An investigation into Functions as Processes

Davide Sangiorgi[1]

Abstract. In [Mil90] Milner examines the encoding of the λ-calculus into the π-calculus [MPW92]. The former is the universally accepted basis for computations with *functions*, the latter aims at being its counterpart for computations with *processes*. The primary goal of this paper is to continue the study of Milner's encodings. We focus mainly on the lazy λ-calculus [Abr87]. We show that its encoding gives rise to a λ-model, in which a weak form of extensionality holds. However the model is not fully abstract: To obtain full abstraction, we examine both the *restrictive* approach, in which the semantic domain of processes is cut down, and the *expansive* approach, in which λ-calculus is enriched with *constants* to obtain a *direct* characterisation of the equivalence on λ-terms induced, via the encoding, by the behavioural equivalence adopted on the processes. Our results are derived exploiting an intermediate representation of Milner's encodings into the *Higher-Order π-calculus*, an ω-order extension of π-calculus where also agents may be transmitted. For this, essential use is made of the fully abstract compilation from the Higher-Order π-calculus to the π-calculus studied in [San92a].

1 Introduction

In [Mil90] Milner examines the encoding of the λ-calculus into the π-calculus [MPW92]; the former is the universally accepted basis for computations with *functions*, the latter aims at being its counterpart for computations with *processes*. More precisely, Milner shows how the evaluation strategies of *lazy λ-calculus* and *call-by-value λ-calculus* [Abr87, Plo75] can be faithfully mimicked. The characterisation of the equivalence induced on λ-terms by the encodings is left as an open problem; it also remains to be studied which kind of λ-model — if any — can be constructed from the process terms.

The primary goal of this paper is to continue the study of Milner's encodings. A deep comparison between a process calculus and λ-calculus is interesting for several reasons; indeed, virtually all proposals for process calculi with the capability of treating — directly or indirectly — processes as first class objects have incorporated attempts at embedding the λ-calculus [Bou89, Tho90]. From the process calculus point of view, it is a significant test of expressiveness, and helps in getting deeper insight into its theory. From the λ-calculus point of view, it provides the means to study λ-terms in contexts other than purely sequential ones, and with the instruments developed in the process calculus. For example, an important behavioural equivalence upon process terms gives rise to an interesting equivalence upon λ-terms. Moreover, the relevance of those λ-calculus evaluation strategies which can be efficiently encoded is strengthened. More practical motivations for describing

---

[1] Address: Department of Computer Science, University of Edinburgh, JCMB, Mayfield road, Edinburgh EH9 3JZ, U.K. Email: sad@dcs.ed.ac.uk

functions as processes are to provide a semantic foundation for languages which combine concurrent and functional programming and to develop parallel implementations of functional languages.

Our other major goal is more centered on process calculi. The paradigm on which π-calculus is constructed is first order: Reductions cause instantiations of *names*. This contrasts with what happens in λ-calculus, where reductions cause instantiations of *terms*. Higher-order communications are avoided in π-calculus because of their complexity and because they can be represented at first order. The latter is showed in [San92a] by comparing π-calculus with the *Higher-Order π-calculus* (HOπ), an ω-order extension of π-calculus where not only names but also processes and parametrised processes of arbitrary high order can be communicated: A compilation from HOπ to π-calculus is defined and proved fully abstract with respect to the semantics of the calculi. Thus, the second goal of this paper is to illustrate the use of HOπ and of the representability result of HOπ into π-calculus.

Using the abstraction power of HOπ, for both the lazy and the call-by-value λ-calculus we give encodings which are easier to understand and to deal with than those available in the π-calculus. By applying the compilation from HOπ to π-calculus, we can turn them into π-calculus encodings which can then be compared with Milner's; this is a significant test for the canonicity of the encodings involved. In the lazy λ-calculus the correspondence is exact. That is, if $\mathcal{P}$ and $\mathcal{H}$ are, respectively, the π-calculus and HOπ encodings, and $\mathcal{C}$ is the compilation from HOπ to π-calculus, then the following diagram commutes:

$$\begin{tikzcd}
\lambda \arrow[r, "\mathcal{H}"] \arrow[dr, "\mathcal{P}"'] & \mathrm{HO}\pi \arrow[d, "\mathcal{C}"] \\
 & \pi
\end{tikzcd}$$

In consequence, since $\mathcal{C}$ is fully abstract, any result proved for one of the encodings can be transferred to the other. By working with HOπ, we show in this paper that the encodings do give rise to a λ-model, where conditional extensionality holds. It is not fully abstract, though. To obtain full abstraction we follow two directions: In the *restrictive* approach, based on the use of *barbed bisimulation* [MS92], the semantic domain of processes is cut down; in the *expansive* approach λ-calculus is enriched with constants to obtain a *direct* characterisation of the equivalence on λ-terms induced, via the encoding, by the behavioural equivalence adopted on the processes.

For call-by-value the situation is less sharp. In [Mil90] Milner presents two candidates for the encoding, and it is not obvious which one should be preferred: The first allows easier reasoning, but the second is more efficient. Moreover, when applied to the HOπ encoding, compilation $\mathcal{C}$ does not return either of them. Apparently, to obtain them some code transformation has to be carried out. The study of these transformations leads to interesting outcomes. Firstly, it suggests a correction of the order in which some actions appear in Milner's encodings. This rearrangement does not affect the operational correspondence between λ- and π-terms. However, it affects the behavioural equivalence on the encoding π-terms, in a way which makes the encoding more faithful to the encoded call-by-value discipline. Secondly, the study of the transformations reveals that β-reduction is not valid in Milner's second encoding, which severely reduces its importance.[1] The counter-example is fairly sophisticated and we doubt we could have obtained it without going through HOπ.

This paper is an extract from (mainly chapter 6 of) the author's Ph.D. thesis [San92a]; we refer to it for details and proofs of the results reported.

Acknowledgements. I wish to thank Robin Milner for discussions and suggestions, and Benjamin Pierce, Peter Sewell and the anonymous referees for comments on an earlier draft. The paper was written during my stay at INRIA-Rocquencourt; I am grateful to Jean-Jacques Levy for having invited me.

2 The π-calculus and the Higher-Order π-calculus

In this section we review the syntax and semantics of π-calculus and HOπ, before moving on to the study of the representation of functions as processes, the core of this paper.

2.1 Syntax

We shall look explicitly only at the syntax and the semantics of the *Higher-Order π-calculus (HOπ)*, since the π-calculus is a subcalculus of it. Actually, we only present a *fragment* of these languages, but one which is sufficient for the encoding of the λ-calculus. A more detailed description of the operators involved and their meaning can be found in [San92a].

A HOπ *agent* (or *term*) can be a *process* or an *abstraction*, i.e. a parametrised process. In the following, P and Q stand for processes, F and G for abstractions, A for agents. We use X, Y to range over the set of variables; as in λ-calculus, a variable is supposed to be instantiated with a term. The letters $a, b, \ldots, x, y, \ldots$ stand for names. Moreover, K stands for an agent or a name and U for a variable or a name. We use a tilde to denote a finite (possibly empty) tuple. In the fragment of HOπ we consider, a process is built from names using the operators of parallel composition, restriction, replication, variable application, input and output prefixing, and nil.

$$P :: P_1 \mid P_2 \;\mid\; \boldsymbol{\nu}\, x\, P \;\mid\; !\,P \;\mid\; X\langle \tilde{K} \rangle \;\mid\; x(\widetilde{U}).P \;\mid\; \overline{x}\langle \widetilde{K} \rangle.P \;\mid\; 0$$

An agent is an abstraction over a process or over a partial application:

$$A :: (\widetilde{U})P \;\mid\; (\widetilde{U})X\langle \tilde{K} \rangle$$

Variable application $X\langle \tilde{K} \rangle$ is needed to provide an abstraction received as an input with the appropriate arguments. The other process operators resemble those of the (polyadic) π-calculus and CCS (see [Mil91, Mil89]); we only remind the reader that

---

[1] The version of [Mil90] which appeared in the *Jour. of Math. Structures* was written when the results in this chapter were already known and thus presents only the first encoding.

the replication $!P$ represents an unbounded number of copies of P in parallel, and allows us to describe processes with infinite behaviour. Application has the highest precedence; abstraction the lowest. Sometimes we abbreviate $\alpha.0$ as α. In the above expressions, when a tuple is empty the surrounding brackets $()$ or $\langle\rangle$ are omitted. Note that also a variable X is an agent, corresponding to the case in which $\widetilde{U}$ and $\widetilde{K}$ are empty.

An abstraction is an agent which takes some arguments before becoming a process. The typical form of an abstraction is $(\widetilde{U})P$; it is like a procedure, in which $(\widetilde{U})$ represents the parameters. For instance, $F \stackrel{def}{=} (Y)(P \mid Y)$ abstracts on a process variable; F takes a process and runs it in parallel with P. We can also abstract on abstraction variables, as in $G \stackrel{def}{=} (X)(P \mid X\langle Q\rangle)$; then F applied to G yields $P \mid P \mid Q$. The machinery can be iterated, progressively increasing the order of the resulting abstraction. In this sense HOπ is an ω-order calculus: There is no bound on the order of the agents which can be written and communicated. In contrast, in π-calculus abstractions and communications can only be first order, i.e. abstractions and communications of names. Thus, in π-calculus syntax, variable application does not appear and tuples $\widetilde{K}$ and $\widetilde{U}$ are replaced by simple tuples of names.

W.r.t. the language in [San92a], we have omitted the operators of matching and summation, and recursive definitions of agents have been replaced by replication. The latter is a limitation because while replication can be encoded using recursive definitions, the other way round only holds if the number of recursive definitions is finite (see [Mil91, section 3.1]). Further, in the presented sublanguage an agent may only have a finite number of free names. By contrast, in the full syntax in [San92a], since infinite recursive definitions and infinite summations are allowed, it is possible to write agents which have an infinite number of free names, like a counter which at each step emits a signal on a different channel.

The restriction $\boldsymbol{\nu}\, b\, P$, the input prefix $a(\widetilde{U}).P$ and the abstraction $(\widetilde{U})P$ are formal binders for names and variables in $\widetilde{U}$ and b; they give rise in the expected way to the definitions of α-conversion, *free names* and *free variables* of a term. An *open* agent is an agent possibly containing free variables.

It is crucial in practice to avoid disagreements in what is carried by a given name or expected in applications; for instance we reject expressions like $\overline{a}\langle b_1, b_2\rangle.P|a(x).Q$ or $X\langle b\rangle \mid X\langle b_1, b_2\rangle$, due to the mismatching in the use of the names. In HOπ this need is very compelling: It is not only a question of arities, but we also have to avoid any confusion between instantiation of names and of agents as well as instantiation of agents of different order. To this end, Milner proposed the use of *sorts* [Mil91]. Sorts have, very roughly, the flavour of types in λ-calculus; however in the process calculi not only terms are assigned a sort (or a type), but also names, the latter depending upon the (sorts of the) objects which that name can carry. The sorting system is also useful to understand the passage from π-calculus to HOπ: The ω-order sorts of HOπ can be derived by removing certain constraints on the first-order sort language of π-calculus. We will not present the sorting system because it is not essential to understand the contents of this paper. The reader should take for granted that all agents described obey a sorting.

It is worth pointing out that we do not lose expressiveness in our language by having application only with variables. In fact, every "well-sorted expression" $A\langle\widetilde{K}\rangle$ can be put into this form by "executing" the applications it contains; for instance from $\big((X)Y\langle X\rangle\big)\langle P\rangle$, we get $Y\langle P\rangle$. This makes the definition of substitution more elaborate, but facilitates the proofs in the calculus. However, we shall sometimes use $A\langle F\rangle$ as metanotation; for instance, if $G \stackrel{def}{=} (X)P$, then $G\langle F\rangle$ is $P\{F/X\}$.

2.2 Operational semantics

Following Milner [Mil90, Mil91], we give the operational semantics of the language as a *reduction system*. We begin by defining *structural congruence*, written $\equiv$, as the smallest congruence over the class of processes which satisfies the rules below.

1. $P \equiv Q$ if P is α-convertible to Q;
2. abelian monoid laws for $\mid$: $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $P \mid 0 \equiv P$;
3. $\boldsymbol{\nu}\, x\, 0 \equiv 0$; $\boldsymbol{\nu}\, x\, \boldsymbol{\nu}\, y\, P \equiv \boldsymbol{\nu}\, y\, \boldsymbol{\nu}\, x\, P$; $(\boldsymbol{\nu}\, x\, P) \mid Q \equiv \boldsymbol{\nu}\, x\, (P \mid Q)$; if $x \notin fn(Q)$;
4. $!\,P \equiv P \mid !\,P$.

The structural congruence axioms are used to act upon the structure of terms so that processes willing to interact can be brought into contiguous positions. Then the *reduction relation* can be described with a few simple rules:

$$\text{COM: } x(\widetilde{U}).P \mid \overline{x}\langle\widetilde{K}\rangle.Q \longrightarrow P\{\widetilde{K}/\widetilde{U}\} \mid Q \qquad \text{PAR: } \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\text{RES: } \frac{P \longrightarrow P'}{\boldsymbol{\nu}\, x\, P \longrightarrow \boldsymbol{\nu}\, x\, P'} \qquad \text{STRUCT: } \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

2.3 Barbed bisimulation

Barbed bisimulation was first proposed in [MS92]. One of the motivations was to be able to *uniformly* define bisimulation-based equivalences in different calculi. This is a very important property for the kind of work conducted in this paper, since it allows us to have the same definition of equivalence in the calculi considered (including the λ-calculus, as we shall see in Section 6).

Barbed bisimulation focuses on the reduction relation. It goes a little further though, since the reduction relation by itself is not enough to yield the desired discriminanting power. The choice in [San92a] was to introduce, for each name a, an *observation predicate* $\downarrow_a$ which detects the possibility of performing a communication with the environment along a. We can check whether $P \downarrow_a$ holds from the syntactic form of P: There must be a prefix $a(\widetilde{U})$ or $\overline{a}\langle\widetilde{K}\rangle$ which is not underneath another prefix and not in the scope of a restriction on a. For example, if P is $(\boldsymbol{\nu}\, c)\big(\overline{c}\,.\,b \mid a\,.\,d\big)$, then $P \downarrow_a$, but not $P \downarrow_c$, $P \downarrow_b$ or $P \downarrow_d$.

Definition 1. *Strong barbed bisimulation*, written $\stackrel{\centerdot}{\sim}$, is the largest symmetrical relation on the class of processes of the language such that $P \stackrel{\centerdot}{\sim} Q$ implies:

1. whenever $P \longrightarrow P'$ then there is a Q' such that $Q \longrightarrow Q'$ and $P' \stackrel{\centerdot}{\sim} Q'$;
2. for each name a, if $P \downarrow_a$ then $Q \downarrow_a$. ☐

By itself, barbed bisimulation is too coarse (it is not even preserved by parallel composition). By parametrisation over contexts, we get a finer relation.

Definition 2. Two processes P and Q are *strong barbed-congruent*, written $P \sim Q$, if for each context $C[\cdot]$, it holds that $C[P] \overset{\cdot}{\sim} C[Q]$. $\square$

It is important to stress that the proofs in [San92a] dealing with barbed congruence use the *full* languages of π-calculus and HOπ — as opposed to the fragments presented here — in the construction of the contexts with which processes are tested. Therefore, these contexts use matching, summation, infinite recursive definitions and infinite free names. A challenging problem for future research is to see whether such results can also be proved in the finitary calculus, i.e. without infinite definitions and names.

The weak version of the equivalence, in which one abstracts away from the length of the reductions in two matching actions, is obtained in the standard way: Let $\Longrightarrow$ be the reflexive and transitive closure of $\longrightarrow$ and $\Downarrow_a$ be $\Longrightarrow\downarrow_a$ (the composition of the two relations). Then *weak barbed bisimulation*, written $\overset{\cdot}{\approx}$, is defined by replacing, in definition 1, the transition $Q \longrightarrow Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$; and *weak barbed congruence*, written $\approx$, by replacing in definition 2 $\overset{\cdot}{\sim}$ with $\overset{\cdot}{\approx}$. The definition of barbed congruence on abstractions and open agents is given in the expected way, by requiring instantiation of variables and of abstracted names with all admissible agents or names.

The discriminatory power of barbed bisimulation is tested in [San92a], by proving that in the strong and in the weak case barbed congruence coincides in CCS and π-calculus with the ordinary bisimilarity congruences.

2.4 The compilation from HOπ to π-calculus

We present the compilation from HOπ to π-calculus on agents which can transmit only *one* value — a name or an abstraction — and which only use *unary* abstractions. This is purely to make the definition of the compilation (and of the operational correspondence for it) more readable — the generalisation to the calculus with arbitrary arities does not introduce semantic complications. We use $(\overline{a}\langle m\rangle.P)\,\{m := F\}$ to stand for $\boldsymbol{\nu}\, m\,(\overline{a}\langle m\rangle.P \mid !\, m(U).F\langle U\rangle)$, where U is a name or a variable, depending upon the sort of m.[1] One should think of m as a pointer to F and $\{m := F\}$ as a "local environment" for P. We call m a *name-trigger*.

The compilation $\mathcal{C}$ from HOπ to π-calculus is defined in Table 1. The idea is that the communication of the HOπ agent F should be represented at first order by the communication of a name-trigger m which gives *access* to (the encoding of) F; the name m is used by the recipient to activate the needed copies of F with the appropriate arguments. The other delicate rules are those for application and for variable. Consider the application $X\langle F\rangle$: When X is instantiated to an agent G, it becomes $G\langle F\rangle$. Translating $X\langle F\rangle$, we expect to receive just a name-trigger to G, and we are expected to use this name to activate G, providing it with the

---

[1] When P is 0, the occurrence of $\mid$ is unnecessary and hence $(\overline{a}\langle m\rangle.0)\,\{m := F\}$ should be read as $\boldsymbol{\nu}\, m\,(\overline{a}\langle m\rangle.\,!\, m(U).F\langle U\rangle)$

$$\mathcal{C}[\![X]\!] \overset{def}{=} \begin{cases} \mathcal{C}[\![(Y)X\langle Y\rangle]\!] \text{ if } X \text{ is a higher-order abstraction} \\ \mathcal{C}[\![(a)X\langle a\rangle]\!] \quad \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![\alpha.P]\!] \overset{def}{=} \begin{cases} (\overline{a}\langle m\rangle.\mathcal{C}[\![P]\!])\,\{m := \mathcal{C}[\![F]\!]\} & \text{if } \alpha = \overline{a}\langle F\rangle \\ a(x).\mathcal{C}[\![P]\!] & \text{if } \alpha = a(X) \\ \alpha.\mathcal{C}[\![P]\!] & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![X\langle F\rangle]\!] \overset{def}{=} (\overline{x}\langle m\rangle.0)\,\{m := \mathcal{C}[\![F]\!]\} \qquad \mathcal{C}[\![X\langle b\rangle]\!] \overset{def}{=} \overline{x}\langle b\rangle.0$$

$$\mathcal{C}[\![P \mid Q]\!] \overset{def}{=} \mathcal{C}[\![P]\!] \mid \mathcal{C}[\![Q]\!] \quad \mathcal{C}[\![\boldsymbol{\nu}\, a\, P]\!] \overset{def}{=} \boldsymbol{\nu}\, a\, \mathcal{C}[\![P]\!] \quad \mathcal{C}[\![\,!\,P]\!] \overset{def}{=} \,!\,\mathcal{C}[\![P]\!]$$

$$\mathcal{C}[\![(X)P]\!] \overset{def}{=} (x)\mathcal{C}[\![P]\!] \qquad \mathcal{C}[\![(a)P]\!] \overset{def}{=} (a)\mathcal{C}[\![P]\!]$$

Table 1. The compilation $\mathcal{C}$

argument F. Since we cannot pass agents at first order, as in the rule for output, this is resolved by sending a name-trigger for F. In the rule for variable an η-conversion is employed. This is to make explicit all possible applications and hence to introduce all necessary name-triggers; the use of full triggered forms is needed to get the soundness of Theorem 3 below [San92a]. In this rule, the termination of $\mathcal{C}$ is guaranteed by the well-sortedness hypothesis which ensures that, in $(Y)X\langle Y\rangle$, the sort of Y is "smaller" than the sort of X. In the table, a variable X is mapped to its lower case letter x; we assume that both this name x and the name-trigger m are fresh, i.e. do not occur in the source agent.

As a simple example, suppose $F \overset{def}{=} (b)0$, Y is a first-order variable of the same sort as F, and X is a second-order variable which may take F or Y as arguments. Then

$$\mathcal{C}[\![(X)\big(X\langle F\rangle \mid \overline{b}\langle Y\rangle\big)]\!] = (x)\big(\overline{x}\langle m\rangle\,\{m := F\} \mid \overline{b}\langle m\rangle\,\{m := (z)\overline{y}\langle z\rangle\}\big)$$

Theorem 3 (full abstraction for $\mathcal{C}$). *For each HOπ agent A_1 and A_2, it holds that $A_1 \approx A_2$ iff $\mathcal{C}[\![A_1]\!] \approx \mathcal{C}[\![A_2]\!]$* ☐

3 The lazy λ-calculus

We take for granted the basic concepts of the λ-calculus (see [Bar84]). We use Λ for the class of closed pure λ-terms and M, N, L to range over Λ. We denote by Ω the divergent term $(\lambda x.xx)(\lambda x.xx)$. In Abramsky's *lazy λ-calculus* [Abr87], a redex is always at the extreme left of a term: The reduction rules are those for reflexivity and transitivity plus

$$(\beta)\ (\lambda x.M)N \Longrightarrow M\{N/x\} \qquad\qquad (App)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}$$

When embedding the λ-calculus into a process calculus, functional application becomes a particular parallel combination of two agents, the function and its argument, and β reduction a particular case of interaction. The encoding below of lazy

λ-calculus into HOπ makes this idea very transparent. The translation of a λ-term is an abstraction over a name; this name will be the only access to that agent and will be used to interact with the appropriate λ-term. Thus $\mathcal{H}[\![\lambda x.M]\!]\langle p\rangle$ receives at p its λ-argument and the name q which will give access to M. In the translation of application, the restriction on q prevents interference from other processes. For simplicity, a variable x of the λ-calculus is mapped to its upper-case variable X in HOπ.

$$\begin{aligned}\mathcal{H}[\![\lambda x.M]\!] &\overset{def}{=} (p)p(X,q).\mathcal{H}[\![M]\!]\langle q\rangle\\ \mathcal{H}[\![x]\!] &\overset{def}{=} X\\ \mathcal{H}[\![MN]\!] &\overset{def}{=} (p)\boldsymbol{\nu}\, q\left(\mathcal{H}[\![M]\!]\langle q\rangle \mid \overline{q}\langle \mathcal{H}[\![N]\!], p\rangle.0\right)\end{aligned}$$

The higher-order features of HOπ allow us a simpler encoding than Milner's into π-calculus [Mil90]. Indeed, there is a one-to-one correspondence between reductions in λ-terms and in their HOπ counterparts. Therefore, following Boudol's terminology [Bou89], we can claim that *lazy λ-calculus is a subcalculus of HOπ.*

Proposition 4 (operational correspondence for $\mathcal{H}$). *Let M and M' be closed λ-terms.*

1. *If $M \longrightarrow M'$ then $\mathcal{H}[\![M]\!]\langle p\rangle \longrightarrow \mathcal{H}[\![M']\!]\langle p\rangle$,*
2. *the converse, i.e. if $\mathcal{H}[\![M]\!]\langle p\rangle \longrightarrow Q$ then there is an M' such that $M \longrightarrow M'$ and $Q \equiv \mathcal{H}[\![M']\!]\langle p\rangle$.*

PROOF: Induction on the structure of M. ☐

If we apply compilation $\mathcal{C}$ to the encoding $\mathcal{H}$, the output is precisely Milner's encoding $\mathcal{P}$ in [Mil90]; the symbol '$\circ$' denotes function composition:

Proposition 5. *$\mathcal{C} \circ \mathcal{H} = \mathcal{P}$.* ☐

Consequently, by appealing to the full abstraction for $\mathcal{C}$, we can freely switch between the two encodings. We shall exploit this in Sections 5 and 6 to study them from the point of view of the model theory of the λ-calculus.

4 The Call-by-Value λ-calculus

In call-by-value λ-calculus, reductions may only occur when the argument is a value, i.e. an abstraction. The reduction relation used by Milner in [Mil90] is described by the usual rules for reflexivity and transitivity plus the rules β_v, App_L, App_R:

$$(\beta_v)\ (\lambda x.M)\lambda y.N \Longrightarrow M\{\lambda y.N/x\} \qquad (App_L)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}$$

$$(App_R)\ \frac{N \Longrightarrow N'}{MN \Longrightarrow MN'}$$

We shall try to repeat for call-by-value what we did in the previous section for lazy λ-calculus: We propose an encoding into HOπ and then we compare it with Milner's into π-calculus through compilation $\mathcal{C}$. The call-by-value encodings are slightly more involved than those for the lazy λ-calculus. They also lose a neat canonicity, which is implicitly confirmed by the fact that in his original work [Mil90], Milner presents *two* candidates for the encoding. Basically, the problems in the translation of call-by-value come from the following *dichotomy* in the behaviour of a λ-abstraction. Take the term MN: Both M and N could reduce to an abstraction; but if M does, then the abstraction is destined to perform the *input* of a value, whereas if N does, the abstraction represents an *output* value. This causes disagreement on whether the process which encodes a λ-abstraction should first perform an input or an output.

In the encoding below into HOπ, in contrast with the one for the lazy λ-calculus, the translation of an application allows the two arguments M and N to run in parallel. The HOπ process $\mathcal{H}[\![M]\!]\langle p\rangle$ uses p to communicate (with an output action) that M has reduced to a value; then the dichotomy in the behaviour of a λ-abstraction is solved by the arbiter $App\langle p, q, r\rangle$ which imposes the correct interaction between M and N.

$$\begin{aligned}
\mathcal{H}[\![\lambda x.M]\!] &\overset{def}{=} (p)\overline{p}\big\langle (w)w(X,q).\mathcal{H}[\![M]\!]\langle q\rangle\big\rangle.0\\
\mathcal{H}[\![x]\!] &\overset{def}{=} (p)\overline{p}\langle X\rangle.0\\
\mathcal{H}[\![MN]\!] &\overset{def}{=} (p)(\boldsymbol{\nu}\, q, r)\big(\mathcal{H}[\![M]\!]\langle q\rangle \mid \mathcal{H}[\![N]\!]\langle r\rangle \mid App\langle p,q,r\rangle\big)\\
&\qquad \text{where } App \overset{def}{=} (p,q,r)q(X).r(Y).\boldsymbol{\nu}\, v\,(X\langle v\rangle \mid \overline{v}\langle Y,p\rangle.0)
\end{aligned}$$

It is enlightening to relate this encoding to the one for the lazy λ-calculus. In the above rules for abstraction and variable, the "core" is the object part of the output at p, and it has the same format as the corresponding rule for the lazy λ-calculus. Now the rule for "call-by-value application" should become clear: The arbiter App receives along p and r the "cores" of $\mathcal{H}[\![M]\!]$ and $\mathcal{H}[\![N]\!]$ and then imposes on them the "lazy application". Therefore a reduction on the λ-terms is matched by *three* reductions on the process side. Let us apply compilation $\mathcal{C}$ to $\mathcal{H}$ and see what we get back.

$$\begin{aligned}
\mathcal{C}[\![\mathcal{H}[\![\lambda x.M]\!]]\!] &\overset{def}{=} (p)\,\overline{p}\langle m\rangle\,\{m := (w)w(x,q).\mathcal{C}[\![\mathcal{H}[\![M]\!]]\!]\langle q\rangle\}\\
\mathcal{C}[\![\mathcal{H}[\![x]\!]]\!] &\overset{def}{=} (p)\,\overline{p}\langle m\rangle\,\{m := (w)\overline{x}\langle w\rangle\}\\
\mathcal{C}[\![\mathcal{H}[\![MN]\!]]\!] &\overset{def}{=} (p)\,(\boldsymbol{\nu}\, q, r)\big(\mathcal{C}[\![\mathcal{H}[\![M]\!]]\!]\langle q\rangle \mid \mathcal{C}[\![\mathcal{H}[\![N]\!]]\!]\langle r\rangle \mid App_\pi\langle p,q,r\rangle\big)
\end{aligned}$$

where with simple algebraic manipulations, $App_\pi\langle p,q,r\rangle$ can be written as

$$App_\pi\langle p,q,r\rangle \overset{def}{=} q(x).r(y).(\boldsymbol{\nu}\, v)\overline{x}\langle v\rangle.\big(\overline{v}\langle m,p\rangle\,\{m := (w)\overline{y}\langle w\rangle\}\big)$$

In his original work [Mil90], Milner presents two candidates for the encoding of call-by-value λ-calculus into π-calculus, which we shall call $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. They only differ in the rule for the translation of a variable: In $\mathcal{P}_2$ this rule is simpler, but $\mathcal{P}_1$ allows easier reasoning and proofs. There are two differences between the encoding $\mathcal{C} \circ \mathcal{H}$ and Milner's $\mathcal{P}_i$'s. The first difference is that the order of the actions $r(y)$ and $\boldsymbol{\nu}\, v\, \overline{x}\langle v\rangle$ in $App_\pi\langle p, q, r\rangle$ is reversed. Let us call $\mathcal{P}'_i$, $i = 1, 2$ the encoding obtained from $\mathcal{P}_i$ by commuting such actions $r(y)$ and $\boldsymbol{\nu}\, v\, \overline{x}\langle v\rangle$. This action rearrangement causes a semantic difference between $\mathcal{P}'_i[\![MN]\!]$ and $\mathcal{P}_i[\![MN]\!]$ only when M and N are open. The encodings $\mathcal{P}'_i$'s appear closer than the $\mathcal{P}_i$'s to the call-by-value intuition. We justify this with an example. Consider the λ-term $\lambda x.x\Omega$: Since the call-by-value application $x\Omega$ has a divergent argument Ω, the term $x\Omega$ is supposed not to produce any visible behaviour. Consequently, we expect that a faithful encoding of call-by-value equates $\lambda x.x\Omega$ and $\lambda x.\Omega$. But this is true only for the encodings $\mathcal{P}'_i$'s, whereas it fails for the $\mathcal{P}_i$'s. In consequence, we consider the former an improvement of the latter.

The second difference between the encoding $\mathcal{C} \circ \mathcal{H}$ and Milner's $\mathcal{P}_i$'s (everything we shall say for the $\mathcal{P}_i$'s holds for their "rectified" $\mathcal{P}'_i$'s) is that the component $\overline{v}\langle m, p\rangle\,\{m := (w)\overline{y}\langle w\rangle\}$ of $App_\pi\langle p, q, r\rangle$ is "optimised" as $\overline{v}\langle y, p\rangle.0$ in $\mathcal{P}_1$ and $\mathcal{P}_2$ and, further, in $\mathcal{P}_2$ a similar optimisation occurs in the rule for variable, which is translated as $(p)\overline{p}\langle x\rangle.0$. We call the former *optimisation 1* and the latter *optimisation 2*. It can be shown that both of them are instances of the same potential optimisation of the compilation $\mathcal{C}$ in the rule for output of a variable, namely

$$\mathcal{C}[\![\overline{a}\langle X\rangle.Q]\!] \overset{def}{=} \overline{a}\langle x\rangle.\mathcal{C}[\![Q]\!] \tag{$*$}$$

The intuitive justification for $(*)$ would be the following. Suppose that previous interactions have instantiated the variable X of the HOπ term $\overline{a}\langle X\rangle.Q$ with F. In the encoding π-calculus terms the simulation of these interactions causes the instantiation of the name x with a trigger, say m_F, to $\mathcal{C}[\![F]\!]$. Now, with the rule $(*)$ this same trigger m_F is then transmitted in the output along a. Instead, with the original rule of Table 1 a new name-trigger m to the term $(y)\overline{m_F}\langle y\rangle$ is transmitted: This just seems to introduce a further level of indirection to the activation of $\mathcal{C}[\![F]\!]$. Indeed, rule $(*)$ is often sound, and we believe that optimisation 1 is. This would give us a factorisation for $\mathcal{P}_1$ (or better, for its rectified $\mathcal{P}'_1$) through the HOπ encoding and the compilation $\mathcal{C}$, up-to some code-optimisation. We defer the analysis of the soundness of optimisation 1, as well as of possible other optimisations of $\mathcal{C}$, for future research.

But rule $(*)$ is not sound *in general*. The problem has to do with sharing. With rule $(*)$ two outputs of the same variable become at first-order outputs of pointers to the same "environment entry"; this identity can be recognised and affects the successive behaviour. For instance, the encodings of the strongly congruent HOπ processes (here F is any abstraction)

$$P \overset{def}{=} \boldsymbol{\nu}\, a\,\big(\overline{a}\langle F\rangle.0 \mid a(X).\overline{b}\langle X\rangle.\overline{b}\langle X\rangle.0\big)$$
$$Q \overset{def}{=} \boldsymbol{\nu}\, a\,\big(\overline{a}\langle F\rangle.0 \mid a(X).\overline{b}\langle F\rangle.\overline{b}\langle F\rangle.0\big)$$

would not be equivalent. For the same reason, β-conversion is not valid for Milner's second encoding $\mathcal{P}_2$, as can be shown using the terms $M = \big(\lambda x.(\lambda y.x)\big)(\lambda z.z)$ and $N = \lambda y.(\lambda z.z)$: In one β-step M reduces to N; however $\mathcal{P}_2[\![M]\!] \not\approx \mathcal{P}_2[\![N]\!]$. The difference between them appears after a sequence of interactions with the external environment of length at least 7 (!).

Nevertheless, $\mathcal{P}_2$ yields a precise operational correspondence between λ-terms and their process encodings and, intuitively, one "expects" $\mathcal{P}_2$ to be correct. Recently, in a collaboration with Benjamin Pierce [PS93] we have studied a stronger sorting discipline than Milner's, in which one distinguishes the ability of using channels of a given sort for performing inputs, outputs or both of them. In this system, we have indeed been able to prove the validity of β-reduction for $\mathcal{P}_2$. It would be interesting to see whether the adoption of such a refined sort systems would also validate rule $(*)$.

5 A λ-model from the process terms

Having shown the exact operational correspondence between lazy λ-terms and their process images (Proposition 4), it is legitimate to ask ourselves whether the encoding gives rise to a λ-model, and if so, what kind of λ-model it represents. We chose the lazy λ-calculus because of the simplicity of its encodings. We shall work within HOπ; therefore from now on up to the end of the paper, the word encoding and the symbol $\mathcal{H}$ refer to the HOπ encoding of lazy λ-calculus given in Section 3. All results can be transported onto Milner's encoding into π-calculus via Theorem 3 and Proposition 5.

There are simple syntax-free definitions of λ-model (i.e. they do not mention λ-terms). However, since we already have the mapping from λ to process terms, it is more convenient to use a definition where we can use such a mapping explicitly. A *valuation* is a function from the set of λ-variables to the domain D of the λ-model; $[\boldsymbol{d}/x]\rho$ is the valuation which maps x to $\boldsymbol{d}$ and which behaves like ρ on the remaining elements.

Definition 6 (λ-model, from [HS86]). A λ-*model* is a triple $< D, \cdot, \mathcal{M} >$, where D is a set with at least two elements, '$\cdot$' is a mapping from $D \times D$ to D and $\mathcal{M}$ is a mapping which assigns, to each λ-term M and valuation ρ, a member $\mathcal{M}[\![M]\!]_\rho \in D$ such that:

1. $\mathcal{M}[\![x]\!]_\rho = \rho(x)$ 2. $\mathcal{M}[\![MN]\!]_\rho = \mathcal{M}[\![M]\!]_\rho \cdot \mathcal{M}[\![N]\!]_\rho$
3. $\mathcal{M}[\![\lambda x.M]\!]_\rho \cdot \boldsymbol{d} = \mathcal{M}[\![M]\!]_{[\boldsymbol{d}/x]\rho}$, for all $\boldsymbol{d} \in D$
4. $\mathcal{M}[\![M]\!]_\rho = \mathcal{M}[\![M]\!]_\sigma$, if $\rho(x) = \sigma(x)$ for all x free in M
5. $\mathcal{M}[\![\lambda x.M]\!]_\rho = \mathcal{M}[\![\lambda y.M\{y/x\}]\!]_\rho$, for y not free in M
6. if $\mathcal{M}[\![M]\!]_{[\boldsymbol{d}/x]\rho} = \mathcal{M}[\![N]\!]_{[\boldsymbol{d}/x]\rho}$ for all $\boldsymbol{d} \in D$, then $\mathcal{M}[\![\lambda x.M]\!]_\rho = \mathcal{M}[\![\lambda x.N]\!]_\rho$. $\Box$

Our λ-model should respect the semantic relation adopted in HOπ. So, let us denote by $[\,A\,]_\approx$ the equivalence class of the agent A, namely

$$[\,A\,]_\approx = \{A' \ : \ A' \text{ is an HO}\pi \text{ agent and } A \approx A'\}$$

The elements of the domain D of the model will be the equivalence classes of the closed HOπ agents with the same sort S as the agents encoding λ-terms.

$$D \overset{def}{=} \{[\,F\,]_\approx \ : \ F \in \text{HO}\pi \text{ and } F \text{ has sort } S\}$$

The definition of application on these elements follows the translation of λ-application in $\mathcal{H}$:

$$[\,G\,]_{\approx} \cdot [\,F\,]_{\approx} \overset{def}{=} [\,(p)\boldsymbol{\nu}\, q\,(G\langle q\rangle \mid \overline{q}\langle F, p\rangle)\,]_{\approx} \quad \text{for } p, q \text{ not free in } G, F$$

Note that the definition of application is consistent: by the congruence properties of $\approx$, the result of the application does not depend upon the representatives G and F chosen from the equivalence classes. We are left with the definition of the mapping $\mathcal{M}[\![M]\!]_{\rho}$. The valuation ρ maps λ-variables to equivalence classes of $\approx$. Given a valuation ρ, we denote by ρ^{H} a substitution from HOπ variables to HOπ agents s.t.

$$\text{for each } \lambda\text{-variable } x, \quad \rho^{H}(\mathcal{H}[\![x]\!]) \in \rho(x)$$

Therefore, ρ^{H} is the "conversion" of ρ which operates on the HOπ variable $\mathcal{H}[\![x]\!]$ and which selects a representative out of the equivalence class of $\rho(x)$. Now, the mapping $\mathcal{M}$ of the λ-model is defined using $\mathcal{H}$ as follows:

$$\mathcal{M}[\![M]\!]_{\rho} \overset{def}{=} [\,\mathcal{H}[\![M]\!]\rho^{H}\,]_{\approx}$$

Note that since $\approx$ is a congruence, this definition is independent of the representatives of the equivalence classes selected by ρ^{H}. We denote by $\mathcal{D}$ be the triple $< D, \cdot, \mathcal{M} >$ so obtained.

Theorem 7. *$\mathcal{D}$ is a λ-model.*

PROOF: Use the definition of $\mathcal{D}$ plus the congruence properties of $\approx$ to show that each clause of Definition 6 is satisfied. ☐

We could have tried to be more selective in the definition of the domain $\mathcal{D}$, and take as domain $D^{*} = \{[\,\mathcal{H}[\![M]\!]\,]_{\approx} \ : \ M \in \Lambda\}$; then $\mathcal{D}^{*} =< D^{*}, \cdot, \mathcal{M} >$ represents the *interior* of $\mathcal{D}$ [HS86]. But it turns out that $\mathcal{D}^{*}$ is *not* a λ-model. Clause (6) in Definition 6 fails. As counterexample, take the terms L_1 and L_2 as will be defined in Section 6. Their encodings are not equivalent, i.e. $\mathcal{H}[\![L_1]\!] \not\approx \mathcal{H}[\![L_2]\!]$; however, for all closed N it holds that $\mathcal{H}[\![L_1\{N/x\}]\!] \approx \mathcal{H}[\![L_2\{N/x\}]\!]$. Therefore $\mathcal{D}$ is an example of a λ-model whose interior is not a λ-model; see [HL80] for two more examples.

Now that we know that $\mathcal{D}$ is a λ-model, we can infer all properties of λ-models for it; in particular we get that

- Every provable equation of $\lambda\beta$ is valid for the encoding, up to $\approx$ (where $\lambda\beta$ is the formal theory given by α and β conversion plus the rules of inference for equivalence and congruence).
- $< D, \cdot >$ is a combinatory algebra (and hence is combinatorially complete) where the two distinguished elements $\boldsymbol{k}$ and $\boldsymbol{s}$ can be defined as $\boldsymbol{k} = [\,\mathcal{H}[\![\lambda xy.x]\!]\,]_{\approx}$, and $\boldsymbol{s} = [\,\mathcal{H}[\![\lambda xyz.xz(yz)]\!]\,]_{\approx}$.

However model $\mathcal{D}$ is *not* extensional, i.e. it is not a $\lambda\eta$ model. As counterexample, take Ω and $\lambda x.\Omega x$. Then $\mathcal{H}[\![\Omega]\!]\langle p\rangle \not\approx \mathcal{H}[\![\lambda x.\Omega x]\!]\langle p\rangle$, since $\mathcal{H}[\![\Omega]\!]\langle p\rangle \approx 0$, whereas $\mathcal{H}[\![\lambda x.\Omega x]\!]\langle p\rangle$ can perform a visible action at p. This failure is not too surprising, since our encoding mimics the lazy λ-calculus, in which the η rule is not valid. However, as in the lazy λ-calculus, the η rule holds if M is convergent:

Theorem 8 conditional extensionality.
$\mathcal{H}[\![M]\!]\langle p\rangle \Downarrow_p$ implies $\mathcal{H}[\![\lambda x.Mx]\!] \approx \mathcal{H}[\![M]\!]$, for $x \notin fv(M)$.

PROOF: Use Proposition 4 and the definition of the encoding. □

6 Full abstraction

Full abstraction, first studied by Milner [Mil77] and Plotkin [Plo77], is the problem of finding a denotational interpretation for a programming language such that the resulting semantic equality coincides with a notion of operational indistinguishability.

Inspired by the work of Milner and Park in concurrency [Par81, Mil89], Abramsky [Abr87] introduces an operational equivalence on the lazy λ-calculus terms called *applicative bisimulation*, built on the idea that convergence is the only observable property.

Definition 9. *Applicative bisimulation* is the largest symmetric relation $\simeq\subseteq \Lambda\times\Lambda$ such that if $M \simeq N$ and $M \Longrightarrow \lambda x.M'$, then there is an N' such that $N \Longrightarrow \lambda x.N'$ and $M'\{L/x\} \simeq N'\{L/x\}$, for all $L \in \Lambda$. □

If we take λ as the only port of the λ-calculus and $M \Downarrow_\lambda$ as meaning "M can reduce to an abstraction", then applicative bisimulation is the λ-calculus version of weak barbed congruence. This follows from the characterisation of applicative bisimulation in terms of "convergence in all contexts" given in [AO89].

The classical setting in which the full abstraction problem has been developed is the simply typed λ-calculus. With the introduction of the operational equivalence resulting from applicative bisimulation, it can be neatly transferred to the untyped λ-calculus and it has motivated elegant works by Abramsky, Ong and Boudol ([AO89, Bou91]).

A denotational interpretation is said to be *sound* if it only equates operationally equivalent terms, *complete* if it equates all operationally equivalent terms, and *fully abstract* if it is sound and complete. Let us consider what happens with the encoding $\mathcal{H}$. It is sound, since $\mathcal{H}[\![M]\!] \approx \mathcal{H}[\![N]\!]$ implies $M \simeq N$; this can be established using (mainly) Proposition 4. However, $\mathcal{H}$ is not complete. For this, take:

$$L_1 = x(\lambda y.(x\Xi\Omega y))\Xi \qquad\qquad L_2 = x(x\Xi\Omega)\Xi.$$

where Ξ is an always-convergent term (that is, for all $\tilde{N}$, $\Xi\tilde{N} \Downarrow$), like the term $(\lambda x.\lambda y.(xx))(\lambda x.\lambda y.(xx))$. Terms L_1 and L_2 are used by Abramsky and Ong [AO89] to show that their canonical model for lazy λ-calculus is not fully abstract. They show that L_1 and L_2 are applicative bisimilar but can be distinguished using *convergence test*, an operator which is definable in the canonical model but is not in the pure lazy λ-calculus. We also have $\mathcal{H}[\![L_1]\!] \not\approx \mathcal{H}[\![L_2]\!]$; by using Theorem 3, this follows from a similar result for the encoding into π-calculus, which Milner obtained by implementing the convergence test as a π-process [Mil90]. In terms of the model $\mathcal{D}$ of the previous section, this inequality means that L_1 and L_2 have different denotations and that $\mathcal{D}$ is not fully abstract. Given a denotational interpretation which is not fully abstract, there are two natural directions to achieve full abstraction:

- to cut down the existing "over-generous" semantic domain (*restrictive approach*);
- to enrich the language (*expansive approach*).

These two approaches are exemplified by the solutions to the full abstraction problem for PCF (a typed λ-calculus extended with fixed points, boolean and arithmetic features) proposed by Milner [Mil77] and Plotkin [Plo77]; in the latter, PCF is augmented with a 'parallel or' operator. We shall see that in our case both directions lead to interesting constructions.

6.1 The restrictive approach

The first approach exploits the possibility of quantifying barbed bisimulation on a *particular* class of contexts, which allows us to specify exactly the way in which certain agents are supposed to be used. As λ-terms are *only* used in λ-calculus contexts, so we can require that their encodings be used only in encodings of λ-contexts. The encoding $\mathcal{H}$ is extended to λ-contexts by mapping the λ hole to the HOπ hole, i.e. $\mathcal{H}[\![[\cdot]]\!] \overset{def}{=} [\cdot]$. Thus, the class of contexts we are interested in is

$$\mathcal{L} = \{\mathcal{H}[\![C_\lambda[\cdot]]\!] \text{ such that } \; C_\lambda[\cdot] \text{ is a } \lambda\text{-context}\}$$

For $P, Q \in HO\pi$, we set $P \approx_{\mathcal{L}} Q$ if for every $\mathcal{L}$-context $C[\cdot]$, the processes $C[P]$ and $C[Q]$ are barbed bisimilar.

Proposition 10. *For each $M, N \in \Lambda$, it holds that $M \simeq N$ iff $\mathcal{H}[\![M]\!] \approx_{\mathcal{L}} \mathcal{H}[\![N]\!]$*

PROOF: By use of the operational correspondence (Proposition 4), the characterisation of $\simeq$ in terms of barbed congruence, the congruence properties of $\simeq$. □

This result allows us to construct a fully abstract model for the lazy λ-calculus. Let $[\,A\,]_{\approx_{\mathcal{L}}}$ be the equivalence class of A modulo $\approx_{\mathcal{L}}$, '$\cdot$' and $\mathcal{M}[\![M]\!]$ as defined in Section 5 but with $[\;]_{\approx_{\mathcal{L}}}$ in place of $[\;]_{\approx}$, and

$$D' = \{[\,\mathcal{H}[\![M]\!]\,]_{\approx_{\mathcal{L}}} \;:\; M \in \Lambda\}.$$

Theorem 11 full abstraction. *$\mathcal{D}' =< D', \cdot, \mathcal{M}>$ is a fully abstract model for the lazy λ-calculus.*

PROOF: Full abstraction follows from Proposition 10. The proof that $\mathcal{D}'$ is a λ-model is analogous to the proof that $\mathcal{D}$ is a λ-model in Theorem 7. (The proof of Theorem 7 used the congruence properties of $\approx$; in this case, we need the congruence of $\approx_{\mathcal{L}}$ on encodings of λ-contexts; that is, if $C[\cdot]$ is the encoding of a λ-context and $\mathcal{H}[\![M]\!] \approx_{\mathcal{L}} \mathcal{H}[\![N]\!]$, then $C[\mathcal{H}[\![M]\!]] \approx_{\mathcal{L}} C[\mathcal{H}[\![N]\!]]$). □

Indeed, the model $\mathcal{D}'$ is also *fully expressive*: all objects of the domain of interpretation are λ-definable. These results show that if from π-calculus and HOπ we discard everything which is extraneous to the encoding, in particular adopting $\approx_{\mathcal{L}}$ as semantic equivalence, then the structure that we get back is the "same thing" as the lazy λ-calculus. In our view, this was really the decisive test for the correctness of the HOπ and π-calculus encodings.

The domain D' weakens the domain D of Section 5 in two aspects: The behavioural equivalence is $\approx_{\mathcal{L}}$ rather than the more discriminanting $\approx$; and only the interior of D is taken into account. The first restriction is necessary to get full abstraction; the second to make the definition of application consistent. It is interesting to note the relationship between the choice of the class of HOπ agents and the choice of the behavioural equivalence in the definition of the domains D and D'. In the former, we took the class $\mathcal{A}$ of *all* admissible agents, and then in the behavioural equivalence we had to use quantification over the class $\mathcal{C}nt$ of *all* admissible contexts (definition of $\approx$); in the latter we restricted to the *"interior"* of $\mathcal{A}$, and then in the behavioural equivalence we had to restrict to the *"interior"* of $\mathcal{C}nt$ (definition of $\approx_{\mathcal{L}}$).

6.2 The expansive approach

We next study the equivalence induced on λ-terms by the encoding, called *λ-observational equivalence*; it equates the λ-terms M and N if $\mathcal{H}[\![M]\!] \approx \mathcal{H}[\![N]\!]$. In other words, we look at the effect on λ-terms of the use of "richer" contexts, in which also concurrent features may be present. To derive a direct characterisation of λ-observational equivalence (i.e. a characterisation not mentioning the encoding) we have to enrich the λ-calculus with constants. A *constant* is a symbol which is added to the language without specifying any operational rule; in this sense they are opposed to *operators*, for which the behavioural rules are given (examples are convergence test and non-deterministic choice). Constants can be found in the well-known technique of the *top down specification and analysis*, where a system is developed through a series of refinement steps each representing a different level of abstraction; a lower level implements some details which at a higher level are left undefined. A constant c is then a high level primitive standing for some lower level procedure K_c; Now for closed terms, $c\tilde{M}$ becomes a sensible normal form. Operationally, we really can see it as the *output* of the tuple $\tilde{M}$ along the channel c and towards K_c.

Let Λ_C be the class of λ-terms enriched with constants. When generalising applicative bisimulation to terms in Λ_C, the main question is which condition should be imposed on the equality between the terms $c\tilde{M}$ and $c\tilde{N}$. According to the above interpretation of constants, it is natural to require that the *ordered* sequence of arguments represented by $\tilde{M}$ and $\tilde{N}$ be equivalent (clause (2) in the following definition).

Definition 12. *Applicative bisimulation over Λ_C*, written $\simeq_{\mathrm{C}}$, is the largest symmetrical relation on $\Lambda \times \Lambda$ such that $M \simeq_{\mathrm{C}} N$ implies:

1. if $M \Longrightarrow \lambda x.M'$ then there is an N' such that $N \Longrightarrow \lambda x.N'$ and $M'\{L/x\} \simeq_{\mathrm{C}} N'\{L/x\}$, for all $L \in \Lambda_C$;
2. if $M \Longrightarrow cM_1...M_n$, for some $n \geq 0$ and $c \in C$, then there are $N_1, \ldots N_n$ such that $N \Longrightarrow cN_1 \ldots N_n$ and $M_i \simeq_{\mathrm{C}} N_i$, $1 \leq i \leq n$.

The proof (in [San92a]) that $\simeq_C$ coincides with λ-observational equivalence (in particular the implication from right to left) is delicate. Milner's encoding is extended to Λ_C by mapping constants to a special kind of agents called *triggers*. These were introduced in [San92a] to obtain a simple characterisation of barbed congruence in HOπ; it turns out that their discriminanting power is the same as that of constants in the λ-calculus.

Theorem 13 (direct characterisation of λ-observational equivalence). *If $M, N \in \Lambda$, it holds that $M \simeq_C N$ iff $\mathcal{H}[\![M]\!] \approx \mathcal{H}[\![N]\!]$* ☐

Let $\mathcal{D}_C$ be the extension to Λ_C of the model $\mathcal{D}$ of Section 5; $\mathcal{D}_C$ is defined as $\mathcal{D}$ with Λ_C in place of Λ, and utilising the extension of $\mathcal{H}$ to Λ_C.

Corollary 14 (full abstraction for $\mathcal{D}$). *$\mathcal{D}_C$ is a fully abstract model for the lazy λ-calculus enriched with constants.* ☐

Starting from these results, the study of λ-observational equivalence has been continued in [San92b]. The outcomes suggest that it is a *robust* equivalence. First, it enjoys simple operational and denotational characterisations. Secondly it coincides with the equivalence obtained when the λ-calculus is augmented with the whole class of *well-formed operators*, a fairly large class of operators whose behaviour depends only on the semantics — not on the syntax — of their operands; that is to say, the encoding into π-calculus/HOπ induces maximal observational discrimination on λ-terms.

References

[Abr87] S. Abramsky. *Domain Theory and the Logic of Observable Properties.* PhD thesis, University of London, 1987.

[AO89] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. Technical report, Imperial College, 1989. To appear in *Information and Computation.*

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic.* North Holland, 1984. Revised edition.

[Bou89] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89*, volume 351 of *Lecture Notes in Computer Science*, pages 149–161, 1989.

[Bou91] G. Boudol. A lambda calculus for (strict) parallel functions. Rapport de recherche 1387, INRIA-Sophia Antipolis, 1991. To appear in Information and Computation.

[HL80] J.R. Hindley and G. Longo. Lambda calculus models and extensionality. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 26:289–310, 1980.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Mil77] R. Milner. Fully abstract models of typed lambda calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathem. Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. To appear in the *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Par81] D.M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conf. on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.

[Plo75] G.D Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G.D Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *8th LICS Conf.* IEEE Computer Society Press, 1993.

[San92a] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* CST–99–93, Department of Computer Science, University of Edinburgh, 1992. Also published as ECS–LFCS–93–266.

[San92b] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. In *7th LICS Conf.* IEEE Computer Society Press, 1992.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Department of Computing, Imperial College, 1990.