A Fully-Abstract Model for the π-calculus

(Extended Abstract)

M.P. Fiore*
LFCS, Univ. of Edinburgh
JCMB, The King's Buildings
Edinburgh EH9 3JZ, UK
mf@dcs.ed.ac.uk

E. Moggi†
DISI, Univ. di Genova
v. Dodecaneso 35
16146 Genova, Italy
moggi@disi.unige.it

D. Sangiorgi‡
INRIA, Sophia Antipolis
2004 r. des Lucioles
06902, Sophia Antipolis, France
davide@cma.cma.fr

Abstract

This paper provides both a fully abstract (domain-theoretic) model for the π-calculus and a universal (set-theoretic) model for the finite π-calculus with respect to strong late bisimulation and congruence. This is done by: considering categorical models, defining a metalanguage for these models, and translating the π-calculus into the metalanguage. A technical novelty of our approach is an abstract proof of full abstraction: The result on full abstraction for the finite π-calculus in the set-theoretic model is axiomatically extended to the whole π-calculus with respect to the domain-theoretic interpretation. In this proof, a central role is played by the description of non-determinism as a free construction and by the equational theory of the metalanguage.

Introduction

The π-calculus [16] is a process algebra for communicating processes with a dynamically-changing topology. *Processes* (or *agents*) interact with each other by exchanging *names*. A name can be *private* (i.e. *local*) to a process, which, however, may decide to export the name thus accepting to share it with other processes. Communication of private names is the main difference between π-calculus and its predecessor CCS, and makes the calculus very expressive. *Late bisimulation* is the operational equivalence on π-calculus processes analised in [16]. It is preserved by all operators of the calculus except for input. *Late congruence* is the induced congruence. Roughly, in late bisimulation free names of processes are viewed as constants, whereas in late congruence they are viewed as free variables and hence can be freely instantiated.

This paper substantiates the claim that "the π-calculus is CCS with local channels". Indeed, we construct a model for the π-calculus combining techniques used for modelling CCS and local variables in the light of an abstract approach to denotational semantics; notably: *powerdomains* as free algebras, *functor categories*, and a kind of *monadic metalanguage*. Powerdomains as free algebras were introduced in [12] for modelling the bounded non-determinism of a parallel imperative language. *Functor categories* were used in [18] to model local variables in Algol-like languages. *Monads* were proposed as a tool for structuring denotational semantics [17, 8].

Using monads we can exhibit the semantics for the π-calculus as a special case of a general construction, which can be instantiated to get semantics for calculi ranging from pure CCS to value-passing CCS to the polyadic π-calculus. This uniform treatment highlights the intrinsic differences and similarities among these calculi. Here, for brevity, we will only present a model for the π-calculus. The model captures essential properties of the role of names in the π-calculus: for instance, that the identity of names does not affect the behaviour of a process and that equality and inequality conditions on names may affect process bisimilarity.

We give a denotational semantics for the π-calculus by: considering categorical models, defining a metalanguage for these models, and translating the π-calculus into the metalanguage. The metalanguage is a simply typed λ-calculus with sums, operations for non-determinism and dynamic allocation, base types for names and agents, and recursion (over exponents of

*Supported by EPSRC grant RR29300 and HCM ERB-CHRXCT 92-0046.
†Supported by ESPRIT BRA CLICS-II, HCM EXPRESS and SCIENCE ERBSC1*CT920795.
‡Supported by HCM EXPRESS and ESPRIT BRA CONFER.

the type of agents). It has an interpretation in a standard *domain-theoretic* model given by a functor category over **Cpo** (the category of cpos —posets closed under lubs of ω-chains possibly without bottom element— and continuous functions) equipped with a powerdomain monad. This model is shown to provide a fully-abstract denotational semantics for the π-calculus with respect to strong late bisimulation and congruence. Indeed, the model also provides fully abstract semantics for *bisimulations under a distinction* [16] and *bisimulations under a constraint* (Section 3), which are bisimulation relations in between late bisimulation and late congruence.

The metalanguage without recursion has an interpretation in a *set-theoretic* model given by a functor category over **Set** (the category of sets and functions) equipped with the free-semilattice monad. This model is in bijective correspondence with certain canonical normal forms of π-calculus processes and provides a *universal* denotational semantics for the finite π-calculus with respect to late bisimulation and congruence.

An important aspect of the metalanguage is its associated equational theory which permits reasoning about the denotational semantics. For example, to validate laws on π-calculus processes we first validate analogous laws between terms of the metalanguage using its equational theory, and then infer the original laws by compositionality of the denotational interpretation. Also, the denotational semantics implicitly defines a model of *synchronisation trees* for the π-calculus under late bisimulation (i.e. an abstract notion of *late transition system*) and process-like operations on these, whose laws can be established using the equational theory of the metalanguage.

A novelty of our approach is an *abstract proof* of full abstraction. The result on full abstraction for canonical normal forms in the set-theoretic model is axiomatically extended to the whole calculus with respect to the domain-theoretic interpretation. This is done by relating the free-semilattice monad and the powerdomain monad by means of their universal properties, and by exploiting the equational theory of the metalanguage. As a technical benefit, an explicit description of the powerdomain is not needed and we can work with cpos instead of bifinite domains.

Related work. In [13] and [10], Hennessy and Plotkin constructed *term models* for CCS-like languages (where actions are pure synchronisations). Later, Abramsky [1] gave a denotational semantics for SCCS using a domain of synchronisation trees defined as the initial solution in **SFP** (the category of bifinite domains and continuous functions) of a domain equation involving Plotkin's powerdomain. Further, he provided two full abstraction results: one for finite SCCS with respect to *strong partial bisimulation* and another one for the whole SCCS with respect to the *finitely observable part of strong bisimulation.* More recently, Aceto and Ingólfsdóttir [4], using the same domain of synchronisation trees, have generalised Abramsky's results to a class of CCS-like languages (those described in the compact GSOS format). Outside the realm of domain theory, in [21], Rutten provides fully abstract semantics with respect to *strong bisimulation* in a non-well-founded set (specified by a recursive equation) for a different class of CCS-like languages (a subclass of those described in the tyft/txft format).

Independently from us, other researchers have been working on a denotational semantics for the π-calculus. Stark has given a denotational semantics in a functor category over **SFP**. His interpretation coincides with ours but, as he has not extracted a metalanguage from the model, his constructions are concrete and do not identify the uniformities that our axiomatic approach highlights. Abramsky and Meredith are investigating a model for the π-calculus based on non-well-founded sets. Details of this work are not known to us.

Organisation of the paper. In Section 1 we review the syntax and the operational semantics of the π-calculus. In Section 2 we present our denotational model and in Section 3 the full abstraction results. Finally, in Section 4 we discuss conclusions and directions for future research.

1 π-calculus

We review the syntax and operational semantics of the π-calculus. $\mathcal{N}$ is the countably infinite set of all names, ranged over by a, b, c, d. The class $\mathcal{P}r$ of processes is built from the operators of inaction, sum, matching, mismatching, prefixing, restriction, parallel composition and guarded replication; a prefix can be an input, a free output, a bound output or an silent prefix:

Definition 1.1 (π-calculus, concrete syntax)

$$\begin{array}{rcl} P & := & \mathbf{0} \;\big|\; P+P \;\big|\; [a=b]P \;\big|\; [a\neq b]P \;\big|\; \alpha.P \;\big| \\ & & \boldsymbol{\nu} a\, P \;\big|\; P \mid P \;\big|\; !\alpha.P \\ \alpha & := & a(b) \;\big|\; \overline{a}b \;\big|\; \overline{a}(b) \;\big|\; \tau \end{array}$$

A process is finite *when it uses only the operators in the first row.*

Remark. It is possible to extend the definition of finite processes to include the parallel composition and restriction operators, but this does not add new process behaviours and complicates some of our technical developments.

We refer to [16] for detailed discussions on the operators of the language. We remind that a matching $[a=b]P$ means "if $a=b$ then P", that a mismatching $[a \neq b]P$ means "if $a \neq b$ then P", and that a replication $!\alpha.\, P$ represents a countably infinite number of copies of $\alpha.\, P$ in parallel. Guarded replications enjoy simpler algebraic laws than plain replication $!P$. The restriction to guarded replications does not affect expressiveness, since every plain replication can be rewritten in terms of guarded replications, up to strong late congruence [23]. A bound output $\overline{a}(b)$ represents the output of a private name b at a; one can think of $\overline{a}(b).\, P$ as an abbreviation for $\boldsymbol{\nu} b\, \overline{a}b.\, P$. In $a(b).\, P$, $\boldsymbol{\nu} b\, P$, and $\overline{a}(b).\, P$ all free occurrences of name b in P are bound. *Free names* (*fn*) and *bound names* (*bn*) of processes and prefixes, name substitution, and alpha conversion are defined as expected.

Terminology and notation. We use V to range over finite lists of *distinct names* and $\mathcal{P}r_V$ for the set of processes with free names in V. We write $[a \notin a_0, \ldots, a_{n-1}]P$ as abbreviation for $[a \neq a_0] \ldots [a \neq a_{n-1}]P$. For a finite list without repetitions $I = i_0, \ldots, i_{n-1}$, we write $\sum_{i \in I} P_i$ as an abbreviation for $P_{i_0} + \ldots + P_{i_{n-1}}$. We use the symbol $\equiv$ for syntactic identity, and $\equiv_\alpha$ for syntactic identity modulo alpha conversion. We assign parallel composition and sum the lowest precedence among the operators.

Table 1 shows the transitional rules for the calculus. We have omitted the symmetric of rules `sum1`, `par1`, `com1`, and `close1`. Note that there are four kinds of transitions, corresponding to the four kinds of prefixes in the syntax.

Definition 1.2 (Late bisimulation [16]) Late bisimulation *is the largest symmetric relation $\sim$ on processes such that $P \sim Q$ implies*

1. *Whenever $P \stackrel{a(b)}{\longrightarrow} P'$ and $b \notin \mathrm{fn}(Q)$, then Q' exists such that $Q \stackrel{a(b)}{\longrightarrow} Q'$ and for all names c, $P'\{c/b\} \sim Q'\{c/b\}$.*
2. *Whenever $P \stackrel{\alpha}{\rightarrow} P'$ for $\alpha \equiv \overline{a}b$ or $\alpha \equiv \tau$, then Q' exists such that $Q \stackrel{\alpha}{\rightarrow} Q'$ and $P' \sim Q'$.*
3. *Whenever $P \stackrel{\overline{a}(b)}{\longrightarrow} P'$ and $b \notin \mathrm{fn}(Q)$, then Q' exists such that $Q \stackrel{\overline{a}(b)}{\longrightarrow} Q'$ and $P' \sim Q'$.*

Late bisimilarity is preserved by all operators except input prefix. The induced congruence, called *late congruence*, is denoted by $\sim^{\mathrm{c}}$ and can be defined thus:

Definition 1.3 (Late congruence [16]) *Two processes P, Q are* late congruent, *written $P \sim^{\mathrm{c}} Q$, if $P\sigma \sim Q\sigma$ for all name substitutions σ.*

In the semantics of Section 2, $\sim$ will correspond to the *closed* interpretation —where free names of processes are treated as constants— and $\sim^{\mathrm{c}}$ will correspond to the *open* interpretation —where free names are treated as free variables.

To have a simpler *axiomatic* proof of full abstraction (Section 3), we will make use of the auxiliary operators of *left merge* and *synchronisation* [6], written $\rfloor$ and $\|$, respectively. They are related to parallel composition by the law:

$$\texttt{Par} \quad P \mid Q = P\rfloor Q + Q\rfloor P + P \parallel Q.$$

In this law, left merge accounts for the behaviour of $P \mid Q$ given by rules `par1-2`, whereas synchronisation accounts for that given by `com1-2` and `close1-2`.

Definition 1.4 ($\mathcal{CNF}_V$) *Given a finite list of distinct names V, the set of processes $\mathcal{NF}_V$, called the* V-normal forms, *is defined from the grammar below, where it is assumed that $a, b \in V$ and that $c \notin V$:*

$$P_V := P_V + P_V \;\Big|\; \mathbf{0} \;\Big|\; \overline{a}b.\, P_V \;\Big|\; \overline{a}(c).\, P_{V,c} \;\Big|$$
$$\tau.\, P_V \;\Big|\; a(c).\Big(\sum_{a \in V}[c=a]P_V + [c \notin V]P_{V,c}\Big)$$

$\mathcal{CNF}_V$, called the canonical V-normal forms, *is the set of canonical representatives of the equivalence classes obtained by quotienting $\mathcal{NF}_V$ by $\sim$. Formally, $\mathcal{CNF}_V$ is defined from the grammar for $\mathcal{NF}_V$ by imposing a canonical choice for bound names and requiring that, in any term of the form $\sum_i P_i$, processes P_i are ordered lexicographically and are not duplicated (i.e. $P_i \equiv P_j$ implies $i = j$).*

2 Denotational semantics

The denotation of a process is given relative to the names which are free (or visible) in it. Thus, our model is a type A of agents which *varies over stages*; intuitively, the number of free names available for interaction. That is, the type A consists of the following data: for every natural number n a type $A(n)$ of 'processes with at most n free names' and for every injective substitution $\iota : n \hookrightarrow m$ of n names into m names a mapping $A(n) \xrightarrow{A(\iota)} A(m)$ which allows us to view every process with n free names as a process with m free names. Of course, these mappings cannot be arbitrary: we expect that $A(\mathrm{id}_n) = \mathrm{id}_{A(n)}$ as the substitution $\mathrm{id}_n : n \hookrightarrow n$ produces no renaming; whilst for

`alpha`	$\dfrac{P \equiv_\alpha P' \quad P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''}$	`pre`	$\alpha.P \xrightarrow{\alpha} P$	`sum1`	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$
`rep`	$!\alpha.P \xrightarrow{\alpha} P \mid !\alpha.P \quad \text{if } \mathrm{bn}(\alpha) \notin \mathrm{fn}(!\alpha.P)$	`par1`	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \text{ if } \mathrm{bn}(\alpha) \notin \mathrm{fn}(Q)$		
`com1`	$\dfrac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}c} Q'}{P \mid Q \xrightarrow{\tau} P'\{c/b\} \mid Q'}$	`close1`	$\dfrac{P \xrightarrow{a(b)} P' \quad Q \xrightarrow{\overline{a}(b)} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu} b\,(P' \mid Q')}$	`open`	$\dfrac{P \xrightarrow{\overline{a}b} P'}{\boldsymbol{\nu} b\, P \xrightarrow{\overline{a}(b)} P'} \text{ if } a \not\equiv b$
`res`	$\dfrac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu} a\, P \xrightarrow{\alpha} \boldsymbol{\nu} a\, P'}\ a \notin \mathrm{names}(\alpha)$	`mismch`	$\dfrac{P \xrightarrow{\alpha} P'}{[a \neq b]P \xrightarrow{\alpha} P'} \text{ if } a \not\equiv b$	`mch`	$\dfrac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'}$

Table 1. The transition system for the π-calculus

injective substitutions $\iota : i \hookrightarrow j$ and $\kappa : j \hookrightarrow k$ we expect that the mapping $A(i) \xrightarrow{A(\kappa\iota)} A(j)$ induced by $\kappa\iota$ decomposes as the mappings $A(i) \xrightarrow{A(\iota)} A(j) \xrightarrow{A(\kappa)} A(k)$ induced by ι and κ. All this amounts to saying that A is a functor from a category $\mathcal{I}$ (of injective maps between finite cardinals) to a category $\mathcal{C}$ (of meanings).

Henceforth we will take the viewpoint that A is a type in the functor category $\mathcal{C}^{\mathcal{I}}$ whose objects are functors from $\mathcal{I}$ to $\mathcal{C}$ and whose arrows are natural transformations between such functors. (Recall that a natural transformation $\varphi : X \to Y$ between functors $X, Y : \mathcal{I} \to \mathcal{C}$ is given by a family $\{\varphi_n : X(n) \to Y(n)\}_{n \in |\mathcal{I}|}$ of morphisms in $\mathcal{C}$ such that for all injective maps $\iota : n \hookrightarrow m$, $Y(\iota) \circ \varphi_n = \varphi_m \circ X(\iota)$.) A process with no free names will have a denotation as a global element of A (i.e. a natural transformation $1 \stackrel{\cdot}{\to} A$) or, equivalently, as an element of $A(0)$ because, by Yoneda, every $p : A$ is uniquely determined by $p_0 : A(0)$ as $p_n = A(0 \hookrightarrow n)(p_0) : A(n)$. Thus, the naturality condition imposes a *uniform* interpretation at all stages. To interpret arbitrary processes we will introduce a type N of names and assign to a process with n free names a denotation as a natural transformation $N^n \stackrel{\cdot}{\to} A$. As before the naturality condition imposes a uniform interpretation which, for example, will account for the irrelevance of the identity of names.

It is possible to *axiomatise* the structure needed from our category $\mathcal{C}$ of meanings in order to support the denotational semantics, in particular $\mathcal{C}$ must be a **Cpo**-enriched bicartesian closed category (**Cpo**-biCCC). We will not give an explicit axiomatisation of $\mathcal{C}$ in this extended abstract, indeed we will only be concerned with two categories $\mathcal{C}$ of meanings; viz. **Cpo** and **Set**. In $\mathbf{Cpo}^{\mathcal{I}}$ we will obtain a domain-theoretic model for the whole π-calculus; whilst in $\mathbf{Set}^{\mathcal{I}}$ we will obtain a set-theoretic model for the finite processes.

Our denotational semantics is given by translation into a *metalanguage*. The metalanguage is a type theory (with an associated equational theory) which is an internal language for the part of $\mathcal{C}^{\mathcal{I}}$ we are interested in. In this way we can perform constructions and definitions in $\mathcal{C}^{\mathcal{I}}$ and prove properties (e.g. about the equality of certain morphisms) without the need of looking at explicit descriptions. The core of the metalanguage is a simply typed λ-calculus $(1, \times, \Rightarrow)$ with sums $(+)$, type constructors for non-determinism and dynamic allocation of names, and base types for names (N) and agents (A). In addition, we have operators for recursion (over exponents of the type of agents): in the domain-theoretic interpretation *arbitrary* recursive definitions are possible; whilst in the set-theoretic interpretation we only consider finite agents.

In the remainder of the section we define the constructors for modelling non-determinism and dynamic allocation; then we introduce the objects of names and agents; finally we present the open and closed interpretations, respectively corresponding to late congruence and bisimulation.

Notation. In expressions of the metalanguage we adopt the following conventions: e is an expression, x, y are variables, f, g are variables of functional type, p, q are variables of computational type.

2.1 Non-determinism

We introduce the *power* type constructor $\mathbb{P}$ for modelling non-determinism. It has associated operations $val_X : X \Rightarrow \mathbb{P}\,X$ and $let_{X,Y} : (X \Rightarrow \mathbb{P}\,Y) \times \mathbb{P}\,X \Rightarrow \mathbb{P}\,Y$ subject to the laws for a *commutative* monad —e.g. $let(\lambda x_1.\, let(\lambda x_2.\, e, p_2), p_1) = let(\lambda x_2.\, let(\lambda x_1.\, e, p_1), p_2)$; and operations $0 : \mathbb{P}\,X$ and $\cup : \mathbb{P}\,X \times \mathbb{P}\,X \Rightarrow \mathbb{P}\,X$ making $(\mathbb{P}\,X, 0, \cup)$ into a semilattice. Moreover, the operations for non-determinism interact with *let* according to the following laws: $let(f, 0) = 0$ and $let(f, \cup(p_1, p_2)) = \cup(let(f, p_1), let(f, p_2))$. To understand these laws think that, roughly, $let(f, p)$ is '$\bigcup\{f(x) \mid x \in p\}$'.

In general these power types arise as free constructions by considering left adjoints to forgetful functors from a category of non-deterministic computations to a category of values (see [12, 5]). Two such constructions that will be used later are:

- **The free-semilattice monad.** Let $\mathcal{S}$ be a **Cpo**-cartesian category, and let $\mathbf{SL}(\mathcal{S})$ be the **Cpo**-category of semilattices in $\mathcal{S}$ and homomorphisms.

  The *free-semilattice monad* on $\mathcal{S}$ is the **Cpo**-monad induced by the **Cpo**-adjunction $\mathbf{SL}(\mathcal{S}) \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathcal{S}$ whenever it exists. When $\mathcal{S} = \mathbf{Set}$ the free-semilattice monad is the *finite* powerset functor equipped with the singleton and the *big* union.

- **Abramsky's powerdomain monad.** Let $\mathcal{D}$ be a **Cpo**-cartesian category. We define the category $\mathbf{ND}(\mathcal{D})$ of non-deterministic computations over $\mathcal{D}$ as the **Cpo**-category with objects $(D, \bot, 0, \cup)$, where $\bot : D$ is the *least element* of D and $(D, 0, \cup)$ is a semilattice in $\mathcal{D}$; arrows are *strict* semilattice homomorphisms.

  Abramsky's Powerdomain monad on $\mathcal{D}$ is the **Cpo**-monad on $\mathcal{D}$ induced by the **Cpo**-adjunction $\mathbf{ND}(\mathcal{D}) \underset{\longrightarrow}{\overset{\longleftarrow}{\perp}} \mathcal{D}$ whenever it exists.

  When $\mathcal{D} = \mathbf{Cpo}$ Abramsky's powerdomain monad exists; see [12, 3].

Below we will be concerned with commutative monads for non-determinism arising as above over functor categories. The following observation incorporates them in our setting:

Proposition 2.1 *If $\mathcal{C}$ admits Abramsky's powerdomain monad (resp. the free-semilattice monad) then so does $\mathcal{C}^{\mathcal{W}}$ for every small category $\mathcal{W}$ and it is given pointwise. That is, writing $\mathbb{P}$ and $\mathbb{P}^{\mathcal{W}}$ for the monad on $\mathcal{C}$ and $\mathcal{C}^{\mathcal{W}}$ respectively, we have $\mathbb{P}^{\mathcal{W}}(X)(w) = \mathbb{P}(X(w))$.*

2.2 Dynamic allocation

We introduce the type constructor δ for modelling dynamic allocation of names. It has associated operations $\delta_{X,Y} : (X \Rightarrow Y) \Rightarrow \delta X \Rightarrow \delta Y$, $\mathsf{up}_X : X \Rightarrow \delta X$, and $\mathsf{swap}_X : \delta^2 X \Rightarrow \delta^2 X$; which are the internal version of categorical structure on $\mathcal{C}^{\mathcal{I}}$ (made explicit below) induced by $\mathcal{I}$. Intuitively δX stands for the type X when one *new* name is available. With this in mind,

- $\mathsf{up}_{X,n}$ is the canonical *coercion* mapping an element of X at stage n to the *same* element at stage $n+1$;
- $\mathsf{swap}_{X,n}$ is an involution mapping an element of X at stage $n+2$ to the element of the same type where the last two names in $n+2$ have been swapped.

To explain how such structure is obtained, we consider an alternative characterisation of the category $\mathcal{I}$ of finite cardinals $n = \{0, \ldots, n-1\}$ and injective maps:

> $\mathcal{I}$ is the strict monoidal category $(\mathcal{I}, 0, +)$ freely generated from one object 1 and morphisms $up : 0 \to 1$ and $swap : 2 \to 2$ (here $2 = 1{+}1$) such that $swap \circ swap = \mathrm{id}_2 : 2 \to 2$, $(up + \mathrm{id}_1) \circ up = (\mathrm{id}_1 + up) \circ up : 0 \to 2$ and $swap \circ (up + \mathrm{id}_1) = (\mathrm{id}_1 + up) : 1 \to 2$.

$\delta : \mathcal{C}^{\mathcal{I}} \to \mathcal{C}^{\mathcal{I}}, \mathsf{up} : \delta^0 \dot{\to} \delta, \mathsf{swap} : \delta^2 \dot{\to} \delta^2$ are defined in terms of the generating data $(1, up, swap)$ for $\mathcal{I}$

- $(\delta X)(n) \stackrel{\mathrm{def}}{=} X(n+1)$
- $\mathsf{up}_{X,n} \stackrel{\mathrm{def}}{=} X(n + up) : X(n) \to X(n+1)$
- $\mathsf{swap}_{X,n} \stackrel{\mathrm{def}}{=} X(n + swap) : X(n+2) \to X(n+2)$

and inherit the equational properties of up and $swap$; i.e. $\mathsf{swap}_X \circ \mathsf{swap}_X = \mathrm{id}_{\delta^2 X}$, $\delta(\mathsf{up}_X) \circ \mathsf{up}_X = \mathsf{up}_{\delta X} \circ \mathsf{up}_X$, and $\mathsf{swap}_X \circ \delta(\mathsf{up}_X) = \mathsf{up}_{\delta X}$. Moreover, δ, up, and swap preserve anything *defined pointwise*; such as limits and colimits, and Abramsky's powerdomain monad with its associated structure ($\bot$, 0, and $\cup$).

The functor δ has a tensorial strength $\mathsf{up}_X \times \mathrm{id}_{\delta Y} : X \times \delta Y \to \delta(X \times Y)$ —here we use that $\delta(X \times Y) = \delta(X) \times \delta(Y)$— which, as usual in the presence of exponentials, allow us to internalise the action of δ on morphisms as a map $\delta_{X,Y} : (X \Rightarrow Y) \Rightarrow \delta X \Rightarrow \delta Y$.

2.3 Names and agents

We introduce the objects N of names and A of agents in $\mathcal{C}^{\mathcal{I}}$ together with their associated operations.

Names. The type of names N is defined as the *inclusion* of $\mathcal{I}$ into the biCCC $\mathcal{C}$; i.e. $N(n) \stackrel{\mathrm{def}}{=} n$ (on the rhs n is the coproduct in $\mathcal{C}$ of n copies of the terminal object). N admits a decidable equality $\mathsf{eq} : N \times N \Rightarrow 2$ (here $2 = 1 + 1$) and satisfies the following properties which are used for interpreting the π-calculus and establishing full-abstraction:

- There is a global element $\mathsf{new} : \delta N$ (viz. $\mathsf{new}_n \stackrel{\mathrm{def}}{=} n$) making $N \xrightarrow{\mathsf{up}_N} \delta N \xleftarrow{\mathsf{new}} 1$ into a coproduct diagram in $\mathcal{C}^{\mathcal{I}}$. Thus we can do case analysis on δN;

to improve readability we write $\left[\begin{array}{l}\lambda x : N.\, e_1 \\ \lambda t : 1.\, e_2\end{array}\right] a$ for $\mathsf{case}\ a\ \mathsf{of}\ \mathsf{up}_N(x)\ \texttt{=>} e_1\ \mathsf{or}\ \mathsf{new}\ \texttt{=>} e_2$. (We use a similar notation for doing case analysis on coproducts.)

- swap_N is an involution that *swaps* $\delta(\mathsf{new})$ and $\delta\ (\mathsf{up}_N)\ \mathsf{new}$; that is, $\mathsf{swap}_N(\delta\ \mathsf{new}) = \delta\ (\mathsf{up}_N)\ \mathsf{new}$.
- For $h : N \times N \Rightarrow X$,

$$\begin{aligned}&\mathsf{swap}_X(\delta\ (\lambda x : N.\, \delta\ (\lambda y : N.\, h(x,y))\ \mathsf{new})\ \mathsf{new}) \\ &\quad = \delta\ (\lambda y : N.\, \delta\ (\lambda x : N.\, h(x,y))\ \mathsf{new})\ \mathsf{new}.\end{aligned} \tag{1}$$

We will also use the following proposition stating that an element of $N \Rightarrow X$ at stage n is determined by n elements of X at stage n together with an element of δX at stage n.

Proposition 2.2 *For any biCCC $\mathcal{K}$, the exponential $N{\Rightarrow}X$ in $\mathcal{K}^{\mathcal{I}}$ exists and is given by $(N{\Rightarrow}X)(n) = X(n)^n \times X(n+1)$.*

Remark. In particular, the natural transformation interpreting $f : N \Rightarrow X \vdash \delta\ f\ \mathsf{new} : \delta X$ is, at stage n, $\pi_{n+1} : X(n)^n \times X(n+1) \to X(n+1)$.

Agents. The type of agents A is defined as $\mu X.\, \mathbb{P}\ (HX)$ where

- $HX \stackrel{\text{def}}{=} N\times(N{\Rightarrow}X) + N\times N\times X + N\times\delta X + X$; each summand respectively giving meaning to inputs, free outputs, bound outputs, and internal actions of processes; and
- in the set-theoretic interpretation $\mathbb{P}$ is the free-semilattice monad on $\mathbf{Set}^{\mathcal{I}}$, whilst in the domain-theoretic interpretation it is Abramsky's powerdomain monad on $\mathbf{Cpo}^{\mathcal{I}}$. Here one can apply the standard techniques for solving recursive domain equations —see [25] and [9, Chapter 7].

By Propositions 2.1 and 2.2, and results on the solution of domain equations, we can give a concrete description of A on objects as the initial solution to the following system of equations in $\mathcal{C}$ (where $\mathbb{P}$ below is the appropriate power monad on $\mathcal{C}$) with $n \in |\,\mathcal{I}\,|$

$$X_n = \mathbb{P}\ (n{\times}{X_n}^n{\times}X_{n+1} + n{\times}n{\times}X_n + n{\times}X_{n+1} + X_n).$$

We introduce some auxiliary notation (related to a suitable monad transformer —see [8]) which simplifies the description of the denotational semantics and helps in validating equational properties:

- $\mathsf{S} : HA \Rightarrow A$ is given (up to the isomorphism $\mathbb{P}\ HA \cong A$) by $\mathsf{S}(x) = val(x)$.

  We write S_i $(i = 1, 4)$ for the i^{th} component of S; so that $\mathsf{S} = [\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3, \mathsf{S}_4]$.
- $\mathsf{C} : (HA{\Rightarrow}A){\times}A \Rightarrow A$ is given (up to the isomorphism $\mathbb{P}\ HA \cong A$) by $\mathsf{C}(f,p) = let(f,p)$.

  We write $\mathsf{CC} : (HA \times HA \Rightarrow A) \times A \times A \Rightarrow A$ for C iterated twice; i.e. $\mathsf{CC}(k, p_1, p_2) = \mathsf{C}(\lambda x.\, \mathsf{C}(\lambda y.\, k(x,y), p_2), p_1)$.

S and C inherit the properties of val and let, and since $\mathbb{P}$ is commutative we have

$$\mathsf{CC}(K, p_1, p_2) \quad = \quad \mathsf{CC}(K \circ \langle \pi_2, \pi_1 \rangle, p_2, p_1). \tag{2}$$

From now on we will consistently treat the isomorphism $\mathbb{P}\ HA \cong A$ as an equality.

Finally we introduce the operation $\mathsf{R} : \delta A \Rightarrow A$, which maps, at stage n, a process with $n+1$ free names to one with n free names by *restricting* on the last allocated name. The definition of R, given in Table 2, relies on the type equalities $\delta A = \mathbb{P}\ (\delta N{\times}\delta(N{\Rightarrow}A) + \delta N{\times}\delta N{\times}\delta A + \delta N{\times}\delta^2 A + \delta A)$ and $\delta N = N+1$, and inspects whether each action capability involves the last allocated name (i.e. the new one) or not:

- In case of an input at a: if a is new, then we cancel the input capability, otherwise we allow the input and restrict on the continuation.
- In case of a free output at a of b: if a is new, then we cancel the output capability; if a is not new but b is, then we make the free output into a bound output; if neither a nor b are new, then we allow the output and restrict on the continuation.
- In case of a bound output at a: if a is new, then we cancel the output capability, otherwise we allow the output and restrict on the second last allocated name (*not* on the last allocated one as this is the one being allocated by the bound output).
- In case of an internal action: we allow the action and restrict on the continuation.

2.4 Interpretation

We give the interpretation of the term constructors for an *abstract syntax* of the π-calculus (c.f. Definition 1.1). The interpretation is defined by translation into the metalanguage of the previous subsections as follows:

$\mathsf{nil} : A$	deadlock
$\mathsf{sum} : A, A \to A$	non-deterministic choice

are given by the semilattice structure of $\mathbb{P}\ HA$.

$\mathsf{M} : N, N, A \to A$	matching
$\mathsf{MM} : N, N, A \to A$	mismatching

$$
\mathsf{R}(p) \stackrel{\text{def}}{=} let(\left[\begin{array}{l}
\lambda a:\delta N, f:\delta(N\Rightarrow A).\left[\begin{array}{l}\lambda a':N.\,\mathsf{S}_1(a',\lambda b:N.\,\mathsf{R}(\delta\ (\lambda f':N\Rightarrow A.\,f'b)\ f))\\ \lambda t:1.\,0\end{array}\right] a\\
\lambda a:\delta N, b:\delta N, p':\delta A.\left[\begin{array}{l}\lambda a':N.\left[\begin{array}{l}\lambda b':N.\,\mathsf{S}_2(a',b',\mathsf{R}\ p'),\\ \lambda t:1.\,\mathsf{S}_3(a',p')\end{array}\right] b\\ \lambda t:1.\,0\end{array}\right] a\\
\lambda a:\delta N, p':\delta^2 A.\left[\begin{array}{l}\lambda a':N.\,\mathsf{S}_3(a',\delta\ \mathsf{R}\ (\mathsf{swap}_A\ p'))\\ \lambda t:1.\,0\end{array}\right] a\\
\lambda p':\delta A.\,\mathsf{S}_4(\mathsf{R}\ p')
\end{array}\right], p)
$$

Table 2. $\mathsf{R} : \delta A \Rightarrow A$

are defined by case analysis using the decidable equality on the type of names.

$\mathsf{in} : N, (N \Rightarrow A) \to A$	input
$\mathsf{out} : N, N, A \to A$	output
$\mathsf{bout} : N, (N \Rightarrow A) \to A$	bound output
$\mathsf{tau} : A \to A$	internal action

correspond, except for bout, to components of the operation $\mathsf{S} : HA \Rightarrow A$; explicitly, $[\mathsf{in}, \mathsf{out}, \mathsf{S}_3, \mathsf{tau}] = \mathsf{S}$. Bounded output is given by $\mathsf{bout}(a, f) = \mathsf{S}_3(a, \delta\ f\ \mathsf{new})$ —see Proposition 2.2 and the remark after it.

$\mathsf{res} : (N \Rightarrow A) \to A$	local name

is given by $\mathsf{res}(f) = \mathsf{R}(\delta\ f\ \mathsf{new})$. (Note that the definition of res in terms of R mimics that of bout in terms of S_3.)

$\mathsf{par} : A, A \to A$	parallel composition
$\mathsf{lm} : A, A \to A$	left merge
$\mathsf{syn} : A, A \to A$	synchronisation

are defined by mutual recursion as follows:

$\mathsf{par}(p, q) = \mathsf{sum}(\mathsf{lm}(p, q), \mathsf{lm}(q, p), \mathsf{syn}(p, q), \mathsf{syn}(q, p))$.

$\mathsf{lm}(p, q) =$

$$
\mathsf{C}(\left[\begin{array}{l}
\lambda a:N, f:N\Rightarrow A.\,\mathsf{in}(a,\lambda b:N.\,\mathsf{par}(fb,q))\\
\lambda a:N, b:N, p':A.\,\mathsf{out}(a,b,\mathsf{par}(p',q))\\
\lambda a:N, p':\delta A.\,\mathsf{S}_3(a,\delta\ \mathsf{par}\ (p',\mathsf{up}\ q))\\
\lambda p':A.\,\mathsf{tau}(\mathsf{par}(p',q))
\end{array}\right], p).
$$

When two processes try to communicate there are 16 possibilities to consider, since each process may perform 4 possible actions. This explains the case analysis in the definition of syn below.

$\mathsf{syn}(p_1, p_2) = \mathsf{CC}(K, p_1, p_2)$; the cases covered by $K : (HA \times HA) \Rightarrow A$ are summarised in the *symmetric* Table 3 where

- $\mathsf{com} : N \times N \times A \times A \Rightarrow A$ is $\mathsf{com}(x, y, p, q) = \mathsf{M}(x, y, \mathsf{tau}(\mathsf{par}(p, q)))$, and
- $\mathsf{close} : N \times N \times \delta A \times \delta A \Rightarrow A$ is $\mathsf{close}(x, y, p', q') = \mathsf{M}(x, y, \mathsf{tau}(\mathsf{R}(\delta\ \mathsf{par}\ (p', q'))))$.

$!\mathsf{in} : N, (N \Rightarrow A) \to A$	replicated input
$!\mathsf{out} : N, N, A \to A$	replicated output
$!\mathsf{bout} : N, (N \Rightarrow A) \to A$	replicated bound output
$!\mathsf{tau} : A \to A$	replicated internal action

are defined by recursion (but not mutual recursion) as follows:

$!\mathsf{in}(a, f) = \mathsf{in}(a, \lambda b : N.\, \mathsf{par}(fb, !\mathsf{in}(a, f)))$.

$!\mathsf{out}(a, b, p) = \mathsf{out}(a, b, \mathsf{par}(p, !\mathsf{out}(a, b, p)))$.

$!\mathsf{bout}(a, f) = \mathsf{bout}(a, \lambda b : N.\, \mathsf{par}(fb, !\mathsf{bout}(a, f)))$.

$!\mathsf{tau}(p) = \mathsf{tau}(\mathsf{par}(p, !\mathsf{tau}(p)))$.

Open interpretation. Given the interpretation/translation of the above term constructors it is straightforward to extend it to names and well-formed processes following the paradigm of categorical logic. For a name $a \in V$, $\mathcal{O}[\![V \vdash a]\!]$ is the obvious projection $N^{|V|} \dot\to N$ in $\mathcal{C}^{\mathcal{I}}$. For a process $P \in \mathcal{P}r_V$, $\mathcal{O}[\![V \vdash P]\!]$ is a morphism $N^{|V|} \dot\to A$ in $\mathcal{C}^{\mathcal{I}}$ defined by induction on the structure of P. For example, $\mathcal{O}[\![V \vdash P_1 \mid P_2]\!] \stackrel{\text{def}}{=} \mathsf{par} \circ \langle \mathcal{O}[\![V \vdash P_1]\!], \mathcal{O}[\![V \vdash P_2]\!]\rangle$ and $\mathcal{O}[\![V \vdash a(b).\, P]\!] \stackrel{\text{def}}{=} \mathsf{in} \circ \langle \mathcal{O}[\![V \vdash a]\!], \lambda\, \mathcal{O}[\![V, b \vdash P]\!]\rangle$.

Closed interpretation. Bisimulation cannot correspond to the above interpretation, because the operational semantics considers free names of a process semantically different. In the denotational semantics this requirement can be captured smoothly by exploiting the functor category structure and adopting a *closed* interpretation such that for V with $|\, V \,| = n$, $\mathcal{C}[\![V \vdash a]\!] \in N(n) = n$ and $\mathcal{C}[\![V \vdash P]\!] \in A(n)$. Then the closed interpretation is defined in terms of the *open* interpretation: $\mathcal{C}[\![V \vdash \_\,]\!] \stackrel{\text{def}}{=} \mathcal{O}[\![V \vdash \_\,]\!]_n(0, \ldots, n-1)$. It

	$x_2 : N, f_2 : N \Rightarrow A$	$x_2 : N, y_2 : N, q_2 : A$	$x_2 : N, q'_2 : \delta A$	$q_2 : A$
$x_1 : N, f_1 : N \Rightarrow A$	nil	$\mathsf{com}(x_1, x_2, f_1(y_2), q_2)$	$\mathsf{close}(x_1, x_2, (\delta\ f_1\ \mathsf{new}), q'_2)$	nil
$x_1 : N, y_1 : N, q_1 : A$	$\mathsf{com}(x_2, x_1, f_2(y_1), q_1)$	nil	nil	nil
$x_1 : N, q'_1 : \delta A$	$\mathsf{close}(x_2, x_1, (\delta\ f_2\ \mathsf{new}), q'_1)$	nil	nil	nil
$q_1 : A$	nil	nil	nil	nil

Table 3. $K : (HA \times HA) \Rightarrow A$

is then possible to *prove* that the close interpretation is *compositional* on all constructs but input. For instance, for parallel composition: $\mathcal{C}[\![V \vdash P_1 \mid P_2]\!] = \mathsf{par}_{|V|}(\mathcal{C}[\![V \vdash P_1]\!], \mathcal{C}[\![V \vdash P_2]\!])$. The clause for input cannot be purely compositional because late bisimulation is not preserved by this operator but, for $V \equiv a_0, \ldots, a_{n-1}$, we have $\mathcal{C}[\![V \vdash a(b).P]\!] = \mathsf{in}_n(\mathcal{C}[\![V \vdash a]\!], \langle \mathcal{C}[\![V \vdash P\{a_i/b\}]\!]\rangle_{i\in n}, \mathcal{C}[\![V, b \vdash P]\!])$. A straightforward consequence of naturality is that the induced semantics is preserved by renaming; i.e. for $b \notin V, a$,

$$\mathcal{C}[\![V, a \vdash P]\!] = \mathcal{C}[\![V, b \vdash P\{b/a\}]\!] \qquad (3)$$

2.5 Examples of denotational validity

To see the denotational machinery at work, we validate the semantic counterpart of the following three laws for late congruence:

Syn1 $\quad P \parallel Q \;=\; Q \parallel P$

Syn2 $\quad x(u).P \parallel \overline{y}(v).Q =$
$\quad [x = y]\tau.\boldsymbol{\nu} z\,(P\{z/u\} \mid Q\{z/v\})$, if $z \notin \mathrm{fn}(P, Q)$

R1 $\quad \boldsymbol{\nu} x\, \overline{z}(y).P \;=\; \overline{z}(y).\boldsymbol{\nu} x\, P$, if $x \notin \{y, z\}$

(Syn1) For $p, q : A$, we have that $\mathsf{CC}(K, p, q) = \mathsf{CC}(K, q, p)$ by (2) because K is symmetric (i.e. $K = K \circ \langle \pi_2, \pi_1\rangle$). Hence $\mathsf{syn}(p, q) = \mathsf{syn}(q, p)$ and then also $\mathsf{par}(p, q) = \mathsf{sum}(\mathsf{lm}(p, q), \mathsf{lm}(q, p), \mathsf{syn}(p, q))$.

Notice below how in translating the laws into the metalanguage π-bindings become λ-bindings; whilst substitutions become applications.

(Syn2) For $x, y : N$ and $p, q : N \Rightarrow A$,

$$\begin{aligned}
&\mathsf{syn}(\mathsf{in}(x, p), \mathsf{bout}(y, q)) \\
=\;& \mathsf{syn}(\mathsf{S}_1(x, p), \mathsf{S}_3(y, \delta\ q\ \mathsf{new})) \\
=\;& \mathsf{close}(x, y, \delta\ p\ \mathsf{new}, \delta\ q\ \mathsf{new}) \\
=\;& \mathsf{M}(x, y, \mathsf{tau}(\mathsf{R}(\delta\ \mathsf{par}\ (\delta\ p\ \mathsf{new}, \delta\ q\ \mathsf{new})))) \\
=\;& \mathsf{M}(x, y, \mathsf{tau}(\mathsf{R}(\delta\ (\lambda z : N.\, \mathsf{par}(pz, qz))\ \mathsf{new}))) \\
=\;& \mathsf{M}(x, y, \mathsf{tau}(\mathsf{res}(\lambda z : N.\, \mathsf{par}(pz, qz)))).
\end{aligned}$$

(R1) For $z : N$ and $p : N \times N \Rightarrow A$,

$$\begin{aligned}
&\mathsf{res}(\lambda x : N.\, \mathsf{bout}(z, \lambda y : N.\, p(x, y))) \\
&= \mathsf{R}(\delta\ (\lambda x.\, \mathsf{S}_3(z, \delta\ (\lambda y.\, p(x, y))\ \mathsf{new}))\ \mathsf{new}) \\
&= \mathsf{R}(\mathsf{S}_3(\delta\ z, \delta\ (\lambda x.\, \delta\ (\lambda y.\, p(x, y))\ \mathsf{new})\ \mathsf{new})) \\
&= \mathsf{S}_3(z, \delta\ \mathsf{R}\ (\mathsf{swap}(\delta\ (\lambda x.\, \delta\ (\lambda y.\, p(x, y))\ \mathsf{new})\ \mathsf{new}))) \\
&= \mathsf{S}_3(z, \delta\ \mathsf{R}\ (\delta\ (\lambda y.\, \delta\ (\lambda x.\, p(x, y))\ \mathsf{new})\ \mathsf{new}))\ \text{, by (1)} \\
&= \mathsf{S}_3(z, \delta\ (\lambda y.\, \mathsf{R}(\delta\ (\lambda x.\, p(x, y))\ \mathsf{new}))\ \mathsf{new}) \\
&= \mathsf{S}_3(z, \delta\ (\lambda y.\, \mathsf{res}(\lambda x.\, p(x, y)))\ \mathsf{new}) \\
&= \mathsf{bout}(z, \lambda y : N.\, \mathsf{res}(\lambda x : N.\, p(x, y))).
\end{aligned}$$

3 Full abstraction

This section contains the main results of our study.

Theorem 3.1 (Universality of the set-theoretic interpretation) *In the set-theoretic interpretation of finite agents, writing $A_0 \in \mathbf{Set}^{\mathcal{I}}$ for the object of agents*

1. *$\mathcal{C}[\![V \vdash \_\,]\!]$ establishes a bijective correspondence $\mathcal{CNF}_V \cong A_0(\,|V|\,)$, and so*

2. *for every finite $P, Q \in \mathcal{P}r_V$, $P \sim Q$ iff $\mathcal{C}[\![V \vdash P]\!] = \mathcal{C}[\![V \vdash Q]\!]$.*

Proof: A_0 is the least solution in $\mathbf{Set}$ of the following system of domain equations (with $n \in |\,\mathcal{I}\,|$)

$$X_n = \mathbb{M}\ (n \times {X_n}^n \times X_{n+1} + n \times n \times X_n + n \times X_{n+1} + X_n)$$

where $\mathbb{M}$ is the free-semilattice monad.

So $A_0(\,|V|\,)$ is a set of finite labelled trees, each tree representing an equivalence class of $\mathcal{NF}_V$ modulo the equational laws stating that $(\mathbf{0}, +)$ is a semilattice. ∎

Theorem 3.2 (Finite full abstraction of the close domain-theoretic interpretation) *In the domain-theoretic interpretation, for every finite $P, Q \in \mathcal{P}r_V$, $P \sim Q$ iff $\mathcal{C}[\![V \vdash P]\!] = \mathcal{C}[\![V \vdash Q]\!]$.*

PROOF: Let $\mathbb{M}$ be the free-semilattice monad, $\mathbb{P}$ Abramsky's powerdomain monad, and H the endofunctor $HX = N \times (N \Rightarrow X) + N \times N \times X + N \times \delta X + X$.

The domain equations $X_0 = \mathbb{M}\,(HX_0)$ and $X = \mathbb{P}\,(HX)$ have initial solutions in $\mathbf{Cpo}^{\mathcal{I}}$, say A_0 and A respectively. Moreover, it can be shown that A_0 is the set-theoretic interpretation (ordered by equality), A has an $\mathbb{M}\,H$-algebra structure, and the unique $\mathbb{M}\,H$-homomorphism $e : A_0 \to A$ is an embedding (with a *partial* projection as right adjoint). Therefore, the domain-theoretic interpretation of a finite agent is the image via e of the set-theoretic interpretation and, since e is monic, the two interpretations make the same identifications. ∎

Such an easy transfer of finite full abstraction from a set-theoretic to a domain-theoretic model relies crucially on extending H to *partial morphisms*; this is possible for the endofunctor $V \Rightarrow \_$, only when V is of a *very simple nature.*

Theorem 3.3 (Full abstraction of the close domain-theoretic interpretation) *In the domain-theoretic interpretation, for every $P, Q \in \mathcal{P}r_V$, $P \sim Q$ iff $\mathcal{C}[\![V \vdash P]\!] = \mathcal{C}[\![V \vdash Q]\!]$.*

PROOF: We highlight the main steps of the proof. First, we prove that $\sim$ coincides with $\bigcap_n \sim^n$, where $\sim^n$ is the n-th approximant of $\sim$ (i.e. only the first n actions of the processes are observed). This, which is similar to the stratification result for strong bisimilarity for CCS finitely branching processes, holds because, modulo α-conversion, the transition system of any π-calculus process is finitely branching.

Second, we introduce a key ingredient: a system $\mathcal{A}_\pi$ of axioms and inference rules on π-calculus processes which allows to rewrite a process $P \in \mathcal{P}r_V$ into an *expanded form up to level* n, say $\mathrm{Exp}^n_V(P)$; that is, a syntactic form where the operators of parallel composition, restriction and replication can only occur underneath n prefixes.

Third, using the approximants $\{\sim^n\}_n$ and the operational validity of the laws in the system $\mathcal{A}_\pi$ we can convert the bisimilarity between any pair of processes (P, Q) into the bisimilarity between a set $\{(\mathrm{Nil}^n_V(P), \mathrm{Nil}^n_V(Q)) \mid n \in \omega\}$ of pairs of processes in normal form. $\mathrm{Nil}^n_V(P)$ is obtained from $\mathrm{Exp}^n_V(P)$ by replacing the subprocesses underneath n prefixes with $\mathbf{0}$.

Having normal forms, we can apply the finite full abstraction Theorem 3.2, therefore $\mathrm{Nil}^n_V(P)$ and $\mathrm{Nil}^n_V(Q)$ are bisimilar iff their closed interpretations are equal. To conclude the proof, we have to establish the denotational counterpart of the previous steps, obtained by *replacing* $\sim$ with semantic equivalence.

The processes P and $\mathrm{Exp}^n_V(P)$ are semantically equivalent, because the laws in $\mathcal{A}_\pi$ are semantically valid. Technically, this is the most delicate point and we discuss it after this proof.

By continuity and the semantic validity of the laws in $\mathcal{A}_\pi$ we can convert semantic equivalence between P and Q into the semantic equivalence of their syntactic approximations $\mathrm{Bot}^n_V(P)$ and $\mathrm{Bot}^n_V(Q)$. $\mathrm{Bot}^n_V(P)$ is obtained from $\mathrm{Exp}^n_V(P)$ by replacing the subprocesses underneath n prefixes with $\perp$ (a new constant representing the least element in the semantic domain).

With some simple considerations one can show that for any $P, Q \in \mathcal{P}r_V$, $\mathrm{Bot}^n_V(P)$ and $\mathrm{Bot}^n_V(Q)$ are semantically equivalent iff $\mathrm{Nil}^n_V(P)$ and $\mathrm{Nil}^n_V(Q)$ are. ∎

We now comment on $\mathcal{A}_\pi$ and its semantic validation. In $\mathcal{A}_\pi$, the axioms for sum, restriction and conditionals are those used in [16] or [19] for axiomatising π-calculus late bisimilarity, and include the semilattice axioms for nil and sum, and distributivity axioms for restriction like `R1` (Section 2.5) and $\boldsymbol{\nu} a\,(P + Q) = \boldsymbol{\nu} a\, P + \boldsymbol{\nu} a\, Q$. We need one axiom for each form of replication, like $!\overline{a}b.\,P = \overline{a}b.\,(P \mid !\overline{a}b.\,P)$. To have a finite set of axioms for parallel composition, we use the auxiliary operators left merge and synchronisation. This yields 12 axioms, among which `Syn1` and `Syn2` (Section 2.5) and `Par` (Section 1). The inference rules are those for equivalence and the congruence rules for prefixing, sum, restriction, left merge and synchronisation. However, since late bisimulation is not preserved by input prefix, the inference rule for input is

$$\frac{\forall\, i \in n.\ P\{{}^{a_i}\!/\!_b\} = Q\{{}^{a_i}\!/\!_b\} \qquad P = Q}{a(b).\,P = a(b).\,Q}$$

with the proviso $\mathrm{fn}(a(b).\,P, a(b).\,Q) = \{a_0, \ldots, a_{n-1}\}$. The statement of validation of the laws says that for all $P, Q \in \mathcal{P}r_V$, if $\mathcal{A}_\pi \models P = Q$ then $\mathcal{C}[\![V \vdash P]\!] = \mathcal{C}[\![V \vdash Q]\!]$, and the proof is by induction on the depth of the derivation of $\mathcal{A}_\pi \models P = Q$. Examples of validation of axioms have been given in Section 2.5.

Because of the link between open and closed interpretation in the functor category $\mathcal{C}^{\mathcal{I}}$, it is easy to get a full-abstraction result for late congruence from one for late bisimulation (and conversely).

Corollary 3.4 (Full abstraction of the open domain-theoretic interpretation) *In the domain-theoretic interpretation, for $P, Q \in \mathcal{P}r_V$, $P \sim^{\mathrm{c}} Q$ iff $\mathcal{O}[\![V \vdash P]\!] = \mathcal{O}[\![V \vdash Q]\!]$.*

Distinctions and constraints. We briefly mention the extension of our theory to *distinctions* and *constraints*. In the π-calculus, it is possible to define bisimilarity equivalences in between late bisimilarity and congruence, using distinctions [16]. These are conjunctions of inequalities between names which must be respected in any use of the processes. Guided by the denotational model, we have generalised distinctions to *constraints*, roughly decidable properties on finite tuples of names. Constraints are more expressive than distinctions, in that they allow us to express more refined forms of process bisimilarities. In particular, for any pair of processes there is an *optimal* constraint which expresses the necessary conditions on names under which the processes are behaviourally equivalent. Full abstraction for bisimilarity under constraint follows as a corollary of full abstraction for late bisimilarity (Theorem 3.3); the proof is similar to that for the open interpretation (Corollary 3.4).

4 Conclusions and future work

The denotational model is superior to the operational approach in revealing a few basic properties of $\sim$ and $\sim^{\mathrm{c}}$. For instance, the invariance of $\sim$ under injective substitutions is a straightforward consequence of (3); and the congruence properties of $\sim$ and $\sim^{\mathrm{c}}$ (e.g. that $\sim^{\mathrm{c}}$ is preserved by all operators, and that $\sim$ is preserved by all operators but input) follow directly from the definitions of $\mathcal{O}[\![\_]\!]$ and $\mathcal{C}[\![\_]\!]$.

The denotational model is also interesting for proving basic laws of the operators of π-calculus, like associativity of parallel composition and the extrusion law for restriction "$\boldsymbol{\nu} a\,(P \mid Q) \sim^{\mathrm{c}} P \mid \boldsymbol{\nu} a\, Q$ if $a \notin \mathrm{fn}(P)$". The operational proofs of these laws in [16] require some ingenuity (e.g. the "bisimilarity up to bisimilarity and up to restriction" technique). For an idea of how these proofs may be carried over in the denotational model, see the validation of laws in Subsection 2.5.

Guarded replication. In our π-calculus syntax the plain replication $!P$ has been replaced by a guarded replication $!\alpha.\,P$. This simplification is justified by the laws of π-calculus strong bisimilarity [23] and allows us to avoid the issue of divergence. For instance, with plain replication a denotational semantics would validate $!\mathbf{0} = \perp$, whilst bisimulation validates $!\mathbf{0} = \mathbf{0}$.

Matching/mismatching. For describing the canonical forms induced by our model and hence to obtain the universality Theorem 3.1 we need both matching and mismatching —operators whose theoretical and pragmatical relevance for π-calculus is often debated. The importance of these constructs for equational reasoning had already been expressed in [19].

The metalanguage. We have factored the denotational semantics for the π-calculus through a metalanguage suggested by model-theoretic considerations.

The metalanguage can easily cope with operators not in the π-calculus syntax. For instance, interrupt operators like those in LOTOS [7], which are *not definable* in the π-calculus —even up to weak bisimulation.

An interesting direction of research is to turn the metalanguage into a typed higher order process calculus, with an operational semantics and notion of bisimulation conservatively extending those of the π-calculus.

The translation of the π-calculus in the metalanguage uses a type of agents $A = \mathbb{P}\,(HA)$, where $\mathbb{P}$ is Abramsky's powerdomain monad and H is an endofunctor corresponding to the action capabilities. It is conceivable to replace $\mathbb{P}$ with some other monad and H with some other endofunctor. This flexibility allows to accommodate smoothly other languages, e.g.

- to deal with global variables one should replace $\mathbb{P}$ with the monad $\mathbb{T}X \stackrel{\mathrm{def}}{=} \mathbb{P}\,(X \times S)^S$, where S is an object of states;
- to deal with the πI-calculus [24] (a symmetric subcalculus of π-calculus where the free-output construct is forbidden and hence only private names can be exchanged), one should use the endofunctor $HX \stackrel{\mathrm{def}}{=} N \times \delta X + N \times \delta X + X$. This gives a fully abstract model for πI-bisimilarity; the proof mimics the one outlined in Section 3.

In particular, by changing H we can define a denotational semantics in $\mathbf{Cpo}^{\mathcal{I}}$ for languages ranging from pure CCS to Higher-Order π-calculus (HOπ) [22], along the lines outlined for the π-calculus.

Also the proof of full abstraction is fairly reusable. It copes with the polyadic π-calculus (where several names can be sent at once) and value-passing CCS (with binary sums and guarded recursion), provided the set of values is finite; but it cannot cope with HOπ and CCS with *infinite*-value passing.

We have obtained full abstraction with respect to a *simple* minded domain-theoretic model (in comparison to other categories proposed for Algol-like languages). The main problem to get full abstraction results for domain-theoretic models is not local names and mobility, but higher order! In fact, *simple* domain-theoretic models cannot achieve full abstraction for PCF nor for the $\lambda\nu$-calculus (a CBV λ-calculus whose base types are *bool* and *unit ref*, see [20]). Therefore, to get full abstraction results for higher-order calculi with static binding, like Plain CHOCS [26] and HOπ, one should consider more refined models (probably based on *game semantics* [2, 14]). Thomsen [26], Hennessy [11], and Jeffrey [15] have given denotational models for higher-order process calculi with *dynamic binding*. But their constructions cannot account for calculi based on static binding like the π-calculus, Plain CHOCS, and HOπ.

Limits of our approach. Our semantics for π-calculus is a special case of a uniform approach to give semantics to a variety of calculi. However, this approach deals only with *strong late bisimulation*, which from a denotational point of view appears to be the simplest equivalence to handle (among those proposed for the π-calculus). We do not know how to capture denotationally other equivalences, namely *early* and *open* bisimulations (which differ from late bisimulation and congruence in the requirements on name instantiations) and *weak bisimulations* (where the internal actions of processes are partially ignored). The problem with weak bisimulations is not specific to the π-calculus semantics as there is no established domain-theoretic model for weak bisimulation even in pure CCS.

References

[1] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92:161–218, 1991.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF (extended abstract). In M. Hagiya and J.C. Mitchell, editors, *Proceedings, Theoretical Aspects of Computer Software. International Symposium TACS'94*, volume 789 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[3] S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994.

[4] L. Aceto and A. Ingólfsdóttir. CPO models for a class of GSOS languages. In *Proceedings, TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[5] S.O. Anderson and A.J. Power. A representable approach to nondeterminism. Manuscript, 1993.

[6] J. Baeten and W. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[7] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*. North Holland, 1989.

[8] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *Proceedings, CTCS-5 (Category Theory and Computer Science Fifth Biennial Meeting)*, pages 9–12. CWI, September 1993.

[9] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, Department of Computer Science, University of Edinburgh, 1994. (To be published by Cambridge University Press in the Distinguished Dissertations Series).

[10] M. Hennessy. A term model for synchronous processes. *Information and Control*, 51(1):58–75, October 1981.

[11] M. Hennessy. A fully abstract denotational model for higher-order processes. *Information and Computation*, 112(1):55–95, July 1994.

[12] M. Hennessy and G. Plotkin. Full abstraction for a simple parallel programming language. In J. Becvar, editor, *Proceedings, 8th Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[13] M. Hennessy and G. Plotkin. A term model for CCS. In P. Dembinski, editor, *Proceedings, 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[14] J.M.E. Hyland and C.-H.L Ong. Pi-calculus, dialogue games and PCF. In *Proceedings, 7th Annual ACM Conference on Functional Programming Languages and Computer Architecture*. ACM, 1995.

[15] A. Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science*, pages 255–264. IEEE Computer Society Press, 1995.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–77, September 1992.

[17] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[18] F.J. Oles. Type algebras, functor categories and block structure. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, 1985.

[19] J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120(2):174–197, 1 August 1995.

[20] A.M. Pitts and I.D.B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In A.M. Borzyszkowski and S. Sokolowski, editors, *Proceedings, MFCS'93*, volume 711 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[21] J. Rutten. Processes as terms: non-well-founded models for bisimulation. *Mathematical Structures in Computer Science*, 2:257–275, 1992.

[22] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, University of Edinburgh, 1992.

[23] D. Sangiorgi. On the bisimulation proof method. Technical Report ECS–LFCS–94–299, LFCS, Department of Computer Science, University of Edinburgh, 1994. (A revised version is available via anonymous ftp from `ftp.dcs.ed.ac.uk` as `pub/sad/bis-proof.ps.Z`).

[24] D. Sangiorgi. πI: A symmetric calculus based on internal mobility. In *Proceedings, TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[25] M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, November 1982.

[26] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Department of Computing, Imperial College, 1990.