deleted. Indexing is defined as the tagging of the first symbol a of any sequence in the language with the string

$$[\![\theta]\!] = \left(\sum_{c \in C_\theta} P_\theta^c\right) \cdot \sum_{k \in K_\theta} R_\theta(k)^*, \text{ where:}$$

$$\begin{array}{rcl}
[\![\tau]\!] & : & P_\tau^v = ? \cdot v, \quad R_\tau = \epsilon, \quad K_\tau = \{\star\}, \quad C_{\mathbf{void}} = \{\star\} \\
 & & C_{\mathbf{nat}} = \{0, 1, \ldots, n_N\} = N, \quad C_{\mathbf{bool}} = \{tt, ff\} \\
[\![\sigma_1 \times \sigma_2]\!] & : & P^c_{\sigma_1 \times \sigma_2} = ? \cdot c, \quad C_{\sigma_1 \times \sigma_2} = C_{\sigma_1} \times C_{\sigma_2} \\
 & & R_{\sigma_1 \times \sigma_2}(k) = \begin{cases} R_{\sigma_1}(k) & \text{if } k \in K_{\sigma_1} \\ R_{\sigma_2}(k) & \text{if } k \in K_{\sigma_2} \end{cases}, K_{\sigma_1 \times \sigma_2} = K_{\sigma_1} \uplus K_{\sigma_2} \\
[\![\mathbf{var}]\!] & : & P_{\mathbf{var}} = ? \cdot \star, \quad C_{\mathbf{var}} = \{\star\}, \quad K_{\mathbf{var}} = N \cup \{\star\} \\
 & & R_{\mathbf{var}} = \sum_{m \in N} read \cdot m + \sum_{m \in N} write(m) \cdot \star \\
[\![\mathbf{array[n]}]\!] & : & P_{\mathbf{array[n]}} = ? \cdot \star, \quad C_{\mathbf{array[n]}} = \{\star\} \\
 & & K_{\mathbf{array[n]}} = \{i | i < n\} \cup \{i | i < n\} \times N \\
 & & R_{\mathbf{array[n]}} = \sum_{\substack{m \in N \\ i < n}} read(i) \cdot m + \sum_{\substack{m \in N \\ i < n}} write(i, m) \cdot \star \\
[\![\sigma \to \tau]\!] & : & P_{\sigma \to \tau} = ? \cdot \star, \quad C_{\sigma \to \tau} = \{\star\} \\
 & & R_{\sigma \to \tau} = \sum_{c \in K_{\sigma \to \tau}} \left(q(c) \cdot \sum_{d \in C_\tau} R_\sigma \langle c \rangle \cdot d\right)^*, \quad K_{\sigma \to \tau} = C_\sigma
\end{array}$$

Fig. 2. Semantics of FOIL types

v, resulting in $a(uv)$, where u is the pre-existing tag of R, possibly empty (ϵ).

A regular-language representation of the game semantics of FOIL is defined as follows. With types we associate games, represented as regular languages over an alphabet denoting the moves. With terms we associate strategies, represented as regular languages over the disjoint sum of the alphabets of the types of the free identifiers and the term itself.

As mentioned in the previous sections, a call-by-value game for a type θ has two stages: a protocol-game P_θ^c followed by a component-game $R_\theta = \sum_{k \in K_\theta} R_\theta(k)$. The protocol part of the play corresponds to the co-product structure which forms the monad, tying together the various components.

A play in the protocol game always has the form $? \cdot c$ for $c \in C_\theta$. We call C_θ *the set of component-selecting moves.* The first move in a play in the component game R_θ has the form $m(k)$, where $k \in K_\theta$. We call K_θ *the set of component-defining moves*; every such component is represented by the regular language $R_\theta(k)$. Notice that C_θ and K_θ are distinct sets. If sets C or K only have one element ($\star$), we will often omit it as an index.

The regular language semantics of FOIL types is the one given in Figure 2. For **void**, **nat** and **bool** the definition is straightforward. Variables **var** are the product of an acceptor and an expression type, not reified in the actual language. Arrays of size n are identified with products of n variables. Proving

these regular expressions correctly represent games is tedious, but routine.

In the case of product and function types it is required that the alphabets (sets of moves) of the types involved are disjoint. This is achieved by systematically tagging all the moves in each alphabet with tags uniquely associated with each type occurrence.

5 Regular language semantics of terms

Terms in FOIL are interpreted as *families of regular expressions*, representing the call-by-value strategies. They have the following form, where P and R are the protocol and the component parts:

$$[\![\iota_1 : \theta_1, \ldots, \iota_n : \theta_n \vdash M : \theta]\!] = \biguplus_{c \in \prod_{i \leq n} C_{\theta_i}} \left\{ \sum_{k \in K_\theta} P_M^{c,k} \cdot R_M(k)^* \right\}.$$

Free identifiers are interpreted as:

$$[\![\iota : \theta \vdash \iota : \theta]\!] \quad : \quad P^{c,k}_{\iota : \theta} = P^c_\theta, \quad k \in K_\theta, \quad R_\iota = R_\theta[m/m{\cdot}m^\iota][n/n^\iota{\cdot}n],$$

for all moves m of odd index in the play (the O-moves) and n of even index (P-moves). Since in the regular expressions moves always occur in pairs, this substitution can be carried out directly on the regular expression defining the plays. This "doubling-up" of moves is the representation of the important *copy-cat* strategy of game semantics. The new tag ι is meant to differentiate between the two occurrences of type θ, in the environment and in the term itself. For example:

$$\begin{aligned} [\![\mathrm{x} : \mathbf{nat} \vdash \mathrm{x} : \mathbf{nat}]\!] &= \{ ? \cdot n \cdot \epsilon \mid n \in N \}, \\ [\![\mathrm{f} : \mathbf{nat} \to \mathbf{nat} \vdash \mathrm{f} : \mathbf{nat} \to \mathbf{nat}]\!] &= \left\{ ? {\cdot} \star {\cdot} R_f^* \right\} \\ &= \left\{ ? {\cdot} \star {\cdot} \left(\sum_{i,n \in N} q(i) {\cdot} q(i)^f {\cdot} n^f {\cdot} n \right)^* \right\}. \end{aligned}$$

For all basic constants of the language we have $R = \epsilon$ and:

$$\begin{aligned} P_{n : \mathbf{nat}} &= ? \cdot n, & P_{\mathbf{true} : \mathbf{bool}} &= ? \cdot tt, & P_{\mathbf{diverge} : \theta} &= \emptyset, \\ P_{\mathbf{skip} : \mathbf{void}} &= ? \cdot \star, & P_{\mathbf{false} : \mathbf{bool}} &= ? \cdot ff. \end{aligned}$$

Binary arithmetic, logic and arithmetic-logic operators can be interpreted as abbreviations involving predefined functions, for which the semantics of application (to be defined later) will be used to compose the meanings of subphrases:

$$[\![+ : \mathbf{nat} \times \mathbf{nat} \to \mathbf{nat}]\!] \quad : \quad P_+^{m,n} = ? \cdot \star, \quad R_+(m,n) = ?(m,n){\cdot}(m+n),$$

with $m, n \in N$. Similarly for all other operators. Sequencing is:

$$[\![-;-\colon \mathbf{void}\times\mathbf{void}\to\mathbf{void}]\!] \quad : \quad P_{;} = ?\cdot\star, \quad R_{;} = \epsilon,$$
$$[\![-;-\colon \mathbf{void}\times\mathbf{nat}\to\mathbf{nat}]\!] \quad : \quad P_{;} = \sum_{n\in N} ?\cdot n, \quad R_{;} = \epsilon.$$

Assignment and dereferencing are respectively:

$$[\![-:=-\colon \mathbf{var}\times\mathbf{nat}\to\mathbf{void}]\!] \quad : \quad P^{n}_{:=} = ?\cdot\star, \quad R_{:=}(n) = write(n)\cdot\star,$$
$$[\![!-\colon \mathbf{var}\to\mathbf{nat}]\!] \quad : \quad P_{!} = ?\cdot\star, \quad R_{!} = \sum_{n\in N} read\cdot n.$$

Abstraction is defined as explicitly reindexing the regular expressions denoting the meaning of a term M with the component moves of the types of identifiers abstracted over:

$$[\![\Gamma\vdash\lambda\iota_1\ldots\iota_k:\sigma_1\times\cdots\times\sigma_k.M\colon\sigma_1\times\cdots\times\sigma_k\to\tau]\!]:$$
$$P^{k}_{\lambda} = ?\cdot\star,\ R_{\lambda}(k) = Q\langle k\rangle,\ k\in\prod_{i\le n} C_{\sigma_i},\ Q\in[\![\Gamma,\iota_1\colon\sigma_1,\ldots,\iota_k\colon\sigma_k\vdash M]\!].$$

The most important, and the most complex, is the meaning of application:

$$[\![\Gamma\vdash MN:\tau]\!] \quad : \quad P_{MN} = ?\cdot P^{c'}_{N}\cdot R'_{M}(c)[x\cdot y/R^{x,y}_{N}], \quad R_{MN} = \epsilon$$
$$\text{where} \qquad ?\cdot P^{c'}_{N}\cdot c\cdot R_N\in[\![\Gamma''\vdash N:\sigma]\!], \quad R_N = \sum_{x,y} x\cdot R^{x,y}_{N}\cdot y$$
$$?\cdot\star\cdot\sum_{c} q(c)\cdot R'_{M}(c)\in[\![\Gamma'\vdash M:\sigma\to\tau]\!], \quad R_\sigma = \sum_{x,y} x\cdot y.$$

The regular expressions and regular expression families involved in the definition above are well defined in general, with one exception. If N is a diverging term then either one of $P^{c'}_{N}$ and R_N may be $\emptyset$ with the other arbitrary, $\emptyset$ being a zero-element for concatenation. This ambiguity is resolved by always choosing $P^{c'}_{N} = \emptyset$, to be consistent with the fact that $P_{\mathbf{diverge}} = \emptyset$, as presented earlier. The choice for R_N is then irrelevant, ϵ by convention.

The semantics of application is derived directly from the game semantics as well, more precisely from compositions of strategies. In composing strategies, which is how application is interpreted, the moves in the game (type occurrence) through which composition is realized serve as "triggers" which switch the thread of execution between the two strategies. In our particular case, whenever such a move x occurs, a regular expression denoting the trace of execution for the argument is inserted in the regular expression denoting the body of the function, up to the point where another context-switching move y occurs. In the process of composing strategies all trigger moves are subsequently hidden. Here, the key technique is to decompose a regular expression into smaller regular expressions and, using systematic substitution and concatenation, create the regular expressions corresponding to the result. This technique will be also used in defining the regular language semantics of terms which are not abbreviations.

Since functions are not curried we need to define pairing. It reflects the left-to-right order of argument evaluation in function call, specific to FOIL:

$$[\![\Gamma \vdash (M,N)\colon \sigma \times \sigma']\!] :$$
$$P_{M,N}^{(c,c')(k,k')} = ?\cdot Q_M^{c,k}\cdot Q_N^{c',k'}\cdot(c,c'), \quad R_{M,N}(k,k') = \big(R_M(k) + R_N(k')\big)^*$$
$$\text{where } P_M^{c,k} = ?\cdot Q_M^{c,k}\cdot c, \quad P_N^{c,k'} = ?\cdot Q_N^{c',k'}\cdot c'.$$

Branching and looping are defined directly, not as abbreviations:

$$[\![\textbf{if } B \textbf{ then } M \textbf{ else } N]\!] \quad : \quad P_{\textbf{if}} = (?\cdot P_B^{tt}\cdot P'_M\cdot\star) + (?\cdot P_B^{ff}\cdot P'_N\cdot\star), \quad R_{\textbf{if}} = \epsilon,$$
$$[\![\textbf{while } B \textbf{ do } M]\!] \quad : \quad P_{\textbf{while}} = ?\cdot(P_B^{tt}\cdot P'_M)^*\cdot P_B^{ff}\cdot\star, \quad R_{\textbf{while}} = \epsilon,$$
$$\text{where} \quad : \quad P_B = \sum_{v\in\{tt,ff\}} ?\cdot P_B^v\cdot v, \quad P_M = ?\cdot P'_M\cdot\star, \quad P_N = ?\cdot P'_N\cdot\star\ .$$

The semantics of **if** is directly specified in the games semantics. Looping in game semantics is defined as an abbreviation using the recursion combinator. A general recursion combinator is not specified in this treatment, but the fixed point can be calculated by hand; the semantics of **while** above is the result of that calculation.

Array element access is also directly defined:

$$[\![\Gamma \vdash \iota[N] : \textbf{var}]\!] \quad :$$
$$P_{\iota[N]}^k = ?\cdot P_N^{k'}\cdot\star, \quad R_{\iota[N]}(k) = \sum_{m\in N} read(k)\cdot m + \sum_{m\in N} write(k,m)\cdot\star,$$
$$\text{where } P_N^{k'} = ?\cdot P_N^k\cdot k.$$

Finally, as in the case of ALGOL, local variables are realized by imposing a *good variable* restriction on the plays and by *hiding* the actions of the local variables. Good-variable behaviour simply means that the last value written in a variable will be the next value read from that variable; this restriction is imposed using intersection with the following regular expression, associated with a variable ι:

$$\gamma^\iota = \overline{\mathcal{A}_\iota}^*\cdot\Big(\overline{\mathcal{A}_\iota}^*\cdot\textstyle\sum_{n\in N} write(n)^\iota\cdot\star^\iota\cdot\overline{\mathcal{A}_\iota}^*\cdot(read^\iota\cdot n^\iota\cdot\overline{\mathcal{A}_\iota}^*)^*\Big)^*,$$

where $\mathcal{A}_\iota = \{read^\iota, write(n)^\iota, n^\iota, \star^\iota | n \in N\}$ is the set of all moves tagged by ι, *i.e.* all moves involving variable ι. Therefore local variable definition is:

$$[\![\textbf{new } \iota \textbf{ in } M]\!] = \{\,(\gamma^\iota \cap Q)\,|_{\mathcal{A}_\iota} \quad | Q \in [\![M]\!]\}\,.$$

For similar reasons, the meaning of array declaration is:

$$[\![\textbf{new } \iota[n] \textbf{ in } M]\!] = \left\{ \left(\bigcap_{i\le n}\gamma^{\iota[i]} \cap Q\right)\Big|_{\bigcup_{i\le n}\mathcal{A}_{\iota[i]}} \;\middle|\; Q \in [\![M]\!]\right\},$$
$$\gamma^{\iota[i]} = \overline{\mathcal{A}_{\iota[i]}}^*\cdot\Big(\overline{\mathcal{A}_{\iota[i]}}^*\cdot\textstyle\sum_{n\in N} write(n,i)^\iota\cdot\star^\iota\cdot\overline{\mathcal{A}_{\iota[i]}}^*\cdot(read(i)^\iota\cdot n^\iota\cdot\overline{\mathcal{A}_{\iota[i]}}^*)^*\Big)^*, i \le n.$$

This concludes the semantic definition of FOIL. We can state that:

Lemma 5.1 (Representation) *The regular language semantics of* FOIL *is a fully correct representation of the games and strategies used in the game semantic model.*

From this, it follows directly from [5] that:

Theorem 5.2 (Full Abstraction) *The regular language semantics of* FOIL *is fully abstract,* i.e. *two terms of* FOIL *are equivalent if and only if they denote the same family of regular languages:*

$$\text{For all } \Gamma \vdash P, Q : \theta, \quad P \equiv Q \iff [\![P]\!] = [\![Q]\!] .$$

In addition, since the representation is by regular languages, for which language equality is decidable, it follows directly that:

Theorem 5.3 (Decidability) *Equivalence of two terms of* FOIL *is decidable.*

6 Example of reasoning

Since one of the stated purposes of this article is to provide a basis for a new and potentially practical tool, we think it is important to show in some detail an example. Space constraints prevent us from presenting a realistic program, so we will instead prove a simple, but theoretically important, equivalence of FOIL:

$$\mathrm{f} : \mathbf{nat} \to \mathbf{void}, \mathrm{v} : \mathbf{nat} \vdash \quad \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x}) \equiv_{\mathbf{void}} \mathrm{f}(\mathrm{v}).$$

Proof:

$$[\![\mathrm{x} : \mathbf{var} \vdash \mathrm{x} : \mathbf{var}]\!] : P_x = ? \cdot \star, \\ R_x = \sum_{n \in N} read \cdot read^x \cdot n^x \cdot n + \sum_{m \in N} write(m) \cdot write(m)^x \cdot \star^x \cdot \star$$

$$[\![\mathrm{x} : \mathbf{var} \vdash !\mathrm{x} : \mathbf{nat}]\!] : P_{!x} = ? \cdot \sum_{n \in N} \cdot read^x \cdot n^x \cdot n, \quad R_{!x} = \epsilon$$

$$[\![\mathrm{f} : \mathbf{nat} \to \mathbf{void} \vdash \mathrm{f} : \mathbf{nat} \to \mathbf{void}]\!] : P_f = ? \cdot \star, \quad R_f = \left(\textstyle\sum_{i \in N} q(i) \cdot q(i)^f \cdot \star^f \cdot \star\right)^*$$

$$[\![\mathrm{f} : \mathbf{nat} \to \mathbf{void}, \mathrm{x} : \mathbf{var} \vdash \mathrm{f}(!\mathrm{x}) : \mathbf{nat}]\!] : \\ P_{f(!x)} = ? \cdot \sum_{n \in N} read^x \cdot n^x \cdot \left(q(n)^f \cdot \star^f\right)^* \cdot \star, \quad R_{f(!x)} = \epsilon$$

$$[\![\mathrm{v} : \mathbf{nat}, \mathrm{x} : \mathbf{var} \vdash \mathrm{x} := \mathrm{v} : \mathbf{void}]\!] : P^v_{x:=v} = ? \cdot write(v)^x \cdot \star^x \cdot \star, \quad R_{x:=v} = \epsilon$$

$$[\![\mathrm{f} : \mathbf{nat} \to \mathbf{void}, \mathrm{v} : \mathbf{nat}, \mathrm{x} : \mathbf{var} \vdash \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x}) : \mathbf{void}]\!] : \\ P^v_{x:=v;f(!x)} = ? \cdot write(v)^x \cdot \star^x \cdot \sum_{n \in N} read^x \cdot n^x \cdot \left(q(n)^f \cdot \star^f\right)^* \cdot \star, \quad R_{x:=v;f(!x)} = \epsilon$$

$$[\![\mathrm{f}\colon \mathbf{nat} \to \mathbf{void}, \mathrm{v}\colon \mathbf{nat} \vdash \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x})\colon \mathbf{void}]\!] :$$
$$P^v = ?\cdot \cancel{write(v)^x \cdot \star^x \cdot read^x \cdot v^x} \cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star = ?\cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star, \quad R = \epsilon.$$

Therefore:

$$[\![\mathrm{f}\colon \mathbf{nat} \to \mathbf{void}, \mathrm{v}\colon \mathbf{nat} \vdash \mathbf{new}\ \mathrm{x}\ \mathbf{in}\ \mathrm{x} := \mathrm{v}; \mathrm{f}(!\mathrm{x})\colon \mathbf{void}]\!]$$
$$= \left\{?\cdot \left(q(v)^f \cdot \star^f\right)^* \cdot \star \,\middle|\, v \in N\right\} = [\![\mathrm{f}\colon \mathbf{nat} \to \mathbf{void}, \mathrm{v}\colon \mathbf{nat} \vdash \mathrm{f}(\mathrm{v})\colon \mathbf{void}]\!].$$

7 Conclusion

We have presented a games-based regular language semantics for an imperative language with first order procedures using call-by-value function application, with arrays and variables passed by-reference. The model is obtained directly from the game semantic model [5,1] by working out the details of the category-theoretical presentation and by observing that much of the games apparatus (justification pointers, *etc.*) is unnecessary in handling the present language subset. Two important and useful features of imperative programs with procedures, recursion and pointers, are omitted. A fixed-point combinator can not be defined using regular languages only, but fixed points of functions can be calculated "by hand," as we did in dealing with the **while** construct. Data pointers also can not be represented directly using the present formalism, but they could be in principle encoded using arrays and array indices—but this method has limitations.

Acknowledgement

Guy McCusker's suggestions and explanations were essential in the writing of this paper, I owe him a great deal. I would like to thank Bob Tennent for his support and encouragement. Many thanks are due to the anonymous referees for providing insightful comments and pertinent corrections.

References

[1] Abramsky, S., K. Honda and G. McCusker, *A fully abstract game semantics for general references*, in: *Proceedings, Thirteenth Annual IEEE Symposium on Logic in Computer Science*, 1998.

[2] Abramsky, S., P. Malacaria and R. Jagadeesan, *Full abstraction for PCF*, Lecture Notes in Computer Science **789** (1994), pp. 1–59.

[3] Abramsky, S. and G. McCusker, *Game semantics*, lecture notes, 1997 Marktoberdorf summer school (available from `http://www.dcs.ed.ac.uk/home/samson/mdorf97.ps.gz`).

[4] Abramsky, S. and G. McCusker, *Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions*, in: O'Hearn and Tennent [14] pp. 297–329, two volumes.

[5] Abramsky, S. and G. McCusker, *Call-by-value games*, in: *CSL: 11th Workshop on Computer Science Logic*, LNCS **1414**, 1998, pp. 1–17.

[6] Abramsky, S. and G. McCusker, *Full abstraction for Idealized Algol with passive expressions*, Theoretical Computer Science **227** (1999), pp. 3–42.

[7] Ghica, D. R. and G. McCusker, *Reasoning about idealized* ALGOL *using regular languages*, in: *Proceedings of 27th International Colloquium on Automata, Languages and Programming ICALP 2000*, LNCS **1853** (2000), pp. 103–116.

[8] Hankin, C. and P. Malacaria, *Generalised flowcharts and games*, Lecture Notes in Computer Science **1443** (1998).

[9] Hankin, C. and P. Malacaria, *Non-deterministic games and program analysis: an application to security*, in: *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, 1999 pp. 443–452.

[10] Hyland, J. M. E. and C.-H. L. Ong, *On full abstraction for PCF: I, II and III*, Information and Computation **163** (2000).

[11] McCusker, G., "Games and Full Abstraction for a Functional Metalanguage with Recursive Types," Distinguished Dissertations, Springer-Verlag Limited, 1998.

[12] Meyer, A. R. and K. Sieber, *Towards fully abstract semantics for local variables: preliminary report*, in: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 191–203, reprinted as Chapter 7 of [14].

[13] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[14] O'Hearn, P. W. and R. D. Tennent, editors, "Algol-like Languages," Progress in Theoretical Computer Science, Birkhäuser, Boston, 1997, two volumes.

[15] Schmidt, D. A., *On the need for a popular formal semantics*, ACM SIGPLAN Notices **32** (1997), pp. 115–116.

MFPS 17 Preliminary Version

Typing Correspondence Assertions for Communication Protocols

Andrew D. Gordon

Microsoft Research, Cambridge

Alan Jeffrey

DePaul University, Chicago

Abstract

Woo and Lam propose correspondence assertions for specifying authenticity properties of security protocols. The only prior work on checking correspondence assertions depends on model-checking and is limited to finite-state systems. We propose a dependent type and effect system for checking correspondence assertions. Since it is based on type-checking, our method is not limited to finite-state systems. This paper presents our system in the simple and general setting of the π-calculus. We show how to type-check correctness properties of example communication protocols based on secure channels. In a related paper, we extend our system to the more complex and specific setting of checking cryptographic protocols based on encrypted messages sent over insecure channels.

1 Introduction

Correspondence Assertions To a first approximation, a correspondence assertion about a communication protocol is an intention that follows the pattern:

> If one principal ever reaches a certain point in a protocol, then some other principal has previously reached some other matching point in the protocol.

We record such intentions by annotating the program representing the protocol with labelled assertions of the form $\mathsf{begin}\, L$ or $\mathsf{end}\, L$. These assertions have no effect at runtime, but notionally indicate that a principal has reached a certain point in the protocol. The following more accurately states the intention recorded by these annotations:

> If the program embodying the protocol ever asserts $\mathsf{end}\, L$, then there is a distinct previous assertion of $\mathsf{begin}\, L$.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Woo and Lam [23] introduce correspondence assertions to state intended properties of authentication protocols based on cryptography. Consider a protocol where a principal a generates a new session key k and transmits it to b. We intend that if a run of b ends a key exchange believing that it has received key k from a, then a generated k as part of a key exchange intended for b. We record this intention by annotating a's generation of k by the label $\mathsf{begin}\ \langle a, b, k\rangle$, and b's reception of k by the label $\mathsf{end}\ \langle a, b, k\rangle$.

A protocol can fail a correspondence assertion because of several kinds of bug. One kind consists of those bugs that cause the protocol to go wrong without any external interference. Other kinds are bugs where an unreliable or malicious network or participant causes the protocol to fail.

This Paper We show in this paper that correctness properties expressed by correspondence assertions can be proved by type-checking. We embed correspondence assertions in a concurrent programming language (the π-calculus of Milner, Parrow, and Walker [17]) and present a new type and effect system that guarantees safety of well-typed assertions. We show several examples of how correspondence assertions can be proved by type-checking.

Woo and Lam's paper introduces correspondence assertions but provides no techniques for proving them. Clarke and Marrero [4] use correspondence assertions to specify properties of e-commerce protocols, such as authorizations of transactions. To the best of our knowledge, the only previous work on checking correspondence assertions is a project by Marrero, Clarke, and Jha [16] to apply model-checking techniques to finite state versions of security protocols. Since our work is based on type-checking, it is not limited to finite state systems. Moreover, type-checking is compositional: we can verify components in isolation, and know that their composition is safe, without having to verify the entire system. Unlike Marrero, Clarke, and Jha's work, however, the system of the present paper does not deal with cryptographic primitives, and nor does it deal with an arbitrary opponent. Still, in another paper [9], we adapt our type and effect system to the setting of the spi-calculus [1], an extension of the π-calculus with abstract cryptographic primitives. This adaptation can show, moreover, that properties hold in the presence of an arbitrary untyped opponent.

Review of The Untyped π-Calculus Milner, Parrow, and Walker's π-calculus is a concurrent formalism to which many kinds of concurrent computation may be reduced. Its simplicity makes it an attractive vehicle for developing the ideas of this paper, while its generality suggests they may be widely applicable. Its basic data type is the *name*, an unguessable identifier for a communications channel. Computation is based on the exchange of messages, tuples of names, on named channels. Programming in the π-calculus is based on the following constructs (written, unusually, with keywords, for

the sake of clarity). The rest of the paper contains many examples. An output process $\mathsf{out}\ x\langle y_1, \ldots, y_n\rangle$ represents a message $\langle y_1, \ldots, y_n\rangle$ sent on the channel x. An input process $\mathsf{inp}\ x(z_1, \ldots, z_n); P$ blocks till it finds a message sent on the channel x, reads the names in the message into the variables $z_1, \ldots, z_n$, and then runs P. The process $P \mid Q$ is the parallel composition of the two processes P and Q; the two may run independently or communicate on shared channels. The name generation process $\mathsf{new}(x); P$ generates a fresh name, calls it x, then runs P. Unless P reveals x, no other process can use this fresh name. The replication process $\mathsf{repeat}\ P$ behaves like an unbounded parallel array of replicas of P. The process stop represents inactivity; it does nothing. Finally, the conditional $\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$ compares the names x and y. If they are the same it runs P; otherwise it runs Q.

2 Correspondence Assertions, by Example

This section introduces the idea of defining correspondence assertions by annotating code with begin- and end-events. We give examples of both safe code and of unsafe code, that is, of code that satisfies the correspondence assertions induced by its annotations, and of code that does not.

A transmit-acknowledge handshake is a standard communications idiom, easily expressed in the π-calculus: along with the actual message, the sender transmits an acknowledgement channel, upon which the receiver sends an acknowledgement. We intend that:

> During a transmit-acknowledge handshake, if the sender receives an acknowledgment, then the receiver has obtained the message.

Correspondence assertions can express this intention formally. Suppose that a and b are the names of the sender and receiver, respectively. We annotate the code of the receiver b with a begin-assertion at the point after it has received the message msg. We annotate the code of the sender a with an end-assertion at the point after it has received the acknowledgement. We label both assertions with the names of the principals and the transmitted message, $\langle a, b, msg\rangle$. Hence, we assert that if after sending msg to b, the sender a receives an acknowledgement, then a distinct run of b has received msg.

Suppose that c is the name of the channel on which principal b receives messages from a. Here is the π-calculus code of the annotated sender and receiver:

$$
\begin{array}{ll}
Rcver(a,b,c) \triangleq & Snder(a,b,c) \triangleq \\
\quad \mathsf{inp}\ c(msg, ack); & \quad \mathsf{new}(msg); \mathsf{new}(ack); \\
\quad \mathsf{begin}\ \langle a, b, msg\rangle; & \quad \mathsf{out}\ c\langle msg, ack\rangle; \mathsf{inp}\ ack(); \\
\quad \mathsf{out}\ ack\langle\rangle & \quad \mathsf{end}\ \langle a, b, msg\rangle
\end{array}
$$

The sender creates a fresh message msg and a fresh acknowledgement channel

ack, sends the two on the channel c, waits for an acknowledgement, and then asserts an end-event labelled $\langle a, b, msg \rangle$.

The receiver gets the message msg and the acknowledgement channel ack off c, asserts a begin-event labelled $\langle a, b, msg \rangle$, and sends an acknowledgement on ack.

We say a program is safe if it satisfies the intentions induced by the begin- and end-assertions. More precisely, a program is *safe* just if for every run of the program and for every label L, there is a distinct begin-event labelled L preceding every end-event labelled L. (We formalize this definition in Section 5.)

Here are three combinations of our examples: two safe, one unsafe.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 1: } \textit{safe}\text{)} \\ \quad \mathit{Snder}(a, b, c) \mid & \\ \quad \mathit{Rcver}(a, b, c) & \end{array}$$

Example 1 uses one instance of the sender and one instance of the receiver to represent a single instance of the protocol. The restriction $\mathsf{new}(c)$; makes the channel c private to the sender and the receiver. This assembly is safe; its only run correctly implements the handshake protocol.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 2: } \textit{safe}\text{)} \\ \quad \mathit{Snder}(a, b, c) \mid & \\ \quad \mathit{Snder}(a, b, c) \mid & \\ \quad \mathsf{repeat}\ \mathit{Rcver}(a, b, c) & \end{array}$$

Example 2 uses two copies of the sender—representing two attempts by a single principal a to send a message to b—and a replicated copy of the receiver—representing the principal b willing to accept an unbounded number of messages. Again, this assembly is safe; any run consists of an interleaving of two correct handshakes.

$$\begin{array}{ll} \mathsf{new}(c); & \text{(Example 3: } \textit{unsafe}\text{)} \\ \quad \mathit{Snder}(a, b, c) \mid & \\ \quad \mathit{Snder}(a', b, c) \mid & \\ \quad \mathsf{repeat}\ \mathit{Rcver}(a, b, c) & \end{array}$$

Example 3 is a variant on Example 2, where we keep the replicated receiver b, but change the identity of one of the senders, so that the two senders represent two different principals a and a'. These two principals share a single channel c to the receiver. Since the identity a of the sender is a parameter of $\mathit{Rcver}(a, b, c)$ rather than being explicitly communicated, this assembly is unsafe. There is a run in which a' generates msg and ack, and sends them to b; b asserts a begin-event labelled $\langle a, b, msg \rangle$ and outputs on ack; then a' asserts an end-event labelled $\langle a', b, msg \rangle$. This end-event has no corresponding begin-event so the assembly is unsafe, reflecting the possibility that the receiver can

be mistaken about the identity of the sender.

3 Typing Correspondence Assertions

3.1 Types and Effects

Our type and effect system is based on the idea of assigning types to names and effects to processes. A type describes what operations are allowed on a name, such as what messages may be communicated on a channel name. An effect describes the collection of labels of events the process may end while not itself beginning. We compute effects based on the intuition that end-events are accounted for by preceding begin-events; a begin-event is a credit while an end-event is a debit. According to this metaphor, the effect of a process is an upper bound on the debt a process may incur. If we can assign a process the empty effect, we know all of its end-events are accounted for by begin-events. Therefore, we know that the process is safe, that is, its correspondence assertions are true.

An essential ingredient of our typing rules is the idea of attaching a *latent effect* to each channel type. We allow any process receiving off a channel to treat the latent effect as a credit towards subsequent end-events. This is sound because we require any process sending on a channel to treat the latent effect as a debit that must be accounted for by previous begin-events. Latent effects are at the heart of our method for type-checking events begun by one process and ended by another.

The following table describes the syntax of types and effects. As in most versions of the π-calculus, we make no lexical distinction between names and variables, ranged over by a, b, c, x, y, z. An *event label*, L, is simply a tuple of names. Event labels identify the events asserted by begin- and end-assertions. An *effect*, e, is a multiset, that is, an unordered list, of event labels, written as $[L_1, \ldots, L_n]$. A *type*, T, takes one of two kinds. The first kind, Name, is the type of pure names, that is, names that only support equality operations, but cannot be used as channels. We use Name as the type of names that identify principals, for instance. The second kind, $\mathsf{Ch}(x_1{:}T_1, \ldots, x_n{:}T_n)e$, is a type of a channel communicating n-tuples of names, of types $T_1, \ldots, T_n$, with latent effect e. The names $x_1, \ldots, x_n$ are bound; the scope of each x_i consists of the types $T_{i+1}, \ldots, T_n$, and the latent effect e. We identify types up to the consistent renaming of bound names.

Names, Event Labels, Effects, and Types:

a, b, c, x, y, z	names, variables
$L ::= \langle x_1, \ldots, x_n \rangle$	event label: tuple of names
$e ::= [L_1, \ldots, L_n]$	effect: multiset of event labels
$T ::=$	type
Name	pure name

$\mathsf{Ch}(x_1{:}T_1, \ldots, x_n{:}T_n)e$	channel with latent effect e

For example:

- $\mathsf{Ch}()[\,]$, a synchronization channel (that is, a channel used only for synchronization) with no latent effect.
- $\mathsf{Ch}(a{:}\mathsf{Name})[\langle b \rangle]$, a channel for communicating a pure name, costing $[\langle b \rangle]$ to senders and paying $[\langle b \rangle]$ to receivers, where b is a fixed name.
- $\mathsf{Ch}(a{:}\mathsf{Name})[\langle a \rangle]$, a channel for communicating a pure name, costing $[\langle a \rangle]$ to senders and paying $[\langle a \rangle]$ to receivers, where a is the name communicated on the channel.
- $\mathsf{Ch}(a{:}\mathsf{Name}, b{:}\mathsf{Ch}()[\langle a \rangle])[\,]$, a channel with no latent effect for communicating pairs of the form a, b, where a is a pure name, and b is the name of a synchronization channel, costing $[\langle a \rangle]$ to senders and paying $[\langle a \rangle]$ to receivers.

The following is a convenient shorthand for the lists of typed variable declarations found in channel types:

Notation for Typed Variables:

$\vec{x}{:}\vec{T} \triangleq x_1{:}T_1, \ldots, x_n{:}T_n$	where $\vec{x} = x_1, \ldots, x_n$ and $\vec{T} = T_1, \ldots, T_n$
$\epsilon \triangleq ()$	the empty list

The following equations define the the sets of free names of our syntax as follows: variable declarations, $\mathsf{fn}(\epsilon{:}\epsilon) \triangleq \varnothing$ and $\mathsf{fn}(\vec{x}{:}\vec{T}, x{:}T) \triangleq \mathsf{fn}(\vec{x}{:}\vec{T}) \cup (\mathsf{fn}(T) - \{\vec{x}\})$; types, $\mathsf{fn}(\mathsf{Name}) \triangleq \varnothing$ and $\mathsf{fn}(\mathsf{Ch}(\vec{x}{:}\vec{T})e) \triangleq \mathsf{fn}(\vec{x}{:}\vec{T}) \cup (\mathsf{fn}(e) - \{\vec{x}\})$; event labels, $\mathsf{fn}(\langle x_1, \ldots, x_n \rangle) \triangleq \{x_1, \ldots, x_n\}$; and events, $\mathsf{fn}([L_1, \ldots, L_1]) \triangleq \mathsf{fn}(L_1) \cup \cdots \cup \mathsf{fn}(L_n)$.

For any of these forms of syntax, we write $-\{x \leftarrow y\}$ for the operation of capture-avoiding substitution of the name y for each free occurrence of the name x. We write $-\{\vec{x} \leftarrow \vec{y}\}$, where $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$ for the iterated substitution $-\{x_1 \leftarrow y_1\} \cdots \{x_n \leftarrow y_n\}$.

3.2 Syntax of our Typed π-Calculus

We explained the informal semantics of begin- and end-assertions in Section 2, and of the other constructs in Section 1.

Processes:

$P, Q, R ::=$	process
$\mathsf{out}\ x\langle y_1, \ldots, y_n \rangle$	polyadic asynchronous output
$\mathsf{inp}\ x(y_1{:}T_1, \ldots, y_n{:}T_n); P$	polyadic input
$\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mathsf{new}(x{:}T); P$	name generation
$P \mid Q$	composition
$\mathsf{repeat}\ P$	replication

stop	inactivity
$\mathsf{begin}\ L; P$	begin-assertion
$\mathsf{end}\ L; P$	end-assertion

There are two name binding constructs: input and name generation. In an input process $\mathsf{inp}\ x(y_1{:}T_1, \ldots, y_n{:}T_n); P$, each name y_i is bound, with scope consisting of T_{i+1}, ..., T_n, and P. In a name restriction $\mathsf{new}(x{:}T); P$, the name x is bound; its scope is P. We write $P\{x{\leftarrow}y\}$ for the outcome of a capture-avoiding substitution of the name y for each free occurrence of the name x in the process P. We identify processes up to the consistent renaming of bound names. We let $\mathsf{fn}(P)$ be the set of free names of a process P. We sometimes write an output as $\mathsf{out}\ x\langle\vec{y}\rangle$ where $\vec{y} = y_1, \ldots, y_n$, and an input as $\mathsf{inp}\ x(\vec{y}{:}\vec{T}); P$, where $\vec{y}{:}\vec{T}$ is a variable declaration written in the notation introduced in the previous section. We write $\mathsf{out}\ x\langle\vec{y}\rangle; P$ as a shorthand for $\mathsf{out}\ x\langle\vec{y}\rangle \mid P$.

3.3 Intuitions for the Type and Effect System

As a prelude to our formal typing rules, we present the underlying intuitions. Recall the intuition that end-events are costs to be accounted for by begin-events. When we say a process P has effect e, it means that e is an upper bound on the begin-events needed to precede P to make the whole process safe. In other words, if P has effect $[L_1, \ldots, L_n]$ then $\mathsf{begin}\ L_1; \cdots; \mathsf{begin}\ L_n; P$ is safe.

Typing Assertions An assertion $\mathsf{begin}\ L; P$ pays for one end-event labelled L in P; so if P is a process with effect e, then $\mathsf{begin}\ L; P$ is a process with effect $e{-}[L]$, that is, the multiset e with one occurrence of L deleted. So we have a typing rule of the form:

$$P : e \quad \Rightarrow \quad \mathsf{begin}\ L; P : e{-}[L]$$

If P is a process with effect e, then $\mathsf{end}\ L; P$ is a process with effect $e{+}[L]$, that is, the concatenation of e and $[L]$. We have a rule:

$$P : e \quad \Rightarrow \quad \mathsf{end}\ L; P : e{+}[L]$$

Typing Name Generation and Concurrency The effect of a name generation process $\mathsf{new}(x{:}T); P$, is simply the effect of P. To prevent scope confusion, we forbid x from occurring in this effect.

$$P : e,\ \ x \notin \mathsf{fn}(e) \quad \Rightarrow \quad \mathsf{new}(x{:}T); P : e$$

The effect of a concurrent composition of processes is the multiset union of the constituent processes.

$$P : e_P,\ \ Q : e_Q \quad \Rightarrow \quad P \mid Q : e_P{+}e_Q$$

The inactive process asserts no end-events, so its effect is empty.

$\mathsf{stop} : [\,]$

The replication of a process P behaves like an unbounded array of replicas of P. If P has a non-empty effect, then its replication would have an unbounded effect, which could not be accounted for by preceding begin-assertions. Therefore, to type $\mathsf{repeat}\ P$ we require P to have an empty effect.

$$P : [\,] \quad \Rightarrow \quad \mathsf{repeat}\ P : [\,]$$

Typing Communications We begin by presenting the rules for typing communications on monadic channels with no latent effect, that is, those with types of the form $\mathsf{Ch}(y{:}T)[\,]$. The communicated name has type T. An output $\mathsf{out}\ x\langle z\rangle$ has empty effect. An input $\mathsf{inp}\ x(y{:}T); P$ has the same effect as P. Since the input variable in the process and in the type are both bound, we may assume they are the same variable y.

$$x : \mathsf{Ch}(y{:}T)[\,],\ z : T \quad \Rightarrow \quad \mathsf{out}\ x\langle z\rangle : [\,]$$
$$x : \mathsf{Ch}(y{:}T)[\,],\ P : e,\ y \notin \mathsf{fn}(e) \quad \Rightarrow \quad \mathsf{inp}\ x(y{:}T); P : e$$

Next, we consider the type $\mathsf{Ch}(y{:}T)e_\ell$ of monadic channels with latent effect e_ℓ. The latent effect is a cost to senders, a benefit to receivers, and is the scope of the variable y. We assign an output $\mathsf{out}\ x\langle z\rangle$ the effect $e_\ell\{y{\leftarrow}z\}$, where we have instantiated the name y bound in the type of the channel with z, the name actually sent on the channel. We assign an input $\mathsf{inp}\ x(y{:}T); P$ the effect $e - e_\ell$, where e is the effect of P. To avoid scope confusion, we require that y is not free in $e - e_\ell$.

$$x : \mathsf{Ch}(y{:}T)e_\ell,\ z : T \quad \Rightarrow \quad \mathsf{out}\ x\langle z\rangle : e_\ell\{y{\leftarrow}z\}$$
$$x : \mathsf{Ch}(y{:}T)e_\ell,\ P : e,\ y \notin \mathsf{fn}(e - e_\ell) \quad \Rightarrow \quad \mathsf{inp}\ x(y{:}T); P : e - e_\ell$$

The formal rules for input and output in the next section generalize these rules to deal with polyadic channels.

Typing Conditionals When typing a conditional $\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$, it is useful to exploit the fact that P only runs if the two names x and y are equal. To do so, we check the effect of P after substituting one for the other. Suppose then process $P\{x{\leftarrow}y\}$ has effect $e_P\{x{\leftarrow}y\}$. Suppose also that process Q has effect e_Q. Let $e_P \vee e_Q$ be the least upper bound of any two effects e_P and e_Q. Then $e_P \vee e_Q$ is an upper bound on the begin-events needed to precede the conditional to make it safe, whether P or Q runs. An example in Section 4.2 illustrates this rule.

$$P\{x{\leftarrow}y\} : e_P\{x{\leftarrow}y\},\ Q : e_Q \quad \Rightarrow \quad \mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q : e_P \vee e_Q$$

3.4 Typing Rules

Our typing rules depend on several operations on effect multisets, most of which were introduced informally in the previous section. Here are the formal definitions.

Operations on effects: $e + e'$, $e \leq e'$, $e - e'$, $L \in e$, $e \vee e'$

$[L_1, \ldots, L_m] + [L_{m+1}, \ldots, L_{m+n}] \triangleq [L_1, \ldots, L_{m+n}]$
$e \leq e'$ if and only if $e' = e + e''$ for some e''
$e - e' \triangleq$ the smallest e'' such that $e \leq e' + e''$
$L \in e$ if and only if $[L] \leq e$
$e \vee e' \triangleq$ the smallest e'' such that $e \leq e''$ and $e' \leq e''$

The typing judgments of this section depend on an environment to assign a type to all the variables in scope.

Environments:

$E ::= \vec{x}{:}\vec{T}$	environment
$\mathsf{dom}(\vec{x}{:}\vec{T}) \triangleq \{\vec{x}\}$	domain of an environment

To equate two names in an environment, needed for typing conditionals, we define a name fusion function. We obtain the fusion $E\{x{\leftarrow}x'\}$ from E by turning all occurrences of x and x' in E into x'.

Fusing x with x' in E: $E\{x{\leftarrow}x'\}$

$$(x_1{:}T_1, \ldots, x_n{:}T_n)\{x{\leftarrow}x'\} \triangleq (x_1\{x{\leftarrow}x'\}){:}(T_1\{x{\leftarrow}x'\}); \ldots; (x_n\{x{\leftarrow}x'\}){:}(T_n\{x{\leftarrow}x'\})$$

$$\text{where } E; x{:}T \triangleq \begin{cases} E & \text{if } x \in \mathsf{dom}(E) \\ E, x{:}T & \text{otherwise} \end{cases}$$

The following table summarizes the five judgments of our type system, which are inductively defined by rules in subsequent tables. Judgment $E \vdash \diamond$ means environment E is well-formed. Judgment $E \vdash T$ means type T is well-formed. Judgment $E \vdash x : T$ means name x is in scope with type T. Judgment $E \vdash \langle \vec{x} \rangle : \langle \vec{y}{:}\vec{T} \rangle$ means tuple $\langle \vec{x} \rangle$ matches the variable declaration $\vec{y}{:}\vec{T}$. Judgment $E \vdash P : e$ means process P has effect e.

Judgments:

$E \vdash \diamond$	good environment
$E \vdash T$	good type T
$E \vdash x : T$	good name x of type T
$E \vdash \langle \vec{x} \rangle : \langle \vec{y}{:}\vec{T} \rangle$	good message $\vec{x}$ matching $\vec{y}{:}\vec{T}$
$E \vdash P : e$	good process P with effect e

The rules defining the first three judgments are standard.

Good environments, types, and names:

$$\frac{}{\varnothing \vdash \diamond}\ (\text{Env } \varnothing) \qquad \frac{E \vdash T \quad x \notin \mathsf{dom}(E)}{E, x{:}T \vdash \diamond}\ (\text{Env } x) \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{Name}}\ (\text{Type Name})$$

$$\frac{E, \vec{x}{:}\vec{T} \vdash \diamond \quad \mathsf{fn}(e) \subseteq \mathsf{dom}(E) \cup \{\vec{x}\}}{E \vdash \mathsf{Ch}(\vec{x}{:}\vec{T})e}\ (\text{Type Chan}) \qquad \frac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}\ (\text{Name } x)$$

The next judgment, $E \vdash \langle \vec{x} \rangle : \langle \vec{y}{:}\vec{T} \rangle$, is an auxiliary judgment used for typing output processes; it is used in the rule (Proc Output) to check that the message $\langle \vec{x} \rangle$ sent on a channel of type $\mathsf{Ch}(\vec{y}{:}\vec{T})e$ matches the variable declaration $\vec{y}{:}\vec{T}$.

Good message:

$$\frac{E \vdash \diamond}{E \vdash \langle\rangle : \langle\rangle}\ (\text{Msg } \langle\rangle) \qquad \frac{E \vdash \langle \vec{x} \rangle : \langle \vec{y}{:}\vec{T} \rangle \quad E \vdash x : (T\{\vec{y} \leftarrow \vec{x}\})}{E \vdash \langle \vec{x}, x \rangle : \langle \vec{y}{:}\vec{T}, y{:}T \rangle}\ (\text{Msg } x) \text{ (where } y \notin \{\vec{y}\} \cup \mathsf{dom}(E))$$

Finally, here are the rules for typing processes. The effect of a process is an upper bound; the rule (Proc Subsum) allows us to increase this upper bound. Intuitions for all the other rules were explained in the previous section.

Good processes:

$$\frac{E \vdash P : e}{E \vdash P : e'}\ (\text{Proc Subsum}) \text{ (where } e \leq e' \text{ and } \mathsf{fn}(e') \subseteq \mathsf{dom}(E))$$

$$\frac{E \vdash x : \mathsf{Ch}(\vec{y}{:}\vec{T})e \quad E \vdash \langle \vec{x} \rangle : \langle \vec{y}{:}\vec{T} \rangle}{E \vdash \mathsf{out}\ x\langle \vec{x} \rangle : (e\{\vec{y} \leftarrow \vec{x}\})}\ (\text{Proc Output})$$

$$\frac{E \vdash x : \mathsf{Ch}(\vec{y}{:}\vec{T})e' \quad E, \vec{y}{:}\vec{T} \vdash P : e}{E \vdash \mathsf{inp}\ x(\vec{y}{:}\vec{T}); P : e - e'}\ (\text{Proc Input}) \text{ (where } \mathsf{fn}(e - e') \subseteq \mathsf{dom}(E))$$

$$\frac{E \vdash x : T \quad E \vdash y : T \quad E\{x \leftarrow y\} \vdash P\{x \leftarrow y\} : e_P\{x \leftarrow y\} \quad E \vdash Q : e_Q}{E \vdash \mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q : e_P \vee e_Q}\ (\text{Proc Cond})$$

$$\frac{E, x{:}T \vdash P : e}{E \vdash \mathsf{new}(x{:}T); P : e}\ (\text{Proc Res}) \text{ (where } x \notin \mathsf{fn}(e)) \qquad \frac{E \vdash P : e_P \quad E \vdash Q : e_Q}{E \vdash P \mid Q : e_P + e_Q}\ (\text{Proc Par})$$

$$\frac{E \vdash P : [\,]}{E \vdash \mathsf{repeat}\ P : [\,]}\ \text{(Proc Repeat)} \qquad \frac{E \vdash \diamond}{E \vdash \mathsf{stop} : [\,]}\ \text{(Proc Stop)}$$

$$\frac{E \vdash P : e}{E \vdash \mathsf{begin}\ L; P : e - [L]}\ \text{(Proc Begin) (where } \mathsf{fn}(L) \subseteq \mathsf{dom}(E))$$

$$\frac{E \vdash P : e}{E \vdash \mathsf{end}\ L; P : e + [L]}\ \text{(Proc End) (where } \mathsf{fn}(L) \subseteq \mathsf{dom}(E))$$

Section 5 presents our main type safety result, Theorem 5.2, that $E \vdash P : [\,]$ implies P is safe. Like most type systems, ours is incomplete. There are safe processes that are not typeable in our system. For example, we cannot assign the process $\mathsf{if}\ x = x\ \mathsf{then}\ \mathsf{stop}\ \mathsf{else}\ (\mathsf{end}\ x; \mathsf{stop})$ the empty effect, and yet it is perfectly safe.

4 Applications

In this section, we present some examples of using correspondence assertions to validate safety properties of communication protocols. For more examples, including examples with cryptographic protocols which are secure against external attackers, see the companion paper [9].

4.1 Transmit-Acknowledge Handshake

Recall the untyped sender and receiver code from Section 2. Suppose we make the type definitions:

$$\begin{array}{rlrl} \mathit{Msg} & \triangleq \mathsf{Name} & \mathit{Ack}(a,b,\mathit{msg}) & \triangleq \mathsf{Ch}()[\langle a,b,\mathit{msg}\rangle] \\ \mathit{Host} & \triangleq \mathsf{Name} & \mathit{Req}(a,b) & \triangleq \mathsf{Ch}(\mathit{msg}{:}\mathit{Msg}, \mathit{ack}{:}\mathit{Ack}(a,b,\mathit{msg}))[\,] \end{array}$$

Suppose also that we annotate the sender and receiver code, and the code of Example 1 as follows:

$$
\begin{array}{ll}
\mathit{Snder}(a{:}\mathit{Host}, b{:}\mathit{Host}, c{:}\mathit{Req}(a,b)) \triangleq & \mathit{Rcver}(a{:}\mathit{Host}, b{:}\mathit{Host}, c{:}\mathit{Req}(a,b)) \triangleq \\
\quad \mathsf{new}(\mathit{msg}{:}\mathit{Msg}); & \quad \mathsf{inp}\ c(\mathit{msg}{:}\mathit{Msg}, \mathit{ack}{:}\mathit{Ack}(a,b,\mathit{msg})); \\
\quad \mathsf{new}(\mathit{ack}{:}\mathit{Ack}(a,b,\mathit{msg})); & \quad \mathsf{begin}\ \langle a, b, \mathit{msg}\rangle; \\
\quad \mathsf{out}\ c\langle \mathit{msg}, \mathit{ack}\rangle; & \quad \mathsf{out}\ \mathit{ack}\langle\rangle \\
\quad \mathsf{inp}\ \mathit{ack}(); & \\
\quad \mathsf{end}\ \langle a, b, \mathit{msg}\rangle &
\end{array}
$$

$$
\begin{array}{l}
\mathit{Example1}\,(a{:}\mathit{Host}, b{:}\mathit{Host}) \triangleq \\
\quad \mathsf{new}(c{:}\mathit{Req}(a,b)); \\
\quad\quad \mathit{Snder}(a,b,c) \mid \\
\quad\quad \mathit{Rcver}(a,b,c)
\end{array}
$$

We can then check that $a{:}\mathit{Host}, b{:}\mathit{Host} \vdash \mathit{Example1}\,(a,b) : [\,]$. Since the system has the empty effect, by Theorem 5.2 it is safe. It is routine to check that Example 2 from Section 2 also has the empty effect, but that Example 3 cannot be type-checked (as to be expected, since it is unsafe).

4.2 Hostname Lookup

In this example, we present a simple hostname lookup system, where a client b wishing to ping a server a can contact a name server *query*, to get a network address *ping* for a. The client can then send a ping request to the address *ping*, and get an acknowledgement from the server. We shall check two properties:

- When the ping client b finishes, it believes that the ping server a has been pinged.
- When the ping server a finishes, it believes that it was contacted by the ping client b.

We write "a was pinged by b" as shorthand for $\langle a, b\rangle$, and "b tried to ping a" for $\langle b, a, a\rangle$. These examples are well-typed, with types which we define later in this section.

We program the ping client and server as follows.

$$
\begin{array}{l}
\mathit{PingClient}(a{:}\mathit{Hostname}, b{:}\mathit{Hostname}, \mathit{query}{:}\mathit{Query}) \triangleq \\
\quad \mathsf{new}(\mathit{res} : \mathit{Res}(a)); \\
\quad \mathsf{out}\ \mathit{query}\langle a, \mathit{res}\rangle; \\
\quad \mathsf{inp}\ \mathit{res}(\mathit{ping} : \mathit{Ping}(a)); \\
\quad \mathsf{new}(\mathit{ack} : \mathit{Ack}(a, b)); \\
\quad \mathsf{begin}\ \text{“}b \text{ tried to ping } a\text{”}; \\
\quad \mathsf{out}\ \mathit{ping}\langle b, \mathit{ack}\rangle; \\
\quad \mathsf{inp}\ \mathit{ack}(); \\
\quad \mathsf{end}\ \text{“}a \text{ was pinged by } b\text{”}
\end{array}
$$

$$
\begin{array}{l}
\mathit{PingServer}(a : \mathit{Hostname}, \mathit{ping} : \mathit{Ping}(a)) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{ping}(b : \mathit{Hostname}, \mathit{ack} : \mathit{Ack}(a, b)); \\
\quad\quad \mathsf{begin}\ \text{“}a \text{ was pinged by } b\text{”}; \\
\quad\quad \mathsf{end}\ \text{“}b \text{ tried to ping } a\text{”}; \\
\quad\quad \mathsf{out}\ \mathit{ack}\langle\rangle
\end{array}
$$

If these processes are safe, then any ping request and response must come as matching pairs. In practice, the name server would require some data structure such as a hash table or database, but for this simple example we just use a large if-statement:

$$
\begin{array}{l}
\mathit{NameServer}(\\
\quad \mathit{query}{:}\mathit{Query}, \\
\quad h_1{:}\mathit{Hostname}, \ldots, h_n{:}\mathit{Hostname}, \\
\quad \mathit{ping}_1{:}\mathit{Ping}(h_1), \ldots, \mathit{ping}_n{:}\mathit{Ping}(h_n) \\
) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{query}(h, \mathit{res}); \\
\quad\quad \mathsf{if}\ h = h_1\ \mathsf{then\ out}\ \mathit{res}\langle \mathit{ping}_1\rangle\ \mathsf{else}\ \cdots \\
\quad\quad \mathsf{if}\ h = h_n\ \mathsf{then\ out}\ \mathit{res}\langle \mathit{ping}_n\rangle\ \mathsf{else\ stop}
\end{array}
$$

To get the system to type-check, we use the following types:

$$
\begin{array}{rcl}
\mathit{Hostname} & \triangleq & \mathsf{Name} \\
\mathit{Ack}(a, b) & \triangleq & \mathsf{Ch}()[\text{“}a \text{ was pinged by } b\text{”}] \\
\mathit{Ping}(a) & \triangleq & \mathsf{Ch}(b{:}\mathit{Hostname}, \mathit{ack}{:}\mathit{Ack}(a, b))[\text{“}b \text{ tried to ping } a\text{”}] \\
\mathit{Res}(a) & \triangleq & \mathsf{Ch}(\mathit{ping}{:}\mathit{Ping}(a))[\,] \\
\mathit{Query} & \triangleq & \mathsf{Ch}(a{:}\mathit{Hostname}, \mathit{res}{:}\mathit{Res}(a))[\,]
\end{array}
$$

The most subtle part of type-checking the system is the conditional in the name server. A typical branch is:

$$h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h)$$
$$\vdash \mathsf{if}\ h = h_i\ \mathsf{then}\ \mathsf{out}\ \mathit{res}\langle \mathit{ping}_i \rangle\ \mathsf{else}\ \cdots : [\,]$$

When type-checking the then-branch, (Proc Cond) assumes $h = h_i$ by applying a substitution to the environment:

$$(h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h))\{h \leftarrow h_i\}$$
$$= (h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), \mathit{res} : \mathit{Res}(h_i))$$

In this environment, we can type-check the then-branch:

$$h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), \mathit{res} : \mathit{Res}(h_i)$$
$$\vdash \mathsf{out}\ \mathit{res}\langle \mathit{ping}_i \rangle : [\,]$$

If (Proc Cond) did not apply the substitution to the environment, this example could not be type-checked, since:

$$h_i : \mathit{Hostname}, \mathit{ping}_i : \mathit{Ping}(h_i), h : \mathit{Hostname}, \mathit{res} : \mathit{Res}(h)$$
$$\nvdash \mathsf{out}\ \mathit{res}\langle \mathit{ping}_i \rangle : [\,]$$

4.3 Functions

It is typical to code the λ-calculus into the π-calculus, using a return channel k as the destination for the result. For instance, the hostname lookup example of the previous section can be rewritten in the style of a remote procedure call. The client and server are now:

$$\mathit{PingClient}(a{:}\mathit{Hostname}, b{:}\mathit{Hostname}, \mathit{query}{:}\mathit{Query}) \triangleq$$
$$\quad \mathsf{let}\ (\mathit{ping} : \mathit{Ping}(a)) = \mathit{query}\ \langle a \rangle;$$
$$\quad \mathsf{begin}\ \text{“}b \text{ tried to ping } a\text{”};$$
$$\quad \mathsf{let}\ () = \mathit{ping}\ \langle b \rangle;$$
$$\quad \mathsf{end}\ \text{“}a \text{ was pinged by } b\text{”}$$

$$\mathit{PingServer}(a : \mathit{Hostname}, \mathit{ping} : \mathit{Ping}(a)) \triangleq$$
$$\quad \mathsf{fun}\ \mathit{ping}(b{:}\mathit{Hostname})\ \{$$
$$\quad\quad \mathsf{begin}\ \text{“}a \text{ was pinged by } b\text{”};$$
$$\quad\quad \mathsf{end}\ \text{“}b \text{ tried to ping } a\text{”};$$
$$\quad\quad \mathsf{return}\ \langle\rangle$$
$$\quad \}$$

The name server is now:

$$
\begin{array}{l}
\mathit{NameServer}(\\
\quad \mathit{query}{:}\mathit{Query}, \\
\quad h_1{:}\mathit{Hostname}, \ldots, h_n{:}\mathit{Hostname}, \\
\quad \mathit{ping}_1{:}\mathit{Ping}(h_1), \ldots, \mathit{ping}_n{:}\mathit{Ping}(h_n) \\
) \triangleq \\
\quad \mathsf{fun}\ \mathit{query}(h{:}\mathit{Hostname})\ \{ \\
\qquad \mathsf{if}\ h = h_1\ \mathsf{then\ return}\ \langle \mathit{ping}_1 \rangle\ \mathsf{else}\ \cdots \\
\qquad \mathsf{if}\ h = h_n\ \mathsf{then\ return}\ \langle \mathit{ping}_n \rangle\ \mathsf{else\ stop} \\
\quad \}
\end{array}
$$

In order to provide types for these examples, we have to provide a function type with *latent effects*. These effects are *precondition/postcondition* pairs, which act like Hoare triples. In the type $(\vec{x}{:}\vec{T})e \to (\vec{y}{:}\vec{U})e'$ we have a precondition e which the callee must satisfy, and a postcondition e' which the caller must satisfy. For example, the types for the hostname lookup example are:

$$
\begin{array}{rl}
\mathit{Ping}(a) & \triangleq (b{:}\mathit{Hostname})[\text{“}b \text{ tried to ping } a\text{”}] \to ()[\text{“}a \text{ was pinged by } b\text{”}] \\
\mathit{Query} & \triangleq (a{:}\mathit{Hostname})[\,] \to (\mathit{ping}{:}\mathit{Ping}(a))[\,]
\end{array}
$$

which specifies that the remote ping call has a precondition “b tried to ping a” and a postcondition “a was pinged by b”.

This can be coded into the π-calculus using a translation [17] in continuation passing style.

$$
\begin{array}{rl}
\mathsf{fun}\ f(\vec{x}{:}\vec{T})\ \{P\} & \triangleq \mathsf{repeat\ inp}\ f(\vec{x}{:}\vec{T}, k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e'); P \\
\mathsf{let}\ (\vec{y}{:}\vec{U}) = f\ \langle \vec{x} \rangle; P & \triangleq \mathsf{new}(k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e'); \mathsf{out}\ f\langle \vec{x}, k \rangle; \mathsf{inp}\ k(\vec{y}{:}\vec{U}); P \\
\mathsf{return}\ \langle \vec{z} \rangle & \triangleq \mathsf{out}\ k\langle \vec{z} \rangle \\
(\vec{x}{:}\vec{T})e \to (\vec{y}{:}\vec{U})e' & \triangleq \mathsf{Ch}(\vec{x}{:}\vec{T}, k{:}\mathsf{Ch}(\vec{y}{:}\vec{U})e')e
\end{array}
$$

This translation is standard, except for the typing. It is routine to verify its soundness.

5 Formalizing Correspondence Assertions

In this section, we give the formal definition of the trace semantics for the π-calculus with correspondence assertions, which is used in the definition of a safe process. We then state the main result of this paper, which is that effect-free processes are safe.

We give the trace semantics as a labelled transition system. Following Berry and Boudol [3] and Milner [17] we use a structural congruence $P \equiv Q$, and give our operational semantics up to $\equiv$.

Structural Congruence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow \mathsf{inp}\ x(\vec{y}{:}\vec{T}); P \equiv \mathsf{inp}\ x(\vec{y}{:}\vec{T}); Q$	(Struct Input)
$P \equiv Q \Rightarrow \mathsf{new}(x{:}T); P \equiv \mathsf{new}(x{:}T); Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow \mathsf{repeat}\ P \equiv \mathsf{repeat}\ Q$	(Struct Repl)
$P \mid \mathsf{stop} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$\mathsf{repeat}\ P \equiv P \mid \mathsf{repeat}\ P$	(Struct Repl Par)
$\mathsf{new}(x{:}T); (P \mid Q) \equiv P \mid \mathsf{new}(x{:}T); Q$	(Struct Res Par) (where $x \notin \mathsf{fn}(P)$)
$\mathsf{new}(x_1{:}T_1); \mathsf{new}(x_2{:}T_2); P \equiv$ $\mathsf{new}(x_2{:}T_2); \mathsf{new}(x_1{:}T_1); P$	(Struct Res Res) (where $x_1 \neq x_2, x_1 \notin \mathsf{fn}(T_2), x_2 \notin \mathsf{fn}(T_1)$)

There are four actions in this labelled transition system:

- $P \xrightarrow{\mathsf{begin}\ L} P'$ when P reaches a $\mathsf{begin}\ L$ assertion.
- $P \xrightarrow{\mathsf{end}\ L} P'$ when P reaches an $\mathsf{end}\ L$ assertion.
- $P \xrightarrow{\mathsf{gen}\ \langle x \rangle} P'$ when P generates a new name x.
- $P \xrightarrow{\tau} P'$ when P can perform an internal action.

For example:

$$\begin{aligned}(\mathsf{new}(x{:}\mathsf{Name}); \mathsf{begin}\ \langle x \rangle; \mathsf{end}\ \langle x \rangle; \mathsf{stop}) &\xrightarrow{\mathsf{gen}\ \langle x \rangle} (\mathsf{begin}\ \langle x \rangle; \mathsf{end}\ \langle x \rangle; \mathsf{stop}) \\ &\xrightarrow{\mathsf{begin}\ \langle x \rangle} (\mathsf{end}\ \langle x \rangle; \mathsf{stop}) \\ &\xrightarrow{\mathsf{end}\ \langle x \rangle} (\mathsf{stop})\end{aligned}$$

Next, we define the syntax of actions α, and their free names and generated names.

Actions:

$\alpha, \beta ::=$	actions
$\mathsf{begin}\ L$	begin-event
$\mathsf{end}\ L$	end-event
$\mathsf{gen}\ \langle x \rangle$	name generation
τ	internal

Free names, $\mathsf{fn}(\alpha)$, and generated names, $\mathsf{gn}(\alpha)$, of an action α:

$\mathsf{fn}(\tau) \triangleq \varnothing$	$\mathsf{fn}(\mathsf{begin}\ L) \triangleq \mathsf{fn}(L)$	$\mathsf{fn}(\mathsf{end}\ L) \triangleq \mathsf{fn}(L)$	$\mathsf{fn}(\mathsf{gen}\ \langle x \rangle) \triangleq \{x\}$
$\mathsf{gn}(\tau) \triangleq \varnothing$	$\mathsf{gn}(\mathsf{begin}\ L) \triangleq \varnothing$	$\mathsf{gn}(\mathsf{end}\ L) \triangleq \varnothing$	$\mathsf{gn}(\mathsf{gen}\ \langle x \rangle \triangleq \{x\}$

The labelled transition system $P \xrightarrow{\alpha} P'$ is defined here.

Transitions: $P \xrightarrow{\alpha} P'$

$\mathsf{out}\ x\langle \vec{x} \rangle \mid \mathsf{inp}\ x(\vec{y}); P \xrightarrow{\tau} P\{\vec{y} \leftarrow \vec{x}\}$	(Trans Comm)
$\mathsf{if}\ x = x\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P$	(Trans Match)
$\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q$	(Trans Mismatch) (where $x \neq y$)
$\mathsf{begin}\ L; P \xrightarrow{\mathsf{begin}\ L} P$	(Trans Begin)
$\mathsf{end}\ L; P \xrightarrow{\mathsf{end}\ L} P$	(Trans End)
$\mathsf{new}(x{:}T); P \xrightarrow{\mathsf{gen}\ \langle x \rangle} P$	(Trans Gen)
$P \xrightarrow{\alpha} P' \Rightarrow P \mid Q \xrightarrow{\alpha} P' \mid Q$	(Trans Par) (where $\mathsf{gn}(\alpha) \cap \mathsf{fn}(Q) = \varnothing$)
$P \xrightarrow{\alpha} P' \Rightarrow \mathsf{new}(x{:}T); P \xrightarrow{\alpha} \mathsf{new}(x{:}T); P'$	(Trans Res) (where $x \notin \mathsf{fn}(\alpha)$)
$P \equiv P', P' \xrightarrow{\alpha} Q', Q' \equiv Q \Rightarrow P \xrightarrow{\alpha} Q$	(Trans $\equiv$)

From this operational semantics, we can define the traces of a process, with reductions $P \xrightarrow{s} P'$ where s is a sequence of actions.

Traces:

$s, t ::= \alpha_1, \ldots, \alpha_n$	trace

Free names, $\mathsf{fn}(s)$, and generated names, $\mathsf{gn}(s)$, of a trace s:

$$\mathsf{fn}(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{fn}(\alpha_1) \cup \cdots \cup \mathsf{fn}(\alpha_n)$$
$$\mathsf{gn}(\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{gn}(\alpha_1) \cup \cdots \cup \mathsf{gn}(\alpha_n)$$

Traced transitions: $P \xrightarrow{s} P'$

$P \equiv P' \Rightarrow P \xrightarrow{\varepsilon} P'$	(Trace $\equiv$)
$P \xrightarrow{\alpha} P'', P'' \xrightarrow{s} P' \Rightarrow P \xrightarrow{\alpha, s} P'$	(Trace Action) (where $\mathsf{fn}(\alpha) \cap \mathsf{gn}(s) = \varnothing$)

We require a side-condition on (Trace Action) to ensure that generated names are unique, otherwise we could observe traces such as

$$(\mathsf{new}(x); \mathsf{new}(y); \mathsf{stop}) \xrightarrow{\mathsf{gen}\ \langle x \rangle, \mathsf{gen}\ \langle x \rangle} (\mathsf{stop})$$

Having formally defined the trace semantics of our π-calculus, we can define when a trace is a correspondence: this is when every $\mathsf{end}\ L$ has a distinct,

matching $\mathsf{begin}\ L$. For example:

$$\begin{array}{rl} \mathsf{begin}\ L, \mathsf{end}\ L & \text{is a correspondence} \\ \mathsf{begin}\ L, \mathsf{end}\ L, \mathsf{end}\ L & \text{is not a correspondence} \\ \mathsf{begin}\ L, \mathsf{begin}\ L, \mathsf{end}\ L, \mathsf{end}\ L & \text{is a correspondence} \end{array}$$

We formalize this by counting the number of $\mathsf{begin}\ L$ and $\mathsf{end}\ L$ actions there are in a trace.

Beginnings, $\mathsf{begins}\ (\alpha)$, and endings, $\mathsf{ends}\ (\alpha)$, of an action α:

$$\begin{array}{rlrl} \mathsf{begins}\ (\mathsf{begin}\ L) & \triangleq [L] & \mathsf{ends}\ (\mathsf{begin}\ L) & \triangleq [\,] \\ \mathsf{begins}\ (\mathsf{end}\ L) & \triangleq [\,] & \mathsf{ends}\ (\mathsf{end}\ L) & \triangleq [L] \\ \mathsf{begins}\ (\mathsf{gen}\ \langle x \rangle) & \triangleq [\,] & \mathsf{ends}\ (\mathsf{gen}\ \langle x \rangle) & \triangleq [\,] \\ \mathsf{begins}\ (\tau) & \triangleq [\,] & \mathsf{ends}\ (\tau) & \triangleq [\,] \end{array}$$

Beginnings, $\mathsf{begins}\ (s)$, and endings, $\mathsf{ends}\ (s)$, of a trace s:

$$\begin{array}{l} \mathsf{begins}\ (\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{begins}\ (\alpha_1) + \cdots + \mathsf{begins}\ (\alpha_n) \\ \mathsf{ends}\ (\alpha_1, \ldots, \alpha_n) \triangleq \mathsf{ends}\ (\alpha_1) + \cdots + \mathsf{ends}\ (\alpha_n) \end{array}$$

Correspondence:

A trace s is a *correspondence* if and only if $\mathsf{ends}\ (s) \leq \mathsf{begins}\ (s)$.

A process is safe if every trace is a correspondence.

Safety:

A process P is *safe* if and only if for all traces s and processes P'
if $P \xrightarrow{s} P'$ then s is a correspondence.

A subtlety of this definition of safety is that although we want each end-event of a safe process to be preceded by a distinct, matching begin-event, a trace st may be a correspondence by virtue of a later begin-event in t matching an earlier end-event in s. For example, a trace like $\mathsf{end}\ L, \mathsf{begin}\ L$ is a correspondence.

To see why our definition implies that a matching begin-event must precede each end-event in each trace of a safe process, suppose a safe process has a trace $s, \mathsf{end}\ L, t$. By definition of traces, the process also has the shorter trace $s, \mathsf{end}\ L$, which must be a correspondence, since it is a trace of a safe process. Therefore, the end-event $\mathsf{end}\ L$ is preceded by a matching begin-event in s.

We can now state the formal result of the paper, Theorem 5.2, that every effect-free process is safe. This gives us a compositional technique for verifying the safety of communications protocols. It follows from a subject reduction result, Theorem 5.1. The most difficult parts of the formal development to check in detail are the parts associated with the (Proc Cond) rule, because of

its use of a substitution applied to an environment.

Theorem 5.1 (Subject Reduction) *Suppose* $E \vdash P : e$.

(1) *If* $P \xrightarrow{\tau} P'$ *then* $E \vdash P' : e$.

(2) *If* $P \xrightarrow{\mathsf{begin}\, L} P'$ *then* $E \vdash P' : e + [L]$.

(3) *If* $P \xrightarrow{\mathsf{end}\, L} P'$ *then* $E \vdash P' : e - [L]$, *and* $L \in e$.

(4) *If* $P \xrightarrow{\mathsf{gen}\, \langle x \rangle} P'$ *and* $x \notin \mathsf{dom}(E)$ *then* $E, x{:}T \vdash P' : e$ *for some type* T.

Theorem 5.2 (Safety) *If* $E \vdash P : [\,]$ *then* P *is safe.*

6 Related Work

Correspondence assertions are not new; we have already discussed prior work on correspondence assertions for cryptographic protocols [23,16]. A contribution of our work is the idea of directly expressing correspondence assertions by adding annotations to a general concurrent language, in our case the π-calculus.

Gifford and Lucassen introduced type and effect systems [10,15] to manage side-effects in functional programming. There is a substantial literature; recent applications include memory management for high-level [22] and low-level [5] languages, race-condition avoidance [7], and access control [20].

Early type systems for the π-calculus [17,19] focus on regulating the data sent on channels. Subsequent type systems also regulate process behaviour; for example, session types [21,11] regulate pairwise interactions and linear types [14] help avoid deadlocks. A recent paper [6] explicitly proposes a type and effect system for the π-calculus, and the idea of latent effects on channel types. This idea can also be represented in a recent general framework for concurrent type systems [13]. Still, the types of our system are dependent in the sense that they may include the names of channels; to the best of our knowledge, this is the first dependent type system for the π-calculus. Another system of dependent types for a concurrent language is Flanagan and Abadi's system [7] for avoiding race conditions in the concurrent object calculus of Gordon and Hankin [8].

The rule (Proc Cond) for typing name equality $\mathsf{if}\ x = y\ \mathsf{then}\ P\ \mathsf{else}\ Q$ checks P under the assumption that the names x and y are the same; we formalize this by substituting y for x in the type environment and the process P. Given that names are the only kind of value, this technique is simpler than the standard technique from dependent type theory [18,2] of defining typing judgments with respect to an equivalence relation on values. Honda, Vasconcelos, and Yoshida [12] also use the technique of applying substitutions to environments while type-checking.

7 Conclusions

The long term objective of this work is to check secrecy and authenticity properties of security protocols by typing. This paper introduces several key ideas in the minimal yet general setting of the π-calculus: the idea of expressing correspondences by begin- and end-annotations, the idea of a dependent type and effect system for proving correspondences, and the idea of using latent effects to type correspondences begun by one process and ended by another. Several examples demonstrate the promise of this system. Unlike a previous approach based on model-checking, type-checking correspondence assertions is not limited to finite-state systems.

A companion paper [9] begins the work of applying these ideas to cryptographic protocols as formalized in Abadi and Gordon's spi-calculus [1], and has already proved useful in identifying known issues in published protocols. Our first type system for spi is specific to cryptographic protocols based on symmetric key cryptography. Instead of attaching latent effects to channel types, as in this paper, we attach them to a new type for nonces, to formalize a specific idiom for preventing replay attacks. Another avenue for future work is type inference algorithms.

The type system of the present paper has independent interest. It introduces the ideas in a more general setting than the spi-calculus, and shows in principle that correspondence assertions can be type-checked in any of the many programming languages that may be reduced to the π-calculus.

Acknowledgements We had useful discussions with Andrew Kennedy and Naoki Kobayashi. Tony Hoare commented on a draft of this paper. Alan Jeffrey was supported in part by Microsoft Research during some of the time we worked on this paper.

References

[1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[2] H. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume II*. Oxford University Press, 1992.

[3] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[4] E. Clarke and W. Marrero. Using formal methods for analyzing security. Available at `http://www.cs.cmu.edu/~marrero/abstract.html`, 2000.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM Symposium on Principles of Programming Languages*, pages 262–275, 1999.

[6] S. Dal Zilio and A.D. Gordon. Region analysis and a π-calculus with groups. In *Mathematical Foundations of Computer Science 2000 (MFCS2000)*, volume 1893 of *Lectures Notes in Computer Science*, pages 1–21. Springer, 2000.

[7] C. Flanagan and M. Abadi. Object types against races. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99: Concurrency Theory*, volume 1664 of *Lectures Notes in Computer Science*, pages 288–303. Springer, 1999.

[8] A.D. Gordon and P.D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*, ENTCS. Elsevier, 1998.

[9] A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001. To appear.

[10] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[11] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, volume 1381 of *Lectures Notes in Computer Science*, pages 122–128. Springer, 1998.

[12] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *European Symposium on Programming*, Lectures Notes in Computer Science. Springer, 2000.

[13] A. Igarashi and N. Kobayashi. A generic type system for the pi calculus. In *28th ACM Symposium on Principles of Programming Languages*, pages 128–141, 2001.

[14] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20:436–482, 1998.

[15] J.M. Lucassen. *Types and effects, towards the integration of functional and imperative programming*. PhD thesis, MIT, 1987. Available as Technical Report MIT/LCS/TR–408, MIT Laboratory for Computer Science.

[16] W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Preliminary version appears as Technical Report TR–CMU–CS–97–139, Carnegie Mellon University, May 1997.

[17] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[18] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990.

[19] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[20] C. Skalka and S. Smith. Static enforcement of security with types. In P. Wadler, editor, *2000 ACM International Conference on Functional Programming*, pages 34–45, 2000.

[21] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *Proceedings 6th European Conference on Parallel Languages and Architecture*, volume 817 of *Lectures Notes in Computer Science*, pages 398–413. Springer, 1994.

[22] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[23] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

MFPS 17 Preliminary Version

Pseudo-commutative Monads

Martin Hyland

Dept of Pure Mathematics and Mathematical Statistics
University of Cambridge
Wilberforce Road, Cambridge, ENGLAND

and

John Power [1]

Laboratory for the Foundations of Computer Science
University of Edinburgh
King's Buildings, Edinburgh EH9 3JZ, SCOTLAND

Abstract

We introduce the notion of pseudo-commutative monad together with that of pseudo-closed 2-category, the leading example being given by the 2-monad on Cat whose 2-category of algebras is the 2-category of small symmetric monoidal categories. We prove that for any pseudo-commutative 2-monad on Cat, its 2-category of algebras is pseudo-closed. We also introduce supplementary definitions and results, and we illustrate this analysis with further examples such as those of small categories with finite products, and examples arising from wiring, interaction, contexts, and the logic of Bunched Implication.

1 Introduction

Symmetric monoidal categories, often with a little extra structure and subject to some extra axioms, such as those required to make symmetric monoidal structure into finite product or finite coproduct structure, play a fundamental foundational role in much of theoretical computer science. For instance, they have long been used to model contexts, typically but not only when in the form of finite product structure (see for instance [4] and, especially relevant here, [5]). They have also long been used to model a parallel operator (see for instance [9]) or interaction [1]. Occasionally, one sees two symmetric monoidal

[1] This work is supported by EPSRC grant GR/M56333: The structure of programming languages : syntax and semantics, and a British Council grant and the COE budget of STA Japan.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

structures interacting with each other, for instance in work on linear logic or more recently on Bunched Implication [11]. Several delicate constructions are made using symmetric monoidal structure. For instance, one often considers the free symmetric monoidal category, possibly with additional structure, on 1, and one sometimes sees study of the free symmetric monoidal closed category on a symmetric monoidal category. One also sees constructions on categories possessing a pair of symmetric monoidal structures as in Bunched Implication.

This all motivates us to seek a calculus of symmetric monoidal categories, possibly with a little extra structure subject to mild axioms as illustrated above. By a calculus, we mean a mathematical account of what constructions one can make on symmetric monoidal categories and still obtain a symmetric monoidal category. For instance, it is routine to verify that a product of symmetric monoidal categories is symmetric monoidal. Formally, such a calculus amounts to study of the structure of the 2-category $SymMon$ of small symmetric monoidal categories and strong symmetric monoidal functors. It has long been known that this is an instance of algebraic structure on Cat [2] and therefore has well-behaved limits and bicolimits, in particular products and bicoproducts for example. But is the 2-category $SymMon$, or at least the variant $SymMon_s$ of small symmetric monoidal categories and strict symmetric monoidal functors, itself a symmetric monoidal category? And is there an axiomatic proof of such a result that would apply to variants of the notion of small symmetric monoidal category such as that of small category with finite products? Positive answers would substantially increase the range of constructions available for use: for instance, considering the free structure on 1 as for example in [5], implicit is the idea that structure on C, which is isomorphic to $Cat(1, C)$, lifts to structure on the category of structure preserving functors from $F(1)$ to C.

There is good reason to hope that the answers to these questions might be positive. A small symmetric monoidal category is, except for some isomorphisms rather than equalities, a commutative monoid in the category Cat, And the category of commutative monoids, $CMon$, in Set, is a symmetric monoidal closed category, the reason being that the monad T on Set for which $CMon$ is isomorphic to T-Alg is a commutative monad (the notion of commutative monad appearing in theoretical computer science in work such as that of Moggi on computational effects [10]), and for any commutative monad T, the category T-Alg is symmetric monoidal closed, with the adjunction between T-Alg and Set being a symmetric monoidal adjunction.

In fact, there is a monad T on Cat for which the category T-Alg is isomorphic to the category of small symmetric monoidal categories and strict symmetric monoidal functors, and that monad has a unique strength. However, that strength is not commutative, the reason being that at precisely one point where one requires an equality, one has an isomorphism. And consequently, $SymMon_s$ is not symmetric monoidal closed. But the 2-category $SymMon$ does have a structure that is a mild weakening of closed structure,

and we can prove that result axiomatically, with axioms that hold equally of the 2-category of small categories with finite products and of variants. So this paper is devoted to spelling out what that mild 2-categorical generalisation of closed structure is, what the corresponding generalisation of the notion of commutative monad is, and giving the proof that for every pseudo-commutative monad on Cat, the 2-category of algebras and pseudo-maps of algebras is pseudo-closed.

Inevitably, with the complexity of coherence required for our definitions, we must be very sketchy with detail for a short conference paper. But much more detail appears in [6]. A definition provably (with considerable effort) equivalent to one we have here was introduced by Max Kelly in [7], but, as he recognised at the time, his axioms were too complicated to be definitive.

The paper is organised very simply: we define the notions of pseudo-commutativity and symmetry for a pseudo-commutativity, given a 2-monad on Cat, and present our leading example, in Section 2; we define the notion of pseudo-closedness in Section 3; and we outline a proof that T-Alg is pseudo-closed if T has a pseudo-commutativity in Section 4.

2 Pseudo-commutativity for a 2-monad

We refer the reader to [2] for 2-categorical terminology: unfortunately, there is not space to include much of it here. Let T be a 2-monad on Cat, for instance the 2-monad for small symmetric strict monoidal categories. Then T possesses a unique strength

$$t_{A,B} : A \times TB \longrightarrow T(A \times B)$$

and, by symmetry, a unique costrength

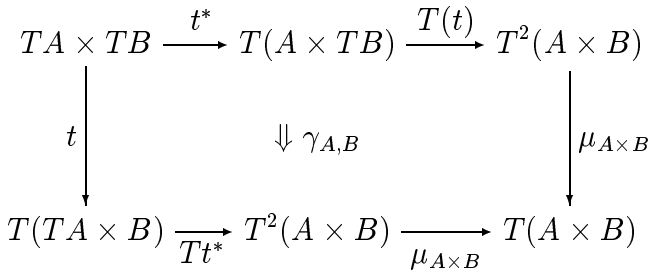
$$t^*_{A,B} : TA \times B \longrightarrow T(A \times B)$$

The 2-functorial behaviour of T corresponds to t via commutativity of

$$\begin{array}{ccc} A & \xrightarrow{in} & [B, A \times B] \\ {\scriptstyle in}\downarrow & & \downarrow{\scriptstyle T} \\ [TB, A \times TB] & \xrightarrow[[TB,t]]{} & [TB, T(A \times B)] \end{array} \qquad \begin{array}{ccc} [A,B] \times TA & \xrightarrow{t} & T([A,B] \times A) \\ {\scriptstyle T \times TA}\downarrow & & \downarrow{\scriptstyle Tev} \\ [TA, TB] \times TA & \xrightarrow[ev]{} & TB \end{array}$$

Definition 2.1 A *pseudo-commutativity* for a 2-monad (T, μ, η) is an isomor-

phic modification



such that the following three strength axioms, two η axioms and two μ axioms hold.

(i) $\gamma_{A\times B,C}\cdot(t_{A,B}\times TC) = t_{A,B\times C}\cdot(A\times\gamma_{B,C})$

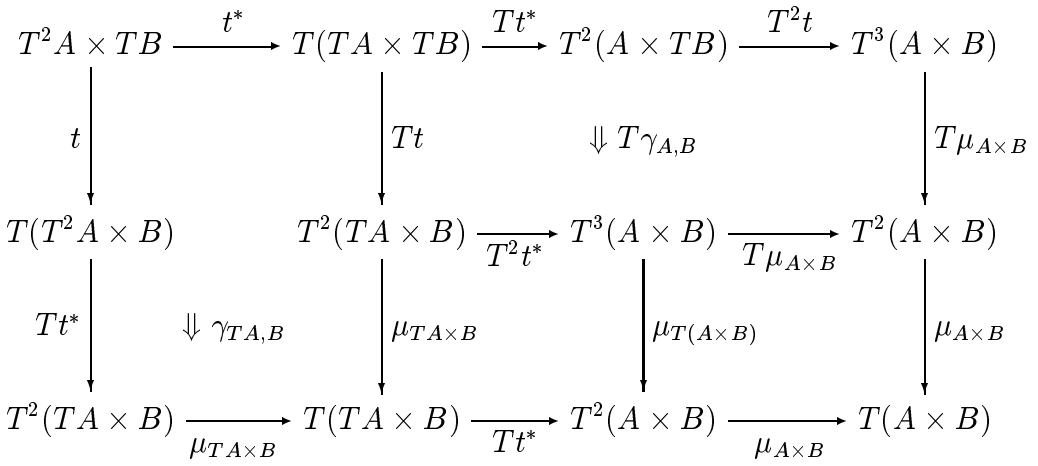
(ii) $\gamma_{A,B\times C}\cdot(TA\times t_{B,C}) = \gamma_{A\times B,C}\cdot(t^*_{A,B}\times TC)$

(iii) $\gamma_{A,B\times C}\cdot(TA\times t^*_{B,C}) = t^*_{A\times B,C}\cdot(\gamma_{A,B}\times C)$

(iv) $\gamma_{A,B}\cdot(\eta_A\times TB)$ is an identity modification

(v) $\gamma_{A,B}\cdot(TA\times\eta_B)$ is an identity modification

(vi) $\gamma_{A,B}\cdot(\mu_A\times TB)$ is equal to the pasting



(vii) the dual of the above μ axiom

There is a little redundancy here, as follows.

Proposition 2.2 *Any two of the strength axioms implies the third.*

If the modification γ were an identity, T would be a commutative 2-monad [7,8] and the axioms would all be redundant. But in our leading example, where T is the 2-monad on Cat for symmetric strict monoidal categories, γ is not an identity but rather is determined by a non-trivial symmetry. We shall soon spell out that example in detail, but first we introduce a further symmetry condition on a pseudo-commutativity: we do not use this condition for our main results, but it simplifies analysis of the examples and we believe

it will be useful in practice, for example in relation to Bunched Implication [11], as we shall explain below.

Definition 2.3 A pseudo-commutativity γ is *symmetric* when $Tc_{A,B} \cdot \gamma_{A,B} \cdot c_{TB,TA}$ is the inverse of $\gamma_{B,A}$.

The simplification that this definition allows is given by the following proposition.

Proposition 2.4 *An isomorphic modification γ as above is a symmetric pseudo-commutativity if the symmetry axiom, one strength axiom, one η axiom, and one μ axiom hold.*

Finally, we spell out our leading example in detail. Most of the other examples, which we list afterwards, work similarly.

Example 2.5 Let T be the 2-monad for symmetric strict monoidal categories.

- Given a category A, the category TA has as objects sequences

$$a_1 \dots a_n$$

of objects of A (with maps generated by symmetries and the maps of A); the tensor product is concatenation.

- Given two categories A and B, the category $TA \times TB$ has as objects pairs

$$((a_1 \dots a_n), (b_1 \dots b_m))$$

and the two maps $TA \times TB \longrightarrow T(A \times B)$ take such pairs to the sequences of all (a_i, b_j) ordered according to the two possible lexicographic orderings. In fact

$$TA \times TB \xrightarrow{t^*} T(TA \times B) \xrightarrow{T(t)} T^2(A \times B) \xrightarrow{\mu_{A\times B}} T(A \times B)$$

gives the ordering

$$(a_1, b_1), (a_1, b_2), \dots$$

in which the first coordinate takes precedence, while

$$TA \times TB \xrightarrow{t} T(TA \times B) \xrightarrow{T(t^*)} T^2(A \times B) \xrightarrow{\mu_{A\times B}} T(A \times B)$$

gives the ordering

$$(a_1, b_1), (a_2, b_1), \dots$$

in which the second coordinate takes precedence.

- The component $\gamma_{A,B}$ of the modification is given by the unique symmetry mediating between the two lexicographic orders.

We now indicate the force of our various axioms as they appear here.

- The strength axioms concern the various lexicographic orderings of the sequences (a_i, b_j, c_k) where again there is just one a_i (or b_j or c_k). Various orderings are identified and as a result there are in each case prima facie two processes for mediating between the orderings: these are equal. So the axioms reflect the fact that there is a unique way to mediate between a pair of orderings.
- The η axioms express the fact that the two lexicographic orderings of the (a_i, b_j) are equal if one of n or m is 1.
- The μ axioms take more explaining. Take a sequence $a^1, \ldots, a^n$ of sequences $a^i_1, \ldots a^i_{m(i)}$. Concatenation gives a sequence a^i_j where the order is determined by the precedence (i, j): that is, i takes precedence over j. Take this concatenated sequence together with a sequence $b_1, \ldots, b_p$. Then $\gamma_{A,B} \cdot (\mu_A \times TB)$ mediates between the order on the (a^i_j, b_k) with precedence (i, j, k) and that with precedence (k, i, j). However we can also use $\mu \cdot T\gamma_{A,B} \cdot t^*$ to mediate between the orders determined by (i, j, k) and (i, k, j), and use $\mu \cdot Tt^* \cdot \gamma_{TA,B}$ to mediate between the orders determined by (i, k, j) and (k, i, j). Composing these two gives the first. So again the axioms reflect the fact that there is a unique way to mediate between a pair of orderings.
- The symmetry axiom just says that if you swap the order twice, you return to where you began.

Further examples of symmetric pseudo-commutative monads, for which we shall not spell out the details, are given by those for

(i) Symmetric monoidal categories.

(ii) Categories with strictly associative finite products. (Categories with strictly associative finite coproducts.)

(iii) Categories with finite products. (Categories with finite coproducts.)

(iv) Categories with an action of a symmetric strictly associative monoidal category.

(v) Symmetric strict monoidal categories with a strict monoidal endofunctor.

(vi) Symmetric monoidal categories with a strong monoidal endofunctor.

These examples are used widely for modelling contexts, or parallelism, or interaction in computer science [1,4,5,9], and one can build combinations as used in [11] or variants. In more detail, finite products are used extensively for modelling contexts, for instance in [4]. A subtle combination of finite products and symmetric monoidal structure is used to model parallelism in [9]. And symmetric monoidal structure is used to model interaction in [1]. And in current research, Plotkin is using a category with an action of a symmetric monoidal category to model call-by-name and call-by-value, along the lines of symmetric premonoidal categories being represented as the action of

a symmetric monoidal category on a category [12]. For a non-example of the symmetry condition, we believe that there is a natural pseudo-commutativity on the 2-monad for braided monoidal categories which is not symmetric.

We can prove that our definition of symmetric pseudo-commutativity implies that adumbrated by Kelly in [7], which tells us

Theorem 2.6 *If T is a symmetric pseudo-commutative monad on Cat, then T lifts to a 2-monad on the 2-category $SymMon$ of small symmetric monoidal categories and strong symmetric monoidal functors.*

This result seems likely to relate to Bunched Implication [11], where the underlying first order structure involves a symmetric monoidal category, so an object of $SymMon$, that possesses finite products, so has T-structure for the symmetric pseudo-commutative monad for small categories with finite products. We do not immediately have a more direct relationship with linear logic, as the latter involves a comonad !, and the 2-category of small categories equipped with a comonad is not an example of the 2-category of algebras for a pseudo-commutative 2-monad.

3 Pseudo-closed 2-categories

In this section, we define the notion of a pseudo-closed 2-category.

Definition 3.1 A *pseudo-closed* 2-category consists of a 2-category $\mathcal{K}$, a 2-functor

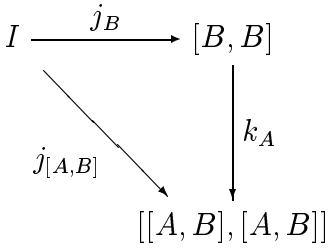
$$[-,-] : \mathcal{K}^{op} \times \mathcal{K} \longrightarrow \mathcal{K}$$

and a 2-functor $V : \mathcal{K} \longrightarrow Cat$, together with an object I of $\mathcal{K}$ and transformations j, e, i, k, with components

- $j_A : I \longrightarrow [A, A]$ pseudo-dinatural in A,
- $e_A : [I, A] \longrightarrow A$ natural in A, and $i_A : A \longrightarrow [I, A]$ pseudo-natural in A,
- $k_{A,B,C} : [B, C] \longrightarrow [[A, B], [A, C]]$ natural in B and C and dinatural in A,

such that $V[-,-] = \mathcal{K}(-,-) : \mathcal{K}^{op} \times \mathcal{K} \longrightarrow Cat$, e and i form a retract equivalence, and

(i)



(ii)

$$\begin{array}{ccc} [A,C] & \xrightarrow{k_A} & [[A,A],[A,C]] \\ \| & & \downarrow{\scriptstyle [j_A,[A,C]]} \\ [A,C] & \xleftarrow[e_{[A,C]}]{} & [I,[A,C]] \end{array}$$

(iii)

$$\begin{array}{ccccc} [C,D] & \xrightarrow{k_A} & [[A,C],[A,D]] & \xrightarrow{k_{[A,B]}} & [[[A,B],[A,C]],[[A,B],[A,D]]] \\ \downarrow{\scriptstyle k_B} & & & & \downarrow{\scriptstyle [k_A,[[A,B],[A,D]]]} \\ [[B,C],[B,D]] & & \xrightarrow[[[B,C],k_A]]{} & & [[B,C],[[A,B],[A,D]]] \end{array}$$

(iv)

$$\begin{array}{ccc} [A,B] & \xrightarrow{k_I} & [[I,A],[I,B]] \\ & \searrow^{[e_A,B]} & \downarrow{\scriptstyle [[I,A],e_B]} \\ & & [[I,A],B] \end{array}$$

(v) The map

$$\mathcal{K}(A,A) = V[A,A] \longrightarrow V[I,[A,A]] = \mathcal{K}(I,[A,A])$$

induced by $i_{[A,A]}$ takes 1_A to j_A.

We compare this definition with that of closed category in [3], where the theory of enriched categories was introduced. Its primary definition was that of a closed category; it then defined monoidal closed categories and proceeded from there. The only reason more modern accounts start with the notion of monoidal category is because it is first order structure: but the closed structure is typically more primitive.

Given our aims, we ask for 2-categories, 2-functors, and 2-natural or 2-dinatural transformations where [3] drops the prefix 2: there is one significant case of pseudo-naturality. Moreover, as $\mathcal{K}(-,-)$ is a 2-functor into Cat, the codomain for V should be Cat rather than Set as in [3].

Allowing for these changes, our five enumerated conditions correspond to Eilenberg and Kelly's five axioms. The fact that e is a retract equivalence

rather than an isomorphism as in [3] is significant. We have no choice if we are to include our leading example: one might hope that the 2-category of small symmetric monoidal categories would have invertible e, but it does not; and because e is not an isomorphism, we do not have the Eilenberg and Kelly versions of conditions 2 and 4 which are expressed in terms of i; and those conditions would fail in our leading example. Moreover i is only pseudo-natural in examples. We note that we are able to give our restricted definition so that T-Alg will be an example where all the structure maps other than i_A are strict maps of T-algebras.

This is not the most general possible notion of pseudo-closedness. Even Eilenberg and Kelly could have asked for an isomorphism between $V[-,-]$ and $\mathcal{K}(-,-)$: their choice of equality means that a monoidal category subject to the usual adjointness condition need not be closed in their sense. But our examples allow us considerable strictness, so we take advantage of that to provide a relatively simple definition.

On the other hand, it does not contain all axioms that hold of our class of examples either. In particular, our pseudo-natural transformation i and our pseudo-dinatural transformation j satisfy strictness conditions along the lines that, for some specific classes of maps, the isomorphism given by pseudo-naturality is in fact an identity. However, at present, we have no theorems that make use of such facts, and adding them to the definition would complicate rather than simplify it, so we have not introduced them as axioms.

4 Pseudo-closed structure on T-Alg

We consider the 2-category T-Alg of strict T-algebras and pseudo-maps of T-algebras as developed in [2], for a 2-monad T on Cat. We can readily generalise beyond Cat, but this contains the examples of primary interest to us: the 2-category of small symmetric monoidal categories and strong symmetric monoidal functors is an example, as is the category of small categories with finite products and finite product preserving functors, etcetera. We write $\mathcal{A} = (A, a)$ for a typical T-algebra. A *pseudo-map* $(f, \bar{f}) : \mathcal{A} \longrightarrow \mathcal{B}$ is given by data

$$\begin{array}{ccc} TA & \xrightarrow{Tf} & TB \\ {\scriptstyle a}\downarrow & \Downarrow \bar{f} & \downarrow{\scriptstyle b} \\ A & \xrightarrow[f]{} & B \end{array}$$

where the isomorphic 2-cell $\bar{f}$ satisfies η and μ conditions. We often write $f = (f, \bar{f}) : \mathcal{A} \longrightarrow \mathcal{B}$ for such a pseudo-map, the 2-cell usually being understood.

Given a pseudo-commutativity for T, we show that for any T-algebras $\mathcal{A}$

and $\mathcal{B}$, the category $T\text{-}Alg(\mathcal{A},\mathcal{B})$ has a T-algebra structure defined pointwise, i.e., it inherits a T-algebra structure from the cotensor, i.e., from the functor category $[A,\mathcal{B}]$ with pointwise T-structure.

In order to express the definition, we recall two sorts of limits in 2-categories. Given a pair of parallel 2-cells $f,g : X \longrightarrow Y$ in a 2-category $\mathcal{K}$, the *iso-inserter* of f and g consists of the universal 1-cell $i : I \longrightarrow X$ and isomorphic 2-cell $\gamma : fi \Rightarrow gi$, universally inserting an isomorphism between f and g. Given parallel 2-cells $\alpha,\beta : f \Rightarrow g : X \longrightarrow Y$, the *equifier* of α and β is the universal 1-cell $e : E \longrightarrow X$ making $\alpha e = \beta e$.

Proposition 4.1 *[2] For any 2-monad T on Cat, the 2-category $T\text{-}Alg$ has and the forgetful 2-functor $U : T\text{-}Alg \longrightarrow Cat$ preserves iso-inserters and equifiers.*

It is routine to describe iso-inserters and equifiers in Cat by considering their universal properties as they apply to functors with domain 1. With these definitions, we can define the pseudo-closed structure of $T\text{-}Alg$ for pseudo-commutative T.

Definition 4.2 Given T-algebras $\mathcal{A} = (A,a)$ and $\mathcal{B} = (B,b)$, we construct a new T-algebra in three steps.

(i) Take the iso-inserter $(i : In \longrightarrow [A,\mathcal{B}],\alpha')$ of

$$[A,\mathcal{B}] \underset{[a,\mathcal{B}]}{\overset{\sigma_{A,\mathcal{B}}}{\rightrightarrows}} [TA,\mathcal{B}]$$

where the underlying 1-cell of $\sigma_{A,\mathcal{B}}$ is defined by the composite

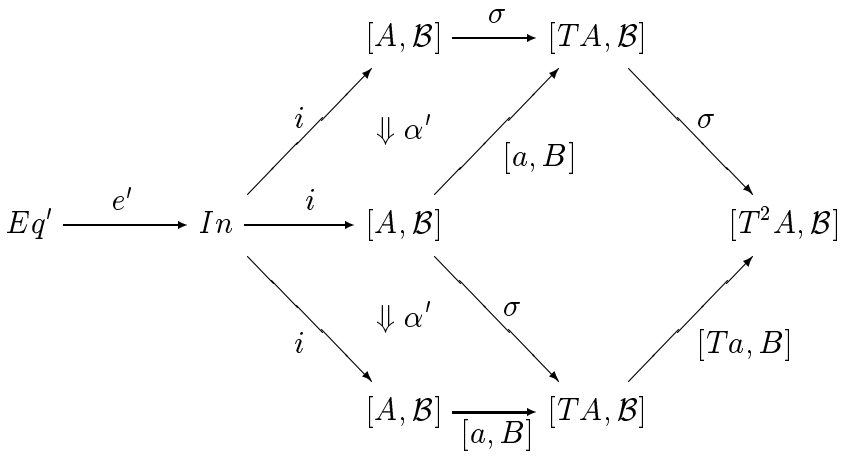
$$[A,B] \xrightarrow{T} [TA,TB] \xrightarrow{[TA,b]} [TA,B]$$

which canonically but not obviously lifts to a map in $T\text{-}Alg$, with 2-cell structure defined by use of γ, So we get a universal 2-cell $\alpha' : \sigma_{A,\mathcal{B}} \cdot i \longrightarrow [a,\mathcal{B}] \cdot i$.

(ii) Take the equifier $e' : Eq' \longrightarrow In$ of $[\eta_A,\mathcal{B}] \cdot \alpha'$ with the identity.

(iii) Take the equifier $e : Eq \longrightarrow Eq'$ of $[\mu_A,B] \cdot \alpha' \cdot e'$ with the following

pasting:



Here the final square commutes by naturality of σ, and the domains of the 2-cells match easily; for the codomains, one must work a little.

We write the resulting T-algebra $[\mathcal{A},\mathcal{B}]$ and call it, equipped with the composite

$$p = i \cdot e' \cdot e : [\mathcal{A},\mathcal{B}] \longrightarrow [A,\mathcal{B}]$$

and the isomorphic 2-cell

$$\alpha = \alpha' \cdot e' \cdot e : \sigma_{A,\mathcal{B}} \cdot p \longrightarrow [a,\mathcal{B}] \cdot p$$

the exponential $\mathcal{A}$ to $\mathcal{B}$.

Taking the canonical constructions of iso-inserters and equifiers in Cat, it transpires that our final Eq is exactly the category of pseudo-maps from $\mathcal{A}$ to $\mathcal{B}$. So the forgetful 2-functor takes $[\mathcal{A},\mathcal{B}]$ to $T\text{-}Alg(\mathcal{A},\mathcal{B})$. Moreover the following universal property follows directly from the construction.

Proposition 4.3 *Given T-algebras $\mathcal{A} = (A,a)$ and $\mathcal{B} = (B,b)$, the T-algebra $[\mathcal{A},\mathcal{B}]$ equipped with*

$$p : [\mathcal{A},\mathcal{B}] \longrightarrow [A,\mathcal{B}] \text{ and an isomorphic 2-cell } \alpha : \sigma_{A,\mathcal{B}} \cdot p \longrightarrow [a,\mathcal{B}] \cdot p$$

satisfies the universal property that for each $\mathcal{D}$, composition with p induces an isomorphism between $T\text{-}Alg(\mathcal{D},[\mathcal{A},\mathcal{B}])$ and the category of cones given by data

$$f : \mathcal{D} \longrightarrow [A,\mathcal{B}] \text{ and an isomorphic 2-cell } \beta : \sigma_{A,\mathcal{B}} \cdot f \longrightarrow [a,\mathcal{B}] \cdot f$$

satisfying two equification conditions: one for μ, the other for η.

To complete the proof of our main theorem, a delicate notion of multilinear map of T-algebras seems of fundamental importance [6]. But the above is the central point, and, taking the unit to be $T1$, the free T-algebra on 1, we have

Theorem 4.4 *If T is a pseudo-commutative 2-monad on Cat, then T-Alg is a pseudo-closed 2-category.*

References

[1] Abramsky, S., *Retracing some paths in process algebra*, "Proc. CONCUR **96**," Lect. Notes in Computer Science **1119** (1996) 1–17.

[2] Blackwell, R., G.M. Kelly, and A.J. Power, *Two-dimensional monad theory*, J. Pure Appl. Algebra **59** (1989) 1–41.

[3] Eilenberg, S., and G.M. Kelly, *Closed categories*, "Proc. Conference on Categorical Algebra (La Jolla 1965)," Springer-Verlag (1966).

[4] Fiore, M., and G.D. Plotkin, *An axiomatisation of computationally adequate domain-theoretic models of FPC*, Proc. LICS **94** (1994) 92–102.

[5] Fiore, M., G.D. Plotkin, and A.J. Power, *Cuboidal sets in axiomatic domain theory*, Proc. LICS **97** (1997) 268–279.

[6] Hyland, M., and A.J. Power, *Pseudo-commutative monads and pseudo-closed 2-categories*, J. Pure Appl. Algebra (to appear).

[7] Kelly, G.M., *Coherence theorems for lax algebras and for distributive laws*, Lecture Notes in Mathematics **420**, Springer-Verlag (1974) 281–375.

[8] Kock, A., *Closed categories generated by commutative monads*, J. Austral. Math Soc. **12** (1971) 405-424.

[9] Milner, R., *Calculi for interaction*, Acta Informatica **33** (1996) 707–737.

[10] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991) 55–92.

[11] O'Hearn, P.W., and D.J. Pym, *The logic of bunched implications*, Bull. Symbolic Logic (to appear)

[12] Power, A.J., and E. P. Robinson, *Premonoidal categories and notions of computation*, Math. Struct. in Comp. Science **7** (1997) 453–468.

MFPS 17 Preliminary Version

Stably Compact Spaces and Closed Relations

Achim Jung

School of Computer Science
The University of Birmingham
Birmingham, B15 2TT
England

Mathias Kegelmann

Fachbereich Mathematik
Technische Universität Darmstadt
Schloßgartenstraße 7
64289 Darmstadt
Germany

M. Andrew Moshier

Computer Science Department
Chapman University
333 N. Glassell Street
Orange, CA 92666
USA

Abstract

Stably compact spaces are a natural generalization of compact Hausdorff spaces in the T_0 setting. They have been studied intensively by a number of researchers and from a variety of standpoints.

In this paper we let the morphisms between stably compact spaces be certain "closed relations" and study the resulting categorical properties. Apart from extending ordinary continuous maps, these morphisms have a number of pleasing properties, the most prominent, perhaps, being that they correspond to preframe homomorphisms on the localic side. We exploit this Stone-type duality to establish that the category of stably compact spaces and closed relations has bilimits.

1 Introduction

The research reported in this paper derives its motivation from two sources. For some time, we have tried to extend Samson Abramsky's *Domain Theory*

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

in Logical Form to continuous domains, [1,15,14,17]. This has led to a number of insights, the most important perhaps being that in order to perform domain constructions *strictly logically*, one can invoke a version of Gentzen's cut elimination theorem. This, however, requires that we consider a purer logic than Abramsky did. Semantically, it then turns out that the notion of morphisms so captured consists of certain *relations*, rather than functions, [14, Proposition 6.5]. This is quite in line with developments in denotational semantics, where the need for (or the advantages of) relations has been noticed for some time, [5,3].

Our second motivation stems from the desire to circumvent some of the difficulties connected to classical domain theory. As is well known, in order to get a cartesian closed category of continuous domains, one has to restrict to a subcategory of FS-domains, [13,1]. Unlike general continuous domains, a straightforward characterisation of FS-domains via their Stone dual, for example, is not known. Perhaps as a result of the relative weakness of our tools for FS-domains, certain basic questions about them remain unresolved. We still do not know whether they coincide with retracts of bifinite domains or whether the probabilistic powerdomain can be restricted to this category, [16].

The semantic spaces which we put forward in this paper, in contrast to FS-domains, are very well behaved and understood. They are the so-called *stably-compact spaces*. Many equivalent characterisations are known and many properties have been discovered for them. Also, they do encompass most categories of continuous domains which have played a role in denotational semantics. As is clear from what we have said at the beginning, we are interested in the category SCS^* of stably compact spaces with closed *relations* as morphisms. Although a similar set-up has been considered some time ago, [26, Prop. 11.2.5], the explicit relational presentation appears to be new.

The purpose of this paper is to examine the suitability of SCS^* as a semantic universe. To this end we look at finitary closure properties and the bilimit construction. The latter, to our great satisfaction, behaves in a very natural and intuitive way. Specifically, we show that the bilimit coincides with a classical topological limit although it is constructed *order-theoretically*.

2 The category of of stably compact spaces and closed relations

2.1 The spaces

We assume standard domain theoretic notation as it is used in [8,1], for example. Slightly less well known, perhaps, are the following notions and results. If X is a topological space and A an arbitrary subset of X then the *saturation* of A is defined as the intersection of all neighborhoods of A. For any T_0-topological space X, the *specialization order* of X is the relation $\sqsubseteq_X$ given by

$x \sqsubseteq_X y$ if every neighborhood of x is also a neighborhood of y. The saturation of a subset A can then also be described as the upward closure with respect to $\sqsubseteq_X$. Open set are always upper, that is, saturated. An important fact is that the saturation of a compact set is again compact, for a set A has exactly the same open covers as its saturation.

For any topological space X the set of open subsets forms a complete lattice $\Omega(X)$ with respect to subset inclusion. Vice versa, for every complete lattice L the set of completely prime filters, denoted $\mathsf{pt}(L)$, carries the topology $\{O_a \mid a \in L\}$ where $F \in O_a$ if $a \in F$. A space is T_0 if the assignment, which associates with a point $x \in X$ the open neighborhood filter $\mathcal{N}(x)$, is injective. A space is called *sober* if the assignment is bijective. See [1, Section 7] for a detailed introduction to this topic. We are now ready to define the objects of interest in this paper:

Definition 2.1 A topological space is called *stably compact* if it is sober, compact, locally compact and finite intersections of compact saturated subsets are again compact.

Stably compact spaces have been studied intensively (and under many different names), [8,10,9,24,19,15] but, unfortunately, apart from [17] there is no single comprehensive reference for their many properties. We therefore state the main facts needed in the sequel. Our principal technical tool is the Hofmann-Mislove Theorem, [11,18]:

Theorem 2.2 *Let X be a sober space. There is an order-reversing bijection between the set $\mathcal{K}(X)$ of compact saturated subsets of X (ordered by reversed inclusion) and Scott-open filters in $\Omega(X)$ (ordered by inclusion). It assigns to a compact saturated set the filter of open neighborhoods and to a Scott-open filter of open sets their intersection.*

One consequence of this which we will need later is that every Scott-open filter in $\Omega(X)$ is equal to the intersection of all completely prime filters containing it. Another is the fact that the set $\mathcal{K}(X)$ is a dcpo when equipped with reversed inclusion. For stably compact spaces even more is true:

Proposition 2.3 *Let X be a stably compact space.*

(i) *$\mathcal{K}(X)$ is a complete lattice in which suprema are calculated as intersections and finite infima as unions.*

(ii) *$\Omega(X)$ and $\mathcal{K}(X)$ are stably continuous frames.*

(iii) *In $\Omega(X)$ we have $O \ll O'$ if and only if there is $K \in \mathcal{K}(X)$ with $O \subseteq K \subseteq O'$.*

(iv) *In $\mathcal{K}(X)$ we have $K \ll K'$ if and only if there is $O \in \Omega(X)$ with $K' \subseteq O \subseteq K$.*

As in [15] we use *stably continuous frame* to denote continuous distributive lattices in which the way-below relation is *multiplicative*, that is, in which

$x \ll y, z$ implies $x \ll y \wedge z$ and in which $1 \ll 1$. They are precisely the Stone duals of stably compact spaces, see [10, Theorem 1.5]. Note that the proposition tells us that the complements of compact saturated sets form another topology on X, called the *co-compact topology* for X and denoted by X_κ. Original and co-compact topology are closely related:

Proposition 2.4 *Let X be a stably compact space.*

(i) *The open sets of X_κ are the complements of compact saturated sets in X.*

(ii) *The open sets of X are the complements of compact saturated sets in X_κ.*

(iii) *X_κ is stably compact and $(X_\kappa)_\kappa$ is identical to X.*

(iv) *The specialization order of X is the inverse of the specialization order of X_κ.*

For a stably compact space X, the *patch topology* of X is the common refinement of the original topology and the co-compact topology. It is denoted by X_π. It is the key to making the connection to much earlier work by Leopoldo Nachbin, [21]: A *partially ordered space* or *pospace* is a topological space X with a partial order relation $\sqsubseteq_X$ such that the graph of $\sqsubseteq_X$ is a closed subset of $X \times X$. Such a space must be Hausdorff because the diagonal relation, i.e., the intersection of $\sqsubseteq_X$ and the opposite partial order $\sqsupseteq_X$, is closed.

Theorem 2.5 *For a stably compact space X the specialization order together with the patch topology makes X_π into a compact ordered space. Conversely, for a compact ordered space $(X, \sqsubseteq)$ the open upper sets $\uparrow U = U \in \Omega(X)$ form the topology for a stably compact space $X^\uparrow$, and the two operations are mutually inverse.*

Moreover, for a stably compact space X the upper closed sets of X_π are precisely the compact saturated sets of X.

Notice that for a compact Hausdorff space X, the diagonal relation Δ_X is a closed (trivial) partial order. By applying Theorem 2.5 to the pospace (X, Δ_X), we see that the upper opens and lower opens are just the opens of the original topology. So $X = X_\kappa = X_\pi$. The converse also holds.

Corollary 2.6 *A space X is compact Hausdorff if and only if it is a stably compact space for which $X = X_\kappa$.*

Proof. The patch topology for any stably compact space is Hausdorff. In the case of a stably compact space for which $X = X_\kappa$, the patch topology is simply the original. □

We can thus think of stably compact spaces as the T_0 generalization of compact Hausdorff spaces. The fact that $X \neq X_\kappa$ in general forces us to tread carefully in Section 2.2 as we generalize from closed relations between compact Hausdorff spaces to closed relations between stably compact spaces.

The importance of stably compact spaces for domain theory is that almost all categories used in semantics are particular categories of stably compact

spaces.

Proposition 2.7 *FS domains, and hence in particular Scott domains and continuous lattices, equipped with their Scott topologies, are stably compact spaces.*

2.2 The morphisms

The obvious category of stably compact spaces is that of continuous functions, i.e. the full subcategory SCS of the category of topological spaces Top. The category that we are really interested in, however, is one that generalizes KHaus^*, the category of compact Hausdorff spaces and closed relations. We quote the basic definitions and results from [14].

The specialization order of a stably compact space X is generally not closed in $X \times X$. Indeed, were it closed, X would be a pospace, hence would be Hausdorff. Thus, specialization would be trivial. Specialization, on the other hand, is reversed by taking the co-compact topology (again, in the Hausdorff case $X = X_\kappa$ so the "reversal" is trivial). Thus:

Proposition 2.8 *The specialization order of a stably compact space X is closed in $X \times X_\kappa$.*

Proof. Suppose that $x \not\sqsubseteq_X y$. Then there is an open set U containing x and not y. By local compactness, we can assume that U is contained in a compact saturated neighborhood K of x that also does not contain y. U is an upper set containing x. The complement of K is a lower set containing y. Thus $U \times (X \setminus K)$ is a neighborhood of $\langle x, y\rangle$ in $X \times X_\kappa$ that does not meet $\sqsubseteq_X$.□

For stably compact spaces X and Y, we call a closed subset $R \subseteq X \times Y_\kappa$ a *closed relation from X to Y* and we write it as $R\colon X \multimap Y$. If we spell out this condition then it means that for $x \in X$ and $y \in Y$ such that $x \not R\, y$ we find an open neighborhood U of x and a compact saturated set $K \subseteq Y$ that doesn't contain y such that $U \times (Y \setminus K) \cap R = \emptyset$. [cf. the proof Proposition 2.8.] Note that every closed relation R satisfies the rule $x' \sqsubseteq_X x\ R\ y \sqsubseteq_Y y' \implies x'\ R\ y'$.

The composition of closed relations is the usual relation product, $R\,;S = \{\langle x, z\rangle \mid (\exists y)\ x\ R\ y \text{ and } y\ S\ z\}$. Note that, following usual practice, we write the composition of relations from left to right, whereas for functions it is from right to left. To avoid ambiguity we use ";" to indicate left-to-right composition. Notice that the specialization order of any stably compact space X acts as identity under taking the relation product with closed relations from or to X and also that the composition of two closed relations is again closed. We call the category of stably compact spaces with closed relations SCS^*.

The Hausdorff case is worth considering separately as it helps to illuminate the definition of closed relations. As we have noted, a stably compact space is Hausdorff if and only if its topology agrees with its co-compact topology. Thus

our closed relations from X to Y are simply closed subsets of $X \times Y = X \times Y_\kappa$ whenever Y is Hausdorff. Thus SCS^* correctly generalizes KHaus^*, in which we could take the morphisms simply as closed subsets of $X \times Y$. The fact that we could get away with this apparently simpler notion of morphism in the Hausdorff setting is due essentially to the fact that in compact Hausdorff spaces the co-compact topology is "hidden from view." In particular, KHaus^* is a full subcategory of SCS^* (as well as being a subcategory of Rel).

Note that the obvious forgetful "functor" from SCS^* to Rel, the category of sets with relations, preserves composition but *not* identities. The only stably compact spaces for which identity is preserved are those with trivial specialization orders, i.e., the compact Hausdorff spaces.

Relations between sets can be understood as multi-functions. As the following proposition shows this carries over to our topological setting in an interesting way.

Proposition 2.9 *Let X and Y be stably compact spaces and $R\colon X \multimap Y$ a closed relation then*

$$f_R(x) := \{y \in Y \mid x \mathrel{R} y\}$$

defines a continuous function from X to $\mathcal{K}(Y)$, where the latter is equipped with the Scott topology. Conversely, if $f\colon X \to \mathcal{K}(Y)$ is continuous then

$$\big\{\langle x, y\rangle \in X \times Y \;\big|\; y \in f(x)\big\}$$

is a closed relation from X to Y. Moreover, these two translations are mutually inverse.

To extend this correspondence to the composition of relations and multi-functions, respectively, we first have to define a law of composition on the latter. To this end recall that $\mathcal{K}(X)$ with its Scott topology is again a stably compact space by Propositions 2.3 and 2.7. Hence we can make $\mathcal{K}$ into an endofunctor on SCS by mapping a continuous function $f\colon X \to Y$ to the function $\mathcal{K}(f)\colon \mathcal{K}(X) \to \mathcal{K}(Y)$ that takes a compact saturated subset $K \subseteq X$ to $\uparrow f[K]$. This endofunctor is part of a monad whose unit takes the saturation of points and whose multiplication is simply union [22]. Consequently, the canonical composition of multi-functions is Kleisli composition which turns out to be the analogue of ordinary relation product.

Proposition 2.10 *The category of closed relations SCS^* is isomorphic to the Kleisli category $\mathsf{SCS}_\mathcal{K}$.*

It is generally the case that a category C with a monad T is embedded in the Kleisli category C_T simply by post-composing with the unit of the monad. Moreover, if the units of the monad are monic, then the embedding is faithful. Hence, SCS is a subcategory of $\mathsf{SCS}_\mathcal{K}$ and thus also of SCS^*. Concretely,

this embedding works by taking the *hypergraph* of a function. The following proposition characterizes those relations that are really embedded functions:

Proposition 2.11 *If $f\colon X \to Y$ is a continuous function then the hypergraph*

$$\big\{\langle x, y\rangle \in X \times Y \;\big|\; f(x) \sqsubseteq y\big\}$$

is a closed relation from X to Y. Conversely, if $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ is a closed relation such that for all $x \in X$ the set $f_R(x)$ has a least element $r(x)$ then $r\colon X \to Y$ is a continuous function, and this operation is the inverse of the previous.

Again, the Hausdorff case may help to illuminate this. If $f\colon X \to Y$ is a continuous function with Y a compact Hausdorff space, then the hypergraph is simply the graph of f. This is a closed relation just as classical topology tells us it should be. Conversely, suppose that a closed relation from X to Y is the graph of a function g. Then clearly $f_R(x)$ has a least element $g(x)$ for each x. Thus g is a continuous function.

2.3 The category

The left adjoint from SCS to the Kleisli category $\mathsf{SCS}_{\mathcal{K}} \cong \mathsf{SCS}^*$ preserves coproducts. Hence, they are given in SCS^* simply as topological coproducts, i.e., as disjoint unions.

In the category Rel of sets and relations for every relation $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ there is the reciprocal relation R_κ that is given by $y\ R_\kappa\ x \iff x\ R\ y$. This is the main ingredient that makes Rel into an allegory [7]. Our category SCS^* fails to be an allegory exactly because, as we shall see, it lacks a true reciprocation operation. On the other hand, if $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ is a closed relation between stably compact spaces then $R_\kappa\colon Y_\kappa \mathrel{-\!\!\!+\!\!\!\rightarrow} X_\kappa$ is a closed relation between the co-compact topologies, and $(\cdot)_\kappa$ is an involution on SCS^*. The problem is that it doesn't fix objects. We can think of X_κ as an upside-down version of X since the specialization order $\sqsubseteq_{X_\kappa}$ for the co-compact topology is simply $\sqsupseteq_X$, i.e. the dual of the one for the original space.

Nonetheless, the maps $X \mapsto X_\kappa$ and $R \mapsto R_\kappa$ comprise a contravariant functor, showing that SCS^* is a self-dual category. Consequently, categorical products (denoted here by $X \times^* Y$ to avoid conflict with topological products $X \times Y$) are also given by disjoint union:

$$X \times^* Y \cong (X_\kappa + Y_\kappa)_\kappa = (X_\kappa \mathbin{\dot\cup} Y_\kappa)_\kappa = (X_\kappa)_\kappa \mathbin{\dot\cup} (Y_\kappa)_\kappa = X \mathbin{\dot\cup} Y = X + Y.$$

If a self-dual category is cartesian closed then all objects are isomorphic and hence the category is equivalent to the category with only one (identity) morphism. This shows that SCS^* cannot be cartesian closed.

Since categorical products in SCS^* are the same as co-products, let us look at cartesian products. In SCS they are the categorical product and we

can lift them to SCS^* to make SCS^* into a symmetric monoidal category. The tensor product takes the cartesian product of the spaces with the product topology and we also embed the morphisms needed for the symmetric monoidal structure from SCS as described in Proposition 2.11. The definition of the tensor product of two closed relations R and S is pointwise, i,e, $\langle x, y\rangle\ R \otimes S\ \langle x', y'\rangle : \iff\ x\ R\ y$ and $x'\ R\ y'$. This defines a closed relation and extends to products of continuous functions; for the details see [17, Section 3.2.4].

With respect to $\otimes$, the category SCS^* is closed: Because of $(X \times Y)_\kappa = X_\kappa \times Y_\kappa$ we see that closed subsets of $(X \times Y) \times Z_\kappa$ are the same thing as closed subsets of $X \times (Y_\kappa \times Z)_\kappa$ which proves $\mathsf{SCS}^*(X \otimes Y, Z) \cong \mathsf{SCS}^*(X, Y_\kappa \otimes Z)$. This internal homset $Y_\kappa \otimes Z$, however, does not correspond to the "real" homset $\mathsf{SCS}^*(Y, Z)$.

The homset $\mathsf{SCS}^*(Y, Z)$ consists of the closed subsets of $Y \times Z_\kappa$ which by Theorem 2.5 are precisely the compact saturated subsets of the dual $(Y \times Z_\kappa)_\kappa$. Hence, we can write the *relation space* as $[Y \Rightarrow Z] := \mathcal{K}(Y_\kappa \times Z)$. With this definition and Proposition 2.10 we get

$$\mathsf{SCS}^*(X \otimes Y, Z) \cong \mathsf{SCS}^*(X, Y_\kappa \otimes Z) \cong \mathsf{SCS}\big(X, \mathcal{K}(Y_\kappa \otimes Z)\big) = \mathsf{SCS}\big(X, [Y \Rightarrow Z]\big).$$

So, we see that $(- \otimes Y)$ and $[Y \Rightarrow -]$ are almost adjoint. The problem is that the induced morphism $X \mathrel{-\!\!\!\vdash\!\!\!\rightarrow} [Y \Rightarrow Z]$ is not uniquely determined.

The canonical evaluation morphism is a *functional* closed relation and for the induced morphism we can always choose a functional one, and as such it is unique, i.e. these morphisms come from SCS rather than SCS^*. In [23] such a situation is called a *Kleisli exponential*. There is an alternative description of the relation space by observing $\mathsf{SCS}^*(Y, Z) \cong \mathsf{SCS}\big(Y, \mathcal{K}(Z)\big)$. Thus the normal function space $[Y \to \mathcal{K}(Z)]$ with the compact-open topology, which is simply the Scott topology, yields a space that is homeomorphic to $[Y \Rightarrow Z]$. This construction was first studied in [25], although it seems that some of subtleties concerning the fact that this is only a Kleisli exponential were overlooked.

3 Stone Duality

Next we develop the Stone duality of closed relations. The morphisms between open set lattices corresponding to closed relations turn out to be preframe homomorphisms, [2], preserving finite meets and directed suprema. They have been studied in a similar framework before, see [26, Prop. 11.2.5], but the duality with relations seems to be new.

3.1 Relational preimage

If $R\colon X \mathrel{-\!\!\!\vdash\!\!\!\rightarrow} Y$ is a relation and $A \subseteq X$ a subset, then we write

$$[A]R := \big\{ y \in Y \;\big|\; (\exists x \in A)\ x\ R\ y \big\}$$

for the usual forward image. The definition of the preimage of a subset $B \subseteq Y$ under the relation R is a bit more tricky as there are several candidates. Here, we are only interested in the *universal preimage* given by

$$(\forall R)[B] := \big\{x \in X \;\big|\; (\forall y \in Y)\; x\; R\; y \implies y \in B\big\}.$$

This definition is useful because $\forall R$ turns out to be the right adjoint to $[\cdot]R$:

Lemma 3.1 *If $R \subseteq X \times Y$ is a relation and A and B are subsets of X and Y, respectively, then we have*

$$[A]R \subseteq B \iff A \subseteq (\forall R)[B].$$

In the usual functional setting the situation is analogous; preimage is right adjoint to direct image. The connection between relational and functional preimage is the following.

Lemma 3.2 *If $f\colon X \to Y$ is a continuous function between stably compact spaces and $F\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ the corresponding closed relation given by the hypergraph, then for all upper sets $A = {\uparrow}A \subseteq Y$ we have*

$$f^{-1}[A] = (\forall F)[A].$$

We now describe the translation from topological spaces to frames in the relational setting.

Proposition 3.3 *If $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ is a closed relation then $\forall R$ is a continuous semilattice homomorphism from $\Omega(Y)$ to $\Omega(X)$, i.e. it preserves finite infima and directed suprema.*

Proof. First, we have to check that for any open $V \subseteq Y$ the preimage $(\forall R)[V]$ is open. So let $x \in (\forall R)[V]$, or equivalently $f_R(x) = [x]R \subseteq V$. We know from Proposition 2.9 that f_R is continuous and thus Proposition 2.3 gives us an open neighborhood U of x such that $f_R(x') \subseteq V$ for all $x' \in U$. We conclude $x \in U \subseteq (\forall R)[V]$, thus showing that for a closed relation the universal preimage of an open set is open.

As we have seen in Lemma 3.1, $\forall R$ as a function between the full powersets is a right adjoint. As such it preserves all intersections and thus the finite meets in $\Omega(Y)$.

Thus, it is a monotone map and, consequently, to show that it also preserves directed suprema we only have to verify $(\forall R)\big[\bigcup^{\uparrow} V_i\big] \subseteq \bigcup^{\uparrow}(\forall R)[V_i]$. So, we consider an $x \in (\forall R)\big[\bigcup^{\uparrow} V_i\big]$ which means $f_R(x) \subseteq \bigcup^{\uparrow} V_i$. But as $f_R(x)$ is compact we can find an index i such that $f_R(x) \subseteq V_i$ and, equivalently, such that $x \in (\forall R)[V_i]$. □

We call $\Omega^* R$ the restriction and co-restriction of $\forall R$ to the open subsets of X and Y to simplify notation. Going from a relation to the forward image function is well-known to be functorial, and so is taking adjoints. By

Lemma 3.1 this implies that universal preimage is also functorial. Clearly, $\Omega^* \sqsubseteq_X$ is the identity on $\Omega^*(X) = \Omega(X)$ as all open sets are upper sets. Thus Ω^* is a contravariant functor from SCS^* to the category of stably continuous frames and Scott continuous semilattice homomorphisms which we denote by SCF^*.

Just like Ω we also have to adjust the functor pt to the relational setting. Consider a homomorphism $\phi\colon L \to M$. We define the relation $\mathsf{pt}^*(\phi)\colon \mathsf{pt}^*(M) \mathrel{-\!\!+\!\!\to} \mathsf{pt}^*(L)$ by

$$Q\ \mathsf{pt}^*(\phi)\ P :\iff \phi^{-1}[Q] \subseteq P$$

where pt^* on objects behaves just like the usual pt, i.e., P and Q are completely prime filters in L and M, respectively. Alternatively, we can identify completely prime filters with their characteristic functions which are frame morphisms to $\mathbf{2}$, the two-element lattice. For two such *points* $p\colon L \to \mathbf{2}$ and $q\colon M \to \mathbf{2}$ the above definition becomes

$$q\ \mathsf{pt}^*(\phi)\ p :\iff q \circ \phi \sqsubseteq p.$$

Proposition 3.4 *If $\phi\colon L \to M$ is a continuous semilattice homomorphism, then $\mathsf{pt}^*(\phi)\colon \mathsf{pt}^*(M) \mathrel{-\!\!+\!\!\to} \mathsf{pt}^*(L)$ is a closed relation.*

Proof. Suppose $Q \subseteq M$ and $P \subseteq L$ are completely prime filters such that $\phi^{-1}[Q] \nsubseteq P$. As ϕ is Scott continuous and Q completely prime and thus, in particular, Scott open, the set $\phi^{-1}[Q]$ is also Scott open. Because it is also not contained in P and L is a continuous lattice we can find an $x \in \phi^{-1}[Q] \setminus P$ such that $\Uparrow x \nsubseteq P$. On the other hand Q, as an upper set, is the union of principal filters $\uparrow y$ for $y \in Q$ and hence we get $\phi^{-1}[Q] = \phi^{-1}\big[\bigcup\{\uparrow y \mid y \in Q\}\big] = \bigcup\big\{\phi^{-1}[\uparrow y] \;\big|\; y \in Q\big\} \ni x$. This means that we can find a $y \in Q$ such that $x \in \phi^{-1}[\uparrow y]$.

As L is stably continuous, the set $\Uparrow x$ is a Scott open filter which corresponds to the compact saturated subset $\big\{P \in \mathsf{pt}^*(L) \;\big|\; \Uparrow x \subseteq P\big\}$ of $\mathsf{pt}^*(L)$ by the Hofmann-Mislove theorem. Now, we consider the open subset of $\mathsf{pt}^*(M) \times \mathsf{pt}^*(L)_\kappa$ which is given as the product of the open set corresponding to y and to the complement of the compact saturated set corresponding to $\Uparrow x$, and we claim that this is a neighborhood of $\langle Q, P\rangle$ that doesn't meet R_ϕ. Clearly, $\langle Q, P\rangle$ is in this set, and if $Q' \in \mathsf{pt}^*(M)$ and $P' \in \mathsf{pt}^*(L)$ are such that $y \in Q'$ and $\Uparrow x \nsubseteq P'$ we get $\phi^{-1}[Q'] \supseteq \phi^{-1}\big[\uparrow y\big] \ni x$ and thus $\phi^{-1}[Q'] \supseteq \Uparrow x$ which implies $\phi^{-1}[Q'] \nsubseteq P'$. □

Now we have all the ingredients for a duality between SCS^* and SCF^*. It remains to check that the categorical conditions are indeed met.

Theorem 3.5 *The contravariant functors Ω^* and pt^* are part of a dual equivalence between the categories SCF^* and SCS^*.*

Proof. We begin by showing that pt^* is indeed a functor. Clearly, $\mathsf{pt}^*(\mathsf{id}_L) = \sqsubseteq_{\mathsf{pt}^*(L)}$, the identity closed relation on $\mathsf{pt}^*(L)$. The interesting direction for

functoriality is to show that $\mathsf{pt}^*(\psi \circ \phi) \subseteq \mathsf{pt}^*(\psi); \mathsf{pt}^*(\phi)$, where $\phi\colon L \to M$ and $\psi\colon M \to N$ are continuous semilattice morphisms. Let $P \in \mathsf{pt}^*(N)$ and $P' \in \mathsf{pt}^*(L)$ be such that $P\ (\mathsf{pt}^*(\psi \circ \phi))\ P'$, or equivalently that $\phi^{-1}\big[\psi^{-1}[P]\big] \subseteq P'$. We need to find a completely prime filter $Q \subseteq M$ that satisfies $\psi^{-1}[P] \subseteq Q$ and $\phi^{-1}[Q] \subseteq P'$. Unfortunately, $\psi^{-1}[P]$ in general is only a Scott open filter, not a point in M.

However, by the Hofmann-Mislove Theorem, 2.2, we have $\psi^{-1}[P] = \bigcap\{Q \in \mathsf{pt}^*(M) \mid \psi^{-1}[P] \subseteq Q\}$. So for the sake of contradiction, assume there exists $x_Q \in \phi^{-1}[Q] \setminus P'$ for all $Q \supseteq \psi^{-1}[P]$. Then the supremum $\bigvee x_Q$ of all these elements does not belong to P' because P' is completely prime; on the other hand, $\phi(\bigvee x_Q)$ belongs to all $Q \supseteq \psi^{-1}[P]$ by monotonicity of ϕ, hence to $\psi^{-1}[P]$. This contradicts the assumption $\phi^{-1}[\psi^{-1}[P]] \subseteq P'$.

To show that Ω^* and pt^* give rise to a duality between SCF^* and SCS^* we have to check that their actions on morphisms are mutually inverse. So, suppose $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ is a closed relation and $\mathcal{N}(x)$ and $\mathcal{N}(y)$ are the open neighborhood filters of two points $x \in X$ and $y \in Y$. We get

$$\begin{aligned}
\mathcal{N}(x)\ (\mathsf{pt}^*(\forall R))\ \mathcal{N}(y) &\iff (\forall R)^{-1}\big[\mathcal{N}(x)\big] \subseteq \mathcal{N}(y) \\
&\iff \big(\forall V \in \Omega^*(Y)\big)\ V \in (\forall R)^{-1}\big[\mathcal{N}(x)\big] \Longrightarrow V \in \mathcal{N}(y) \\
&\iff \big(\forall V \in \Omega^*(Y)\big)\ x \in (\forall R)[V] \Longrightarrow y \in V \\
&\iff \big(\forall V \in \Omega^*(Y)\big)\ [x]R \subseteq V \Longrightarrow y \in V
\end{aligned}$$

Clearly, $x\ R\ y$ implies this last condition and the converse follows from the fact that $[x]R$ is saturated.

Finally, we take a continuous semilattice morphism $\phi\colon L \to M$ and show that $\big(\Omega^*(\mathsf{pt}^*(\phi))\big)\big(\{P \in \mathsf{pt}^*(L) \mid x \in P\}\big) = \big\{Q \in \mathsf{pt}^*(M) \;\big|\; \phi(x) \in Q\big\}$ for any $x \in L$:

$$\begin{aligned}
&\big(\forall\, \mathsf{pt}^*(\phi)\big)\Big(\big\{P \in \mathsf{pt}^*(L) \mid x \in P\big\}\Big) \\
&= \Big\{Q \in \mathsf{pt}^*(M) \;\Big|\; \big(\forall P \in \mathsf{pt}^*(L)\big)\ Q\ (\mathsf{pt}^*(\phi))\ P \Longrightarrow x \in P\Big\} \\
&= \Big\{Q \in \mathsf{pt}^*(M) \;\Big|\; \big(\forall P \in \mathsf{pt}^*(L)\big)\ \phi^{-1}[Q] \subseteq P \Longrightarrow x \in P\Big\}
\end{aligned}$$

As before we use the fact that $\phi^{-1}[Q]$ is a Scott-open filter and hence by the Hofmann-Mislove Theorem equal to the intersection of all completely prime filters containing it. The expression then re-writes to $\{Q \in \mathsf{pt}^*(M) \mid x \in \phi^{-1}[Q]\}$ which is equal to $\{Q \in \mathsf{pt}^*(M) \mid \phi(x) \in Q\}$ as desired. □

It is interesting to consider the Stone dual of the involution on SCS^* that we discussed in Section 2.3. The co-compact topology on a stably compact space has precisely the compact saturated subsets of the original space as closed sets which implies $\Omega^*(X_\kappa) = \Omega(X_\kappa) \cong \mathcal{K}(X)$. From the Hofmann-Mislove Theorem we know that $\mathcal{K}(X)$ is in one-to-one correspondence to the Scott open filters in $\Omega(X)$. The latter can also be understood via their characteristic functions which are precisely the continuous semilattice homomorphisms to $\mathbf{2}$, the two-element lattice. Putting it all together we get $\Omega(X_\kappa) \cong \mathcal{K}(X) \cong$

$\mathsf{SCF}^*\big(\Omega(X), \mathbf{2}\big)$ and we see that this self-duality in localic terms is exactly the Lawson duality of stably continuous semilattices [20].

3.2 Functions revisited

We know from Proposition 2.11 that SCS embeds faithfully in SCS^* and also how to recognize the morphisms that arise from this embedding as hypergraphs of functions. We refer to a closed relation as *functional* if it is the hypergraph of a continuous function. Similarly the category SCF^* contains a subcategory of functional arrows.

Proposition 3.6 *If $R\colon X \mathrel{-\!\!\!+\!\!\!\rightarrow} Y$ is a functional closed relation then $\Omega^*(R)$ preserves finite (and consequently all) suprema. Conversely, if $\phi\colon L \to M$ is a frame homomorphism then $\mathsf{pt}^*(L)$ is functional.*

Proof. If ϕ is a frame homomorphism then for any completely prime filter $Q \subseteq M$ the preimage $\phi^{-1}[Q]$ is completely prime. Hence, this is the least completely prime filter $P \subseteq L$ such that $\phi^{-1}[Q] \subseteq P$.

For the converse observe that the forward image $[x]R$ of any point x has a least element and hence will be contained in either U or V iff it is contained in $U \cup V$. This shows that $\forall R$ preserves finite suprema. □

This result, of course, is very similar to the classical Stone duality between SCS, the category of stably compact spaces with continuous functions, and $\mathsf{SCF}^*{}_\vee$, stably continuous lattices with frame homomorphisms. There the functors Ω and pt act on morphisms as follows: $\Omega(f)$ is simply the preimage function $f^{-1}[\cdot]$ and similarly $\mathsf{pt}(\phi)$ takes a completely prime filter P to the completely prime filter $\phi^{-1}[P]$. As a corollary of the previous proposition we get that pt^* and Ω^* commute with the embeddings of the functional subcategories.

Corollary 3.7 *The diagram of functors*

$$\begin{array}{ccc} \mathsf{SCS} & \overset{\Omega}{\underset{\mathsf{pt}}{\rightleftarrows}} & \mathsf{Frm} \\ {\scriptstyle i}\downarrow & & \downarrow{\scriptstyle j} \\ \mathsf{SCS}^* & \overset{\Omega^*}{\underset{\mathsf{pt}^*}{\rightleftarrows}} & \mathsf{SCF}^* \end{array}$$

commutes in the sense that $j \circ \Omega = \Omega^* \circ i$ and $i \circ \mathsf{pt}^* = \mathsf{pt} \circ j$.

Proof. The first equality was proved in Lemma 3.2. For the second, take a frame morphism $\phi\colon L \to M$. It is mapped by $i \circ \mathsf{pt}$ to the hypergraph of the preimage function, i.e. the closed relation that relates $Q \in \mathsf{pt}(M) = \mathsf{pt}^*(M)$ to $P \in \mathsf{pt}(L) = \mathsf{pt}^*(L)$ if and only if $\phi^{-1}[Q] \subseteq P$ which is precisely $\mathsf{pt}^*(j(\phi))$.□

As a consequence of this corollary the operation which extracts from a functional relation the underlying continuous function (which exists by Proposition 2.11) is just the composition $\mathsf{pt} \circ \Omega^*$. It follows that this is functorial. We denote it by U.

There is a more categorical way to identify the functional morphisms in the two dual categories. As we have seen in Section 2.3, the products on the functional subcategory give rise to a symmetric monoidal structure on the larger relational category. In addition, the diagonals $\Delta_A \colon A \to A \times A$ and morphisms $!_A$ to the terminal object induce a *diagonal structure*. The functional morphisms are then characterized as the *total* and *deterministic* morphisms, i.e. the ones for which ! and Δ, respectively, are natural transformations. For more details see [17, Section 3.3].

4 Subspaces

There are a number of different concepts of "good subspace" in Topology as often simply carrying the induced topology is too weak. One very useful one that is well-known in domain theory is that of an *embedding-projection pair*. It combines the categorical notion of section retraction pair with the order theoretic notion of adjunction. It is then an immediate corollary that the space that is the codomain of the section carries the subspace topology. In the following we will generalize this to the relational setting.

4.1 Perfect relations

We start by defining a special class of relations that will be important when we characterize relations that have adjoints.

Definition 4.1 We say that a closed relation $R \colon X \mathrel{-\!\!\!\vdash\!\!\!\rightarrow} Y$ is *perfect* if for all compact saturated sets $K \subseteq Y$ the preimage $(\forall R)[K]$ is compact.

Perfect relations can alternatively be characterized in terms of their Stone duals.

Proposition 4.2 *A closed relation $R \colon X \mathrel{-\!\!\!\vdash\!\!\!\rightarrow} Y$ is perfect if and only if $\Omega^*(R)$ preserves the way-below relation.*

Proof. Let us assume that R is perfect and $U \ll V$ are open subsets of Y. Then there is a compact saturated set $K \subseteq Y$ such that $U \subseteq K \subseteq V$ and we get $\Omega^*(R)(U) = (\forall R)[U] \subseteq (\forall R)[K] \subseteq (\forall R)[V] = \Omega^*(R)(V)$. By assumption $(\forall R)[K]$ is compact and hence we conclude $\Omega^*(R)(U) \ll \Omega^*(R)(V)$.

Conversely, suppose $\Omega^*(R)$ preserves way-below and $K \subseteq Y$ is compact saturated. As a saturated set, K it is the intersection of all the open sets that contain it and we compute

$$(\forall R)[K] = (\forall R)\big[\bigcap\nolimits_{\downarrow} \{U \in \Omega^*(Y) \mid K \subseteq U\}\big] = \bigcap\nolimits_{\downarrow} \big\{(\forall R)[U] \;\big|\; K \subseteq U\big\}$$

where the last equality follows because, by Lemma 3.1, $\forall R$ is a right adjoint and hence preserves arbitrary intersections in $\mathfrak{P}(Y)$. Now we claim that this last intersection is taken over a filterbase for a Scott open filter in $\Omega^*(X) = \Omega(X)$. The set $\big\{(\forall R)[U] \;\big|\; K \subseteq U\big\}$ is clearly filtered. To see that it is generates a Scott open filter take $U \in \Omega(Y)$ that contains K. Since Y is locally compact, the neighborhood filter of the compact set K has a basis of compact saturated sets. This means that there is an open set V and a compact set K' such that $K \subseteq V \subseteq K' \subseteq U$. This implies $V \ll U$ and hence by assumption $(\forall R)[V] \ll (\forall R)[U]$.

By the Hofmann-Mislove Theorem the intersection over a Scott open filter of open sets, and hence also of a filterbase for such a filter, is compact saturated. This shows that $(\forall R)[K]$ is compact and finishes the proof. □

This extends the classical situation of functions between stably compact spaces (or, more generally, locally compact sober spaces), [10, Remark 1.3]. Since the Stone dual of a function has an upper adjoint, perfectness in that situation can be further characterized by the adjoint being Scott-continuous (loc. cit.). Because of Corollary 3.7 we have that a continuous function between stably compact spaces is perfect in the classical sense if and only if the corresponding relation given by the hypergraph is perfect in our sense.

It may be worthwhile to add a few words about terminology here. As we quoted, perfect maps have (at least) three different characterizations and furthermore many useful properties. Depending on what is considered essential in a given situation, additional assumptions are made in order to preserve certain key properties in the absence of local compactness, sobriety or both. This has led to an abundance of different concepts for which it now appears impossible to establish a coherent terminology. Either of "proper" [4,10] or "perfect" [12,9,6] is usually used but it is not clear where the boundary between the two ought to be drawn. Our choice of "perfect" follows the more recent custom of reserving "proper" for slightly stronger requirements even in the case of locally compact sober spaces.

We also note that perfect *functions* between stably compact spaces are exactly those which are continuous with respect to both original and co-compact topology. This implies that they are exactly those maps which are monotone and patch continuous. To summarize:

Proposition 4.3 *Let $f\colon X \to Y$ be a function between stably compact spaces and $R\colon X \multimap Y$ the corresponding hypergraph. Then the following are equivalent:*

(i) *R is perfect;*

(ii) *f is perfect with respect to the original topologies;*

(iii) *f is perfect with respect to the co-compact topologies;*

(iv) *f is monotone and patch continuous.*

There is yet another approach to perfectness via uniform continuity: For

every stably compact space there is a unique quasi-uniformity $\mathcal{U}$ such that $\mathcal{U}$ induces the topology and $\mathcal{U}^{-1}$ induces the co-compact topology. A continuous function $f\colon X \to Y$ between stably compact spaces is perfect if and only if it is uniformly continuous with respect to these unique quasi-uniformities on X and Y. For details see [25, Theorem 3].

In a way, perfect continuous functions seem to be a better notion of morphisms for the category SCS than just continuous ones, as open and compact saturated sets play similarly important roles. Moreover, with these morphisms we can explain in which way the patch topology is a "natural" construction: Every continuous function between compact Hausdorff spaces is perfect, and hence this category embeds fully and faithfully into SCS with perfect maps. Now, taking the patch topology is simply the right adjoint, i.e. the co-reflector, for this inclusion functor, [6].

Returning to closed relations again, perfectness is linked to openness. We say that a closed relation $R\colon X \mathrel{-\!\!\!+\!\!\!\to} Y$ is *open* if for all open sets $U \subseteq X$ the forward image $[U]R$ is open.

For the next proposition we need the following observation which relates forward image, universal preimage, complementation and reciprocation:

Lemma 4.4 *If $R\colon X \mathrel{-\!\!\!+\!\!\!\to} Y$ is a relation in Rel and $M \subseteq X$ is an arbitrary subset then $[X \setminus M]R = Y \setminus (\forall R_\kappa)[M]$.*

Proof. For $y \in Y$ we have

$$\begin{aligned} y \in [X \setminus M]R &\iff (\exists x \in X \setminus M)\, x\ R\ y \\ &\iff y \notin (\forall R_\kappa)[M] \\ &\iff y \in Y \setminus (\forall R_\kappa)[M]. \end{aligned}$$

□

Proposition 4.5 *A closed relation $R\colon X \mathrel{-\!\!\!+\!\!\!\to} Y$ is open if and only if the reciprocal relation $R_\kappa\colon Y_\kappa \mathrel{-\!\!\!+\!\!\!\to} X_\kappa$ is perfect.*

Proof. Let us assume that R is open. We take a compact saturated set $K \in \mathcal{K}(X_\kappa)$ and have to show that $(\forall R_\kappa)[K]$ is compact in Y_κ. By Theorem 2.5 the condition $K \in \mathcal{K}(X_\kappa)$ is equivalent to $X \setminus K \in \Omega(X)$ and the openness of R means that $[X \setminus K]R$ is open. By the previous lemma we have $[X \setminus K]R = Y \setminus (\forall R_\kappa)[K] \in \Omega(Y_\kappa)$ which, again by Theorem 2.5, implies that $(\forall R_\kappa)[K]$ is a compact saturated subset of Y_κ.

Conversely, if R_κ is perfect and $U \in \Omega(X)$ then $X \setminus U$ is compact saturated in X_κ. From the previous lemma we get $(\forall R_\kappa)[X \setminus U] = Y \setminus Y \setminus (\forall R_\kappa)[X \setminus U] = Y \setminus [X \setminus (X \setminus U)]R = Y \setminus [U]R$ which is a compact saturated subset of Y_κ because of the perfectness of R_κ. Consequently, its complement $[U]R$ is an open subset of Y. □

4.2 Adjunctions

As usual in an order-enriched category, we say that for two closed relations $R\colon X \multimap Y$ is the *left* or *lower adjoint* of $S\colon Y \multimap X$ if $S\,;R\colon X \multimap X$ is below the identity and if $R\,;S\colon Y \multimap Y$ is above the identity on Y. Likewise, S is called the *right* or *upper adjoint* of R. The question is what is the right order on the homsets $\mathsf{SCS}^*(X,Y)$. One choice is subset inclusion but it turns out to be better to use the one induced from the corresponding homsets $\mathsf{SCS}\big(X,\mathcal{K}(Y)\big)$, in keeping with Proposition 2.10. Since $\mathcal{K}(Y)$ is ordered by reverse inclusion this means that the relations in the homsets for SCS^* are also ordered by reverse inclusion of their graphs. Note that adjoints determine each other uniquely as is the case in any order-enriched category.

Lemma 4.6 *The functors Ω^* and pt^* preserve the order on the homsets, thus making SCS^* and SCF^* dually equivalent as order-enriched categories. Consequently, we have $R \dashv S$ for closed relations if and only if $\Omega^*(S) \dashv \Omega^*(R)$.*

Proof. The first claim can easily be verified from the definition of the two functors. Then the second is an immediate consequence. Note, however, that because of contravariance the role of lower and upper adjoint are reversed. □

Upper adjoints have a very concise characterization:

Theorem 4.7 *A closed relation $R\colon X \multimap Y$ has a lower adjoint if and only if it is perfect and functional.*

Proof. From the previous lemma we know that R has a lower adjoint if and only if $\Omega^*(R)$ has an upper adjoint. As we know, $\Omega^*(R)$ is a continuous semilattice homomorphism and as a monotone function between the complete lattices $\Omega^*(Y) = \Omega(Y)$ and $\Omega^*(X) = \Omega(X)$ it is a lower adjoint if and only if it preserves all suprema. By Proposition 3.6 this is the case precisely when R is functional.

In this case we have an upper adjoint $u\colon \Omega^*(X) \to \Omega^*(Y)$, but it need not be a continuous semilattice homomorphism. As an upper adjoint it preserves all infima, but it is Scott continuous if and only if its adjoint $\Omega^*(R)$ preserves the way-below relation (see [1, Proposition 3.1.14]). From Proposition 4.2 we know that this is equivalent to R being perfect. □

Using Proposition 4.3 above we can rephrase this as follows.

Corollary 4.8 *A closed relation has a lower adjoint if and only if it is functional and the corresponding function is patch continuous, i.e. continuous with respect to the patch topologies.*

In the case of Hausdorff spaces the last condition is trivially true since the patch topology is simply the original topology. Hence, we get the following result.

Corollary 4.9 *A closed relation between compact Hausdorff spaces is a continuous function if and only if it has a lower adjoint in SCS^*.*

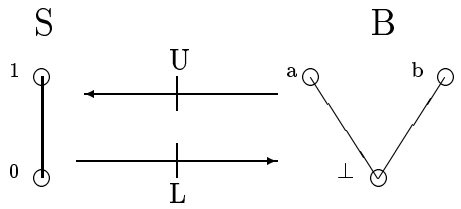


Fig. 1. A non-functional embedding retraction pair.

Consider the two posets given in Figure 1. We define two closed relations $L := \{0\} \times B \cup \{1\} \times \{a, b\}$ and $U := \{\bot\} \times S \cup \{a, b\} \times \{1\}$ which is the hypergraph of the function that maps $\bot$ to 0 and identifies a and b by mapping them to 1. We have $L\,;U = id_S$ and also $U\,;L \sqsubseteq id_B$ which shows that they form a embedding-projection pair in the sense that L is a lower adjoint section and U the corresponding upper adjoint retraction. This example shows that embeddings need not be functional.

We can, however, say explicitly what this lower adjoint does. Essentially it is just taking preimages under the function corresponding to its adjoint:

Proposition 4.10 *Let $u\colon X \to Y$ be a perfect continuous function between stably compact spaces, $U\colon X \multimap Y$ its hypergraph and L the lower adjoint. Then we have*

$$y\ L\ x \iff x \in (\forall U)[\uparrow y] \iff y \leq u(x)$$

and the corresponding multi-function $f_L\colon Y \to \mathcal{K}(X)$ satisfies

$$f_L(y) = u^{-1}[\uparrow y].$$

Proof. Note that we have $x \in (\forall U)[\uparrow y] \iff x \in u^{-1}[\uparrow y]$ by Lemma 3.2, and hence the descriptions of the adjoint given in the proposition agree.

We begin by showing that L is a closed relation. The easiest proof is to show that f_L is continuous: It factorizes as $Y \xrightarrow{\uparrow} \mathcal{K}(Y) \xrightarrow{u^{-1}[\cdot]} \mathcal{K}(X)$ where the first function is already known to be continuous. The spaces $\mathcal{K}(Y)$ and $\mathcal{K}(X)$ carry the Scott topology and directed suprema are given by filtered intersections which are preserved by the preimage function $u^{-1}[\cdot]$. So, f_L is a composition of continuous functions.

To show $L \dashv U$ we have to check $\sqsubseteq_X = \mathsf{id}_X \subseteq U;L$ and $L;U \subseteq \mathsf{id}_Y = \sqsubseteq_Y$ since the order on the homsets is reversed inclusion. So, for $x \sqsubseteq x'$ we have $x\ U\ u(x)\ L\ x'$ since $u(x) \sqsubseteq u(x')$. For the second inclusion, $y\ L\ x\ U\ y'$ implies $y \sqsubseteq u(x) \sqsubseteq y'$.

□

5 Bilimits

As our final topic we consider bilimits in SCS^*. In domain theory such bilimits are usually taken over directed diagrams of embedding-projection pairs. As pointed out in [1] the construction doesn't depend on the fact that the morphisms are sections and retractions but exclusively on the properties of the adjunctions. Hence, we discuss the construction of bilimits using this setup.

Both SCS^* and SCF^* are order enriched categories and support the notion of an adjoint pair. We denote the subcategories of lower adjoints by SCS^*_l and SCF^*_l, respectively. The dual categories of upper adjoints are denoted by SCS^*_u and SCF^*_u.

In the following we discuss bilimits of directed diagrams of adjoint closed relations between stably compact spaces, or to be more precise, colimits for functors from a directed poset I to the subcategory of lower adjoint closed relations SCS^*_l.

Theorem 5.1 *Every directed diagram in SCS^*_l has a bilimit.*

This means that it has a colimit which is also a colimit for the whole category SCS^*. Moreover, the corresponding upper adjoints for the colimiting cocone make it into limit for the upper adjoints of the diagram and this is also a limit in the ambient category SCS^*.

Proof. We prove this via the Stone dual. So let I be a directed set and $D\colon I \to \mathsf{SCS}^*_l$ a directed diagram. We consider the composition $\Omega^* \circ D \to \mathsf{SCF}^*_u$ where we denote the objects as $L_i := \Omega^*(D(i))$ and the morphisms as ϕ_{ji} and their upper adjoints as ψ_{ij}. Such a diagram can be considered to consist of dcpo's and Scott-continuous maps. Hence the general domain theoretic machinery can be brought to bear, cf. [1, Section 3.3] and [8, Section IV-3]. From this we know that the (domain-theoretic) bilimit is given by

$$\Big\{(x_i)_{i\in I} \in \prod_{i\in I} L_i \;\Big|\; (\forall i < j)\, \psi_{ij}(x_j) = x_i\Big\}$$

and that the (Scott-continuous) maps $\psi_j : L \to L_j$, $\psi_j((x_i)_{i\in I}) = x_j$ form a limiting cone over the diagram $((L_i)_{i\in I}, (\psi_{ij})_{i\le j})$ in the category DCPO. Furthermore, the (Scott-continuous) maps $\phi_i\colon L_i \to L$, $\phi_i(x) = \big(\bigsqcup^{\uparrow}{}_{k\ge i,j} \psi_{jk}(\phi_{ki}(x))\big)_{j\in I}$ form a colimiting cocone of the diagram $((L_i)_{i\in I}, (\phi_{ji})_{i\le j})$ in DCPO. The following relationships hold:

(i) For all $i \in I$, ϕ_i is a lower adjoint of ψ_i.

(ii) $\mathsf{id}_L = \bigsqcup^{\uparrow}{}_{i\in I} \phi_i \circ \psi_i$.

(iii) $(\forall i, j \in I)\, \psi_j \circ \phi_i = \bigsqcup^{\uparrow}{}_{k\ge i,j} \psi_{jk} \circ \phi_{ki}$.

(iv) For any cone $(M, (\mu_i)_{i\in I})$ (of Scott-continuous maps) over the diagram $((L_i)_{i\in I}, (\psi_{ij})_{i\le j})$ the mediating morphism $\mu\colon M \to L$ is given by $\mu = \bigsqcup^{\uparrow}{}_{i\in I} \phi_i \circ \mu_i$.

(v) For any cocone $(M, (\mu_i)_{i\in I})$ (of Scott-continuous maps) over the diagram $((L_i)_{i\in I}, (\phi_{ji})_{i\leq j})$ the mediating morphism $\mu\colon L \to M$ is given by $\mu = \bigsqcup^{\uparrow}_{i\in I} \mu_i \circ \psi_i$.

The objects and morphisms of the category SCF^* have additional structure, so we need to show the following:

(a) L is a complete lattice.

(b) L is continuous.

(c) L is distributive.

(d) The way-below relation on L is multiplicative and $\underline{1} \ll \underline{1}$.

(e) For all $i \in I$, ϕ_i and ψ_i preserve finite infima.

(f) Assuming that the cone (resp. cocone) maps preserve finite infima, so do the mediating morphisms.

For the sake of brevity, we will from now on write $\underline{x}$ for a sequence $(x_i)_{i\in I}$ wherever possible.

(a) The ψ_{ij}, as upper adjoints, preserve all infima. Hence these are calculated pointwise in L.

(b) Continuity follows for dcpo's already, see Theorem 3.3.11 in [1]. However, it will be necessary for the remaining claims to have a characterization of the way-below relation on L at hand. For this observe that the ϕ_i preserve way-below, [1, Proposition 3.1.14(2)]; we can therefore employ property 2 above to get $\underline{x} \ll \underline{y}$ iff there exists an index $j \in I$ and elements $x \ll y$ in L_i such that $\underline{x} \leq \phi_j(x) \ll \phi_j(y) \leq \underline{y}$.

We need to do (e) next: The ψ_i preserve infima because they are upper adjoints. For the lower adjoints we exploit the fact that finite meets commute with directed joints in continuous lattices, [8, Corollary I-2.2]. The claim then follows directly from the formula for the ϕ_i.

(c) We need to invoke the continuity of L for this: Assume $\underline{a} \ll \underline{x} \wedge (\underline{y} \vee \underline{z})$. Using the continuity of supremum and infimum we know that there are additional sequences $\underline{a'}$, $\underline{b}$ and $\underline{c}$ such that $\underline{a} \leq \underline{a'} \wedge (\underline{b} \vee \underline{c})$ and $\underline{a'} \ll \underline{x}$, $\underline{b} \ll \underline{y}$ and $\underline{c} \ll \underline{z}$. By our characterization of way-below on L it follows that we can find elements x, y, z in some approximating lattice L_j such that $\underline{a'} \leq \phi_j(x) \leq \underline{x}$, etc. Now we can calculate $\underline{a} \leq \underline{a'} \wedge (\underline{b} \vee \underline{c}) \leq \phi_j(x) \wedge (\phi_j(y) \vee \phi_j(z)) = \phi_j(x \wedge (y \vee z)) = \phi_j((x \wedge y) \vee (x \wedge z)) = (\phi_j(x) \wedge \phi_j(y)) \vee (\phi_j(x) \wedge \phi_j(z)) \leq (\underline{x} \wedge \underline{y}) \vee (\underline{x} \wedge \underline{z})$.

(d) This is similar to the previous item: For $\underline{x} \ll \underline{y}, \underline{z}$ find $x \ll y$, $x' \ll z$ in some L_j such that $\underline{x} \leq \phi_j(x) \ll \phi_j(y) \leq \underline{y}$ and $\underline{x} \leq \phi_j(x') \ll \phi_j(z) \leq \underline{z}$. The claim then follows from multiplicativity of $\ll$ in L_j: $\underline{x} \leq \phi_j(x) \wedge \phi_j(x') = \phi_j(x \wedge x') \ll \phi_j(y \wedge z) = \phi_j(y) \wedge \phi_j(z) \leq \underline{y} \wedge \underline{z}$.

For $\underline{1} \ll \underline{1}$ just observe that $1 \ll 1$ holds in each L_i and the lower adjoints are SCF^* maps, that is, they preserve the empty meet.

(f) Like (e), this follows from the defining formulas for mediating morphisms and the fact that finite meets commute with directed suprema. □

The limit-colimit coincidence for SCF^* which we established in the preceding proof says (among other things) that directed colimits in ${\mathsf{SCF}^*}_l$ are also colimits in the original category of semilattice homomorphisms. Both the diagram maps ϕ_{ji} and the cocone maps ϕ_i are in fact lower adjoints and consequently sup-preserving, which means that they are frame maps. Frame maps between continuous semilattices, however, are not necessarily lower adjoints. Nonetheless, directed colimits in ${\mathsf{SCF}^*}_l$ are also colimits of frames, as our next lemma shows.

Lemma 5.2 *The embedding of* ${\mathsf{SCF}^*}_l$ *into the category* Frm *of frames and frame homomorphisms preserves directed colimits.*

Proof. The colimit L of a directed diagram $((L_i)_{i\in I}, (\phi_{ji})_{i\leq j})$ in ${\mathsf{SCF}^*}_l$ as constructed in the proof of the previous theorem yields a distributive continuous lattice, hence a (spatial) frame, [8, Theorem 5.5]. The colimiting maps ϕ_i are lower adjoints in addition to being SCF^* morphisms, so they are frame homomorphisms. What needs to be shown is that the mediating morphism μ for a cocone $(\mu_i)_{i\in I}$ of frame homomorphisms is again a frame homomorphism. Since we already know that μ will be a continuous semilattice homomorphisms all that remains to be shown is preservation of (finite) suprema. The proof of this property is a beautiful interplay between formulas 2 and 3 from the preceding theorem. Let X be a set of elements of the colimit L. We calculate for the non-trivial inequality:

$$
\begin{aligned}
\mu(\bigsqcup X) &= \bigsqcup_{j\in I}^{\uparrow} \mu_j \circ \psi_j(\bigsqcup X) && \text{definition of } \mu \\
&= \bigsqcup_{j\in I}^{\uparrow} \mu_j \circ \psi_j(\bigsqcup_{\underline{x}\in X} \bigsqcup_{i\in I}^{\uparrow} \phi_i \circ \psi_i(\underline{x})) && \text{formula 2} \\
&= \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \mu_j \circ \psi_j(\bigsqcup_{\underline{x}\in X} \phi_i \circ \psi_i(\underline{x})) && \text{associativity} \\
&= \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \mu_j \circ \psi_j \circ \phi_i(\bigsqcup_{\underline{x}\in X} \psi_i(\underline{x})) && \phi_i\text{'s are lower adjoints} \\
&= \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \mu_j\big(\bigsqcup_{k\geq i,j}^{\uparrow} \psi_{jk} \circ \phi_{ki}(\bigsqcup_{\underline{x}\in X} \psi_i(\underline{x}))\big) && \text{formula 3} \\
&= \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \bigsqcup_{k\geq i,j}^{\uparrow} \mu_j \circ \psi_{jk}(\bigsqcup_{\underline{x}\in X} \phi_{ki} \circ \psi_i(\underline{x})) && \begin{array}{l}\mu_j\text{'s are continuous \&} \\ \phi_{ki}\text{'s are lower adjoints}\end{array} \\
&= \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \bigsqcup_{k\geq i,j}^{\uparrow} \mu_k \circ \phi_{kj} \circ \psi_{jk}(\bigsqcup_{\underline{x}\in X} \phi_{ki} \circ \psi_{ik} \circ \psi_k(\underline{x})) && \text{(co)cone condition} \\
&\leq \bigsqcup_{j\in I}^{\uparrow} \bigsqcup_{i\in I}^{\uparrow} \bigsqcup_{k\geq i,j}^{\uparrow} \mu_k(\bigsqcup_{\underline{x}\in X} \psi_k(\underline{x})) && \text{adjointness of } \phi \text{ and } \psi \\
&= \bigsqcup_{k\in I}^{\uparrow} \mu_k(\bigsqcup_{\underline{x}\in X} \psi_k(\underline{x})) && \text{redundant indices}
\end{aligned}
$$

$$
\begin{aligned}
&= \bigsqcup_{k\in I}^{\uparrow} \bigsqcup_{\underline{x}\in X} \mu_k \circ \psi_k(\underline{x}) && \mu_k\text{'s are frame maps}\\
&= \bigsqcup_{\underline{x}\in X} \bigsqcup_{k\in I}^{\uparrow} \mu_k \circ \psi_k(\underline{x}) && \text{associativity}\\
&= \bigsqcup_{\underline{x}\in X} \mu(\underline{x}) && \text{definition of } \mu
\end{aligned}
$$

□

Theorem 5.3 *The functor U from SCS^*_u to SCS preserves inverse limits.*

Proof. The dual equivalence between SCS^*_u and ${\mathsf{SCF}^*}_l$ transforms inverse limits into direct colimits. The latter are preserved by the inclusion of ${\mathsf{SCF}^*}_l$ into Frm according to the preceding lemma. Stone duality translates them into inverse limits in Top. □

The reader may still feel a bit numb from all these calculations and not immediately recognize the force of this theorem. Let us therefore elaborate on its content a little bit. Top is a complete category and limits are calculated in the usual way: If $D\colon \mathsf{I} \to \mathsf{Top}$ is a functor (for any diagram D) then the points of $\lim D$ are given by *threads*:

$$\lim D = \Big\{(x_i)_{i\in \mathrm{obj(I)}} \in \prod_{i\in \mathrm{obj(I)}} D(i) \;\Big|\; (\forall (f\colon i \to j) \in \mathrm{mor(I)})\ \mathrm{D(f)(x_i) = x_j}\Big\}$$

The topology is inherited from the product space $\prod_{i\in \mathrm{obj(I)}} D(i)$. Upper adjoint relations between stably compact spaces are functional and the functor U associates with every such relation the generating (perfect) function. Theorem 5.3 then states that a bilimit in N^* is calculated topologically as the limit of the corresponding inverse diagram of perfect maps. One can turn this around and say that the content of the theorem is to recognize inverse limits of perfect maps as bilimits in an order-enriched setting, yielding a limit-colimit coincidence with respect to closed relations. This appears to be an important first step in making stably compact spaces a suitable universe for semantic interpretations.

Acknowledgements

The authors are grateful for the many comments they received when parts of this research were presented at earlier occasions. Special thanks go to Martín Escardó for his careful reading of a draft version of this paper.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3,

pages 1–168. Clarendon Press, 1994.

[2] B. Banaschewski. Another look at the Localic Tychonoff Theorem. *Commentationes Mathematicae Universitatis Carolinae*, 29:647–656, 1988.

[3] R. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

[4] N. Bourbaki. *General Topology*. Elements of Mathematics. Springer Verlag, 1989.

[5] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing Science. Springer Verlag, 1996.

[6] M. H. Escardó. The regular-locally-compact coreflection of stably locally compact locale. *Journal of Pure and Applied Algebra*, 157(1):41–55, 2001.

[7] P. J. Freyd and A. Scedrov. *Categories, Allegories*. North-Holland, 1990.

[8] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer Verlag, 1980.

[9] R.-E. Hoffmann. The Fell compactification revisited. In R.-E. Hoffmann and K. H. Hofmann, editors, *Continuous Lattices and their Applications, Proceedings of the third conference on categorical and topological aspects of continuous lattices (Bremen 1982)*, volume 101 of *Lecture Notes in Pure and Applied Mathematics*, pages 57–116. Marcel-Dekker, 1985.

[10] K. H. Hofmann. Stably continuous frames and their topological manifestations. In H. L. Bentley, H. Herrlich, M. Rajagopalan, and H. Wolff, editors, *Categorical Topology, Proceedings of 1983 Conference in Toledo*, volume 5 of *Sigma Series in Pure and Applied Mathematics*, pages 282–307, Berlin, 1984. Heldermann.

[11] K. H. Hofmann and M. Mislove. Local compactness and continuous lattices. In B. Banaschewski and R.-E. Hoffmann, editors, *Continuous Lattices, Proceedings Bremen 1979*, volume 871 of *Lecture Notes in Mathematics*, pages 209–248. Springer Verlag, 1981.

[12] P. T. Johnstone. Factorization and pullback theorems for localic geometric morphisms. Technical Report 79, Université catholique de Louvain, 1979.

[13] A. Jung. The classification of continuous domains. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 35–40. IEEE Computer Society Press, 1990.

[14] A. Jung, M. Kegelmann, and M. A. Moshier. Multi lingual sequent calculus and coherent spaces. *Fundamenta Informaticae*, 37:369–412, 1999.

[15] A. Jung and Ph. Sünderhauf. On the duality of compact vs. open. In S. Andima, R. C. Flagg, G. Itzkowitz, P. Misra, Y. Kong, and R. Kopperman, editors, *Papers on General Topology and Applications: Eleventh Summer Conference at the University of Southern Maine*, volume 806 of *Annals of the New York Academy of Sciences*, pages 214–230, 1996.

[16] A. Jung and R. Tix. The troublesome probabilistic powerdomain. In A. Edalat, A. Jung, K. Keimel, and M. Kwiatkowska, editors, *Proceedings of the Third Workshop on Computation and Approximation*, volume 13 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1998. 23 pages, URL: `http://www.elsevier.nl/locate/entcs/volume13.html`.

[17] M. Kegelmann. *Continuous domains in logical form.* PhD thesis, School of Computer Science, The University of Birmingham, 1999.

[18] K. Keimel and J. Paseka. A direct proof of the Hofmann-Mislove theorem. *Proceedings of the AMS*, 120:301–303, 1994.

[19] H. P. A. Künzi and G. C. L. Brümmer. Sobrification and bicompletion of totally bounded quasi-uniform spaces. *Mathematical Proceedings of the Cambridge Philosophical Society*, 101:237–246, 1987.

[20] J. D. Lawson. The duality of continuous posets. *Houston Journal of Mathematics*, 5:357–394, 1979.

[21] L. Nachbin. *Topology and Order.* Von Nostrand, Princeton, N.J., 1965. Translated from the 1950 monograph "Topologia e Ordem" (in Portugese). Reprinted by Robert E. Kreiger Publishing Co., Huntington, NY, 1967.

[22] A. Schalk. *Algebras for Generalized Power Constructions.* Doctoral thesis, Technische Hochschule Darmstadt, 1993. 174 pp.

[23] A. K. Simpson. Recursive types in Kleisli categories. Manuscript (available from `http://www.dcs.ed.ac.uk`), 1992.

[24] M. B. Smyth. Stable compactification I. *Journal of the London Mathematical Society*, 45:321–340, 1992.

[25] Ph. Sünderhauf. Constructing a quasi-uniform function space. *Topology and its Applications*, 67:1–27, 1995.

[26] S. J. Vickers. *Topology Via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

MFPS 17 Preliminary Version

A Game Semantics of Idealized CSP

J. Laird

COGS, University of Sussex
Brighton BN1 9QH, UK
E-mail: jiml@cogs.susx.ac.uk

Abstract

A semantics is described for a typed functional language which includes primitives for parallel composition and for synchronous communication along private channels. The category of games on which this semantics is based is an extension of that introduced by Hyland and Ong, with multiple threads of control represented using a new notion of "concurrency pointer" together with relaxation of the condition which requires alternation of Player and Opponent moves. The semantics is proved to be fully abstract for "channel-free" types with respect to a *may-and-must* notion of operational equivalence, using factorization results to reduce definability to the sequential case.

1 Introduction

Game semantics has been successfully used to give models of various sequential programming languages, with the distinctive feature that many of the underlying notions are intensional, combining ideas from concurrency theory with those from traditional domain theory. A hierarchy of fully abstract models has emerged, based on the Hyland-Ong (HO) model of PCF [11] extended with non-functional (but still sequential) features such as mutable state and store [1,4] and control [12], to the point where there is now a thorough analysis of *sequential* functional computation which is to a large extent *based on concurrency* (not to mention "concurrent games" [3] — we shall be using the traditional token-based approach to game semantics here). Extending existing games models to give an interpretation of concurrency features is therefore a natural development.

This paper describes such an extension of HO games to model (a synchronous version of) Brookes' Idealized CSP [6]; a typed call-by-name λ-calculus with arithmetic, a parallel composition operator, local declaration of channel names and operations `send` and `recv` for (ground-type) message passing. A functor-category semantics of the synchronous language is described in [6]. The semantic analysis provided by the games model is complementary;

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

its distinguishing feature is that it captures observational equivalence (with respect to may-and-must testing), as we prove with a full abstraction result. (Although this is restricted to types which do not include occurrences of the base type of channels it could be extended to all types by adding a "bad channel" constructor to the language, analogous to the "bad variable" constructor `mkvar` which has been added to Idealized Algol for the same purpose [1].)

The basic framework of Hyland-Ong games gives us an interpretation of the λ-calculus. The extension to model concurrency has three key features; *multi-threading, synchronous message-passing* and *non-determinism.*

The interpretation of multiple threads of control requires the most radical extension to previous work in game semantics. It is based upon adding a new set of "concurrency pointers" (in addition to the existing "justification pointers") to the sequences which represent interaction of strategies. The new pointers allow a "control thread" to be extracted for each move. This allows all of the conditions which constrain sequences in HO games — *well-bracketing*, *visibility* and (in a new departure) *alternation* — to be relaxed with the result being total chaos; each control thread must satisfy all of these conditions, even though the sequence as a whole may not. There is a natural operation of parallel composition of strategies in the multi-threaded setting, based on interleaving of control threads. Moreover, we show (by factorization) that all branching of control threads in finitary strategies can be obtained from parallel composition.

Message passing along private channels is interpreted in "object-oriented" style, as in the functor-category model [6], and the games models of locally bound references [1,4] and exceptions [13]. In fact the interpretation of the `chan` type as the products of its "methods", `send` and `recv`, is precisely the same as the interpretation of the `var` and its methods (assignment and dereferencing) in the games model of Idealized Algol [1]. Thus the only difference between the semantics of the two features is in the interpretations of the new-variable and new-channel declarations. Both are given in terms of composition with a strategy which violates some of the constraints obeyed by elements corresponding to purely functional terms, and captures the causal relationships between writing and reading or sending and receiving. The key property of the "new-channel" strategy is that it contains a synchronous communication between control threads, and as we show by a factorization result, it can be used to generate all such behaviour in finitary strategies in the model.

Non-determinism arises quite naturally in the semantics, being implicit in the interpretation of message passing. A game semantics of Idealized Algol with explicit non-determinism has been given by Harmer and McCusker [7], fully abstract for a "may and must equivalence". The approach to non-determinism taken here has many features in common with that work; it leads to a similar may-and-must full abstraction result, for instance. However, many of the details are different, as the representations of divergence used in [7] apply only in a sequential setting.

The remainder of the paper is organized as follows. In section 2, a syntax, operational semantics and notion f observational equivalence are given for ICSP. Section 3 defines the category of "multi-threaded games" which is the basis for the model. Section 4 gives the denotational semantics of ICSP in this category, and shows that it is sound. Section 5 uses two factorization results to show that every finitary strategy in the model is definable as a term of ICSP, from which full abstraction follows.

2 Syntax and operational semantics of Idealized CSP

Idealized CSP [6], or ICSP, is based on the call-by-name λ-calculus, with types generated from a basis consisting of `comm` (commands), `nat` (natural numbers) and `chan` (channels for sending and receiving natural numbers).

$$T ::= \mathtt{comm} \mid \mathtt{nat} \mid \mathtt{chan} \mid T \Rightarrow T$$

Terms are formed according to the following grammar:

$$M ::= x \mid \mathtt{skip} \mid 0 \mid \mathtt{succ}\ M \mid \mathtt{pred}\ M \mid \ \mathrm{IF0}\ M\, M\, M \mid \lambda x.M \mid M\, M$$

$$\mathtt{nil} \mid M; M \mid M \| M \mid \mathtt{newchan}\ M \mid \mathtt{send}\, M\, M \mid \mathtt{recv}\, M$$

The language combines the λ-calculus with concurrency primitives (based on CSP [9]) much as Idealized Algol [18] does with imperative features (an intellectual debt acknowledged in its name); declaring a new channel name and sending and receiving on it are analogous to declaring, allocating and deallocating a variable. Typing judgements for the concurrency features are as follows:

$$\frac{\Gamma \vdash M{:}B \quad \Gamma \vdash N{:}B}{\Gamma \vdash M \| N{:}B}\ B = \mathtt{nat} \mid \mathtt{comm} \qquad \frac{\Gamma \vdash M{:}\mathtt{chan} \Rightarrow B}{\Gamma \vdash \mathtt{newchan}\, M{:}B}$$

$$\frac{\Gamma \vdash M{:}\mathtt{chan} \quad \Gamma \vdash N{:}\mathtt{nat}}{\Gamma \vdash \mathtt{send}\, M\, N{:}\mathtt{comm}} \qquad \frac{\Gamma \vdash M{:}\mathtt{chan}}{\Gamma \vdash \mathtt{recv}\, M{:}\mathtt{nat}}$$

Parallel composition of terms $M \| N$ is evaluated by splitting the control thread in two and (concurrently) evaluating M in one thread N in the other. The operation `newchan` allows local or private channels to be declared — $\mathtt{newchan}\, M$ supplies a new channel name as an argument to $M : \mathtt{chan} \Rightarrow B$ and adds it to the environment. Evaluation of `send` is by reducing its second argument to a value (numeral) and then its first argument to a channel name, `recv` evaluates its argument to a channel name, and $\mathtt{send}\, a\, v$ and $\mathtt{recv}\, a$ reduce in parallel to `skip` and v respectively. The key difference from the language described in [6] is that message passing is synchronous. We have also omitted local, assignable, ground type references á la Idealized Algol (adding these to our model by following [1] is straightforward) and, more significantly, any form of recursion.

The operational semantics is given in terms of a "small-step" reduction

relation on configurations of ICSP.A configuration $C = N_1, \ldots, N_k$ is a multiset of *threads* — ICSP programs of the same (base) type — containing the free channel names $\mathrm{Ch}(C)$. The reduction rules use the notion of *evaluation context* to pick out a unique next redex for each thread which is not a value.

Definition 2.1 Evaluation contexts are given by the following grammar:

$$E[\cdot] ::= [\cdot] \mid E[\cdot]\, M \mid \mathtt{pred}\ E[\cdot] \mid \mathtt{succ}\ E[\cdot] \mid \mathrm{IF0}\ E[\cdot]\, M\, N$$
$$E[\cdot]; M \mid \mathtt{send}\, M\, E[\cdot] \mid \mathtt{send}\, E[\cdot]\, n \mid \mathtt{recv}\, E[\cdot]$$

The (non-arithmetic) small-step reduction rules are as follows:

$$E[\lambda x.M\ N], C \longrightarrow E[M[N/x]], C$$
$$E[\mathtt{skip}; M], C \longrightarrow E[M], C$$
$$E[M \| N], C \longrightarrow E[M], E[N], C$$
$$E[\mathtt{newchan}\, M], C \longrightarrow E[M\, c], C \ : \ c \notin Ch(E[\mathtt{newchan}\, M], C)$$
$$E[\mathtt{send}\, a\, n], E'[\mathtt{recv}\, a], C \longrightarrow E[\mathtt{skip}], E[n], C.$$

The fact that evaluation of a configuration always terminates can be established using a standard "computability predicate" based proof.

Proposition 2.2 *There is no infinite series of configurations $C_1, C_2, \ldots C_n, \ldots$ such that $C_1 \longrightarrow C_2 \longrightarrow \ldots \longrightarrow C_n \longrightarrow \ldots$.* □

Various programming constructs such as Algol-style store and non-deterministic choice can be expressed in ICSP. A useful example; for any $n > 0$ define the program $\mathtt{oracle}_n$ which erratically produces one of the numbers less than n.

$$\mathtt{oracle}_n = \mathtt{newchan}\ \lambda c.((\mathtt{send}\, c\, 0 \| \mathtt{send}\, c\, 1 \| \ldots \| \mathtt{send}\, c\, (n-1)); \mathtt{nil} \| \mathtt{recv}\, c)$$

For examples of solutions to more subtle programming problems involving concurrency see e.g. [6].

2.1 Convergence testing and operational equivalence

Observational equivalence is defined with respect to a simple test — having at least one convergent thread.

Definition 2.3 Define the predicate $\Downarrow^{may}$ (may convergence) on configurations of type `comm` as follows:

$$\frac{}{C, \mathtt{skip} \Downarrow^{may}} \qquad \frac{\exists C'.C \to C' \wedge C' \Downarrow^{may}}{C \Downarrow^{may}}$$

Terms $M, N : T$ are *may-equivalent* if for all program contexts $C[\cdot] : \mathtt{comm}$, $C[M] \Downarrow^{may}$ if and only if $C[N] \Downarrow^{may}$.

May-equivalence of an inherently non-deterministic language such as ICSP is rather weak — it leads to a failure to distinguish programs which always

converge from those which may converge or diverge [8,7]. Our model will reflect *may-and-must* equivalence in a similar way to [7].

Definition 2.4 Define the predicate $\Downarrow^{must}$ (must convergence) on configurations:

$$\frac{}{C, \mathtt{skip} \Downarrow^{must}} \qquad \frac{\forall C'.(C \to C') \Longrightarrow C' \Downarrow^{must}}{C \Downarrow^{must}}$$

Terms $M, N : T$ are *must-equivalent* if for all closing contexts $C[\cdot] : \mathtt{comm}$, $C[M] \Downarrow^{must}$ if and only if $C[N] \Downarrow^{must}$. We shall write $M \simeq_T^{M\&M} N$ if M and N are may-equivalent *and* must-equivalent.

3 Multi-threaded games

The games constructions which will be used to model ICSP are based on those given by Hyland and Ong [11] and Nickau [15] (and developed in [14,1,4,10]), in which states of the game are represented as *justified sequences* of moves. However, in order to model concurrency it is necessary to enrich this structure by adding a new notion of 'concurrency pointer' to justified sequences, which allows multiple threads of control to be represented in a single sequence.

The structure of a game (the moves, their labels, how they are related) is specified by its *arena*, defined essentially as in [11]. An *arena* is a triple $A = \langle M_A, \vdash_A \subseteq (M_A)_* \times M_A, \lambda_A : M_A \to \{Q, A\}\rangle$, where:

M_A is a set of tokens called moves,

$\vdash_A \subseteq (M_A)_* \times M_A$ is a relation called *enabling*, from which a *unique* polarity for all of the moves in M_A can be inferred using the rule:

v m is an O-move if it is *initial* (i.e. $* \vdash m$), or enabled by a P-move,
m is a P-move if it is enabled by an O-move,

$\lambda^A : M_A \to \{Q, A\}$ is a function which labels moves as *answers* (A) or *questions* (Q), such that every answer has a unique enabling move which is a question.

In a sequence of moves sa, a justification pointer for s is a pointer from a to some move in s which enables a. A *justified sequence* over an arena A is a sequence of elements of M_A in which each non-initial move a has a justification pointer. We shall write J_A for the set of justified sequences over A. The transitive closure of justification is referred to as *hereditary justification*.

Various *rules* — alternation, visibility and well-bracketing — constraining justified sequences have been introduced [11] for defining the set of *legal sequences* (i.e. valid Player-Opponent interactions) over an arena.

Definition 3.1 A justified sequence t satisfies the *alternation* condition if Opponent moves are always followed by Player moves, and vice-versa.

To define the *visibility* condition requires the notion of Player (and Opponent) *view*. These represent a certain kind of "relevant history of the sequence" from Player's (Opponent's) perspective.

Definition 3.2 The Player view $\ulcorner s \urcorner$ of a justified sequence s is a sequence with justification pointers, defined inductively on the length of s, as follows:

$\ulcorner \varepsilon \urcorner = \varepsilon$,

$\ulcorner sa \urcorner = \ulcorner s \urcorner a$, if a is a Player move,

$\ulcorner sa \urcorner = a$, if a is an initial Opponent move,

$\ulcorner sa \cdot tb \urcorner = \ulcorner s \urcorner ab$, if b is an Opponent move justified by a.

There is a dual notion of Opponent view, $\llcorner s \lrcorner$, defined:

$\llcorner \varepsilon \lrcorner = \varepsilon$,

$\llcorner sa \lrcorner = \llcorner s \lrcorner a$, if a is an Opponent move,

$\llcorner sa \cdot tb \lrcorner = \llcorner s \lrcorner ab$, if b is a Player move justified by a.

Definition 3.3 A justified sequence s obeys the *visibility* condition if the Player and Opponent views of every prefix of s are well-formed justified sequences.

The *bracketing condition* requires that each answer must be justified by the most recent open question.

Definition 3.4 Define the *pending questions* (a set containing at most one move) of a justified sequence as follows:

$\mathsf{pending}(\varepsilon) = \{\}$,

$\mathsf{pending}(sb) = \{b\}$, if $\lambda(b) = Q$,

$\mathsf{pending}(sbtc) = \mathsf{pending}(s)$ if $\lambda(c) = A$ and c is justified by b.

A single-threaded sequence s is *well-bracketed* if for every prefix $ratb \sqsubseteq s$, if $\lambda(b) = A$ and b is justified by a, then $\mathsf{pending}(sat) = \{a\}$.

The set L_A of *legal sequences* over an arena A consists of the set of justified sequences (over A) which obey the alternation, visibility and well-bracketing conditions.

3.1 Multi-threaded Sequences

We shall now define the *multi-threaded sequences* on which our model is based. They are formed by adding new "concurrency pointers" to justified sequences.

Definition 3.5 Let sa be a sequence of moves with justification pointers. A concurrency pointer for a is a pointer from a to some single (occurrence of a) move in s, which distinct from its justification pointer (if any). A move is *thread-initial* if it does not have a concurrency pointer. A justified sequence without concurrency pointers (that is, the standard notion of justified sequence used in [11]) is said to be "single-threaded".

By tracing back concurrency pointers, we can extract a series of single-threaded subsequences, or "control threads" from each multi-threaded sequence.

Definition 3.6 The *control thread* of the last move in a sequence s is defined by induction on length as follows:

$\mathsf{CT}(a) = a$, if a is thread-initial,

$\mathsf{CT}(sa \cdot tb) = \mathsf{CT}(sa)b$ if b has a concurrency pointer to a.

A *multi-threaded justified sequence* is a sequence with both concurrency and justification pointers, such that each thread is a justified sequence — i.e. every non-initial move has a justification pointer to a move in its control thread.

Definition 3.7 For an arena A, let MT_A be the set of multi-threaded sequences over A, and define the set of multi-threaded justified sequences as follows: $JM_A = \{s \in MT_A \mid \forall t \sqsubseteq s.\mathsf{CT}(t) \in J_A\}$.

To define a notion of multi-threaded *legal* sequence, we use the conditions of alternation, visibility and well-bracketing, applying them not to the sequence itself, but to its threads. Thus, in a new departure for games models, the condition of *alternation* has been relaxed.

Definition 3.8 Let $LM_A = \{s \in JM_A \mid \forall t \sqsubseteq s.\mathsf{CT}(t) \in L_A\}$.

Note that in a multi-threaded legal sequence, the concurrency pointers must obey the conditions which apply to justification pointers — i.e. Opponent moves point to Player moves and vice-versa, and only (initial) Opponent moves may be thread-initial (but initial moves need not, in general, be thread-initial).

Idealized CSP contains limited facilities for synchronizing events occurring in different threads; correspondingly, in the model, Player will have only limited power to observe and control the actual ordering of moves in different threads. Player can wait until a given O-move has occurred before giving a response (in the language it is possible to suspend evaluation of a thread until an event occurs in another thread). However, Player cannot force a P-move to occur before another (Player or Opponent) move, and cannot observe the order in which two contiguous O-moves occur. To reflect these constraints, we define a relation $\ll$, such that $s \ll t$ if s can be obtained from t by migrating P moves forward, and migrating O-moves back.

Definition 3.9 Let $\ll$ be the least preorder on multi-threaded legal sequences such that for all $sab \cdot t, sba \cdot t \in LM_A$ such that $\lambda^{OP}(a) = O$ or $\lambda^{OP}(b) = P$, $sab \cdot t \ll sba \cdot t$.

The original representation of sequential, deterministic strategies in [11], is as sets of even-length sequences in which the final move represents the response of Player to the preceding sequence. Harmer and McCusker [7] observe that

this form of representation is not sufficient to model a simple non-deterministic functional language up to may *and must* equivalence; for example, it identifies a strategy which always converges with one which may converge or diverge. The solution adopted in [7] is to represent each strategy using two sets of traces; a set of even-length sequences representing Player responses in the usual way, and a set of odd-length sequences representing the *divergences* of the strategy — positions in which it may fail to respond. Without the alternation condition, it is no longer the case that the space of positions is partitioned between those in which it is Opponent's turn to move, and those in which it is Player's turn to move. Thus it is not clear what should count as a divergence — Player might require several O-moves to prompt a response. Similarly, Player may a string of moves without waiting for Opponent's response. So we represent a multi-threaded strategy σ on an arena A as a set of legal sequences of A which σ *may* perform in conjunction wit an Opponent, and having done so *may* wait for further O-moves before doing anything else. We shall write T_σ for the set $\{s \in LM_A \mid \exists t \in \sigma. s \sqsubseteq t\}$ of reachable *traces* of σ.

Definition 3.10 A (multi-threaded) strategy $\sigma : A$ is a subset of LM_A subject to the following conditions:

- $\varepsilon \in \sigma$,
- σ is closed with respect to $\ll$ — $s \ll t$ and $t \in T_\sigma$ implies $s \in \sigma$.
- the extension of a reachable position with an O-move is reachable — if $s \in T_\sigma$, and a is an O-move such that $sa \in LM_A$ then $sa \in T_\sigma$,
- in any reachable position, σ must either play a P-move or wait for another O-move —if $s \in T_\sigma$ then either $s \in \sigma$ or there is some P-move a such that $sa \in T_\sigma$.

A single-threaded strategy on A is a subset of L_A satisfying these conditions (closure under $\ll$ is clearly trivial).

A category based on multi-threaded games can now be defined. Because no new structure at the level of arenas is required, the standard constructions — products and function-spaces — are unaffected.

Product For any set-indexed family of arenas $\{A_i \mid i \in I\}$, form the product $A = \Pi_{i \in I} A_i$ as follows:

- $M_{\Pi_{i \in I} A_i} = \coprod_{i \in I} M_{A_i}$,
- $\langle m, i \rangle \vdash_{\Pi_{i \in I} A_i} \langle n, j \rangle$ if $i = j$ and $m \vdash_{A_i} n$, and $* \vdash_{\Pi_{i \in I} A_i} \langle n, j \rangle$ if $* \vdash_{A_j} n$,
- $\lambda^{QA}_{\Pi_{i \in I} A_i}(\langle m, i \rangle) = \lambda_{A_i}(m)$.

We shall write A^k for $\Pi_{i=1}^k A$, and $\mathbf{1}$ for the empty game (0-ary product).

Function Space For arenas A_1, A_2, form $A_1 \Rightarrow A_2$ as follows:

- $M_{A_1 \Rightarrow A_2} = M_{A_1} + M_{A_2}$,
- $\langle m, i \rangle \vdash_{A_1 \Rightarrow A_2} \langle n, j \rangle$ if $i = j$ and $m \vdash n$
  or $m \in M_{A_2}$, $n \in M_{A_1}$ and $* \vdash_{A_2} m$ and $* \vdash_{A_1} n$,
  $* \vdash \langle m, i \rangle$ if $m \in M_{A_2}$ and $* \vdash_{A_2} m$,

- $\lambda^{QA}_{A_1 \Rightarrow A_2}(\langle m, i \rangle) = \lambda_{A_i}(m)$.

We can now define, in a standard fashion [5,11], a category $\mathcal{G}_{\mathcal{M}}$ of multi-threaded games, in which the objects are arenas, and the morphisms from A to B are the multi-threaded strategies on $A \Rightarrow B$. We shall write $\sigma : A$ for a morphism $\sigma : \mathbf{1} \to A$.

Composition in the category is defined using a notion of restriction on multi-threaded sequences, which "mends" both justification and concurrency pointers.

Definition 3.11 Given $s \in L_{(A_1 \Rightarrow A_2) \Rightarrow A_3}$, $s \upharpoonright (A_i, A_j)$ $(i < j)$ is a multi-threaded sequence on $A_i \Rightarrow A_j$ defined as follows:

$$\varepsilon \upharpoonright (A_i, A_j) = \varepsilon$$

$$sa \upharpoonright (A_i, A_j) = s \upharpoonright (A_i, A_j) \qquad \text{if } a \notin M_{A_i}, M_{A_j}$$

$$sa \upharpoonright (A_i, A_j) = (s \upharpoonright (A_i, A_j))a \quad \text{if } a \in M_{A_i}, M_{A_j}.$$

The justifier of a in $sa \upharpoonright (A_i, A_j)$ is the most recently played move from A_i or A_j which *hereditarily justifies* a. (If there is no such move, then a is initial.) The concurrency pointer from a points to the most recently played move (if any) from A_i or A_j which is in $\mathsf{CT}(sa)$.

Lemma 3.12 *If $s \in LM_{(A_1 \Rightarrow A_2) \Rightarrow A_3}$ then $\mathsf{CT}(s \upharpoonright (A_i, A_j)) = \mathsf{CT}(s) \upharpoonright (A_i, A_j)$.* □

As in other models, canonical morphisms are *copycat* strategies which simply copy Opponent moves between different parts of a game. (However, there is a difference in the treatment of concurrency and justification pointers; the copy of an O-move a has a concurrency pointer to a itself, and a justification pointer to the move of which the justifier of a was a copy.) The identity strategy is the prime example.

Definition 3.13 Say that $s \in LM_{A \Rightarrow A}$ is a *copycat* sequence if for all $t \sqsubseteq^{even} s$, $t \upharpoonright A^- = t \upharpoonright A^+$, and every P-move in s has a concurrency pointer to the preceding O-move. Then $\mathtt{id}_A = \{s \in LM_{A \Rightarrow A} \mid \exists t. s \ll t \wedge t \text{ is a copycat}\}$.

Composition of strategies is obtained as the standard 'parallel composition with hiding' [5].

Definition 3.14 [Composition of multi-threaded strategies] Given $\sigma : A_1 \to A_2$ and $\tau : A_2 \to A_3$, let $\sigma; \tau = \{t \in LM_{A_1 \Rightarrow A_3} \mid \exists s \in LM_{(A_1 \Rightarrow A_2) \Rightarrow A_3}. t = (s \upharpoonright A_1, A_3) \wedge (s \upharpoonright A_1, A_2) \in \sigma \wedge (s \upharpoonright A_2, A_3) \in \tau\}$

Lemma 3.15 *If $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, then $\sigma; \tau : A \Rightarrow C$.*

Proof. The key points are that if $s \in LM_{(A_1 \Rightarrow A_2) \Rightarrow A_3}$, then $s \upharpoonright A_i, A_j$ is a legal sequence, and closure under $\ll$. The former follows from the single-threaded case [11], using Lemma 3.12. The latter is proved by showing that, given $s \in LM_{(A_1 \Rightarrow A_2) \Rightarrow A_3}$ and $t \in LM_{A_1 \Rightarrow A_3}$, such that $s \upharpoonright A_1, A_3 \ll t$, we can find $r \in LM_{(A_1 \Rightarrow A_2) \Rightarrow A_3}$ such that $r \upharpoonright A_1, A_3 = t$, $s \upharpoonright A_1, A_2 \ll r \upharpoonright A_1, A_2$ and

$s\restriction A_2, A_3 \ll r\restriction A_2, A_3$. We derive r from s by migrating P-moves in A_3 forward, and O-moves in A_1 back, and swapping contiguous O-moves and contiguous P-moves. □

The proof that composition is associative follows the standard argument from [11], which is little altered by the presence of concurrency pointers and absence of the alternation condition.

Proposition 3.16 *For all* $\sigma : A \Rightarrow B, \tau : B \Rightarrow C, \rho : C \Rightarrow D$ $\sigma;(\tau;\rho) = (\sigma;\tau);\rho$. □

Moreover, the SMCC structure on single-threaded games [14] transfers smoothly to $\mathcal{G}_{\mathcal{M}}$.

Proposition 3.17 $\mathcal{G}_{\mathcal{M}}, \mathbf{1}, \times, \Rightarrow$ *is a symmetric monoidal closed category.* □

To construct a CCC we consider a subcategory of $\mathcal{G}_{\mathcal{M}}$ in which morphisms are *well-opened* strategies (as in the original HO model).

Definition 3.18 For a (multi-threaded legal) sequence s containing an initial move a, define $s\restriction a$ to be the the (multi-threaded legal) subsequence of s hereditarily justified by a:

$sa\restriction a = a$,

$sb\restriction a = (s\restriction a)b$ if a hereditarily justifies b (b has a concurrency pointer to a move in $s\restriction b$ by visibility),

$sb\restriction a = s\restriction a$ otherwise.

Definition 3.19 A multi-threaded legal sequence is *well-opened* if it contains at most one initial move. A well-opened strategy σ is a set of well-opened sequences satisfying the conditions laid down in Definition 3.10 with respect to well-opened sequences rather than legal sequences.

The well-opened strategies form a category with finite products given by the product of arenas (for example, the well-opened identity strategy on A, $\mathtt{Id}_A$, is the restriction of $\mathtt{id}_A$ to well-opened sequences).

Definition 3.20 For a well-opened strategy $\sigma : A$, define the strategy $\sigma^\dagger : A$:

$s \in \sigma^\dagger$ if and only if for all initial moves a in s, $s\restriction a \in \sigma$.

For the definability and full abstraction results we will require a further restriction on strategies; Player cannot (directly) observe the concurrency pointers from O-moves. This corresponds to the fact that in ICSP it is not directly observable in which *location* (i.e. which thread) computation is occurring. The simplest example of this is that $\mathtt{newchan}\,\lambda c.\lambda x.((\mathtt{send}\,c\,n;\mathtt{nil})\|\mathtt{recv}\,c)$ is observationally equivalent to $\lambda x.x$.

Definition 3.21 Let $\sim_O$ to be the least equivalence relation on multi-threaded *legal* sequences closed under the condition:

$sa_1t_1a_2t_2br \sim_O sa_1t_1a_2t_2br$, if b is an O-move with a concurrency pointer to a_1 in the first sequence, and to a_2 in the second sequence, and the sequences are otherwise identical.

A strategy σ is (concurrency) pointer-*blind* if it is closed with respect to $\sim_O$ — $s \in \sigma$ and $s \sim_O t$ implies $t \in \sigma$.

Lemma 3.22 *If $\sigma : A \to B, \tau : B \to C$ are pointer-blind then $\sigma;\tau$ is pointer-blind.* □

Define the category ${\mathcal{G}_{\mathcal{M}}}^\dagger$ of multi-threaded games with *arenas* as objects, and *well-opened, pointer-blind strategies on $A \Rightarrow B$* as morphisms from A to B, and composition defined $\tau \cdot \sigma = \sigma^\dagger;\tau$.

Proposition 3.23 *${\mathcal{G}_{\mathcal{M}}}^\dagger, \times, \mathbf{1}, \Rightarrow$ is a cartesian closed category.*

Proof. We use the following facts.
For any A, $\mathtt{Id}_A^\dagger = \mathtt{id}_A$, and for any well-opened σ, $\sigma^\dagger;\mathtt{Id}_B = \sigma$.
For well-opened strategies $\sigma : A \Rightarrow B, \tau : B \Rightarrow C$, $\sigma^\dagger;\tau^\dagger = (\sigma^\dagger;\tau)^\dagger$. □

Hence we have the basis for a model of the functional part of ICSP (the interpretation of the ground types `comm` and `nat` as "flat arenas" with a single initial question enabling one and countably many answers respectively is standard). We can also define properties of strategies (*sequentiality* and *innocence*) which pick out the elements in the multi-threaded model which are the denotations of the purely functional terms.

A sequential strategy is one which never spawns new threads of control.

Definition 3.24 A multi-threaded sequence s is *Player sequential* if every O-move in s is the target of at most one concurrency pointer: i.e. if $tarc, tar'c' \sqsubseteq s$ where c, c' are P-moves with concurrency pointers to a, then $rc = r'c'$.

A strategy σ is *sequential* if every $s \in T_\sigma$ is Player sequential.

The notion of *innocence* used in [11] etc. generalises readily to the multi-threaded framework; a (sequential) strategy is *innocent* if its response in each control-thread depends only on the P-view of that thread (Definition 3.2).

Definition 3.25 For a multi-threaded strategy σ define $\ulcorner\sigma\urcorner = \{\ulcorner \mathsf{CT}(s) \urcorner \mid s \in \sigma\}$. (If σ is single-threaded, $\ulcorner\sigma\urcorner = \{\ulcorner s \urcorner \mid s \in \sigma\}$.) A sequential strategy σ is *innocent* (and deterministic) if $\ulcorner s \urcorner$ is evenly branching: i.e. if $s, t \in \ulcorner\sigma\urcorner$ then if $s \sqcap t$ is odd-length then $s = t$.

Note that all innocent strategies are pointer-blind; if an O-move is in the $P-$view, then the preceding move is its justifier, which must be no later than the target of its justification pointer.

Definition 3.26 From a multi-threaded and innocent strategy σ, define the single-threaded innocent strategy $\mathsf{threads}(\sigma) = \{\mathsf{CT}(s) \mid s \in \sigma\}$.

The following lemma follows from Lemma 3.12.

Lemma 3.27 *For single-threaded* $\sigma : A \to B, \tau : B \to C$, $\mathsf{threads}(\sigma); \mathsf{threads}(\tau) = \mathsf{threads}(\sigma; \tau)$. □

Lemma 3.28 *Given a single-threaded and innocent strategy* $\sigma : A$, *define the multi-threaded strategy* $\mathsf{threads}^{-1}(\sigma) = \{s \in LM_A |\ \forall r \sqsubseteq s.\mathsf{CT}(r) \in T_\sigma \wedge \forall t.((t \ll s \vee s \ll t) \Longrightarrow \mathsf{CT}(t) \in \sigma)\}$.
Then $\mathsf{threads}(\mathsf{threads}^{-1}(\sigma)) = \sigma$, *and for any multi-threaded, sequential and innocent strategy* τ, $\mathsf{threads}^{-1}(\mathsf{threads}(\tau)) = \tau$. □

Hence $\mathsf{threads}$ is an isomorphism between the multi-threaded, innocent and sequential strategies, and the innocent single-threaded strategies.

Proposition 3.29 *The sequential and innocent strategies form a subcategory of* $\mathcal{G}_{\mathcal{M}}{}^{\dagger}$ *which is isomorphic to the category of single-threaded games and innocent strategies.* □

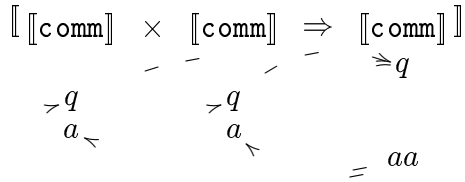
4 Semantics of ICSP

Parallel composition is interpreted using a corresponding operation on strategies which *interleaves* their responses to the initial move.

Definition 4.1 Say that an *arena* is well-opened if it has an unique initial move. Let $s = a_0 a_1 \ldots a_n$ and $t = b_0 b_1 \ldots b_n$ be well-opened sequences in LM_A, where A is a well-opened arena. A *tail-interleaving* of s and t is a sequence $ar \in LM_A$ such that r is an interleaving of $a_1 \ldots a_n$ with $b_1 \ldots b_n$ which preserves justification and concurrency pointers — i.e. if a_j points to a_0 in s then a_j points to a in ar, otherwise if a_j points to a_{i+1} in s, then a_{i+1} in a_r. We shall write $s|t$ for the set of tail-interleavings of s and t.

Proposition 4.2 *For any well-opened arena* A *and (pointer-blind) strategies* $\sigma, \tau : A$ *the parallel composition of* σ *and* τ *—* $\sigma|\tau = \bigcup\{s|t \mid s \in \sigma \wedge t \in \tau\}$ *— is a well-defined (pointer-blind) strategy.* □

So, for instance, we have a general *parallel composition* morphism for arenas with unique initial moves, $\mathsf{para}_A : A \times A \to A = \pi^l_A | \pi^r_A$. A typical play (with concurrency pointers) of $\mathsf{para}_{[\![\mathtt{comm}]\!]}$ is as follows (moves aligned horizontally can occur in either order):



Proposition 4.3 *If* A, B *are well-opened arenas and* $\sigma : A \to B$ *is a strict strategy then for any* $\tau, \rho : C \to A$, $(\tau|\rho); \sigma = (\tau; \sigma)|(\rho; \sigma)$. □

By defining $[\![\Gamma \vdash M || N : B]\!] = [\![\Gamma \vdash M]\!] | [\![\Gamma \vdash N]\!]$ we have an interpretation of the λ-calculus with parallel composition in $\mathcal{G}_{\mathcal{M}}{}^{\dagger}$. So it remains to give the

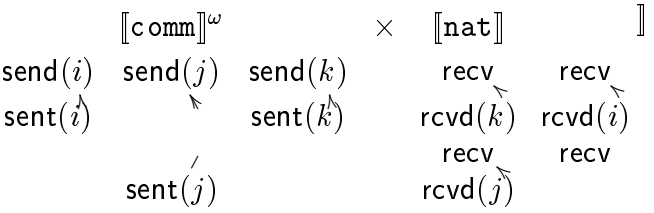
semantics of locally bound channels. This is based on viewing elements of type `chan` as 'objects' defined by their 'methods' — in this case `send` and `recv`. This was suggested as an interpretation for reference types by Reynolds [17] and used to give a a functor-category semantics for idealized CSP by Brookes [6]. The interpretation described here is particularly close to the game semantics of store in Idealized Algol [1].

- We define $[\![\mathtt{chan}]\!] = [\![\mathtt{comm}]\!]^\omega \times [\![\mathtt{nat}]\!]$ (which is the same as the interpretation of the type `var` given in [1]).
- For sending messages (and assigning to variables) there is a sequential and innocent strategy $\mathsf{write} : [\![\mathtt{nat}]\!] \times [\![\mathtt{comm}]\!]^\omega \to [\![\mathtt{comm}]\!]$ described in [1] which responds to the initial move by asking the question in $[\![\mathtt{nat}]\!]$, given the answer n it asks the initial question in the nth part of the product $[\![\mathtt{comm}]\!]^\omega$. When this is answered, it answers the initial question.
  We define $[\![\Gamma \vdash \mathtt{send}\, M\, N]\!] = \langle [\![\Gamma \vdash M]\!]; \pi_l, [\![\Gamma \vdash N]\!]\rangle; \mathsf{write}$, which is precisely the same as the interpretation of assignment in [1].
- We define $[\![\Gamma \vdash \mathtt{recv}\, M]\!] = [\![\Gamma \vdash M]\!]; \pi_r$, (the interpretation of deallocation in [1]).

These operations preserve innocence (and determinacy) and sequentiality, and hence all denotations of terms in ICSP - $\{||, \mathtt{newchan}\}$ satisfy these conditions. Thus we have the following definability result for innocent sequential strategies, which is a minor adaptation of the definability theorem for PCF [11], and precisely analogous to definability in Idealized Algol without bad variables.

Proposition 4.4 *If Γ is a* `chan`*-free context and T is a* `chan`*-free type, and $\sigma : [\![\Gamma, \mathtt{chan}^k]\!] \to [\![T]\!]$ is a sequential and innocent strategy such that $\ulcorner\sigma\urcorner$ is finite, then there is a term $\Gamma, \mathtt{chan}^k \vdash M_\sigma : T$ of ICSP - $\{||, \mathtt{newchan}\}$ such that $\sigma = [\![M_\sigma]\!]$.* □

Thus the only part of the semantics of channels which is non-functional — and moreover the only part which differs from the game semantics of store in Idealized Algol — is the new-channel generator. This can be defined by a (pointer-blind) strategy $\mathsf{ccell} : [\![\mathtt{nat}]\!] \times [\![\mathtt{comm}]\!]^\omega$ which is similar to the cell strategy used to interpret `new` in the model of Idealized Algol [1] in the way that it causes interaction between the two read/write or send/receive components of $[\![\mathtt{chan}]\!]$ (and violates innocence in the process) but the significant difference is that communication between sending and receiving is concurrent and synchronous rather than sequential. A typical play of ccell (with concurrency pointers) is given below (the questions in the ith "send component" $[\![\mathtt{comm}]\!]^\omega$ have been labelled $\mathsf{send}(i)$, and their answers as $\mathsf{sent}(i)$, the question in the "receive" component $[\![\mathtt{nat}]\!]$ has been labelled recv, and its ith answer $\mathsf{rcvd}(i)$).



Informally, the behaviour of ccell can be described in the following terms. It must respond to any play in which there is both an unanswered send(i) question and an unanswered recv question. In response to such a play ccell matches up any such pairs of questions by giving the answer rcvd(i) to the recv question, and the answer sent to send(i).

So ccell is implicitly non-deterministic, as answers can be exchanged between any pair of open `send` and `recv` moves. And in order to satisfy the visibility and alternation conditions, the `send` and `recv` moves must always be in different threads — as one would expect, as synchronous message passing requires the sender and recipient to be in different threads.

Definition 4.5 Formally, ccell can be defined as follows. Let the *balanced* sequences of ccell, $B_{\mathsf{ccell}} \subseteq \mathsf{ccell}$, be the least set of sequences containing ε and closed under the following rule:

if $s \in B_{\mathsf{ccell}}$, then $s \cdot \mathsf{send}(i) \cdot \mathsf{recv} \cdot \mathsf{sent} \cdot \mathsf{rcvd}(i) \in \mathsf{ccell}$ (where sent has concurrency and justification pointers to send(i), and rcvd(i) to recv).

Let the "waiting to send" sequences of ccell, S_{ccell}, be the least superset of B_{ccell} such that if $s \in S_{\mathsf{ccell}}$, then $s \cdot \mathsf{recv} \in \mathsf{ccell}$.

Similarly, the set of "waiting to send" sequences, R_{ccell}, is the least superset of B_{ccell} such that if $s \in R_{\mathsf{ccell}}$, then $s \cdot \mathsf{send}(i) \in \mathsf{ccell}$.

Now let $\mathsf{ccell} = \{s \in LM_{[\![\mathtt{chan}]\!]} \mid \exists t.s \ll t \wedge (t \in S_{\mathsf{ccell}} \vee t \in R_{\mathsf{ccell}})\}$

We define $[\![\Gamma \vdash \mathtt{newchan}\ M : B]\!] = ([\![\Gamma \vdash M : \mathtt{chan} \Rightarrow B]\!] \times \mathsf{ccell}); \mathsf{App}$.

4.1 Soundness of the semantics

A program denotation is *may convergent* if it can answer the initial question at least once. A denotation is *must convergent* if it always gives some response (i.e. does not wait) in response to Opponent's initial move.

Definition 4.6 Let q be the initial Opponent question in the arena $[\![\mathtt{comm}]\!]$, and a its answer. For a program $M : \mathtt{comm}$, define $M \downarrow^{may}$ if $qa \in T_{[\![M]\!]}$ and $[\![M]\!] \downarrow^{must}$ if $q \notin [\![M]\!]$.

Soundness of the interpretation — i.e. correspondence between the notions of may and must convergence in the operational and denotational semantics — can now be established. First, commutativity and associativity of parallel composition, and commutativity of new-channel declaration means that the definition of the denotation of a configuration can be given without ambiguity.

Definition 4.7 For a configuration $C = M_1 : \mathtt{comm}, \ldots, M_n : \mathtt{comm}$ such that $Ch(C) = c_1, \ldots, c_k$, define $[\![C]\!] = [\![\mathtt{newchan}\ \lambda c_1. \ldots \mathtt{newchan}\ \lambda c_k. M_1 || \ldots || M_n]\!]$.

Say that a configuration is converged if it has the form $C, \mathtt{skip}$, and deadlocked if it is not converged and cannot be further reduced.

Lemma 4.8 *If C is converged, then $[\![C]\!] \downarrow^{may}$ and $[\![C]\!] \downarrow^{must}$ and if C is deadlocked then $[\![C]\!] \not\downarrow^{may}$ and $[\![C]\!] \not\downarrow^{must}$.* □

Proposition 4.9 *For any non-converged and non-deadlocked configuration C, $[\![C]\!] \downarrow^{may}$ if and only if there is some C' such that $[\![C']\!] \downarrow^{may}$ and $C \longrightarrow C'$, and $[\![C]\!] \downarrow^{must}$ if and only if $[\![C']\!] \downarrow^{must}$ for all C' such that $C \longrightarrow C'$.*

Proof. The proof is based on the standard properties of a cartesian closed category, together with Proposition 4.3 and the following lemmas.

Lemma 4.10 $[\![(\mathtt{newchan}\ \lambda c.M)||N]\!] = [\![\mathtt{newchan}\ \lambda c.(M||N)]\!]$ $(c \notin FV(N)$
$[\![(\mathtt{newchan} \lambda c.M); N]\!] = [\![\mathtt{newchan}\ \lambda c.(M; N)]\!]$, $(c \notin FV(N))$
$[\![\mathtt{newchan}\ \lambda c.E[\mathtt{send}\ c\ v]||E'[\mathtt{recv}\ c]||M]\!] \subseteq [\![\mathtt{newchan}\ \lambda c.E[\mathtt{skip}]||E'[v]||M]\!]$. □

To prove must-soundness we also need the following lemma, which is based on analysis of the strategy ccell.

Lemma 4.11 *For terms $M_1, M_2, \ldots M_n$, define $||_{i \leq n} M_i = M_1||M_2|| \ldots ||M_n$. Suppose we have terms $M^i_j = E^i_j[\mathtt{send}\, a_i\, v^i_j] : j \leq m_i$ and $N^i_k = D^i_k[\mathtt{recv}\, a_i] : k \leq l_i$ for $i \leq n$. Then*

$$||_{i \leq n}((||_{j \leq m_i} M^i_j)||(||_{k \leq l_i} N^i_k)) = \bigcup_{I \leq n J \leq m_I K \leq l_J} ||_{i \leq n}((||_{j \leq m_i} M(I,J))^i_j)||(||_{k \leq l_i} N(I,J,K)^i_k)),$$

where

$$M(I,J)^i_j = \begin{cases} E^i_j[\mathtt{skip}], & \text{if } i = I \text{ and} \\ j = J & M(I,J)^i_j = M^i_j \quad \text{otherwise,} \end{cases}$$

$N(I,J,K)^i_k = E^i_k[v^I_J]$ *if $i = I$ and $k = K$, and*

$N(I,J,K)^i_k = N^i_k$. □

We can now prove Proposition 4.9. Suppose C is non-converged. Then by Lemma 4.10, if $C \longrightarrow C'$ then $[\![C']\!] \subseteq [\![C]\!]$, and hence if $[\![C]\!] \downarrow^{must}$ then $[\![C']\!] \downarrow^{must}$, and if $[\![C']\!] \downarrow^{may}$ then $[\![C]\!] \downarrow^{may}$. To prove that if $[\![C']\!] \downarrow^{must}$ for all C' such that $C \longrightarrow C'$ then $[\![C]\!] \downarrow^{must}$, suppose that there is some reduction $C \longrightarrow C'$ which is not an instance of the communication rule. Then by Lemma 4.10, $[\![C]\!] = [\![C']\!]$ and $[\![C]\!] \downarrow^{must}$ as required. On the other hand, if there is no reduction which is not an instance of communication, then all threads have the form $E[\mathtt{send}\ c\ n]$ or $E[\mathtt{send}\ c\ n]$ and hence by Lemma 4.11, $q \notin [\![C]\!] = \bigcup_{C' : C \longrightarrow C'} [\![C']\!]$ as required. □

Corollary 4.12 *If $M : \mathtt{comm}$ is a program of ICSP then $M \Downarrow^{may}$ if and only if $[\![M]\!] \downarrow^{may}$ and $M \Downarrow^{must}$ if and only if $[\![M]\!] \downarrow^{must}$.*

Proof. $M \Downarrow^{may}$ implies $[\![M]\!] \downarrow^{may}$ and $M \Downarrow^{must}$ implies $[\![M]\!] \downarrow^{must}$ by induction on reduction. The converse follows from the fact that evaluation always terminates (Proposition 2.2). □

5 Definability and full abstraction

To prove a full abstraction result for `chan`-free types it remains to show that every finitary strategy at these types is definable as a term. All strategies on arenas denoting types of rank 1 or above contain infinitely many sequences, so finiteness is defined in terms of the smallest set required to generate these sequences.

Definition 5.1 A *generator* for a strategy σ is a set of sequences $S \subseteq \sigma$ such that for all $t \in \sigma$ there exists $sr \ll t$ such that $s \in S$ and r contains only O-moves. We say that σ is *finitary* if it has a finite generator S, and that it is bounded by k if the length of all sequences in S is less than k.

Definability is reduced in two steps to the case of sequential and innocent (deterministic) strategies, which are definable in ICSP $-\{\texttt{newchan}, \|\}$, by the now-standard technique of *factorization*[1,12,4,7]. The first stage is factorization of each finitary multi-threaded strategy into the composition of a sequential strategy with parallel composition.

Definition 5.2 Suppose $satb \in LM_A$, where the concurrency pointer from b goes to a. Define $\mathsf{branches}(satb)$ to be the number of moves in $satb$ which point to b. For finitary σ, let $\mathsf{branches}(\sigma) = \max(\{\mathsf{branches}(sa) \mid sa \in T_\sigma \wedge \lambda^{OP}(a) = P\})$.

Thus σ is sequential if and only if $\mathsf{branches}(\sigma) \leq 1$.

Proposition 5.3 *Let $\sigma : A$ be a finitary strategy, such that $\mathsf{branches}(\sigma) = n$. Then there is a finitary sequential strategy $\mathsf{seq}(\sigma) : [\![\texttt{comm}]\!] \to A$ such that $[\![\|_{i \leq n} \texttt{skip}]\!]; \mathsf{seq}(\sigma) = \sigma$.*

Proof. The idea behind the factorization is simple. In response to each O-move a in A, $\mathsf{seq}(\sigma)$ makes a move with a concurrency pointer to a in `comm`, prompting Opponent (playing as $[\![\|_{i \leq n} \texttt{skip}]\!]$) to give n answers. Then $\mathsf{seq}(\sigma)$ plays as σ, except that where σ plays a move with a pointer a, $\mathsf{seq}(\sigma)$ plays the same move with a pointer to a fresh instance of one of these answers.

Suppose that $\forall s \sqsubseteq t.\mathsf{branches}(s) \leq n$. Define a Player-sequential sequence $\mathsf{seq}(t)$, by the following induction:

$\mathsf{seq}(\varepsilon) = \varepsilon$,

$\mathsf{seq}(sa) = \mathsf{seq}(s)aq\widehat{a_1}\widehat{a_2}\ldots\widehat{a_n}$ (where a is an O-move, q is the initial question in $[\![\texttt{comm}]\!]$ (pointing to a), and $\widehat{a_1}, \widehat{a_2}, \ldots \widehat{a_n}$ are n answers with pointers to q).

If c is a P-move pointing to b, and $\mathsf{branches}(sbtc) = i$ then $\mathsf{seq}(sbtc) = \mathsf{seq}(sbt)\widehat{c}$ (where $\widehat{c}$ points to $\widehat{b_i}$).

Thus $\mathsf{seq}(\_)$ is a map from LM_A to $LM_{[\![\mathtt{comm}]\!]\to A}$ with the following properties:

$\mathsf{seq}(s)$ is player-sequential, $\mathsf{seq}(s)\restriction A = s$ and

$\mathsf{seq}(s)\restriction[\![\mathtt{comm}]\!] \in [\![||_{i\leq n}\mathtt{skip}]\!]$.

Let $\mathsf{seq}(\sigma) = \{t \in LM_{[\![\mathtt{comm}]\!]\Rightarrow A} \mid \exists s \in \sigma.t \ll \mathsf{seq}(s)\}$. □

The factorization of sequential strategies into innocent strategies via the `chan` type and ccell strategy is more complex. It resembles the factorization of innocence via the cell strategy [1] in that it uses ccell to encode information about the entire history of the play, but it also uses synchronization in a fundamental way *and* incorporates the factorization of non-determinism via an "oracle" given in [7]. The factorized strategy runs two threads in parallel, a "master strategy" which operates by sending and receiving messages in a context of channels; it observes Opponent moves and dictates Player moves in A via communication with a (generalized) "slave strategy" which moves back and forth between the context and A, reporting on the progress of play in A to the master strategy and executing its commands.

For a finitary strategy σ let $\mathsf{responses}(\sigma)$ be the greatest number of different P-moves which σ can give in response to one position — i.e. $\mathsf{responses}(\sigma) = \max\{|\{sa \ : \ sa \in T_\sigma \wedge \lambda^{OP}(a) = P\}| \ : \ s \in T_\sigma\}$.

Proposition 5.4 *For each arena A and $k \in \omega$ there exists a finitary, sequential and innocent (deterministic) strategy $\mathsf{slave}_k : \mathtt{nat} \times \mathtt{nat} \times \mathtt{chan}^{k+2} \to A$ such that if $\sigma : A$ is a finitary strategy bounded by k and $\mathsf{responses}(\sigma) = n$, then there is an innocent strategy $\mathsf{mas}(\sigma) : (\mathtt{nat} \times \mathtt{nat}) \times \mathtt{chan}^{k+2} \times \mathtt{nat} \to A$ and such that $\mathtt{oracle}_n \times [\![0||1]\!] \times \mathsf{ccell}^{k+2}; (\mathsf{mas}(\sigma)|\mathsf{slave}_k) = \sigma$.*

Proof. Factorized plays proceed as follows. The slave responds to each Opponent move a in A by trying to send an encoding of it (and its view) on channel 1. The master maintains a single open receive question on channel 1, allowing it to learn the entire history of play in A (up to $\ll$ and $\sim_O$). Once the master and slave have concluded a successful communication on channel 1, the master sends the slave a number $2 < i \leq k + 2$ on channel 2. This number is the name of a private channel, in which the slave immediately plays a receive move, waiting for communication from the master. When the master has observed that play in A has taken place to which σ would respond with one of the P-moves $b_0, b_2, \ldots b_m$, he splits the thread of control using the parallel composition $[\![0||1]\!]$, in a special case of the sequentialization factorization of Proposition 5.3. In one of the threads thus created, he maintains the surveillance of channel 1 by repeating the receive question. In the other, he uses the oracle to generate $j \leq m$ non-deterministically. If the move b_j has a pointer to a_i the master then sends an encoding of b_j on channel i which is received by the waiting slave, who plays the move b_j with a pointer to a_i as

required. □

The factorization theorems together with Proposition 4.4 yield the following definability result.

Corollary 5.5 *Every finitary (pointer-blind) strategy σ over a channel-free type-object $[\![T]\!]$ is definable as an ICSP term $M_\sigma : T$.* □

The definability result means that the "intrinsic equivalence" on strategies corresponds to observational equivalence. Thus we can now define a fully abstract model of ICSP by characterizing this intrinsic equivalence directly, via a "dual" to the "equivalence up to redirection of pointers", used to define the pointer-blindness condition (Definition 3.21).

Definition 5.6 Let $\sim_P$ be the least equivalence relation on multi-threaded sequences such that $sa_1t_1a_2t_2br \sim_P sa_1t_1a_2t_2br$ if b is a Player move with a concurrency pointer to a_1 in the first sequence and to a_2 in the second, and the sequences are otherwise identical.

Let $\sigma \sim \tau$ if $\forall s \in \sigma.\exists t \in \tau.s \sim t$ and $\forall t \in \tau.\exists s \in \sigma.s \ll \tau$.

Theorem 5.7 *For all closed terms $M, N : T$ ICSP, $M \simeq^{M\&M} N$ if and only if $[\![M]\!] \sim [\![N]\!]$.*

Proof. It is straightforward that if $[\![M]\!] \sim [\![N]\!]$ then no pointer-blind strategy can distinguish them To establish the converse, suppose $[\![M]\!]$ contains a sequence s such that for all $t \in [\![N]\!]$, $t \not\sim_P s$. Assume that s is maximal (with this property) with respect to $\ll$.

There are two cases to consider, depending on whether s is *reachable* (up to $\sim_P$) in t, or not Suppose s is not reachable — i.e. for all $t \in T_{[\![N]\!]}$, $t \not\sim_P s$. Then let $\rho : [\![T]\!] \to [\![\mathtt{comm}]\!]$ be the strategy generated by $qsa \in [\![T]\!] \Rightarrow [\![\mathtt{comm}]\!]$ (where q is the initial question in $[\![\mathtt{comm}]\!]$ and a is its answer) — i.e. ρ is generated by $\{t \in LM_{[\![T \Rightarrow \mathtt{comm}]\!]} \mid \exists r \sqsubseteq qsa.t \ll r\}$. By definability, there exists a term $L_\rho : T \Rightarrow \mathtt{comm}$ such that $\rho = [\![L_\rho]\!]$. Then $qa \in [\![M]\!]; \rho$, so by soundness, $L_\rho\ M \Downarrow^{may}$. And $qa \notin [\![N]\!]; \rho$, since if $qs'a \ll qsa$, then $s \ll s'$ and hence $s' \notin [\![N]\!]$ by maximality of s with respect to $\ll$. Hence by soundness $L_\sigma\ N \not\Downarrow^{may}$.

Suppose s is reachable in $[\![N]\!]$ — i.e. there is a sequence $sr \in [\![N]\!]$. Then let ρ be the strategy which converges when it encounters any trace from $[\![M]\!]$ which is distinct from s, and diverges otherwise — i.e. ρ is generated by the set of sequences $\{t \in LM_{[\![T \Rightarrow \mathtt{comm}]\!]} \mid \exists rb \in [\![M]\!].r \sqsubseteq s \wedge t \ll qra\}$ (which is finite because $[\![M]\!]$ is finite branching). Then $q \notin [\![M]\!]$, but $q \in [\![N]\!]$, so by the argument from definability, $L_\rho\ M \Downarrow^{must}$ but $L_\rho\ N \not\Downarrow^{must}$. □

6 Conclusions

The semantics of ICSP described here can be considered a first step in an attempt to describe concurrency in functional languages more generally using

game semantics. The most obvious next step is a characterisation of recursion in such a setting. To do so is straightforward for a semantics of *may*-testing; but whilst there is a natural model of must-testing, the difficulty is to define an operational semantics with respect to which it is adequate, as it is necessary that evaluation is *fair*. Another desirable development would be an extension of the full abstraction result to types including channels. Recent work by McCusker on the game semantics of Idealized Algol without bad variables suggests that this is possible using an ordering on legal plays, although in the concurrent case it seems rather more complicated.

Variations on the language itself can be considered. For instance, asynchronous message passing (without queuing) has a natural interpretation, the interesting problem is to give a definability result for this model. Or we could consider a call-by-value language with thread-identifiers and passing of functions as messages (a kind of "core CML" [16]). It is straightforward to give a call-by-value semantics in the style described here (either directly, or by using the $\mathsf{Fam}(\mathcal{C})$ construction [2]). Message-passing can be extended from ground types to higher types much as the games interpretation of Idealized Algol extends to general references [4]. Comparison with the various syntactic encodings of these features may be of interest.

As we have described a framework for modelling concurrency using HO-games, as well as the specific interpretation of ICSP, there is plenty of scope for more general developments — the language and semantics can be extended with features such as control (as in [12,13]) or higher-order references (as in [4]), as has been done in the sequential case. Here again there are some expressiveness issues which are well known such as encodings of references using channels, but in general a formal and fully abstract semantics could provide a powerful tool for reasoning precisely about the interaction of sequential and concurrent effects.

References

[1] S. Abramsky and G. McCusker. Linearity Sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In P.W. O'Hearn and R. Tennent, editors, *Algol-like languages*. Birkhauser, 1997.

[2] S. Abramsky and G. McCusker. Call-by-value games. In M. Neilsen and W. Thomas, editors, *Computer Science Logic:* 11^{th} *Annual workshop proceedings*, LNCS, pages 1–17. Springer-Verlag, 1998.

[3] S. Abramsky and P.-A. Mellies. Concurrent games and full completeness. In *Proceedings of the 14th annual Symposium on Logic In Computer Science, LICS '99*, 1999.

[4] S. Abramsky, K. Honda, G. McCusker. A fully abstract games semantics for general references. In *Proceedings of the 13th Annual Symposium on Logic In Computer Science, LICS '98*, 1998.

[5] S. Abramsky, R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59:543–574, 1994.

[6] S. Brookes. Idealized CSP: Combining procedures with communicating processes. In *Proceedings of MFPS '97*, Electronic notes in Theoretical Computer Science. Elsevier-North Holland, 1997.

[7] R. Harmer and G. McCusker. A fully abstract games semantics for finite nondeterminism. In *Proceedings of the Fourteenth Annual Symposium on Logic in Computer Science, LICS '99*. IEEE press, 1998.

[8] M. Hennessy, and E. Ashcroft. A mathematical semantics for a nondeterministic typed λ-calculus. *Theoretical Computer Science*, 11:227–245, 1980.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[10] K. Honda and N. Yoshida. Game theoretic analysis of call-by-value computation. In *Proceedings of 24th International Colloquium on Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[11] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III, 1995. To appear in Theoretical Computer Science.

[12] J. Laird. Full abstraction for functional languages with control. In *Proceedings of the Twelfth International Symposium on Logic In Computer Science, LICS '97*, 1997.

[13] J. Laird. A fully abstract game semantics of local exceptions. In *Proceedings of the Sixteenth International Symposium on Logic In Computer Science, LICS '01*, 2001. To appear.

[14] G. McCusker. *Games and full abstraction for a functional metalanguage with recursive types*. PhD thesis, Imperial College London, 1996.

[15] H. Nickau. Hereditarily sequential functionals. In *Proceedings of the Symposium on Logical Foundations of Computer Science:Logic at St. Petersburg*, LNCS. Springer-Verlag, 1994.

[16] J. Reppy. *Higher Order Concurrency*. PhD thesis, Cornell University, 1992.

[17] J. Reynolds. Syntactic control of interference. In *Conf. Record 5^{th} ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[18] J. Reynolds. The essence of Algol. In *Algorithmic Languages*, pages 345–372. North Holland, 1981.

MFPS 17 Preliminary Version

Unique Fixed Points in Domain Theory

Keye Martin

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD
http://web.comlab.ox.ac.uk/oucl/work/keye.martin

Abstract

We unveil new results based on measurement that guarantee the existence of unique fixed points which *need not* be maximal. In addition, we establish that least fixed points are *always* attractors in the μ topology, and then explore the consequences of these findings in analysis. In particular, an extension of the Banach fixed point theorem on compact metric spaces is obtained.

1 Introduction

The standard fixed point theorem in domain theory states that a Scott continuous map $f : D \to D$ on a dcpo D with least element $\perp$ has a least fixed point given by

$$\mathrm{fix}(f) := \bigsqcup_{n \geq 0} f^n(\perp).$$

This is perhaps the single most important result in domain theory, given its effectiveness in handling the semantics of recursion, and the fact that its reasoning extends naturally to the categorical level to explain why it is that equations like $D \simeq [D \to D]$ may be solved.

It could be argued that one of its faults is that it only applies to *continuous* mappings, since there are now more general fixed point theorems available [4]. However, within the context of continuous mappings, the only criticism that seems plausible is that its canonical fixed points are not as canonical as they could be. There is, after all, one thing more satisfying than a least fixed point: A unique fixed point.

Using ideas all originally introduced in [5], we establish that there are natural fixed point theorems in domain theory which guarantee the existence of unique, attractive fixed points. In the next three sections, we discuss domains, content and invariance. These are preliminary ideas needed later on. We then introduce contractions on domains and prove a fixed point theorem about them very reminiscent of the Banach theorem in analysis. In fact, this new result has the Banach theorem as one of its consequences.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Finally, in the case of a domain with a least element, we learn that least fixed points are *always* attractive in the μ topology and that the results on contractions also hold for a larger and more natural class of *nonexpansive* mappings. Because of the latter, an improvement of the Banach fixed point theorem on compact metric spaces can be obtained.

2 Background

A *poset* is a partially ordered set [1].

Definition 2.1 Let $(P, \sqsubseteq)$ be a partially ordered set. The *least element* $\bot$ of P satisfies $\bot \sqsubseteq x$ for all x, when it exists. A nonempty subset $S \subseteq P$ is *directed* if $(\forall x, y \in S)(\exists z \in S)\, x, y \sqsubseteq z$. The *supremum* of a subset $S \subseteq P$ is the least of all its upper bounds provided it exists. This is written $\bigsqcup S$. A *dcpo* is a poset in which every directed subset has a supremum.

Definition 2.2 For a subset X of a dcpo D, set

$$\uparrow X := \{y \in D : (\exists x \in X)\, x \sqsubseteq y\} \;\; \& \;\; \downarrow X := \{y \in D : (\exists x \in X)\, y \sqsubseteq x\}.$$

We write $\uparrow x = \uparrow\{x\}$ and $\downarrow x = \downarrow\{x\}$ for elements $x \in X$. The set of *maximal elements* in a dcpo D is $\max D = \{x \in D : \uparrow x = \{x\}\}$.

By Hausdorff maximality, every dcpo has at least one maximal element.

Definition 2.3 A subset U of a dcpo D is *Scott open* if

(i) U is an upper set: $x \in U \;\&\; x \sqsubseteq y \Rightarrow y \in U$, and

(ii) U is inaccessible by directed suprema: For every directed $S \subseteq D$,

$$\bigsqcup S \in U \Rightarrow S \cap U \neq \emptyset.$$

The collection σ_D of all Scott open sets on D is called the Scott topology.

Proposition 2.4 *A map $f : D \to E$ between dcpo's is Scott continuous iff*

(i) *f is monotone: $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.*

(ii) *f preserves directed suprema: For every directed $S \subseteq D$,*

$$f(\bigsqcup S) = \bigsqcup f(S).$$

Definition 2.5 In a dcpo $(D, \sqsubseteq)$, $a \ll x$ iff for all directed subsets $S \subseteq D$, $x \sqsubseteq \bigsqcup S \Rightarrow (\exists s \in S)\, a \sqsubseteq s$. We set $\Downarrow x = \{a \in D : a \ll x\}$. A dcpo D is *continuous* if $\Downarrow x$ is directed with supremum x for all $x \in D$.

The sets $\Uparrow x = \{y \in D : x \ll y\}$ for $x \in D$ form a basis for the Scott topology on a continuous dcpo D. Finally, we adopt the following definition of 'domain' in this paper.

Definition 2.6 A *domain* is a continuous dcpo D such that for all $x, y \in D$, there is $z \in D$ with $z \sqsubseteq x, y$.

For example, a continuous dcpo with *least element* $\perp$ is a domain in the present sense.

3 Content

The ideas in this and the next section are covered in embarassing detail in [5]. Let $[0,\infty)^*$ be the domain of nonnegative reals ordered as $x \sqsubseteq y \Leftrightarrow y \leq x$.

Definition 3.1 A Scott continuous map $\mu : D \to [0,\infty)^*$ on a continuous dcpo D *induces the Scott topology near* $X \subseteq D$ if for all $x \in X$ and all sequences $x_n \ll x$,

$$\lim_{n\to\infty} \mu x_n = \mu x \Rightarrow \bigsqcup x_n = x,$$

and this supremum is directed. We write this as $\mu \to \sigma_X$.

That is, if we *observe* that a sequence (x_n) of approximations calculates x, it *actually does* calculate x. The map μ measures the information content of the objects in X. For this reason, we sometimes say that μ *measures* X.

Definition 3.2 A *measurement* is a Scott continuous map $\mu : D \to [0,\infty)^*$ that measures the set $\ker \mu = \{x \in D : \mu x = 0\}$.

The nature of the idea is that information imparted to us by a measurement about the environment may be taken as true. Here is an illustration of this principle.

Proposition 3.3 *Let $\mu : D \to [0,\infty)$ be a measurement with $\mu \to \sigma_D$. Then*

(i) *For all $x \in D$, $\mu x = 0 \Rightarrow x \in \max D$.*

(ii) *For all $x, y \in D, x \sqsubseteq y \;\&\; \mu x = \mu y \Rightarrow x = y$.*

(iii) *A monotone map $f : D \to D$ is Scott continuous iff $\mu f : D \to [0,\infty)^*$ is Scott continuous.*

One of the first motivations for measurement was the desire to prove useful fixed point theorems. Generally speaking, 'useful' means theorems which are easy to apply to nonmonotonic mappings, or results which say more about monotonic maps than the Scott fixed point theorem. Here is the first example ever found of the latter type [5].

Theorem 3.4 *Let $f : D \to D$ be a monotone map on a domain D with a measurement μ for which there is a constant $c < 1$ such that*

$$(\forall x)\, \mu f(x) \leq c \cdot \mu x.$$

If there is a point $x \in D$ with $x \sqsubseteq f(x)$, then

$$x^\star = \bigsqcup_{n\geq 0} f^n(x) \in \max D$$

is the unique fixed point of f on D. Furthermore, $x^\star$ is an attractor in two different senses:

(i) *For all $x \in \ker \mu$, $f^n(x) \to x^\star$ in the Scott topology on $\ker \mu$, and*

(ii) *For all $x \sqsubseteq x^\star$, $\bigsqcup_{n\geq 0} f^n(x) = x^\star$, and this supremum is a limit in the Scott topology on D.*

The only problem with this theorem is that it requires fixed points maximal. Very shortly we will uncover some new results that overcome this difficulty in what appears to be a more elegant approach.

4 Invariance

All ways of measuring a domain appeal to a common objective.

Definition 4.1 The *μ topology* on a continuous dcpo D has as a basis all sets of the form $\Uparrow x \cap \downarrow y$, for $x, y \in D$. It is denoted μ_D.

One unsatisfying aspect of the Scott topology is its weak notion of limit. Ideally, one would hope that any sequence with a limit in the Scott topology had a supremum. But this is far from true. For instance, on a continuous dcpo with least element $\bot$, all sequences converge to $\bot$.

Lemma 4.2 (Martin [5]) *Let D be a continuous dcpo. Then*

(i) *A sequence (x_n) converges to a point x in the μ topology iff it converges to x in the Scott topology and $(\exists n)\, x_k \sqsubseteq x$, for all $k \geq n$.*

(ii) *If $x_n \to x$ in the μ topology, then there is a least integer n such that*

$$\bigsqcup_{k\geq n} x_k = x.$$

(iii) *If (x_n) is a sequence with $x_n \sqsubseteq x$, then $x_n \to x$ in the μ topology iff $x_n \to x$ in the Scott topology.*

In a phrase, μ limits are the Scott limits with computational significance.

Proposition 4.3 *A monotone map $f : D \to E$ between continuous dcpo's is μ continuous iff it is Scott continuous.*

So what does all this have to do with information content? Given a measurement $\mu \to \sigma_D$, consider the elements ε-close to $x \in D$, for $\varepsilon > 0$, given by

$$\mu_\varepsilon(x) := \{y \in D : y \sqsubseteq x \;\&\; |\mu x - \mu y| < \varepsilon\}.$$

Regardless of the measurement we use, these sets are *always* a basis for the μ topology. In fact, it is this property which *defines* content on a domain.

Theorem 4.4 (Martin [5]) *For a Scott continuous map $\mu : D \to [0, \infty)^*$, $\mu \to \sigma_D$ iff $\{\mu_\varepsilon(x) : x \in D \;\&\; \varepsilon > 0\}$ is a basis for the μ topology on D.*

This realization not only improves our understanding of the μ topology, it also allows us to make more effective use of measurement.

Lemma 4.5 *Let (D, μ) be a continuous dcpo with a measurement $\mu \to \sigma_D$.*

(i) *If (x_n) is a sequence with $x_n \sqsubseteq x$, then*

$$x_n \to x \text{ in the } \mu \text{ topology iff } \lim_{n\to\infty} \mu x_n = \mu x.$$

(ii) *A monotone map $f : D \to D$ is Scott continuous iff $\mu f : D \to [0, \infty)^*$ is μ continuous.*

Notice that the Scott topology can always be recovered from the μ topology as $\sigma_D = \{\uparrow U : U \in \mu_D\}$.

5 Fixed points of contractions

In this section, (D, μ) is a *domain* with a measurement $\mu \to \sigma_D$.

Definition 5.1 Let f be a monotone selfmap on (D, μ). If there exists a constant c such that

$$x \sqsubseteq y \Rightarrow \mu f(x) - \mu f(y) \leq c \cdot (\mu x - \mu y),$$

for all $x, y \in D$, then f is a *contraction* if $c < 1$ and *nonexpansive* if $c \leq 1$.

Proposition 5.2 *A contraction is Scott continuous.*

Proof. First, μf is μ continuous. By Theorem 4.4, the μ topology on D is first countable, and so we can work with sequences in verifying this assertion.

Let $x_n \to x$ in the μ topology on D. Then we can assume $x_n \sqsubseteq x$. Hence

$$0 \leq \mu f(x_n) - \mu f(x) \leq c \cdot (\mu x_n - \mu x)$$

which means

$$\lim_{n\to\infty} \mu f(x_n) = \mu f(x)$$

since $\mu x_n \to \mu x$. Then μf is μ continuous. By Lemma 4.5(ii), f is Scott continuous. □

The last proposition does not require a contraction: The same proof works for any value of $c \geq 0$. It is our next result that requires $c < 1$.

Theorem 5.3 *Let f be a contraction on (D, μ). If there is a point $x \sqsubseteq f(x)$, then*

$$\mathrm{fix}(f) := \bigsqcup_{n\geq 0} f^n(x)$$

is the unique fixed point of f on D.

Proof. By Prop. 5.2, f is Scott continuous, so it is clear that $\mathrm{fix}(f)$ is a fixed point of f.

Let $x = f(x)$ and $y = f(y)$ be two fixed points of f. By our assumption on D, there is an element $z \in D$ with $z \sqsubseteq x, y$. Then

$$f^n(z) \sqsubseteq x = f^n(x),$$

for all $n \geq 1$. By induction, we have

$$\mu f^n(z) - \mu f^n(x) = \mu f^n(z) - \mu x \leq c^n \cdot (\mu z - \mu x),$$

for all $n \geq 1$. Then $\mu f^n(z) \to \mu x$ and so

$$\bigsqcup_{n \geq 0} f^n(z) = x$$

since $\mu \to \sigma_D$. But the same argument applies to y. □

In fact, careful inspection of the proof of the last theorem shows that the unique fixed point is an attractor in the μ topology.

Definition 5.4 A fixed point $p = f(p)$ of a continuous map $f : X \to X$ on a space X is called an *attractor* if there is an open set U around p such that

$$f^n(x) \to p$$

for all $x \in U$. We also refer to p as *attractive.*

Examples of attractive fixed points are easy to find: In the analysis of hyperbolic iterated function systems, where they are sometimes called fractals, or in the study of iterative methods in numerical analysis like Newton's method, where they arise as the solutions to nonlinear equations.

Corollary 5.5 *Let f be a contraction on (D, μ). If $a \sqsubseteq \mathrm{fix}(f)$, then*

$$\bigsqcup_{n \geq 0} f^n(a) = \mathrm{fix}(f),$$

and this supremum is a limit in the μ topology. That is, $\mathrm{fix}(f)$ is an attractor in the μ topology.

Proof. The claim is implicitly established in the previous theorem. For the attractor bit, let $U = {\downarrow}\mathrm{fix}(f)$. This is a lower set and hence μ open. □

Thus, beginning with *any* approximation a of $\mathrm{fix}(f)$, the iterates $f^n(a)$ converge to $\mathrm{fix}(f)$, even if $a \not\sqsubseteq f(a)$. In addition, we can obtain a good estimate of how many iterations are required to achieve an ε-approximation of $\mathrm{fix}(f)$.

Proposition 5.6 *Let f be a contraction on (D, μ) with unique fixed point $\mathrm{fix}(f)$. Then for any $x \sqsubseteq \mathrm{fix}(f)$ and $\varepsilon > 0$,*

$$n > \frac{\log(\mu x - \mu\,\mathrm{fix}(f)) - \log \varepsilon}{\log(1/c)} \;\Rightarrow\; |\mu f^n(x) - \mu\,\mathrm{fix}(f)| < \varepsilon,$$

for any integer $n \geq 0$, provided $x \neq \mathrm{fix}(f)$.

Proof. If $c = 0$, then f is constant, and the statement holds trivially, adopting the convention that $\log(1/0) = \log \infty = \infty$. Let $0 < c < 1$.

For an integer $n \geq 0$, we have

$$|\mu f^n(x) - \mu\,\mathrm{fix}(f)| = \mu f^n(x) - \mu\,\mathrm{fix}(f) \leq c^n \cdot (\mu x - \mu\,\mathrm{fix}(f)).$$

Thus,

$$n > \frac{\log(\mu x - \mu\,\mathrm{fix}(f)) - \log\varepsilon}{\log(1/c)} \Rightarrow c^n \cdot (\mu x - \mu\,\mathrm{fix}(f)) < \varepsilon,$$

which proves the claim. □

Of course, for the estimate to be useful we must know the measure of $\mathrm{fix}(f)$. One case when this is easy to calculate is if $\mu f(x) \leq c \cdot \mu x$. Then $\mu\,\mathrm{fix}(f) = 0$. Surprisingly, this condition amounts to saying that f is the extension to D of a continuous map on $\ker\mu$.

Proposition 5.7 *For a contraction f on (D, μ) with $\ker\mu = \max D$, the following are equivalent:*

(i) *The map f preserves maximal elements.*

(ii) *For all $x \in D$, $\mu f(x) \leq c \cdot \mu x$.*

In either case, $\mathrm{fix}(f) \in \max D$.

Proof. Let f have contraction constant c.

(i) $\Rightarrow$ (ii): Let $x \in D$. By the directed completeness of D, there is an element $y \in {\uparrow}x \cap \max D$. Then

$$\mu f(x) - \mu f(y) = \mu f(x) \leq c \cdot (\mu x - \mu y) = c \cdot \mu x$$

which holds since $\mu f(y) = 0$ by (i).

(ii) $\Rightarrow$ (i): Let $x \in \max D$. Then $\mu x = 0$ so $0 \leq \mu f(x) \leq c \cdot \mu x = 0$. Thus, $f(x) \in \ker\mu = \max D$. □

In fact, every contraction on a complete metric space can be represented as a contraction on a domain of the type above.

Example 5.8 Let $f : X \to X$ be a contraction on a complete metric space X with Lipschitz constant $c < 1$. The mapping $f : X \to X$ extends to a monotone map $\bar{f} : \mathbf{B}X \to \mathbf{B}X$ on the formal ball model $\mathbf{B}X$ [2] given by

$$\bar{f}(x, r) = (fx, c \cdot r),$$

which satisfies

$$\pi\bar{f}(x, r) - \pi\bar{f}(y, s) = c \cdot \pi(x, r) - c \cdot \pi(y, s) = c \cdot (\pi(x, r) - \pi(y, s)),$$

where $\pi : \mathbf{B}X \to [0, \infty)^*$, $\pi(x, r) = r$, is the standard measurement on $\mathbf{B}X$. Now choose r so that $(x, r) \sqsubseteq \bar{f}(x, r)$. By Theorem 5.3, $\bar{f}$ has a unique fixed point which implies that f does too.

Thus, Theorem 5.3 has the Banach fixed point theorem as a consequence. A constant mapping taking any value off the top is a contraction with a unique fixed point that is not maximal, and hence not of the sort mentioned in Prop. 5.7. We will see a more substantial example later on.

6 Fixed points of nonexpansive maps

We begin with a fundamental result on a well-known theme.

Theorem 6.1 *Let $f : D \to D$ be a Scott continuous map on a continuous dcpo D with least element $\bot$. Then its least fixed point is an attractor in the μ topology: For all $x \sqsubseteq \mathrm{fix}(f)$,*

$$\bigsqcup_{n\geq 0} f^n(x) = \mathrm{fix}(f),$$

and this supremum is a limit in the μ topology.

Proof. By monotonicity, we have

$$f^n(\bot) \sqsubseteq f^n(x) \sqsubseteq \mathrm{fix}(f),$$

for all $n \geq 0$. Then since $f^n(\bot) \to \mathrm{fix}(f)$ in the μ topology, $f^n(x) \to \mathrm{fix}(f)$ in the μ topology. □

Proposition 6.2 *Let f be a monotone map on (D, μ) with least element $\bot$ and measurement $\mu \to \sigma_D$ such that*

$$x \sqsubseteq y \Rightarrow \mu f(x) - \mu f(y) < \mu x - \mu y,$$

for all distinct pairs $x, y \in D$. Then

$$\mathrm{fix}(f) := \bigsqcup_{n\geq 0} f^n(\bot)$$

is the unique fixed point of f on D.

Proof. The map f is nonexpansive and hence Scott continuous by the remark following Prop. 5.2. Thus, $\mathrm{fix}(f)$ is its least fixed point.

If x is any fixed point of f, then $\mathrm{fix}(f) \sqsubseteq x$. If these two are different, then $\mu\,\mathrm{fix}(f) - \mu x = \mu f(\mathrm{fix}(f)) - \mu f(x) < \mu\,\mathrm{fix}(f) - \mu x$. Then they are the same. This proves that $\mathrm{fix}(f)$ is the only fixed point of f. □

If the map in the last result preserved $\max D = \ker \mu$, it would satisfy $\mu f(x) < \mu x$, for $\mu x > 0$. Happily, for maps like these, we can prove the last result assuming only a measurement.

Theorem 6.3 *Let (D, μ) be a continuous dcpo with measurement μ and least element $\bot$. If $f : D \to D$ is a Scott continuous map with $\mu f(x) < \mu x$ for $\mu x > 0$, then*

$$\mathrm{fix}(f) := \bigsqcup_{n\geq 0} f^n(\bot) \in \max D$$

is the unique fixed point of f on D. In addition, if $f(\ker \mu) \subseteq \ker \mu$, then

$$(\forall x \in \ker \mu)\, f^n(x) \to \mathrm{fix}(f),$$

in the relative Scott topology on $\ker \mu$.

Proof. If $\mu\,\mathrm{fix}(f) > 0$, then $\mu\,\mathrm{fix}(f) = \mu f(\mathrm{fix}(f)) < \mu\,\mathrm{fix}(f)$. Hence $\mu\,\mathrm{fix}(f) = 0$ which means $\mathrm{fix}(f) \in \ker \mu \subseteq \max D$. But if a least fixed point is maximal, it must be unique.

To see that $\mathrm{fix}(f)$ is an attractor in the relative Scott topology on $\ker \mu$, let $U \subseteq D$ be a Scott open set around $\mathrm{fix}(f)$. Then there is K such that

$$n \geq K \Rightarrow f^n(\bot) \in U$$

which means

$$n \geq K \Rightarrow f^n(x) \in U \cap \ker \mu$$

since $f^n(\bot) \sqsubseteq f^n(x)$ and $f(\ker \mu) \subseteq \ker \mu$. Hence, $f^n(x) \to \mathrm{fix}(f)$ in the relative Scott topology on $\ker \mu$, for any initial guess $x \in \ker \mu$. □

This is the same result as Theorem 3.4 extended to a larger class of mappings on domains with least elements. In addition, in each of the last two results, Theorem 6.1 implies $\mathrm{fix}(f)$ is an attractor in the μ topology on D. Now for why all this matters.

7 Applications

Time to be sixteen again. Let $f : [0, \pi/2] \to [0, \pi/2]$ be $f(x) = \sin x$. As is well-known, beginning with any point $x \in [0, \pi/2]$ and successively applying f yields a sequence of iterates $(f^n(x))$ that magically tends to zero. Why?

At first glance, one thinks of the Banach theorem, which explains that contractions behave this way. Upon closer inspection, however, we see that things are more interesting in the case of the sine wave. Because $f'(0) = 1$, $f(x) = \sin x$ is not a contraction on $[0, \pi/2]$, and so the Banach theorem is not applicable. But domain theory is.

Example 7.1 Let $D = [0, \pi/2]^*$ be the domain with

$$x \sqsubseteq y \Leftrightarrow y \leq x$$

and natural measurement $\mu x = x$.

The function $f(x) = \sin x$ is a monotone selfmap on D. By the mean value theorem, if $x \sqsubseteq y$ and $x \neq y$, there is $c \in (y, x)$ such that

$$\mu f(x) - \mu f(y) = f(x) - f(y) = f'(c)(x - y) = (\cos c)(\mu x - \mu y).$$

Hence, $\mu f(x) - \mu f(y) < \mu x - \mu y$, since $0 < \cos c < 1$.

Then Theorem 6.2 implies that f has a unique fixed point, given by

$$\mathrm{fix}(f) = \bigsqcup_{n \geq 0} f^n(\pi/2).$$

However, $f(0) = 0$, so we must have $\mathrm{fix}(f) = 0$, by uniqueness.

Now the interesting part. By Theorem 6.1, $\mathrm{fix}(f)$ is an attractor in the μ topology. Thus, for *any* $x \sqsubseteq \mathrm{fix}(f)$, $f^n(x) \to \mathrm{fix}(f)$ in the μ topology. But convergence in the μ topology on D implies convergence in the euclidean topology. Thus, for all $x \in [0, \pi/2]$, $f^n(x) \to 0$.

In fact, the reasoning in the last example extends to *any* compact metric space, since they can all be modeled [3] as the kernel of a measurement.

Proposition 7.2 *Let $f : X \to X$ be a function on a compact metric space (X, d) such that*

$$d(fx, fy) < d(x, y)$$

for all $x, y \in X$ with $x \neq y$. Then f has a unique fixed point x^* such that for all $x \in X$, $f^n(x) \to x^*$.

Proof. Let $\mathbf{U}X$ be the domain of nonempty compact subsets of X ordered under reverse inclusion. The map f has a Scott continuous extension to $\mathbf{U}X$ given by $\bar{f} : \mathbf{U}X \to \mathbf{U}X :: K \mapsto f(K)$.

The domain $\mathbf{U}X$ has a natural measurement, $\mu x = \operatorname{diam} x$, the diameter mapping derived from the metric d. In addition, the space X can be recovered as $\ker \mu = \{\{x\} : x \in X\} = \max \mathbf{U}X \simeq X$ in the relative Scott topology.

For $\mu x > 0$, either $\mu \bar{f}(x) = 0 < \mu x$, or the compactness of x yields distinct points $a, b \in x$ such that

$$\mu \bar{f}(x) = d(fa, fb) < d(a, b) \leq \mu x.$$

The result now follows from Theorem 6.3. □

That is, the Banach fixed point theorem holds on compact metric spaces under weaker assumptions. The impressive aspect of the last result is *not* the uniqueness of x^*: It is that x^* is a global attractor.

Corollary 7.3 *Let $f : [a, b] \to [a, b]$ be a continuous map on a nonempty compact interval. If $|f'(x)| < 1$ for all $x \in (a, b)$, then f has a unique fixed point $x^* \in [a, b]$ such that $f^n(x) \to x^*$, for all $x \in [a, b]$.*

Proof. By the mean value theorem,

$$|f(b) - f(a)| = |f'(c)||b - a| < |b - a|$$

for some $c \in (a, b)$. Now Prop. 7.2 applies. □

The map $f(x) = \sin x$ satisifes $0 < f'(x) < 1$ on $(0, \pi/2)$, but as we have already seen, it is not a contraction. Thus, Theorem 6.3 yields another insight into why the sine wave behaves the way it does. But the reader should not be misled into thinking that these ideas are only applicable to domain theoretic fragments of classical mathematics.

Example 7.4 Let $D = [\mathbb{N} \to \mathbb{N}_\perp]$ be the domain of partial functions on the naturals. The elements of D are measured as

$$\mu f = \sum_{f(n)=\perp} \frac{1}{2^{n+1}}.$$

Then $\ker \mu = \{f \in D : \operatorname{dom}(f) = \mathbb{N}\}$ is the set of total functions. Now consider the operator $\phi : D \to D$ given by

$$\phi(f)(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1; \\ n + f(n-2)/\cos(n\pi/2) & \text{otherwise.} \end{cases}$$

Because $\phi(f)(n) = \perp \Rightarrow f(n-2) = \perp$ or ($n > 1$ and odd), we are able to write for elements $f \sqsubseteq g$ that

$$\begin{aligned}\mu\phi(f) - \mu\phi(g) &= \sum_{f(n-2)=\perp} \frac{1}{2^{n+1}} - \sum_{g(n-2)=\perp} \frac{1}{2^{n+1}} \\ &= \sum_{f(n)=\perp} \frac{1}{2^{n+3}} - \sum_{g(n)=\perp} \frac{1}{2^{n+3}} \\ &= \frac{1}{4}(\mu f - \mu g).\end{aligned}$$

By Theorem 5.3, ϕ has a unique fixed point. Clearly this fixed point is not maximal: It is undefined on the set $\{2k+1 : k > 1\}$.

Then contractions with fixed points off the top are worth studying too.

8 Ideas

It would be nice to see a metric based approach to semantics replaced with one based on results like Theorem 5.3. Especially on a model of CSP. Trying to obtain estimates in the spirit of Prop. 5.6 for *nonexpansive* maps also seems like a fun question. It would be neat to find out if the informatic derivative [5] (derivative of a map on a domain with respect to a measurement) can be useful in this regard.

References

[1] S. Abramsky and A. Jung. *Domain Theory.* In S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, editors, Handbook of Logic in Computer Science, vol. III. Oxford University Press, 1994.

[2] A. Edalat and R. Heckmann. *A Computational Model for Metric Spaces.* Theoretical Computer Science 193 (1998) 53–73.

[3] K. Martin. *Nonclassical techniques for models of computation.* Topology Proceedings, vol. 24, 1999.

[4] K. Martin. *The measurement process in domain theory.* Proceedings of the 27^{th} International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, vol. 1853, Springer-Verlag, 2000.

[5] K. Martin. *A foundation for computation.* Ph.D. Thesis, Tulane University, Department of Mathematics, 2000.

MFPS 17 Preliminary Version

A Generalisation of Stationary Distributions, and Probabilistic Program Algebra

A.K. McIver [1,2]

Department of Computing
Macquarie University
NSW, Australia

Abstract

We generalise the classical notion of stationary distributions of Markov processes to a model of probabilistic programs which includes demonic nondeterminism. As well as removing some of the conditions normally required for stationarity, our generalisation allows the development of a complete theory linking stationary behaviour to long-term average behaviour — the latter being an important property that lies outside the expressive range of standard logics for probabilistic programs.

Keywords: Probabilistic program semantics, probability, demonic nondeterminism, Markov process, stationary distribution, Markov decision processes.

1 Introduction

Programs or processes which can make probabilistic choices during their execution exhibit a range of (probabilistic) behaviours outside those describable by purely *qualitative* formalisms; moreover even well-known *quantitative* adaptations of familiar program logics — the foremost being probabilistic temporal logic [18,2] — are still not expressive enough in some cases. One such is the so-called "average long-term" behaviour [3,4], which we illustrate in the context of the program presented in Fig. 1. The program *FP* represents a specification of a simple failure-repair mechanism. The system it describes is intended to execute repeatedly, and the state evolves according to the specified probabilistic statements. The average long-term behaviour of *FP* determines (for example) the proportion of time that the state is *ok*, and is always well-defined [4]. Other related terms are "availability" [17] and "the stationary probability of *ok*" [8]. In this particular case an elementary analysis reveals that *ok* holds

[1] This work was done at Oxford University, UK, and was funded by the EPSRC.
[2] Email: anabel@ics.mq.edu.au

$$
\begin{array}{l}
FP := \textbf{if } ok \\
\qquad \textbf{then } (ok \,{}_{2/3}\!\oplus \neg ok) \;[]\; ok \\
\qquad \textbf{else } (ok \,{}_{1/2}\!\oplus \neg ok) \;[]\; ok \\
\textbf{fi}
\end{array}
$$

ok and $\neg ok$ toggle the states corresponding to working and broken behaviour. The operator ${}_{2/3}\!\oplus$ records a probabilistic update, whereas [] records a nondeterministic update. Used together like this, we are able to specify tolerances on failure rates — at every execution, there is *at least* a probability $1/2$ of reestablishing ok (since the only other alternative to the probabilistic branch establishes ok with certainty).

Fig. 1. An abstract failure-repair mechanism

on average at least $3/5$ of the time — yet probabilistic temporal logic cannot describe that behaviour. (de Alfaro gives a nice discussion of the issues [3].)

In elementary probability theory, long term average behaviour is, in some special cases, determined by "stationary distributions" — a property of (some) Markov processes. Though some authors [10,16] have used Markov processes as a model for probabilistic programs, more recently a generalised form [13,9,2] has been found to be more suitable, since it supports the notions of (demonic) nondeterminism (or abstraction) and the induced partial order known as refinement. That is the model we shall work with here, and we give details in Sec. 5.

Thus our main contribution (in Sec. 3) is to give an axiomatic account of stationary behaviour and convergence to it, one which extends and simplifies the classical notion. Not only is our notion of generalised convergence applicable to all Markov processes (rather than only to some special cases) but it completes the theory linking stationary behaviour to average long-term behaviour. The details are set out in Sec. 5.

We develop our theory following the algebraic style already available in theories of concurrency, where it has proved a powerful tool for analysing nondeterministic programs that execute repeatedly.

We use "." for function application; $\leq$, $+$ and $\sqcap$ denote respectively "is no more than", addition and minimum applied pointwise to real-valued functions. Throughout S is a finite state space and $\overline{S}$ is $\{F \mid S \to [0,1] \cdot \sum_{s:S} F.s = 1\}$, the set of (discrete) probability distributions over S. For real k, we write $\underline{k}$ for the constant real-valued function with range $\{k\}$. If α is a real-valued function over S then $(\sqcup \alpha)$ and $(\sqcap \alpha)$ denote respectively the maximum and minimum value taken by α as the state varies over S; and $(k\alpha)$ or $k(\alpha)$ represents the the function α pointwise multiplied by the real k. We introduce other notation as we need it.

2 Probabilistic sequential programs

We summarise two equivalent models for probabilistic programs; more details are given elsewhere [13,9]. The semantics for probabilistic sequential programs supports the interpretation of traditional programming control structures together with a binary probabilistic choice operator ${}_p\oplus$, where the operational meaning of the expression $A\ {}_p\oplus\ B$ is that either A or B is executed, with probability respectively p or $1-p$. Since there is no determined output, that behaviour is sometimes called "probabilistic nondeterminism". Probabilistic nondeterminism is however very different from "demonic nondeterminism", denoted by "[]", already present in standard guarded commands [5], and which can model underspecification or demonic scheduling in distributed systems.

And the two operators are modelled very differently — as usual probabilistic information is described by (output) probability distributions over final states, whereas demonic behaviour is described by subsets of possible outputs. Putting those two ideas together leads to a model in which programs correspond to functions from initial state to sets of distributions over final states, where the multiplicity of the result set represents a degree of nondeterminism and the distribution records the probabilistic information after that nondeterminism has been resolved. We have the following definition for the probabilistic program space $\mathcal{H}S$ [9,13] for programs operating over the abstract state space S,[3] and its treatment of nondeterminism is similar to that of other models [2,15,4]:

$$\mathcal{H}S := S \to \mathbb{P}\overline{S}\ .$$

More generally, like Markov processes, every program in $\mathcal{H}S$ can be considered to be a function from probability distributions over initial states, but in this case to *sets* of probability distributions over final states [9].

We order programs using program refinement, which compares the extent of nondeterminism — programs higher up the refinement order exhibit less nondeterminism than those lower down:

$$Q \sqsubseteq P \ \text{ iff } \ (\forall s: S \bullet P.s \subseteq Q.s)\ .$$

Classical Markov processes can be identified with the subclass of "deterministic", or purely probabilistic programs in $\mathcal{H}S$, and as such are maximal with respect to $\sqsubseteq$. For instance the (demonically deterministic) program $ok_{1/2}\oplus\neg ok$ has no proper refinements at all.

One consequence of $\sqsubseteq$ above is that (worst case) quantitative properties improve as programs become more refined. If Q guarantees to establish a predicate ϕ with probability at least p (irrespective of the nondeterminism), then P must also establish ϕ with probability at least that same p.

That observational view of probabilistic systems (in which the frequency of outputs is recorded) is captured more generally with the idea of "expected values". Kozen was the first to exploit this fact in his probabilistic program logic

[3] This basic model can also be enhanced to include nontermination [13] and miracles [14].

(but for deterministic programs). His insight was to regard programs as operators which transform real-valued functions in a goal-directed fashion, in the same way that standard programs can be modelled as predicate transformers [5]. The use of real-valued functions instead of predicates allows expressions to incorporate quantitative (as well as qualitative) information. The idea has been extended by others [13] to include demonic nondeterminism as well as probability. We write $\mathcal{E}S$ for the space of real-valued functions (expectations) over S, and $\mathcal{T}S$ for the associated space of "expectation transformers", defined next.

Definition 2.1 Let $r: S \to \mathbb{P}(\overline{S})$ be a program taking initial states in S to sets of final distributions over S. Then the *greatest guaranteed* pre-expectation at state s of program r, with respect to post-expectation α in $\mathcal{E}S$, is defined

$$wp.r.\alpha.s \quad := \quad (\sqcap F: r.s \bullet \int_F \alpha) \ ,$$

where $\int_F \alpha$ denotes the expected value of α with respect to distribution F.[4] We say that $wp.r$ is an *expectation transformer* corresponding to r, and we define $\mathcal{T}S$ to be $wp.\mathcal{H}S$.

Programs are ordered by comparing the results of qualitative observations: thus

$$t \sqsubseteq t' \quad \text{iff} \quad (\forall \alpha : \mathcal{E}^+S \bullet t.\alpha \leq t'.\alpha) \ ,$$

where $\mathcal{E}^+S$ denote the non-negative expectations. There is no conflict in using "$\sqsubseteq$" to denote the order in both $\mathcal{H}S$ and $\mathcal{T}S$, since the definitions correspond [13].

In the special case that the post-expectation takes values in $\{0,1\}$ and thus represents a predicate, the pre-expectation represents the greatest guaranteed probability of the program establishing that predicate. Nondeterminism, as for predicate transformers, is interpreted demonically.

Although the two views are equivalent [13], we usually use $\mathcal{T}S$ because its arithmetic properties make it more convenient for proof than $\mathcal{H}S$. Transformers in $\mathcal{T}S$ are continuous (in the sense of real-valued functions) and subadditive, that is

$$t.(k\alpha + k'\beta - \underline{k''}) \quad \geq \quad k(t.\alpha) + k'(t.\beta) - \underline{k''} \ ,$$

which can be strengthened to additivity in the case of deterministic programs (classical Markov processes). We interpret basic program constructs as operations on transformers: thus $(t;t').\alpha := t.(t'.\alpha)$; $(t \,[]\, t').\alpha := t.\alpha \,[]\, t'.\alpha$ and $(t \,{}_p\!\oplus\, t').\alpha := p(t.\alpha) + (1-p)(t'.\alpha)$, from which we see that determinism is preserved by ${}_p\!\oplus$ and $;$, but not by $[]$.

The next lemma can be proved very simply using the notions of $\mathcal{T}S$. Define the norm $||.||$ on expectations as $||\alpha|| := (\sqcup\alpha) - (\sqcap\alpha)$. Our definitions imply

[4] In fact $\int_F \alpha$ is just $\sum_{s:S} \alpha.s \times F.s$ because S is finite and F is discrete [6]. We use the $\int$-notation because it is less cluttered, and to be consistent with the more general case.

that if $||\alpha|| = 0$ then α is constant on S.

Lemma 2.2 *Let t, t' be an expectation transformers in $\mathcal{T}S$. If t is deterministic, and $t'; t = t'$; and furthermore if there is some $0 \leq c < 1$ such that for any α we have $||t.\alpha|| \leq c||\alpha||$, then t' is deterministic.*

Proof*: The above discussion suggests that we just need to show that t' is additive, which follows by continuity of transformers in $\mathcal{T}S$.*

Even though Lem. 2.2 is more generally true for any programs in $\mathcal{T}S$ satisfying the conditions, it actually characterises the property which underlies whether a Markov process converges to its so-called stationary distribution or not, namely that it acts like a contraction with respect to $||.||$. The term "contraction" however is more general and can be applied to the whole of $\mathcal{T}S$, not just to its deterministic portion: FP in Fig. 1 is a contraction for instance, though it is not a Markov process.

Conversely, if t^n is not a contraction for any power of t then it can be shown that there is some proper subset of states that is left invariant by t^n, for some n. Such programs are also called "periodic", and we shall return to them later.

3 A program-algebraic treatment of 'stationary behaviour'

In this section we study some algebraic properties of programs or systems that execute repeatedly. Algebraic approaches have proved to be very powerful in the development of concurrency theory [1]; we find them to be extremely effective in this context as well.

Our basic language (in Fig. 2) consists of two binary operators (";", sequential composition and "[]", demonic nondeterministic choice), one constant (1, "do nothing") and a unary operator ("*", the "Kleene star"). Both ; and [] are associative and [] is commutative; 1 is the identity of ;. Observe that for probabilistic models [] fails to distribute to the left. (Other nonprobabilistic interpretations would allow full distributivity [1].) We interpret x^* in $\mathcal{T}S$ as the transformer $x^*.\alpha := (\nu Y \cdot \alpha \sqcap x; Y)$,[5] which corresponds to the program that from initial state s outputs the strongest set of invariant states containing s. We shall also use the special program **chaos** which denotes a nondeterministic selection over all the states in S. A program t which can reach all states from all initial states (with probability 1) has no proper invariants, and thus satisfies $t^* = \mathbf{chaos}$.

Next we introduce our first generalisation — a probabilistic operator ${}_p\oplus$; its properties [9] also appear in Fig. 2. Observe that the sub-distribution of ${}_p\oplus$ corresponds to subadditivity of $\mathcal{T}S$.

[5] ν forms the greatest fixed point with respect to $\leq$ on $\mathcal{E}S$.

$$x \sqsubseteq y \quad \Leftrightarrow \quad x \,[]\, y = x \qquad\qquad x^* = 1 \,[]\, x \,[]\, x^*; x^*$$

$$x; (y \,[]\, 1) \sqsupseteq x \quad \Rightarrow \quad x; y^* = x$$

$$x; (y \,[]\, z) \quad \sqsubseteq \quad x; y \,[]\, x; z \qquad\qquad x; y \sqsupseteq y \quad \Rightarrow \quad x^*; y = y$$

$$(y \,[]\, z); x \quad = \quad y; x \,[]\, z; x$$

$$x \,{}_p\!\oplus\, y = y \,{}_{1-p}\!\oplus\, x \qquad\qquad x \,[]\, y \sqsubseteq x \,{}_p\!\oplus\, y$$

$$x; y \,{}_p\!\oplus\, x; z \quad \sqsubseteq \quad x; (y \,{}_p\!\oplus\, z) \qquad\qquad (y \,{}_p\!\oplus\, z); x \quad = \quad y; x \,{}_p\!\oplus\, z; x$$

$$x \,{}_p\!\oplus\, (y \,{}_q\!\oplus\, z) = (x \,{}_{\frac{p}{(p+q-pq)}}\!\oplus\, y) \,{}_{(p+q-pq)}\!\oplus\, z$$

x, y, z are interpreted as programs in $\mathcal{T}S$, and $0 < p < 1$. The axioms without ${}_p\!\oplus$ are similar to Kozen's axiomatisation of Kleene's language for regular expressions [11].

Fig. 2. Basic axioms

We say that a probability distribution F in $\overline{S}$ is *stationary* with respect to a Markov process t if whenever the input states are distributed as F, the output states are also distributed exactly according to F. In this section we generalise this idea to all programs in $\mathcal{T}S$.

Observe first that any F in $\overline{S}$ can be modelled as the program that outputs F — we call such programs deterministic assignments. Writing $\hat{F}$ for the deterministic assignment that outputs F for any initial state, we can see that the definition of stationarity above is the same as saying that $\hat{F}; t = \hat{F}$ holds as an equality in $\mathcal{T}S$.

Our crucial generalising step is now to consider *any* program t' satisfying $t'; t = t'$ to represent stationary behaviour (rather than only those programs $\hat{F}$ generated from distributions F as above); that takes us beyond the classical treatment.

To fill in the details, we begin with the idea of weakest stationary program, as follows. We make use of x^* to encode "all invariants of x", noted above.

Definition 3.1 Define x^∞ to be the the least program that is stationary with respect to x (that is, which satisfies $x^\infty; x = x^\infty$) *and* which preserves all invariants of x (that is $x^* \sqsubseteq x^\infty; x^*$). We have

$$x^\infty \quad := \quad ([]y : \mathcal{H}S \bullet y; x \sqsubseteq y \quad \wedge \quad x^* \sqsubseteq y; x^*) \; .$$

Note that an important intuitive property of x^∞ is that it preserves all invariants of x — an alternative definition that only considers stationarity (the first conjunct in Def. 3.1) gives the incorrect

$$([]y : \mathcal{H}S \bullet y; 1 \sqsubseteq y) \;=\; \mathbf{chaos} \;\neq\; 1^\infty = 1$$

for the case $x = 1$.

$$x^*; x^\infty = x^\infty \qquad x^{\infty\infty} = x^\infty$$
$$x^\infty \sqsubseteq x^{n\infty} \qquad x^* \sqsubseteq x^{n*}$$
$$x^* \sqsubseteq x^\infty \qquad x^{*\infty} = x^*$$
$$x; (y; x)^\infty \sqsubseteq (x; y)^\infty; x \qquad x; x^\infty = x^\infty = x^\infty; x$$

$$(p > 0) \;\Rightarrow\; (x \,{}_p\!\oplus 1)^\infty \sqsubseteq x^\infty \quad x^*; x = x^* \;\Rightarrow\; x^* = x^\infty$$
$$x; y \sqsubseteq z; x \;\Rightarrow\; x; y^\infty \sqsubseteq z^\infty; y \quad x^{n*} = x^* \;\Rightarrow\; x^\infty = x^{n\infty}$$

To avoid clutter, we write $x^{n\infty}$ etc. instead of $(x^n)^\infty$.

Fig. 3. A selection of basic theorems

Program t^∞ can be thought of as delivering from initial state s the strongest invariant reachable from s, whilst preserving the probabilistic stationary behaviour. In fact t^∞ in $\mathcal{T}S$ is the the limit of the increasing chain of programs $t^* \sqsubseteq t^*; t \sqsubseteq t^*; t^2 \sqsubseteq \ldots \sqsubseteq t^*; t^n \sqsubseteq \ldots$ That limit is well-defined since $\mathcal{T}S$ is directed-complete, and hence we have the additional fact

$$(1) \qquad (\forall n > 0 \bullet x^*; x^n \sqsubseteq y) \quad \Rightarrow \quad x^\infty \sqsubseteq y \;.$$

In Fig. 3 we set out some general theorems about $^\infty$ and *, all implied by the axioms of Fig. 2 and the properties of $^\infty$ set out in Def. 3.1 and (1).

To see the difference between * and $^\infty$ we reconsider FP from Fig. 1. The only nontrivial invariant set of states is $\{ok, \neg ok\}$, hence $FP^* = ok \,[]\, \neg ok$; but this program is not stationary with respect to FP, and so $FP^\infty \neq FP^*$. In fact $FP^\infty = (ok_{3/5} \oplus \neg ok) \,[]\, ok$, the generalised distribution in which the probability of ok is at least $3/5$.

4 Extended Markov theory

From (1) it is easy to see that in the general setting, any program t (if executed for long enough) achieves some notion of stationary behaviour encapsulated by the program t^∞. But that is not the view taken by classical Markov process theory. To see where the general and the classical theories diverge, consider the program $b := 1{-}b$, where the variable b can only take values in $\{0, 1\}$. The classical theory says that this program does not converge (because it oscillates between b's two values). On the other hand $(b := 1{-}b)^\infty \;=\; (b := 1{-}b)^* \;=\; (b := 0 \,[]\, b := 1)$, which says that the long term stationary behaviour is a program that assigns to b nondeterministically from its type. That behaviour is disqualified by the classical theory because it is not deterministic and so does not represent a distribution. We discuss the "observational" intuition behind this solution in the next section.

For now we end this section by demonstrating that our generalised notion

of convergence really supersedes the classical theory. We present a new proof of the important result about convergence to a stationary distribution of "aperiodic" Markov processes; the proof relies crucially on the ability to postulate the existence of t^∞ for all Markov processes, and not just those permitted by the classical theory.

Recall that a distribution is modelled as a deterministic assignment which is independent of the initial state. A transformer t which corresponds to such an assignment is additive and, for any α, the expectation $t.\alpha$ is a constant function. For example $wp.(ok\,{}_{2/3}\!\oplus \neg ok).\alpha$ returns the expected (final) value of α, which is constant at $2(\alpha.ok)/3 + \alpha.(\neg ok)/3$, whatever the initial value.

Hence in our terms all we need do is show that $^\infty$ maps the aperiodic deterministic programs to transformers that correspond to deterministic assignments.

Aperiodicity is a property of t provided that all states are eventually reachable from all other states, and the probability of returning to the original state with a definite period is strictly less than 1 [8]. The first property is the same as saying that $t^* = \mathbf{chaos}$, and the second is the same as saying that $t^{n*} = t^*$ for all $n > 1$ — in the case that the equality fails for some n, we are saying that t exhibits a period of n. The general theorem about convergence of Markov processes is then as follows.

Theorem 4.1 *If t in $\mathcal{T}S$ is deterministic and aperiodic then t^∞ is a deterministic assignment.*

Proof*: The comment after Lem. 2.2 implies that t^n must be a contraction for some $n > 0$, and hence $t^{n\infty}$ must be a deterministic assignment (also by Lem. 2.2). The result follows from Fig. 3 since $t^{n*} = t^*$.*

5 Applications to long-term average behaviour

The properties of systems that execute indefinitely are usually investigated using an adaptation of temporal logic — in our case *probabilistic* temporal logic. Formulae are interpreted over trees of execution paths — in our case *probabilistic distributions* over execution paths [15,2]. The interpretation of a typical formula ϕ over a path-distribution yields the proportion of paths satisfying ϕ. As de Alfaro points out [3] however, this kind of "probabilistic satisfaction" refers to the aggregate path-distribution; put another way it measures the chance of a single event occurring *among* paths, and ignores the frequency with which events occur *along* paths. But this is precisely what is called for in *availability* or long-term average analyses of failing systems. In this section we show that both are determined by t^∞ — even for systems that include nondeterminism, such as *FP* in Fig. 1.

We define long-term average behaviour as de Alfaro [3] does. Given a sequence seq of expectations, let seq_i be the i'th element, and define the partial sum $\sum_k seq = seq_1 + seq_2 + \ldots + seq_k$.

Definition 5.1 Let t in $\mathcal{T}S$ execute indefinitely, and let α be a predicate. The long-term average number of occurrences of α observed to hold as t executes is given by $V_t.\alpha$ in

$$V_t.\alpha \quad := \quad \liminf_{k\to\infty} \ \frac{\sum_k seq}{k} \ ,$$

where in this case $seq_k := t^*; t^k.\alpha$.

Def. 5.1 corresponds to the average result after sampling the state of the system at arbitrary intervals of time as t executes repeatedly. Here we assume that at the k'th sample point, the system has executed *at least* k times — and in that case the chance that α holds at the time of the test is $t^*; t^k.\alpha$. When t corresponds to a Markov process that converges classically, that average is determined by the stationary distribution. We have a corresponding result here, but it is valid for *all* programs.

Lemma 5.2 *Let t be a program in $\mathcal{H}S$ and α an expectation in $\mathcal{E}S$. Then we have $t^\infty.\alpha \ = \ V_P.\alpha$.*

To illustrate the above, recall the program $b := 1{-}b$, and let $[b = 0]$ represent the expectation that evaluates to 1 at states where b is 0 and to 0 elsewhere. To calculate $V_{b:=1-b}.[b = 0]$ we consider

$$wp.(b := 1{-}b)^*; (b := 1{-}b)^n.[b = 0] \ = \ \underline{0} \ ,$$

hence $V_{b:=1-b}.[b = 0] = \underline{0}$ as well.

Alternatively, $(b := 1{-}b)^\infty = (b := 0 \,[]\, b := 1)$, hence $wp.(b := 1{-}b)^\infty.[b = 0] = 0$ also.

These results can be understood operationally in the context of a tester who is allowed to choose when to sample the state of the program. Clearly if the tester only observes the state after an even number of executions of $b := 1{-}b$ then he will deduce that b is never 0 on average (or even at all). The point about aperiodic programs in the classical theory is that the average measurement is to an extent robust against such accidental testing bias. And the same applies here: whatever the proposed testing regime, the proportion of time that FP is ok will be found to be at least $3/5$, since $FP^\infty = (ok \,{}_{3/5}\!\oplus \neg ok) \,[]\, ok$.

6 Conclusion

Our main contribution is to extend the notion of stationary behaviour of Markov processes to a model that includes demonic nondeterminism, setting it on a par with other programming concepts. The main insight was to model stationary behaviour explicitly as a distribution-generating program in $\mathcal{T}S$; that allows access to the techniques of program algebra and probabilistic models [1,13]. The generalisation proposed here allows the completion of the theory linking long-term average behaviour and stationary behaviour — both are now

always defined, and they determine each other. Moreover our generalisation provides a striking simplification to classical theory of convergence.

The operator t^* presented here is unable to express many of the esperiments offered by the much more elaborate framework due to de Alfaro [3]. The main difference is that results are assigned to states rather than transitions. Nevertheless many useful performance measures are covered by this simpler framework. Examples include average waiting times and availability measures.

Further work is needed to incorporate other programming notions such as coercions [12], which significantly increase the power of algebraic reasoning.

An important consequence is that stationary behaviour is now susceptible to other programming techniques such as refinement and data abstraction [7].

References

[1] Ernie Cohen. Separation and reduction. In *Mathematics of Program Construction, 5th International Conference, Portugal, July 2000*, number 1837 in LNCS, pages 45–59. Springer Verlag, 2000.

[2] L. de Alfaro. Temporal logics for the specification of performance and reliability. *Proceedings of STACS '97*, LNCS volume 1200, 1997.

[3] L. de Alfaro. How to specify and verify the long-run average behavior of parobabilistic systems. In *Proceedings of 'LICS '98, 23-24 June, Indianapolis*, 1998.

[4] C. Derman. *Finite State Markov Decision Processes.* Academic Press, 1970.

[5] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall International, Englewood Cliffs, N.J., 1976.

[6] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, second edition, 1971.

[7] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[8] G. Grimmett and D. Welsh. *Probability: an Introduction.* Oxford Science Publications, 1986.

[9] Jifeng He, K. Seidel, and A. K. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2,3):171–192, January 1997.

[10] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[11] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:336–390, 1994.

[12] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, second edition, 1994.

[13] C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[14] C.C. Morgan. Private communication. 1995.

[15] R. Segala. Modeling and verification of randomized distributed real-time systems. PhD Thesis, 1995.

[16] M. Sharir, A. Pnueli, and S. Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, May 1984.

[17] N. Storey. *Safety-critical computer systems.* Addison-Wesley, 1996.

[18] M. Vardi. Automatic verification of probabilistic concurrent finite-state systems. *Proceedings of 26th IEEE Symposium on Found. of Comp. Sci.*, pages 327–338, 1985.

MFPS 17 Preliminary Version

A Selective CPS Transformation

Lasse R. Nielsen

BRICS[1]

Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
E-mail: lrn@brics.dk

Abstract

The CPS transformation makes all functions continuation-passing, uniformly. Not all functions, however, need continuations: they only do if their evaluation includes computational effects. In this paper we focus on control operations, in particular "call with current continuation" and "throw". We characterize this involvement as a control effect and we present a selective CPS transformation that makes functions continuation-passing if they have a control effect, and that leaves the others in direct style. We formalize this selective CPS transformation with an operational semantics and a simulation theorem à la Plotkin.

1 Introduction

This paper defines, and proves correct, a selective Continuation-Passing Style (CPS) transformation, i.e., one that preserves part of the program in direct style. It uses information about a program's effect-behavior to guide the transformation. The particular computational effect we use is the control-transfer effect exemplified by "call with current continuation" [3].

1.1 Related work

Selectively CPS transforming a program is not a new idea.

Danvy and Hatcliff defined a selective CPS transformation based on strictness analysis [6]. When dealing with the effect of non-termination, a strict function is indifferent to the evaluation order of its argument, and as such arguments of strict functions could be transformed by a call-by-value transformation while the rest of the program was transformed by a call-by-name transformation. The same authors also investigated CPS transformation based

[1] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

on totality information, defining a selective transformation [7]. There is no immediate generalization of strictness to other effects than non-termination, though, whereas triviality (the absence of effects) generalizes immediately to all other computational effects, as we have exemplified with control effects.

Kim, Yi, and Danvy implemented a selective CPS transformation for the core language of SML of New Jersey to reduce the overhead of their CPS-transformation which λ-encoded the exception effects of SML [15]. The implementation is very similar to the present work, though they treat exceptions instead of control operations and base the annotation on a specific exception analysis.

Recent work has focused on selective transformations for different reasons.

Reppy introduced a local CPS transformation in an otherwise direct-style compiler to improve the efficiency of nested loops [18]. Kim and Yi defined coercions between direct style and CPS terms with no other effects than non-termination, allowing arbitrary subexpressions to be transformed, and facilitating interfacing to external code in both direct style and CPS. The selective transformation was proven correct [14].

The present paper defines a selective CPS transformation for the simply typed λ-calculus extended with recursive functions and computational effects, namely CALLCC and THROW, into λ-calculus with only recursive functions, i.e., with only non-termination as an effect. The transformation is based on an effect analysis, and it is proven correct with respect to the dynamic behavior of the program. We conjecture that the methodology extends to other types of effects, only differing in the details of the encoding of effectful primitives into λ-expressions.

1.2 Overview

Section 2 gives the syntax and semantics of a small, typed functional language with control effects, and shows the traditional (non-selective) CPS transformation. Section 3 extends the language with effect annotations which are verified by an effect system, and defines the selective CPS transformation guided by these annotations. Section 4 proves the correctness of the selective CPS transformation using Plotkin-inspired colon translations, and Section 5 concludes.

2 Definitions

We define the source and target language of our selective CPS transformation, giving the syntax and type predicates as well as an operational semantics.

2.1 Syntax

The source language is a call-by-value typed functional language with recursive functions and the canonical control operators: CALLCC and THROW. The

syntax is given in Figure 1, where x and f range over a set of identifiers and c ranges over a set of constants.

$$e \ ::= \ \mathrm{c} \mid \mathrm{x} \mid \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e \mid e \ @ \ e \mid \textsc{callcc}\ \mathrm{x}.e \mid \textsc{throw}\ e\ e$$

Fig. 1. Abstract syntax of the source language

We identify expressions up to renaming of bound variables, i.e., FUN f x.x = FUN g y.y. We use the shorthand $\lambda \mathrm{x}.e$ for a function-abstraction FUN f x.e where f does not occur in e.

We require expressions to be well typed with regard to the typing rules in Figure 2, similar to those given by Harper, Duba, and MacQueen [12].

The syntax of types is given by the grammar:

$$\tau ::= b \mid \tau \rightarrow \tau \mid \langle \tau \rangle$$

where b ranges over a set of base types, and $\langle \tau \rangle$ is the type of continuations expecting a value of type τ.

In the rules ConstType is a mapping from constants to base types, giving the type of a constant, and Γ is a mapping from identifiers to types.

$$\frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash \mathrm{x} : \tau} \qquad \frac{\textsc{ConstType}(\mathrm{c}) = b}{\Gamma \vdash \mathrm{c} : b}$$

$$\frac{\Gamma[\mathrm{x} : \tau_1][\mathrm{f} : \tau_1 \rightarrow \tau_2] \vdash e : \tau_2}{\Gamma \vdash \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ @ \ e_2 : \tau_2}$$

$$\frac{\Gamma[\mathrm{x} : \langle \tau \rangle] \vdash e : \tau}{\Gamma \vdash \textsc{callcc}\ \mathrm{x}.e : \tau} \qquad \frac{\Gamma \vdash e_1 : \langle \tau \rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsc{throw}\ e_1\ e_2 : \tau_2}$$

Fig. 2. The typing rules of the source language.

We define a program to be a closed expression of type b_0, an arbitrary base type, and we only prove the correctness of the transformation on entire programs.

2.2 Semantics

The source language has a left-to-right call-by-value(CBV) operational semantics given using evaluation contexts. We introduce an intermediate value to the syntax to represent captured continuations, and define the values and contexts recursively, and give the context-based reduction rules (Figure 3).

Extended expressions, values, and evaluation contexts:

$$e ::= \ldots \mid \langle \mathrm{E} \rangle$$
$$v ::= \mathrm{c} \mid \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e \mid \langle \mathrm{E} \rangle$$
$$\mathrm{E} ::= [\,] \mid \mathrm{E}\ @\ e \mid v\ @\ \mathrm{E} \mid \textsc{throw}\ \mathrm{E}\ e \mid \textsc{throw}\ v\ \mathrm{E}$$

Reduction rules:

$$\mathrm{E}[(\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e)\ @\ v] \to \mathrm{E}[e\,[\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e/\mathrm{f}]\,[v/\mathrm{x}]]$$

$$\mathrm{E}[\textsc{callcc}\ \mathrm{x}.e] \to \mathrm{E}[e\,[\langle \mathrm{E} \rangle/\mathrm{x}]]$$

$$\mathrm{E}[\textsc{throw}\ \langle \mathrm{E}_1 \rangle\ v] \to \mathrm{E}_1[v]$$

Fig. 3. Semantics of the source language

Notice that continuations are represented as contexts, so throwing a continuation amounts to reinstating a context.

This semantics uses evaluation contexts in a way similar to Felleisen [9], capturing the fact that for any expression, there is at most one enabled reduction at a time.

Subject reduction ($e : \tau$ and $e \to e'$ implies $e' : \tau$) can be proven by a completely standard proof, which has been omitted.

2.3 The CPS transformation

The CPS transformation can be used to transform a program in the source language into a program in a similar language without CALLCC and THROW, while preserving the computational behavior of the source program. That is, the translated program, when applied to an initial continuation, terminates if the source program did, and the result of the transformed program, which is of a base type, is the same as that of the original program.

The CPS-transformation has other interesting properties:

- It generates programs that can be evaluated under both a call-by-name (CBN) and a CBV semantics and yield the same result, which corresponds to the result of the source program. A number of CPS transformations exist, each corresponding to an evaluation-order for the source language [13].
- It generates programs where all applications are tail-calls, i.e., no application is in an argument position.

CPS transformations are used in many places, but primarily in compilers to give an intermediate representation [1,21] and as a way to simplify the language of a program before applying other transformations or analyses to it [4]. It is the last application that is the motivation for the present work.

The standard CBV CPS transformation is defined in Figure 4.

$$
\begin{aligned}
\mathcal{C}[\![\mathrm{c}]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_v[\![\mathrm{c}]\!]) \\
\mathcal{C}[\![\mathrm{x}]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_v[\![\mathrm{x}]\!]) \\
\mathcal{C}[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_v[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!]) \\
\mathcal{C}[e_1 \mathbin{@} e_2] &= \lambda \mathrm{k}.\mathcal{C}[\![e_1]\!] \mathbin{@} \lambda \mathrm{v}.\mathcal{C}[\![e_2]\!] \mathbin{@} \lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} \mathrm{k} \\
\mathcal{C}[\![\textsc{callcc}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.\lambda \mathrm{x}.\mathcal{C}[\![e]\!] \mathbin{@} \mathrm{k} \mathbin{@} \mathrm{k} \\
\mathcal{C}[\![\textsc{throw}\ e_1\ e_2]\!] &= \lambda \mathrm{k}.\mathcal{C}[\![e_1]\!] \mathbin{@} \lambda \mathrm{v}.\mathcal{C}[\![e_2]\!] \mathbin{@} \mathrm{v}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_v[\![\mathrm{x}]\!]) &= \mathrm{x} \\
\mathcal{C}_v[\![\mathrm{c}]\!]) &= \mathrm{c} \\
\mathcal{C}_v[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!]) &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{C}[\![e]\!]
\end{aligned}
$$

Fig. 4. The CPS transformation

The $\mathcal{C}_v[\![\cdot]\!])$ function is used to coerce values and identifiers into CPS form, which consists of transforming the bodies of function abstractions. Values and identifiers share the property that Reynolds called "being trivial" [19] and Moggi called "being a value" (as opposed to a computation) [16], in the sense that they have no computational effects, including nontermination. The $\mathcal{C}[\![\cdot]\!]$ function is used on expressions with potential effects, the "serious" expressions.

3 The selective CPS transformation

As stated in the previous section, we treat trivial and serious expressions differently. Trivial expressions are those that have no computational effects and serious expressions are those that might have effects. The safe approximation used by the standard CPS transformation assumes that any application might have effects, which is not unreasonable when one considers nontermination as an effect.

In the source language we have added control effects, and it makes sense only to focus on those, and let the termination behavior be preserved by only evaluating the result in a CBV semantics. If we do that, we can use an effect analysis to find the parts of the program that are guaranteed to be free of the control effects generated by CALLCC and THROW. In the following we will use the words "trivial" and "non-trivial" about the absence or possible presence of *control* effects only, while ignoring the partiality effect of non-termination. That is, an expression that has an infinite reduction sequence can still be said to be "trivial" with regards to control effects. We are not aiming for evaluation-order independence, rather the source and target languages are assumed to have the same evaluation-order (call-by-value), so the translation need not take any measures to preserve or prevent non-termination in otherwise effect-free expressions.

This section defines effect-annotated expressions, an effect type system to check the consistency of the annotation, and a selective CPS transformation that keeps trivial applications in direct style.

3.1 Annotated source language

We annotate a program with annotations taken from the set $\{\mathrm{T}, \mathrm{N}\}$, which is a partial order with the ordering relation $\mathrm{N} < \mathrm{T}$.

We mark some applications as trivial, with a T, and some as (potentially) non-trivial, with an N. The trivial ones are kept in direct style, and as such do not expect to receive a continuation.

For an expression, an effect analysis can tell us one of three things:

- Some evaluation of the expression will certainly give rise to effects (meaning, in this case, the reduction sequence of the expression contains evaluations of CALLCC or THROW expressions),
- no evaluation of the expression will give rise to effects, or
- we just don't know either way, which can happen since the problem is generally undecidable.

The present transformation aims to keep the expressions in the second case in direct style. Since any effectful expression *must* be put into CPS, we must treat the first and third cases equally, and they are both marked N. Unifying these two cases gives rise to the stated ordering where greater means more

information is known: certainty of the absence of effects as opposed to only possible presence.

$$\mathrm{A} ::= \mathrm{T} \mid \mathrm{N}$$

$$e ::= \mathrm{c} \mid \mathrm{x} \mid \textsc{fun}^{\mathrm{A}}\ \mathrm{f}\ \mathrm{x}.e \mid (e\ @^{\mathrm{A}}\ e)^{\mathrm{A}} \mid \textsc{callcc}\ \mathrm{x}.e \mid \textsc{throw}\ e\ e$$

These are the minimal annotations needed for our purpose. We treat values and identifiers (the traditional trivial expressions) as if they were annotated as such, i.e., $(e)^{\mathrm{T}}$ is a match for any trivial expression, just as $(e)^{\mathrm{N}}$ matches the two control operators.

We require that an expression annotated as trivial actually is so, i.e., whenever it is evaluated, the reduction sequence contains no steps corresponding to reductions of CALLCC or THROW expressions. Since we use this annotation as a basis for the selective CPS transformation, we will want to CPS transform all expressions that are not marked trivial.

We have to treat functions and applications with special care. When a lambda abstraction is applied at an application point, the body of the abstraction is also evaluated at that point. If the body is not trivial, then neither is the application, and after selective CPS transformation, the transformed application must pass a continuation to the transformed body, and the body should expect a continuation.

In a higher-order program, more than one abstraction can be applied at the same application point, and after transformation, all of these abstractions must either expect a continuation or not. That means that all functions that can end up in a given application must be transformed in the same way. That divides the abstractions into two groups, those transformed into CPS, i.e., expecting a continuation, and those kept in direct style, i.e., not expecting a continuation. Some abstractions with a trivial body might be transformed to expect a continuation in order to match the other abstractions that reach the same application points.

We will say that the annotation is "consistent" (with regards to the behavior of the program) if:

- All expressions marked trivial are trivial,
- all abstractions marked trivial, or non-trivial, are only applied at application points marked trivial, or non-trivial respectively, and
- all abstractions whose body are marked non-trivial, are themselves marked as non-trivial.

To check all this, we use an extension of the type system to an effect type system that guarantees that the annotation is consistent. The types are also

annotated, so the grammar of types is:

$$\tau ::= b \mid \tau \xrightarrow{\mathrm{A}} \tau \mid \langle \tau \rangle$$

The effect system is shown in Figure 5.

$$\frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash \mathrm{x} : \tau, \mathrm{T}} \qquad \frac{\textsc{ConstType}(\mathrm{c}) = b}{\Gamma \vdash \mathrm{c} : b, \mathrm{T}}$$

$$\frac{\Gamma[\mathrm{x} : \tau_1][\mathrm{f} : (\tau_1 \xrightarrow{\mathrm{A}_1} \tau_2)] \vdash e : \tau_2, \mathrm{A} \quad \mathrm{A}_1 \leq \mathrm{A}}{\Gamma \vdash \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e : \tau_1 \xrightarrow{\mathrm{A}_1} \tau_2, \mathrm{T}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\mathrm{A}_3} \tau_2, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau_1, \mathrm{A}_2 \quad \mathrm{A} \leq \min(\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3)}{\Gamma \vdash e_1 \ @\ e_2 : \tau_2, \mathrm{A}}$$

$$\frac{\Gamma[\mathrm{x} : \langle \tau \rangle] \vdash e : \tau, \mathrm{A}}{\Gamma \vdash \textsc{callcc}\ \mathrm{x}.e : \tau, \mathrm{N}} \qquad \frac{\Gamma \vdash e_1 : \langle \tau \rangle, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau, \mathrm{A}_2}{\Gamma \vdash \textsc{throw}\ e_1\ e_2 : \tau_2, \mathrm{N}}$$

Fig. 5. Effect type system

If an expression is typeable in the original type system, then there exists at least one annotation that is typeable in the effect system, namely the one where all functions and applications are marked non-trivial.

The $\leq$ in the rule for function abstractions is exactly due to the restriction on which functions that can flow where. Functions with trivial bodies can be annotated with an N, allowing them to flow into an application expecting to pass a continuation, but this is reflected in the type, which is the only thing that is known at the application point. There are two different annotated function types, one for each annotation, and only one of these is allowed at each application point.

The $\leq$ on the rule for applications is discussed in the next section,

We only consider well-annotated expressions from here on, i.e., expressions that are allowed by the effect typing rules.

3.2 The Selective CPS Transformation

We define a CPS transformation that transforms well-annotated expressions and leaves trivial applications in direct style (Figure 6).

$$
\begin{aligned}
\mathcal{S}[\![e^{\mathrm{T}}]\!] &= \mathrm{k} \mathbin{@} \mathcal{S}_v[\![e^{\mathrm{T}}]\!] \\
\mathcal{S}[\![(e_1 \mathbin{@^{\mathrm{N}}} e_2)^{\mathrm{N}}]\!] &= \lambda \mathrm{k}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} \mathrm{k})) \\
\mathcal{S}[\![\textsc{callcc}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.(\lambda \mathrm{x}.\mathcal{S}[\![e]\!] \mathbin{@} \mathrm{k}) \mathbin{@} \mathrm{k} \\
\mathcal{S}[\![\textsc{throw}\ e_1\ e_2]\!] &= \lambda \mathrm{k}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_v[\![\mathrm{x}]\!] &= \mathrm{x} \\
\mathcal{S}_v[\![\mathrm{c}]\!] &= \mathrm{c} \\
\mathcal{S}_v[\![\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}[\![e]\!] \\
\mathcal{S}_v[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_v[\![e]\!] \\
\mathcal{S}_v[\![e_1 \mathbin{@^{\mathrm{T}}} e_2]\!] &= \mathcal{S}_v[\![e_1]\!] \mathbin{@} \mathcal{S}_v[\![e_2]\!]
\end{aligned}
$$

Fig. 6. The selective CPS transformation

3.3 Semantics of annotated syntax

We do not change the semantics of the language, since the annotation is just a mark on the expressions, and it is only used by the CPS transformation. Still, in order to prove the correctness of the transformation, we define a reduction relation on annotated expressions that updates the annotation.

$$
\begin{aligned}
\mathrm{E}\big[((\textsc{fun}^{\mathrm{A}_1}\ \mathrm{f}\ \mathrm{x}.(e)^{\mathrm{A}_3}) \mathbin{@^{\mathrm{A}_2}} v)^{\mathrm{A}}\big] &\to \mathrm{E}\big[(e)^{\mathrm{A}_3}\ \big[\textsc{fun}^{\mathrm{A}_1}\ \mathrm{f}\ \mathrm{x}.(e)^{\mathrm{A}_3}/\mathrm{f}\big]\ [v/\mathrm{x}]\big] \\
\mathrm{E}\big[\textsc{callcc}\ \mathrm{x}.(e)^{\mathrm{A}}\big] &\to \mathrm{E}\big[(e)^{\mathrm{A}}\ [\langle \mathrm{E} \rangle/\mathrm{x}]\big] \\
\mathrm{E}[\textsc{throw}\ \langle \mathrm{E}' \rangle\ v] &\to \mathrm{E}'[v]
\end{aligned}
$$

The point of this reduction relation is that values and identifiers are always marked trivial, and no expression marked trivial can ever reduce to one marked as non-trivial.

With these reduction rules, an expression marked non-trivial can reduce to one marked trivial, typically by reducing it to a value. If that happens to one of the subexpressions of an application, we can suddenly be in the situation where both of the subexpressions are trivial as well as the bodies of the functions expected to be applied there, and the entire application could now be consistently annotated as trivial. The weakening in the effect-typing rule for applications is there to avoid that such a change would mandate changes to annotations not local to the reduction taking place.

All these properties make a proof of Subject Reduction a trivial extension of the proof for the unannotated syntax.

One reason for having both annotations and an effect system, and not, e.g., only the effect system, is for ease of representation. Even if a reduced program allows a more precise effect-analysis than the original program, the transformation is based on the original program, and the annotation keeps the original annotation throughout the reduction sequence.

4 Proof of correctness

To prove the correctness of the transformation, we must first specify a notion of correctness. In this case we require that the transformed program reduces to the same result as the original program.

Theorem 4.1 (Correctness of the Selective CPS Transformation) *If e is a closed and well-annotated expression of type b_0 then*

$$e \to^* v \Leftrightarrow \mathcal{S}[\![e]\!] \mathbin{@} (\lambda \mathrm{x.x}) \to^* v$$

In Plotkin's original proof, the result of the transformed program would be $\mathcal{S}_v[\![v]\!]$, but since the program has a type where the only values are constants, and all constants satisfy $\mathcal{S}_v[\![\mathrm{c}]\!] = \mathrm{c}$, we can state the theorem as above.

4.1 The selective colon-translations

The proof uses a method similar to Plotkin's in his original proof of the correctness of the CPS transformation [17]. It uses a so-called "colon-translation" to bypass the initial administrative reductions and focus on the evaluation point.

The intuition that drives the normal CPS transformation is that if e reduces to v then $(\mathcal{C}[\![e]\!] \mathbin{@} k)$ should evaluate to $(k \mathbin{@} \mathcal{C}_v[\![v]\!])$. Plotkin captured this in his colon translation where if $e \to e'$ then $e : k \to^* e' : \mathrm{k}$, and at the end of the derivation, values satisfied $v : k = k \mathbin{@} \Psi(v)$, where $\Psi(\cdot)$ is what we write $\mathcal{S}_v[\![\cdot]\!]$.

The idea of the colon translation is that in $e : k$, the k represents the *context* of e, which in the transformed program has been collected in a continuation: a function expecting the result of evaluating e. The colon separates the source program to the left and the transformed program to the right of it. In the selective CPS transform, some contexts are not turned into continuations, namely the contexts of expressions marked trivial, since such expressions are not transformed to CPS expressions, and as such does not expect a continuation.

Therefore we have two colon translations, one for non-trivial expressions, with a continuation function after the colon, and one for trivial expressions with an evaluation context after the colon. The definition is shown in Figure 7. In both cases, what is to the left of the colon is a piece of source syntax, and

what is to the right is a representation of the context of that expression in the source program translated to the target language. If the expression is trivial, the source context is represented by a context in the target language, and the translation of the expression is put into this context. If the expression is not trivial, then the source context is represented by a continuation function which is passed to the translation of the expression.

$$
\begin{array}{lll}
e^{\mathrm{T}} : k = e^{\mathrm{T}} : [k \mathbin{@} [\,]] & & \\
(e_1 \mathbin{@^{\mathrm{N}}} e_2)^{\mathrm{N}} : k = e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k) & & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{N}}} e_2)^{\mathrm{N}} : k = e_2 : \lambda \mathrm{v}'.\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathrm{v}' \mathbin{@} k & & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{N}}} v_2)^{\mathrm{N}} : k = \mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathcal{S}_v[\![v_2]\!] \mathbin{@} k & & \\
(e_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{N}} : k = e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.k \mathbin{@} (\mathrm{v} \mathbin{@} \mathrm{v}')) & & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{N}} : k = e_2 : \lambda \mathrm{v}'.k \mathbin{@} (\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathrm{v}') & & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{T}}} v_2)^{\mathrm{N}} : k = k \mathbin{@} (\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathcal{S}_v[\![v_2]\!]) & & \\
\textsc{callcc}\ \mathrm{x}.e : k = (\lambda \mathrm{x}.\mathcal{S}[\![e]\!] \mathbin{@} k) \mathbin{@} k & & \\
\textsc{throw}\ e_1\ e_2 : k = e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v} & & \text{if } e_1 \text{ is not a value} \\
\textsc{throw}\ v_1\ e_2 : k = e_2 : \mathcal{S}_v[\![v_1]\!] & & \text{if } e_2 \text{ is not a value} \\
\textsc{throw}\ v_1\ v_2 : k = \mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathcal{S}_v[\![v_2]\!] & &
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{x} : \mathrm{E} = \mathrm{E}[\mathrm{x}] & & \\
\mathrm{c} : \mathrm{E} = \mathrm{E}[\mathrm{c}] & & \\
\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e : \mathrm{E} = \mathrm{E}[\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}[\![e]\!]] & & \\
\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e : \mathrm{E} = \mathrm{E}[\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_v[\![e]\!]] & & \\
(e_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{T}} : \mathrm{E} = e_1 : \mathrm{E}\big[([\,] \mathbin{@^{\mathrm{T}}} \mathcal{S}_v[\![e_2]\!])^{\mathrm{T}}\big] & & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{T}} : \mathrm{E} = e_2 : \mathrm{E}\big[(\mathcal{S}_v[\![v_1]\!] \mathbin{@^{\mathrm{T}}} [\,])^{\mathrm{T}}\big] & & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@^{\mathrm{T}}} v_2)^{\mathrm{T}} : \mathrm{E} = \mathrm{E}\big[(\mathcal{S}_v[\![v_1]\!] \mathbin{@^{\mathrm{T}}} \mathcal{S}_v[\![v_2]\!])^{\mathrm{T}}\big] & &
\end{array}
$$

Fig. 7. The selective colon translation on expressions

In Plotkin's colon translation, $v : k = k \mathbin{@} \Phi(v)$. This also holds for this colon translation pair, since $v : k = v : [k \mathbin{@} [\,]]$, since v is trivial, and $v : [k \mathbin{@} [\,]] = k \mathbin{@} v$ by the definition of the $e : \mathrm{E}$-translation.

The e : E-translation is not as significant as the $e : k$-translation, since all it does is apply the Ψ-function to the argument, i.e., if e is a trivial expression then e : E = E$[\mathcal{S}_v[\![e]\!]]$. There are no administrative reductions to bypass in direct style.

We plan to use the colon translations on the result of reducing on the annotated expressions, so we extend it to work on continuation values, $\langle \mathrm{E} \rangle$, which are values and as such trivial.

$$\langle \mathrm{E}' \rangle : \mathrm{E} \;=\; \mathrm{E}[\mathrm{E}' : \mathrm{id}]$$

where id $= \lambda$x.x and E : k defines either a continuation function or a context as displayed in Figure 8, where E^{T} represents any non-empty context with a top-most annotation as trivial.

$$\begin{aligned}
[\,] : k &= k \\
\mathrm{E}^{\mathrm{T}} : k &= \mathrm{E}^{\mathrm{T}} : [k \mathbin{@} [\,]] \\
(\mathrm{E} \mathbin{@^{\mathrm{N}}} e_2)^{\mathrm{N}} : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k) \\
(\mathrm{E} \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{N}} : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.k \mathbin{@} (\mathrm{v} \mathbin{@} \mathrm{v}')) \\
(v_1 \mathbin{@^{\mathrm{N}}} \mathrm{E})^{\mathrm{N}} : k &= \mathrm{E} : (\lambda \mathrm{v}'.\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathrm{v}' \mathbin{@} k) \\
(v_1 \mathbin{@^{\mathrm{T}}} \mathrm{E})^{\mathrm{N}} : k &= \mathrm{E} : (\lambda \mathrm{v}'.k \mathbin{@} (\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathrm{v}')) \\
\textsc{throw}\ \mathrm{E}\ e_2 : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v} \\
\textsc{throw}\ v_1\ \mathrm{E} : k &= \mathrm{E} : \mathcal{S}_v[\![v_1]\!]
\end{aligned}$$

$$\begin{aligned}
[\,] : \mathrm{E} &= \mathrm{E} \\
(\mathrm{E} \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{T}} : \mathrm{E}' &= \mathrm{E} : \mathrm{E}'[[\,] \mathbin{@} \mathcal{S}_v[\![e_2]\!]] \\
(v_1 \mathbin{@^{\mathrm{T}}} \mathrm{E})^{\mathrm{T}} : \mathrm{E}' &= \mathrm{E} : \mathrm{E}'[\mathcal{S}_v[\![v_1]\!] \mathbin{@} [\,]]
\end{aligned}$$

Fig. 8. The selective colon translation on contexts.

The E : k-translation yields either continuation functions or contexts, depending on the annotation of the innermost levels of the context argument, and the E : E-translation always gives a context, but requires that the first argument's outermost annotation is trivial.

These colon-translations satisfy a number of correspondences.

Proposition 4.2 *For all contexts* E_1, E_2, *and* E_3, *and continuation functions*

(closed functional values) the following equalities hold.

$$\mathrm{E}_1[\mathrm{E}_2[\ ]] : k = \mathrm{E}_2 : (\mathrm{E}_1 : k)$$
$$\mathrm{E}_1[\mathrm{E}_2[\ ]] : \mathrm{E}_3 = \mathrm{E}_2 : (\mathrm{E}_1 : \mathrm{E}_3)$$

Proof. The proof is by simple induction on the context E_1.

- If $\mathrm{E}_1 = [\ ]$ then $(\mathrm{E}_1[\mathrm{E}_2[\ ]] : k) = (\mathrm{E}_2 : k) = (\mathrm{E}_2 : (\mathrm{E}_1 : k))$ and $(\mathrm{E}_1[\mathrm{E}_2[\ ]] : \mathrm{E}_3) = (\mathrm{E}_2 : \mathrm{E}_3) = (\mathrm{E}_2 : (\mathrm{E}_1 : \mathrm{E}_3))$.
- If $\mathrm{E}_1 = \left[(\mathrm{E}\ @^{\mathrm{N}}\ e_2)^{\mathrm{N}}\right]$ then

$$\begin{aligned}
\mathrm{E}_1[\mathrm{E}_2[\ ]] : k &= (\mathrm{E}[\mathrm{E}_2]\ @^{\mathrm{N}}\ e_2)^{\mathrm{N}} : k \\
&= \mathrm{E}[\mathrm{E}_2] : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @\ (\lambda \mathrm{v}'.\mathrm{v}\ @\ \mathrm{v}'\ @\ k) && \text{(def. of } \mathrm{E} : k) \\
&= \mathrm{E}_2 : (\mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @\ (\lambda \mathrm{v}'.\mathrm{v}\ @\ \mathrm{v}'\ @\ k)) && \text{(I.H.)} \\
&= \mathrm{E}_2 : (\left[(\mathrm{E}\ @^{\mathrm{N}}\ e_2)^{\mathrm{N}}\right] : k) && \text{(def. E} : k)
\end{aligned}$$

- The remaining cases are similar.

□

One would expect that similar equalities hold for the colon translations on expressions, i.e., $\mathrm{E}[e] : k = e : (\mathrm{E} : k)$ and $\mathrm{E}[e] : \mathrm{E}' = e : (\mathrm{E} : \mathrm{E}')$, and indeed these equalities hold in most cases. The exception is when E is non-empty and the "innermost" expression of the context is not annotated as trivial, e.g., $\mathrm{E}_1\left[([\ ]\ @\ e_1)^{\mathrm{N}}\right]$ for some context E_1 and expression e_1, and e is a value. Normally the $e : k$ translation descends the left-hand side and rebuilds the context on the right hand side, either as a continuation function or as a context, depending on the annotation. The exception mentioned, $\mathrm{E}[e] : k$, the focus of the colon translation, the expression on the left hand side of the colon, would never descend all the way down to a value. We have made special cases for $v\ @\ e$ to bypass administrative reductions, so $\mathrm{E}[v] : k$ would not equal $v : (\mathrm{E} : k)$, because the latter introduces an administrative reduction. Reducing that administrative reduction, applying k to $\mathcal{S}_v[\![v]\!]$, does lead to $v : (\mathrm{E} : k)$ again in one or more reduction steps. That is, if e is a value and E is not a trivial context then $e : (\mathrm{E} : k) = (\mathrm{E} : k)\ @\ \mathcal{S}_v[\![e]\!] \to^* \mathrm{E}[e] : k$, and likewise for the $e :$ E-relation.

Proposition 4.3 *For all contexts* E *and* E′*, expressions* e*, and continuation functions* k

$$e : (\mathrm{E} : k) \to^* \mathrm{E}[e] : k$$
$$e : (\mathrm{E} : \mathrm{E}') \to^* \mathrm{E}[e] : \mathrm{E}' \quad (\textit{if } e \textit{ trivial})$$

and $\to^*$ *is* $\to^0$*, i.e., equality, if* e *is not a value.*

Proof. Omitted. □

4.2 Colon-translation lemmas

Plotkin used four lemmas to prove his simulation and indifference theorems. We only prove simulation, which corresponds to Plotkin's simulation, since we already know that indifference does not hold for a selective CPS transformation (at least unless the selectivity is based on the effect of nontermination as well).

Lemma 4.4 (Substitution) *If* $\Gamma[\mathrm{x} : \tau_1] \vdash e : \tau_2, \mathrm{A}$ *and* $\vdash v : \tau_1, \mathrm{T}$ *is a closed value then*

$$\mathcal{S}[\![e]\!]\,[\mathcal{S}_v[\![v]\!]/\mathrm{x}] = \mathcal{S}[\![e\,[v/\mathrm{x}]]\!]$$
$$\mathcal{S}_v[\![e]\!]\,[\mathcal{S}_v[\![v]\!]/\mathrm{x}] = \mathcal{S}_v[\![e\,[v/\mathrm{x}]]\!] \text{ (} \textit{if } e \textit{ is trivial}\text{)}$$

Proof. The proof is by induction on the structure of e, using the distributive properties of substitution and taking the trivial cases before the non-trivial ones (because the $\mathcal{S}[\![\cdot]\!]$ translation defers trivial subexpressions to the Ψ transformation). The details have been omitted. □

Lemma 4.5 (Initial reduction) *If* $\Gamma \vdash e : \tau, \mathrm{A}$ *and* k *is a continuation function of appropriate type then*

$$\mathcal{S}[\![e]\!] \mathbin{@} k \to^* e : k$$
$$\mathrm{E}[\mathcal{S}_v[\![e]\!]] = e : \mathrm{E} \text{ (} \textit{if } e \textit{ is trivial}\text{)}$$

Proof. Again, the proof is by induction on the structure of e with the $\mathcal{S}[\![\cdot]\!]$ case taken after the Ψ case for trivial expressions.

The $\mathrm{E}[\mathcal{S}_v[\![\cdot]\!]] = \cdot : \mathrm{E}$ **case:** There are four cases covering all trivial expressions:

- If e is a value or an identifier then $e : \mathrm{E} = \mathrm{E}[\mathcal{S}_v[\![e]\!]]$ by definition of $e : \mathrm{E}$.
- If $e = (e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}$ (e_1 not a value) then

$$\begin{aligned}(e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} &= e_1 : \mathrm{E}[[\,] \mathbin{@} \mathcal{S}_v[\![e_2]\!]] && \text{(def. } e : \mathrm{E}\text{)}\\ &= \mathrm{E}[\mathcal{S}_v[\![e_1]\!] \mathbin{@} \mathcal{S}_v[\![e_2]\!]] && \text{(I.H.)}\\ &= \mathrm{E}\big[\mathcal{S}_v[\![(e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}]\!]\big] && \text{(def. } \Psi\text{)}\end{aligned}$$

- If $e = (v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}$ (e_2 not a value) then

$$\begin{aligned}(v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} &= e_2 : \mathrm{E}[\mathcal{S}_v[\![v_1]\!] \mathbin{@} [\,]] && \text{(def. } e : \mathrm{E}\text{)}\\ &= \mathrm{E}[\mathcal{S}_v[\![v_1]\!] \mathbin{@} \mathcal{S}_v[\![e_2]\!]] && \text{(I.H.)}\\ &= \mathrm{E}\big[\mathcal{S}_v[\![(v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}]\!]\big] && \text{(def. } \Psi\text{)}\end{aligned}$$

- If $e = (v_1 \ @^{\mathrm{T}} \ v_2)^{\mathrm{T}}$ then

$$\begin{aligned}(v_1 \ @^{\mathrm{T}} \ v_2)^{\mathrm{T}} : \mathrm{E} &= \mathrm{E}[\mathcal{S}_v[\![v_1]\!] \ @ \ \mathcal{S}_v[\![v_2]\!]] \quad \text{(I.H.)} \\ &= \mathrm{E}\big[\mathcal{S}_v[\![(v_1 \ @^{\mathrm{T}} \ v_2)^{\mathrm{T}}]\!]\big] \ \text{(def. } \Psi)\end{aligned}$$

This accounts for all trivial expressions.

The $\mathcal{S}[\![\cdot]\!] \ @ \ k \to^* \cdot : k$ case: There is one sub-case for each non-trivial expression, and one case for all trivial expressions:

- If e is trivial then $\mathcal{S}[\![e]\!] \ @ \ k = k \ @ \ \mathcal{S}_v[\![e]\!] = e : [k \ @ \ [\,]] = e : k$ from the above cases and the definition of $e : k$.
- If $e = (e_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}$ (e_1 not a value) then

$$\begin{aligned}&\mathcal{S}[\![(e_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}]\!] \ @ \ k \\ &\to \ \mathcal{S}[\![e_1]\!] \ @ \ (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) && \text{(def. } \mathcal{S}[\![\cdot]\!]) \\ &\to^* \ e_1 : (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) && \text{(I.H.)} \\ &= \ (e_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}} : k && \text{(def. } e : k)\end{aligned}$$

- If $e = (v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}$ (e_2 not a value) then

$$\begin{aligned}&\mathcal{S}[\![(v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}]\!] \ @ \ k \\ &\to \ \mathcal{S}[\![v_1]\!] \ @ \ (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) && \text{(def. } \mathcal{S}[\![\cdot]\!]) \\ &\to \ (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) \ @ \ \mathcal{S}_v[\![v_1]\!] && \text{(def. } \mathcal{S}[\![v]\!]) \\ &\to \ \mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathcal{S}_v[\![v_1]\!] \ @ \ \mathrm{v}' \ @ \ k) \\ &\to^* \ e_2 : \lambda \mathrm{v}'.\mathcal{S}_v[\![v_1]\!] \ @ \ \mathrm{v}' \ @ \ k && \text{(I.H.)} \\ &= \ (v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}} : k && \text{(def. } e : k)\end{aligned}$$

- If $e = (v_1 \ @^{\mathrm{N}} \ v_2)^{\mathrm{N}}$ then the proof is similar to the previous case except two values need to be applied to continuations instead of just one.
- If $e = (e_1 \ @^{\mathrm{T}} \ e_2)^{\mathrm{N}}$ then the proofs are similar to the ones for $e = (e_1 \ @^{\mathrm{T}} \ e_2)^{\mathrm{N}}$ except that the innermost application is $k \ @ \ (\mathrm{v} \ @ \ \mathrm{v}')$ instead of $(\mathrm{v} \ @ \ \mathrm{v}') \ @ \ k$.
- If $e = \textsc{callcc} \ \mathrm{x}.e_1$ then $\mathcal{S}[\![\textsc{callcc} \ \mathrm{x}.e_1]\!] \ @ \ k \to (\lambda \mathrm{x}.\mathcal{S}[\![e_1]\!] \ @ \ k) \ @ \ k = \textsc{callcc} \ \mathrm{x}.e_1 : k$.
- If $e = \textsc{throw} \ e_1 \ e_2$ the proofs are similar to the ones for application.

□

Lemma 4.6 (Simulation) *If* $\mathrm{E}[e] \to \mathrm{E}'[e']$ *is one of the reduction rules for the annotated language, then*

$$\mathrm{E}[e] : \mathrm{id} \to^* \mathrm{E}'[e'] : \mathrm{id}$$

and if the reduction is not of a THROW *expression, then the* $\rightarrow^*$ *is actually one or more steps.*

Proof. Counting annotations, there are five cases:

- If $e = (\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{N}}\ v)^{\mathrm{N}}$ then

$$
\begin{array}{lll}
& \mathrm{E}\big[(\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{N}}\ v)^{\mathrm{N}}\big] : \mathrm{id} & \\
= & (\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{N}}\ v)^{\mathrm{N}} : (\mathrm{E} : \mathrm{id}) & \text{(Prop. 4.3)} \\
= & \mathcal{S}_v[\![\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1]\!] \ @\ \mathcal{S}_v[\![v]\!] \ @\ (\mathrm{E} : \mathrm{id}) & \text{(def. } e : k\text{)} \\
= & \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}[\![e_1]\!] \ @\ \mathcal{S}_v[\![v]\!] \ @\ (\mathrm{E} : \mathrm{id}) & \text{(def. } \Psi\text{)} \\
\rightarrow & \mathcal{S}[\![e_1]\!]\left[\mathcal{S}_v[\![\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1]\!]/\mathrm{f}\right]\left[\mathcal{S}_v[\![v]\!]/\mathrm{x}\right] \ @\ (\mathrm{E} : \mathrm{id}) & \\
= & \mathcal{S}[\![e_1\left[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right]]\!] \ @\ (\mathrm{E} : \mathrm{id}) & \text{(Lemma 4.4)} \\
\rightarrow^* & e_1\left[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right] : (\mathrm{E} : \mathrm{id}) & \text{(Lemma 4.5)} \\
\rightarrow^* & \mathrm{E}\big[e_1\left[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right]\big] : \mathrm{id} & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}}$ then we know that e_1 is trivial, since otherwise the function would be annotated N, and $\mathrm{E} : k$ is a continuation since E has no trivial inner sub-contexts.

$$
\begin{array}{lll}
& \mathrm{E}\big[(\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}}\big] : \mathrm{id} & \\
= & (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}} : (\mathrm{E} : \mathrm{id}) & \text{(Prop. 4.3)} \\
= & (\mathrm{E} : \mathrm{id}) \ @\ (\mathcal{S}_v[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!] \ @\ \mathcal{S}_v[\![v]\!]) & \text{(def. } e : k\text{)} \\
= & (\mathrm{E} : \mathrm{id}) \ @\ (\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_v[\![e_1]\!] \ @\ \mathcal{S}_v[\![v]\!]) & \text{(def. } \Psi\text{)} \\
\rightarrow & (\mathrm{E} : \mathrm{id}) \ @\ \mathcal{S}_v[\![e_1]\!]\left[\mathcal{S}_v[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!]/\mathrm{f}\right]\left[\mathcal{S}_v[\![v]\!]/\mathrm{x}\right] & \\
= & (\mathrm{E} : \mathrm{id}) \ @\ \mathcal{S}_v[\![e_1\left[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right]]\!] & \text{(Lemma 4.4)} \\
\rightarrow^* & e_1\left[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right] : [(\mathrm{E} : \mathrm{id}) \ @\ [\ ]] & \text{(Lemma 4.5, } e_1 \text{ trivial)} \\
= & e_1\left[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right] : (\mathrm{E} : \mathrm{id}) & \text{(def. } e : k\text{)} \\
\rightarrow^* & \mathrm{E}\big[e_1\left[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\right]\left[v/\mathrm{x}\right]\big] : \mathrm{id} & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = (e_1\ @^{\mathrm{T}}\ e_2)^{\mathrm{T}}$ then either $\mathrm{E} : k$ is a context or a continuation. If it is a continuation the proof proceeds just as the previous case. If it is a context

then

$$
\begin{array}{lll}
& \mathrm{E}\big[(\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{T}}\big] : \mathrm{id} & \\
= & (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{T}} : (\mathrm{E} : \mathrm{id}) & \text{(Prop. 4.3)} \\
= & (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_v[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!] \mathbin{@} \mathcal{S}_v[\![v]\!]\big] & \text{(def. } e : k\text{)} \\
= & (\mathrm{E} : \mathrm{id})[\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_v[\![e_1]\!] \mathbin{@} \mathcal{S}_v[\![v]\!]] & \text{(def. } \Psi\text{)} \\
\to & (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_v[\![e_1]\!]\ \big[\mathcal{S}_v[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!]/\mathrm{f}\big]\ [\mathcal{S}_v[\![v]\!]/\mathrm{x}]\big] & \\
= & (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_v[\![e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]]\!]\big] & \text{(Lemma 4.4)} \\
\to^* & e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) & \text{(Lemma 4.5, } e_1 \text{ trivial)} \\
\to^* & \mathrm{E}\big[e_1\ \big[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]\big] : \mathrm{id} & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = \textsc{callcc}\ \mathrm{x}.e_1$ then

$$
\begin{array}{lll}
& \mathrm{E}[\textsc{callcc}\ \mathrm{x}.e_1] : \mathrm{id} & \\
= & \textsc{callcc}\ \mathrm{x}.e_1 : (\mathrm{E} : \mathrm{id}) & \text{(Prop. 4.3)} \\
= & (\lambda \mathrm{x}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\mathrm{E} : \mathrm{id})) \mathbin{@} (\mathrm{E} : \mathrm{id}) & \text{(def. } e : k\text{)} \\
\to & \mathcal{S}[\![e_1]\!] \mathbin{@} (\mathrm{E} : \mathrm{id})\,[(\mathrm{E} : \mathrm{id})/\mathrm{x}] & \\
= & \mathcal{S}[\![e_1]\!]\,[(\mathrm{E} : \mathrm{id})/\mathrm{x}] \mathbin{@} (\mathrm{E} : \mathrm{id}) & (\mathrm{E} : k \text{ is closed}) \\
= & \mathcal{S}[\![e_1]\!]\,[\mathcal{S}_v[\![\langle \mathrm{E} \rangle]\!]/\mathrm{x}] \mathbin{@} (\mathrm{E} : \mathrm{id}) & \text{(def. } \mathcal{S}_v[\![\langle \mathrm{E} \rangle]\!]\text{)} \\
= & \mathcal{S}[\![e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}]]\!] \mathbin{@} (\mathrm{E} : \mathrm{id}) & \text{(Lemma 4.4)} \\
\to^* & e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) & \text{(Lemma 4.5)} \\
\to^* & \mathrm{E}[e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}]] : \mathrm{id} & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = \textsc{throw}\ \langle \mathrm{E}' \rangle\ v$ then

$$
\begin{array}{lll}
\mathrm{E}[\textsc{throw}\ \langle \mathrm{E}' \rangle\ v] : \mathrm{id} & = & \mathcal{S}_v[\![\langle \mathrm{E}' \rangle]\!] \mathbin{@} \mathcal{S}_v[\![v]\!] \quad \text{(def. } e : k\text{)} \\
& = & (\mathrm{E}' : \mathrm{id}) \mathbin{@} \mathcal{S}_v[\![v]\!] \quad \text{(def. } \mathcal{S}_v[\![\langle \mathrm{E} \rangle]\!]\text{)} \\
& = & v : [(\ @\ \mathrm{E}' : \mathrm{id})[\ ]] \quad \text{(def. } v : \mathrm{E}\text{)} \\
& = & v : (\mathrm{E}' : \mathrm{id}) \qquad \text{(def. } (e)^{\mathrm{T}} : k\text{)} \\
& \to^* & \mathrm{E}'[v] : \mathrm{id} \qquad \text{(Prop. 4.3)}
\end{array}
$$

In all cases except THROW, there is at least one reduction step.

□

4.3 Proof of correctness

To prove the correctness of the selective CPS transformation, we use the simulation lemma in two ways.

Proof. The proof of $e \to^* \mathrm{c} \implies \mathcal{S}[\![e]\!] \ @ \ \mathrm{id} \to^* \mathrm{c}$ follows directly from the lemma 4.5 and repeated use of lemma 4.6. Assume $e \to^* \mathrm{c}$.

$$
\begin{array}{rll}
\mathcal{S}[\![e]\!] \ @ \ \mathrm{id} & \to^* \ e : \mathrm{id} & \text{(Lemma 4.5)} \\
& \to^* \ \mathrm{c} : \mathrm{id} & \text{(Lemma 4.6, repeated)} \\
& = \ \mathrm{c} : [\mathrm{id} \ @ \ [\,]] & \text{(def. } (e)^{\mathrm{T}} : k) \\
& = \ \mathrm{id} \ @ \ \mathrm{c} & \text{(def. c : E)} \\
& \to \ \mathrm{c} &
\end{array}
$$

.

The other direction of correctness, $\mathcal{S}[\![e]\!] \ @ \ \mathrm{id} \to^* \mathrm{c} \implies e \to^* \mathrm{c}$, is shown by contraposition. Assuming that for no c does $e \to^* \mathrm{c}$, that is, e diverges, allows us to show that the same holds for $\mathcal{S}[\![e]\!]$.

The proof that transformation preserves divergence also follow from Lemmas 4.5 and 4.6. Since $\mathcal{S}[\![e]\!] \ @ \ \mathrm{id} \to^* e : \mathrm{id}$ it suffices to show that $e : \mathrm{id}$ has an arbitrary long reduction sequence.

Assume that e diverges. We show that for any n there exists an m such that if $e \to^m e_1$ then $(e : \mathrm{id}) \to^* (e_1 : \mathrm{id})$ in n or more reduction steps.

This is proven by induction on n. The base case ($n = 0$) is trivial. For the induction case ($n + 1$) look at the n case. There exists m such that $e \to^m e_1$ and $e : \mathrm{id} \to^* e_1 : \mathrm{id}$. Look at the reduction sequence from e_1.

- If the first reduction step ($e_1 \to e_2$) is not the reduction of a THROW expression, then $e_1 : \mathrm{id} \to^+ e_2 : \mathrm{id}$, and $m + 1$ gives us our $n + 1$ or longer reduction sequence of $e : \mathrm{id}$.
- If the first reduction step ($e_1 \to e_2$) is of a THROW expression, then $e_1 : \mathrm{id} \to^* e_2 : \mathrm{id}$. In that case we look at the next step in the same way. Either we find a reduction that is not a THROW, and we get the m needed for the proof, or there is nothing but reductions of THROW expressions in the infinite reduction sequence of e_1.

  There can not be an infinite sequence of reductions of THROW expressions, since reducing a THROW expression necessarily reduces the size of the entire program. A substitution into a context corresponds to the application of a linear function, and it reduces the size of the expression if one counts it as, e.g., number of distinct subexpressions or number of THROW-expressions.

That means that $e : \mathrm{id}$ has an infinite reduction sequence. □

5 Conclusion

We have proven the correctness of a selective CPS transformation based on an effect analysis. Similar proofs can be made for other λ-encodings and computational effects (e.g., with monads), where the immediate choice would be the effect of non-termination. That is the effect that is encoded by the traditional CPS transformation of languages with no other effects, and if one has an annotation of such a program, marking terminating (effect-free) expressions to keep in direct style, then the method works just as well.

5.1 Perspectives

Danvy and Hatcliff's CPS transformation after strictness analysis [6] generalizes the call-by-name and the call-by-value CPS transformations. The same authors' CPS transformation after totality analysis [7] generalizes the call-by-name CPS transformation and the identity transformation. In the same manner, the present work generalizes the call-by-value CPS transformation and the identity transformation, and proves this generalization correct.

Danvy and Filinski introduced the one-pass CPS-transformation [5] that removes the administrative reductions from the result by performing them at transformation time. This optimization can be applied to the selective CPS-transformation presented here as well. A proof of the correctness of the one-pass CPS-transformation also using Plotkin's colon translation exists [8]. We expect that the methods used for proving correctness of the selective- and the one-pass CPS transformations are orthogonal, and can easily be combined.

The selective CPS transformation presented here is based on an effect analysis and should generalize to other computational effects than control, e.g., state or I/O. The proof will not carry over to other effects, since it relies on the choice of λ-encoding of the effect primitives, but we expect that the structure of the proof can be preserved.

The approach taken is "Curry-style" in the sense that we have given a language and its operational meaning, and only after the fact we have associated types and effect annotation to the untyped terms. A "Church-style" approach, such as Filinski's [10,11], would have defined the language with explicit types and effect annotation, so that only well-typed, consistently annotated programs are given a semantics.

5.2 Future work

It is possible to prove results similar to the present ones for other choices of effects and combinations of effects. A sensible choice would be a monadic effect of state and control, since it is sufficient to implement all other choices of layered monads [11]. A proof similar to the present one for both state and control effects would be a logical next step.

Acknowledgments:
The method of extending the colon translation to selective CPS transformation was originally developed in cooperation with Junk-taek Kim and Kwangkeun Yi from KAIST in Korea, and with Olivier Danvy from BRICS in Denmark. The present work would not have been possible without their inspiration. Thanks are also due to Andrzej Filinski and to the anonymous referees for their comments.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[2] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[3] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[4] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 209–220, Montréal, Canada, September 2000. ACM Press.

[5] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[6] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[7] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[8] Olivier Danvy and Lasse R. Nielsen. A higher-order colon translation. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 78–91, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-33.

[9] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD

thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[10] Andrzej Filinski. Representing monads. In Boehm [2], pages 446–457.

[11] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[12] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[13] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [2], pages 458–471.

[14] Jung-taek Kim and Kwangkeun Yi. Interconnecting Between CPS Terms and Non-CPS Terms. In Sabry [20].

[15] Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In Greg Morrisett, editor, *Record of the 1998 ACM SIGPLAN Workshop on ML and its Applications*, Baltimore, Maryland, September 1998. Also appears as BRICS technical report RS-98-15.

[16] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[17] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[18] John Reppy. Local CPS conversion in a direct-style compiler. In Sabry [20].

[19] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[20] Amr Sabry, editor. *Proceedings of the Third ACM SIGPLAN Workshop on Continuations CW'01*, number 545 in Technical Report, Computer Science Department, Indiana University, Bloomington, Indiana, December 2000.

[21] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

MFPS 17 Preliminary Version

Semantics for Algebraic Operations

Gordon Plotkin and John Power [1]

Laboratory for the Foundations of Computer Science
University of Edinburgh
King's Buildings
Edinburgh EH9 3JZ
SCOTLAND

Abstract

Given a category C with finite products and a strong monad T on C, we investigate axioms under which an ObC-indexed family of operations of the form $\alpha_x : (Tx)^n \longrightarrow Tx$ provides a definitive semantics for algebraic operations added to the computational λ-calculus. We recall a definition for which we have elsewhere given adequacy results for both big and small step operational semantics, and we show that it is equivalent to a range of other possible natural definitions of algebraic operation. We outline examples and non-examples and we show that our definition is equivalent to one for call-by-name languages with effects too.

1 Introduction

Eugenio Moggi, in [6,8], introduced the idea of giving a unified category theoretic semantics for computational effects such as nondeterminism, probabilistic nondeterminism, side-effects, and exceptions, by modelling each of them uniformly in the Kleisli category for an appropriate strong monad on a base category C with finite products. He supported that construction by developing the computational λ-calculus or λ_c-calculus, for which it provides a sound and complete class of models. The computational λ-calculus is essentially the same as the simply typed λ-calculus except for the essential fact of making a careful systematic distinction between computations and values. However, it does not contain operations, and operations are essential to any programming language. So here, in beginning to address that issue, we provide a unified semantics for algebraic operations, supported by equivalence theorems to indicate definitiveness of the axioms.

[1] This work is supported by EPSRC grant GR/L89532: Notions of computability for general datatypes.

We distinguish here between algebraic operations and arbitrary operations. The former are, in a sense we shall make precise, a natural generalisation, from Set to an arbitrary category C with finite products, of the usual operations of universal algebra. The key point is that the operations

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

are parametrically natural in the Kleisli category for a strong monad T on C, as made precise in Definition 2.1: in that case, we say that the monad T supports the operations; the leading class of examples has T being generated by the operations subject to equations accompanying them. Examples of such operations are those for nondeterminism and probabilistic nondeterminism, and for raising exceptions. A non-example is given by an operation for handling exceptions.

In a companion paper [11], we have given the above definition, given a syntactic counterpart in terms of the computational λ-calculus, and proved adequacy results for small and big-step operational semantics. But such results alone leave some scope for a precise choice of appropriate semantic axioms. So in this paper, we prove a range of equivalence results, which we believe provide strong evidence for a specific choice of axioms, namely those for parametric naturality in the Kleisli category as mentioned above. Our most profound result is essentially about a generalisation of the correspondence between finitary monads and Lawvere theories from Set to a category with finite products C and a strong monad T on C: this result characterises algebraic operations as generic effects. The generality of our analysis is somewhat greater than in the study of enriched Lawvere theories in [12]: the latter require C to be locally finitely presentable as a closed category, which is not true of all our leading examples.

Moggi gave a semantic formulation of a notion of operation in [7], with an analysis based on his computational metalanguage, but he only required naturality of the operations in C, and we know of no way to provide operational semantics in such generality. Our various characterisation results do not seem to extend to such generality either. Evident further work is to consider how other operations such as those for handling exceptions should be modelled. That might involve going beyond monads, as Moggi has suggested to us; one possibility is in the direction of dyads [13].

We formulate our paper in terms of a strong monad T on a category with finite products C. We could equally formulate it in terms of closed $Freyd$-categories in the spirit of [1], which provides a leading example for us in its analysis of finite nondeterminism.

The paper is organised as follows. In Section 2, we recall the definition of algebraic operation given in [11] and we exhibit some simple reformulations of it. In Section 3, we give direct equivalent versions of these statements in terms of enrichment under the assumption that C is closed. In Section 4, we give a more substantial reformulation of the notion in terms of operations on

homs, both when C is closed and more generally when C is not closed. In Section 5, we give what we regard as the most profound result of the paper, which is a formulation in terms of generic effects, generalising a study of Lawvere theories. Finally, in Section 6, we characterise algebraic operations in terms of operations on the category T-Alg, as this gives an indication of how to incorporate call-by-name languages with computational effects into the picture. And we give conclusions and an outline of possible future directions in Section 7.

2 Algebraic operations and simple equivalents

In this section, we give the definition of algebraic operation as we made it in [11]. In that paper, we gave the definition and a syntactic counterpart in terms of the computational λ-calculus, and we proved adequacy results for small and big-step operational semantics for the latter in terms of the former. Those results did not isolate definitive axioms for the notion of algebraic operation. So in this section, we start with a few straightforward equivalence results on which we shall build later.

We assume we have a category C with finite products together with a strong monad $< T, \eta, \mu, st >$ on C with Kleisli exponentials, i.e., such that for all objects x and z of C, the functor $C_T(-\times x, z) : C^{op} \longrightarrow Set$ is representable. We do not take C to be closed in general: we shall need to assume it for some later results, but we specifically do not want to assume it in general, and we do not require it for any of the results of this section.

Given a map $f : y \times x \longrightarrow Tz$ in C, we denote the parametrised lifting of f, i.e., the composite

$$y \times Tx \xrightarrow{st} T(y \times x) \xrightarrow{Tf} T^2 z \xrightarrow{\mu_z} Tz$$

by $f^\dagger : y \times Tx \longrightarrow Tz$.

Definition 2.1 An *algebraic operation* is an ObC-indexed family of maps

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

such that for every map $f : y \times x \longrightarrow Tz$ in C, the diagram

$$\begin{array}{ccc} y \times (Tx)^n & \xrightarrow{\langle f^\dagger \cdot (y \times \pi_i)\rangle_{i=1}^n} & (Tz)^n \\ {\scriptstyle y \times \alpha_x}\downarrow & & \downarrow{\scriptstyle \alpha_z} \\ y \times Tx & \xrightarrow[f^\dagger]{} & Tz \end{array}$$

commutes.

For some examples of algebraic operations, for $C = Set$, let T be the nonempty finite power-set monad with binary choice operations [9,1]; alternatively, let T be the monad for probabilistic nondeterminism with probabilistic choice operations [2,3]; or take T to be the monad for printing with printing operations [10]. Observe the non-commutativity in the latter example. One can, of course, generalise from Set to categories such as that of ω-cpo's, for instance considering the various power-domains together with binary choice operators. One can also consider combinations of these, for instance to model internal and external choice operations. Several of these examples are treated in detail in [11].

There are several equivalent formulations of the coherence condition of the definition. Decomposing it in a maximal way, we have

Proposition 2.2 *An ObC-indexed family of maps*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

is an algebraic operation if and only if

(i) *α is natural in C*

(ii) *α respects st in the sense that*

$$\begin{array}{ccc} y \times (Tx)^n & \xrightarrow{\langle st \cdot (y \times \pi_i)\rangle_{i=1}^n} & (T(y \times x))^n \\ {\scriptstyle y \times \alpha_x}\downarrow & & \downarrow{\scriptstyle \alpha_{y \times x}} \\ y \times Tx & \xrightarrow[st]{} & T(y \times x) \end{array}$$

commutes

(iii) *α respects μ in the sense that*

$$\begin{array}{ccc} (T^2x)^n & \xrightarrow{\mu_x^n} & (Tx)^n \\ {\scriptstyle \alpha_{Tx}}\downarrow & & \downarrow{\scriptstyle \alpha_x} \\ T^2x & \xrightarrow[\mu_x]{} & Tx \end{array}$$

commutes.

Proof. It is immediately clear from our formulation of the definition and the proposition that the conditions of the proposition imply the coherence requirement of the definition. For the converse, to prove naturality in C, put $y = 1$ and, given a map $g : x \longrightarrow z$ in C, compose it with η_z and apply the coherence condition of the definition. For coherence with respect to st, take

$f : y \times x \longrightarrow Tz$ to be $\eta_{y \times x}$. And for coherence with respect to μ, put $y = 1$ and take f to be id_{Tx}. □

There are other interesting decompositions of the coherence condition of the definition too. In the above, we have taken T to be an endo-functor on C. But one often also writes T for the right adjoint to the canonical functor $J : C \longrightarrow C_T$ as the behaviour of the right adjoint on objects is given precisely by the behaviour of T on objects. So with this overloading of notation, we have functors $(T-)^n : C_T \longrightarrow C$ and $T : C_T \longrightarrow C$, we can speak of natural transformations between them, and we have the following proposition.

Proposition 2.3 *An ObC-indexed family of maps*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

is an algebraic operation if and only if α is natural in C_T and α respects st.

In another direction, as we shall investigate further below, it is sometimes convenient to separate the μ part of the coherence condition from the rest of it. We can do that with the following somewhat technical result.

Proposition 2.4 *An ObC-indexed family*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

forms an algebraic operation if and only if α respects μ and, for every map $f : y \times x \longrightarrow z$ in C, the diagram

$$\begin{array}{ccccc} y \times (Tx)^n & \xrightarrow{\langle st \cdot (y \times \pi_i) \rangle_{i=1}^n} & (T(y \times x))^n & \xrightarrow{(Tf)^n} & (Tz)^n \\ \downarrow{\scriptstyle y \times \alpha_x} & & & & \downarrow{\scriptstyle \alpha_z} \\ y \times Tx & \xrightarrow[st]{} & T(y \times x) & \xrightarrow[Tf]{} & Tz \end{array}$$

commutes.

3 Equivalent formulations if C is closed

For our more profound results, it seems best first to assume that C is closed, explain the results in those terms, and later to drop the closedness condition and explain how to reformulate the results without essential change. So for the results in this section, we shall assume C is closed.

Let the closed structure of C be denoted by $[-,-]$. Given a monad $< T, \eta, \mu >$ on C, to give a strength for T is equivalent to giving an enrichment of T in C: given a strength, one has an enrichment

$$T_{x,y} : [x, y] \longrightarrow [Tx, Ty]$$

given by the transpose of

$$[x,y] \times Tx \xrightarrow{st} T([x,y] \times x) \xrightarrow{Tev} Ty$$

and given an enrichment of T, one has a strength given by the transpose of

$$x \longrightarrow [y, x \times y] \xrightarrow{T_{y,x \times y}} [Ty, T(x \times y)]$$

It is routine to verify that the axioms for a strength are equivalent to the axioms for an enrichment. So, given a strong monad $< T, \eta, \mu, st >$ on C, the monad T is enriched in C, and so is the functor $(-)^n : C \longrightarrow C$.

The category C_T also canonically acquires an enrichment in C, i.e, the homset $C_T(x,y)$ of C_T lifts to a homobject of C: the object $[x, Ty]$ of C acts as a homobject, applying the functor $C(1,-) : C \longrightarrow Set$ to it giving the homset $C_T(x,y)$; composition

$$C_T(y,z) \times C_T(x,y) \longrightarrow C_T(x,z)$$

lifts to a map in C

$$[y,Tz] \times [x,Ty] \longrightarrow [x,Tz]$$

determined by taking a transpose and applying evaluation maps twice and each of the strength and the multiplication once; and identities and the axioms for a category lift too.

The canonical functor $J : C \longrightarrow C_T$ becomes a C-enriched functor with a C-enriched right adjoint. The main advantage of the closedness condition for us is that it allows us to dispense with the parametrisation of the naturality, or equivalently with the coherence with respect to the strength, as follows.

Proposition 3.1 *If C is closed, an ObC-indexed family*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

forms an algebraic operation if and only if

$$\begin{array}{ccc} [x,Tz] & \xrightarrow{(-)^n \cdot [Tx,\mu_z] \cdot T_{x,Tz}} & [(Tx)^n, (Tz)^n] \\ \Big\downarrow{\scriptstyle [Tx,\mu_z] \cdot T_{x,Tz}} & & \Big\downarrow{\scriptstyle [(Tx)^n, \alpha_z]} \\ [Tx,Tz] & \xrightarrow[[\alpha_x, Tz]]{} & [(Tx)^n, Tz] \end{array}$$

commutes.

The left-hand vertical map in the diagram here is exactly the behaviour of the C-enriched right adjoint $T : C_T \longrightarrow C$ to the canonical C-enriched functor $J : C \longrightarrow C_T$ on homs, and the top horizontal map is exactly the behaviour of the C-enriched functor $(T-)^n : C_T \longrightarrow C$ on homs. So the

coherence condition in the proposition is precisely the statement that α forms a C-enriched natural transformation from the C-enriched functor $(T-)^n : C_T \longrightarrow C$ to the C-enriched functor $T : C_T \longrightarrow C$.

Proof. Given a map $f : y \times x \longrightarrow Tz$ in C, the transpose of the map gives a map from y to $[x, Tz]$. Precomposing the coherence condition here with that map, then transposing both sides, one obtains the coherence condition of the definition. For the converse, given a map $g : y \longrightarrow [x, Tz]$, taking its transpose, using the coherence condition of the definition, and transposing back again, shows that the above square precomposed with g commutes. So by the Yoneda lemma, we are done. □

The same argument can be used to give a further characterisation of the notion of algebraic operation if C is closed by modifying Proposition 2.4. This yields

Proposition 3.2 *If C is closed, an ObC-indexed family*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

forms an algebraic operation if and only if α respects μ and

$$\begin{array}{ccc} [x,z] & \xrightarrow{(-)^n \cdot T_{x,z}} & [(Tx)^n, (Tz)^n] \\ {\scriptstyle T_{x,z}}\downarrow & & \downarrow{\scriptstyle [(Tx)^n, \alpha_z]} \\ [Tx, Tz] & \xrightarrow[[\alpha_x, Tz]]{} & [(Tx)^n, Tz] \end{array}$$

commutes.

This proposition says that if C is closed, an algebraic operation is exactly a C-enriched natural transformation from the C-enriched functor $(T-)^n : C \longrightarrow C$ to the C-enriched functor $T : C \longrightarrow C$ that is coherent with respect to μ.

4 Algebraic operations as operations on homs

In our various formulations of the notion of algebraic operation so far, we have always had an ObC-indexed family

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

and considered equivalent conditions on it under which it might be called an algebraic operation. In computing, this amounts to considering an operator on expressions. But there is another approach in which arrows of the category C_T may be seen as primitive, regarding them as programs. This was the underlying idea of the reformulation [1] of the semantics for finite nondeterminism of [9]. So we should like to reformulate the notion of algebraic operation in

these terms. Proposition 3.1 allows us to do that. In order to explain the reason for the coherence conditions, we shall start by expressing the result assuming C is closed; after which we shall drop the closedness assumption and see how the result can be re-expressed using parametrised naturality.

We first need to explain an enriched version of the Yoneda lemma as in [4]. If D is a small C-enriched category, then D^{op} may also be seen as a C-enriched category. We do not assume C is complete here, but if we did, then we would have a C-enriched functor category $[D^{op}, C]$ and a C-enriched Yoneda embedding

$$Y_D : D \longrightarrow [D^{op}, C]$$

The C-enriched Yoneda embedding Y_D is a C-enriched functor and it is fully faithful in the strong sense that the map

$$D(x, y) \longrightarrow [D^{op}, C](D(-, x), D(-, y))$$

is an isomorphism in the category C: see [4] for all the details. It follows by applying the functor $C(1, -) : C \longrightarrow Set$ that this induces a bijection from the set of maps from x to y in D to the set of C-enriched natural transformations from the C-enriched functor $D(-, x) : D^{op} \longrightarrow C$ to the C-enriched functor $D(-, y) : D^{op} \longrightarrow C$.

This is the result we need, except that we do not want to assume that C is complete, and the C-enriched categories of interest to us are of the form C_T, so in general are not small. These are not major problems although they go a little beyond the scope of the standard formulation of enriched category theory in [4]: one can embed C into a larger universe C' just as one can embed Set into a larger universe Set' when necessary, and the required mathematics for the enriched analysis appears in [4]. We still have what can reasonably be called a Yoneda embedding of D into $[D^{op}, C]$, with both categories regarded as C'-enriched rather than C-enriched, and it is still fully faithful as a C'-enriched functor. However, we can formulate the result we need more directly without reference to C' simply by stating a restricted form of the enriched Yoneda lemma: letting $Fun_C(D^{op}, C)$ denote the (possibly large) category of C-enriched functors from D^{op} to C, the underlying ordinary functor

$$D \longrightarrow Fun_C(D^{op}, C)$$

of the Yoneda embedding is fully faithful.

We use this latter statement both here and in the following section. Now for our main result of this section under the assumption that C is closed.

Theorem 4.1 *If C is closed, to give an algebraic operation is equivalent to giving an $ObC^{op} \times ObC$ family of maps*

$$a_{y,x} : [y, Tx]^n \longrightarrow [y, Tx]$$

that is C-natural in y as an object of C^{op} and C-natural in x as an object of

C_T, i.e., such that

$$\begin{array}{ccc} [y,Tx]^n \times [y',y] & \xrightarrow{\langle comp \cdot (\pi_i \times [y',y])\rangle_{i=1}^n} & [y',Tx]^n \\ \downarrow{\scriptstyle a_{y,x} \times [y',y]} & & \downarrow{\scriptstyle a_{y',x}} \\ [y,Tx] \times [y',y] & \xrightarrow[comp]{} & [y',Tx] \end{array}$$

and

$$\begin{array}{ccc} [x,Tz] \times [y,Tx]^n & \xrightarrow{\langle comp_K \cdot ([x,Tz] \times \pi_i)\rangle_{i=1}^n} & [y,Tz]^n \\ \downarrow{\scriptstyle [x,Tz] \times a_{y,x}} & & \downarrow{\scriptstyle a_{y,z}} \\ [x,Tz] \times [y,Tx] & \xrightarrow[comp_K]{} & [y,Tz] \end{array}$$

commute, where comp is the C-enriched composition of C and $comp_K$ is C-enriched Kleisli composition.

Proof. First observe that $[y,Tx]^n$ is isomorphic to $[y,(Tx)^n]$. Now, it follows from our C-enriched version of the Yoneda lemma that to give the data together with the first axiom of the proposition is equivalent to giving an ObC-indexed family

$$\alpha : (Tx)^n \longrightarrow Tx$$

By a further application of our C-enriched version of the Yoneda lemma, it follows that the second condition of the proposition is equivalent to the coherence condition of Proposition 3.1. □

As mentioned earlier, we can still state essentially this result even without the condition that C be closed. There are two reasons for this. First, for the paper, we have assumed the existence of Kleisli exponentials, as are essential in order to model λ-terms. But most of the examples of the closed structure of C we have used above are of the form $[y,Tx]$, which could equally be expressed as the Kleisli exponential $y \Rightarrow x$. The Kleisli exponential routinely extends to a functor

$$- \Rightarrow - : C_T^{op} \times C_T \longrightarrow C$$

Second, in the above, we made one use of a construct of the form $[y',y]$ with no T protecting the second object. But we can replace that by using the ordinary Yoneda lemma to express the first condition of the theorem in terms of maps $f : w \times y' \longrightarrow y$.

Summarising, we have

Corollary 4.2 *To give an algebraic operation is equivalent to giving an* $ObC^{op} \times ObC$ *family of maps*

$$a_{y,x} : (y \Rightarrow x)^n \longrightarrow (y \Rightarrow x)$$

in C*, such that for every map* $f : w \times y' \longrightarrow y$ *in* C*, the diagram*

$$\begin{array}{ccc} (y \Rightarrow x)^n \times w \times y' & \xrightarrow{(f \Rightarrow x)^n \times w \times y'} & ((w \times y') \Rightarrow x)^n \times w \times y' \\ \downarrow{\scriptstyle a_{y,x} \times f} & & \downarrow{\scriptstyle ev \cdot (a_{w \times y', x} \times w \times y')} \\ (y \Rightarrow x) \times y & \xrightarrow[ev]{} & x \end{array}$$

commutes, and the diagram

$$\begin{array}{ccc} (x \Rightarrow z) \times (y \Rightarrow x)^n & \xrightarrow{\langle comp_K \cdot ((x \Rightarrow z) \times \pi_i) \rangle_{i=1}^n} & (y \Rightarrow z)^n \\ \downarrow{\scriptstyle (x \Rightarrow z) \times a_{y,x}} & & \downarrow{\scriptstyle a_{y,z}} \\ (x \Rightarrow z) \times (y \Rightarrow x) & \xrightarrow[comp_K]{} & (y \Rightarrow z) \end{array}$$

commutes, where $comp_K$ *is the canonical internalisation of Kleisli composition.*

5 Algebraic operations as generic effects

In this section, we apply our formulation of the C-enriched Yoneda lemma to characterise algebraic operations in entirely different terms again as maps in C_T, i.e., in terms of generic effects. Observe that if C has an n-fold coproduct $\mathbf{n}$ of 1, the functor $(T-)^n : C_T \longrightarrow C$ is isomorphic to the functor $\mathbf{n} \Rightarrow - : C_T \longrightarrow C$. If C is closed, the functor $\mathbf{n} \Rightarrow -$ enriches canonically to a C-enriched functor, and that C-enriched functor is precisely the representable C-functor $C_T(\mathbf{n}, -) : C_T \longrightarrow C$, where C_T is regarded as a C-enriched category. So by Proposition 3.1 together with our C-enriched version of the Yoneda lemma, we immediately have

Theorem 5.1 *If* C *is closed, the* C*-enriched Yoneda embedding induces a bijection between maps* $1 \longrightarrow \mathbf{n}$ *in* C_T *and algebraic operations*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

This result is essentially just an instance of an enriched version of the identification of maps in a Lawvere theory with operations of the Lawvere theory. Observe that it follows that there is no mathematical reason to restrict

attention to algebraic operations of arity n for a natural number n. We could just as well speak, in this setting, of algebraic operations of the form

$$\alpha_x : (\mathbf{a} \Rightarrow -) \longrightarrow (\mathbf{b} \Rightarrow -)$$

for any objects $\mathbf{a}$ and $\mathbf{b}$ of C. So for instance, we could include an account of infinitary operations as one might use to model operations involved with state. For specific choices of C such as $C = Poset$, one could consider more exotic arities such as that given by Sierpinski space.

Once again, by use of parametrisation, we can avoid the closedness assumption on C here, yielding the stronger statement

Theorem 5.2 *Functoriality of $- \Rightarrow - : C_T^{op} \times C_T \longrightarrow C$ in its first variable induces a bijection from the set of maps $1 \longrightarrow \mathbf{n}$ in C_T to the set of algebraic operations*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

We regard this as the most profound result of the paper. This result shows that to give an algebraic operation is equivalent to giving a generic effect, i.e., a constant of type the arity of the operation. For example, to give a binary nondeterministic operator for a strong monad T is equivalent to giving a constant of type 2, and to give equations to accompany the operator is equivalent to giving equations to be satisfied by the constant. The leading example here has T being the non-empty finite powerset monad or a powerdomain. Given a nondeterministic operator $\vee$, the constant is given by $true \vee false$, and given a constant c, the operator is given by $M \vee N =$ if c then M else N. There are precisely three non-empty finite subsets of the two element set, and accordingly, there are precisely three algebraic operations on the non-empty finite powerset monad, and they are given by the two projections and choice.

The connection of this result with enriched Lawvere theories [12] is as follows. If C is locally finitely presentable as a closed category, one can define a notion of finitary C-enriched monad on C and a notion of C-enriched Lawvere theory, and prove that the two are equivalent, generalising the usual equivalence in the case that $C = Set$. Given a finitary C-enriched monad T, the corresponding C-enriched Lawvere theory is given by the full sub-C-category of C_T determined by the finitely presentable objects. These include all finite coproducts of 1. So our results here exactly relate maps in the Lawvere theory with algebraic operations, generalising Lawvere's original idea. Of course, in this paper, we do not assume the finiteness assumptions on either the category C or the monad T, but our result here is essentially the same.

Theorem 5.2 extends with little fuss to the situation of finitely presentable objects $\mathbf{a}$ and $\mathbf{b}$; one just requires a suitable refinement of the construct $(T-)^n$ to account for $\mathbf{a}$ and $\mathbf{b}$ being objects of C rather than finite numbers. This follows readily by inspection of the work of [12], and, in a special case, it seems to provide an account of some of the operations associated with state,

as suggested to us by Moggi.

6 Algebraic operations and the category of algebras

Finally, in this section, we characterise the notion of algebraic operation in terms of the category of algebras T-Alg. The co-Kleisli category of the comonad on T-Alg induced by the monad T is used to model call-by-name languages with effects, so this formulation gives us an indication of how to generalise our analysis to call-by-name computation or perhaps to some combination of call-by-value and call-by-name, cf [5].

If C is closed and has equalisers, generalising Lawvere, the results of the previous section can equally be formulated as equivalences between algebraic operations and operations

$$\alpha_{(A,a)} : U(A,a)^n \longrightarrow U(A,a)$$

natural in (A,a), where $U : T\text{-}Alg \longrightarrow C$ is the C-enriched forgetful functor: equalisers are needed in C in order to give an enrichment of T-Alg in C. We prove the result by use of our C-enriched version of the Yoneda lemma again, together with the observation that the canonical C-enriched functor $I : C_T \longrightarrow T\text{-}Alg$ is fully faithful. Formally, the result is

Theorem 6.1 *If C is closed and has equalisers, the C-enriched Yoneda embedding induces a bijection between maps $1 \longrightarrow \mathbf{n}$ in C_T and C-enriched natural transformations*

$$\alpha : (U-)^n \longrightarrow U - .$$

Combining this with Theorem 5.1, we have

Corollary 6.2 *If C is closed and has equalisers, to give an algebraic operation*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

is equivalent to giving a C-enriched natural transformation

$$\alpha : (U-)^n \longrightarrow U.$$

One can also give a parametrised version of this result if C is neither closed nor complete along the lines for C_T as in the previous section. It yields

Theorem 6.3 *To give an algeberaic operation*

$$\alpha_x : (Tx)^n \longrightarrow Tx$$

is equivalent to giving an $Ob(T\text{-}Alg)$-indexed family of maps

$$\alpha_{(A,a)} : U(A,a)^n \longrightarrow U(A,a)$$

such that, for each map

$$f : x \times U(A,a) \longrightarrow U(B,b)$$

commutativity of

$$\begin{array}{ccc} x \times TA & \xrightarrow{x \times Tf} & x \times TB \\ {\scriptstyle x \times a}\downarrow & & \downarrow{\scriptstyle x \times b} \\ x \times A & \xrightarrow[x \times f]{} & x \times B \end{array}$$

implies commutativity of

$$\begin{array}{ccc} x \times U(A,a)^n & \xrightarrow{\langle f \cdot (x \times \pi_i)\rangle_{i=1}^n} & U(B,b)^n \\ {\scriptstyle x \times \alpha_{(A,a)}}\downarrow & & \downarrow{\scriptstyle \alpha_{(B,b)}} \\ x \times U(A,a) & \xrightarrow[f]{} & U(B,b) \end{array}$$

7 Conclusions and Further Work

For some final comments, we note that little attention has been paid in the literature to the parametrised naturality condition on the notion of algebraic operation that we have used heavily here. And none of the main results of [11] used it, although they did require naturality in C_T. So it is natural to ask why that is the case.

For the latter point, in [11], we addressed ourselves almost exclusively to closed terms, and that meant that parametrised naturality of algebraic operations did not arise as we did not have any parameter.

Regarding why parametrised naturality does not seem to have been addressed much in the past, observe that for $C = Set$, every monad has a unique strength, so parametrised naturality of α is equivalent to ordinary naturality of α. More generally, if the functor $C(1, -) : C \longrightarrow Set$ is faithful, i.e., if 1 is a generator in C, then parametrised naturality is again equivalent to ordinary naturality of α. That is true for categories such as $Poset$ and that of ω-cpo's, which have been the leading examples of categories studied in this regard. The reason we have a distinction is because we have not assumed that 1 is a generator, allowing us to include examples such as toposes or Cat for example.

Of course, in future, we hope to address other operations that are not algebraic, such as one for handling exceptions. It seems unlikely that the approach of this paper extends directly. Eugenio Moggi has recommended we look beyond monads. We should also like to extend and integrate this work with work addressing other aspects of giving a unified account of computa-

tional effects. We note here especially Paul Levy's work [5] which can be used to give accounts of both call-by-value and call-by-name in the same setting, and work on modularity [13], which might also help with other computational effects.

References

[1] Anderson, S.O., and A. J. Power, *A Representable Approach to Finite Nondeterminism*, Theoret. Comput. Sci. **177** (1997) 3–25.

[2] Jones, C., "Probabilistic Non-Determinism," Ph.D. Thesis, University of Edinburgh, Report ECS-LFCS-90-105, 1990.

[3] Jones, C., and G. D. Plotkin, *A Probabilistic Powerdomain of Evaluations*, Proc. LICS **4** (1989) 186–195.

[4] Kelly, G.M., "Basic Concepts of Enriched Category Theory," Cambridge: Cambridge University Press, 1982.

[5] Levy, P.B., *Call-by-Push-Value: A Subsuming Paradigm*, "Proc. TLCA **99**" Lecture Notes in Computer Science **1581** 228–242.

[6] Moggi, E., *Computational lambda-calculus and monads*, Proc. LICS **89** (1989) 14–23.

[7] Moggi, E., *An abstract view of programming languages*, University of Edinburgh, Report ECS-LFCS-90-113, 1989.

[8] Moggi, E., *Notions of computation and monads*, Inf. and Comp. **93** (1991) 55–92.

[9] Plotkin, G.D., *A Powerdomain Construction*, SIAM J. Comput. **5** (1976) 452–487.

[10] Plotkin, G.D., "Domains," (http://www.dcs.ed.ac.uk/home/gdp/), 1983.

[11] Plotkin, G.D., and A. J. Power, *Adequacy for Algebraic Effects*, Proc. FOSSACS **2001** (to appear).

[12] Power, A.J., *Enriched Lawvere Theories*, Theory and Applications of Categories (2000) 83–93.

[13] Power, A.J., and E. P. Robinson, *Modularity and Dyads*, "Proc. MFPS **15**" Electronic Notes in Thoeret. Comp. Sci. **20**, 1999.

An Algebraic Foundation for Graph-based Diagrams in Computing

John Power [1,3] and Konstantinos Tourlas [2,4]

Division of Informatics
The University of Edinburgh
Edinburgh EH9 3JZ
United Kingdom

Abstract

We develop an algebraic foundation for some of the graph-based structures underlying a variety of popular diagrammatic notations for the specification, modelling and programming of computing systems. Using hypergraphs and higraphs as leading examples, a locally ordered category $Graph(C)$ of graphs in a locally ordered category C is defined and endowed with symmetric monoidal closed structure. Two other operations on higraphs and variants, selected for relevance to computing applications, are generalised in this setting.

1 Introduction

Recent years have witnessed a rapid, ongoing popularisation of diagrammatic notations in the specification, modelling and programming of computing systems. Most notable among them are Statecharts [4], a notation for modelling reactive systems, and the Unified Modelling Language (UML) [10], a family of diagrammatic notations for object-based modelling. Invariably, underlying such complex diagrams is some notion of graph, upon which labels and other linguistic or visual annotations are added according to application-specific needs (see e.g. [10,9,3] for a variety of examples).

Beyond ordinary graphs, the two leading examples studied here are hypergraphs and higraphs [5]. The latter underlie a number of sophisticated diagrammatic formalisms including, most prominently, Statecharts, the state

[1] This work has been done with the support of EPSRC grant GR/M56333 and a British Council grant, and the COE budget of STA Japan.
[2] Support EPSRC grant GR/N12480 and of the COE budget of STA Japan is gratefully acknowledged.
[3] Email: ajp@dcs.ed.ac.uk
[4] Email: kxt@dcs.ed.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

diagrams of UML, and the domain-specific language Argos [8] for programming reactive systems. Higraphs allow for vitally concise, economical representations of complex state-transition systems, such as those underlying realistic reactive systems, by drastically reducing the number of edges required to specify the transition relation. This is achieved by replacing a number of transitions having, say, a common target state with a *single* transition having the same target but with source a new "super-state" containing all the source states of the original transitions. The resulting reduction in complexity is of the order of n^2, where n is the number of states.

We begin our analysis by observing that graphs, hypergraphs and higraphs are all instances of the same structure, that of a graph in a category C, with C being respectively *Set*, *Rel* and *Poset*. Other variants are also considered. The case of higraphs is motivated and studied extensively and concretely in the draft paper [13]. The latter assumes only elementary knowledge of category theory on the part of the reader, so as to be accessible to a wide audience of computer scientists who have immediate scientific and practical interest in higraphs and their applications in UML and Statecharts. In the present paper, Section 2 introduces our leading examples, followed by a definition in Section 2.4 of a category $Graph(C)$ of graphs in a locally ordered category C.

Underlying Statecharts is a binary operation which given Statecharts S and S' yields a third corresponding to the semantics of S and S' operating concurrently. We show how the same applies to higraphs and hypergraphs. Here we formulate this precisely and uniformly in algebraic terms by defining a symmetric monoidal closed structure on $Graph(C)$. We do so in Section 3. It is further shown that symmetric monoidal closed adjunction linking $Graph(C)$ to $Cat(C)$ exists when the latter category bears a generalisation of the "other" symmetric monoidal closed structure on Cat.

Hierarchies of edges in higraphs are exploited in practical applications to produce concise specifications of complex reactive systems. To understand the meaning of higher-level edges we introduce in Section 4 a completion operation on higraphs. This is shown to be an instance of the right adjoint to the inclusion of $Graph(C)$ into $Graph_{opl}(C)$, the latter having oplax natural transformations as arrows. A theorem stating conditions for the existence of such right adjoints is proved.

To support users in working with large, hierarchically structured diagrams representing complex systems, one requires effective mechanisms for re-organising, abstracting and filtering the information present in diagrams [9]. The leading example studied here is of a filtering operation on higraphs, introduced and motivated by Harel in [5] under the name of *zooming out*. We show in Section 5 how it generalises to graphs in non-trivially locally ordered categories.

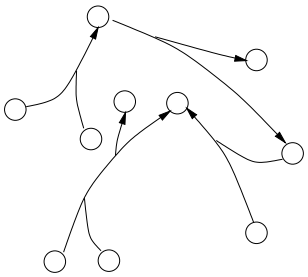


Fig. 1. A simple hypergraph.

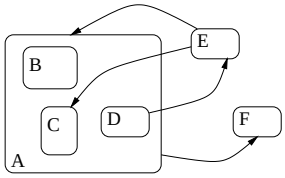


Fig. 2. A simple higraph.

2 Leading examples and main definition

We begin by recalling the standard definition of a (directed, multi-)*graph* as consisting of a set V of vertices, a set E of edges and two functions $s, t : E \longrightarrow V$ giving the source and target of each edge. That is, a graph is a pair of parallel arrows $s, t : E \longrightarrow V$ in the category *Set*.

2.1 Hypergraphs

Hypergraphs are a generalisation of graphs in which each edge may have sets of vertices as its source and target. The typical pictorial representation of this kind of directed hypergraph is illustrated in Figure 1.

Thus, a hypergraph consists of a set V of vertices, a set E of edges and two functions $s, t : E \longrightarrow 2^V$ giving sources and targets. Equivalently, s and t may be seen as relations from E to V, thus arriving at the following

Definition 2.1 *A hypergraph is a pair of parallel maps in the category Rel of (small) sets and relations.*

2.2 Higraphs

Higraph is a term coined-up by Harel[5] as short for *hierarchical graph*, but is often used to include several variants. The definitive feature of higraphs, common to all variants, is referred to as *depth*, meaning that nodes may be contained inside other nodes. Figure 2 illustrates the standard pictorial representation of a higraph consisting of six nodes and four edges, with the nodes labelled B, C and D being spatially contained within the node labelled A. It is therefore common, and we shall hereafter adhere to convention, to call the nodes of a higraph *blobs*, as an indication of their pictorial representation by

convex contours on the plane. For further details the reader is referred to [13].

The containment relation on blobs is captured by requiring poset structure on the set of blobs. The notion of higraph developed here extends this requirement to the set of edges:

Definition 2.2 *A higraph is a pair of parallel arrows $s, t : E \longrightarrow B$ in the category Poset.*

In practice, a higraph typically arises as a graph (B, E, s, t) together with a partial order $\leq_B$ on B. In that case, the poset structure on E may be taken to be the discrete one. However, other choices of orders on E are often useful, e.g. for encoding the conflict resolution schemes [6] adopted in Statecharts.

In most applications of higraphs, especially Statecharts, the intuitive understanding of en edge e is as *implying* the presence of "lower-level", *implicit* edges from all blobs contained in $s(e)$ to all blobs contained in $t(e)$. The point in general is that a multitude of edges is made implicit in a single, explicitly shown higher-level edge. In Statecharts, this device is employed for representing *interrupt transitions*, thus drastically reducing the number of edges required to specify the transition relation among the states of the represented transition system.

2.3 Combinations and variants

To deal with realistic diagrams, one may additionally wish to combine features found in different notions of graph, e.g. to allow edges in higraphs to have multiple sources and targets, as is indeed allowed in some Statecharts. The resulting notion of graph, a combination of simple higraphs (as defined above) and hypergraphs, could be approached by considering the category of posets and relations between their underlying sets. The category *BSup* of posets with all binary sups (and sup-preserving monotone maps) gives a better model of depth in Statecharts. One may also consider graphs in the category ω-Cpo of ω-complete partial orders.

2.4 Graphs in locally ordered categories

Each of our leading examples of "notions of graph" has been cast in terms of a pair of parallel maps in a suitable category C. Another, less obvious commonality among our examples is that C has been a *locally ordered* category, i.e. a category enriched in the cartesian closed category *Poset* of posets, a fact of which substantial use will be made later. (The category *Set* is locally ordered in a trivial sense: each hom-object is a discrete poset.) Generalising from our situation one has:

Definition 2.3 *Let C be a locally ordered category. Let $Graph(C)$ denote the locally ordered category of graphs in C, that is the functor category $[\cdot \rightrightarrows \cdot, C]$ where the category $\cdot \rightrightarrows \cdot$ consists of two objects and two non-identity maps as shown.* □

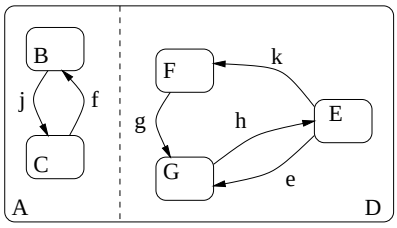


Fig. 3. A simple Statechart

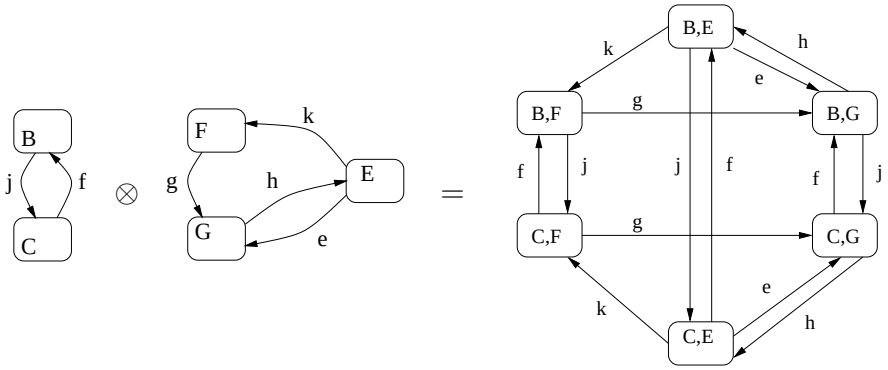


Fig. 4. Operation underlying the Statechart of Fig. 3

So, an object of $Graph(C)$ consists of a pair of objects E and V of C, together with a pair of maps $s, t : E \longrightarrow V$ in C. An arrow of $Graph(C)$ from $(E, V, s, t : E \longrightarrow V)$ to $(E', V', s', t' : E' \longrightarrow V')$ consists of maps $f_E : E \longrightarrow E'$ and $f_V : V \longrightarrow V'$ such that $f_V s = s' f_E$ and $f_V t = t' f_E$. The local order of $Graph(C)$ is generated by that of C, i.e., $(f_E, f_V) \leq (g_E, g_V)$ if $f_E \leq g_E$ and $f_V \leq g_V$.

3 A symmetric monoidal closed structure on $Graph(C)$

We now proceed to study some extra structure on $Graph(C)$, for well-behaved C. Our motivation arises from the application of higraphs in Statecharts. Specifications of complex reactive systems directly in terms of transition systems become impractical to visualise owing to the large number of states involved. Statecharts deal with this problem by allowing the modelling of reactive systems directly in terms of their identifiable concurrent subsystems:

Example 3.1 *Consider the Statechart in Figure 3 representing two subsystems A and D operating concurrently. Assuming an interleaving model of concurrency, as is the case with Statecharts, the meaning of this picture is captured precisely by the operation where the resulting transition system is exactly the intended behaviour of the complete system.* □

A consequence of our results in this section is that the above operation, which in [5] is referred to as "a sort of product of automata", generalises

smoothly to higraphs. This is an essential step in pinpointing the precise mathematical structures underpinning the semantics of Statecharts. For, more generally, the specifications of the subsystems A and D in Figure 3 typically bear higraph structure.

So for our next main result, we observe that, generalising the situation for $C = Set$ in Example 3.1, here not requiring local order structure on C, we have

Theorem 3.2 *For any cartesian closed category C with finite coproducts, the category $Graph(C)$ has a symmetric monoidal structure given as follows: given $G = (E, V, s, t)$ and $G' = (E', V', s', t')$, the graph $G \otimes G'$ has vertex object $V \times V'$ and edge object $(E \times V') + (V \times E')$, with source and target maps evident. The unit of this symmetric monoidal structure is given by $V = 1$ and $E = 0$.*

Proof. That $\otimes$ is a bifunctor follows directly from the properties of the binary products and coproducts in C. The required isomorphisms are easily deduced from those associated with the symmetric monoidal structure induced on C by its cartesian structure, and the verification of the required coherence conditions is routine. □

Example 3.3 *On higraphs $\otimes$ yields a straightforward generalisation of the operation in Figure 4. Specifically $\chi \otimes \chi'$ contains an edge $\langle b_1, b' \rangle \to \langle b_2, b' \rangle$ for every edge $b_1 \to b_2$ in χ and blob b' in χ', and an edge $\langle b, b'_1 \rangle \to \langle b, b'_2 \rangle$ for every edge $b'_1 \to b'_2$ in χ' and blob b in χ. Containment is given by $\langle b_1, b'_1 \rangle \leq \langle b_2, b'_2 \rangle$ iff $b_1 \leq b_2$ and $b'_1 \leq b'_2$. In the case of hypergraphs, $H \otimes H'$ contains an edge $\{\langle x_1, x' \rangle, \ldots, \langle x_n, x' \rangle\} \to \{\langle y_1, x' \rangle, \ldots, \langle y_m, x' \rangle\}$ for each edge $\{x_1, \ldots, x_n\} \to \{y_1, \ldots, y_m\}$ in H and vertex x' in χ', and similarly for the edges in H'.*

Theorem 3.4 *For any cartesian closed category C with finite coproducts and finite limits, the symmetric monoidal structure on $Graph(C)$ given in Theorem 3.2 is closed.*

Proof. The exponential object $[G', G'']$ has object of vertices the domain of the equaliser of the two maps from $[V', V''] \times [E', E'']$ to $[E', V'] \times [E', V']$ given by $\langle [s', V'], [t', V'] \rangle \circ \pi_0$ and $\langle [E', s'], [E', t'] \rangle \circ \pi_1$ where π_0, π_1 are the projections from $[V', V''] \times [E', E'']$. The object of edges of $[G', G'']$ is the domain of the equaliser of the maps $\langle \pi_0 \circ q \circ \pi'_0, \pi_0 \circ q \circ \pi'_2 \rangle$ and $\langle [V', s''] \circ \pi'_1, [V', t''] \circ \pi'_1 \rangle$, both having domain $V \times [V', E''] \times V$ and codomain $[V', V''] \times [V', V'']$, where π'_i are the three projections out of $V \times [V', E''] \times V$. □

Notice, in particular, that the exponential in the category $Graph(C)$ with the tensor product defined in the theorem is particularly natural. The object of vertices represents all graph homomorphisms from G to G', and the object of edges represents all transformations between graph homomorphisms.

3.1 A symmetric monoidal closed adjunction

It is well known that one may define categories in any category C with finite limits, the usual category Cat being isomorphic to the category of models $Cat(Set)$ in Set of an appropriate finite limit sketch [1]. We shall write $Cat(C)$ for the category of categories in C, implicitly asserting C to have finite limits as required.

While it is well known that Cat is a cartesian closed category, it is far less well known that there is precisely one other symmetric monoidal closed structure on Cat [2,12]. We refer to the other one as the *other* symmetric monoidal closed structure on Cat, which may be outlined as follows:

- The exponential $A \longrightarrow B$ is given by the set of functors from A to B, with a morphism from g to h being the assignment of an arrow $\alpha_x : gx \longrightarrow hx$ to each object x of A. The composition is obvious. We shall call an arrow of $A \longrightarrow B$ a *transformation*.
- The tensor product may be described in terms of a universal property: it is the universal D for which one has, for each object x of A, a functor $h_x : B \longrightarrow D$ and for each object y of B, a functor $k_y : A \longrightarrow D$ such that $h_x y = k_y x$ for each (x, y). The unit of the tensor product is the unit category.

Explicitly, the tensor product $A \otimes B$ of A and B has as object set $ObA \times ObB$, and an arrow from (x, y) to (x', y') consists of a finite sequence of non-identity arrows, with alternate arrows forming a directed path in A, and the others forming a directed path in B. Composition is given by concatenation, then cancellation accorded by the composition of A and B. The symmetry is obvious.

It is routine to verify that if, in addition to having finite limits, C is cocomplete and cartesian closed, the other symmetric monoidal closed structure extends to $Cat(C)$. We are now in position to state our theorem relating $Cat(C)$ to $Graph(C)$:

Theorem 3.5 *For a cocomplete cartesian closed category C with finite limits, the forgetful functor $U : Cat(C) \longrightarrow Graph(C)$ is part of a symmetric monoidal closed adjunction with respect to the other tensor product on $Cat(C)$ and the above symmetric monoidal closed structure on $Graph(C)$.*

Proof. For a proof, consider the case that C is Set and simply internalise the argument there. □

Note that a corresponding result does not hold for the cartesian closed structures of $Cat(C)$ and $Graph(C)$ even in the case of $C = Set$, so we regard this result as strong evidence of the naturalness of this structure. Finally, in this vein, we observe

Theorem 3.6 *For cartesian closed C with finite coproducts, the forgetful functor from $Graph(C)$ to C is part of a symmetric monoidal closed adjunc-*

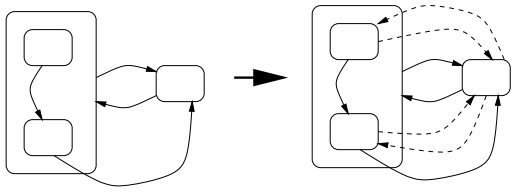


Fig. 5. Completion of a simple higraph, where the added edges are shown dashed.

tion with respect to the above symmetric monoidal structure on $Graph(C)$.

Proof. For a proof, consider the proof in the case of $C = Set$ and routinely internalise it to C. □

Again, even in the case of $C = Set$, a corresponding result does not hold in respect of the cartesian closed structure of $Graph(C)$ as the left adjoint does not preserve the unit, i.e., it does not send 1 to the terminal object of *Graph* as the latter has an edge.

4 A completion operation

A construction useful in understanding the semantics of higraphs and variants (for instance that involving the categories *BSup* or ω-Cpo) is to explicate all edges which are understood as being implicitly present in a higraph (recall the discussion near the end of Section 2.2). This "completion" operation is illustrated in Figure 5.

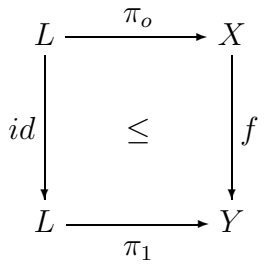
Definition 4.1 *Let* $\chi = s, t : E \longrightarrow B$ *be a higraph. The higraph* $T(\chi)$*, called the* completion *of* χ*, has blobs* B *and edges the subset of* $E \times (B \times B)$ *consisting of those pairs* $\langle e, \langle b, b' \rangle \rangle$ *such that* $b \leq_B s(e)$ *and* $b' \leq_B t(e)$*, partially ordered pointwise, with source and target given by projections.* □

Definition 4.2 *Given a locally ordered category* C*, we denote by* $Graph_{opl}(C)$ *the locally ordered category whose objects are graphs in* C *and whose arrows are oplax transformations, i.e. pairs* $(f_E : E \longrightarrow E', f_V : V \longrightarrow V')$ *such that* $f_V s \leq s' f_E$ *and* $f_V t \leq t' f_E$*, with local order structure induced by that of* C*.* □

To state our theorem, it is convenient to use a little of the theory of 2-categories, specifically some finite limits. A convenient account of such limits is [7]. In particular, we need to use the notion of an oplax limit of a map. So we recall it here.

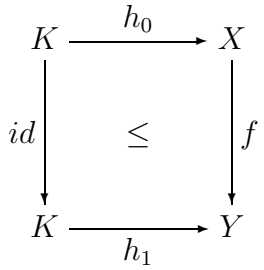
Definition 4.3 Given an arrow $f : X \longrightarrow Y$ in a locally ordered category C,

an *oplax limit* of f is given by a diagram of the form

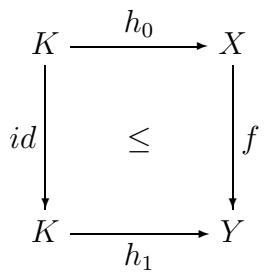


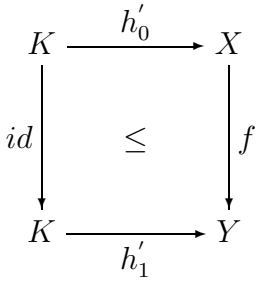
satisfying two properties:

- for any other diagram of the form



  there is a unique arrow $u : K \longrightarrow L$ such that $\pi_0 u = h_0$ and $\pi_1 u = h_1$, and
- (the two-dimensional property) for any two diagrams of the form





  with $h_0 \leq h_0'$ and $h_1 \leq h_1'$, it follows that $u \leq u'$.

□

Theorem 4.4 *If the locally ordered category C has finite limits, then the inclusion of $Graph(C)$ into $Graph_{opl}(C)$ has a right adjoint.*

Proof. Given a graph $G = (E, V, s, t)$, the right adjoint has vertex object given by V and object of edges given by the oplax limit of the map $\langle s, t\rangle : E \longrightarrow V \times V$. It is a routine exercise in 2-categories to prove that this construction yields a right adjoint. □

The 2-category theory expert will observe that we have only used pie-limits in C, which may become important in due course [11]. Perhaps a more familiar expression for the oplax limit used in the proof is in terms of a comma object in C from the identity map on $V \times V$ to the map $\langle s, t\rangle : E \longrightarrow V \times V$. If

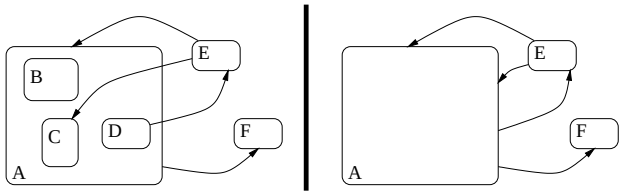


Fig. 6. Zooming out of a blob in a higraph

C were the locally ordered category *Poset*, then the right adjoint could be described explicitly by placing an edge from v to v' if there is an edge from a vertex greater than or equal to v to a vertex greater than or equal to v' in G. This matches exactly our explicit description of T in Definition 4.1.

Dually, if C has finite colimits, the inclusion of $Graph(C)$ into $Graph_{opl}(C)$ has a left adjoint.

5 Zooming out

We begin by recalling Harel's simple instance of a zooming operation on higraphs: the selection of a single blob and the subsequent removal from view of all blobs contained in it. An example is illustrated in the transition from the left to the right half of Figure 6.

To capture the notion of selecting a blob in a higraph we need the following:

Definition 5.1 *A* pointed higraph ψ *consists of an ordinary higraph* $\chi = s, t : E \longrightarrow B$ *together with a distinguished blob, given as a map* $1 \longrightarrow B$ *in Poset and called the* point *of* ψ. *The category* $\mathcal{H}_\star$ *has pointed higraphs as its objects and maps those ones which preserve points. Let* $\mathcal{H}_{\star,min}$ *be the full subcategory of* $\mathcal{H}_\star$ *consisting of all objects (pointed higraphs) in which the point is minimal wrt. the partial order on blobs; in other words, the point is an atomic blob. Let* I *be the full functor including* $\mathcal{H}_{\star,min}$ *into* $\mathcal{H}_\star$. □

Consider a pointed higraph ψ with $\chi = (s, t : E \longrightarrow B)$ and point, say, $p \in B$. The pointed higraph $Z(\psi)$, obtained by zooming out of the point in ψ, is determined by the following data:

- blobs: $B' = B \setminus \{b \mid b < p\}$ (ordered by the restriction to B' of the partial order on B);
- edges: E, with the source and target functions being $q \circ s$ and $q \circ t$ respectively, where $q : B \longrightarrow B'$ is the (obviously monotone) function mapping each $b \not< p$ in B to $b \in B'$ and each $b < p$ to $p \in B'$;
- point: p

One now has the following [13]:

Proposition 5.2 *The function* Z *extends to a functor from* $\mathcal{H}_\star$ *to* $\mathcal{H}_{\star,min}$ *which is left adjoint to the inclusion functor* I. □

This proposition will be shown an instance of Theorem 5.5 below. Gener-

alising the essential structure underlying our leading example one has:

Definition 5.3 *Given a locally ordered category C, denote by $Graph(C)_*$ the locally ordered category for which an object consists of a graph (E, V, s, t) in C together with a map $v : 1 \longrightarrow V$ in C. The maps are pairs of maps that strictly preserve the structure.* □

Definition 5.4 *Given a locally ordered category C, denote by $Graph(C)_{*min}$ the locally ordered full subcategory of $Graph(C)_*$ such that the point $v : 1 \longrightarrow V$ is a minimal element in the poset $C(1, V)$.* □

Theorem 5.5 *If C is a cocomplete locally ordered category, then the inclusion of $Graph(C)_{*min}$ in $Graph(C)_*$ has a left adjoint.*

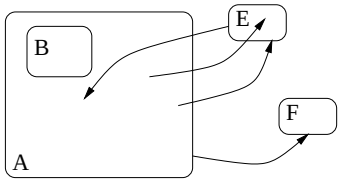
Proof. Given (E, V, s, t) and $v : 1 \longrightarrow V$, take the joint coequaliser of v with all of the elements of the poset $C(1, V)$ that are less than or equal to it. It is routine to verify that this gives the left adjoint. □

Example 5.6 *For graphs in BSup the theorem gives the expected generalisation of the zoom-out operation on graphs in Poset in the presence of the extra structure given by binary sups. However, zoom-outs do not generalise to graphs in Rel, or the category of posets and relations between their underlying sets, as the terminal object is the empty set (poset).*

6 Further work

Our aim is to develop, in an incremental and principled way, structures which bear sufficient detail to model realistic diagrammatic notations. Currently we are working towards providing such a model for a large class of Statecharts, which include features found in higraphs and hypergraphs. The work herein presented lays the abstract foundations for our approach, in which notions of graph and combinations thereof may be studied.

Another strand of our work is to study extensions to such notions of graph, as required to support users in performing specification and reasoning tasks with diagrams. For instance, a mild extension to higraphs was briefly introduced by Harel in [5], permitting edges to be "loosely" attached to nodes, the four possibilities being illustrated in



.

The rationale was to indicate transitions or relations between some as yet unspecified, or purposefully omitted (e.g. as the result of zooming out) parts of the represented system. For motivation and details the reader is referred to [13]. We conclude by noting that such graphs with "loose edges" can be added easily to our framework, provided that the locally ordered category C

has finite (pie) colimits, thereby allowing one to define tensors with the arrow poset.

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[2] F. Foltz, C.M. Kelly, and C. Lair. Algebraic categories with few monoidal biclosed structures or none. *Journal of Pure and Applied Algebra*, 17:171–177, 1980.

[3] Corin Gurr and Konstantinos Tourlas. Towards the principled design of software engineering diagrams. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 509–520. ACM, IEEE Computer Society, ACM Press, 2000.

[4] David Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3):231–275, 1987.

[5] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[6] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4), October 1996.

[7] G.M. Kelly. Elementary observations on 2-categorical limits. *Bull. Austral. Math. Soc.*, pages 301–317, 1989.

[8] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[9] Bonnie M. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*. MIT Press, 1993.

[10] Rob Pooley and Perdita Stevens. *Using UML*. Addison Wesley, 1999.

[11] A.J. Power and E.P. Robinson. A characterization of pie-limits. *Math. Proc. Cambridge Philos. Soc.*, 110:33–47, 1991.

[12] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Comp. Science*, 11, 1993.

[13] John Power and Konstantinos Tourlas. An algebraic foundation for higraphs. Submitted for publication, March 2001.

Comparing Control Constructs by Double-barrelled CPS Transforms

Hayo Thielecke

H.Thielecke@cs.bham.ac.uk
School of Computer Science
University of Birmingham
Birmingham
United Kingdom

Abstract

We investigate continuation-passing style transforms that pass two continuations. Altering a single variable in the translation of λ-abstraction gives rise to different control operators: first-class continuations; dynamic control; and (depending on a further choice of a variable) either the `return` statement of C; or Landin's **J**-operator. In each case there is an associated simple typing. For those constructs that allow upward continuations, the typing is classical, for the others it remains intuitionistic, giving a clean distinction independent of syntactic details.

1 Introduction

Control operators come in bewildering variety. Sometimes the same term is used for distinct constructs, as with `catch` in early Scheme or `throw` in Standard ML of New Jersey, which are very unlike the `catch` and `throw` in Lisp whose names they borrow. On the other hand, this Lisp `catch` is fundamentally similar to exceptions despite their dissimilar and much more ornate appearance.

Fortunately it is sometimes possible to glean some high-level "logical" view of a programming language construct by looking only at its type. Specifically for control operations, Griffin's discovery [3] that `call/cc` and related operators can be ascribed classical types gives us the fundamental distinction between languages that have such classical types and those that do not, even though they may still enjoy some form of control. This approach complements comparisons based on contextual equivalences [10,14].

Such a comparison would be difficult unless we blot out complication. In particular, exceptions are typically tied in with other, fairly complicated features of the language which are not relevant to control as such: in ML with the datatype mechanism, in Java with object-orientation. In order to

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

simplify, we first strip down control operators to the bare essentials of labelling and jumping, so that there are no longer any distracting syntactic differences between them. The grammar of our toy language is uniformly this:

$$M ::= x \mid \lambda x.M \mid MM \mid \texttt{here}\, M \mid \texttt{go}\, M.$$

The intended meaning of `here` is that it labels a "program point" or expression without actually naming any particular label—just uttering the demonstrative "here", as it were. Correspondingly, `go` jumps to a place specified by a `here`, without naming the "to" of a `goto`.

Despite the simplicity of the language, there is still scope for variation: not by adding bells and whistles to `here` and `go`, but by varying the meaning of λ-abstraction. Its impact can be seen quite clearly in the distinction between exceptions and first-class continuations. The difference between them is as much due to the meaning of λ-abstraction as due to the control operators themselves, since λ-abstraction determines what is statically put into a closure and what is passed dynamically. Readers familiar with, say, Scheme implementations will perhaps not be surprised about the impact of what becomes part of a closure. But the point of this paper is twofold:

- small variations in the meaning of λ completely change the meaning of our control operators;
- we can see these differences at an abstract, logical level, without delving into the innards of interpreters.

We give meaning to the λ-calculus enriched with `here` and `go` by means of continuations in Section 2, examining in Sections 3–5 how variations on λ-abstraction determine what kind of control operations `here` and `go` represent. For each of these variations we present a simple typing, which agrees with the transform (Section 6). We conclude by explaining the significance of these typings in terms of classical and intuitionistic logic (Section 7).

2 Double-barrelled CPS

Our starting point is a continuation-passing style (CPS) transform. This transform is double-barrelled in the sense that it always passes *two* continuations. Hence the clauses start with $\lambda kq.\ldots$ instead of $\lambda k.\ldots$. Other than that, this CPS transform is in fact a very mild variation on the usual call-by-value one [8]. As indicated by the $\boxed{?}$, we leave one variable, the extra continuation passed to the body of a λ-abstraction, unspecified.

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_? x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{?}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\texttt{here}\, M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\texttt{go}\, M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

The extra continuation may be seen as a jump continuation, in that its

manipulation accounts for the labelling and jumping. This is done symmetrically: `here` makes the jump continuation the same as the current one k, whereas `go` sets the current continuation of its argument to the jump continuation q. The clauses for variables and applications do not interact with the additional jump continuation: the former ignores it, while the latter merely distributes it into the operator, the operand and the function call.

Only in the clause for λ-abstraction do we face a design decision. Depending on which continuation (static s, dynamic d, or the return continuation r) we fill in for "?" in the clause for λ, there are three different flavours of λ-abstraction.

$$[\![\lambda_{\mathsf{s}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s})$$
$$[\![\lambda_{\mathsf{d}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d})$$
$$[\![\lambda_{\mathsf{r}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r})$$

The lambdas are subscripted to distinguish them, and the box around the last variable is meant to highlight that this is the crucial difference between the transforms. Formally there is also a fourth possibility, the outer continuation k, but this seems less meaningful and would not fit into simple typing.

For all choices of λ, the operation `go` is always a jump to a place specified by a `here`. For example, for any M, the term `here` $((\lambda x.M)(\texttt{go}\, N))$ should be equivalent to N, as the `go` jumps past the M. But in more involved examples than this, there may be different choices *where* `go` can go to among several occurrences of `here`. In particular, if s is passed as the second continuation argument to M in the transform of $\lambda x.M$, then a `go` in M will refer to the `here` that was in scope at the point of definition (unless there is an intervening `here`, just as one binding of a variable x can shadow another). By contrast, if d is passed to M in $\lambda x.M$, then the `here` that is in scope at the point of definition is forgotten; instead `go` in M will refer to the `here` that is in scope at the point of call when $\lambda x.M$ is applied to an argument. In fact, depending upon the choice of variable in the clause for λ as above, `here` and `go` give rise to different control operations:

- first-class continuations like those given by `call/cc` in Scheme [4];
- dynamic control in the sense of Lisp, and typeable in a way reminiscent of checked exceptions;
- a `return`-operation, which can be refined into the **J**-operator invented by Landin in 1965 and ancestral to `call/cc` [4,6,7,13].

We examine these constructs in turn, giving a simple type system in each case. An unusual feature of these type judgements is that, because we have two continuations, there are two types in the succedent on the right of the turnstile, as in

$$\Gamma \vdash M : A, B.$$

The first type on the right accounts for the case that the term returns a value; it corresponds to the current continuation. The second type accounts for the

Fig. 1. Typing for static here and go

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{s}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathtt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} M : B, C}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : A \to B, C \qquad \Gamma \vdash_{\mathsf{s}} N : A, C}{\Gamma \vdash_{\mathsf{s}} MN : B, C}$$

jump continuation. In logical terms, the comma on the right may be read as a disjunction. It makes a big difference whether this disjunction is classical or intuitionistic. That is our main criterion of comparing and contrasting the control constructs.

3 First-class continuations

The first choice of which continuation to pass to the body of a function is arguably the cleanest. Passing the static continuation s gives control the same static binding as ordinary λ-calculus variables. In the static case, the transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{\mathsf{s}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!] r \boxed{s}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\, M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

We type our source language with here and go as in Figure 1.

In logical terms, both here and go are a combined right weakening and contraction. By themselves, weakening and contraction do not amount to much; but it is the combination with the rule for $\to$-introduction that makes the calculus "classical", in the sense that there are terms whose types are propositions of classical, but not of intuitionistic, minimal logic.

To see how $\to$-introduction gives classical types, consider λ-abstracting over go.

$$\frac{x : A \vdash_{\mathsf{s}} \mathtt{go}\, x : B, A}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\mathtt{go}\, x : A \to B, A}$$

If we read the comma as "or", and $A \to B$ for arbitrary B as "not A", then this judgement asserts the classical excluded middle, "not A or A". We build on the classical type of $\lambda_{\mathsf{s}} x.\mathtt{go}\, x$ for another canonical example: Scheme's

`call-with-current-continuation` (`call/cc` for short) operator [4]. It is syntactic sugar in terms of static `here` and `go`:

$$\texttt{call/cc} = \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\,x))).$$

As one would expect [3], the type of `call/cc` is Peirce's law "if not A implies A, then A". We derive the judgement

$$\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\,x))) : ((A \to B) \to A) \to A, C$$

as follows. Let Γ be the context $f : (A \to B) \to A$. Then we derive:

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{s}} f : (A \to B) \to A, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{s}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{s}} \texttt{go}\,x : B, A}}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\texttt{go}\,x : A \to B, A}}{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{s}} (f\,(\lambda_{\mathsf{s}} x.\texttt{go}\,x)) : A, A}{\Gamma \vdash_{\mathsf{s}} \texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\,x)) : A, C}}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\,x))) : ((A \to B) \to A) \to A, C}}$$

As another example, let Γ be any context, and assume we have $\Gamma \vdash_{\mathsf{s}} M : A, B$. Right exchange is admissible in that we can also derive $\Gamma \vdash_{\mathsf{s}} M' : B, A$ for some M'.

In the typing of `call/cc`, a `go` is (at least potentially, depending on f) exported from its enclosing `here`. Conversely, in the derivation of right exchange, a `go` is imported into a `here` from without. What makes everything work is static binding.

4 Dynamic control

Next we consider the dynamic version of `here` and `go`. The word "dynamic" is used here in the sense of dynamic binding and dynamic control in Lisp. Another way of phrasing it is that with a dynamic semantics, the `here` that is in scope at the point where a function is *called* will be used, as opposed to the `here` that was in scope at the point where the function was *defined*—the latter being used for the static semantics.

In the dynamic case, the transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.kx \\
[\![\lambda_{\mathsf{d}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\texttt{here}\,M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\texttt{go}\,M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

In this transform, the jump continuation acts as a handler continuation; since it is passed as an extra argument on each call, the dynamically enclosing handler is chosen. Hence under the dynamic semantics, `here` and `go` become a stripped-down version of Lisp's `catch` and `throw` with only a single catch

Fig. 2. Typing for dynamic here and go

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{d}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \mathtt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{d}} M : B, C}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.M : A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} N : A, C}{\Gamma \vdash_{\mathsf{d}} MN : B, C}$$

tag. These `catch` and `throw` operation are themselves a no-frills version of exceptions with only identity handlers. We can think of `here` and `go` as a special case of these more elaborate constructs:

$$\mathtt{here}\, M \equiv \mathtt{(catch\ 'e}\ \ M\mathtt{)}$$

$$\mathtt{go}\, M \equiv \mathtt{(throw\ 'e}\ \ M\mathtt{)}$$

Because the additional continuation is administered dynamically, we cannot fit it into our simple typing without annotating the function type. So for dynamic control, we write the function type as $A \to B \vee C$. Syntactically, this should be read as a single operator with the three arguments in mixfix. We regard the type system as a variant of intuitionistic logic in which $\to$ and $\vee$ always have to be introduced or eliminated together.

This annotated arrow can be seen as an idealization of the Java `throws` clause in method definitions, in that $A \to B \vee C$ could be written as

$$B\mathtt{(}A\mathtt{)\ throws}\ C$$

in a more Java-like syntax. A function of type $A \to B \vee C$ may throw things of type C, so it may only be called inside a `here` with the same type. Our typing for the language with dynamic `here` and `go` is presented in Figure 2.

We do not attempt to idealize the ML way of typing exceptions because ML uses a universal type `exn` for exceptions, in effect allowing a carefully delimited area of untypedness into the language. The typing of ML exceptions is therefore much less informative than that of checked exceptions.

Note that `here` and `go` are still the same weakening and contraction hybrid as in the static setting. But here their significance is a completely different one because the $\to$-introduction is coupled with a sort of $\vee$-introduction. To see the difference, recall that in the static setting λ-abstracting over a `go` reifies the jump continuation and thereby, at the type level, gives rise to classical disjunction. This is not possible with the version of λ that gives `go` the dynamic semantics. Consider the following inference:

$$\frac{x : A \vdash_{\mathsf{d}} \mathtt{go}\, x : B, A}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\mathtt{go}\, x : A \to B \vee A, C}$$

The C-accepting continuation at the point of definition is not accessible to the $\mathtt{go}$ inside the λ_{d}. Instead, the $\mathtt{go}$ refers only to the A-accepting continuation that will be available at the point of call. Far from the excluded middle, the type of $\lambda_{\mathsf{d}} x.\mathtt{go}\, x$ is thus "A implies A or B; or anything".

In the same vein, as a further illustration how fundamentally different the dynamic `here` and `go` are from the static variety, we revisit the term that, in the static setting, gave rise to `call/cc` with its classical type:

$$\lambda f.\mathtt{here}\,(f\,(\lambda x.\mathtt{go}\, x)).$$

Now in the dynamic case, we can only derive the intuitionistic formula

$$((A \to B \vee A) \to A \vee A) \to A \vee C$$

as the type of this term.

Let Γ be the context $f : (A \to B \vee A) \to A \vee A$. Then we have:

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{d}} f : (A \to B \vee A) \to A \vee A, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{d}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{d}} \mathtt{go}\, x : B, A}}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\mathtt{go}\, x : A \to B \vee A, A}}{\dfrac{\dfrac{\Gamma \vdash_{\mathsf{d}} (f\,(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, A}{\Gamma \vdash_{\mathsf{d}} \mathtt{here}\,(f\,(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, C}}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} f.\mathtt{here}\,(f\,(\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : ((A \to B \vee A) \to A \vee A) \to A \vee C, D}}$$

5 Return continuation

Our last choice is passing the return continuation as the extra continuation to the body of a λ-abstraction. So the CPS transform is this:

$$\begin{aligned}
[\![x]\!] &= \lambda kq.qx \\
[\![\lambda_{\mathsf{r}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\
[\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] &= \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\, M]\!] &= \lambda kq.[\![M]\!]qq
\end{aligned}$$

This transform grants λ_{r} the additional role of a continuation binder. The original operator for this purpose, `here`, is rendered redundant, since $\mathtt{here}\, M$ is now equivalent to $(\lambda_{\mathsf{r}} x.M)(\lambda_{\mathsf{r}} y.y)$ where x is not free in M. At first sight, binding continuations seems an unusual job for a λ; but it becomes less so if we think of `go` as the `return` statement of C or Java.

5.1 Non-first class `return`

Because the enclosing λ determines which continuation `go` jumps to with its argument, the `go`-operator has the same effect as a `return` statement. The

Fig. 3. Typing for go as a return-operation

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{r}} x : A, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{r}} N : A, C}{\Gamma \vdash_{\mathsf{r}} MN : B, C}$$

type of extra continuation assumed by go needs to agree with the return type of the nearest enclosing λ:

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C}$$

The whole type system for the calculus with λ_{r} is in Figure 3.

The agreement between go and the enclosing λ_{r} is comparable with the typing in C, where the expression featuring in a `return` statement must have the return type declared by the enclosing function. For instance, M needs to have type `int` in the definition:

$$\texttt{int f()\{...return } M\texttt{;...\}}$$

With λ_{r}, the special form go cannot be made into a first-class function. If we try to λ-abstract over $\mathtt{go}\, x$ by writing $\lambda_{\mathsf{r}} x.\mathtt{go}\, x$ then go will refer to that λ_{r}.

The failure of λ_{r} to give first-class returning can be seen logically as follows. In order for λ_{r} to be introduced, both types on the right have to be the same:

$$\frac{x : A \vdash_{\mathsf{r}} \mathtt{go}\, x : A, A}{\vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.\mathtt{go}\, x : A \to A, C}$$

Rather than the classical "not A or A" this asserts merely the intuitionistic "A implies A; or anything".

One has a similar situation in Gnu C, which has both the `return` statement and nested functions, without the ability to refer to the return address of another function. If we admit go as a first-class function, it becomes a much more powerful form of control, Landin's **JI**-operator.

5.2 The **JI**-operator

Keeping the meaning of λ_{r} as a continuation binder, we now consider a control operator **JI** that always refers to the statically enclosing λ_{r}, but which, unlike the special form go, is a first-class expression, so that we can pass the return continuation to some other function f by writing $f(\mathbf{JI})$. The CPS of this operator is this:

$$[\![\mathbf{JI}]\!] = \lambda ks.k(\lambda xrd.\boxed{s}x)$$

That is almost, but not quite, the same as if we tried to define **JI** as $\lambda_{\mathsf{r}} x.\mathtt{go}\, x$:

Fig. 4. Typing for **JI**

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{j}} x : A, C} \qquad \frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{j}} M : B, B}{\Gamma \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{j}} N : A, C}{\Gamma \vdash_{\mathsf{j}} MN : B, C}$$

$$\begin{aligned}[\![\mathbf{JI}]\!] &= [\![\lambda_{\mathsf{r}} x.\mathtt{go}\, x]\!] \\ &= \lambda ks.k(\lambda xrd.\boxed{r}x)\end{aligned}$$

We can, however, define **JI** in terms of `go` if we use the static λ_{s}, that is $\mathbf{JI} = \lambda_{\mathsf{s}} x.\mathtt{go}\, x$, as this does not inadvertently shadow the continuation s that we want **JI** to refer to.

The whole transform for the calculus with **JI** is this:

$$\begin{aligned}[\![x]\!] &= \lambda kq.qx \\ [\![\lambda_{\mathsf{r}} x.M]\!] &= \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\ [\![MN]\!] &= \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\ [\![\mathbf{JI}]\!] &= \lambda ks.k(\lambda xrd.\boxed{s}x)\end{aligned}$$

Recall that the role of `here` has been usurped by λ_{r}, and we replaced `go` by its first-class cousin **JI**.

In the transform for **JI**, the jump continuation is the current "dump" in the sense of the SECD-machine. The dump in the SECD-machine is a sort of call stack, which holds the return continuation for the procedure whose body is currently being evaluated. Making the dump into a first-class object was precisely how Landin invented first-class control, embodied by the **J**-operator.

The typing for the language with **JI** is given in Figure 4. In particular, the type of **JI** is the classical disjunction

$$\frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

As an example of the type system for the calculus with the **JI**-operator, we see that Reynolds's [9] definition of `call/cc` in terms of **JI** typechecks. (Strictly speaking, Reynolds used `escape`, the binding-form cousin of `call/cc`, but `call/cc` and `escape` are syntactic sugar for each other.) We infer the type of $\mathtt{call/cc} \equiv \lambda_{\mathsf{r}} f.((\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI}))$ to be:

$$((A \to B) \to A) \to A)$$

To write the derivation, we abbreviate some contexts as follows:

$$\begin{aligned}\Gamma_{fk} &\equiv f : (A \to B) \to A, k : (A \to B) \\ \Gamma_f &\equiv f : (A \to B) \to A\end{aligned}$$

Then we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma_{fk} \vdash_{\mathsf{j}} f : (A \to B) \to A, A} \qquad \overline{\Gamma_{fk} \vdash_{\mathsf{j}} k : (A \to B), A}}{\Gamma_{fk} \vdash_{\mathsf{j}} f\, k : A, A}}{\Gamma_f \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} k.fk : (A \to B) \to A, A} \qquad \overline{\Gamma_f \vdash_{\mathsf{j}} \mathbf{JI} : A \to B, A}}{\Gamma_f \vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI}) : A, A}}{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.((\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI})) : ((A \to B) \to A) \to A), C}$$

Because **JI** has such evident logical meaning as classical disjunction, we have considered it as basic. Landin [6] took another operator, called **J**, as primitive, while **JI** was derived as the special case of **J** applied to the identity combinator:

$$\mathbf{J}\,\mathbf{I} = \mathbf{J}\,(\lambda x.x)$$

This explains the name "**JI**", as "**J**" stands for "jump" and **I** for "identity". We were able to start with **JI**, since (as noted by Landin) the **J**-operator is syntactic sugar for **JI** by virtue of:

$$\mathbf{J} = (\lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx))\,(\mathbf{JI}).$$

To accommodate **J** in our typing, we use this definition in terms of **JI** to derive the following type for **J**:

$$\vdash_{\mathsf{j}} \mathbf{J} : (A \to B) \to (A \to C), B$$

Let Γ be the context $x : A, r : B \to C, f : A \to B$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash_{\mathsf{j}} r : B \to C, C} \qquad \dfrac{\overline{\Gamma \vdash_{\mathsf{j}} f : A \to B, C} \qquad \overline{\Gamma \vdash_{\mathsf{j}} x : A, C}}{\Gamma \vdash_{\mathsf{j}} fx : B, C}}{\Gamma \vdash_{\mathsf{j}} r(fx) : C, C}}{r : B \to C, f : A \to B \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.r(fx) : A \to C, A \to C}}{r : B \to C \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (A \to B) \to (A \to C), (A \to B) \to (A \to C)}}{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (B \to C) \to (A \to B) \to (A \to C), B}}{\vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx))\,(\mathbf{JI}) : (A \to B) \to (A \to C), B}$$

This type reflects the behaviour of the **J**-operator in the SECD machine. When **J** is evaluated, it captures the B-accepting current dump continuation; it can then be applied to a function of type $A \to B$. This function is composed with the captured dump, yielding a non-returning function of type $A \to C$, for arbitrary C. By analogy with `call-with-current-continuation`, we may read the **J**-operator as "`compose-with-current-dump`" [13].

The logical significance, if any, of the extra function types in the general **J** seems unclear. There is a curious, though vague, resemblance to exception handlers in dynamic control, since they too are functions only to be applied on jumping. This feature of **J** may be historical, as it arose in a context where

greater emphasis was given to attaching dumps to functions than to dumps as first-class continuations in their own right.

6 Type preservation

The typings agree with the transforms in that they are preserved in the usual way for CPS transforms: we have a "double-negation" transform for types, contexts and judgements. The only (slight) complication is in typing the dynamic continuation in those transforms that ignore it.

The function type of the form $A \to B \vee C$ for the dynamic semantics is translated as follows:

$$[\![A \to B \vee C]\!] = [\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to ([\![C]\!] \to \mathbf{Ans}) \to \mathbf{Ans}$$

Each call expects not only the B-accepting return continuation, but also the C-accepting continuation determined by the `here` that encloses the call.

Because we have not varied the transform of application, functions defined with λ_{s} and λ_{r} are also passed this dynamic continuation, even though they ignore it:

$$[\![\lambda_{\mathsf{s}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s})$$
$$[\![\lambda_{\mathsf{r}} x.M]\!] = \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r})$$

In both of these cases, the dynamic jump continuation d is fed to each function call, but never needed. Each function definition must expect this argument to be of certain type. Because different calls of the same function may have dynamically enclosing `here` operators with different types, the type ascribed to d should be polymorphic.

So the function type of the form $A \to B$ is transformed so as to accept this unwanted argument polymorphically:

$$[\![A \to B]\!] = \forall\beta.[\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to \beta \to \mathbf{Ans}$$

That is, a function of type $A \to B$ accepts an argument of type A, a B-accepting return continuation, and the continuation determined by the `here` dynamically enclosing the call.

For all the transforms we have preservation of the respective typing: if $\Gamma \vdash_? M : A, B$, then

$$[\![\Gamma]\!] \vdash [\![M]\!] : ([\![A]\!] \to \mathbf{Ans}) \to ([\![B]\!] \to \mathbf{Ans}) \to \mathbf{Ans}.$$

The proof is a straightforward induction over the derivation.

As a typical example, consider how the classical axiom of excluded middle

$$\vdash_{\mathsf{j}} \mathbf{JI} : A \to B, A$$

is translated to the λ-term $[\![\mathbf{JI}]\!] = \lambda ks.k(\lambda xrd.rx)$ with the type

$$((\forall\beta.[\![A]\!] \to ([\![B]\!] \to \mathbf{Ans}) \to \beta \to \mathbf{Ans}) \to \mathbf{Ans}) \to ([\![A]\!] \to \mathbf{Ans}) \to \mathbf{Ans}.$$

Fig. 5. Comparison of the type systems as logics

Static `here` and `go`, implies `call/cc`

$$\frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{s}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{s}} B, C}{\Gamma \vdash_{\mathsf{s}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} A \to B, C \quad \Gamma \vdash_{\mathsf{s}} A, C}{\Gamma \vdash_{\mathsf{s}} B, C}$$

Dynamic `here` and `go`, like checked exceptions

$$\frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{d}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} A, C}{\Gamma \vdash_{\mathsf{d}} B, C}$$

Non-first class `return`-operation

$$\frac{\Gamma \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{r}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} A \to B, C \quad \Gamma \vdash_{\mathsf{r}} A, C}{\Gamma \vdash_{\mathsf{r}} B, C}$$

Landin's **JI**-operator

$$\frac{}{\Gamma \vdash_{\mathsf{j}} B \to C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{j}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{j}} B, B}{\Gamma \vdash_{\mathsf{j}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} A \to B, C \quad \Gamma \vdash_{\mathsf{j}} A, C}{\Gamma \vdash_{\mathsf{j}} B, C}$$

7 Conclusions

As a summary of the four control constructs we have considered, we present their typings in Figure 5, omitting the terms for conciseness. As logical systems, these toy logics may seem a little eccentric, with two succedents that can only be manipulated in a slightly roundabout way. But they are sufficient for our purposes here, which is to illustrate the correspondence of first-class continuations with classical logic and weaker control operation with intuitionistic logic, and the central role of the arrow type in this dichotomy.

Recall the following fact from proof theory (see for example [15]). Suppose

one starts from a presentation of intuitionistic logic with sequents of the form $\Gamma \vdash \Delta$. If a rule like the following is added that allows $\rightarrow$-introduction even if there are multiple succedents, the logic becomes classical.

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}$$

In continuation terms, the significance of this rule is that the function closure of type $A \rightarrow B$ may contain any of the continuations that appear in Δ; to use the jargon, these continuations become "reified". The fact that the logic becomes classical means that once we can have continuations in function closures, we gain first-class continuations and thereby the same power as `call/cc`. We have this form of rule for static `here` and `go`; though not for **JI**, since **JI** as the excluded middle is already blatantly classical by itself.

But the logic remains intuitionistic if the $\rightarrow$-introduction is restricted. The rule for this case typically admits only a single formula on the right:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B, \Delta}$$

Considered as a restriction on control operators, this rule prohibits λ-abstraction for terms that contain free continuation variables. There are clearly other possibilities how we can prevent assumptions from Δ to become hidden (in that they can be used in the derivation of $A \rightarrow B$ without showing up in this type itself). We could require these assumptions to remain explicit in the arrow type, by making Δ a singleton that either coincides with the B on the right of the arrow, or is added to it:

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \rightarrow B, C} \qquad \frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \rightarrow B \vee C, D}$$

These are the rules for $\rightarrow$-introduction in connection with the `return`-operation, and dynamic `here` and `go`, respectively. Neither of which gives rise to first-class continuations, corresponding to the fact that with these restrictions on $\rightarrow$-introduction the logics remain intuitionistic.

The distinction between static and dynamic control in logical terms appears to be new, as is the logical explanation of Landin's **JI**-operator.

7.1 Related work

Following Griffin [3], there has been a great deal of work on classical types for control operators, mainly on `call/cc` or minor variants thereof. A similar CPS transforms for dynamic control (exceptions) has appeared in [5], albeit for a very different purpose. Felleisen describes the **J**-operator by way of CPS, but since his transform is not double-barrelled, **J** means something different in each λ [2]. Variants of the `here` and `go` operators are even older than the notion of continuation itself: the operations `valof` and `resultis` from CPL later appeared in Strachey and Wadsworth's report on continuations [11,12].

These operators led to the modern `return` in C. As we have shown here, they lead to much else besides if combined with different flavours of λ.

7.2 Further work

In this paper, control constructs were compared by CPS transforms and typing of the *source*. A different, but related approach compares them by typing in the *target* of the CPS [1]. On the source, we have the dichotomy between intuitionistic and classical typing, whereas on the target, the distinction is between linear and intuitionistic. We hope to relate these in further work.

References

[1] Berdine, J., P. W. O'Hearn, U. Reddy and H. Thielecke, *Linearly used continuations*, in: A. Sabry, editor, *Proceedings of the 3rd ACM SIGPLAN Workshop on Continuations*, 2001.

[2] Felleisen, M., *Reflections on Landin's J operator: a partly historical note.*, Computer Languages **12** (1987), pp. 197–207.

[3] Griffin, T. G., *A formulae-as-types notion of control*, in: *Proc. 17th ACM Symposium on Principles of Programming Languages*, San Francisco, CA USA, 1990, pp. 47–58.

[4] Kelsey, R., W. Clinger and J. Rees, editors, *Revised^5 report on the algorithmic language Scheme*, Higher-Order and Symbolic Computation **11** (1998), pp. 7–105.

[5] Kim, J., K. Yi and O. Danvy, *Assessing the overhead of ML exceptions by selective CPS transformation*, in: *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

[6] Landin, P. J., *A generalization of jumps and labels*, Report, UNIVAC Systems Programming Research (1965).

[7] Landin, P. J., *A generalization of jumps and labels*, Higher-Order and Symbolic Computation **11** (1998), reprint of [6].

[8] Plotkin, G., *Call-by-name, call-by-value, and the λ-calculus*, Theoretical Computer Science **1** (1975), pp. 125–159.

[9] Reynolds, J. C., *Definitional interpreters for higher-order programming languages*, in: *Proceedings of the 25th ACM National Conference* (1972), pp. 717–740.

[10] Riecke, J. G. and H. Thielecke, *Typed exceptions and continuations cannot macro-express each other*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS **1644** (1999), pp. 635–644.

[11] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps*, Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK (1974).

[12] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps*, Higher-Order and Symbolic Computation **13** (2000), pp. 135–152, reprint of [11].

[13] Thielecke, H., *An introduction to Landin's "A generalization of jumps and labels"*, Higher-Order and Symbolic Computation **11** (1998), pp. 117–124.

[14] Thielecke, H., *On exceptions versus continuations in the presence of state*, in: G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000,*, number 1782 in LNCS (2000), pp. 397–411.

[15] Troelstra, A. S. and H. Schwichtenberg, "Basic Proof Theory," Cambridge University Press, 1996.

MFPS 17 Preliminary Version

Distance and Measurement in Domain Theory

Paweł Waszkiewicz

School of Computer Science
The University of Birmingham
Birmingham, United Kingdom

Abstract

We investigate the notion of distance on domains. In particular, we show that measurement is a fundamental concept underlying partial metrics by proving that a domain in its Scott topology is partially metrizable only if it admits a measurement. Conversely, the natural notion of a distance associated with a measurement not only yields meaningful partial metrics on domains of essential importance in computation, such as $\mathbf{I}\mathbb{R}$, Σ^∞ and $\mathcal{P}\omega$, it also serves as a useful theoretical device by allowing one to establish the existence of partial metrics on arbitrary ω-continuous dcpo's.

1 Introduction

The theory by Keye Martin, introduced in [5], investigates domains equipped not only with order but also with a quantitative notion of measurement. The theory is easy to understand, being based on the "informatic" intuition behind domain theory. It is widely applicable. Most of the domains arising in applications of domain theory have measurements, including the class of all countably based domains. Two central notions of the theory are a *measurement* and the *μ-topology* called here the *Martin topology*. The last one is Hausdorff on a domain and finer than both Scott and Lawson topologies. It is well-suited for computation: both continuity and completeness of a domain can be described in terms of the Martin topology.

The main theme of this paper is the study of the notion of distance on domains. Our work in this direction is very much inspired by questions posed by Reinhold Heckmann in [4] and Keye Martin in [5]. One obvious candidate for a distance on domains is a partial metric such that the partial metric topology agrees with the Scott topology of the induced order (see Section 2.2 for definitions). Another one is a symmetric map d_μ built from a measurement μ by a standard construction.

[1] Email: P.Waszkiewicz@cs.bham.ac.uk

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The first problem of Heckmann's is to characterize partial metric spaces which are continuous dcpo's with respect to the induced order and such that the Scott topology and the partial metric topology agree. The other challenge is to show which continuous dcpo's are partially metrizable.

We show that answers to both questions can be achieved by introducing methods of measurement theory into the study of partial metric spaces. In Section 3 we show that a continuous poset, which is partially metrizable in its Scott topology must admit a measurement. Under some additional, mild restrictions, the converse also holds: if the self-distance mapping for the partial metric is a measurement, then as a consequence, the partial metric topology agrees with the Scott topology.

Our thesis is that d_μ, called here the distance function associated with a measurement μ, deserves its name. We study its basic properties in Section 4. It is well-known [5] that d_μ induces the Scott topology. We prove that it also encodes the underlying order, in the same fashion as partial metrics do. Therefore, it is natural to ask if d_μ is a partial metric. We demonstrate (see Section 5) that for *arbitrary* measurements the answer is positive for a restricted class of domains, which is, however, large enough to advance O'Neill's construction from [7]. Our final argument in favour of d_μ being a distance between elements of a domain is presented in the last section. We show that every ω-continuous dcpo is partially metrizable and the partial metric is the distance function d_μ associated with *some* measurement μ on the domain. This result solves the second problem of Heckmann's for the class of all countably based domains.

1.1 Convention

In the paper we adopt the following convention: original results are the numbered ones unless they are acknowledged explicitly. For instance, all the examples of measurements on domains from Section 2.5 are taken from [5].

2 Background

2.1 Domain theory

We review some basic notions from domain theory, mainly to fix the language and notation. See [1] for more information. Let P be a poset. A pair of elements $x, y \in P$ is *consistent (bounded)* if there exists an element $z \in P$ such that $z \sqsupseteq x, y$. We say that a poset is *bounded-complete* if each bounded pair of elements has a supremum. A subset $A \subseteq P$ of P is *directed* if it is nonempty and any pair of elements of A has an upper bound in A. If a directed set A has a supremum, it is denoted $\bigsqcup^{\uparrow} A$. A poset P in which every directed set has a supremum is called a *dcpo*.

Let x and y be elements of a poset P. We say that x *approximates (is way-below)* y if for all directed subsets A of P, $y \sqsubseteq \bigsqcup^{\uparrow} A$ implies $x \sqsubseteq a$ for

some $a \in A$. We denote it as $x \ll y$. Now, $\downarrow x$ is the set of all approximants of x below it. $\uparrow x$ is defined dually. We say that a subset B of a dcpo P is a *(domain-theoretic)* basis for P if for every element x of P, the set $\downarrow x \cap B$ is directed with supremum x. A poset is called *continuous* if it has a basis. It can be shown that a poset P is continuous iff $\downarrow x$ is directed with supremum x, for all $x \in P$. A poset is called a *domain* if it is a continuous dcpo.

A subset $U \subseteq P$ of a poset P is *upper* if $x \sqsupseteq y \in U \Rightarrow x \in U$. Upper sets inaccessible by directed suprema form a topology called the *Scott topology*; it is denoted σ_P. A domain admits a countable domain-theoretic basis iff the Scott topology is second countable. In this case the domain is called an ω-*continuous domain*. The Scott topology encodes the underlying order: $x \sqsubseteq y$ in P iff $\forall U \in \sigma.\ (x \in U \Rightarrow y \in U)$. This is the general definition of the so-called *specialisation order* for a topology. The collection $\{\uparrow x \mid x \in D\}$ forms a basis for the Scott topology on a continuous poset D. The Scott topology satisfies only weak separation axioms: it is always T_0 on a poset but T_1 only if the order is trivial. The topology is *sober* on a domain (a topological space is sober iff it is T_0 and every nonempty closed subset which is not the union of two closed proper subsets is the closure of a point). Sobriety of a space implies that the underlying specialisation order is a dcpo. For continuous posets, being a dcpo and sobriety of the Scott topology are equivalent conditions.

The poset $[0, \infty)^{op}$ figures prominently in Martin's work and also in this note. It is a domain without least element. We use $\sqsubseteq$ to refer to its order which is dual to the natural one, $\leq$, and try to avoid the latter entirely. ($\leq$ is used in this paper whenever we work with $[0, \infty)$.)

2.2 Partial metrics

We will briefly review basic definitions and facts about partial metric spaces from Heckmann's [4] and Matthew's articles [6].

A partial metric on a set X is a map $p : X \times X \to [0, \infty)$ which satisfies for all $x, y, z \in X$,

1. $p(x, y) = p(y, x)$ (symmetry),

2. $p(x, y) = p(x, x) = p(y, y)$ implies $x = y$ (T_0 separation axiom),

3. $p(x, y) \leq p(x, z) + p(z, y) - p(z, z)$ ($\Delta^\sharp$),

4. $p(x, x) \leq p(x, y)$ (SSD - "small self-distances").

If we abandon Axiom 4, p is called a *weak partial metric*. From the topological point of view, weak partial metrics and partial metrics are equivalent since for every weak partial metric p there is a corresponding one which satisfies SSD [4], given by $p'(x, y) := \max\{p(x, y), p(x, x), p(y, y)\}$.

The topology τ_p induced by the partial metric p is the topology which has

a basis consisting of open balls of the form

$$B_\varepsilon(x) := \{y \in X \mid p(x,y) < p(x,x) + \varepsilon\}$$

for an $x \in X$ and a radius $\varepsilon > 0$. The definition is well-formed since the collection of open balls indeed forms a basis for a topology on X.

The name "T_0 separation axiom" is justified by the fact that it is a necessary and sufficient condition for X to be a T_0 space w.r.t. τ_p. It is not Hausdorff in general, as the example of the formal ball model shows. Therefore, the specialisation order $\sqsubseteq_{\tau_P}$ of τ_p will be non-trivial in general.

All of the τ_p-open sets, the open balls among them, are upper sets with respect to the order.

We have that the following are equivalent for all $x, y \in X$:

1. $x \sqsubseteq_{\tau_P} y$,
2. $p(x,y) = p(x,x)$,
3. $\forall \varepsilon > 0 \quad y \in B_\varepsilon(x)$.

We will say $x \sqsubseteq_p y$ if one of the above conditions holds.

A *weighted quasi-metric* on a set X is a pair of maps (q, w) consisting of a *quasi-metric* $q\colon X^2 \to [0,\infty)$ (satisfies all metric axioms but symmetry) and a *weight function* $w\colon X \to [0,\infty)$ where for all $x, y \in X$, $q(x,y) + w(x) = q(y,x) + w(y)$. q induces order and topology in the usual manner: for all $x, y \in X$, $x \sqsubseteq_q y$ iff $q(x,y) = 0$ and $B^q_\varepsilon(x) = \{y \in X \mid q(x,y) < \varepsilon\}$ is a basis for the induced topology τ_q. Matthews [6] proves that there is an algebraic equivalence between a partial metric p on X and a weighted quasi-metric (q, w) given by $p(x,y) := q(x,y) + w(x)$ and conversely $q(x,y) := p(x,y) - w(x)$ and, moreover, p and q induce the same order and topology. We will exploit this in the last theorem of the paper.

Finally, for every partial metric space (X, p), if X is equipped with the topology τ_p induced by p and $[0,\infty)^{op}$ with the Scott topology, then the mapping $p : X \times X \to [0,\infty)^{op}$ is continuous. Since every continuous map is monotone with respect to the specialisation orders of its domain and codomain, $p : X \times X \to [0,\infty)^{op}$ and the corresponding weight function $w : X \to [0,\infty)^{op}$ are monotone. This is one of the reasons why one can hope for the weight (self-distance) function to be a measurement.

2.3 Martin's theory

We give a summary of the main elements of Keye Martin's theory of measurements on domains. Our main reference is [5].

Let P be a poset and E a domain. For a monotone mapping $\mu\colon P \to E$ and any $x \in P$, $\varepsilon \in E$ we define

$$\mu_\varepsilon(x) := \{y \in P \mid y \sqsubseteq x \wedge \varepsilon \ll \mu y\} = \mu^{-1}(\uparrow\!\!\!\uparrow \varepsilon) \cap \downarrow x.$$

We say that $\mu_\varepsilon(x)$ is the set of elements of P which are *ε-close* to $x \in P$. Since in most cases we assume $E = [0,\infty)^{op}$, we read $\varepsilon \ll \mu(y)$ as $\mu(y) < \varepsilon$ in the natural order, which matches the intuition behind the name of $\mu_\varepsilon(x)$. The map μ can be thought of as a quantitative measure of a relative "distance" between elements in P. Immediately we have that $\mu_\varepsilon(x) \neq \emptyset$ iff $x \in \mu_\varepsilon(x)$ and for any $y \in P$, if $y \in \mu_\varepsilon(x)$, then $y \in \mu_\varepsilon(y) \subseteq \mu_\varepsilon(x)$.

We say that a monotone mapping $\mu\colon P \to E$ *induces the Scott topology* on a subset X of a poset P if $\forall U \in \sigma_P\ \forall x \in X.\ x \in U \Rightarrow (\exists \varepsilon \in E)\ x \in \mu_\varepsilon(x) \subseteq U$. We denote it as $\mu \longrightarrow_X \sigma_P$. If $X = P$, we write $\mu \longrightarrow \sigma_P$, which reads: μ *induces the Scott topology everywhere (on P).*

In the paper, the following observation will often be referred to as the *measurement property*: for a map $\mu\colon P \to [0,\infty)^{op}$ on a continuous poset P and for any $X \subseteq P$, the following are equivalent:
(i) μ is Scott-continuous and induces the Scott topology everywhere on X,
(ii) for all $x \in X$ and all subsets $S \subseteq \mathop{\downarrow\!\!\!\downarrow} x$, S is directed with supremum x iff $\bigsqcup\{\mu s \mid s \in S\} = \mu x$.

It is not hard to show that the identity mapping on a domain P induces the Scott topology everywhere on P. Moreover, the property is preserved by the composition of maps. A *measurement* on a continuous poset D is a Scott-continuous mapping $\mu\colon D \to [0,\infty)^{op}$ which induces the Scott topology on its kernel $\ker\mu := \{x \in D \mid \mu(x) = 0\}$.

Martin's theory has a rich topological dimension. The *Martin topology* (also called the *μ topology*) arises naturally in the consideration of measurements. For any monotone mapping $\mu\colon D \to E$ between domains, the collection $\{\mu_\varepsilon(x) \mid x \in D, \varepsilon \in E\}$ forms a basis for a topology on D. In particular, if μ is taken to be the identity map on D, we obtain a topology with a basis $\{\uparrow\!\!\!\uparrow x \cap \downarrow y \mid x, y \in D\}$. We call this topology the *Martin topology* on D. The following important *Invariance Theorem* holds: if $\mu\colon D \to E$ is Scott-continuous, then μ induces the Scott topology on D iff $\{\mu_\varepsilon(x) \mid x \in D, \varepsilon \in E\}$ is a basis for the Martin topology on D. That is, no matter how we measure a domain, all measurements give rise to the same μ topology on the domain. The Martin topology is always Hausdorff on a domain. The study of its properties is the subject of a chapter in Martin's thesis [5].

In our paper we work on posets equipped with a particularly pleasant class of measurements which induce the Scott topology everywhere on their domains. We are able to characterize both the order (see Sections 4) and completeness of a domain strictly in terms of the measurement.

2.4 Completeness

For any topology τ, the collection of intersections $C \cap O$ of a closed set C and an open set O of τ forms a basis of a topology, the so-called b-topology for τ. Sünderhauf [8] shows that τ is sober iff every *observative net* converges in the b-topology for τ. (A net $(x_i)_{i\in I}$ is observative if for all $i \in I$ and for all

$U \in \tau$, $x_i \in U$ implies that the net is eventually in U.) In the case of posets with measurements, we can confine our attention to observative sequences:

Lemma 2.1 *Let P be a continuous poset with a measurement $\mu\colon P \to [0,\infty)^{op}$ such that $\mu \longrightarrow \sigma_P$. The Scott topology on P is first-countable.*

Proof. P is first countable since $\{\uparrow \mu_{\mu x + \frac{1}{n}}(x) \mid n \in \mathbb{N}\}$ is a countable neighbourhood base at $x \in P$. □

It comes as no surprise that:

Proposition 2.2 *The Martin topology is the b-topology for the Scott topology on a continuous poset P.*

Proof. The collection $\{\Uparrow x \cap \downarrow y \mid x, y \in P\}$ is a basis for the Martin topology on P. Thus, the Martin topology is always coarser than the b-topology. To prove the converse, denote the b-topology for the Scott topology by τ and let $x \in U \in \tau$. We can assume U is a basic-open set in τ and hence $U = O \cap C$, where O is a Scott-open set and C is Scott-closed. Let us choose an element $y \in U$ way-below x such that $y \in O$. Also, $y \in C$, since C is downward closed. Consequently, $y \in U$. We claim that the set $A := \Uparrow y \cap \downarrow x$ is a subset of U. Indeed, if $z \in A$, then $z \in \Uparrow y \subseteq O$. Also, $z \in \downarrow x \subseteq C$. Therefore, $z \in U$. Since A is basic-Martin open, we are done. □

Therefore, Martin's Invariance Theorem states that the b-topology for the Scott topology on P can be constructed from a measurement with $\mu \longrightarrow \sigma_P$ (the proof of the Theorem holds *verbatim*, even if P is not a dcpo). Now, Sünderhauf's result gives that a continuous poset is sober (equivalently: is a dcpo) iff every observative sequence in P Martin-converges in P. However, it happens that with much simpler reasoning we can prove a stronger result. We need to know a few simple facts about convergence in the Martin topology, all proved in [5]. Firstly, given a measurement $\mu\colon P \to [0,\infty)^{op}$ on a continuous poset P, a sequence (x_n) converges to an $x \in P$ in the Martin topology on P iff $\lim \mu x_n = \mu x$ and (x_n) is eventually in $\downarrow x$. Secondly, a sequence (x_n) Martin-converges to an x iff it Scott-converges and (x_n) is eventually in $\downarrow x$.

Lemma 2.3 *A continuous poset P with a measurement $\mu\colon P \to [0,\infty)^{op}$ with $\mu \longrightarrow \sigma_P$ is a dcpo iff every increasing sequence (x_n) Martin-converges in P.*

Proof. Let (x_n) be a sequence with $x = \bigsqcup^{\uparrow} x_n$. Since μ is Scott-continuous,

$$\mu x = \mu(\bigsqcup{}^{\uparrow} x_n) = \bigsqcup \{\mu x_n \mid n \in \mathbb{N}\} = \lim_{n\to\infty} \mu x_n.$$

Since $x_n \sqsubseteq x$ for every $n \in \mathbb{N}$, (x_n) Martin-converges. The proof of the converse is essentially the content of Corollary 3.1.3 of [5] and we give it only for the sake of completeness: Martin-convergence of (x_n) to x implies that the sequence is eventually below x. Since the sequence is increasing, all x_n are

below x. Let u be another upper bound for the sequence. For every Scott-open set U around x, there exists k such that $x_k \in U$, by Scott-convergence. Now, since U is upper, $x_k \sqsubseteq u \in U$. This proves $x \sqsubseteq u$. □

We conclude this section with a summary of results:

Theorem 2.4 *Let P be a continuous poset with a measurement $\mu\colon P \to [0,\infty)^{op}$ with $\mu \longrightarrow \sigma_P$. The following are equivalent:*

(i) *the Scott topology on P is sober,*

(ii) *P is a dcpo,*

(iii) *all increasing sequences converge in the Scott topology on P,*

(iv) *all increasing sequences converge in the Martin topology on P,*

(v) *all observative sequences converge in the Martin topology on P.* □

2.5 Examples of domains with measurements

Cantor set model Σ^∞. Let Σ^∞ denote the set of all finite and infinite words over a finite alphabet Σ, with the prefix ordering. This is an ω-algebraic domain. For all $x, y \in \Sigma^\infty$, $x \ll y$ holds iff $x \sqsubseteq y$ and x is finite. The mapping

$$\frac{1}{2^{|\cdot|}} : \Sigma^\infty \to [0,\infty)^{op}$$

where $|\cdot| : \Sigma^\infty \to \mathbb{N} \cup \{\infty\}$ takes a string to its length is a measurement on Σ^∞. Moreover, it induces the Scott topology everywhere on Σ^∞.

The interval domain $\mathbf{I}\mathbb{R}$. The collection $\mathbf{I}\mathbb{R}$ of compact intervals of the real line ordered under reverse inclusion is an ω-continuous domain. The supremum of a directed set $S \subseteq \mathbf{I}\mathbb{R}$ is $\bigcap S$ and for all intervals $x, y \in \mathbf{I}\mathbb{R}$ we have $x \ll y$ iff x is contained in the interior of y. The length function $|\cdot| : \mathbf{I}\mathbb{R} \to [0,\infty)^{op}$ given by $|x| = \overline{x} - \underline{x}$, where $x = [\underline{x}, \overline{x}] \in \mathbf{I}\mathbb{R}$, is a measurement on $\mathbf{I}\mathbb{R}$. It induces the Scott topology everywhere on $\mathbf{I}\mathbb{R}$.

The powerset of naturals $\mathcal{P}\omega$. The collection of all subsets of $\mathbb{N}$ ordered by inclusion is an ω-algebraic domain. The supremum of a directed set $S \subseteq \mathcal{P}\omega$ is $\bigcup S$ and for all elements x, y of $\mathcal{P}\omega$ the approximation relation is given by $x \ll y$ iff $x \subseteq y$ and x finite. The mapping $|\cdot| : \mathcal{P}\omega \to [0,\infty)^{op}$ given by

$$|x| = 1 - \sum_{n \in x} \frac{1}{2^{n+1}}$$

is a measurement on $\mathcal{P}\omega$. It induces the Scott topology everywhere on $\mathcal{P}\omega$.

The formal ball model $\mathbf{B}X$, introduced in [2]. The mapping $\mu : \mathbf{B}X \to [0,\infty)^{op}$ given by $\mu(x,r) = r$ is a measurement on $\mathbf{B}X$. It induces the Scott topology everywhere on $\mathbf{B}X$.

The domain of finite lists $[S]$ over a set S. A list x over a set S is a map $x\colon\{1,2,...,n\} \to S$ for $n \geq 0$. Informally, for $x, y \in [S]$, y is a *sublist* of x if y matches some convex subset of x, e.g. $[a,b]$ is a sublist of $[c,a,b,d]$, while $[a,d]$ is not. We define a partial order on $[S]$ by $x \sqsubseteq y$ iff y is a sublist of x. With this order, $[S]$ is an algebraic dcpo, where every element is compact. $[S]$ is ω-continuous iff S is countable. The *length of the list*, $len\colon[S] \to \mathbb{N}$, given by $len(x) := |dom(x)|$ (cardinality of the domain of x) is a measurement on $[S]$, which induces the Scott topology everywhere on $[S]$.

In all the examples above, the kernel of the measurement is precisely the set of maximal elements. However, we do not know if for arbitrary ω-continuous dcpo, the set of maximals is the kernel of some measurement on the domain. This is already a 3-year old problem. Below, we show that it is the condition on the kernel which causes the difficulty, since it is easy to find a measurement on a domain with countable basis (with possibly empty kernel).

Example 2.5 [5] For any continuous dcpo D with a countable basis $\{U_n \mid n \in \mathbb{N}\}$ for the Scott topology, a mapping $\mu\colon D \to [0,\infty)^{op}$ given by

$$\mu(x) := 1 - \sum_{\{n\in\mathbb{N}:x\in U_n\}} \frac{1}{2^{n+1}}$$

is a measurement which induces the Scott topology everywhere on D.

3 The necessity of measurement on partially metrizable domains

In this paper, we are mainly concerned with the case when a partial metric topology is the Scott topology of the induced order, $\tau_p = \sigma$ in symbols. We demonstrate that such a class of partial metrics is intimately connected to measurements. We give a construction of a measurement from a given partial metric with $\tau_p = \sigma_X$ on an arbitrary set X. Precisely, for a partial metric p on a set X, the self-distance mapping $\mu\colon X \to [0,\infty)^{op}$ given by $\mu(x) := p(x,x)$ for all $x \in X$ is Scott-continuous and induces the Scott-topology everywhere on X.

Moreover, it happens that under some mild, computationally meaningful restrictions on an underlying poset X, the converse also holds: if the self-distance map μ is a measurement which induces the Scott topology everywhere, then $\tau_p = \sigma$.

We use $\sigma_X \subseteq \tau_p$ to denote the fact that the partial metric topology is larger than the Scott topology of the induced order $\sqsubseteq_p$. The meaning of $\tau_p \subseteq \sigma_X$ is analogous. Also, in this section, $\mu \longrightarrow \sigma$ means that the mapping μ induces the Scott topology everywhere on X.

Theorem 3.1 *Let (X,p) be a partial metric space such that the Scott topology of the order $\sqsubseteq_p$ agrees with the partial metric topology τ_p. Then the self-*

distance map $\mu\colon X \to [0,\infty)^{op}$ *is Scott-continuous and has property* $\mu \longrightarrow \sigma$.

Proof. First, we will show that if $\sigma_X \subseteq \tau_p$, then $\mu \longrightarrow \sigma$. Indeed, let $x \in U \in \sigma_X$. Since $\sigma_X \subseteq \tau_p$, there exists an $\varepsilon > 0$ such that $x \in B_\varepsilon(x) \subseteq U$. Define $\delta := \mu(x) + \varepsilon$. Since $\mu(x) < \delta$, $x \in \mu_\delta(x)$. Now, let $y \in \mu_\delta(x)$. Since $p(x,y) \leq \mu(y)$ as $y \sqsubseteq_p x$ and $\mu(y) < \delta = \mu(x) + \varepsilon$, we have $p(x,y) < \mu(x) + \varepsilon$. This means $y \in B_\varepsilon(x)$. Therefore $\mu_\delta(x) \subseteq B_\varepsilon(x)$.

Now, it remains to show that if $\tau_p \subseteq \sigma_X$, then the self-distance map $\mu\colon X \to [0,\infty)^{op}$ is Scott-continuous. For, since $p : X \times X \to [0,\infty)^{op}$ is τ_p-continuous, also μ is τ_p-continuous. The Scott-continuity of μ follows immediately from the assumption. □

Therefore, we obtained a necessary condition for partial metrizability of the Scott topology on continuous posets.

Corollary 3.2 *Every partially metrizable continuous poset admits a measurement which induces the Scott topology everywhere.*

It happens that there is a class of partial metric spaces where inducing the Scott topology by the self-distance map is equivalent to the agreement of the Scott and partial metric topologies.

Definition 3.3 We call a partial metric space stable if

$$\forall x, y \in X.\ p(x,y) = \bigsqcup\{\mu z \mid z \sqsubseteq_p x, y\}.$$

Notice that the last condition is equivalent to

$$\forall x, y \in X\ \forall \varepsilon > 0\ \exists z \sqsubseteq_p x, y.\ \mu(z) < p(x,y) + \varepsilon.$$

Moreover, if X is a continuous poset with respect to the induced order, then stability can be written as $\forall x, y \in X.\ p(x,y) = \bigsqcup\{\mu z \mid z \ll_p x, y\}$, where $\ll_p$ is the way-below relation obtained from the order $\sqsubseteq_p$.

Theorem 3.4 *Let* (X,p) *be a partial metric space such that:*

1. *X is stable, and*
2. *the induced order $\sqsubseteq_p$ makes X a continuous poset.*

Then the Scott topology of the order $\sqsubseteq_p$ agrees with the partial metric topology τ_p iff the self-distance map $\mu\colon X \to [0,\infty)^{op}$ is a measurement with property $\mu \longrightarrow \sigma$.

Proof. The proof consists of two observations. The first one states that, if (X,p) is a stable space, then $\sigma_X \subseteq \tau_p$ holds iff $\mu \longrightarrow \sigma$. ($\Rightarrow$) has already been shown in the proof of the preceding theorem. For the converse, let $x \in U \in \sigma_X$. By $\mu \longrightarrow \sigma$, we can assume $x \in \mu_\delta(x) \subseteq U$, where $\delta := \mu(x) + \varepsilon$ for some $\varepsilon > 0$. Set $\varepsilon' := \frac{1}{2}\varepsilon$. We want to show $B_{\varepsilon'}(x) \subseteq \uparrow(\mu_\delta(x))$. Let $y \in B_{\varepsilon'}(x)$. Then by

definition, $p(x,y) < \mu(x) + \varepsilon'$. By assumption, there exist $z \sqsubseteq_p x, y$ such that we have

$$\mu(z) < p(x,y) + \varepsilon' < \mu(x) + 2\varepsilon' = \mu(x) + \varepsilon = \delta.$$

Hence we have shown that $z \in \mu_\delta(x)$. Moreover, since $z \sqsubseteq_p y$, $y \in \uparrow\mu_\delta(x)$. Therefore the claim that $B_{\varepsilon'}(x) \subseteq \uparrow(\mu_\delta(x))$ is now proved. Consequently, we have

$$x \in B_{\varepsilon'}(x) \subseteq \uparrow\mu_\delta(x) \subseteq \uparrow U = U,$$

which gives $\sigma_X \subseteq \tau_p$. The proof of the first observation is completed.

The second one states that if (X,p) is a partial metric space such that the induced order $\sqsubseteq_p$ makes X a continuous poset, then $\tau_p \subseteq \sigma_X$ iff the self-distance map $\mu\colon X \to [0,\infty)^{op}$ is Scott-continuous. For ($\Leftarrow$), let $x \in V \in \tau_p$. Take any open ball around x in V, that is, choose $\varepsilon > 0$ such that $x \in B_\varepsilon(x) \subseteq V$. It is easy to show that $x \in \mu_\delta(x) \subseteq B_\varepsilon(x) \subseteq V$, where $\delta := \mu(x) + \varepsilon$. Since $B_\varepsilon(x)$ is an upper set, $x \in \uparrow\mu_\delta(x) \subseteq B_\varepsilon(x) \subseteq V$. Finally, by continuity of X and μ, the set $\uparrow\mu_\delta(x)$ is Scott-open (see also the next section for more detailed explanation). Therefore $\tau_p \subseteq \sigma_X$.

The converse has already been shown in the proof of the preceding theorem. □

4 The distance map associated with a measurement

In the last section we saw that whenever a partial metric induces the Scott topology on the underlying domain, the domain admits a measurement which induces the Scott topology everywhere. This result tells us we should look to measurement in defining a notion of distance on domains. We start with a standard construction from [5].

Given a continuous poset P equipped with a measurement $\mu\colon P \to E$ with $\mu \longrightarrow \sigma_P$ one can define a mapping $d_\mu\colon P^2 \to E$ given by $d_\mu(x,y) := \bigsqcup\{\mu(z) \mid z \ll x, y\}$, providing that any two elements x, y of P are bounded from below and E is a dcpo. Martin proves that d_μ is Scott-continuous on P^2. Our thesis is that d_μ may serve as a distance function between elements of a domain. In this section we examine basic properties of d_μ.

Definition 4.1 Let P be a continuous poset with a measurement $\mu\colon P \to [0,\infty)^{op}$. The map $d_\mu : P^2 \to [0,\infty)^{op}$ defined by

$$d_\mu(x,y) := \bigsqcup\{\mu(z) \mid z \ll x, y\}$$

is the distance function associated with μ.

Notice that for a continuous poset P with a measurement, we can always assume that d_μ is defined: we simply scale the measurement to $[0,1)^{op}$ by $\mu^* x := \frac{\mu x}{1+\mu x}$, add bottom to P with $\mu^* \bot := 1$ and study d_{μ^*}.

d_μ induces a topology on P. The collection of open balls $\{B_\varepsilon(x) \mid x \in P, \varepsilon > 0\}$ is a basis for the topology, where $B_\varepsilon(x) := \{y \in P \mid d_\mu(x,y) < \varepsilon\}$.

If $\mu: P \to E$ is a Scott-continuous mapping on a continuous poset P with $\mu \longrightarrow \sigma_P$, then $\{\uparrow\mu_\varepsilon(x) \mid x \in P, \varepsilon \in E\}$ is a basis for the Scott topology on P. Now, Martin proved that for all $x \in P$ and $\varepsilon > 0$, $B_\varepsilon(x) = \uparrow\mu_\varepsilon(x)$, that is, the topology induced by d_μ is always the Scott topology. Thanks to this crucial fact, from now on it is clear that d_μ is a computationally important object to study.

First of all, we are going to show that whenever a continuous poset is equipped with a measurement, the induced distance d_μ captures order between elements. Let us start with a well-known fact:

Lemma 4.2 ([5]) *Let P be a continuous poset with a monotone map $\mu: P \to [0,\infty)^{op}$. The following are equivalent:*

(i) *μ is Scott-continuous,*

(ii) *$\mu x = d_\mu(x,x)$ for any $x \in P$,*

(iii) *$x \sqsubseteq y \Rightarrow d_\mu(x,y) = \mu x$ for any $x, y \in P$.*

Theorem 4.3 *Let P be a continuous poset with a measurement $\mu: P \to [0,\infty)^{op}$ with $\mu \longrightarrow \sigma_P$. Then for all $x, y \in P$,*

$$x \sqsubseteq y \iff d_\mu(x,y) = \mu x.$$

Proof. ($\Rightarrow$) by Lemma 4.2. For ($\Leftarrow$) assume $d_\mu(x,y) = \mu x$. Let (x_n) be a sequence with $x_n \ll x, y$ and $\lim \mu x_n = d_\mu(x,y)$. Then $\lim \mu x_n = \mu x$ and by the measurement property, (x_n) is directed with supremum x. Therefore, $x = \bigsqcup^{\uparrow} x_n \sqsubseteq y$. □

Observe an immediate corollary of the result and Example 2.5. We are able to characterize the order relation on arbitrary ω-continuous dcpo.

Corollary 4.4 *For any continuous dcpo D with a countable basis $\{U_n \mid n \in \mathbb{N}\}$ for the Scott topology, $x \sqsubseteq y \iff d_\mu(x,y) = \mu x$, where $\mu: D \to [0,\infty)^{op}$ is given in Example 2.5.* □

Now we have an elementary proof of some properties of d_μ. The first one, below, can be treated as the T_0 axiom in the case when d_μ is a partial metric on D. The second property states the antisymmetry of the order.

Corollary 4.5 *With assumptions of Theorem 4.3, d_μ has the following properties:*

1. *$d_\mu(x,y) = \mu x = \mu y \iff x = y$,*
2. *$d_\mu(x,y) = 0 \iff x = y \in \ker\mu$.* □

The characterization of the order given in Theorem 4.3 reminds us of the definition of the order induced by a partial metric. Therefore one can ask when d_μ is a partial metric.

5 When distance is a partial metric

We now try to justify the intuition that d_μ provides a measure of distance between elements of a domain. In particular, we start with a sufficient condition for d_μ to be a partial metric.

Proposition 5.1 *Let P be a continuous poset with a measurement $\mu: P \to [0,\infty)^{op}$ with $\mu \longrightarrow \sigma_P$. If for all consistent pairs $a, b \in P$ and for all upper bounds r of a and b, there exists an $s \sqsubseteq a, b$ such that*

$$\mu r + \mu s \leq \mu a + \mu b,$$

then $d_\mu: P \to [0,\infty)$ is a partial metric on P such that its induced order agrees with the order on P and the partial metric topology τ_p is the Scott topology on P.

Proof. Proofs of this and next proposition are extensions of Martin's argument in Corollary 5.4.1 of [5].

It is enough to prove that d_μ satisfies $\Delta^\sharp$. Take any $x, y, z \in P$. By definition of d_μ, there exists an $a \sqsubseteq x, z$ and $b \sqsubseteq y, z$ such that

$$d_\mu(x,z) + \frac{\varepsilon}{2} \geq \mu a \quad \wedge \quad d_\mu(y,z) + \frac{\varepsilon}{2} \geq \mu b,$$

for any $\varepsilon > 0$. Since a, b are consistent, there is $s \sqsubseteq a, b$ such that

$$d_\mu(x,y) \leq d_\mu(a,b) \leq \mu s \leq \mu a + \mu b - \mu z.$$

Hence,

$$d_\mu(x,y) + \mu z \leq d_\mu(x,y) + d_\mu(y,z) + \varepsilon,$$

for all $\varepsilon > 0$. This proves that d_μ satisfies $\Delta^\sharp$. Agreement of orders and topologies claimed in the hypothesis follows from general properties of d_μ. □

Notice that if P is bounded-complete and μ is modular, that is, for all consistent pairs $x, y \in P$ we have $\mu(x \sqcup y) + \mu(x \sqcap y) = \mu x + \mu y$, then the conditions of the proposition hold and $d_\mu = \mu(x \sqcap y)$ is a partial metric on P. Hence we advanced the result by O'Neill [7] who gave a construction of a partial metric from a valuation on a so called *valuation space*, i.e. on a bounded-complete inf-semilattice. However, as our last result shows, the existence of suprema and infima is not necessary.

Proposition 5.1 guarantees the existence of a partial metric which induces the Scott topology on $\mathbf{I}\mathbb{R}, \Sigma^\infty, \mathcal{P}\omega$ since their natural measurements are modular.

The mapping $p_{\mathbf{I}\mathbb{R}}: \mathbf{I}\mathbb{R} \times \mathbf{I}\mathbb{R} \to [0,\infty)$ given by

$$p_{\mathbf{I}\mathbb{R}}([\underline{x}, \overline{x}], [\underline{y}, \overline{y}]) := \max\{\overline{x}, \overline{y}\} - \min\{\underline{x}, \underline{y}\}$$

where $[\underline{x}, \overline{x}], [\underline{y}, \overline{y}] \in \mathbf{I}\mathbb{R}$, is a partial metric on $\mathbf{I}\mathbb{R}$.

The mapping $p_{\Sigma^\infty} : \Sigma^\infty \times \Sigma^\infty \to [0, \infty)$ given by

$$p_{\Sigma^\infty}(x, y) := 2^{-|r|},$$

where r is the largest common prefix of x and y, is a partial metric on Σ^∞.

The mapping $p_{\mathcal{P}\omega}: \mathcal{P}\omega \times \mathcal{P}\omega \to [0, \infty)$ given by

$$p_{\mathcal{P}\omega}(x, y) := 1 - \sum_{n \in x \cap y} 2^{-(n+1)}$$

is a partial metric on $\mathcal{P}\omega$.

In more general cases, d_μ is usually no longer a partial metric. Sometimes, however, d_μ still satisfies the classical triangle inequality for metrics.

Proposition 5.2 *Let P be a continuous poset with a measurement $\mu: P \to [0, \infty)^{op}$ with $\mu \longrightarrow \sigma_P$ such that*

$$\exists z \sqsubseteq x, y.\ \mu z \leq \mu x + \mu y.$$

Then $d_\mu: P \to [0, \infty)$ satisfies the triangle inequality and induces the Scott topology on P.

Proof. The reasoning is essentially the same as in the proof of the preceding Proposition. □

Interestingly, in the case above, the restriction of d_μ to $\ker\mu$ is a metric which yields the relative Scott topology on $\ker\mu$. This fact is investigated in detail in Martin's thesis. Further generalization is still possible, but this involves applying a valuable construction due to Frink [3] to the map d_μ, and is beyond our present concern.

6 The existence of partial metrics on countably based domains

The results in the last section make us think that d_μ may serve as a distance map on domains only in restricted cases and hence is not a useful theoretical device in establishing the existence of partial metrics. However, the following result shows that this is not true. It also provides a practical illustration of the techniques developed in sections 3 and 4.

Theorem 6.1 *Let D be an ω-continuous dcpo. Then there is a Scott-continuous partial metric $p : D^2 \to [0, \infty)$ such that*

(i) $\sqsubseteq_p = \sqsubseteq_D$,

(ii) *the Scott topology on D is the partial metric topology τ_p.*

In short, all countably based domains are partially metrizable.

Note the nice analogy between this result and Urysohn's lemma: All regular, second-countable spaces are metrizable.

Proof. Let $\{U_n \mid n \in \mathbb{N}\}$ be a countable base for the Scott topology on D, consisting of Scott-open filters [1]. The map

$$p(x,y) := 1 - \sum_{\{n\in\mathbb{N}: x,y\in U_n\}} \frac{1}{2^{n+1}},$$

is a Scott-continuous partial metric on D. Indeed,

$$\begin{aligned} p(x,y) &= 1 - \sum_{\{n: x,y\in U_n\}} \frac{1}{2^{n+1}} \\ &= \bigsqcup\{1 - \sum_{\{n: z\in U_n\}} \frac{1}{2^{n+1}} \mid z \ll x,y\} \\ &= \bigsqcup\{\mu z \mid z \ll x,y\} \\ &= d_\mu(x,y), \end{aligned}$$

where μ is a measurement with $\mu \longrightarrow \sigma_D$ given by Example 2.5 and d_μ is the associated distance map. Note that because every U_n for $n \in \mathbb{N}$ is a filter, the condition $x, y \in U_n \Rightarrow \exists z \in U_n.\ z \ll x, y$ holds and the second equality above is indeed correct.

Now, we will check the partial metric axioms for p. The condition $p(x,y) \geq 0$ for all $x, y \in D$ and symmetry follow straight from the definition. T_0 axiom for p holds by Corollary 4.5. For $\Delta^\sharp$: take any $x, y, z \in P$. Notice that the inequality is equivalent to:

$$\sum_{\{n: x,z\in U_n\}} \frac{1}{2^{n+1}} + \sum_{\{n: y,z\in U_n\}} \frac{1}{2^{n+1}} \leq \sum_{\{n: x,y\in U_n\}} \frac{1}{2^{n+1}} + \sum_{\{n: z\in U_n\}} \frac{1}{2^{n+1}}.$$

We need to distinguish three cases where an open set $U_k, k \in \mathbb{N}$ is counted in both sums and in one of the sums on the left-hand side. But in every case every index k, which contributes to the sums on the left-hand side also contributes to the sums on the right-hand side. Hence, the inequality is proved.

Agreement of orders, $\sqsubseteq_p = \sqsubseteq_D$, is established by Theorem 4.3.

The partial metric is stable by the remark following Definition 3.3. Theorem 3.4 gives that the partial metric topology is the Scott topology of the induced order $\sqsubseteq_p$ and so the order on D. □

Finally, it is easy to check that the associated quasi-metric which induces the same order and topology is given by

$$q(x,y) = 1 - \sum_{\{n: x\in U_n \Rightarrow y\in U_n\}} \frac{1}{2^{n+1}}$$

and is weighted by μ.

Acknowledgement

The author wishes to thank Achim Jung and Keye Martin for their valuable criticisms and Dagmara Bogucka for her support and friendship.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] A. Edalat and R. Heckmann. A computational model for metric spaces. *Theoretical Computer Science*, 193:53–73, 1998.

[3] A.H. Frink. Distance functions and the metrization problem. *Bulletin of the American Mathematical Society*, 43:133–142, 1937.

[4] Reinhold Heckmann. Approximation of metric spaces by partial metric spaces. *Applied Categorical Structures*, 7:71–83, 1999.

[5] Keye Martin. *A Foundation for Computation.* PhD thesis, Department of Mathematics, Tulane University, New Orleans, LA 70118, 2000.

[6] Steve G. Matthews. Partial metric topology. In *Proceedings of the 8th Summer Conference on Topology and Its Application*, volume 728, pages 176–185, 1992.

[7] Simon J. O'Neill. Partial metrics, valuations and domain theory. Research Report CS-RR-293, Department of Computer Science, University of Warwick, Coventry, UK, October 1995.

[8] Philipp Sünderhauf. Sobriety in terms of nets. *Applied Categorical Structures*, 8:649–653, 2000.

Recent BRICS Notes Series Publications

NS-01-2 **Stephen Brookes and Michael Mislove, editors.** ***Preliminary Proceedings of the 17th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS '01,*** **(Aarhus, Denmark, May 24–27, 2001), May 2001. viii+279 pp.**

NS-01-1 **Nils Klarlund and Anders Møller.** ***MONA Version 1.4 — User Manual*****. January 2001. 83 pp.**

NS-00-8 **Anders Møller and Michael I. Schwartzbach.** ***The XML Revolution*****. December 2000. 149 pp.**

NS-00-7 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***Document Structure Description 1.0*****. December 2000. 40 pp.**

NS-00-6 **Peter D. Mosses and Hermano Perrelli de Moura, editors.** ***Proceedings of the Third International Workshop on Action Semantics, AS 2000,*** **(Recife, Brazil, May 15–16, 2000), August 2000. viii+148 pp.**

NS-00-5 **Claus Brabrand.** ***<bigwig> Version 1.3 — Tutorial*****. September 2000. ii+92 pp.**

NS-00-4 **Claus Brabrand.** ***<bigwig> Version 1.3 — Reference Manual*****. September 2000. ii+56 pp.**

NS-00-3 **Patrick Cousot, Eric Goubault, Jeremy Gunawardena, Maurice Herlihy, Martin Raussen, and Vladimiro Sassone, editors.** ***Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency Theory, GETCO '00,*** **(State College, USA, August 21, 2000), August 2000. vi+116 pp.**

NS-00-2 **Luca Aceto and Björn Victor, editors.** ***Preliminary Proceedings of the 7th International Workshop on Expressiveness in Concurrency, EXPRESS '00,*** **(State College, Pennsylvania, USA, August 21, 2000), August 2000. vi+130 pp.**

NS-00-1 **Bernd Gärtner.** ***Randomization and Abstraction — Useful Tools for Optimization*****. February 2000. 106 pp.**

NS-99-3 **Peter D. Mosses and David A. Watt, editors.** ***Proceedings of the Second International Workshop on Action Semantics, AS '99,*** **(Amsterdam, The Netherlands, March 21, 1999), May 1999. iv+172 pp.**