# Interpreting functions as $\pi$-calculus processes: a tutorial*

Davide Sangiorgi
INRIA Sophia-Antipolis, France†

August 1998; revised February 1999

**Abstract**

This paper is concerned with the relationship between $\lambda$-calculus and $\pi$-calculus. The $\lambda$-calculus talks about *functions* and their *applicative* behaviour. This contrasts with the $\pi$-calculus, that talks about *processes* and their *interactive* behaviour. Application is a special form of interaction, and therefore functions can be seen as a special form of processes. We study how the functions of the $\lambda$-calculus (the *computable* functions) can be represented as $\pi$-calculus processes. The $\pi$-calculus semantics of a language induces a notion of equality on the terms of that language. We therefore also analyse the equality among functions that is induced by their representation as $\pi$-calculus processes.

This paper is intended as a tutorial. It however contains some original contributions. The main ones are: the use of well-known *Continuation Passing Style* transforms to derive the encodings into $\pi$-calculus and prove their correctness; the encoding of *typed* $\lambda$-calculi.

**Note:** This is a draft of a chapter of a book that I am writing with David Walker. In the hope of making the paper self-contained, I have added appendices with background material on $\pi$-calculus.

*Comments, remark, etc. are warmly appreciated!!*

**Acknowledgements:** I am indebted to David Walker for detailed comments on a previous draft of this paper. I also benefited from discussions with Gérard Boudol, Femke Van Raamsdonk, Christine Röckl.

*Also appeared as INRIA Technical Report RR-3470.

†Research supported by the project CONFER-2 (WG-21836).

# Contents


**1 The $\lambda$-calculus** **6**

**2 Contrasting $\lambda$ and $\pi$** **7**
- 2.1 Reduction axioms . . . . . 7
- 2.2 Sequentiality . . . . . 8
- 2.3 Confluence . . . . . 9

**3 Reduction strategies for the $\lambda$-calculus** **9**
- 3.1 Call-by-name . . . . . 10
- 3.2 Call-by-value . . . . . 11
- 3.3 Call-by-need . . . . . 12

**4 Interpreting $\lambda$-calculi into $\pi$-calculus** **13**
- 4.1 Continuation Passing Style . . . . . 14

**5 The interpretation of call-by-value** **15**
- 5.1 The three steps . . . . . 15
- 5.2 Composing the steps . . . . . 20
- 5.3 Parallel call-by-value . . . . . 24

**6 The interpretation of call-by-name** **25**
- 6.0.1 Composing the steps . . . . . 27

**7 A uniform encoding** **31**

**8 Optimisations of the call-by-name encoding** **34**

**9 Encoding the $\xi$ rule** **35**

**10 Interpreting typed $\lambda$-calculi** **37**
- 10.1 The interpretation of typed call-by-value . . . . . 38
- 10.2 The interpretation of typed call-by-name . . . . . 42

**11 Full abstraction** **44**
- 11.1 Applicative bisimilarity . . . . . 45
- 11.2 Soundness and non-completeness . . . . . 46

**12 Extending the $\lambda$-calculus** **49**
- 12.1 The discriminating power of extended $\lambda$-calculi . . . . . 52
- 12.2 Encoding the operators of the $\lambda$-extensions into $\pi$-calculus . . . . 53

**13 The local structure of the $\pi$-interpretation** **56**
- 13.1 Sensible theories and lazy theories . . . . . 57
- 13.2 Lévy-Longo Trees and the local structure theorem . . . . . 57
- 13.3 The proof of the local structure theorem . . . . . 60
- 13.4 Böhm Trees . . . . . 66
- 13.5 Local structure of the call-by-value encoding . . . . . 66

**14 Notes** **68**

**A The $\pi$-calculus** **72**
A.0.1 Processes . . . 72
A.0.2 Operational semantics . . . 74
A.0.3 The asynchronous $\pi$-calculus . . . 74

**B Behavioural equivalences** **74**
B.0.4 Barbed congruence . . . 74
B.0.5 Ground bisimilarity . . . 75
B.0.6 Weak equivalences . . . 76
B.0.7 The expansion relation . . . 77
B.0.8 Techniques of "bisimulation up to" . . . 77
B.0.9 Some laws for the $\pi$-calculus . . . 77
B.1 Extensions and types . . . 78
B.1.1 Some conventions and notation for typed calculi . . . 78
B.1.2 Polyadicity . . . 79
B.1.3 i/o types . . . 79
B.1.4 Linearity and receptiveness . . . 80
B.1.5 Types and behavoural equivalences . . . 80
B.1.6 The strong replication theorems . . . 81
B.1.7 Abstractions . . . 82
B.1.8 The delayed input . . . 82

**C The Higher-Order $\pi$-calculus** **83**

**D Compiling process-passing into name-passing** **85**
D.0.9 Extensions . . . 87

## Motivations

A deep study of representations of functions as $\pi$-calculus processes is of interest for several reasons. From the $\pi$-calculus point of view, the representation is a significant test of expressiveness, and is helpful in getting deeper insight into the theory. From the $\lambda$-calculus point of view, the representation makes it possible to apply process-calculus techniques to $\lambda$-calculus, and also to analyse $\lambda$-terms in contexts which are not purely sequential. This study may be useful for providing a semantic foundation for languages having constructs for functions and for concurrency, and techniques for reasoning about them (behavioural equalities between functions preserved in sequential contexts may not be preserved in non-sequential contexts, we shall see examples of this in Section 11). The study may also be helpful in development of parallel implementations of functional languages and in the design of programming languages based on process calculi.

### Summary

The paper is ideally divided into four parts. The first (comprising Sections 1 to 3) is about the $\lambda$-calculus. The second (Sections 4 to 9) is about the encoding of the untyped $\lambda$-calculus into $\pi$-calculus. Part 3 (Section 10) does the same for the typed $\lambda$-calculus. Part 4 (Sections 11 to 13) is about the full abstraction problem for the simplest of the $\pi$-calculus encodings, namely that of the untyped call-by-name $\lambda$-calculus. A more detailed summary follows.

In Section 1 we review the syntax and reduction rules of the untyped $\lambda$-calculus. In Section 2, we look at some properties of the $\lambda$-calculus that make it strikingly different from the $\pi$-calculus: sequentiality and confluence. We also touch on the differences and the similarities between the basic computational rules of the two calculi.

A $\lambda$-term may have several reducible subterms. A *reduction strategy* specifies a reduction order. The most important reduction strategies are call-by-name and call-by-value, and variants of these such as parallel call-by-value and call-by-need (the last is actually an implementation technique, rather than a reduction strategy). We review these strategies and their properties in Section 3.

Section 4 begins the core of the paper, which is about the encodings of $\lambda$-calculus strategies into $\pi$-calculus. We analyse the encoding of call-by-value in Section 5, and that of call-by-name in Section 6. We derive both encodings from well-known *Continuation Passing Style* transforms (which transform functions by adding continuations to them) and the compilation of the *Higher-Order $\pi$-calculus* (HO$\pi$) into $\pi$-calculus. Variants of these strategies and encodings are examined in Sections 7 and 8. The strategies considered up to this point do not allow reductions inside the body of a function. Technically speaking, they do not allow the $\xi$ rule of the $\lambda$-calculus. In Section 9 we discuss when and how the $\xi$ rule may be encoded.

In Section 10 we address what happens to the type structure of functions when they are represented as processes. In $\lambda$-calculi, types are assigned to terms, and provide an abstract view of their behaviour. In contrast, in the type systems for the $\pi$-calculus types are assigned to names and hence reveal very little about behavioural properties of the processes. The semantic relationship between the two forms of types is therefore not obvious. Understanding how $\lambda$-calculus types are transformed by the $\pi$-calculus encodings is important if we wish to use the $\pi$-calculus as a semantic basis for typed programming languages. We look at the simply-typed $\lambda$-calculus in detail, and briefly discuss extensions.

Both for typed and for untyped $\lambda$-calculi, we derive $\pi$-calculus encodings of call-by-name and call-by-value in three steps: a CPS transform, the inclusion of CPS terms into HO$\pi$, the compilation from HO$\pi$ to $\pi$-calculus. This is useful for understanding the encodings and for proving their properties. Conversely, *a reader familiar with $\pi$-calculus might find the encodings helpful for understanding the CPS transforms;* indeed, one can also go the other way round, and use the $\pi$-calculus encodings to derive results about the correctness of the CPS transforms—see Remark 5.22, for instance.

The encoding of untyped call-by-name $\lambda$-calculus ($\lambda N$) is the simplest and perhaps most natural encoding of $\lambda$-calculus into $\pi$-calculus. In Section 11 we study the equality on $\lambda$-terms induced by this encoding. This equality, $=_\pi$, relates two $\lambda$-terms if their encodings are behaviourally-equivalent $\pi$-calculus processes. As behavioural equivalence for the $\pi$-calculus we choose barbed congruence. The results obtained are, however, largely independent of this choice, due to the special form of the processes encoding $\lambda$-terms. The same results hold, for instance, for testing equivalence or trace equivalence.

We begin by comparing $=_\pi$ with the equality given by the operational semantics of $\lambda N$. The latter can be formulated as a form of bisimilarity, *applicative bisimilarity.* An interpretation of a calculus is *sound* if it equates only operationally equivalent terms, *complete* if it equates all operationally equivalent terms, and *fully abstract* if it is sound and complete. We shall see that the $\pi$-calculus interpretation of $\lambda N$ is sound, but not complete.

When an interpretation of a calculus is sound but not fully abstract, one may hope to achieve full abstraction by enriching the calculus. (This is exemplified by the solutions to the full abstraction problem for PCF—a typed $\lambda$-calculus extended with fixed points, boolean and arithmetic features—proposed by Plotkin [Plo77], in which PCF is augmented with a 'parallel or' operator.) We follow this approach for $\lambda N$ in Section 12. We augment $\lambda N$ with operators, that is symbols equipped with reduction rules defining their behaviour. We prove that the addition of certain operators that yield non-confluent reductions is necessary and sufficient to make the $\pi$-calculus interpretation fully abstract (on pure $\lambda$-terms). The operators needed are rather simple. One example is a unary operator which when applied to an argument either behaves like the argument itself or diverges. These results imply that the operational equivalence of simple non-confluent extensions of $\lambda N$ is robust: its equalities remain valid in rich extensions of $\lambda N$, possibly including operators for expressing concurrency.

In Section 13 we investigate the meaning of $=_\pi$: Which $\lambda$-terms does it equate? To determine this, we prove a few characterisations of $=_\pi$ on the pure $\lambda$-terms. The most important one is a characterisation in terms of a tree structure of the $\lambda$-calculus, the *Lévy-Longo Trees.* We also discuss how to obtain a characterisation in terms of the other main tree structure of the $\lambda$-calculus, the *Böhm Trees.* Tree structures are an important part of the theory of the $\lambda$-calculus; they are especially useful in studying its models. A corollary of the characterisations in terms of trees is that the equality induced by the $\pi$-calculus encoding is the same as that induced by well-known models of the $\lambda$-calculus. This is a remarkable agreement between the classical theory of functions and their interpretation as processes.

**Note:** The paper proposes several exercizes. They normally require some familiarity with the $\pi$-calculus. Those marked with a asterisk are the hardest.

# 1 The $\lambda$-calculus

The beauty of the $\lambda$-calculus is that it achieves Turing completeness (all computable functions are definable in it) with a very simple syntax. The basic operators of the $\lambda$-calculus, in its pure form, are $\lambda$-abstraction, for forming functions, and application, for applying a function to an argument. A $\lambda$-abstraction has the form $\lambda x.\, M$; in the body $M$ of the function, the variable $x$ is a placeholder for the argument. Letting $x$ and $y$ range over the set of $\lambda$-calculus variables, the set $\Lambda$ of *$\lambda$-terms* is defined by the grammar

$$M := x \;\Big|\; \lambda x.\, M \;\Big|\; M_1 M_2\,.$$

In $\lambda x.\, M$, the initial $x$ is a static binder, binding all free occurrences of $x$ in $M$. We omit the definitions of $\alpha$-conversion, free variable, substitution, etc. We identify $\alpha$-convertible terms, and therefore write $M = N$ if $M$ and $N$ are $\alpha$-convertible. A $\lambda$-term is *closed* if it contains no free variables. The set of free variables of a term $M$ is written $\mathrm{fv}(M)$. The subset of $\Lambda$ containing only the closed terms is $\Lambda^0$.

To avoid too many brackets, we assume that application associates to the left, so that $MNL$ should read $(MN)L$, and that the scope of a $\lambda$ extends as far as possible to the right, so that $\lambda x.\, MN$ should read $\lambda x.\,(MN)$. We also abbreviate $\lambda x_1.\cdots.\lambda x_n.\, M$ to $\lambda x_1\cdots x_n.\, M$, or $\lambda\widetilde{x}.\, M$ if the length of $\widetilde{x}$ is not important. We follow [Bar84, HS86] in notations and terminology for the $\lambda$-calculus.

The basic computational step of the $\lambda$-calculus is $\beta$-reduction:

$$\beta\,\frac{}{(\lambda x.\, M)N \longrightarrow M\{{}^{N}\!/\!_{x}\}}$$

in which the placeholder $x$ is replaced by the argument $N$ in the body $M$ of the function. An expression of the form $(\lambda x.\, M)N$ is called a *$\beta$-redex*, and the derivative $M\{{}^{N}\!/\!_{x}\}$ is its *contractum.*

The following rules of inference allow us to replace a $\beta$-redex by its contractum in any context:

$$\mu\,\frac{M \longrightarrow M'}{MN \longrightarrow M'N} \qquad \nu\,\frac{N \longrightarrow N'}{MN \longrightarrow MN'} \qquad \xi\,\frac{M \longrightarrow M'}{\lambda x.\, M \longrightarrow \lambda x.\, M'} \tag{1}$$

We write $M \longrightarrow_\beta N$ if $M \longrightarrow N$ is derivable from the rules $\beta, \mu, \nu, \xi$. Relation $\longrightarrow_\beta$ is the *reduction relation of the $\lambda$-calculus*, also called *full $\beta$-reduction.*

Relation $\longrightarrow_\beta$ defines a directed form of rewriting. A $\lambda$-term $M$ without $\beta$-redexes (i.e., for which no $N$ exists such that $M \longrightarrow_\beta N$) is a *normal form* (briefly *nf*). A *reduction path* is a sequence of reduction steps $M_1 \longrightarrow_\beta M_2 \longrightarrow_\beta \ldots$ that may have finite or infinite length.

The axiom $\beta$ and the rules of inference (1) define a single-step reduction relation on $\lambda$-terms. Adding rules for reflexivity and transitivity we obtain the multistep reduction relation, $\Longrightarrow_\beta$. The resulting formulas $M \Longrightarrow_\beta N$ define the *formal theory of $\beta$-reduction* of the $\lambda$-calculus. The predicate $\downarrow$ distinguishes the abstractions; that is, $M\downarrow$ holds just if $M$ is of the form $\lambda x.\, N$. Further,

$$\mu\frac{M = M'}{MN = M'N} \qquad \nu\frac{N = N'}{MN = MN'} \qquad \xi\frac{M = M'}{\lambda x.\, M = \lambda x.\, M'}$$

$$\textsc{Refl}\frac{}{M = M} \qquad \textsc{Sym}\frac{M = N}{N = M} \qquad \textsc{Trans}\frac{M = N \qquad N = L}{M = L}$$

Table 1: The rules for a $\lambda$-calculus congruence

---

$M \Downarrow_\beta N$ means that $M \Longrightarrow_\beta N$ and $N \downarrow$, and $M \Downarrow_\beta$ means that $M \Downarrow_\beta N$ for some $N$.

Turning the oriented rules defining $\Longrightarrow_\beta$ into equations gives rise to the $\lambda\beta$ *theory*, also called the *formal theory of $\beta$ equality*. It is defined by the axiom

$$\beta\frac{}{(\lambda x.\, M)N = M\{^{N}\!/\!_{x}\}}$$

plus the axiom and inference rules for congruence in Table 1. We write $\lambda\beta \vdash M = N$ if $M = N$ can be proved in the $\lambda\beta$ theory.

We give names to some special $\lambda$-terms:

$$\begin{array}{rcl} I & \stackrel{\text{def}}{=} & \lambda x.\, x \\ \Omega & \stackrel{\text{def}}{=} & (\lambda x.\, xx)(\lambda x.\, xx) \\ \Xi & \stackrel{\text{def}}{=} & (\lambda x.\, \lambda y.\, xx)(\lambda x.\, \lambda y.\, xx). \end{array}$$

$I$ is the identity function, because for all $N$, we have $IN \longrightarrow_\beta N$; we may call $\Omega$ a "purely divergent term", because $\Omega \longrightarrow_\beta \Omega \longrightarrow_\beta \ldots$; and we may call $\Xi$ a "purely convergent term", because $\Xi \longrightarrow_\beta \lambda y.\, \Xi \longrightarrow_\beta (\lambda y.\,)^2\Xi \ldots \longrightarrow_\beta \ldots$; indeed, for all $n$, we have $\Xi \Longrightarrow_\beta (\lambda y.\,)^n\Xi$.

# 2 Contrasting $\lambda$ and $\pi$

Without doubt the $\lambda$-calculus had an important influence on the development of process calculi like CCS, CSP and ACP in the early 80's. In the case of calculi for mobile processes and higher-order calculi, an important heritage from the $\lambda$-calculus is static binding, a concept that is understood largely as a result of work on $\lambda$-calculus.

Two important features of $\lambda$-calculus that distinguish it from $\pi$-calculus are *sequentiality* and *confluence*. We discuss these below. First, we briefly compare the basic reduction axioms of the two calculi.

## 2.1 Reduction axioms

The basic reduction axioms of $\lambda$-calculus and $\pi$-calculus are $\beta$ and COM:

$$(\lambda x.\, M)N \longrightarrow M\{^{N}\!/\!_{z}\} \qquad p(x).\, P \mid \overline{p}\langle a\rangle.\, Q \longrightarrow P\{^{a}\!/\!_{x}\} \mid Q$$

There are three noticeable differences:

1. The $\lambda$-terms $\lambda x. M$ and $N$ are committed to interacting with each other, even if $(\lambda x. M)N$ is part of a larger term; this is so because different $\lambda$-redexes act on different parts of a term. In contrast, interference from the environment can prevent the interaction between the $\pi$-calculus terms $p(x). P$ and $\overline{p}\langle a\rangle. Q$. For instance, the environment could contain a process $\overline{p}\langle r\rangle. R$ that is in competition with $\overline{p}\langle a\rangle. Q$ for access to $p(x). P$. For this reason, $\pi$-calculus is not confluent. Using a terminology from term-rewriting, we may say that the $\lambda$-calculus is *orthogonal*, whereas the $\pi$-calculus is not.

2. The $\lambda$-calculus reduction requires a term substitution, whereas the $\pi$-calculus reduction requires a (simpler) name substitution.

3. The $\lambda$-reduction is asymmetric: the argument $N$ is completely swallowed by the function $\lambda x. M$. The $\pi$-reduction is more symmetric (and in variants such as $\pi$I [San96] it *is* symmetric) as both interacting subterms persist after the reduction.

## 2.2 Sequentiality

The $\lambda$-calculus captures *sequential* computations. That is, $\lambda$-terms express functions that, algorithmically speaking, look at the arguments they need in sequence. If in doing so a function reaches a divergent argument, then the whole computation will diverge. The $\lambda$-calculus, in its untyped or typed versions, cannot describe functions whose algorithmic definition requires that some arguments be run in parallel. An enlightening example is the non-definability in PCF (a typed $\lambda$-calculus with fixed points, booleans and integers) of a 'parallel or' function `Por`, where for closed terms $M$ and $N$,

$$\texttt{Por}\ MN \begin{cases} \text{reduces to } \texttt{true}\text{, if } M \text{ or } N \text{ reduces to } \texttt{true} \\ \text{diverges, otherwise.} \end{cases}$$

With parallelism, this function is algorithmically easy to compute: just let $M$ and $N$ reduce concurrently and return `true` if and when one of them evaluates to `true`.[1] There is no PCF term that behaves as `Por`. The sequentiality of the untyped $\lambda$-calculus is already clear, at least intuitively, in the *normalisation theorem*, a corollary of one of the main syntactic theorems of the $\lambda$-calculus, the *standardisation theorem*. The normalisation theorem asserts that if a term has a normal form, then this will be found by the *leftmost* strategy. This strategy selects the reduction path of a term in which, at each stage, the contracted redex is the one with the leftmost $\lambda$-symbol.

The leftmost strategy is clearly sequential: once begun, the evaluation of a subterm continues until a value (a $\lambda$-abstraction) is found; only then can control pass to another subterm. Therefore simultaneous or interleaved evaluation of subterms is forbidden.

In contrast, the $\pi$-calculus naturally describes parallel computations. The $\pi$-calculus 'world' is that of processes, rather than functions. Since functions can

[1] In the untyped $\lambda$-calculus, where there are no ground data types and abstractions are the only closed values, 'parallel or' should be probably defined as the parallel convergence test (Section 11, or sensible versions of it such as in [Bar84, pag 375]). However, in typed $\lambda$-calculi the two operators are different: parallel convergence is usually more powerful (one can derive 'parallel or' from it, but the converse is often false).

be seen as special forms of processes, parallel functions like `Por` can be described in the $\pi$-calculus (we show the encodings of similar functions in Sections 11 and 12.2).

### 2.3 Confluence

The reduction relations of both $\lambda$-calculus and $\pi$-calculus are non-deterministic, because a term may have more than one redex. However only in the $\pi$-calculus is non-determinism semantically significant. Consider a term

$$Q \stackrel{\text{def}}{=} \overline{a}\langle b\rangle \mid \overline{a}\langle c\rangle \mid a(x).\,\overline{x}$$

The two outputs are in competition for the input. If $\overline{a}\langle b\rangle$ wins, then $\overline{a}\langle c\rangle$ may remain without a partner, and conversely if $\overline{a}\langle c\rangle$ wins. Indeed, the two immediate derivatives of $Q$, namely $\mathbf{0} \mid \mathbf{0} \mid \overline{b}$ and $\mathbf{0} \mid \mathbf{0} \mid \overline{c}$ are behaviourally very different.

This situation cannot happen in the $\lambda$-calculus, where contracting a redex never damages other redexes, in the sense of leaving a subterm without a partner. The fact that the non-determinism of the $\lambda$-calculus is harmless is expressed by the *confluence* (often called *Church-Rosser*) property, that says that if a term $M$ has two derivatives $N_1$ and $N_2$, then we can always find a third term $L$ to close the diamond:

$$\begin{tikzcd} & M \arrow[dl, Rightarrow, "\beta"] \arrow[dr, Rightarrow, "\beta"'] & \\ N_1 \arrow[dr, Rightarrow, "\beta"'] & & N_2 \arrow[dl, Rightarrow, "\beta"] \\ & L & \end{tikzcd}$$

Some consequences of the CR property of the $\lambda$-calculus are: (1) the normal form of a term $M$ (if it exists) is unique, up to alpha-conversion; (2) the $\lambda$-calculus is consistent, that is there are terms $M$ and $N$ such that $\lambda\beta \vdash M = N$ does not hold (just take $M$ and $N$ to be different normal forms, for instance $\lambda x.\,x$ and $\lambda x.\,xy$; by point (1) they are not provably equal); (3) the order in which redexes of a term are reduced is unimportant, in the sense that all finite reduction sequences can be continued to reach a common derivative.

Point (3) does not imply that all reduction paths must meet. For instance, a term with a normal form may have an infinite reduction sequence that never finds it. An example is the term $(\lambda x.\,I)\Omega$; it has normal form $I$, obtained by contracting the outermost redex, but it also has the infinite reduction sequence

$$(\lambda x.\,I)\Omega \longrightarrow_\beta (\lambda x.\,I)\Omega \longrightarrow_\beta \ldots$$

obtained by contracting the innermost redex.

## 3 Reduction strategies for the $\lambda$-calculus

As the reduction relation $\longrightarrow_\beta$ of the $\lambda$-calculus is non-deterministic, a term may have more than one $\beta$-redex and therefore several reduction paths. Some of these may lead to a normal form while others may not (as illustrated for the term $(\lambda x.\, I)\Omega$ above).

Different paths from a term to its normal form may have different lengths. A *reduction strategy* specifies which $\beta$-redexes in a term may be contracted. Reduction strategies are useful both for theoretical reasons (for instance, to prove that a term has no normal form), and for practical purposes (for instance, to obtain efficient implementations). Usually a reduction strategy is deterministic; in that case the one-step reduction relation is a partial function on $\lambda$-terms.

Formally, a reduction strategy $R$ is defined by fixing a reduction relation $\longrightarrow_R \subseteq \Lambda \times \Lambda$ that we will usually write in infix notation. The reflexive and transitive closure of $\longrightarrow_R$ is $\Longrightarrow_R$. We say that $M$ *is an $R$-normal form* (*$R$-nf*) if there is no $N$ such that $M \longrightarrow_R N$; and that $R$ *has an $R$-normal form* if there is a $R$-nf $N$ such that $M \Longrightarrow_R N$. We also write $M \Downarrow_R N$ if $M \Longrightarrow_R N \downarrow$, and $M \Downarrow_R$ if $M \Downarrow_R N$ for some $N$. For instance, $\lambda x.\, M$ is a normal form for any reduction strategy that does not allow the $\xi$ rule.

An example of a strategy is the leftmost strategy mentioned in Section 2.2. Important strategies in programming languages are *call-by-name* and *call-by-value*, and variants of them such as *call-by-need*, *parallel call-by-value* and *strong call-by-name*.

A notion of reduction gives rise to a $\lambda$-theory, when one adds the rules that turn the reduction relation into a congruence relation.

There is no 'best' reduction strategy. Different languages, even single languages, adopt different strategies for the evaluation of function or procedure applications. The coding or implementation of different reduction strategies may require different techniques (there are also reduction strategies that are non-computable, and hence impossible to implement!). Below we consider a few important reduction strategies; then we show how they can be encoded in $\pi$-calculus.

## 3.1 Call-by-name

The idea of the *call-by-name* strategy is that the redex is always the leftmost, but reduction should stop when a constructor is at the top; in the untyped $\lambda$-calculus the only constructor is $\lambda$-abstraction; in typed $\lambda$-calculi there may also be constructors for data, such as '`cons`' for lists. A benefit of not continuing the evaluation underneath an abstraction or a data constructor is that among the call-by-name normal forms are terms representing infinite objects, for instance a term that evaluates to the list of all natural numbers. Therefore one can write meaningful programs for manipulating infinite objects. These programs diverge under other evaluation strategies.

On open terms, the call-by-name strategy also stops on terms with a variable in head position, that is terms of the form $xM_1 \ldots M_n$; this is so because, intuitively, we need to know what the variable is instantiated to in order to decide what reduction to do next. The *one-step call-by-name reduction relation* $\longrightarrow_{\mathtt{N}} \subseteq \Lambda \times \Lambda$ is defined by the two rules

$$\beta \frac{}{(\lambda x.\, M)N \longrightarrow_{\mathtt{N}} M\{^{N}\!/\!_{x}\}} \qquad \mu \frac{M \longrightarrow_{\mathtt{N}} M'}{MN \longrightarrow_{\mathtt{N}} M'N}$$

Reduction is deterministic: the redex is always at the extreme left of a term. As the $\xi$ rule is absent, evaluation does not continue underneath an abstraction; and as the $\nu$ rule is absent, the argument of a function is not evaluated. Examples of call-by-name reductions are

$$(\lambda x.\, I)\Omega \longrightarrow_{\mathtt{N}} I \tag{2}$$

$$(\lambda x.\, xx)(II) \longrightarrow_{\mathtt{N}} (II)(II) \longrightarrow_{\mathtt{N}} I(II) \longrightarrow_{\mathtt{N}} II \longrightarrow_{\mathtt{N}} I \tag{3}$$

$$(\lambda xy.\, x)z(II) \longrightarrow_{\mathtt{N}} (\lambda y.\, z)(II) \longrightarrow_{\mathtt{N}} z\ . \tag{4}$$

Since call-by-name is based on the $\beta$ rule, the $\lambda$-theory induced by call-by-name is the same as $\lambda\beta$. Therefore a correct semantics of call-by-name should (at least) validate the equalities of $\lambda\beta$.

### 3.2 Call-by-value

In the call-by-name strategy, the contraction of a redex $(\lambda x.\, M)N$ is performed without restriction on the form of the argument $N$. This contrasts with the *call-by-value* (or *eager*) strategy, where the argument $N$ is reduced to a *value* before the redex is contracted. The values of the untyped call-by-value $\lambda$-calculus are the functions (that is, the $\lambda$-abstractions) and, on open terms, also the variables (it makes sense that variables be values because in call-by-value substitutions replace variables with values—not arbitrary terms—and therefore the closed terms that can be obtained from a variable are closed values):

$$\text{Values} \qquad V := \lambda x.\, M \;\Big|\; x \qquad\qquad M \in \Lambda$$

The *one-step call-by-value reduction relation* $\longrightarrow_{\mathtt{V}} \subseteq \Lambda \times \Lambda$ is defined by these rules:

$$\beta_{\mathtt{v}}\,\frac{}{(\lambda x.\, M)V \longrightarrow_{\mathtt{V}} M\{{}^{V}\!/\!_{x}\}}$$

$$\mu\,\frac{M \longrightarrow_{\mathtt{V}} M'}{MN \longrightarrow_{\mathtt{V}} M'N} \qquad \nu_{\mathtt{v}}\,\frac{N \longrightarrow_{\mathtt{V}} N'}{(\lambda x.\, M)N \longrightarrow_{\mathtt{V}} (\lambda x.\, M)N'}$$

Examples of call-by-value reductions are

$$(\lambda x.\, I)\Omega \longrightarrow_{\mathtt{V}} (\lambda x.\, I)\Omega \longrightarrow_{\mathtt{V}} \ldots \tag{5}$$

$$(\lambda x.\, xx)(II) \longrightarrow_{\mathtt{V}} (\lambda x.\, xx)I \longrightarrow_{\mathtt{V}} II \longrightarrow_{\mathtt{V}} I \tag{6}$$

$$(\lambda xy.\, x)z(II) \longrightarrow_{\mathtt{V}} (\lambda y.\, z)(II) \longrightarrow_{\mathtt{V}} (\lambda y.\, z)I \longrightarrow_{\mathtt{V}} z \tag{7}$$

Call-by-value is based on the $\beta_{\mathtt{v}}$ rule; the $\lambda$-theory induced by this rule is the $\lambda\beta_{\mathtt{v}}$ *theory*, also called the *formal theory of* $\beta_{\mathtt{v}}$ *equality*, defined by the axiom

$$\beta_{\mathtt{v}}\,\frac{}{(\lambda x.\, M)V = M\{{}^{V}\!/\!_{x}\}}$$

plus the inference rules in Table 1. A correct semantics of call-by-value should (at least) validate the equalities of $\lambda\beta_{\mathtt{v}}$.

Call-by-value is very common in language implementations. Advantages of call-by-value are: (i) if in $(\lambda x.\, M)N$ the variable $x$ occurs more than once in the body $M$, then evaluating $N$ before replacing $x$ may avoid having to reduce several copies of $N$ (contrast example (3) with (6)); and (ii) in languages with side effects, call-by-value is easier to understand and mathematically more tractable. On the other hand, call-by-name has the advantages that: (i) one does not perform useless reductions of the argument $N$ of a redex $(\lambda x.\, M)N$ if

$x$ does not occur in $M$ and therefore $N$ is not used (contrast example (4) with (7)); and (ii) on certain terms, the call-by-name strategy terminates whereas call-by-value fails (an example is the term $(\lambda x.\, I)\Omega$, as shown in (2) and (5)).

A variant of call-by-value is *parallel call-by-value* that has the ordinary $\nu$ rule in place of $\nu_{\mathtt{v}}$. Its one-step reduction relation is denoted by $\longrightarrow_{\mathtt{PV}}$. This strategy is not deterministic, because in an application $MN$ both the function and the argument can be reduced. Normal forms are, however, unique (that is, if $M \Longrightarrow_{\mathtt{PV}} M'$ and $M \Longrightarrow_{\mathtt{PV}} M''$ and both $M'$ and $M''$ are PV-nfs, then $M' = M''$), and if $M$ has a PV-nf, then all reduction sequences from $M$ are finite. Because of these properties, behavioural equivalence under parallel call-by-value is usually the same as that under (sequential) call-by-value (we shall talk about behavioural equivalences for $\lambda$-calculus in Section 11).

### 3.3 Call-by-need

In a language without side effects, where the evaluation of a term always yields the same result, the inefficiency problems of call-by-name arising from repeated evaluation of copies of the argument of a function can be avoided as follows. The first time the argument is evaluated, its value is saved in an environment; if needed subsequently, the value is fetched from the environment. In this way, the evaluation of the argument is shared among all places where the argument is used. This implementation technique is called *call-by-need.* It is usually presented as a reduction strategy on graphs, where *sharing* of subterms is easy to represent. Alternatively, call-by-need can be formalised in a $\lambda$-calculus with a `let` construct, to model sharing. The $\beta$ rule is replaced by the `let` rule

$$\frac{}{(\lambda x.\, M)N \longrightarrow_{\mathtt{NE}} \mathtt{let}\ x = N\ \mathtt{in}\ M}\, x \notin \mathrm{fv}(N) \tag{8}$$

In the derivative, the evaluation continues on $M$ and only when the value associated to $x$ is needed, is the subterm $N$ evaluated. The value to which $N$ reduces replaces the occurrence of $x$ in question, and all subsequent occurrences of $x$ when their value is needed. There are also some structural rules for manipulating `let` expressions, that usually make use of a notion of *evaluation context* to define the next redex.

For instance, in call-by-need we have

$$\begin{array}{ll} & II = (\lambda x.\, x)(\lambda y.\, y) \\ \longrightarrow_{\mathtt{NE}} & \mathtt{let}\ x = \lambda y.\, y\ \mathtt{in}\ x \\ \longrightarrow_{\mathtt{NE}} & \mathtt{let}\ x = \lambda y.\, y\ \mathtt{in}\ \lambda z.\, z \\ \equiv & \lambda z.\, z = I \end{array} \tag{9}$$

where $\equiv$ indicates application of the garbage-collection rule

$$(\ \mathtt{let}\ x = M\ \mathtt{in}\ N) \equiv N \qquad x \notin \mathrm{fv}(N) \tag{10}$$

(Garbage-collection rules may or may not be present in the definition of a call-by-need reduction; in any case, a rule such as (10) can be proved valid for behavioural equivalences.) Similarly, we have

$$Iy \Longrightarrow_{\mathtt{NE}} \equiv y \tag{11}$$

Using derivations (9) and (11) to compress reductions, here is a more interesting call-by-need computation. Note that the work for the evaluation of the argument $II$ is done only once:

$$
\begin{array}{lll}
 & & (\lambda x.\,(Ix)x)(II) \\
\longrightarrow_{\mathtt{NE}} & & \mathtt{let}\ x = II\ \mathtt{in}\ (Ix)x \\
\Longrightarrow_{\mathtt{NE}} & \equiv & \mathtt{let}\ x = II\ \mathtt{in}\ xx \\
\Longrightarrow_{\mathtt{NE}} & \equiv & \mathtt{let}\ x = I\ \mathtt{in}\ xx \\
\longrightarrow_{\mathtt{NE}} & & \mathtt{let}\ x = I\ \mathtt{in}\ Ix \\
\Longrightarrow_{\mathtt{NE}} & \equiv & \mathtt{let}\ x = I\ \mathtt{in}\ x \\
\longrightarrow_{\mathtt{NE}} & \equiv & I
\end{array}
$$

We shall not present a formal system for $\lambda$-calculus with `let` ; the intuitions given above should be enough to understand our uses of call-by-need.

As call-by-need is an implementation technique for call-by-name, its theory is closely related to that of call-by-name. The sets of $\lambda$-terms that have a normal form under call-by-need and call-by-name coincide; and two $\lambda$-terms are behaviourally equivalent in call-by-need iff they are so in call-by-name.

# 4 Interpreting $\lambda$-calculi into $\pi$-calculus

All of the translations of $\lambda$-calculus strategies into $\pi$-calculus that we shall present have two common features:

- Function application is translated as a form of parallel combination of two processes, the function and its argument, and $\beta$-reduction is modelled as an interaction between them.
- The encoding of a $\lambda$-term is parametric over a name. This name is used by (the translation of) the $\lambda$-term to interact with the environment.

In Section 2.1 we observed that a redex of the $\lambda$-calculus gives rise to a private interaction between two terms. In contrast, a redex of the $\pi$-calculus is susceptible to interference from the environment. This interference is avoided if the name used by the two processes to communicate is private to them. Therefore, the appearance of a $\beta$-redex in a $\lambda$-term should correspond, in the $\pi$-calculus translation of that term, to the appearance of two processes that can communicate along a *private* name.

We also observed in Section 2.1 that the $\beta$ rule of the $\lambda$-calculus is strongly asymmetric. To cope with this, the $\pi$-calculus language that we will use for encoding $\lambda$-calculus strategies is the *asynchronous $\pi$-calculus*, whose asymmetry of the communication rule reflects that of the $\lambda$-calculus. Recall that in the asynchronous $\pi$-calculus there is no process underneath an output prefix. For convenience, however, we sometimes use output prefixes of the form $\boldsymbol{\nu}\widetilde{b}\,\overline{a}\langle\widetilde{c},b\rangle.\,\pi.\,P$, where $b \in \widetilde{b}$ and $b$ is the subject of prefix $\pi$, to highlight ordering among actions (under those hypotheses, $\boldsymbol{\nu}\widetilde{b}\,\overline{a}\langle\widetilde{c},b\rangle.\,\alpha.\,P$ is semantically the same as $\boldsymbol{\nu}\widetilde{b}\,(\overline{a}\langle\widetilde{c},b\rangle \mid \alpha.\,P)$).

A name parameter is needed in the $\pi$-calculus encodings of $\lambda$-terms for the following reason. Roughly speaking, in the $\lambda$-calculus, $\lambda$ is the only port; a $\lambda$-term receives its argument at $\lambda$. In the $\pi$-calculus, there are many ports, so one needs to specify at which port (the encoding of) a $\lambda$-term interacts with its environment.

## 4.1 Continuation Passing Style

The parameter of the $\pi$-calculus encoding of a function can also be thought of as a *continuation*. In functional languages, a continuation is a parameter of a function that represents the 'rest' of the computation. Functions taking continuations as arguments are called *functions in Continuation Passing Style* (briefly *CPS functions*), and have a special syntactic form: they terminate their computation by passing the result to the continuation. The continuation parameter may also be thought of as an address to which the result of the function is to be delivered. For an informal example, take the following function from integers to integers:

$$f \stackrel{\text{def}}{=} \lambda y.\ \mathtt{let}\ g = \lambda n.\, n + 2\ \mathtt{in}\ (gy) + (gy)$$

Here is a CPS version of $f$; in the body, $g_{\text{CPS}}$ is a CPS version of $g$, and $k$ is the continuation parameter:

$$\begin{aligned} f_{\text{CPS}} \stackrel{\text{def}}{=} \lambda k\, y.\, \mathtt{let}\ g_{\text{CPS}} &= \lambda k\, n.\ \ \mathtt{let}\ m = n + 2\ \ \mathtt{in}\ k(m) \\ &\mathtt{in}\ g_{\text{CPS}}\Big(\lambda v.\, g_{\text{CPS}}\Big(\lambda w.\ \ \mathtt{let}\ u = u + n\ \ \mathtt{in}\ k(u)\Big)y\Big)y \end{aligned}$$

In programming languages continuations are widely used, for programming, as an implementation technique (to generate an intermediate language that is easier to optimise and manipulate), and for giving denotational semantics. A fairly vast literature of functional programming studies transformations of functions into CPS functions. They are called CPS transforms. The best known are the CPS transforms for call-by-name and call-by-value $\lambda$-calculus studied by Plotkin in his seminal paper [Plo75].

We shall develop the analogy between $\pi$-calculus encodings and CPS transforms. We shall derive $\pi$-calculus encodings of call-by-name and call-by-value $\lambda$-calculus via the CPS transforms of [Plo75]. We shall observe that the target of the CPS transforms are essentially subcalculi of HO$\pi$. We shall therefore be able to apply compilation $\mathcal{C}$ from HO$\pi$ to $\pi$-calculus to derive $\pi$-calculus encodings ($\mathcal{C}$ is defined in Section D). This is the programme for Sections 5 and 6, and is summarised in Figure 1, where: $\lambda N$ and $\lambda V$ are the call-by-name and call-by-value $\lambda$-calculi; $\mathcal{C}_{\mathtt{V}}$ and $\mathcal{C}_{\mathtt{N}}$ are the call-by-value and the call-by-name CPS transforms; $\text{CPS}_{\mathtt{V}}$ and $\text{CPS}_{\mathtt{N}}$ are languages of CPS $\lambda$-terms, that is the target languages of the two CPS transforms; $\mathcal{H}$ is the injection of these CPS languages into HO$\pi$; and $\mathcal{C}$ is the compilation from HO$\pi$ to $\pi$-calculus. There is a vast literature on CPS transforms; we used the transforms that, in our view, yield the simplest and most robust encodings.

The schema of Figure 1 also applies to *typed* $\lambda$-calculi, by extending the translations of terms to translations of types.

Having obtained encodings of call-by-name and call-by-value, in Sections 5.3, 7, and 9 we can then play with them and derive encodings of similar strategies, such as call-by-need, parallel call-by-value, and strong call-by-name.

**Convention.** For simplicity, we adopt the convention that $\lambda$-calculus variables are also $\pi$-calculus names.

$$
\begin{tikzcd}
\lambda V \arrow[dr, "\mathcal{C}_{\mathtt{V}}"] \arrow[dddrr, bend right, "\mathcal{V}"'] & & & & \lambda N \arrow[dl, "\mathcal{C}_{\mathtt{N}}"'] \arrow[dddll, bend left, "\mathcal{N}"] \\
& \mathrm{CPS}_{\mathtt{V}} \arrow[dr, "\mathcal{H}"] & & \mathrm{CPS}_{\mathtt{N}} \arrow[dl, "\mathcal{H}"'] & \\
& & \mathrm{HO}\pi \arrow[d, "\mathcal{C}"'] & & \\
& & \pi & &
\end{tikzcd}
$$

Figure 1: The derivations of the $\pi$-calculus encodings of $\lambda N$ and $\lambda V$.

---

# 5 The interpretation of call-by-value

In this section we develop the left part of the diagram of Figure 1. The $\pi$-calculus encoding of call-by-value $\lambda$-calculus ($\lambda V$) is obtained in three steps, the first of which is the call-by-value CPS transform of [Plo75]. The reader who is eager to see the $\pi$-calculus encoding, and does not want to go through the CPS transform, may go directly to Section 5.2.

We will not usually give detailed proofs of results about the CPS transforms, as they are not the subject of the paper; see Section 14 for references.

## 5.1 The three steps

### Step 1: the call-by-value CPS transform

The *call-by-value CPS transform*, $\mathcal{C}_{\mathtt{V}}$, transforms functions of $\lambda V$ into CPS functions. In its definition, the translation of values uses the auxiliary translation function $\mathcal{C}^{\star}_{\mathtt{V}}$ (which will be particularly useful when considering types). We call a term $\mathcal{C}^{\star}_{\mathtt{V}}[V]$ a *CPS-value*. The transform is presented in Table 2. Its definition introduces a new variable, the *continuation variable* $k$, that represents continuations and that has to be kept separate from the other variables. In the $\pi$-calculus encodings, and in typed versions of the CPS transform, the continuation variable and the other variables will have different types. (In the definition of $\mathcal{C}_{\mathtt{V}}$ we also use special symbols $v, w$ for the formal parameters of continuations. The distinction between these variables and ordinary variables is, however, somewhat artificial because the former may be instantiated by the latter.)

We explain informally how the CPS transform works. The CPS image of a $\lambda$-term $L$ immediately needs a continuation. When a continuation is provided, $L$ is reduced to a value, and this value (precisely its CPS-value) is passed to the continuation. Therefore if $L$ is itself a value, then it can be passed directly to the continuation. If, however, $L$ is an application $MN$ then the following happens. First $M$ is evaluated with continuation $\lambda v.\, \mathcal{C}_{\mathtt{V}}[\![N]\!](\lambda w.\, vwk)$. When $M$ becomes a function, say $\lambda x.\, M_1$, this function is passed to the continuation, and the body

*In this table we abbreviate* $\mathcal{C}_{\mathtt{V}}[\![M]\!]$ *as* $[\![M]\!]$ *and* $\mathcal{C}^{\star}_{\mathtt{V}}[V]$ *as* $[V]$*:*

$$\text{Call-by-value values} \quad V \ := \ \lambda x.\, M \ \big| \ x$$

$$\begin{aligned}
[\![V]\!] &\stackrel{\text{def}}{=} \lambda k.\, k[V] \\
[\![MN]\!] &\stackrel{\text{def}}{=} \lambda k.\, [\![M]\!](\lambda v.\, [\![N]\!](\lambda w.\, vwk))
\end{aligned}$$

$$\begin{aligned}
[x] &\stackrel{\text{def}}{=} x \\
[\lambda x.\, M] &\stackrel{\text{def}}{=} \lambda x.\, [\![M]\!]
\end{aligned}$$

Table 2: The call-by-value CPS transform

---

of the continuation is evaluated. This means evaluating $N$ with continuation $\lambda w.\, vwk\{\mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]/v\}$. When $N$ in turn becomes a value $V$, this value is passed to the continuation, and the body of the continuation is evaluated. This body is the term $(vwk)\{\mathcal{C}^{\star}_{\mathtt{V}}[V]/w\}\{\mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]/v\}$, that is $(\lambda x.\, \mathcal{C}_{\mathtt{V}}[\![M_1]\!])\mathcal{C}^{\star}_{\mathtt{V}}[V]k$. This term reduces to $\mathcal{C}_{\mathtt{V}}[\![M_1]\!]\{\mathcal{C}^{\star}_{\mathtt{V}}[V]/x\}k$, which is the same as $\mathcal{C}_{\mathtt{V}}[\![M_1\{V/x\}]\!]k$. The reduction of $\mathcal{C}_{\mathtt{V}}[\![M_1\{V/x\}]\!]k$ continues and, at the end, the value that $M_1\{V/x\}$ reduces to is passed to $k$. Note that the flow of control of $\lambda V$ on application is correctly mimicked: first the operator $M$ of the application is evaluated, then the argument $N$ is evaluated, and finally the two derivatives of $M$ and $N$ are contracted.

To help understanding the behaviour of application, we report below the details of how a $\beta_{\mathtt{V}}$ reduction

$$(\lambda x.\, M_1)V \longrightarrow_{\mathtt{V}} M_1\{V/x\} \tag{12}$$

is simulated. We use the call-by-name reduction $\longrightarrow_{\mathtt{N}}$ for the target CPS terms, but we could just as well have chosen call-by-value, since these strategies coincide on CPS terms (see Remark 5.3).

$$\begin{aligned}
& \mathcal{C}_{\mathtt{V}}[\![(\lambda x.\, M_1)V]\!]k \\
= \ & \Big(\lambda k.\, \mathcal{C}_{\mathtt{V}}[\![\lambda x.\, M_1]\!](\lambda v.\, \mathcal{C}_{\mathtt{V}}[\![V]\!](\lambda w.\, vwk))\Big)k \\
\longrightarrow_{\mathtt{N}} \ & \mathcal{C}_{\mathtt{V}}[\![\lambda x.\, M_1]\!](\lambda v.\, \mathcal{C}_{\mathtt{V}}[\![V]\!](\lambda w.\, vwk)) \\
= \ & \lambda h.\, h\mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1](\lambda v.\, \mathcal{C}_{\mathtt{V}}[\![V]\!](\lambda w.\, vwk)) \\
\longrightarrow_{\mathtt{N}} \ & (\lambda v.\, \mathcal{C}_{\mathtt{V}}[\![V]\!](\lambda w.\, vwk))\mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1] \\
\longrightarrow_{\mathtt{N}} \ & \mathcal{C}_{\mathtt{V}}[\![V]\!](\lambda w.\, \mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]wk) \\
= \ & \lambda h.\, h\mathcal{C}^{\star}_{\mathtt{V}}[V](\lambda w.\, \mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]wk) \\
\longrightarrow_{\mathtt{N}} \ & (\lambda w.\, \mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]wk)\mathcal{C}^{\star}_{\mathtt{V}}[V] \\
\longrightarrow_{\mathtt{N}} \ & \mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M_1]\mathcal{C}^{\star}_{\mathtt{V}}[V]k \\
= \ & (\lambda x.\, \mathcal{C}_{\mathtt{V}}[\![M_1]\!])\mathcal{C}^{\star}_{\mathtt{V}}[V]k \\
\longrightarrow_{\mathtt{N}} \ & \mathcal{C}_{\mathtt{V}}[\![M_1]\!]\{\mathcal{C}^{\star}_{\mathtt{V}}[V]/x\}k \\
= \ & \mathcal{C}_{\mathtt{V}}[\![M_1\{V/x\}]\!]k
\end{aligned} \tag{13}$$

All but the last reduction can be regarded as *administrative* reductions, because they do not correspond to reductions of the source terms. The last reduction can be regarded as a *proper* reduction, because it directly corresponds to the reduction on the source terms.

The call-by-value CPS transform maps $\lambda$-terms onto a subset of the $\lambda$-terms. The closure of that subset under $\beta$-conversion gives the language $\mathrm{CPS}_{\mathtt{V}}$ of the call-by-value CPS:

$$\mathrm{CPS}_{\mathtt{V}} \stackrel{\text{def}}{=} \{A \;:\; \exists M \in \Lambda \text{ with } \mathcal{C}_{\mathtt{V}}[\![M]\!] \Longrightarrow_{\beta} A\}$$

We will call the terms of $\mathrm{CPS}_{\mathtt{V}}$ the *CPS terms*.

The first theorem shows that on CPS terms, $\beta$- and $\beta_{\mathtt{v}}$-redexes coincide.

**Theorem 5.1 (indifference of $\mathrm{CPS}_{\mathtt{V}}$ on reductions)** *Let $M \in \mathrm{CPS}_{\mathtt{V}}$ and let $N$ be any subterm of $M$. For all $N'$, we have: $N \longrightarrow_{\mathtt{N}} N'$ iff $N \longrightarrow_{\mathtt{V}} N'$.*

**Proof:** Below we shall give a grammar, Grammar (15), that generates all CPS terms. It is immediate to check that on terms generated by that grammar, $\beta$-redexes and $\beta_{\mathtt{v}}$-redexes coincide, because all arguments of functions are values of $\lambda V$ (abstractions and variables). ■

Essentially as a consequence of Theorem 5.1, we obtain

**Theorem 5.2 (indifference of $\mathrm{CPS}_{\mathtt{V}}$ on $\lambda$-theories)** *For all $M, N \in \mathrm{CPS}_{\mathtt{V}}$, we have: $\lambda\beta \vdash M = N$ iff $\lambda\beta_{\mathtt{v}} \vdash M = N$.*

**Proof:** The implication from right to left holds because a $\beta_{\mathtt{v}}$-redex is also a $\beta$-redex. We consider the implication from left to right. Since full $\beta$-reduction $\Longrightarrow_{\beta}$ is confluent, $\lambda\beta \vdash L_1 = L_2$ implies that there is $L_3$ such that $L_1 \Longrightarrow_{\beta} L_3$ and $L_2 \Longrightarrow_{\beta} L_3$. When $L_1$ and $L_2$ are CPS terms, $L_3$ is also a CPS term. By Theorem 5.1, all $\beta$-redexes contracted in the reductions $L_1 \Longrightarrow_{\beta} L_3$ and $L_2 \Longrightarrow_{\beta} L_3$ are also $\beta_{\mathtt{v}}$-redexes. Hence $\lambda\beta_{\mathtt{v}} \vdash L_1 = L_2$. ■

**Remark 5.3** These indifference properties allow us to take either call-by-name or call-by-value as the reduction strategy and the $\lambda$-theory on CPS terms. We choose the call-by-name versions, because they are simpler.

The next two theorems are about the correctness of the CPS transform. The first shows that the computation of a $\lambda$-term is correctly mimicked by its CPS image. The second shows that the CPS transform preserves $\beta_{\mathtt{v}}$-conversion.

**Theorem 5.4 (adequacy for $\mathcal{C}_{\mathtt{V}}$)** *Let $M \in \Lambda^0$.*

1. *If $M \Longrightarrow_{\mathtt{V}} V$ then $\mathcal{C}_{\mathtt{V}}[\![M]\!]k \Longrightarrow_{\mathtt{N}} k\,\mathcal{C}^{\star}_{\mathtt{V}}[V]$ (note that the term $k\,\mathcal{C}^{\star}_{\mathtt{V}}[V]$ is a $\mathtt{N}$-nf, that is a normal form for call-by-name).*
2. *The converse: If $\mathcal{C}_{\mathtt{V}}[\![M]\!]k \Longrightarrow_{\mathtt{N}} N$ and $N$ is a $\mathtt{N}$-nf, then there is a call-by-value value $V$ such that $M \Longrightarrow_{\mathtt{V}} V$ and $N = k\,\mathcal{C}^{\star}_{\mathtt{V}}[V]$.*

This theorem can be proved by going through an intermediate CPS transform obtained from the original one by removing some administrative reductions. Doing so is useful because administrative reductions complicate the operational correspondence between source and target terms of the CPS transform. This is clear by considering the open term $xx$: this term is not reducible (it is a $\mathtt{V}$-nf), but its image $\mathcal{C}_{\mathtt{V}}[\![xx]\!]k$ has 5 (administrative) reductions.

**Theorem 5.5 (validity of the $\beta$-theory for $\mathcal{C}_{\mathtt{V}}$)** *Let $M, N \in \Lambda$. If $\lambda\beta_{\mathtt{v}} \vdash M = N$ then $\lambda\beta \vdash \mathcal{C}_{\mathtt{V}}[\![M]\!] = \mathcal{C}_{\mathtt{V}}[\![N]\!]$.*

**Proof:** [sketch] First one shows that if $M \Longrightarrow_{\mathtt{V}} N$ then $\lambda\beta \vdash \mathcal{C}_{\mathtt{V}}[\![M]\!] = \mathcal{C}_{\mathtt{V}}[\![N]\!]$. Then one concludes using the fact that $\beta_v$ is confluent. ■

The converse of Theorem 5.5 fails. For instance if

$$
\begin{aligned}
M &\stackrel{\text{def}}{=} \Omega y = (\lambda x.\, xx)(\lambda x.\, xx)y \\
N &\stackrel{\text{def}}{=} (\lambda x.\, xy)\Omega = (\lambda x.\, xy)((\lambda x.\, xx)(\lambda x.\, xx))
\end{aligned}
\tag{14}
$$

then $\lambda\beta \vdash \mathcal{C}_{\mathtt{V}}[\![M]\!] = \mathcal{C}_{\mathtt{V}}[\![N]\!]$ holds, but $\lambda\beta_{\mathtt{v}} \vdash M = N$ does not.

The statements on the correctness of the CPS transform complete step 1 of the left part of Figure 1.

### Step 2: from $\mathrm{CPS}_{\mathtt{V}}$ to HO$\pi$

The next step is to show that, modulo the different syntax for abstraction and application, the terms of $\mathrm{CPS}_{\mathtt{V}}$ are also terms of HO$\pi$. To do this we present a grammar that generates all $\mathrm{CPS}_{\mathtt{V}}$ terms, and show that the terms generated by this grammar are also terms of HO$\pi$.

The grammar has four non-terminals, for *principal terms*, *continuations*, *CPS-values*, and *answers*. Principal terms are abstractions on continuations; they describe the images of the $\lambda$-terms under the CPS transform. CPS-values correspond, intuitively, to the values of $\lambda V$; they are used as arguments to continuations. Answers are the results of computations: what we obtain when we evaluate a principal term applied to a continuation. Answers are the terms in which computation ($\beta$-reductions) takes place.

$$
\begin{array}{rlll}
\text{continuation variable} & k & & \\
\text{ordinary variables} & x, \ldots & & \\
\text{answers} & P & := & KV \;\big|\; VVK \;\big|\; AK \\
\text{CPS-values} & V & := & \lambda x.\, \lambda k.\, P \;\big|\; x \\
\text{continuations} & K & := & k \;\big|\; \lambda x.\, P \\
\text{principal terms} & A & := & \lambda k.\, P
\end{array}
\tag{15}
$$

**Remark 5.6** In the grammar for CPS-values, the production $\lambda x.\, \lambda k.\, P$ can be simplified to $\lambda x.\, A$, but the expanded form is better for the comparison with HO$\pi$ below. ■

**Remark 5.7** Having only one continuation variable guarantees that the continuation occurs free exactly once in the body of each abstraction $\lambda k.\, P$. (When working up to $\alpha$-conversion, the continuation variable $k$ may be renamed, but the linearity constraint on continuations remains.) ■

The relationships among the four categories of non-terminals in Grammar (15) can be expressed using types. Assuming a distinguished type $\diamond$ for answers, the types $T_V$, $T_K$ and $T_A$ of CPS-values, continuations and principal terms are

$$
\begin{aligned}
T_V &\stackrel{\text{def}}{=} \mu X.\, \Big((X \to (X \to \diamond) \to \diamond)\Big) \\
T_K &\stackrel{\text{def}}{=} T_V \to \diamond \\
T_A &\stackrel{\text{def}}{=} T_K \to \diamond
\end{aligned}
\tag{16}
$$

The type judgements for the terms $M$ generated by Grammar (15) are of the form

$$\Gamma \vdash M : T \tag{17}$$

where $T \in \{T_V, T_K, T_A, \diamond\}$ and $\Gamma$ is either $\widetilde{x} : T_V$ or $\widetilde{x} : T_V, k : T_K$, for some $\widetilde{x}$ with $\widetilde{x} \subseteq \mathrm{fv}(M)$. We shall see in Section 10 that there is a general schema for translating type judgements on $\lambda$-terms to type judgements on the CPS-images of the $\lambda$-terms, and that the schema applies also to untyped $\lambda$-calculus.

**Remark 5.8** To be precise we should include some linear information in the types (16), so to show the linear usage of continuations. In the step of encoding into the $\pi$-calculus these linear information allow to avoid some replication operators. We do not show the linear types for these reasons: the linear information is already highlighted in (15) by the syntactic separation between continuation and ordinary variables; linearity is not fundamental—the simplification on replications in the $\pi$-calculus is pretty straightforward and could anyway be omitted; linear types are anyhow not needed in the final $\pi$-calculus encoding—linearity can be "forgotten" when translating types, see the discussion in Section D.0.9. ■

All CPS terms are indeed generated by Grammar (15):

**Proposition 5.9** *If $M \in \mathrm{CPS}_{\mathtt{V}}$, then $M$ is a principal term of Grammar (15).*

**Proof:** One can show that the set of principal terms includes the set $\{\mathcal{C}_{\mathtt{V}}[\![M]\!] : M \in \Lambda\}$ and is closed under $\beta$-conversion. ■

It is easy to see that the set of terms generated by Grammar (15) is, essentially, a subset of HO$\pi$ terms. Precisely, answers may be regarded as HO$\pi$ processes, and CPS-values, continuations, and principal terms as HO$\pi$ abstractions. Recall from Section C that the grammar of (polyadic) HO$\pi$ requires that abstractions be either variables or parametrised processes. CPS-values, continuations, and principal terms of Grammar (15) are indeed of this form, if we read $P$ as a 'process', and we uncurry a CPS-value $\lambda x.\, \lambda k.\, P$ to $\lambda(x,k).\, P$ and an answer $VVK$ to $V\langle V, K\rangle$ (correspondingly, in Table 2 $\mathcal{C}^{\star}_{\mathtt{V}}[\lambda x.\, M]$ becomes $\lambda(x,k).\, \mathcal{C}_{\mathtt{V}}[\![M]\!]k$, and in the translation of $MN$, the term $\lambda w.\, vwk$ becomes $\lambda w.\, v\langle w, k\rangle$). It is therefore straightforward to define the injection $\mathcal{H}$ from the terms of Grammar (15) to HO$\pi$. On terms, modulo this uncurrying, the injection rewrites $\lambda$-abstractions into HO$\pi$ abstractions, and $\lambda$-applications into HO$\pi$ applications (with the obvious type annotations in applications, given by (16) and the translation of types). The injection is the identity on (uncurried) types; thus

$$\mathcal{H}[\![T_V]\!] \stackrel{\mathrm{def}}{=} \mu X.\, \Big( (X \times (X \to \diamond)) \to \diamond \Big) . \tag{18}$$

We have $\mathcal{H}[\![\diamond]\!] \stackrel{\mathrm{def}}{=} \diamond$, which explains the abuse of notation whereby the same symbol $\diamond$ is used for the type of answers of Grammar (15) and for the type of processes of HO$\pi$. The image of $\mathcal{H}$ is a HO$\pi$ language that has recursive and product types.

The proofs of these lemmas are straightforward:

**Lemma 5.10** *Suppose that $M$ is a term generated by Grammar (15), and that $\Gamma \vdash M : T$ for $\Gamma$ and $T$ as in (17). Then $\mathcal{H}[\![\Gamma]\!] \vdash \mathcal{H}[\![M]\!] : \mathcal{H}[\![T]\!]$ .*

$$\mathcal{V}[\![\lambda x.\, M]\!] \stackrel{\mathrm{def}}{=} (p).\, \overline{p}(y).\, !y(x)\, \mathcal{V}[\![M]\!]$$

$$\mathcal{V}[\![x]\!] \stackrel{\mathrm{def}}{=} (p).\, \overline{p}\langle x\rangle$$

$$\mathcal{V}[\![MN]\!] \stackrel{\mathrm{def}}{=} (p).\, \boldsymbol{\nu} q \left(\mathcal{V}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} r\, (\mathcal{V}[\![N]\!]r \mid r(w).\, \overline{v}\langle w, p\rangle)\right)$$

Table 3: The encoding of $\lambda V$ into the $\pi$-calculus

---

As a corollary of Lemma 5.10, if $M \in \Lambda$ with $\mathrm{fv}(M) \subseteq \widetilde{x}$, then

$$\widetilde{x} \,:\, \mathcal{H}[\![T_V]\!] \vdash \mathcal{H}[\![\mathcal{C}_{\mathtt{V}}[\![M]\!]]\!] \,:\, \mathcal{H}[\![T_A]\!] = (\mathcal{H}[\![T_V]\!] \to \diamond) \to \diamond \ . \tag{19}$$

We remind that if $P, Q$ are HO$\pi$ processes, then $P \longrightarrow_{\beta} Q$ and $P =_{\beta} Q$ denote $\beta$-reduction and $\beta$-convertibility in HO$\pi$, respectively (Section C).

**Lemma 5.11** *For all terms $M$ of Grammar (15), we have: if $M \longrightarrow_{\mathtt{N}} M'$ then $\mathcal{H}[\![M]\!] \longrightarrow_{\beta} =_{\beta} \mathcal{H}[\![M']\!]$; and conversely, if $\mathcal{H}[\![M]\!] \longrightarrow_{\beta} P$ then there is $M'$ such that $M \longrightarrow_{\mathtt{N}} M'$ and $P =_{\beta} \mathcal{H}[\![M']\!]$.*

(In the lemma above, the use of $=_{\beta}$ is due to the uncurrying that is used in the injection $\mathcal{H}$.)

**Corollary 5.12** *For all terms $M, N$ generated by Grammar (15), $\lambda\beta \vdash M = N$ implies $\mathcal{H}[\![M]\!] =_{\beta} \mathcal{H}[\![N]\!]$.*

**Proof:** Since $\Longrightarrow_{\beta}$ is confluent, $\lambda\beta \vdash M = N$ holds iff there is some $L$ such that $M \Longrightarrow_{\beta} L$ and $N \Longrightarrow_{\beta} L$. Therefore, by Lemma 5.11, $\mathcal{H}[\![M]\!] =_{\beta} \mathcal{H}[\![L]\!] =_{\beta} \mathcal{H}[\![N]\!]$. ■

This concludes the second step of the left part of Figure 1.

**Step 3: compilation $\mathcal{C}$**

The third and final step for the left part of Figure 1 is from HO$\pi$ to $\pi$-calculus. This step is given by the compilation $\mathcal{C}$ from HO$\pi$ to $\pi$-calculus of Section D. In Section D we showed that there is a precise operational correspondence between source and target terms of the compilation, and studied its behavioural properties. Recall that the compilation acts also on types, and that for translating the arrow types of HO$\pi$ we used the $\mathtt{i}/\mathtt{o}$ types of the $\pi$-calculus.

## 5.2 Composing the steps

Composing the three steps, from $\lambda V$ to CPS$_{\mathtt{V}}$, from CPS$_{\mathtt{V}}$ to HO$\pi$, and from HO$\pi$ to $\pi$-calculus, we obtain the encoding $\mathcal{V}$ of $\lambda V$ into $\pi$-calculus in Table 3. Precisely, the encoding $\mathcal{V}[\![M]\!]$ of a $\lambda$-term $M$ is obtained thus (we can omit the type environment index in $\mathcal{C}$, for $M$ is untyped and therefore all free variables are translated in the same way):

$$\mathcal{V}[\![M]\!] \stackrel{\mathrm{def}}{=} \mathcal{C}[\![\ \mathcal{H}[\![\ \mathcal{C}_{\mathtt{V}}[\![M]\!]\ ]\!]\ ]\!]$$

The encoding $\mathcal{V}$ uses two kinds of name:

- *Location names* $(p, q, r)$ that are used as arguments of the encodings of $\lambda$-terms. These names correspond to the continuation variable of the CPS Grammar (15).
- *Value names* $(x, y)$ that are used to access values. These names correspond to the ordinary variables of the CPS Grammar (15).

(As $\mathcal{V}[\![M]\!]$ is an abstraction, for all $M$, we remind these notations abstractions: suppose $\mathcal{V}[\![M]\!]$ is $(q).\,Q$; then $y(x)\,\mathcal{V}[\![M]\!]$ is $y(x,q).\,Q$, and $\mathcal{V}[\![M]\!]r$ is $Q\{r/q\}$. These and other notations for abstractions are discussed in Section B.1.7.)

We explain informally how the encoding works. Suppose a $\lambda$-term $M$ reduces to a value $V$. This value represents the result of the $\lambda$-term. In the $\pi$-calculus encoding, $\mathcal{V}[\![M]\!]p$ reduces to a process that, very roughly, returns the value $V$ at $p$ (this is the analogue of passing the result to the continuation in the CPS transform). The value $V$ can be a variable or an abstraction. If $V$ is a variable then the corresponding $\pi$-calculus name is returned at $p$. If $V$ is a function then it cannot be passed directly at $p$ because $\pi$-calculus does not allow communication of terms; instead, a *pointer* to the function is passed. Thus the function sits within the process as a resource that can be accessed arbitrarily many times, via the pointer: when a client sends a value $v$ and a return name $q$, the function will answer by sending a result at $q$ (if a result exists).

The encoding of an application $MN$ at location $p$ is a process that first runs $M$ at some location $q$. When $M$ signals that it has become a value $v$, the argument $N$ is run at some location $r$; names $q$ and $r$ are private, to avoid interference from the environment. When also $N$ signals that it has become a value $w$, the application occurs: the pair $\langle w, p\rangle$ is sent at $v$. This communication is the step that properly simulates the $\beta_{\mathtt{v}}$-reduction of $\lambda V$; the previous communications were "administrative" (exactly as it happens in the CPS transform, see Section 5). The second argument of the pair $\langle w, p\rangle$ is the location where the final result of $MN$ will be delivered.

**Remark 5.13** In Table 3, the inputs at location names ($q$ and $r$) are not replicated because, in the step of translation from HO$\pi$ to $\pi$-calculus, we take into account the linearity constraint on continuations (see Remark 5.7) and therefore adopt the optimisation of Section D.0.9. As discussed in Section D.0.9, linear types are however not needed in the $\pi$-calculus. ■

For the sake of readability the translation of Table 3 is not annotated with types. Types are needed, however, to prove its correctness. The $\pi$-calculus translation of type $\mathcal{H}[\![T_V]\!]$ in (18) is the recursive type

$$\mathrm{Val} \stackrel{\mathrm{def}}{=} \mu X.\,(\mathsf{o}\,\langle X, \mathsf{o}\,X\rangle)\ .$$

We do not need the $\mathsf{i}$ tag on types, because both location and trigger names that are communicated may only be used by a recipient for sending. Here is the encoding in which names bound by a restriction are annotated with their type (we recall that if $T$ is a $\pi$-calculus type, then $(T)^{\leftarrow\sharp}$ is the type obtained by replacing the outermost $\mathsf{i}/\mathsf{o}$ tag in $T$ with $\sharp$, possibly after unfolding $T$ if its outermost construct is recursion):

$$\begin{array}{rcl} \mathcal{V}[\![\lambda x.\, M]\!] & \stackrel{\mathrm{def}}{=} & (p).\, \overline{p}(y \,:\, (\mathrm{Val})^{\leftarrow\sharp}).\, !y(x)\, \mathcal{V}[\![M]\!] \\ \mathcal{V}[\![x]\!] & \stackrel{\mathrm{def}}{=} & (p).\, \overline{p}\langle x\rangle \\ \mathcal{V}[\![MN]\!] & \stackrel{\mathrm{def}}{=} & (p).\ \ (\boldsymbol{\nu} q : \sharp\,(\mathrm{Val})\,) \\ & & \qquad \Big(\mathcal{V}[\![M]\!]q \mid q(x).\,(\boldsymbol{\nu} r : \sharp\,(\mathrm{Val})\,)\,(\mathcal{V}[\![N]\!]r \mid r(y).\,\overline{x}\langle y, p\rangle)\Big) \end{array}$$

Types Val and $\mathsf{o}\,(\mathrm{Val})$ are, respectively, the types of free trigger and free location names of process $\mathcal{V}[\![M]\!]p$. These types change the outermost tag to $\sharp$ on names local to $\mathcal{V}[\![M]\!]p$, thus becoming, respectively, $(\mathrm{Val})^{\leftarrow\sharp}$ and $\sharp\,(\mathrm{Val})$.

The translation of (19) into $\pi$-calculus gives

**Lemma 5.14** *Suppose* $\mathrm{fv}(M) \subseteq \widetilde{x}$. *Then* $\widetilde{x} \,:\, \mathrm{Val}, p \,:\, \mathsf{o}\,(\mathrm{Val}) \vdash \mathcal{V}[\![M]\!]p$ .

From the correctness of the 3 steps from which encoding $\mathcal{V}$ has been derived, we get the two corollaries below. Thereafter, we shall also derive direct proofs of these corollaries, which do not go through the CPS transforms but, instead, appeal to the theory of the $\pi$-calculus (Exercise 5.21).

**Corollary 5.15 (adequacy of $\mathcal{V}$)** *Let* $M \in \Lambda^0$. *It holds that* $M \Downarrow_{\mathsf{v}}$ *iff* $\mathcal{V}[\![M]\!]p \Downarrow$, *for any* $p$.

**Proof:** From Theorem 5.4, Lemma 5.11, and the operational correctness of the encoding of HO$\pi$ into the $\pi$-calculus (Lemma D.2 and its extensions, Section D.0.9). ∎

**Corollary 5.16 (validity of $\lambda\beta_{\mathsf{v}}$ theory for $\mathcal{V}$)** *Suppose* $\mathrm{fv}(M, N) \subseteq \widetilde{x}$, *and let* $H \stackrel{def}{=} \widetilde{x} \,:\, \mathrm{Val}\,;\, \mathsf{o}\,(\mathrm{Val})$ . *If* $\lambda\beta_{\mathsf{v}} \vdash M = N$ *then* $\mathcal{V}[\![M]\!] \approxeq^{\mathsf{c}}_{H} \mathcal{V}[\![N]\!]$.

**Proof:** From Theorem 5.5, Corollary 5.12, and (the appropriate extension of) Lemma D.3. ∎

**Remark 5.17 (effect of $\mathsf{i}/\mathsf{o}$ types on behavioural equivalences)** In the relation of barbed congruence of Corollary 5.16, the presence of $\mathsf{i}/\mathsf{o}$ types is important. The result would not hold if each $\mathsf{i}/\mathsf{o}$ type $I\,T$ (for $I \in \{\mathsf{i}, \mathsf{o}, \sharp\}$) were replaced by the (less informative) channel type $\sharp\,T$. As a counterexample, take $M \stackrel{\mathrm{def}}{=} (\lambda x.\, (\lambda y.\, x))\lambda z.\, z$ and $N \stackrel{\mathrm{def}}{=} \lambda y.\, (\lambda z.\, z)$. With a $\beta_{\mathsf{v}}$-conversion, $M$ reduces to $N$. However, without $\mathsf{i}/\mathsf{o}$ types $\mathcal{V}[\![M]\!]p$ and $\mathcal{V}[\![N]\!]p$ could be distinguished; see [San92, page 128]. ∎

By Corollary 5.16, we know that if $M \longrightarrow_{\mathsf{v}} M'$ then $\mathcal{V}[\![M]\!]$ and $\mathcal{V}[\![M']\!]$ are behaviourally indistinguishable. One might like, however, to see how the reduction of $\lambda V$ is simulated in the $\pi$-calculus. This is shown below. Recall that the local-environment notation $P\{x = (\widetilde{z}).\, R\}$ stands for $\boldsymbol{\nu} x\,(P \mid !x(\widetilde{z}).\, R)$, under the hypothesis that $P$ and $R$ only possess the output capability on $x$ (Section B.0.9).

**Notation**. We write $P \longrightarrow_{\mathsf{d}} P'$ if the reduction $P \longrightarrow P'$ is *deterministic* (i.e. $P \longrightarrow P''$ implies $P' = P''$, and there is no $p$ such that $P \downarrow_p$). If $P \longrightarrow_{\mathsf{d}} P'$ then in any context, $P \longrightarrow P'$ is necessarily the first action that $P$ can participate in. We write $P \longrightarrow^n_{\mathsf{d}} P'$ if $P$ evolves to $P'$ by performing $n$ deterministic reductions.

**Lemma 5.18** $\mathcal{V}[\![(\lambda x.\, M)\lambda y.\, N]\!]p \longrightarrow_{\mathtt{d}}^{3} \sim \mathcal{V}[\![M]\!]p\{x = (y)\,\mathcal{V}[\![N]\!]\}$.

**Proof:** We apply the laws of Lemma B.9 and the laws 5-7 of Theorem B.10:

$$\begin{array}{rl}
& \mathcal{V}[\![(\lambda x.\, M)\lambda y.\, N]\!]p \\
= & \boldsymbol{\nu} q\left(\overline{q}(z).\, !z(x)\,\mathcal{V}[\![M]\!] \mid q(z).\, \boldsymbol{\nu} r\,(\overline{r}(u).\, !u(y)\,\mathcal{V}[\![N]\!] \mid r(u).\, \overline{z}\langle u, p\rangle)\right) \\
\longrightarrow_{\mathtt{d}} \sim & (\boldsymbol{\nu} r, z\,)\left(!z(x)\,\mathcal{V}[\![M]\!] \mid \overline{r}(u).\, !u(y)\,\mathcal{V}[\![N]\!] \mid r(u).\, \overline{z}\langle u, p\rangle\right) \\
\longrightarrow_{\mathtt{d}} \sim & (\boldsymbol{\nu} u, z\,)\left(!z(x)\,\mathcal{V}[\![M]\!] \mid !u(y)\,\mathcal{V}[\![N]\!] \mid \overline{z}\langle u, p\rangle\right) \\
\longrightarrow_{\mathtt{d}} \sim & (\boldsymbol{\nu} u, z\,)\left(\mathcal{V}[\![M]\!]p\{{}^{u}\!/\!{}_{x}\} \mid !z(x)\,\mathcal{V}[\![M]\!] \mid !u(y)\,\mathcal{V}[\![N]\!]\right) \\
\sim & (\boldsymbol{\nu} x\,)\left(\mathcal{V}[\![M]\!]p \mid !x(y)\,\mathcal{V}[\![N]\!]\right) \\
= & \mathcal{V}[\![M]\!]p\{x = (y)\,\mathcal{V}[\![N]\!]\}
\end{array}$$

This proves the lemma, since $\sim$ commutes with $\longrightarrow$. ■

The reader might like to compare the derivation in the proof of the lemma with that in (13), to see the similarities.

**Lemma 5.19** *Suppose* $\mathrm{fv}(M, N) \subseteq \widetilde{x}$*, and let* $H \stackrel{def}{=} \widetilde{x} : \mathrm{Val}\,;\, \mathtt{o}\,(\mathrm{Val})$ *. We have* $\mathcal{V}[\![M]\!]\{x = (y)\,\mathcal{V}[\![N]\!]\} \simeq^{\mathtt{c}}_{H} \mathcal{V}[\![M\{{}^{N}\!/\!{}_{x}\}]\!]$.

**Proof:** We proceed by structural induction on $M$. There are three cases: when $M$ is a variable, an abstraction or an application.

**Variable:** $M = z$.

Both the case $z = x$ and the case $z \neq x$ are easy.

**Abstraction:** $M = \lambda z.\, M'$. We have

$$\begin{array}{rll}
\mathcal{V}[\![M]\!]\{x = (y)\,\mathcal{V}[\![N]\!]\} & = & \\
\left(\overline{p}(y).\, !y(z)\,\mathcal{V}[\![M']\!]\right)\{x = (y)\,\mathcal{V}[\![N]\!]\} & \sim & (20) \\
\overline{p}(y).\, \left((!y(z)\,\mathcal{V}[\![M']\!])\{x = (y)\,\mathcal{V}[\![N]\!]\}\right) & \simeq^{\mathtt{c}}_{H} & (21) \\
\overline{p}(y).\, !y(z)\,(\mathcal{V}[\![M']\!]\{x = (y)\,\mathcal{V}[\![N]\!]\}) & \simeq^{\mathtt{c}}_{H} & (22) \\
\overline{p}(y).\, !y(z)\,\mathcal{V}[\![M'\{{}^{N}\!/\!{}_{x}\}]\!] & = & \\
\mathcal{V}[\![M\{{}^{N}\!/\!{}_{x}\}]\!] & &
\end{array}$$

where, in (20), the scope of the local environment $\{x = (y)\,\mathcal{V}[\![N]\!]\}$ has been reduced, since $x$ does not occur in $\overline{p}(y)$; the equality (21) pushes the local environment inside prefixes and replication, and is derived from the strong replicator theorem (Section B.1.6); finally the equality (22) follows from the inductive assumption.

**Application:** $M = M_1 M_2$.

This case is similar to that of abstraction; we leave it as an exercise.

■

**Corollary 5.20** *Let* $M \in \Lambda^0$.

1. *If* $M \longrightarrow_{\mathtt{V}} M'$ *then* $\mathcal{V}[\![M]\!]p \longrightarrow_{\mathtt{d}}^{3} P \simeq^{\mathtt{c}}_{p\,:\,\mathtt{o}\,(\mathrm{Val})} \mathcal{V}[\![M']\!]p$.
2. *The converse, i.e., if* $\mathcal{V}[\![M]\!]p \longrightarrow P$ *then there is* $M'$ *such that* $M \longrightarrow_{\mathtt{V}} M'$, $P \longrightarrow_{\mathtt{d}}^{2} \simeq^{\mathtt{c}}_{p\,:\,\mathtt{o}\,(\mathrm{Val})} \mathcal{V}[\![M']\!]p$, *and moreover the reduction* $\mathcal{V}[\![M]\!]p \longrightarrow P$ *is deterministic.*

**Proof:** An easy exercise, using the previous two lemmas. ■

Corollary 5.20 shows that, in response to $M \longrightarrow_{\mathtt{V}} M'$, process $\mathcal{V}[\![M]\!]p$ deterministically performs three reductions, whose derivative $P$ is not precisely the encoding of $M'$, but can be proved equal to it using some laws that "reorder" the structure of $P$ (relation $\simeq^{\mathtt{c}}_{p\,:\,\mathtt{o}\,(\mathrm{Val})}$).

**Exercise 5.21** *Use Corollary 5.20 to prove Corollaries 5.16 and 5.15.*

**Remark 5.22** The results on the CPS transform, notably Theorems 5.4 and 5.5, have been used to derive results about the correctness of the $\pi$-calculus encoding. Sometimes, we may also go in the opposite direction, that is use the $\pi$-calculus encoding to understand, and reason about, the CPS transform. For instance, in Exercise 5.21 we have derived a direct proof of Corollary 5.15. Using this, Lemma 5.11, and Lemma D.2, we can derive the adequacy of the CPS transform (Theorem 5.4). Similarly, in the same Exercise 5.21 we have derived Corollary 5.16; using this (and Lemma D.4), we can prove a weaker version of Theorem 5.5, saying that if $\lambda\beta_{\mathtt{v}} \vdash M = N$ then $\mathcal{C}_{\mathtt{V}}[\![M]\!]$ and $\mathcal{C}_{\mathtt{V}}[\![N]\!]$ are observationally equivalent as terms of $\lambda N$. (Observational equivalence equates terms that are behaviourally indistinguishable and will be defined and studied in Section 11; informally, two $\lambda N$-terms $L_1$ and $L_2$ are observationally indistinguishable if for for all $\lambda$-calculus contexts $C$ it holds that $C[L_1] \Downarrow_{\mathtt{N}}$ iff $C[L_2] \Downarrow_{\mathtt{N}}$.)

## 5.3 Parallel call-by-value

From the call-by-value encoding it is easy to obtain variants with different disciplines for reducing the operator and the operand of an application. One such variant is the *parallel call-by-value* strategy, that has the ordinary $\nu$ rule in place of $\nu_{\mathtt{v}}$. Here is the definition of application for the parallel call-by-value encoding, denoted by $\mathcal{PV}$:

$$\mathcal{PV}[\![MN]\!] \stackrel{\mathrm{def}}{=} (p).\,(\boldsymbol{\nu} q, r\,)\Big(\mathcal{PV}[\![M]\!]q \mid \mathcal{PV}[\![N]\!]r \mid q(x).\,r(y).\,\overline{x}\langle y, p\rangle\Big)$$

The operator and the operand are run in parallel. The clauses for abstractions and values for $\mathcal{PV}$ are the same as for (the sequential) $\mathcal{V}$.

**Exercise 5.23** *Exhibit a term* $M$ *on which the encodings of sequential and parallel call-by-value do not give the same behaviours, i.e.,* $\mathcal{V}[\![M]\!] \not\approx \mathcal{PV}[\![M]\!]$.

**Exercise 5.24**
1. *Suppose* $\mathrm{fv}(M,N) \subseteq \widetilde{x}$, *and let* $H \stackrel{def}{=} \widetilde{x} : \mathrm{Val}\,;\mathtt{o}\,(\mathrm{Val})$. *Prove that* $\mathcal{PV}[\![(\lambda x.\,M)\lambda y.\,N]\!] \cong^{\mathtt{c}}_{H} \mathcal{PV}[\![M\{^{\lambda y.\,N}\!/x\}]\!]$. *(Hint: it is similar to the proofs of Lemmas 5.18 and 5.19.)*
2. *Conclude that if* $\lambda\beta_{\mathtt{v}} \vdash M = N$, *then* $\mathcal{V}[\![M]\!] \cong^{\mathtt{c}}_{H} \mathcal{V}[\![N]\!]p$.

*In this table we abbreviate* $\mathcal{C}_{\mathtt{N}}[\![M]\!]$ *as* $[\![M]\!]$ *and* $\mathcal{C}^{\star}_{\mathtt{N}}[V]$ *as* $[V]$*:*

$$\text{call-by-name values} \quad V \quad := \quad \lambda x.\, M$$

$$\begin{array}{rcl} [\![x]\!] & \stackrel{\text{def}}{=} & \lambda k.\, xk \\ [\![V]\!] & \stackrel{\text{def}}{=} & \lambda k.\, k[V] \\ [\![MN]\!] & \stackrel{\text{def}}{=} & \lambda k.\, [\![M]\!](\lambda v.\, v[\![N]\!]k) \\ \\ [\lambda x.\, M] & \stackrel{\text{def}}{=} & \lambda x.\, [\![M]\!] \end{array}$$

Table 4: The call-by-name CPS transform

## 6 The interpretation of call-by-name

In this section we develop a $\pi$-calculus encoding of call-by-name $\lambda$-calculus. The approach is similar to that for call-by-value in Section 5. The reader not interested in the CPS transform may see directly the encoding into $\pi$-calculus of Section 6.0.1.

#### Step 1: the call-by-name CPS

Table 4 shows the call-by-name CPS transform of [Plo75] (in fact, a rectified variant of it, see the notes in Section 14). Here is how a $\beta$ reduction

$$(\lambda x.\, M)N \longrightarrow_{\mathtt{N}} M\{^{N}\!/\!_{x}\}$$

is simulated in the transform. (As in call-by-value, we choose to take call-by-name for the reduction relation on the images of the CPS, but these terms are evaluation-order independent: see Theorem 6.1.)

$$\begin{array}{cl} & \mathcal{C}_{\mathtt{N}}[\![(\lambda x.\, M)N]\!]k \\ \longrightarrow_{\mathtt{N}} & \mathcal{C}_{\mathtt{N}}[\![\lambda x.\, M]\!](\lambda v.\, v\mathcal{C}_{\mathtt{N}}[\![N]\!]k) \\ = & (\lambda k.\, (k\mathcal{C}^{\star}_{\mathtt{N}}[\lambda x.\, M]))(\lambda v.\, v\mathcal{C}_{\mathtt{N}}[\![N]\!]k) \\ \longrightarrow_{\mathtt{N}} & (\lambda v.\, v\mathcal{C}_{\mathtt{N}}[\![N]\!]k)\mathcal{C}^{\star}_{\mathtt{N}}[\lambda x.\, M] \\ \longrightarrow_{\mathtt{N}} & \mathcal{C}^{\star}_{\mathtt{N}}[\lambda x.\, M]\mathcal{C}_{\mathtt{N}}[\![N]\!]k \\ = & (\lambda x.\, \mathcal{C}_{\mathtt{N}}[\![M]\!])\mathcal{C}_{\mathtt{N}}[\![N]\!]k \\ \longrightarrow_{\mathtt{N}} & \mathcal{C}_{\mathtt{N}}[\![M]\!]\{^{\mathcal{C}_{\mathtt{N}}[\![N]\!]}\!/\!_{x}\}k \end{array}$$

In general $\mathcal{C}_{\mathtt{N}}[\![M]\!]\{^{\mathcal{C}_{\mathtt{N}}[\![N]\!]}\!/\!_{x}\}k$ is not equal to $\mathcal{C}_{\mathtt{N}}[\![M\{^{N}\!/\!_{x}\}]\!]k$, because $\mathcal{C}_{\mathtt{N}}$ does not commute with substitution. The two terms are, however, $\beta$-convertible; indeed $\mathcal{C}_{\mathtt{N}}[\![M]\!]\{^{\mathcal{C}_{\mathtt{N}}[\![N]\!]}\!/\!_{x}\}k \Longrightarrow_{\beta} \mathcal{C}_{\mathtt{N}}[\![M\{^{N}\!/\!_{x}\}]\!]k$.

Closing the image of the transform under $\beta$-reduction gives the language $\mathrm{CPS}_{\mathtt{N}}$ of the call-by-name CPS:

$$\mathrm{CPS}_{\mathtt{N}} \stackrel{\text{def}}{=} \{A \;:\; \exists M \in \Lambda \text{ with } \mathcal{C}_{\mathtt{N}}[\![M]\!] \Longrightarrow_{\beta} A\}$$

When there is no ambiguity we call the terms of $\mathrm{CPS}_{\mathtt{N}}$ the *CPS terms*.

The theorems below are the call-by-name versions of Theorems 5.1-5.5. They show the indifference of the CPS terms to the choice between call-by-name and call-by-value, and the correctness of the CPS transform. Theorem 6.3 is slightly weaker than the corresponding result for call-by-value. The reason is that, as illustrated above, $\mathcal{C}_{\mathtt{N}}$ commutes with substitution only up to $\beta$-conversion. In contrast, Theorem 6.4 is stronger than the corresponding result for call-by-value: it asserts a logical equivalence rather than an implication.

**Theorem 6.1 (indifference of $\mathrm{CPS}_{\mathtt{N}}$ on reductions)** *Let $M \in \mathrm{CPS}_{\mathtt{N}}$ and let $N$ be any subterm of $M$. For all $N'$, we have: $N \longrightarrow_{\mathtt{N}} N'$ iff $N \longrightarrow_{\mathtt{V}} N'$.*

**Theorem 6.2 (indifference of $\mathrm{CPS}_{\mathtt{N}}$ on $\lambda$-theories)** *For all $M, N \in \mathrm{CPS}_{\mathtt{N}}$, we have: $\lambda\beta \vdash M = N$ iff $\lambda\beta_{\mathtt{v}} \vdash M = N$.*

**Theorem 6.3 (adequacy of $\mathcal{C}_{\mathtt{N}}$)** *Let $M \in \Lambda^0$.*

1. *If $M \Longrightarrow_{\mathtt{N}} V$ where $V$ is a call-by-name value, then there is an $\mathtt{N}$-nf $N$ such that $\mathcal{C}_{\mathtt{N}}[\![M]\!]k \Longrightarrow_{\mathtt{N}} N$ and $\lambda\beta \vdash N = k\,\mathcal{C}^{\star}_{\mathtt{N}}[V]$.*
2. *The converse, i.e., if $\mathcal{C}_{\mathtt{N}}[\![M]\!]k \Longrightarrow_{\mathtt{N}} N$ and $N$ is an $\mathtt{N}$-nf, then there is a call-by-name value $V$ such that $M \Longrightarrow_{\mathtt{N}} V$ and $\lambda\beta \vdash N = k\,\mathcal{C}^{\star}_{\mathtt{N}}[V]$.*

**Theorem 6.4 (validity of the $\beta$-theory for $\mathcal{C}_{\mathtt{N}}$)** *Let $M, N \in \Lambda$. Then $\lambda\beta \vdash M = N$ iff $\lambda\beta \vdash \mathcal{C}_{\mathtt{N}}[\![M]\!] = \mathcal{C}_{\mathtt{N}}[\![N]\!]$.*

### Step 2: from $\mathrm{CPS}_{\mathtt{N}}$ to HO$\pi$

The grammar below generates all terms of $\mathrm{CPS}_{\mathtt{N}}$. The intuitive meaning of the various syntactic categories in the grammar is the same as for the call-by-value Grammar (15). The main differences are the addition of the value variable $v$, representing the parameter of continuations, and the splitting of the set of answers into the sets $P_1$ and $P_2$. These modifications are made in order to capture the linear use of the parameters of continuations within the grammar (dropping linearity we would have the CPS language of Table 6). As in the call-by-value grammar, there is only one continuation variable $k$ because continuations are used linearly.

$$
\begin{array}{rrcl}
\text{continuation variable} & k & & \\
\text{ordinary variables} & x, \ldots & & \\
\text{value variable} & v & & \\
\text{answers} & P & := & P_1 \mid P_2 \\
 & P_1 & := & KV \mid VAK \mid AK \\
 & P_2 & := & vAK \\
\text{CPS-values} & V & := & \lambda x.\, \lambda k.\, P_1 \\
\text{continuations} & K & := & k \mid \lambda v.\, P_2 \\
\text{principal terms} & A & := & \lambda k.\, P_1 \mid x
\end{array} \tag{23}
$$

Using $\diamond$ as the type of answers, the types $T_V$, $T_K$ and $T_A$ of CPS-values, continuations, and principal terms are

$$
\begin{array}{rcl}
T_V & \stackrel{\text{def}}{=} & \mu X.\, \Big( ((X \to \diamond) \to \diamond) \to ((X \to \diamond) \to \diamond) \Big) \\
T_K & \stackrel{\text{def}}{=} & T_V \to \diamond \\
T_A & \stackrel{\text{def}}{=} & T_K \to \diamond
\end{array} \tag{24}
$$

The value variable $v$ has the type $T_V$ of the CPS values. The typing judgments for the terms generated by the grammar are as expected, given these types.

**Proposition 6.5** *If $M \in \mathrm{CPS}_{\mathtt{N}}$, then $M$ is also a principal term of Grammar (23).*

**Proof:** One can show that the set of principal terms includes the set $\{\mathcal{C}_{\mathtt{N}}[\![M]\!] : M \in \Lambda\}$ and is closed under $\beta$-conversion. ∎

The injection $\mathcal{H}$, from the terms generated by Grammar (23) to HO$\pi$, is defined as for the call-by-value Grammar (15), and similar results hold:

**Lemma 6.6** *Suppose $M$ is a term generated by Grammar (23), and $\Gamma \vdash M : T$. Then also $\mathcal{H}[\![\Gamma]\!] \vdash \mathcal{H}[\![M]\!] : \mathcal{H}[\![T]\!]$.*

Therefore, if $M \in \Lambda$ with $\mathrm{fv}(M) \subseteq \widetilde{x}$, then

$$\widetilde{x} : \mathcal{H}[\![T_A]\!] \vdash \mathcal{H}[\![\mathcal{C}_{\mathtt{N}}[\![M]\!]]\!] : \mathcal{H}[\![T_A]\!] = (\mathcal{H}[\![T_V]\!] \to \diamond) \to \diamond \tag{25}$$

**Lemma 6.7** *For all terms $M$ of Grammar (23), we have: if $M \longrightarrow_{\mathtt{N}} M'$ then $\mathcal{H}[\![M]\!] \longrightarrow_{\beta} =_{\beta} \mathcal{H}[\![M']\!]$; and conversely, if $\mathcal{H}[\![M]\!] \longrightarrow_{\beta} P$ then there is $M'$ such that $M \longrightarrow_{\mathtt{N}} M'$ and $P =_{\beta} \mathcal{H}[\![M']\!]$.*

**Corollary 6.8** *For all terms $M, N$ generated by Grammar (15), $\lambda\beta \vdash M = N$ implies $\mathcal{H}[\![M]\!] =_{\beta} \mathcal{H}[\![N]\!]$.*

**Step 3: compilation $\mathcal{C}$**

This step is given by compilation $\mathcal{C}$ of HO$\pi$ into $\pi$-calculus.

### 6.0.1 Composing the steps

Composing the three steps, from $\lambda N$ to $\mathrm{CPS}_{\mathtt{N}}$, from $\mathrm{CPS}_{\mathtt{N}}$ to HO$\pi$, and from HO$\pi$ to $\pi$-calculus, we obtain the encoding of $\lambda N$ into $\pi$-calculus in Table 5.

The encoding uses three kinds of name: *location names* $(p, q, r)$, *trigger names* $(x, y)$, and *value names* $(v)$. Location names are arguments of the encoding, and are the counterpart of the continuation variable of the CPS Grammar (23). Trigger names are pointers to $\lambda$-terms, and are the counterpart of the ordinary variables of the CPS grammar. Value names are pointers to values (more precisely, to CPS-values, in the terminology of Grammar (23)), and are the counterpart of the value variable of the CPS grammar.

We explain briefly how the encoding works. As in the call-by-value encoding, a $\lambda$-term $M$ that reduces to a value $V$ is translated as a process $\mathcal{N}[\![M]\!]p$ that, roughly speaking, emits $V$ at $p$. But, in contrast to call-by-value, in call-by-name a value can only be a function. Moreover, a $\lambda$-term that is evaluated may only be the operand — not the argument — of an application. As such, it may not be copied; that is, it may be used only *once*. In the encoding of abstraction in Table 5, this linearity is evident in the fact that the input at $v$ is not replicated. Another difference from call-by-value is that in call-by-name the argument passed to a function may be an arbitrary $\lambda$-term, which, in the body of the function, has to be evaluated every time its value is needed. This difference is evident in the encoding of a variable $x$: the corresponding $\pi$-calculus name

$$
\begin{array}{rcl}
\mathcal{N}[\![\lambda x.\,M]\!] & \stackrel{\text{def}}{=} & (p).\,\overline{p}(v).\,v(x)\,\mathcal{N}[\![M]\!] \\
\mathcal{N}[\![x]\!] & \stackrel{\text{def}}{=} & (p).\,\overline{x}\langle p\rangle \\
\mathcal{N}[\![MN]\!] & \stackrel{\text{def}}{=} & (p).\,\boldsymbol{\nu} q\,\Big(\mathcal{N}[\![M]\!]q \mid\ q(v).\,\boldsymbol{\nu} x\,(\overline{v}\langle x,p\rangle.\,!x\,\mathcal{N}[\![N]\!])\Big)
\end{array}
$$

Table 5: The encoding of $\lambda N$ into the $\pi$-calculus

$x$ is used not as a value but as a trigger for activating a term and providing it with a location. In the encoding of an application $MN$ at location $p$, when $M$ signals that it has become a function, it receives a trigger for the argument $N$ and the location $p$ for interacting with the environment.

**Remark 6.9** In the table, in the definitions of function and application, the inputs at $v$ and $q$ are not replicated because of the linearity constraint on value variables and on continuation variable of Grammar (23) that, in the compilation of HO$\pi$ to the $\pi$-calculus, enables us to adopt the optimisation of Section D.0.9; we did similarly in the call-by-value encoding, see Remark 5.13. ■

**Remark 6.10** The linearity of the value names of Table 5 (and similarly, of the value variables of Grammar (23)) may be lost whenever one adds further constructs to the $\lambda$-calculus, as we will do in Section 11. To support extensions, the definition of abstraction in the table requires a replication in front of the input at $v$ — as in the call-by-value encoding. We derive such an encoding in the next section (we also use it in Exercise 6.17). When possible, however, it is good to avoid the replication, to keep the encoding simpler and therefore make it easier to prove properties of the encoding (which we shall do in this section and in Sections 11-13). ■

In the encoding of Table 5, we have omitted type annotations. Below is the complete encoding including types. The type

$$
\mathrm{Trig} \stackrel{\text{def}}{=} \mu X.\,(\mathsf{o}\,\langle \mathsf{o}\,\mathsf{o}\,X, \mathsf{o}\,X\rangle) \tag{26}
$$

is the translation into $\pi$-calculus of the type $T_V$ in (24).

$$
\begin{array}{rcl}
\mathcal{N}[\![\lambda x.\,M]\!] & \stackrel{\text{def}}{=} & (p).\,\overline{p}(v : (\mathrm{Trig})^{\leftarrow\sharp}).\,v(x)\,\mathcal{N}[\![M]\!] \\
\mathcal{N}[\![x]\!] & \stackrel{\text{def}}{=} & (p).\,\overline{x}\langle p\rangle \\
\mathcal{N}[\![MN]\!] & \stackrel{\text{def}}{=} & (p).\,(\boldsymbol{\nu} q : \sharp\,(\mathrm{Trig})\,) \\
 & & \Big(\mathcal{N}[\![M]\!]q \mid q(v).\,(\boldsymbol{\nu} x : \sharp\,\mathsf{o}\,(\mathrm{Trig})\,)\,(\overline{v}\langle x,p\rangle.\,!x\,\mathcal{N}[\![N]\!])\Big)
\end{array} \tag{27}
$$

The translation of (25) into $\pi$-calculus gives

**Lemma 6.11** *Suppose* $\mathrm{fv}(M) \subseteq \widetilde{x}$*. Then* $\widetilde{x} : \mathsf{o}\,\mathsf{o}\,(\mathrm{Trig})\,, p : \mathsf{o}\,(\mathrm{Trig}) \vdash \mathcal{N}[\![M]\!]p$.

We also derive the following two results about the operational correctness of the encoding:

**Corollary 6.12 (adequacy of $\mathcal{N}$)** *Let $M \in \Lambda^0$. Then $M \Downarrow_{\mathtt{N}}$ iff $\mathcal{N}[\![M]\!]p \Downarrow$, for any $p$.*

**Proof:** From Theorem 6.3, Corollary 6.7, and (the appropriate extension of) Lemma D.2. ■

**Corollary 6.13 (validity of $\lambda\beta$ theory for $\mathcal{N}$)** *Suppose $\mathrm{fv}(M, N) \subseteq \widetilde{x}$ and let $H \stackrel{def}{=} \widetilde{x} : \mathsf{o}\,\mathsf{o}\,(\mathrm{Trig})\ ;\ \mathsf{o}\,(\mathrm{Trig})$. If $\lambda\beta \vdash M = N$ then $\mathcal{N}[\![M]\!] \approx^{\mathsf{c}}_{H} \mathcal{N}[\![N]\!]$.*

**Proof:** From Theorem 6.4, Lemma 5.12, Lemma B.5, and (the appropriate extension of) Lemma D.3. ■

Encoding $\mathcal{N}$ validates rule $\beta$ of the $\lambda$-calculus. By contrast, as Exercise 6.15 shows, $\mathcal{N}$ does not validate rule $\eta$, namely

$$\lambda x.\,(Mx) = M \qquad\qquad \text{if } x \notin \mathrm{fv}(M)$$

Neither does the call-by-value encoding $\mathcal{V}$ satisfy $\eta$. These failures make sense, as $\eta$ is not operationally valid either in call-by-name or in call-by-value $\lambda$-calculus (see Exercise 11.4). The $\eta$ rule is operationally valid, however, if $M$ is a value; indeed the encoding does validate this restricted form of $\eta$ (Exercise 6.16).

**Exercise 6.14** *Show that $\mathcal{N}[\![\Omega]\!]p \approx \mathbf{0}$*

**Exercise 6.15 (non-validity of the $\eta$ rule)** *Show that there is a $\lambda$-term $M$ with $x \notin \mathrm{fv}(M)$ such that if $\mathrm{fv}(M) = \{\widetilde{x}\}$ and $H \stackrel{def}{=} \widetilde{x} : \mathsf{o}\,\mathsf{o}\,(\mathrm{Trig})\,;\mathsf{o}\,(\mathrm{Trig})$, then $\mathcal{N}[\![\lambda x.\,(Mx)]\!] \not\approx^{\mathsf{c}}_{H} \mathcal{N}[\![M]\!]$. (Hint: use Exercise 6.14.)*

**Exercise 6.16 (validity of the conditional $\eta$ rule)** *Let $V \in \Lambda$ be a call-by-name value. Show that if $x \notin \mathrm{fv}(V)$ then $\mathcal{N}[\![\lambda x.\,(Vx)]\!] \approx \mathcal{N}[\![V]\!]$.*

**Exercise 6.17 (from call-by-value to call-by-name via thunks)** *The CPS transform $\mathcal{C}_{\mathtt{N}}$ gives us an encoding of call-by-name ($\lambda N$) into call-by-value ($\lambda V$) $\lambda$-calculus. Another way to achieve this goal is by means of* thunks*. A thunk is a parameterless procedure; it may also be thought of as a suspended computation. To represent thunks, we can introduce a constructor $\mathtt{delay}(M)$ and a destructor $\mathtt{force}(M)$. Keeping in mind that they represent parameterless abstraction and application, it is straightforward to add them to $\lambda V$ and to its $\pi$-calculus encoding $\mathcal{V}$ (Table 3); the additional operational rules are:*

$$\frac{M \longrightarrow_{\mathtt{V}} M'}{\mathtt{force}(M) \longrightarrow_{\mathtt{V}} \mathtt{force}(M')} \qquad\qquad \frac{}{\mathtt{force}(\mathtt{delay}(M)) \longrightarrow_{\mathtt{V}} M}$$

*and the additional clauses for $\mathcal{V}$ are:*

$$\begin{array}{rcl} \mathcal{V}[\![\mathtt{delay}(M)]\!] & \stackrel{def}{=} & (p).\,\overline{p}(v).\,!v\,\mathcal{V}[\![M]\!] \\ \mathcal{V}[\![\mathtt{force}(M)]\!] & \stackrel{def}{=} & (p).\,\boldsymbol{\nu} q\,(\mathcal{V}[\![M]\!]q \mid q(v).\,\overline{v}\langle p\rangle) \end{array} \qquad (28)$$

*We can now encode $\lambda N$ into this extended $\lambda V$ thus:*

$$\begin{array}{rcl} \mathcal{D}[\![x]\!] & \stackrel{def}{=} & \mathtt{force}(x) \\ \mathcal{D}[\![\lambda x.\,M]\!] & \stackrel{def}{=} & \lambda x.\,\mathcal{D}[\![M]\!] \\ \mathcal{D}[\![MN]\!] & \stackrel{def}{=} & \mathcal{D}[\![M]\!](\mathtt{delay}(\mathcal{D}[\![N]\!])) \end{array}$$

*An adequacy results holds for $\mathcal{D}$ that is similar to the adequacy of the CPS transform $\mathcal{C}_{\mathtt{N}}$ (Theorem 6.3).*

*Let $\mathcal{M}$ be the encoding of $\lambda N$ into $\pi$-calculus defined as $\mathcal{N}$ (Table 5) but with a replication in the clause of abstraction:*

$$\mathcal{M}[\![\lambda x.\, M]\!] \stackrel{def}{=} (p).\, \overline{p}(v).\, !v(x)\, \mathcal{M}[\![M]\!]$$

*Prove that $\mathcal{M}$ factorises through the above encoding $\mathcal{D}$ and $\mathcal{V}$ (extended with (28)); i.e., for all $M \in \Lambda$*

$$\mathcal{V}[\![\mathcal{D}[\![M]\!]]\!] \approx \mathcal{M}[\![M]\!] \,.$$

We conclude the section by proving some properties of insensitivity to behavioural equivalences for the call-by-name encoding; we shall use some of these properties in later sections (Sections 11-13).

The next Lemma show that $\mathsf{i}/\mathsf{o}$ types are not actually necessary in the assertion of Corollary 6.13. This is in contrast with the corresponding result for call-by-value, see Remark 5.17. (The reason types are needed for call-by-value but not for call-by-name has to do with the replication theorems: for the proof of Lemma 5.19, the strong replication theorems of Section B.1.6 are necessary, and these theorems use $\mathsf{i}/\mathsf{o}$ types. They are necessary because, in the assertion of Lemma 5.19 name $x$ may appear in output object position in $M$ due to the definition of the encoding $\mathcal{V}$ on variables. The same does not happen in the call-by-name encoding $\mathcal{N}$, which uses a different translation of variables.)

Let us call $\mathcal{N}_\sharp$ the encoding into the plain polyadic $\pi$-calculus, without $\mathsf{i}/\mathsf{o}$ types; omitting types, the definition of $\mathcal{N}_\sharp$ is the same as that of $\mathcal{N}$; the difference is that in $\mathcal{N}_\sharp$ each $\mathsf{i}/\mathsf{o}$ type $I\,T$ (for $I \in \{\mathsf{i}, \mathsf{o}, \sharp\}$) is replaced by the less informative channel type $\sharp\, T$. We remind that in the plain polyadic $\pi$-calculus we write barbed congruence as $\approxeq^{\mathsf{c}}$, omitting the type environment index.

**Lemma 6.18** *Suppose* $\mathrm{fv}(M, N) \subseteq \widetilde{x}$ *and let* $H \stackrel{def}{=} \widetilde{x} \,:\, \mathsf{o}\,\mathsf{o}\,(\mathrm{Trig}) \,;\, \mathsf{o}\,(\mathrm{Trig})$ . *Then* $\mathcal{N}[\![M]\!] \approxeq^{\mathsf{c}}_H \mathcal{N}[\![N]\!]$ *iff* $\mathcal{N}_\sharp\,[\![M]\!] \approxeq^{\mathsf{c}} \mathcal{N}_\sharp\,[\![N]\!]$.

The proof of Lemma 6.18 is rather complex; rather than giving it, we invite the reader to prove the variant of Corollary 6.13 without $\mathsf{i}/\mathsf{o}$ types, through the following four exercises.

**Exercise 6.19** *Show that, for all $M, N \in \Lambda$ with $x \notin \mathrm{fv}(N)$,*

$$\mathcal{N}_\sharp\,[\![(\lambda x.\, M)N]\!]p \longrightarrow^2_{\mathtt{d}} \mathcal{N}_\sharp\,[\![M]\!]p\{x = \mathcal{N}_\sharp\,[\![N]\!]\}$$

**Exercise 6.20** *Show that, for all $M, N \in \Lambda$ with $x \notin \mathrm{fv}(N)$,*

$$\mathcal{N}_\sharp\,[\![M]\!]\{x = \mathcal{N}_\sharp\,[\![N]\!]\} \approx \mathcal{N}_\sharp\,[\![M\{^N\!/x\}]\!] \,.$$

*(Hint: proceed by induction on $M$, and use the replication theorems B.10.)*

**Exercise 6.21**
1. *From the two previous exercises, conclude that if $M \longrightarrow_{\mathtt{N}} N$ then $\mathcal{N}_\sharp\,[\![M]\!]p \longrightarrow^2_{\mathtt{d}} \approx \mathcal{N}_\sharp\,[\![N]\!]p$.*
2. *Check that in item (1), relation $\approx$ can be refined to $\gtrsim$ (the expansion relation, Definition B.6).*

**Exercise 6.22** *Use Exercise 6.21 to prove that if* $\lambda\beta \vdash M = N$ *then* $\mathcal{N}_\sharp [\![M]\!] \approxeq^c \mathcal{N}_\sharp [\![N]\!]$.

Exercise 6.21 can also be used to get a direct proof of Corollary 6.12.

The encoding of call-by-name satisfies some further properties of insensitivity to the behavioural equivalence chosen for the $\pi$-calculus. We show one of these (insensitivity to the choice between barbed congruence and ground bisimilarity), that will be useful in the study of full abstraction for this encoding, in Sections 11-13. We state the result, whose proof takes the reminder of the subsection, and is partly conducted through exercises (these are technical and fairly advanced exercises).

**Corollary 6.23** *For all* $M, N \in \Lambda$, *it holds that* $\mathcal{N}_\sharp [\![M]\!] \approxeq^c \mathcal{N}_\sharp [\![N]\!]$ *iff* $\mathcal{N}_\sharp [\![M]\!] \approx \mathcal{N}_\sharp [\![N]\!]$.

**Definition 6.24** *A process* $P$ *is* $\tau$-insensitive under ground transitions *if* $P \longrightarrow P'$ *implies* $P \approx P'$ *and, moreover, this property is preserved under ground transitions.*

**Lemma 6.25** *Suppose that* $P$ *and* $Q$ *are processes of the asynchronous polyadic* $\pi$*-calculus and* $\Gamma \vdash P, Q$. *If* $P$ *and* $Q$ *are* $\tau$*-insensitive under ground transitions, then* $P \approxeq^c Q$ *iff* $P \approx Q$.

**Proof:** [sketch] On processes of the asynchronous $\pi$-calculus, ground bisimulation implies barbed congruence (Theorem B.4).
If a process $R$ is $\tau$-insensitive under ground transitions, then for each ground action $\mu$ the set $\{R' : R \stackrel{\mu}{\Longrightarrow} R'\}$, modulo $\approx$, is finite. Using this fact, and proceeding in a way similar to that used for characterisations of barbed congruence in terms of labeled bisimilarity (such as early bisimilarity) [San92, ACS96], one proves the opposite direction. ■

**Definition 6.26** *A name* $a$ *is* directional *in* $P$ *if* $a$ *appears free in* $P$ *either only in input prefixes, or only in output prefixes.*

**Exercise* 6.27** *Suppose a* $\pi$*-calculus process* $P$ *satisfies these two properties:*

1. *all free names of* $P$ *are directional in* $P$*;*
2. *each name of* $P$ *is either linear receptive, or* $\omega$ *receptive (that is the process is typable using only the types for linear receptiveness and for* $\omega$ *receptiveness, Section B.1.4).*

*Then* $P$ *is* $\tau$*-insensitive under ground transitions.*

**Exercise* 6.28** *Prove that* $\mathcal{N}_\sharp [\![M]\!]p$ *satisfies the hypothesis of Exercise 6.27, for all* $M \in \Lambda$. *Conclude then that Corollary 6.23 is true.*

# 7 A uniform encoding

The differences between the definitions of application in the $\pi$-calculus encodings of call-by-name and call-by-value are inevitable — just because application is precisely where these strategies differ. One may wonder, however, whether the

*In this table we abbreviate $\mathcal{C}_{\mathtt{U}}[\![M]\!]$ as $[\![M]\!]$:*

$$\begin{array}{rcl} [\![x]\!] & \stackrel{\mathrm{def}}{=} & \lambda k.\, xk \\ [\![\lambda x.\, M]\!] & \stackrel{\mathrm{def}}{=} & \lambda k.\, k(\lambda x.\, [\![M]\!]) \end{array}$$

*call-by-name application:*

$$[\![MN]\!] \stackrel{\mathrm{def}}{=} \lambda k.\, [\![M]\!](\lambda v.\, v[\![N]\!]k)$$

*call-by-value application:*

$$[\![MN]\!] \stackrel{\mathrm{def}}{=} \lambda k.\, [\![M]\!](\lambda v.\, [\![N]\!](\lambda w.\, v(\lambda k.\, kw)k))$$

Table 6: A uniform CPS transform

definitions of abstraction and variable need to differ too. In this section, we make some simple modifications to these encodings to obtain new ones that *differ only* in the definitions of application. We shall then show that other forms of application, such as call-by-need application, can be easily defined.

Having a uniform encoding for a variety of strategies makes it easier to compare them. It also facilitates translation to $\pi$-calculus of programming languages that employ different strategies for evaluating arguments of functions.

We obtain the new encodings again by going through a CPS transform, an injection into HO$\pi$, and the compilation of HO$\pi$ into $\pi$-calculus. We begin with the CPS transform. Starting from the call-by-value and call-by-name transforms examined in the previous sections (Tables 2 and 4), it is easy to give a CPS transform that is *uniform* for call-by-value and call-by-name, in that it has the same clauses for abstraction and variables. The call-by-value and call-by-name CPS have the same clause for abstractions; to obtain a uniform CPS it suffices to adopt the call-by-name CPS, and modify the definition of application of the call-by-value CPS to compensate for the different clauses for variables. (We cannot adopt the definition of variables of the call-by-value CPS transform because it treats variables as values, and this is correct only when variables are always substituted by values.)

The uniform CPS transform is given in Table 6. The associated grammar, which is similar to that for the call-by-name CPS transform but lacks the constraint on linear occurrence of value variables, is this:

$$\begin{array}{rlll} \text{continuation variable} & k & & \\ \text{ordinary variables} & x, \ldots & & \\ \text{value variables} & v, w, \ldots & & \\ \text{answers} & P & := & KV \;\big|\; VAK \;\big|\; AK \\ \text{CPS-values} & V & := & \lambda x.\, \lambda k.\, P \;\big|\; v \\ \text{continuations} & K & := & k \;\big|\; \lambda v.\, P \\ \text{principal terms} & A & := & \lambda k.\, P \;\big|\; x \end{array} \qquad (29)$$

$$\mathcal{U}[\![\lambda x.\, M]\!] \stackrel{\text{def}}{=} (p).\, \overline{p}(v).\, !v(x)\, \mathcal{U}[\![M]\!]$$

$$\mathcal{U}[\![x]\!] \stackrel{\text{def}}{=} (p).\, \overline{x}\langle p\rangle$$

*call-by-value application:*

$$\mathcal{U}[\![MN]\!] \stackrel{\text{def}}{=} (p).\, (\boldsymbol{\nu} q\,)\Big(\mathcal{U}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} r\ (\ \mathcal{U}[\![N]\!]r \mid r(w).\, \boldsymbol{\nu} x\, \overline{v}\langle x, p\rangle.\, !x(r').\, \overline{r'}\langle w\rangle)\Big)$$

*call-by-name application:*

$$\mathcal{U}[\![MN]\!] \stackrel{\text{def}}{=} (p).\, (\boldsymbol{\nu} q\,)\Big(\mathcal{U}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} x\, \overline{v}\langle x, p\rangle.\, !x\, \mathcal{U}[\![N]\!]\Big)$$

*call-by-need application:*

$$\mathcal{U}[\![MN]\!] \stackrel{\text{def}}{=} (p).\, (\boldsymbol{\nu} q\,)\Big(\mathcal{U}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} x\, \overline{v}\langle x, p\rangle.\, x(r).\, \boldsymbol{\nu} q'\ (\ \mathcal{U}[\![N]\!]q' \mid q'(w).\, (\overline{r}\langle w\rangle \mid !x(r').\, \overline{r'}\langle w\rangle))\Big)$$

Table 7: The uniform encoding of call-by-name, call-by-value, call-by-need

The types of the non-terminals of the grammar are the same as those of the call-by-name CPS transform. With the usual injection on terms and on types, the terms generated by this grammar and their types become a sublanguage of HO$\pi$. Applying the compilation of HO$\pi$ into $\pi$-calculus we obtain the encoding given in Table 7. In this table, there is also the code for the call-by-need application. We present call-by-need directly on the $\pi$-calculus — without going through HO$\pi$ — because, as explained in Section 3.3, call-by-need is an implementation technique with explicit environment in which $\beta$-reduction does not require substituting a term for a variable but just substituting a reference to a term for a variable. Therefore for the encoding of call-by-need the process-passing features of HO$\pi$ are not so helpful.

We explain the encoding of a call-by-need application $MN$. When $\mathcal{U}[\![M]\!]q$ becomes a function it signals so on $q$, and receives a pointer $x$ to the argument $N$ together with the location $p$ for the next interaction. Now the evaluation of $M$ continues. When the argument $N$ is needed for the first time, a request is made on $x$. Then $\mathcal{U}[\![N]\!]r$ is evaluated and, when it becomes a value, a pointer to this value instantiates $w$. This pointer is returned to the process that requested $N$. When further requests for $N$ are made, the pointer is returned immediately. Thus, by contrast with call-by-name, in call-by-need the argument $N$ of the application is evaluated once. To appreciate this, the reader might like to compare the reductions of the encodings of $(\lambda.\, xx)(II)$, in call-by-name and in call-by-need. In the former, $II$ is evaluated twice, in the latter once.

It is not by chance that the call-by-need encoding is best derived from the uniform encoding of Table 7, because call-by-need combines elements of the call-by-name and call-by-value strategies. Indeed it can be defined as call-by-name plus sharing, but can also be seen as a variant of call-by-value where the argument of an application is evaluated at a different point.

We do not prove the correctness of the call-by-need encoding, for it would require formally introducing the call-by-need system. See the notes of Section 14 for references, and see also Exercise 7.3.

**Exercise 7.1** *Add type annotations for local names to the encodings in Table 7. What are the types of the free names? Show that these types are correct, by proving the appropriate type judgements on encodings of $\lambda$-terms.*

**Remark 7.2** Proving properties of the encodings of call-by-value $\lambda$-calculus and call-by-need in Table 7 (for instance, in call-by-value, the validity of $\beta_{\mathtt{v}}$ reduction) may require $\mathsf{i}/\mathsf{o}$ types. $\mathsf{i}/\mathsf{o}$ types may be avoided by adopting the modifications proposed in Exercise 7.3.

**Exercise 7.3** *Let $\mathcal{U}_{\mathrm{V}}^{*}$ be the encoding of call-by-value as defined in Table 7 with the exception that in the clause for application the last output $\overline{r'}\langle w\rangle$ is replaced by $\overline{r'}(w').\,!w' \multimap w$ (where $w' \multimap w$ is a* link*, Section B.1.5). Similarly, let $\mathcal{U}_{\mathrm{Ne}}^{*}$ be the encoding of call-by-need as defined in Table 7 with the exception that in the clause for application the last output $\overline{r'}\langle w\rangle$ is replaced by $\overline{r'}(w').\,!w' \multimap w$. Finally, let $\mathcal{U}_{\mathrm{N}}$ be the encoding of call-by-name in the same table.*

*Take a $\lambda$-term $M$ inside which all applications have the form $N(\lambda x.\,L)$ (i.e., the argument is an abstraction). Show that*

$$\mathcal{U}_{\mathrm{V}}^{*}[\![M]\!] \approx \mathcal{U}_{\mathrm{N}}[\![M]\!] \approx \mathcal{U}_{\mathrm{Ne}}^{*}[\![M]\!] \;.$$

# 8 Optimisations of the call-by-name encoding

We now examine a possible optimisation of the encoding $\mathcal{N}$ of $\lambda N$ into $\pi$-calculus (Table 5). Consider the encoding of abstraction in that table. The name $v$ sent via $p$ is immediately used in an input. Dually, in the definition of application the name received at $q$ is immediately used in output. We can compress the two communications into a single one where an abstraction uses its location in input. The resulting optimised encoding is shown in Table 8. This is the simplest encoding of the $\lambda$-calculus into $\pi$-calculus we are aware of. (It is also Milner's original encoding of the $\lambda$-calculus into $\pi$-calculus [Mil91].)

The optimisation from which the new encoding is obtained is valid in the standard semantics of the $\pi$-calculus; but it is not valid in an *asynchronous* semantics. Indeed, the encoding is not quite satisfactory with respect to asynchronous semantics such as asynchronous must testing or typed asynchronous barbed congruence. For instance, these semantics equate processes $\mathcal{M}[\![\lambda x.\,\Omega]\!]p$ and $\mathcal{M}[\![\Omega]\!]p$, but $\lambda x.\,\Omega$ and $\Omega$ are not behaviourally equivalent in $\lambda N$ (technically speaking, with these semantics the encoding is not sound). Behavioural equivalence in $\lambda N$ will be discussed in Section 11.

In summary, the encoding of Table 8 is attractive because it is the shortest of all the $\lambda$-calculus encodings presented, but it is less robust than the encoding of Table 5. In the remainder of the chapter we shall prefer the more robust encoding.

An optimisation of the encoding $\mathcal{N}$ (and of the other call-by-name encodings in this section and in Section 7) is possible using types. The definition of

$$\begin{array}{rcl}\mathcal{M}[\![\lambda x.\, M]\!] & \stackrel{\mathrm{def}}{=} & (p).\, p(x)\, \mathcal{M}[\![M]\!] \\ \mathcal{M}[\![x]\!] & \stackrel{\mathrm{def}}{=} & (p).\, \overline{x}\langle p\rangle \\ \mathcal{M}[\![MN]\!] & \stackrel{\mathrm{def}}{=} & (p).\, \boldsymbol{\nu} q\, (\mathcal{M}[\![M]\!]q \mid \boldsymbol{\nu} x\, \overline{q}\langle x, p\rangle.\, !x\, \mathcal{M}[\![N]\!])\end{array}$$

Table 8: An optimised encoding of $\lambda N$

---

application when the argument is a variable is, including type annotations:

$$\begin{array}{rcl}\mathcal{N}[\![My]\!] & \stackrel{\mathrm{def}}{=} & (p).\;\; (\boldsymbol{\nu} q : \sharp\, \mathrm{Trig}) \\ & & \quad \Big(\mathcal{N}[\![M]\!]q \mid\;\; q(v).\, (\boldsymbol{\nu} x : \sharp\, \mathsf{o}\, \mathrm{Trig})\, (\overline{v}\langle x, p\rangle.\, !x(r).\, \overline{y}\langle r\rangle)\Big)\end{array}$$

From the type $\sharp$ Trig of $q$ it follows that $v$ is used in the body with type Trig. The definition of Trig shows that only the output capability of names may be communicated along $v$. Therefore if we confine ourselves to the asynchronous $\pi$-calculus, then we can apply the law of Lemma B.14, and optimise the above clause to

$$\mathcal{N}[\![My]\!] \stackrel{\mathrm{def}}{=} (p).\, (\boldsymbol{\nu} q : \sharp\, \mathrm{Trig})\, \Big(\mathcal{N}[\![M]\!]q \mid\;\; q(v).\, \overline{v}\langle y, p\rangle\Big) \tag{30}$$

This optimisation is a useful one, because the appearance of variables as arguments of applications is rather frequent. It is the analogue of a very common tail-call-like optimisation of functional languages.

# 9 Encoding the $\xi$ rule

The only rule of the $\lambda\beta$ theory so far totally neglected is the $\xi$ rule, for evaluating underneath the $\lambda$. This rule is normally disallowed in implementation of programming languages. Nevertheless, it is interesting to see how the $\xi$ rule can be encoded, for at least two reasons. First, it is a test of expressiveness for the process calculus. Secondly, certain optimisations of compilers of programming languages act on the body of functions and have similarities with the $\xi$ rule.

In this section we show how to add the $\xi$ rule to the call-by-name encodings. Adding $\xi$ to the call-by-value and call-by-need encodings is much harder; we explain why at the end of the section. We shall work in the $\pi$-calculus only: we have gained enough experience by now with encodings of the $\lambda$-calculus in the previous sections so that it is not necessary to go through HO$\pi$ again. The strategy defined by the rules of call-by-name plus $\xi$ is called *strong call-by-name*.

We obtain an encoding of strong call-by-name by modifying the definition of abstraction in the call-by-name encoding of Section 6. This clause is, expanding the input at $v$:

$$\mathcal{N}[\![\lambda x.\, M]\!] \stackrel{\mathrm{def}}{=} (p).\, \overline{p}(v).\, v(x, q).\, \mathcal{N}[\![M]\!]q\ .$$

Intuitively, to allow the $\xi$ rule, we need to relax the sequentiality imposed by the input prefix $v(x, q)$ that guards the body $\mathcal{N}[\![M]\!]q$ of the function. Precisely,

$$
\begin{array}{rcl}
\mathcal{N}[\![\lambda x.\, M]\!] & \stackrel{\text{def}}{=} & (p).\, (\boldsymbol{\nu} x, q\,)\Big(\overline{p}(v).\, v(y,r).\, !x \multimap y \mid q \multimap r \mid \mathcal{N}[\![M]\!]q\Big) \\
\mathcal{N}[\![x]\!] & \stackrel{\text{def}}{=} & (p).\, \overline{x}\langle p\rangle \\
\mathcal{N}[\![MN]\!] & \stackrel{\text{def}}{=} & (p).\, \boldsymbol{\nu} q\, \Big(\mathcal{N}[\![M]\!]q \mid q(v).\, \boldsymbol{\nu} x\, (\overline{v}\langle x, p\rangle.\, !x\, \mathcal{N}[\![N]\!])\Big)
\end{array}
$$

Table 9: The encoding of strong $\lambda N$ into the $\pi$-calculus

we would like to replace this input with the delayed input introduced in Section B.1.8 thus (recall that the difference between a strong input $a(\widetilde{z}) : P$ and an ordinary input $a(\widetilde{z}).\, P$ is that the former allows reductions in the continuation $P$):

$$
\mathcal{N}[\![\lambda x.\, M]\!] \stackrel{\text{def}}{=} (p).\, \boldsymbol{\nu} v\, (\overline{p}\langle v\rangle \mid v(x,q) : \mathcal{N}[\![M]\!]q) \tag{31}
$$

The resulting encoding is correct for strong call-by-name, because it is obtained from an encoding that is correct for call-by-name and because the introduction of the strong input in (31) has precisely the effect of the $\xi$ rule of the $\lambda$-calculus.

It is shown in Section B.1.8 that, under certain conditions, a strong input $a(\widetilde{z}) : P$ can be coded up, applying transformation (57). The conditions are that processes be asynchronous and the continuation $P$ have only the output capability on the bound names $\widetilde{z}$. Both conditions hold in (31); the condition of the output holds because the type of $v$ is $(\mathrm{Trig})^{\leftarrow \sharp} = \sharp\, \langle \mathsf{o}\, \mathsf{o}\, \mathrm{Trig}, \mathsf{o}\, \mathrm{Trig}\rangle$, as shown in (26).

We can therefore apply transformation (57) to (31) to get

$$
\mathcal{N}[\![\lambda x.\, M]\!]p \stackrel{\text{def}}{=} (\boldsymbol{\nu} v, x, q\,)\Big(\overline{p}\langle v\rangle.\, v(y,r).\, !x \multimap y \mid q \multimap r \mid \mathcal{N}[\![M]\!]q\Big)
$$

(The links on the location names $r, q$ are not replicated because locations are used linearly, i.e., at most once.) The correctness of transformation (57) (and of its linear variant without replication on links) guarantees that the result is still a correct encoding of strong call-by-name. Table 9 gives the complete encoding of strong call-by-name, including the unchanged clauses for variable and application.

It is actually possible to eliminate the link $!x \multimap y$ in the translation of abstraction. The exercise below invites the reader to show this.

**Exercise 9.1** *A small modification of the encoding of Table 9 allows the elimination of the link $!x \multimap y$. Write down this encoding, and prove a correctness result for it analogous to that in Exercise 6.21. (Hint: begin from an encoding of call-by-name in which the definition of abstraction is*

$$
[\![\lambda x.\, M]\!] \stackrel{\mathit{def}}{=} (\boldsymbol{\nu} x, v\,)\big(\overline{p}\langle x, v\rangle \mid v\, [\![M]\!]\big)
$$

*and where the clause for application is changed accordingly.)*

In contrast, encoding call-by-value plus the $\xi$ rule is much more complex. To repeat the trick we used in the call-by-name case, we would need to allow reduction underneath replication, that is to add the rule

$$\frac{P \longrightarrow P'}{!P \longrightarrow !P'} \tag{32}$$

Rule (32) appears necessary to encode in $\pi$-calculus any strategy having at least rules $\beta_{\mathrm{v}}, \nu_{\mathrm{v}}, \xi$ (another such strategy is full $\beta$ reduction $\Longrightarrow_\beta$). The reason is this: Suppose the argument $N$ of an application $(\lambda x.\, M)N$ reduces to a function $\lambda y.\, N'$. As this function could be used arbitrarily-many times in $M$, the $\pi$-calculus encoding of $N$ should reduce to a replicated process. The $\xi$ would allow reduction within the body $N'$, and to model this in $\pi$-calculus we would need rule (32).

For similar reasons, rule (32) appears necessary for the encoding of any reduction strategy that allow rules $\beta, \mu, \nu$.

**Remark 9.2** Reasoning as above, we can also obtain an encoding of strong call-by-name by modifying the definition of abstraction in the optimised call-by-name encoding of Section 8. In this case, we need both transformation (57) and transformation (58). After eliminating a replication in front of a link because of linearity, we obtain

$$\mathcal{M}[\![\lambda x.\, M]\!] \stackrel{\text{def}}{=} (\boldsymbol{\nu} x, q\,)\Big(p(y,r).\,(!x \twoheadrightarrow y \mid q \twoheadrightarrow r) \mid \mathcal{M}[\![M]\!]q\Big)$$

However, we recall from Section B.1.8 that the proof of correctness of transformation (58) requires some advanced properties of process with $\mathsf{i}/\mathsf{o}$ types, that we have not examined in the paper. The same properties are needed to prove the correctness of the encoding above. ∎

**Exercise 9.3** *What are the types of the local and free names of the encoding in Table 9? Show that these types are correct, by proving the appropriate type judgements.*

## 10 Interpreting typed $\lambda$-calculi

In this section we show that the encodings of the previous section can be extended to encodings of typed $\lambda$-calculi. To do this we have to define translations on types to match those on terms. We analyse the case of the *simply-typed $\lambda$-calculus* in detail, and discuss subtyping and recursive types. For studies of other type systems, see the notes of Section 14.

In a core simply-typed $\lambda$-calculus, types are built from *base types*, such as integers and booleans, using the *arrow* type constructor. The syntax of terms is that of the untyped $\lambda$-calculus plus *base constants*. Each constant has a unique predefined type. We use only constants of base types: this is sufficient to have a non-empty set of (closed) well-typed terms. This simply typed $\lambda$-calculus is presented in Table 10. As usual, arrow type associates to the right, so $T \to S \to U$ reads $T \to (S \to U)$. Base types are ranged over by $t$, base constants by $c$. The reduction relation and the reduction strategies are defined as for the untyped calculus; the only difference is that the set of values for a reduction strategy normally contains the constants. We call the typed versions of $\lambda V$ and $\lambda N$ (*simply-*)*typed call-by-value* ($\lambda V^{\to}$) and (*simply-*)*typed call-by-name* ($\lambda N^{\to}$), respectively.

$$
\begin{array}{rrcll}
\textit{Terms} & M & := & x \;\big|\; c \;\big|\; \lambda x : T.\, M \;\big|\; MN & c \in \text{base constants} \\[1ex]
\textit{Types} & T & := & T_1 \to T_2 \;\big|\; t & t \in \text{base types} \\[1ex]
\textit{Type environments} & \Gamma & := & \emptyset \;\big|\; \Gamma, x : T &
\end{array}
$$

$$
\textit{Typing rules} \quad \frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x : S.\, M : S \to T} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}
$$

$$
\frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash MN : T}
$$

Table 10: The (core) simpy-typed $\lambda$-calculus

---

We add the same base constants and base types to HO$\pi$ and $\pi$-calculus and repeat the diagram of Figure 1, this time for $\lambda V^{\to}$ and $\lambda N^{\to}$. We show how to extend the encodings of $\lambda V$ and $\lambda N$ (from Sections 5 and 6), and their correctness results, to take account of types. The encodings in Sections 7-9 can be extended similarly.

**Lemma 10.1** *In the simply-typed $\lambda$-calculus, for every $\Gamma$ and $M$ there is at most one $T$ such that $\Gamma \vdash M : T$.*

## 10.1 The interpretation of typed call-by-value

We begin with the left part of Figure 1, that concerns $\lambda V^{\to}$. We follow a schema similar to that of Section 5, pointing out the main additions. The set of values of $\lambda V^{\to}$ also include constants:

$$V \quad := \quad \lambda x.\, M \;\big|\; x \;\big|\; c$$

As usual when translating typed calculi, the definition of $\mathcal{C}_{\mathtt{V}}$ (Table 2) on a term uses the type environment of the term as an index to be able to put the necessary type annotations in the target term. We shall not discuss these type annotations any further; they are determined by the definition of $\mathcal{C}_{\mathtt{V}}$ on types, explained below. We have to add a clause for constants:

$$\mathcal{C}^{\star}_{\mathtt{V}}[c]^{\Gamma} \quad \stackrel{\text{def}}{=} \quad c$$

It is important to understand how the CPS transform acts on types. Recalling from Section 5 that $\diamond$ is a distinguished type of answers (answers being the 'results' of CPS terms), the call-by-value CPS-transform modifies the types of $\lambda V^{\to}$ -terms as follows:

$$\mathcal{C}_{\mathtt{V}}[\![T]\!] \quad \stackrel{\text{def}}{=} \quad (\mathcal{C}^{\star}_{\mathtt{V}}[T] \to \diamond) \to \diamond \tag{33}$$

$$
\begin{array}{rcll}
\mathcal{C}^{\star}_{\mathtt{V}}[t] & \stackrel{\text{def}}{=} & t & \text{if } t \text{ is a base type} \\
\mathcal{C}^{\star}_{\mathtt{V}}[S \to T] & \stackrel{\text{def}}{=} & \mathcal{C}^{\star}_{\mathtt{V}}[S] \to \mathcal{C}_{\mathtt{V}}[\![T]\!] &
\end{array}
$$

The translation of arrow types is sometimes called the 'double-negation construction' because, writing $\neg T$ for $T \to \diamond$, we have

$$\mathcal{C}^{\star}_{\mathtt{V}}[S \to T] = \mathcal{C}^{\star}_{\mathtt{V}}[S] \to \neg\neg\mathcal{C}^{\star}_{\mathtt{V}}[T] \; .$$

Type environments are modified accordingly:

$$\begin{array}{rcl} \mathcal{C}_{\mathtt{V}}[\![\emptyset]\!] & \stackrel{\mathrm{def}}{=} & \emptyset \\ \mathcal{C}_{\mathtt{V}}[\![\Gamma, x : S]\!] & \stackrel{\mathrm{def}}{=} & \mathcal{C}_{\mathtt{V}}[\![\Gamma]\!], x : \mathcal{C}_{\mathtt{V}}[\![S]\!] \end{array}$$

and similarly for $\mathcal{C}^{\star}_{\mathtt{V}}$. The correctness of this translation of types is given in

**Theorem 10.2 (correctness of call-by-value CPS on types)** *If $M \in \Lambda$ then*

$$\Gamma \vdash M : T \quad \textit{iff} \quad \mathcal{C}^{\star}_{\mathtt{V}}[\Gamma] \vdash \mathcal{C}_{\mathtt{V}}[\![M]\!]^{\Gamma} : \mathcal{C}_{\mathtt{V}}[\![T]\!] \; .$$

(It follows that for any value $V$,

$$\Gamma \vdash V : T \quad \text{iff} \quad \mathcal{C}^{\star}_{\mathtt{V}}[\Gamma] \vdash \mathcal{C}^{\star}_{\mathtt{V}}[V]^{\Gamma} : \mathcal{C}^{\star}_{\mathtt{V}}[T]$$

which shows the agreement between the definitions of the auxiliary function $\mathcal{C}^{\star}_{\mathtt{V}}$ on terms and on types.)

**Remark 10.3** Schema (33) is also useful for understanding the types of the CPS images of the untyped $\lambda V$ in (16), because the untyped $\lambda$-calculus can be described as a typed $\lambda$-calculus in which all terms have the recursive type

$$T \stackrel{\mathrm{def}}{=} \mu X . (X \to X). \tag{34}$$

To apply the type translation to (34) we just need to add the clauses for type variables and for recursion to those in (33):

$$\mathcal{C}^{\star}_{\mathtt{V}}[X] \stackrel{\mathrm{def}}{=} X \qquad \mathcal{C}^{\star}_{\mathtt{V}}[\mu X . T] \stackrel{\mathrm{def}}{=} \mu X . \mathcal{C}^{\star}_{\mathtt{V}}[T] \; . \tag{35}$$

The translation of type $T$ in (34) is then

$$\mathcal{C}^{\star}_{\mathtt{V}}[T] \quad = \quad \mu X . (X \to (X \to \diamond) \to \diamond)$$

which is precisely type $T_V$ in (16); moreover

$$\begin{array}{rcl} \mathcal{C}_{\mathtt{V}}[\![T]\!] & = & (\mathcal{C}^{\star}_{\mathtt{V}}[T] \to \diamond) \to \diamond \\ & = & (T_V \to \diamond) \to \diamond \\ & = & T_A \; . \end{array}$$

■

The grammar of CPS terms is obtained from grammar 15 by adding a production for constants to those defining CPS-values. The relationship between the CPS grammar and HO$\pi$ is as in the untyped case, both on terms and on types. That is, modulo a modification of the syntax and some uncurrying, the CPS grammar generates a sublanguage of HO$\pi$. Thus Theorem 10.2 can also be read as a result about the encoding of $\lambda V^{\to}$ into HO$\pi$.

Finally, we apply the compilation $\mathcal{C}$ of HO$\pi$ terms and types into $\pi$-calculus terms and types, and we obtain the encoding of $\lambda V^{\to}$ into typed $\pi$-calculus

*In this table we abbreviate $\mathcal{V}[\![M]\!]$ as $[\![M]\!]$, and for a type or type expression $E$, we abbreviate $\mathcal{V}^\star[E]$ as $[E]$ and $\mathcal{V}[\![E]\!]$ as $[\![E]\!]$:*

*Translation of types:*

$$[\![T]\!] \stackrel{\text{def}}{=} \mathsf{o}\,\mathsf{o}\,[T]$$

$$\begin{array}{rcll} [t] & \stackrel{\text{def}}{=} & t & t \in \text{base types} \\ [S \to T] & \stackrel{\text{def}}{=} & [S] \frown [\![T]\!] = \mathsf{o}\,\langle [S], \mathsf{o}\,[T]\rangle & \end{array}$$

*Translation of type environments:*

$$\begin{array}{rcl} [\![\emptyset]\!] = [\emptyset] & \stackrel{\text{def}}{=} & \emptyset \\ [\![\Gamma, x : S]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!], x : [\![S]\!] \\ [\Gamma, x : S] & \stackrel{\text{def}}{=} & [\Gamma], x : [S] \end{array}$$

*Translation of terms:*

$$\begin{array}{rcl} [\![\lambda x : S.\, M]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{p}(y : [S \to T]^{\leftarrow\sharp}).\, !y(x)\, [\![M]\!]^{\Gamma, x : S} \\ [\![x]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{p}\langle x\rangle \\ [\![c]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{p}\langle c\rangle \\ [\![MN]\!]^\Gamma & \stackrel{\text{def}}{=} & \end{array}$$

$$(p).\,(\boldsymbol{\nu} q : \sharp\,[S \to T])\,\Big([\![M]\!]^\Gamma q \mid q(x).\,(\boldsymbol{\nu} r : \sharp\,[S])\,([\![N]\!]^\Gamma r \mid r(y).\,\overline{x}\langle y, p\rangle)\Big)$$

where, in the encoding of application, $S$ is the unique type such that $\Gamma \vdash N : S$ (the type is unique by Lemma 10.1).

Table 11: The encoding of $\lambda V^{\to}$ into $\pi$-calculus

in Table 11, and the results below about its correctness. Apart from type annotations, the translation of terms is the same as for the untyped calculus, with the addition of the clause for translating constants. Recalling that $(T)^-$ is the type obtained from $T$ by cancelling the outermost $\mathsf{i}/\mathsf{o}$ tag, and therefore $(\mathcal{V}[\![T]\!])^-$ is $\mathsf{o}\,\mathcal{V}^\star[T]$, the translation of Theorem 10.2 into $\pi$-calculus gives:

**Corollary 10.4** *Let $M$ be a term of a simply typed $\lambda$-calculus. Then*

$$\Gamma \vdash M : T \quad \textit{iff} \quad \mathcal{V}^\star[\Gamma], p : (\mathcal{V}[\![T]\!])^- \vdash \mathcal{V}[\![M]\!]^\Gamma p \ .$$

The results for the encoding of untyped $\lambda V$, namely validity of $\beta_{\mathtt{v}}$-rule and adequacy (Corollary 5.16 and 5.15), remain valid for the typed calculus, with the necessary modifications to the statements to take account of types. For instance, Corollary 5.16 becomes

**Corollary 10.5 (validity of $\beta_{\mathtt{v}}$ theory)** *Suppose that $\Gamma \vdash M : T$ and $\Gamma \vdash N : T$, and let $H \stackrel{def}{=} \mathcal{V}^\star[\Gamma] ; (\mathcal{V}[\![T]\!])^-$. If $\lambda\beta_{\mathtt{v}} \vdash M = N$ then $\mathcal{V}[\![M]\!]^\Gamma \approxeq^{\mathsf{c}}_H \mathcal{V}[\![N]\!]^\Gamma$.*

**Exercise 10.6** *Derive the subject reduction property of $\lambda V^{\to}$ from the $\pi$-calculus encoding; that is, prove that if $\Gamma \vdash M : T$ and $M \longrightarrow_{\mathtt{v}} N$ then also $\Gamma \vdash N : T$.*

**Remark 10.7** The types of $\pi$-calculus names used for the encoding of untyped $\lambda V$ in Section 5 agree with those used for $\lambda V^{\to}$ in this section, when we view the untyped $\lambda$-calculus as a typed $\lambda$-calculus where the only type is $\mu X.\, X \to X$ (Remark 10.3). From (35), the translation of recursive types and type variables of $\lambda V$ into $\pi$-calculus is

$$\mathcal{V}^\star[X] \stackrel{\text{def}}{=} X \qquad \mathcal{V}^\star[\mu X.\, T] \stackrel{\text{def}}{=} \mu X.\, \mathcal{V}^\star[T]$$

Therefore type Val of Section 5 is precisely $\mathcal{V}^\star[\mu X.\,(X \to X)]$, so that Lemma 5.14 can be presented thus: If $\mathrm{fv}(M) \subseteq \widetilde{x}$, then

$$\widetilde{x} : \mathcal{V}^\star[\mu X.\,(X \to X)], p : (\mathcal{V}[\![\mu X.\,(X \to X)]\!])^- \vdash \mathcal{V}[\![M]\!]^{\widetilde{x}\,:\,\mu X.(X \to X)} p$$

■

**Remark 10.8 (subtyping)** In typed $\lambda$-calculi with subtyping, the arrow type is contravariant in the first argument, and covariant in the second. Therefore, if $<$ is the subtype relation, then $S \to T < S' \to T'$ holds if $S' < S$ and $T < T'$ hold. For instance, suppose `Int` and `Real` are the types of integers and real numbers. It holds that `Int` $<$ `Real` (an integer is also a real number) and therefore `Real` $\to$ `Int` $<$ `Int` $\to$ `Real`. This is correct because a function that takes a real and returns an integer may also be used as a function that takes an integer and returns a real (but the converse is false).

The $\mathsf{i}/\mathsf{o}$ tag $\mathsf{o}$ is a contravariant type constructor. In the translation of an arrow type $S \to T$, component $\mathcal{V}^\star[S]$ is in contravariant position, because it is underneath an odd number of $\mathsf{o}$ tags; in contrast $\mathcal{V}^\star[T]$ is in covariant position, because it is underneath an even number of $\mathsf{o}$ tags. Therefore the $\pi$-calculus translation of types validates the subtyping rule for arrow type. As a consequence, the translation of this section can be extended to one of $\lambda V^{\to}$ with subtyping.

■

**Exercise* 10.9** *Give a direct proof (i.e., without going through the CPS transform) of the left to right implication of Corollary 10.4 (if* $\Gamma \vdash M : T$ *then* $\mathcal{V}^\star[\Gamma], p : (\mathcal{V}[\![T]\!])^- \vdash \mathcal{V}[\![M]\!]^\Gamma p$ *). (Hint: proceed by induction on the structure of* $M$*; the hardest case is application, where you will need Lemmas B.11 and B.12.)*

## 10.2 The interpretation of typed call-by-name

Constants are also among the values of $\lambda N^{\rightarrow}$:

$$\text{Values} \quad V \quad := \quad \lambda x.\, M \;\Big|\; c$$

Therefore we add a clause for constants to the definition of the CPS transform in Table 4:

$$\mathcal{C}^\star_{\mathtt{N}}[c]^\Gamma \quad \stackrel{\text{def}}{=} \quad c$$

The call-by-name CPS modifies types as follows.

$$\begin{array}{rcll}
\mathcal{C}_{\mathtt{N}}[\![T]\!] & \stackrel{\text{def}}{=} & (\mathcal{C}^\star_{\mathtt{N}}[T] \to \diamond) \to \diamond & \\[1em]
\mathcal{C}^\star_{\mathtt{N}}[t] & \stackrel{\text{def}}{=} & t & \text{if } t \text{ is a base type} \\
\mathcal{C}^\star_{\mathtt{N}}[S \to T] & \stackrel{\text{def}}{=} & \mathcal{C}_{\mathtt{N}}[\![S]\!] \to \mathcal{C}_{\mathtt{N}}[\![T]\!] &
\end{array} \tag{36}$$

Type environments are then translated thus:

$$\begin{array}{rcl}
\mathcal{C}_{\mathtt{N}}[\![\emptyset]\!] & \stackrel{\text{def}}{=} & \emptyset \\
\mathcal{C}_{\mathtt{N}}[\![\Gamma, x : S]\!] & \stackrel{\text{def}}{=} & \mathcal{C}_{\mathtt{N}}[\![\Gamma]\!], x : \mathcal{C}_{\mathtt{N}}[\![S]\!]
\end{array}$$

and similarly for $\mathcal{C}^\star_{\mathtt{N}}$.

**Theorem 10.10 (correctness of call-by-name CPS on types)** *Let* $M \in \Lambda$*. Then*

$$\Gamma \vdash M : T \quad \textit{iff} \quad \mathcal{C}_{\mathtt{N}}[\![\Gamma]\!] \vdash \mathcal{C}_{\mathtt{N}}[\![M]\!]^\Gamma : \mathcal{C}_{\mathtt{N}}[\![T]\!]\,.$$

A corollary is that for every value $V$ of $\lambda N^{\rightarrow}$,

$$\Gamma \vdash V : T \quad \text{iff} \quad \mathcal{C}_{\mathtt{N}}[\![\Gamma]\!] \vdash \mathcal{C}^\star_{\mathtt{N}}[V]^\Gamma : \mathcal{C}^\star_{\mathtt{N}}[T]$$

**Remark 10.11** Adding clauses for type variables and for recursion as in (35), the above translation of types can also be applied to untyped $\lambda N$. The translation of $T \stackrel{\text{def}}{=} \mu X.\,(X \to X)$ is

$$\mathcal{C}^\star_{\mathtt{N}}[T] \quad = \quad \mu X.\,\Big(((X \to \diamond) \to \diamond) \to ((X \to \diamond) \to \diamond)\Big)$$

and

$$\mathcal{C}_{\mathtt{N}}[\![T]\!] \quad = \quad (\mathcal{C}^\star_{\mathtt{N}}[T] \to \diamond) \to \diamond$$

which is type $T_A$. ■

*In this table we abbreviate $\mathcal{N}[\![M]\!]$ as $[\![M]\!]$, and for a type or type expression $E$, we abbreviate $\mathcal{N}^\star[E]$ as $[E]$ and $\mathcal{N}[\![E]\!]$ as $[\![E]\!]$:*

*Translation of types:*

$$[\![T]\!] \stackrel{\text{def}}{=} \mathsf{o}\,\mathsf{o}\,[N]$$

$$\begin{array}{lcll} [N] & \stackrel{\text{def}}{=} & t & t \in \text{base types} \\ [N] & \stackrel{\text{def}}{=} & [\![S]\!] \frown [\![T]\!] = \mathsf{o}\,\langle [\![S]\!], \mathsf{o}\,[N] \rangle & \end{array}$$

*Translation of type environments:*

$$\begin{array}{rcl} [\![\emptyset]\!] = [N] & \stackrel{\text{def}}{=} & \emptyset \\ [\![\Gamma, x : S]\!] & \stackrel{\text{def}}{=} & [\![\Gamma]\!], x : [\![S]\!] \\ [N] & \stackrel{\text{def}}{=} & [N], x : [N] \end{array}$$

*Translation of terms:*

$$\begin{array}{rcl} [\![\lambda x : S.\, M]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{p}(v : [N]^{\leftarrow\sharp}).\, v(x)\, [\![M]\!]^{\Gamma, x : S} \\ [\![x]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{x}\langle p \rangle \\ [\![c]\!]^\Gamma & \stackrel{\text{def}}{=} & (p).\, \overline{p}\langle c \rangle \\ [\![MN]\!]^\Gamma & \stackrel{\text{def}}{=} & \end{array}$$

$$(p).\, (\boldsymbol{\nu} q : \sharp\, [N]) \Big( [\![M]\!]^\Gamma q \mid\ q(v).\, (\boldsymbol{\nu} x : \sharp\, [N])\, (\overline{v}\langle x, p \rangle.\, !x\, [\![N]\!]^\Gamma) \Big)$$

where, in the encoding of application, $S$ is the unique type such that $\Gamma \vdash N : S$.

Table 12: The encoding of $\lambda N^{\rightarrow}$ into the $\pi$-calculus

The modifications needed to the CPS grammar for untyped $\lambda N$, and the injection of the terms generated by the grammar to HO$\pi$, are the same as for call-by-value.

The final step of the compilation of HO$\pi$ into $\pi$-calculus gives us the encoding of types, type environments, and terms of $\lambda N^{\rightarrow}$ into $\pi$-calculus in Table 12. Theorem 10.10 then gives

**Corollary 10.12 (correctness of $\mathcal{N}$ on types)** *Let $M$ be a term of a simply typed $\lambda$-calculus. Then*

$$\Gamma \vdash M : T \quad \textit{iff} \quad \mathcal{N}[\![\Gamma]\!], p : (\mathcal{N}[\![T]\!])^{-} \vdash \mathcal{N}[\![M]\!]^{\Gamma} p \ .$$

The results on $\mathcal{N}$ from the untyped case, namely Corollaries 6.12 and 6.13 and Exercise 6.22, remain valid, with the expected modifications to the types in the statement of Corollary 6.13.

**Remark 10.13** The types of $\pi$-calculus names used for the encoding of untyped $\lambda N$ in Section 6 agree with those used for $\lambda N^{\rightarrow}$ in this section, once we add clauses

$$\mathcal{N}^{\star}[X] \stackrel{\text{def}}{=} X \qquad \mathcal{N}^{\star}[\mu X . T] \stackrel{\text{def}}{=} \mu X . \mathcal{N}^{\star}[T]$$

The type Trig of Section 6 is precisely $\mathcal{N}^{\star}[\mu X . (X \rightarrow X)]$, so that Lemma 6.11 can be presented thus: Suppose $\text{fv}(M) \subseteq \widetilde{x}$. Then

$$\widetilde{x} : \mathcal{N}[\![\mu X . (X \rightarrow X)]\!], p : (\mathcal{N}[\![\mu X . (X \rightarrow X)]\!])^{-} \vdash \mathcal{N}[\![M]\!]^{\widetilde{x} : \mu X . (X \rightarrow X)} p \ .$$

■

**Remark 10.14 (subtyping)** As in call-by-value, so in call-by-name the encoding of types validates the standard subtyping rule for arrow types. ■

**Exercise* 10.15** *Give a direct proof (i.e., without going through the CPS transform) of the left to right implication of Corollary 10.12 (if $\Gamma \vdash M : T$ then $\mathcal{N}[\![\Gamma]\!], p : (\mathcal{N}[\![T]\!])^{-} \vdash \mathcal{N}[\![M]\!]^{\Gamma} p$ .). (Hint: similar to Exercise 10.9.)*

**Exercise* 10.16** *Prove the analogue of Corollary 10.12 in the case that $\mathcal{N}$ is the encoding of Table 8.*

# 11 The full abstraction problem for the $\pi$-interpretation of call-by-name

An interpretation of the $\lambda$-calculus into $\pi$-calculus, as a translation of one language into another, can be considered a form of denotational semantics. The denotation of a $\lambda$-term is an equivalence class of processes. These equivalence classes are the quotient of the $\pi$-calculus processes with respect to the behavioural equivalence (barbed congruence) adopted for the $\pi$-calculus.

In the previous sections, we have seen various $\pi$-calculus interpretations of $\lambda$-calculi and have shown their soundness with respect to the *axiomatic* semantics of the calculi (where equivalence between $\lambda$-terms means provable equality from an appropriate set of axioms and inference rules). In this section we go further and compare the $\pi$-calculus semantics with the *operational* semantics of the $\lambda$-calculus. We study the important case of the untyped call-by-name $\lambda$-calculus ($\lambda N$); the problem is harder in the call-by-value case, and is briefly discussed in Section 13.5. The encoding of $\lambda N$ into $\pi$-calculus will be that of Table 4, but without $\mathsf{i/o}$ types. We can assume that the encoding is into the plain polyadic $\pi$-calculus, without $\mathsf{i/o}$ types, because Lemma 6.18 shows that $\mathsf{i/o}$ types do not affect behavioural equivalence; in Lemma 6.18 we indicated the encoding without $\mathsf{i/o}$ types as $\mathcal{N}_\sharp$. From now on we omit indices that signify call-by-name; thus the relations $\longrightarrow_{\mathtt{N}}$ and $\Longrightarrow_{\mathtt{N}}$ become $\longrightarrow$ and $\Longrightarrow$, and the encoding $\mathcal{N}_\sharp [\![ \,.\, ]\!]$ becomes $[\![ \,.\, ]\!]$.

An interpretation of a calculus is said to be *sound* if it equates only operationally equivalent terms, *complete* if it equates all operationally equivalent terms, and *fully abstract* if it is sound and complete. We show in this section that the $\pi$-calculus interpretation of $\lambda N$ is sound, but not complete.

When an interpretation of a calculus is not fully abstract, one may hope to achieve full abstraction by

1. enriching the calculus,

2. choosing a finer notion of operational equivalence for the calculus, or

3. cutting down the codomain of the interpretation.

In Sections 12 and 13 we prove full abstraction results for the $\pi$-interpretation by following (1) and (2). In Section 14 we shall hint that our main theorems are, by large, independent of the behavioural equivalence chosen for the $\pi$-calculus, which suggests that (3) is less interesting. We begin by presenting the standard operational semantics of $\lambda N$.

Sometimes we simply refer to $\lambda N$ as 'the $\lambda$-calculus'; this is partly justified by the fact that the call-by-name strategy is a *weakly normalising* strategy, that is a term converges under the full $\beta$ reduction of the $\lambda$-calculus iff it does so under call-by-name (therefore the operational equivalence relation of Definition 11.1 below does not change if the convergence predicate is taken to mean convergence under the full $\beta$ relation $\longrightarrow_\beta$).

## 11.1 Applicative bisimilarity

In an operational semantics, two terms are deemed equivalent if they have the same observable behaviour in all contexts. What is an appropriate notion of observability for $\lambda$-terms? We have adopted the viewpoint that what is observable of a *process* are its interactions with its environment. Regarding functions as processes, it is natural to stipulate that a (closed) $\lambda$-term is observable if it is an abstraction, which interacts with its environment by consuming an argument. (In the typed $\lambda$-calculus, also constants like integers and booleans would be observable; the integer or the boolean that a term reduces to can be thought as the output of that term to the environment.)

Having decided what is observable, we can define barbed congruence for any reduction strategy we like, in particular for call-by-name, the strategy that interests us. As call-by-name is confluent (indeed deterministic), the definition of barbed congruence can be simplified, by removing the bisimulation clause on interactions. The definition then becomes the same as that of *Morris's context-equivalence*, sometimes called *observation equivalence* in the literature. A *closed context* is one without free variables.

**Definition 11.1 (observation equivalence, or barbed congruence for $\lambda N$)** *Let $M, N \in \Lambda$. We say that $M$ and $N$ are* observationally equivalent*, or* barbed congruent*, if, in all closed contexts $C$, it holds that $C[M] \Downarrow_{\mathtt{N}}$ iff $C[N] \Downarrow_{\mathtt{N}}$.*

As for the $\pi$-calculus, a tractable characterisation of barbed congruence on $\lambda$-terms is useful.

**Definition 11.2** *A symmetric relation $\mathcal{R} \subseteq \Lambda^0 \times \Lambda^0$ is an* applicative bisimulation *if $M \mathrel{\mathcal{R}} N$ and $M \Longrightarrow \lambda x.\, M'$ imply that there is an $N'$ such that $N \Longrightarrow \lambda x.\, N'$ and $M'\{^L\!/\!_x\} \mathrel{\mathcal{R}} N'\{^L\!/\!_x\}$, for all $L \in \Lambda^0$. Two terms $M, N \in \Lambda^0$ are* applicative bisimilar*, written $M \approx_\lambda N$, if $M \mathrel{\mathcal{R}} N$ holds, for some applicative bisimulation $\mathcal{R}$.*

Applicative bisimilarity is extended to open terms using closing substitutions: if $M, N \in \Lambda$ with $\mathrm{fv}(M, N) \subseteq \widetilde{x}$, then $M \approx_\lambda N$ if for all $\widetilde{L} \subseteq \Lambda^0$, we have $M\{^{\widetilde{L}}\!/\!_{\widetilde{x}}\} \approx_\lambda N\{^{\widetilde{L}}\!/\!_{\widetilde{x}}\}$.

**Theorem 11.3** *Applicative bisimilarity and observation equivalence coincide.*

**Proof:** The proof in [AO93, page 11] is by Stoughton and uses a variant of Berry's context lemma [Ber81]. ■

Therefore, applicative bisimilarity is a direct characterisation of barbed congruence on $\lambda N$ in the same way as ground and early bisimilarities are direct characterisations of barbed congruence on the $\pi$-calculus.

In the light of this characterisation, and of the fact that applicative bisimilarity is a mathematically more tractable relation than barbed congruence, we shall prefer to use applicative bisimilarity rather than barbed congruence.

**Exercise 11.4 ($\eta$ rule)**

1. *Show that the $\eta$ rule is not operationally valid in the call-by-name $\lambda$-calculus, i.e., show that there is a $\lambda$-term $M$ with $x \notin \mathrm{fv}(M)$ such that $\lambda x.\, (Mx) \not\approx_\lambda M$.*

2. *Show that the $\eta$ rule is valid if $M$ is a value, i.e., if $x \notin \mathrm{fv}(\lambda y.\, N)$ then $\lambda x.\, ((\lambda y.\, N)x) \approx_\lambda \lambda y.\, N$.*

## 11.2 Soundness and non-completeness

We now compare applicative bisimilarity with the equivalence on $\lambda$-terms induced by the encoding into $\pi$-calculus.

**Definition 11.5** *We write $M =_\pi N$ if $[\![M]\!] \approxeq^{\mathrm{c}} [\![N]\!]$. We call $=_\pi$ the* local structure of the $\pi$-interpretation*.*

Because of Corollary 6.23, we can work with ground bisimilarity ($\approx$) in place of barbed congruence. Ground bisimilarity is easier to work with.

**Remark 11.6** The terminology in Definition 11.5 is consistent with the standard terminology of the $\lambda$-calculus. The local structure of a $\lambda$-model is the equality on $\lambda$-terms induced by that model (two $\lambda$-terms are equal if they have the same interpretation in the model). The encoding $[\![.]\!]$ of the $\lambda$-calculus into $\pi$-calculus gives rise to a $\lambda$-model. This follows from the facts that the encoding is compositional and that it validates the $\lambda\beta$ theory (Corollary 6.13); see [San95a] for the details. ■

From the adequacy of the interpretation (Corollary 6.12), we can prove its operational soundness.

**Proposition 11.7** *$M =_\pi N$ implies $M \approx_\lambda N$.*

**Proof:** We have to show that for all $M, N \in \Lambda$ and $p$, if $[\![M]\!]p \approx [\![N]\!]p$ then $M \approx_\lambda N$. We exploit the characterisation of $\approx_\lambda$ as observation equivalence (Theorem 11.3).
From $[\![M]\!]p \approx [\![N]\!]p$ we deduce that $[\![M]\!]p \Downarrow$ iff $[\![N]\!]p \Downarrow$. Therefore since $[\![.]\!]$ is compositional and $\approx$ is a congruence on the asynchronous $\pi$-calculus, from $[\![M]\!]p \approx [\![N]\!]p$ we also deduce that $[\![C[M]]\!]p \Downarrow$ iff $[\![C[N]]\!]p \Downarrow$, for all closing $\lambda$-calculus contexts $C$. So by Corollary 6.12, $C[M] \Downarrow$ iff $C[N] \Downarrow$, for any $C$, which proves that $M$ and $N$ are observationally equivalent. ■

While soundness is a necessary requirement for an interpretation, completeness (the converse of soundness) is a very strong demand, one that often fails. We would not expect the $\pi$-calculus semantics of the $\lambda$-calculus to be complete: the class of $\pi$-calculus contexts is much richer than the class of $\lambda$-calculus contexts, and hence potentially more discriminating. In the $\pi$-calculus one can express parallelism and non-determinism that, as discussed in Section 2.2, are not expressible in the $\lambda$-calculus.

More concretely, there are at least two reasons for not expecting the $\pi$-calculus semantics to be complete. The first has to do with the call-by-name CPS transform, from which the $\pi$-calculus encoding was derived. This transform, as a translation of $\lambda N$ into $\lambda V$, is not complete: there are terms of $\lambda N$ that are applicative bisimilar (and therefore operationally indistinguishable) but whose CPS images are distinguishable as terms of $\lambda V$. It is reasonable to expect that the distinctions made by $\lambda V$ contexts can be exposed by $\pi$-calculus contexts. The second concrete reason for not expecting the $\pi$-calculus semantics to be complete is some results on the canonical model of $\lambda N$. This model is defined as the solution to a domain equation [Abr87, Abr89]. The model is sound but not complete. Completeness fails because the model contains the denotation of terms that are not definable in $\lambda N$, and whose addition to $\lambda N$ increases the discriminating power of the contexts of the language. Examples are *convergence test* and *parallel convergence test*, fairly simple operators (defined below) that one expects to be expressible in $\pi$-calculus.

Both the CPS transform and the models of $\lambda N$ offer good candidates for counterexamples to the completeness of the $\pi$-calculus encoding. We begin by looking at those from the models as they give insight into what makes the $\pi$-calculus more discriminating than the $\lambda$-calculus.

Convergence test is a unary operator, $\mathtt{C}$, that can detect whether its argument converges; it is defined by these rules:

$$\mathtt{C1}\frac{M\downarrow}{\mathtt{C}M \longrightarrow I} \qquad\qquad \mathtt{C2}\frac{M \longrightarrow N}{\mathtt{C}M \longrightarrow \mathtt{C}N}$$

Parallel convergence test is a binary operator, $\mathtt{P}$, that can detect whether either of its arguments converges; it is defined by these rules:

$$\mathtt{P1}\frac{M\downarrow}{\mathtt{P}MN \to I} \qquad\qquad \mathtt{P2}\frac{M\downarrow}{\mathtt{P}NM \longrightarrow I}$$

$$\mathtt{P3}\frac{M \longrightarrow M'}{\mathtt{P}MN \to \mathtt{P}M'N} \qquad\qquad \mathtt{P4}\frac{M \longrightarrow M'}{\mathtt{P}NM \to \mathtt{P}NM'}$$

Let $\lambda\mathtt{C}$ be $\lambda N$ with the addition of convergence test, and $\lambda\mathtt{P}$ be $\lambda N$ with the addition of parallel convergence test. In $\lambda\mathtt{P}$, convergence test is definable as $\mathtt{C}M \stackrel{\mathrm{def}}{=} \mathtt{P}M\Omega$.

Here are two terms that are operationally indistinguishable in the pure $\lambda$-calculus but that can be are distinguished in $\lambda\mathtt{C}$:

$$\begin{array}{rcl} M & \stackrel{\mathrm{def}}{=} & \lambda x.\,(x(\lambda y.\,(x\Xi\Omega y))\Xi) \\ N & \stackrel{\mathrm{def}}{=} & \lambda x.\,(x(x\Xi\Omega)\Xi). \end{array} \tag{37}$$

These terms are further discussed in Remark 13.9.

**Remark 11.8** There are also $\lambda$-terms that are indistinguishable in $\lambda\mathtt{C}$ but that can be distinguished in $\lambda\{\mathtt{C},\mathtt{P}\}$ with the help of the parallel convergence test, see [BL96]. ■

Now, to prove that the $\pi$-calculus semantics is strictly finer than the operational semantics of the $\lambda$-calculus, it suffices to show that convergence or parallel convergence test are definable in $\pi$-calculus. Here are their definitions:

$$[\![\mathtt{C}M]\!] \stackrel{\mathrm{def}}{=} (p).\,\boldsymbol{\nu}q\,(q(x).\,[\![I]\!]p \mid [\![M]\!]q) \qquad q \text{ fresh} \tag{38}$$

$$[\![\mathtt{P}MN]\!] \stackrel{\mathrm{def}}{=} (p).\,(\boldsymbol{\nu}q,r,a\,)(q(x).\,\overline{a} \mid r(x).\,\overline{a} \mid [\![M]\!]q \mid [\![N]\!]r \mid a.\,[\![I]\!]p) \quad q,r,a \text{ fresh} \tag{39}$$

**Exercise 11.9** *1. Show that $[\![\mathtt{C}\lambda x.\,M]\!] \approx [\![I]\!]$, whereas $[\![\mathtt{C}\Omega]\!] \gtrsim [\![\Omega]\!]$.*

*2. Show that, for all $M,N \in \lambda\mathtt{P}$:*

*(a) $[\![\mathtt{P}(\lambda x.\,M)N]\!] \approx [\![I]\!]$, and $[\![\mathtt{P}N(\lambda x.\,M)]\!] \approx [\![I]\!]$;*

*(b) $[\![\mathtt{P}\Omega\Omega)]\!] \approx [\![\Omega]\!]$.*

**Corollary 11.10 (non-completeness of the $\pi$-interpretation)** *$M \approx_\lambda N$ does not imply $M =_\pi N$.*

**Proof:** Take the terms $M$ and $N$ in (37). These terms are applicative bisimilar (see Remark 13.9). However, $M \neq_\pi N$ (Exercise 11.11). ■

**Exercise 11.11** *Prove that $[\![M]\!] \not\approx [\![N]\!]$, for $M,N$ as in (37). (Hint: prove that $[\![\mathtt{C}M]\!]p \not\approx [\![\mathtt{C}N]\!]p$, and for this use Exercise 11.9.)*

We may now ask whether the addition of parallel convergence test to $\lambda N$ is enough to give the same discriminating power as the $\pi$-calculus. The canonical domain model mentioned earlier is fully abstract for $\lambda\mathtt{P}$, therefore tackling this question is also comparing the model with the $\pi$-calculus interpretation. We shall see in the next section that the answer is negative: the $\pi$-calculus semantics is strictly finer.

**Exercise* 11.12** . *This exercise invites the reader to prove that in the simpler encoding of call-by-name in Table 8, convergence test is not definable if we only use the operators of the asynchronous $\pi$-calculus. (To make the exercise simpler, we work with ground bisimulation instead of barbed congruence.) Let $\mathcal{M}$ be the encoding of Table 8. Prove that there is no context $C$ of the asynchronous $\pi$-calculus and name $q$ such that*

$$\begin{array}{rcl} C[\mathcal{M}[\![\Xi]\!]q] & \approx & \mathcal{M}[\![I]\!]p \\ C[\mathcal{M}[\![\Omega]\!]q] & \approx & \mathcal{M}[\![\Omega]\!]p \end{array}$$

*(Hint: prove, by induction on $n$, that there is no $n \geq 0$ and context $C$ such that*

$$\begin{array}{rcl} C[\mathcal{M}[\![\Xi]\!]q] & (\longrightarrow)^n \stackrel{p(x,r)}{\longrightarrow}\Longrightarrow\approx & \mathcal{M}[\![x]\!]r \\ C[\mathcal{M}[\![\Omega]\!]q] & \approx & \mathcal{M}[\![\Omega]\!]p \end{array}$$

*For this, reason by contradiction. You might need also the following result: for all $\widetilde{p}$, $P, Q$, if $\boldsymbol{\nu}\widetilde{p}\,(P \mid Q) \approx \mathbf{0}$ then also $\boldsymbol{\nu}\widetilde{p}\,P \approx \mathbf{0}$.)*

## 12 Extending the $\lambda$-calculus

We have seen that the $\pi$-interpretation is sound but not fully abstract. We now study how to achieve full abstraction by enriching the $\lambda$-calculus. We have already seen extensions of the $\lambda$-calculus: $\lambda\mathtt{C}$ obtained by adding convergence test, and $\lambda\mathtt{P}$ obtained by adding parallel convergence test. Another interesting extension is $\lambda\mathtt{U}$ obtained by adding the *unconditional choice* (or *internal choice*) operator $\mathtt{U}$ defined by

$$\mathtt{U1}\frac{}{\mathtt{U}MN \longrightarrow M} \qquad \mathtt{U2}\frac{}{\mathtt{U}MN \longrightarrow N}$$

**Exercise 12.1** *Extend the encoding of Table 5 to an encoding of $\lambda\mathtt{U}$.*

In general by an operator we mean a symbol with reduction rules defining its behaviour:

**Definition 12.2** *A* signature $\Sigma$ *is a pair $(\mathcal{O}, r)$ where $\mathcal{O}$ is a set of* operator symbols*, disjoint from the set of $\lambda$-variables, and $r$ is a* rank function *which assigns an arity to each operator symbol.*

**Definition 12.3** *If $\Sigma = (\mathcal{O}, r)$ is a signature then the set $\Lambda\Sigma$ of* $\lambda$-terms extended with operators in $\Sigma$ *is defined by*

$$M := pM_1 \ldots M_{r(p)} \;\Big|\; x \;\Big|\; \lambda x.\, M \;\Big|\; M_1 M_2, \qquad \textit{where } p \in \mathcal{O}. \qquad (40)$$

*An* extended $\lambda$-term *is a member of some $\Lambda\Sigma$. We write $\Lambda\Sigma^0$ for the subterms of $\Lambda\Sigma$ without free variables (the closed terms).*

To define the behaviour of extended $\lambda$-terms, we need operational rules for the operators of the extension. In general, the operational rules of an operator are of two kinds: *evaluation rules* such as $\mathtt{C2}$, $\mathtt{P3}-4$ for evaluating arguments of an operator, and *$\delta$-rules* such as $\mathtt{C1}$, $\mathtt{P1}-2$, $\mathtt{U1}-2$ for manipulating the operator. To give a formal definition of these kinds of rules, we need a metalanguage for talking about terms of a generic extended $\lambda$-calculus. The grammar for the metalanguage is just that of the extended $\lambda$-calculus (Grammar 40) plus metavariables. We have used letters $M$ and $N$ for metavariables so far, so we stick to this convention. We write $T \in \Lambda\Sigma^0(M_1, \ldots, M_n)$ if $T$ is a closed metaterm that may contain metavariables among $M_1, \ldots, M_n$.

**Definition 12.4 (well-formed operators)** *Let $\Sigma = (\mathcal{O}, r)$ be a signature and $p \in \mathcal{O}$.*

*A* $\delta$-rule for $p$ (in $\Sigma$) *is an axiom of the form*

$$\frac{\{M_i \downarrow : \quad i \in I\}}{pM_1 \ldots M_{r(p)} \longrightarrow T} I \subseteq \{1, \ldots n\}$$

*where $T \in \Lambda\Sigma^0(M_1, \ldots, M_{r(p)})$. In this case, we also say that the rule* tests position $i$, *for all $i \in I$.*

*An* evaluation-rule for $p$ (in $\Sigma$) *is an inference rule of the form*

$$\frac{M_i \longrightarrow M_i'}{pM_1 \ldots M_i \ldots M_{r(p)} \longrightarrow pM_1 \ldots M_i' \ldots M_{r(p)}} i \in \{1, \ldots, r(p)\}$$

*We say that the rule* evaluates position $i$.

*Let $R$ be a set of $\delta$-rules and evaluation rules for the operators in $\Sigma$ such that for all $p \in \mathcal{O}$ and $1 \leq i \leq r(p)$, if in $R$ there is a $\delta$-rule for $p$ that tests position $i$ then there is also an evaluation rule for $p$ that evaluates that position. In this case, we say that $(\Sigma, R)$ is a* specification of well-formed operators.

**Definition 12.5** *If $\mathcal{S} \stackrel{def}{=} (\Sigma, R)$ is a specification of well-formed operators, and $M, N \in \Lambda\Sigma$, we write $M \longrightarrow_{\mathcal{S}} N$ if $M \longrightarrow N$ is derivable from the rules in $R$ together with the rules $\beta$ and $\mu$ of $\lambda N$.*

*Further, $\lambda\mathcal{S}$ is the extended $\lambda$-calculus with set of terms $\Lambda\Sigma$ and reduction relation $\longrightarrow_{\mathcal{S}}$.*

*The rules in $R$ together with the rules $\beta$ and $\mu$ are* the operational rules of $\lambda\mathcal{S}$. *Relation $\Longrightarrow_{\mathcal{S}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathcal{S}}$, and $M \Downarrow_{\mathcal{S}} N$ means $M \Longrightarrow_{\mathcal{S}} N \downarrow$.*

When there is no ambiguity, we drop some indices, writing $\Lambda\mathcal{O}$ for $\Lambda\Sigma$, $\longrightarrow_{\mathcal{O}}$ (or even $\longrightarrow$) for $\longrightarrow_{\Sigma}$, and $M \Downarrow_{\mathcal{O}} N$ (or even $M \Downarrow N$) for $M \Downarrow_{\mathcal{S}} N$.

The reader might like to check that the rules describing the operators $\mathtt{C}, \mathtt{P}$, $\mathtt{U}$ in this and in the previous sections, fit the format of Definition 12.4.

**Lemma 12.6** *Let $\mathcal{S} = (\Sigma, R)$ and $M \in \Lambda\Sigma^0$. If $M \longrightarrow_{\mathcal{S}} N$ then $N \in \Lambda\Sigma^0$.*

**Exercise 12.7** *We can define an operator @ that expresses call-by-value application. Its rules use an auxiliary operator @′:*

$$@1\frac{M \longrightarrow M'}{@MN \longrightarrow @M'N} \qquad @2\frac{M \downarrow}{@MN \longrightarrow @'MN}$$

$$@'1\frac{N \longrightarrow N'}{@'MN \longrightarrow @MN'} \qquad @'2\frac{N \downarrow}{@'MN \longrightarrow MN}$$

*For $M \in \Lambda$ let $\underline{M}$ be the term of $\Lambda\{@, @'\}$ obtained by replacing all subterms of $M$ of the form $M_1 M_2$ with $@M_1M_2$. Show that if $M, N \in \Lambda^0$, then $M \longrightarrow_{\mathtt{V}} N$ iff $\underline{M}\ (\longrightarrow_{\{@,@'\}})^3\ \underline{N}$.*

**Definition 12.8** *Let $\mathcal{S} = (\Sigma, R)$ be a specification of well-formed operators. We say that $\mathcal{S}$ is* Church-Rosser *(*CR*) if $\Longrightarrow_{\mathcal{S}}$ has the Church-Rosser property; that is, for all $M, N, L \in \Lambda\Sigma$, if $M \Longrightarrow_{\mathcal{S}} N$ and $M \Longrightarrow_{\mathcal{S}} L$, then there is $M'$ such that $N \Longrightarrow_{\mathcal{S}} M'$ and $L \Longrightarrow_{\mathcal{S}} M'$.*

We sometimes abbreviate the terminology, saying that $\mathcal{O}$ is a set of well-formed operators or, if $\rightarrow_{\mathcal{O}}$ is CR, that $\mathcal{O}$ is CR. Observe that if $\rightarrow_{\mathcal{O}}$ is deterministic then $\mathcal{O}$ is CR.

The sets $\{\mathtt{C}\}$ and $\{\mathtt{P}\}$ are CR. But many operators break the CR property. A simple example is unconditional choice $\mathtt{U}$; for instance, we have both $\mathtt{U}\Omega I \Longrightarrow \Omega$ and $\mathtt{U}\Omega I \Longrightarrow I$, but $I$ and $\Omega$ have no common derivative. An even simpler example of a non-CR operator is $\mathtt{U}_\perp$ defined by the rules

$$\mathtt{U}_\perp 1 \frac{}{\mathtt{U}_\perp M \longrightarrow M} \qquad \mathtt{U}_\perp 2 \frac{}{\mathtt{U}_\perp M \longrightarrow \Omega}$$

(This operator is definable from $\mathtt{U}$ thus: $\mathtt{U}_\perp \stackrel{\text{def}}{=} \mathtt{U}\Omega$).

To generalise the definition of applicative bisimilarity to an extended $\lambda$-calculus, we add a clause for internal activity (clause (2) of Definition 12.9 below). This clause is important if the set of operators is not CR, for detecting the branching structure of terms, as Example 12.10 shows. The clause can be omitted when the added operators are CR since then a derivative of a term is bisimilar to that term (Lemma 12.13).

**Definition 12.9** *Let $\mathcal{S} = (\Sigma, R)$ be a specification of well-formed operators. A symmetric relation $\mathcal{R} \subseteq \Lambda\Sigma^0 \times \Lambda\Sigma^0$ is an* applicative $\mathcal{S}$-bisimulation *if $M \mathrel{\mathcal{R}} N$ implies:*

1. *whenever $M \Longrightarrow_{\mathcal{S}} \lambda x.\, M'$, there is $N'$ such that $N \Longrightarrow_{\mathcal{S}} \lambda x.\, N'$ and $M'\{^L\!/\!_x\} \mathrel{\mathcal{R}} N'\{^L\!/\!_x\}$, for all $L \in \Lambda\Sigma^0$;*
2. *whenever $M \Longrightarrow_{\mathcal{S}} M'$, there is $N'$ such that $N \Longrightarrow_{\mathcal{S}} N'$ and $M' \mathrel{\mathcal{R}} N'$.*

*Two terms $M, N \in \Lambda\Sigma^0 \times \Lambda\Sigma^0$ are* applicative $\mathcal{S}$-bisimilar*, written $M \approx_{\mathcal{S}} N$, if $M \mathrel{\mathcal{R}} N$ holds, for some applicative $\mathcal{S}$-bisimulation $\mathcal{R}$.*

It is easy to see that $\approx_{\mathcal{S}}$ is an equivalence relation. (Indeed it is a congruence, as can be proved using Howe's technique [How96].)

Again, if $\mathcal{S} = \{(\mathcal{O}, r), R\}$ we sometimes drop indices and write $\approx_{\mathcal{O}}$ for $\approx_{\mathcal{S}}$. Further, when $\mathcal{O}$ is a singleton $\{p\}$, we write $\approx_p$ for $\approx_{\mathcal{S}}$. Relation $\approx_{\mathcal{S}}$ is extended to open terms in the usual way, by means of closing substitutions.

**Example 12.10** *It holds that $I \not\approx_{\mathtt{U}} \mathtt{U}I\Omega$, since the latter has the reduction $\mathtt{U}I\Omega \Longrightarrow \Omega$ which the former cannot match. This distinction is sensible, since $I$ always accepts an input whereas $\mathtt{U}I\Omega$ can also refuse it. The terms $I$ and $\mathtt{U}I\Omega$ would be equated without clause (2) of Definition 12.9 .*

**Remark 12.11** The format of the rules that we have used for defining well-formed operators ensures that the operators are well-behaved, in the sense that their behaviour depends only on the semantics — not on the syntax — of their operands. The format captures a large and interesting class of such well-behaved operators, but not all of them. For instance, the format allows the evaluation of the argument of an operator, but does not allow evaluating such an argument in some context. Also, the format does not allow rules with negative premises. The format can be extended in several ways to capture larger classes of well-behaved operators. It is good to realise, however, that the format *cannot be arbitrary*, if we wish the behaviour of operators not to depend on the syntax of their operands. For instance, an operator $p$ with a rule

$$\frac{M \longrightarrow \Omega}{pM \longrightarrow I} \tag{41}$$

is disastrous. The rule looks at the syntax of the derivative to which the operand $M$ reduces to, by demanding that this derivative be syntactically equal to $\Omega$. Using $p$ we can distinguish terms such as $\Omega$ from $\Omega\Omega$, since $p\Omega \Longrightarrow I$ whereas $p\Omega\Omega$ has no reductions. It is quite unnatural to distinguish between $\Omega$ and $\Omega\Omega$, however: both are divergent terms, without any observable behaviour. With rules like (41) it would be difficult to define interesting behavioural equivalences that are congruences. ■

We base our operational study of $\lambda\mathcal{S}$ on $\approx_{\mathcal{S}}$, because it seems a reasonable notion of behavioural equivalence, and because it is more tractable than barbed congruence. We do not know whether, in general, $\approx_{\mathcal{S}}$ coincides with the appropriate notion of barbed congruence, although we know it does in certain cases and we do not know any counterexample.

## 12.1 The discriminating power of extended $\lambda$-calculi

We now compare $=_\pi$ (the local structure of the $\pi$-interpretation) and the relations $\approx_{\mathcal{S}}$. Since they may be defined on different classes of terms, we compare them on the common core of closed pure $\lambda$-terms. We present some of the results without digging into the details of their proofs, which are fairly elaborate.

The first result is that $=_\pi$ is at least as discriminating as any applicative $\mathcal{S}$-bisimilarity. That is, $\lambda$-terms that cannot be distinguished as $\pi$-calculus processes cannot be distinguished by any extension of the $\lambda$-calculus.

**Theorem 12.12** *Let $M, N \in \Lambda$. Then $M =_\pi N$ implies $M \approx_{\mathcal{S}} N$, for any $\mathcal{S}$.*

**Proof:** This theorem is proved [San94] by going through a characterisation of $=_\pi$ similar to that in terms of LTs that we shall study in Section 13. We omit the details. ■

The next question is whether there are sets of well-formed operators for which the converse of Theorem 12.12 is true and, if so, what is a minimal such set. These are interesting problems, for their solution will tell us *what it is necessary to add to the $\lambda$-calculus to make it as discriminating as the $\pi$-calculus.*

We begin by looking at sets of CR operators. We show that CR sets of operators do not give full discriminating power. In the remainder of this section, we write $CR$ for an arbitrary set of CR operators.

**Lemma 12.13** *Let $M \in \Lambda CR$. If $M \Longrightarrow N$ then $M \approx_{CR} N$.*

**Corollary 12.14** *In $\lambda CR$ the following conditional $\eta$-rule holds:*

$$M \Downarrow_{CR} \text{ implies } \lambda y.\,(My) \approx_{CR} M$$

**Proof:** By hypothesis, $M \Longrightarrow \lambda x.\, M'$. By repeated applications of Lemma 12.6, and since $\approx_{CR}$ is a congruence,

$$\lambda y.\,(My) \approx_{CR} \lambda y.\,((\lambda x.\, M')y) \approx_{CR} \lambda y.\,(M'\{y/x\}) = \lambda x.\, M' \approx_{CR} M.$$

■

**Theorem 12.15** *$M \approx_{CR} N$ does not imply $M =_\pi N$.*

**Proof:** Take $M = \lambda x.\,(xx)$ and $N = \lambda x.\,(x\lambda y.\,(xy))$. It holds that $M \neq_\pi N$ (this can be proved directly, but it is even simpler to derive it from the characterisation of $=_\pi$ in terms of Lévy-Longo Trees in Section 13).
However $M \approx_{CR} N$. For this, we have to prove that for each $R \in \Lambda CR$, $RR \approx_{CR} R\lambda y.\,(Ry)$. There are two cases, depending on whether $R$ is convergent or not. If $R$ is convergent, then $RR \approx_{CR} R\lambda y.\,(Ry)$ using Corollary 12.14; if it is not then $RR \approx_{CR} \Omega \approx_{CR} R\lambda y.\,(Ry)$. ■

**Remark 12.16** Since the parallel convergence operator test $\mathtt{P}$ is Church-Rosser, Theorem 12.15 also proves that $=_\pi$ is finer than $\approx_{\mathtt{P}}$. Therefore the local structure of the canonical domain of $\lambda N$ [Abr87, Abr89], which coincides with $\approx_{\mathtt{P}}$, is different from that of the $\pi$-interpretation. ■

Theorem 12.15 shows that we cannot achieve the discriminating power of $\pi$-calculus by a confluent extension to the $\lambda$-calculus. The next theorem shows that, in contrast, this is possible with non-confluent extensions. For this, one of the simplest forms of non-confluent operator one could think of, the unary operator $\mathtt{U}_\perp$ that when applied to some argument either returns this argument or diverges, suffices.

**Theorem 12.17** *Let $M, N \in \Lambda$. Then $M \approx_{\mathtt{U}_\perp} N$ implies $M =_\pi N$.*

This theorem is proved in [San94] using a variant of the Böhm-out technique. This result, together with Theorems 12.12, shows that any non-confluent extension of $\lambda N$ in which $\mathtt{U}_\perp$ is expressible can be encoded in $\pi$-calculus in a way which is fully abstract on pure $\lambda$-terms. (By developing this result further, it should actually be possible to prove that the encoding is fully abstract on all terms of the extended $\lambda$-calculus, provided that the operators of the extension can be faithfully encoded.)

## 12.2 Encoding the operators of the $\lambda$-extensions into $\pi$-calculus

In this section we show how to encode extended $\lambda$-calculi into $\pi$-calculus. This should be mainly thought of as an exercize with the $\pi$-calculus; the non-interested reader may safely skip the section.

We add the following constraint to the Definition 12.4 of well-formed operators: for each operator $p$, if $p$ has a rule that evaluates position $i$, then each of the $\delta$-rules for $p$ either tests position $i$ or the rule has a conclusion $pM_1 \ldots M_n \longrightarrow T$ where $M_i$ does not occur in $T$. This is, pragmatically, a reasonable constraint; it is satisfied by the operators encountered so far, such as C, P, U, and @. Without this constraint the encoding would be very complex (for instance, it is possible to define a $\lambda$-calculus reduction strategy that has rules $\beta$, $\mu$ and $\nu$; as discussed in Section 9 this combination of rules is hard to encode).

Fix one such signature $\Sigma$ and specification of well-formed operators $\mathcal{S} \stackrel{\text{def}}{=} (\Sigma, R)$. To make the encoding of $\lambda\mathcal{S}$ into the $\pi$-calculus easier to read, we assume that $\Sigma$ contains a single operator Q of arity 2, with $\delta$-rules

$$\mathtt{Q1}\frac{N_1 \downarrow \quad N_2 \downarrow}{\mathtt{Q}N_1 N_2 \longrightarrow T_1} \qquad \mathtt{Q2}\frac{N_1 \downarrow \quad N_2 \downarrow}{\mathtt{Q}N_1 N_2 \longrightarrow T_2}$$

(where $T_1$ and $T_2$ are some terms in $\Lambda\mathtt{Q}^0(N_1, N_2)$) and evaluation rules

$$\mathtt{ev1}\frac{N_1 \longrightarrow N_1'}{\mathtt{Q}N_1 N_2 \longrightarrow \mathtt{Q}N_1' N_2} \qquad \mathtt{ev2}\frac{N_2 \longrightarrow N_2'}{\mathtt{Q}N_1 N_2 \longrightarrow \mathtt{Q}N_1 N_2'}$$

for evaluating the two arguments of Q. We wish to encode this extended $\lambda$-calculus $\lambda\mathtt{Q}$ into $\pi$-calculus.

The main problem is to define the encoding of a $\Lambda\mathtt{Q}$-term $\mathtt{Q}M_1M_2$. The encoding of the other constructs (abstraction, application, and variable) is as in Section 6, but with the non-linear encoding of abstraction discussed in Remark 6.10 and used in the encoding of $\lambda N$ in Table 7. (It is necessary to use this encoding as, due to the rules of Q, a function that appears as argument to Q may be used more than once.)

**Remark 12.18** Adding the replication in the definition of abstraction does not affect the local structure of the $\pi$-interpretation (see discussions in Remark 13.23). ■

We present an encoding that is easy to reason about and is flexible—so that if we modify the rules for Q it is easy to modify the encoding (see Exercise 12.20-12.21). We shall consider optimisations of the encoding in Exercises 12.22-12.25. The enterprising reader might like to try the encoding of $\mathtt{Q}M_1M_2$ before reading further.

We define the encoding of a term $\mathtt{Q}M_1M_2 \in \Lambda\mathtt{Q}$ thus:

$$\begin{array}{lcll} [\![\mathtt{Q}M_1M_2]\!] & \stackrel{\text{def}}{=} & (p).\,(\boldsymbol{\nu}\widetilde{h}, \widetilde{d}, \widetilde{e}, a\,)\Big( & !h_1\,[\![M_1]\!] \mid\, !h_2\,[\![M_2]\!] \\ & & \quad\mid & \mathtt{EV}_1\langle h_1, d_1, d_2\rangle \mid \mathtt{EV}_2\langle h_2, e_1, e_2\rangle \\ & & \quad\mid & \mathtt{OP}_1\langle d_1, e_1, a\rangle \mid \mathtt{OP}_2\langle d_2, e_2, a\rangle \\ & & & \overline{a}\langle p\rangle\Big) \end{array} \tag{42}$$

where $\widetilde{h}$ stands for $h_1, h_2$, and similarly for $\widetilde{d}$ and $\widetilde{e}$, and where

$$\mathtt{OP}_1\langle d_1, e_1, a\rangle \stackrel{\text{def}}{=} d_1(x_1).\,e_1(x_2).\,a\,[\![T_1\{x_1, x_2/N_1, N_2\}]\!] \qquad x_1, x_2 \text{ fresh for } T_1$$

$$\mathtt{EV}_1\langle h_1, d_1, d_2\rangle \stackrel{\text{def}}{=} \overline{h_1}(r).\,r(v).\,\boldsymbol{\nu}z\,(\overline{d_1}\langle z\rangle \mid \overline{d_2}\langle z\rangle \mid\, !z(r).\,\overline{r}(w).\,w \rightarrow v)$$

and similarly for $\mathtt{OP}_2\langle d_2, e_2, a\rangle$ and $\mathtt{EV}_2\langle h_2, e_1, e_2\rangle$. The process $w \multimap v$ is a link, as defined in Section B.1.5. We briefly explain these definitions. The process $\mathtt{OP}_1\langle d_1, e_1, a\rangle$ implements the $\delta$-rule $\mathtt{Q1}$. It waits for signals at $d_1$ and $d_2$ from $\mathtt{EV}_1\langle h_1, d_1, d_2\rangle$ and $\mathtt{EV}_2\langle h_2, e_1, e_2\rangle$, indicating that the arguments $M_1$ and $M_2$ have become values (i.e., functions). Then $\mathtt{OP}_1\langle d_1, e_1, a\rangle$ tries to grab the lock at $a$; if it succeeds, then the rule may be completed, and $T_1$ gets evaluated. If the lock cannot be grabbed, $\mathtt{OP}_1\langle d_1, e_1, a\rangle$ is garbage. Only one between $\mathtt{OP}_1\langle d_1, e_1, a\rangle$ and $\mathtt{OP}_2\langle d_2, e_2, a\rangle$ can grab the lock, so only one of the $\delta$-rules can be completed.

$\mathtt{EV}_1\langle h_1, d_1, d_2\rangle$ controls the evaluation of the argument $M_1$ (and $\mathtt{EV}_2\langle h_2, e_1, e_2\rangle$ that of $M_2$). First, $\mathtt{EV}_1\langle h_1, d_1, d_2\rangle$ triggers the evaluation of $M_1$. When $M_1$ signals that it has reduced to a value, $M_1'$ say, then $\mathtt{EV}_1\langle h_1, d_1, d_2\rangle$ informs $\mathtt{OP}_1\langle d_1, e_1, a\rangle$ and $\mathtt{OP}_2\langle d_2, e_2, a\rangle$, using names $d_1$ and $d_2$. In these actions at $d_1$ and $d_2$, a pointer $z$ is passed, which the recipient can use to activate copies of $M_1'$.

**Exercise 12.19** *1. For all $M_1, M_2 \in \Lambda\mathtt{Q}$, it holds that*

$$\mathtt{Q}(\lambda x.\, M_1)(\lambda y.\, M_2) \longrightarrow T_1\{^{\lambda x.\, M_1,\, \lambda y.\, M_2}/_{N_1,\, N_2}\}.$$

*Prove that*

$$\begin{aligned}[\![\mathtt{Q}(\lambda x.\, M_1)(\lambda y.\, M_2)]\!]p \approx\!\longrightarrow\!\approx (\boldsymbol{\nu} z_1, z_2)(\; & [\![T_1\{^{z_1,\, z_2}/_{N_1,\, N_2}\}]\!]p \\ & \mid\, !z_1\, [\![\lambda x.\, M_1]\!] \\ & \mid\, !z_2\, [\![\lambda y.\, M_2]\!])\end{aligned}$$

*2. Show that, for all $M, N \in \Lambda\mathtt{Q}$ with $x \notin \mathrm{fv}(N)$,*

$$\mathcal{N}[\![M]\!]\{x = \mathcal{N}[\![N]\!]\} \approx \mathcal{N}[\![M\{^N/_x\}]\!]\;.$$

*(Hint: add one case in the inductive proof of Exercise 6.20.)*

**Exercise 12.20** *Suppose that rules $\mathtt{Q1}$ and $\mathtt{Q2}$ do not test position 2 (i.e., their premise is simply $N_1 \downarrow$) and there is no evaluation rule $\mathtt{ev2}$. How should the definition of $\mathtt{EV}_2\langle h_2, e_1, e_2\rangle$ in (42) be modified?*

**Exercise 12.21** *Suppose that the $\delta$-rules of $\mathtt{Q}$ are modified so that rule $\mathtt{Q}i$ ($i = 1, 2$) only tests position $i$ (i.e., the premise of $\mathtt{Q}i$ is simply $N_i \downarrow$) and its derivative $T_i$ does not contain metavariable $N_j$ ($j \neq i$). How should the definitions of $\mathtt{EV}_1\langle h_1, d_1, d_2\rangle$ and $\mathtt{EV}_2\langle h_2, e_1, e_2\rangle$ in (42) be modified?*

In some cases, the encodings obtained following the schema above can be optimised, as shown in the exercises below.

**Exercise* 12.22** *1. Prove that, for $[\![\mathtt{Q}M_1M_2]\!]$ as defined in (42), we have:*

$$\begin{aligned}[\![\mathtt{Q}M_1M_2]\!]p \approx (\boldsymbol{\nu} d, e, q_1, q_2)\Big(\; & [\![M_1]\!]q_1 \mid [\![M_2]\!]q_2 \\ & \mid q_1(v).\, \overline{d}(z).\, !z(r).\, \overline{r}(w).\, w \multimap v \\ & \mid q_2(v).\, \overline{e}(z).\, !z(r).\, \overline{r}(w).\, w \multimap v \\ & \mid d(x_1).\, e(x_2).\, (\;\tau.\, [\![T_1\{^{x_1,\, x_2}/_{N_1,\, N_2}\}]\!]p \\ & \qquad\qquad + \tau.\, [\![T_2\{^{x_1,\, x_2}/_{N_1,\, N_2}\}]\!]p)\Big)\end{aligned}$$

*2. Exhibit two terms $M_1$ and $M_2$ such that*

$$\begin{array}{l} [\![ \mathtt{Q} M_1 M_2 ]\!] p \not\approx (\boldsymbol{\nu} d, e, q_1, q_2) \Big( \; [\![ M_1 ]\!] q_1 \mid [\![ M_2 ]\!] q_2 \\ \qquad\qquad\qquad\qquad\qquad \mid \; q_1(v_1).\, q_2(v_2). \\ \qquad\qquad\qquad\qquad\qquad\qquad (\boldsymbol{\nu} z_1, z_2) \; \big( !z_1(r).\, \overline{r}(w).\, w \multimap v_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid !z_2(r).\, \overline{r}(w).\, w \multimap v_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid ( \;\; \tau.\, [\![ T_1\{^{x_1, x_2}/_{N_1, N_2}\} ]\!] p \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad + \tau.\, [\![ T_2\{^{x_1, x_2}/_{N_1, N_2}\} ]\!] p ))\Big) \end{array}$$

**Exercise 12.23** *Write down an encoding $\mathcal{W}$ of $\lambda\mathtt{P}$ (operator $\mathtt{P}$ is parallel convergence test, defined in Section 11.2) following the schema for encoding well-formed operators indicated above*

*Prove that if $\mathcal{A}$ is the encoding defined as $\mathcal{W}$ but with clause (39) (in place of (42)), then for all $M \in \Lambda\mathtt{P}$, it holds that $\mathcal{W}[\![M]\!] \approx \mathcal{A}[\![M]\!]$.*

**Exercise 12.24** *Repeat Exercise 12.23 for $\lambda\mathtt{U}$, where the encoding $\mathcal{A}$ uses the clause for $\mathtt{U}$ from Exercise 12.1.*

**Exercise* 12.25** *Let @ and @′ be the operators, and $\underline{M}$ the transformation of a $\lambda$-term $M$ defined in Exercize 12.7. Define an encoding $\mathcal{W}$ of $\lambda\{@, @'\}$ following the schema above for translating well-formed operators. Prove that, if $\mathcal{U}^*_{\mathrm{V}}$ is the encoding of $\lambda V$ in Exercise 7.3 (this encoding is a simple variant of that in Table 7), then for all $M \in \Lambda$ it holds that $\mathcal{W}[\![\underline{M}]\!] \approx \mathcal{U}^*_{\mathrm{V}}[\![M]\!]$. (Note: In this exercise we are using $\mathcal{U}^*_{\mathrm{V}}$, which is a variant of the encoding in Table 7, because in this way the proof does not require $\mathsf{i/o}$ types and therefore can be carried out with the standard theory of the $\pi$-calculus (see also Remark 7.2).)*

## 13 The local structure of the $\pi$-interpretation

The local structure of the $\pi$-interpretation, $=_\pi$, is the behavioural equivalence induced on $\lambda$-terms by their encoding into $\pi$-calculus. In the previous section we proved a characterisation of $=_\pi$ as the operational equivalence of extended $\lambda$-calculi.

We continue the study of $=_\pi$ in this section, with the purpose of understanding the meaning of function equality when functions are interpreted as processes. We shall prove characterisations of $=_\pi$ in terms of tree structures which are an important part of the theory of the $\lambda$-calculus. This will show that the equivalence induced by the $\pi$-calculus encoding is a natural one. It will also show the utility of some of the $\pi$-calculus proof techniques: using techniques such as 'bisimulation up to context' and 'bisimulation up to expansion', the proofs of the main theorems will be much easier than they would have been otherwise.

The main result says that $=_\pi$ coincides with *LT-equality*, whereby two $\lambda$-terms are equal iff they have the same *Lévy-Longo Trees* (*LT*s). Lévy-Longo Trees are the lazy variant of *Böhm Trees* (*BT*s). We shall also discuss modifications of the $\pi$-calculus interpretation so that its local structure is the analogous *BT-equality*.

We begin by recalling what LTs are. A reader familiar with them may go straight to Theorem 13.8.

## 13.1 Sensible theories and lazy theories

Böhm Trees (BT) are the most well-known tree structure in the $\lambda$-calculus. BTs play a central role in the classical theory of the $\lambda$-calculus. The local structure of some of the most influential models of the $\lambda$-calculus, like Scott and Plotkin's $P\omega$, Plotkin's $T^{\omega}$, Plotkin and Engeler's $D_A$ is precisely the BT equality; and the local structure of Scott's $D_{\infty}$ (historically the first mathematical, i.e., non-syntactical, model of the untyped $\lambda$-calculus) is the equality of the "infinite $\eta$ contraction" of BTs.

BTs naturally give rise to a tree topology that has been used for the proof of some seminal results of the $\lambda$-calculus like Berry's sequentiality theorem (briefly discussed in Section 2.2). BTs were introduced by Barendregt [Bar77], and called so after Böhm's proof and theorem about separability of $\lambda$-terms. The proof technique for this theorem, called the *Böhm-out technique*, roughly consists in finding a context capable of isolating a given subtree of a BT; in this way, certain $\lambda$-terms that have different BTs may be separated.

BTs are at the heart of the classical theory of the $\lambda$-calculus, sometimes referred to as the *sensible theory*. In this theory, a $\lambda$-term is meaningful just if it *solvable*, that is it has a *head normal form* (*hnf*). The *unsolvable* terms, that is the terms without hnf, are identified as the "meaningless terms". In the mathematical models for this theory, the unsolvable terms are those terms whose image is the least element of the model (the undefined or bottom element). The BT of a term conveys the essential behavioural content of that term under this proposal.

A hnf is a term of the form $\lambda \widetilde{x}.\, y\widetilde{M}$. Examples of terms without a hnf are $\Omega$, $\lambda x.\,\Omega$, and $\Xi$ (recall that $\Xi$ satisfies the equation $\lambda\beta \vdash \Xi = (\lambda x.\,)^n\Xi$, for all $n$). Finding the hnf of a term requires computing underneath $\lambda$, in order to uncover the head variable after a sequence of $\lambda$'s. Computing underneath $\lambda$ is a debatable decision; for instance it does not reflect the practice of programming language implementations. An alternative proposal for identifying the meaningful terms does not require computing under $\lambda$: it uses *weak head normal forms* (*whnf*'s) in place of hnf's. A whnf is a term of the form $\lambda x.\, M$ or $x\widetilde{M}$. This second proposal forms the basis of the *lazy* theory. Its tree structures under this proposal are the Lévy-Longo Trees. There are interesting mathematical models of the $\lambda$-calculus whose local structure is precisely the LT-equality, see [Ong88].

As an example, the terms $\lambda x.\,\Omega$ and $\Omega$ are distinguished in the lazy theory because only the former has a whnf; but they are identified as meaningless in the sensible theory because neither has an hnf. Similarly, $\Xi$ and $\lambda x.\,\Omega$, are meaningless in a sensible theory, but are meaningful and distinguished in a lazy theory (they are distinguished by feeding an argument since, for all $N$, $\Xi N$ has a whnf, whereas $(\lambda x.\,\Omega)N$ has not).

## 13.2 Lévy-Longo Trees and the local structure theorem

To define LTs we need the notions of the *proper order* of a term, and *head reduction*, which we now introduce. We use $n$ to range over the set of nonnegative integers and $\omega$ to represent the first ordinal limit.

The *order* of a term $M$ expresses the maximum length of the outermost sequence of $\lambda$-abstractions in a term to which $M$ is $\beta$-convertible; it says how "higher-order" $M$ is. More precisely, $M$ has order $n$ if $n$ is the largest $i$ such that

$\lambda\beta \vdash M = \lambda x_1 \cdots x_i.\, N$, for some $N$. Therefore a term has order 0 if it is not $\beta$-convertible to any abstraction. The remaining terms are assigned order $\omega$; they are terms such as $\Xi$ which can reduce to an unbounded number of nested abstractions. A term has *proper order n* if it has order of unsolvability $n$, i.e., after the initial $n$ $\lambda$-abstractions it behaves like $\Omega$. Formally,

- $M$ has *proper order* 0, written $M \in PO_0$, if $M$ has order 0 and there is no $\widetilde{N}$ such that $\lambda\beta \vdash M = x\widetilde{N}$;
- $M$ has *proper order* $n+1$, written $M \in PO_{n+1}$, if $\lambda\beta \vdash M = \lambda x.\, N$ for some $N \in PO_n$;
- $M$ has *proper order* $\omega$, written $M \in PO_\omega$, if $M$ has order $\omega$.

A $\lambda$-term is either of the form $\lambda\widetilde{x}.\, y\widetilde{M}$, or of the form $\lambda\widetilde{x}.\,(\lambda x.\, M_0)M_1 \cdots M_n$, $n \geq 1$. In the latter, the redex $(\lambda x.\, M_0)M_1$ is called the *head redex*. If $M$ has a head redex, then $M \longrightarrow_{\mathtt{h}} N$ holds if $N$ results from $M$ by $\beta$-reducing its head redex; $\Longrightarrow_{\mathtt{h}}$ is the reflexive and transitive closure of $\longrightarrow_{\mathtt{h}}$. *Head reduction* $\Longrightarrow_{\mathtt{h}}$ is different from call-by-name reduction $\Longrightarrow$: a call-by-name redex is also a head redex, but the converse is false as a head redex can also be located underneath an abstraction. We have, however,

**Lemma 13.1**

1. *$M \Longrightarrow_{\mathtt{h}} \lambda x.\, N$ iff $M \Longrightarrow \lambda x.\, N'$, for some $N'$ such that $N' \Longrightarrow_{\mathtt{h}} N$.*
2. *$M \Longrightarrow_{\mathtt{h}} x\widetilde{N}$ iff $M \Longrightarrow x\widetilde{N}$.*
3. *$M \Longrightarrow_{\mathtt{h}} \lambda x_1. \cdots x_n.\, y\widetilde{N}$ iff there are terms $M_i$, $1 \leq i \leq n$, such that $M \Longrightarrow \lambda x_1.\, M_1$, $M_i \Longrightarrow \lambda x_{i+1}.\, M_{i+1}$, $1 \leq i < n$, and $M_n \Longrightarrow y\widetilde{N}$.*

**Proof:** (1) and (2) hold because both $\Longrightarrow$ and $\Longrightarrow_{\mathtt{h}}$ progress using the leftmost redex; (3) follows from (1) and (2). ■

The definition below of LT is simple but informal. A precise definition would require formalising the notion of labeled tree.

**Definition 13.2 (Lévy-Longo Trees)** *The* Lévy-Longo Tree *of $M$ is the labelled tree, LT(M), defined inductively as follows:*

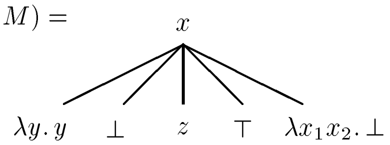
*1) $LT(M) = \top$ if $M \in PO_\omega$,*
*2) $LT(M) = \lambda x_1 \cdots x_n.\, \bot$ if $M \in PO_n$, $0 \leq n < \omega$,*
*3)*

$$LT(M) = \begin{array}{c} \lambda\widetilde{x}.\, y \\ \swarrow \qquad \searrow \\ LT(M_1) \qquad \cdots \qquad LT(M_n) \end{array}$$

*if $M \Longrightarrow_{\mathtt{h}} \lambda\widetilde{x}.\, yM_1 \cdots M_n$, $n \geq 0$.*

**Example 13.3** *Let $M = x(\lambda y.\, y)\Omega z \Xi(\lambda x_1 x_2.\, \Omega)$. Then*

$LT(M) =$




We identify $\alpha$-convertible LT's. LT-equality can also be presented as a form of bisimilarity. This bisimilarity, *open applicative bisimilarity*, is a refinement of applicative bisimilarity; indeed it is perhaps the simplest way to extend applicative bisimilarity to open terms.

**Definition 13.4** *A symmetric relation $\mathcal{R} \subseteq \Lambda \times \Lambda$ is an* open applicative bisimulation *if $M \mathrel{\mathcal{R}} N$ implies:*

1. *if $M \Longrightarrow \lambda x.\, M'$, then there exists $N'$ such that $N \Longrightarrow \lambda x.\, N'$ and $M' \mathrel{\mathcal{R}} N'$;*

2. *if $M \Longrightarrow x M_1 \dots M_n$, for some $n \geq 0$, then then there exist $N_1, \dots, N_n$ such that $N \Longrightarrow x N_1 \dots N_n$ and $M_i \mathrel{\mathcal{R}} N_i$, for all $1 \leq i \leq n$.*

*Two terms $M, N \in \Lambda$ are* open applicative bisimilar*, written $M \approx_\lambda^{\mathtt{open}} N$, if $M \mathrel{\mathcal{R}} N$, for some open applicative bisimulation $\mathcal{R}$.*

Clause (2), concerning terms with a variable in head position, was absent from the definition of applicative bisimilarity, where all terms are closed. Moreover, in contrast with applicative bisimilarity, in clause (1) no term instantiation on $\lambda$-abstractions is required. This simplification is possible because we work on open terms and is justified by the congruence of $\approx_\lambda^{\mathtt{open}}$. (A straightforward proof that $\approx_\lambda^{\mathtt{open}}$ is a congruence utilises the full abstraction Theorems 13.20 and 13.21 and the congruence of $\pi$-calculus bisimilarity $\approx$.) Two useful facts are:

**Lemma 13.5** *If $M \Longrightarrow N$, then $M \approx_\lambda^{\mathtt{open}} N$.*

**Lemma 13.6** *Let $\rho$ be a substitution from $\lambda$-variables to $\lambda$-variables. If $M \approx_\lambda^{\mathtt{open}} N$, then $M\rho \approx_\lambda^{\mathtt{open}} N\rho$.*

**Theorem 13.7** *$M \approx_\lambda^{\mathtt{open}} N$ iff $LT(M) = LT(N)$.*

**Proof:** Using Lemma 13.1 we can show that $\approx_\lambda^{\mathtt{open}}$ also coincides with the largest relation $\mathcal{R}$ on $\Lambda \times \Lambda$ such that $M \mathrel{\mathcal{R}} N$ implies:

1. $M \in PO_\omega$ iff $N \in PO_\omega$,

2. $M \in PO_n$ iff $N \in PO_n$,

3. If $x_i \notin \mathrm{fv}(M, N)$, $1 \leq i \leq n$, then $M \Longrightarrow_{\mathtt{h}} \lambda x_1 \dots x_n.\, y M_1 \dots M_m$ iff $N \Longrightarrow_{\mathtt{h}} \lambda x_1 \dots x_n.\, y N_1 \dots N_m$ and $M_i \mathrel{\mathcal{R}} N_i$, $1 \leq i \leq m$.

Finally, it is immediate to see that $\mathcal{R}$ is the LT equality. ∎

Theorem 13.7 is useful for relating LT-equality to other behavioural equivalences on $\lambda$-terms. We shall use it to prove that LT-equality is the same as equality in the $\pi$-interpretation. We state the result and then briefly discuss it, deferring the proof to Section 13.3.

**Theorem 13.8 (local structure of the $\pi$-interpretation)** *$M =_\pi N$ iff $LT(M) = LT(N)$.*

**Remark 13.9** Consider the terms

$$M \stackrel{\text{def}}{=} \lambda x.\,(x(\lambda y.\,(x\Xi\Omega y))\Xi) \qquad\qquad N \stackrel{\text{def}}{=} \lambda x.\,(x(x\Xi\Omega)\Xi).$$

These terms are applicative bisimilar; this can be proved by a case analysis on the order of the $\lambda$-term that is given as an input to the two terms. The two terms can be distinguished using the convergence test $\mathtt{C}$, for $M(\lambda x.\,\mathtt{C}x)$ reduces to an abstraction, whereas $N(\lambda x.\,\mathtt{C}x)$ diverges. Since $M$ and $N$ are distinguished using the operator $\mathtt{C}$, by Theorem 12.12 they are distinguished by $=_\pi$.

Using Theorem 13.8, we can prove $M \neq_\pi N$ simply by observing that their LT's are different:

$$LT(M) = \begin{array}{ccc} & \lambda x.\,x & \\ \lambda y.\,x & & \top \\ \top \;\; \bot \;\; y & & \end{array} \qquad\qquad LT(N) = \begin{array}{ccc} & \lambda x.\,x & \\ x & & \top \\ \top \;\; \bot & & \end{array}$$

■

## 13.3 The proof of the local structure theorem

In order to prove Theorem 13.8, we first need to establish the operational correspondence on weak transitions between functions and their process encodings (Propositions 13.14 and 13.16 below).

The encoding of a $\lambda$-term is an abstraction, that is a function from names to processes; abstractions are ranged over by $F, G$. We introduce a process notation which allows us to give a simpler description of encodings of $\lambda$-terms with a variable in head position (see Lemma 13.12).

**Definition 13.10** *For $n > 0$ we define:*

$$\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle \stackrel{\mathit{def}}{=}$$
$$\boldsymbol{\nu} r_1, \ldots, r_{n-1}, x_1, \ldots, x_n \Big( \; r_0(v_1).\,\overline{v}_1\langle x_1, r_1\rangle \mid \ldots \mid r_{n-1}(v_n).\,\overline{v}_n\langle x_n, r_n\rangle \mid\, !x_1\, F_1 \mid \ldots \mid\, !x_n\, F_n \Big)$$

*where names $r_1, \ldots, r_{n-1}, x_1 \ldots, x_n$, and $q$ are fresh.*

The $i$-th process $r_{i-1}(v_i).\,\overline{v}_i\langle x_i, r_i\rangle$ of $\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle$ liberates the agent $!x_i\, F_i$ and the $(i+1)^{\text{th}}$ process.

**Lemma 13.11** *Suppose $n > 1$.*

1. *$\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle \sim$ $(\boldsymbol{\nu} r_{n-1}, x_n)\Big(\mathcal{O}_{n-1}\langle r_0, r_{n-1}, F_1, \ldots, F_{n-1}\rangle \mid r_{n-1}(v_n).\,\overline{v}_n\langle x_n, r_n\rangle \mid\, !x_n\, F_n\Big)$.*

2. *If $\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle \xrightarrow{\mu_1}\xrightarrow{\mu_2} P$, then $\mu_1 = r_0(v_1)$, $\mu_2 = (\boldsymbol{\nu} x_1, r_1)\overline{v}_1\langle x_1, r_1\rangle$ and $P \sim \mathcal{O}_{n-1}\langle r_1, r_n, F_2, \ldots, F_n\rangle \mid\, !x_1\, F_1$.*

**Lemma 13.12** *If $n > 0$, then*

$$[\![xM_1 \dots M_n]\!]r_n \sim \boldsymbol{\nu} r_0\ (\overline{x}\langle r_0\rangle \mid \mathcal{O}_n\langle r_0, r_n, [\![M_1]\!], \dots, [\![M_n]\!]\rangle).$$

**Proof:** By induction on $n$. For $n = 1$,

$$\begin{aligned}
[\![xM_1]\!]r_1 &= \\
\boldsymbol{\nu} r_0 \left([\![x]\!]r_0 \mid \boldsymbol{\nu} x_1\ (r_0(v_1).\, \overline{v}_1\langle x_1, r_1\rangle \mid !x_1\ [\![M_1]\!])\right) &\sim \\
\boldsymbol{\nu} r_0\ (\overline{x}\langle r_0\rangle \mid \mathcal{O}_1\langle r_0, r_1, [\![M_1]\!]\rangle)\,.
\end{aligned}$$

For $n > 1$, we have, abbreviating $r_{n-1}(v_n).\, \overline{v}_n\langle x_n, r_n\rangle \mid !x_n\ [\![M_n]\!]$ to $P$:

$$\begin{aligned}
[\![xM_1 \dots M_n]\!]r_n &\sim \\
(\boldsymbol{\nu} r_{n-1}, x_n\ )\Big([\![xM_1 \dots M_{n-1}]\!]r_{n-1} \mid P\Big) &\sim \qquad (43)\\
(\boldsymbol{\nu} r_{n-1}, x_n\ )\Big(\boldsymbol{\nu} r_0\ (\overline{x}\langle r_0\rangle \mid \mathcal{O}_{n-1}\langle r_0, r_{n-1}, M_1, \dots, M_{n-1}\rangle) \mid P\Big) &\sim \\
\boldsymbol{\nu} r_0 \Big(\overline{x}\langle r_0\rangle \mid (\boldsymbol{\nu} r_{n-1}, x_n\ )\Big(\mathcal{O}_{n-1}\langle r_0, r_{n-1}, M_1, \dots, M_{n-1}\rangle \mid P\Big) &\sim \qquad (44)\\
\boldsymbol{\nu} r_0 \Big(\overline{x}\langle r_0\rangle \mid \mathcal{O}_n\langle r_0, r_n, M_1, \dots, M_n\rangle\Big) & \qquad\quad (45)
\end{aligned}$$

where (43) uses induction and (44) uses Lemma 13.11(1). ■

We recall the $\lesssim$ is the expansion relation, Definition B.6.

**Lemma 13.13**

1. *If $M \longrightarrow N$, then $[\![M]\!]p \gtrsim [\![N]\!]p$.*
2. *If $M = \lambda x.\, N$, then $[\![M]\!]p \xrightarrow{\overline{p}(v)} \xrightarrow{v(x,q)} [\![N]\!]q$.*
3. *If $M = x$, then $[\![M]\!]p \xrightarrow{\overline{x}\langle p\rangle} \mathbf{0}$.*
4. *If $M = xM_1 \dots M_n$ where $n > 0$, then $[\![M]\!]p \xrightarrow{\overline{x}(q)}\sim \mathcal{O}_n\langle q, p, [\![M_1]\!], \dots, [\![M_n]\!]\rangle$.*

**Proof:** (2) and (3) are immediate from the definition of the encoding. (4) follows from Lemma 13.12(2). (1) is an immediate consequence of Exercise 6.21. ■

**Proposition 13.14 (operational correspondence on the reductions of $M$)**

1. *If $M \Longrightarrow N$, then $[\![M]\!]p \gtrsim [\![N]\!]p$.*
2. *If $M \Longrightarrow \lambda x.\, N$, then $[\![M]\!]p \overset{\overline{p}(v)}{\Longrightarrow} \gtrsim v(x)\ [\![N]\!]$.*
3. *If $M \Longrightarrow x$, then $[\![M]\!]p \overset{\overline{x}\langle q\rangle}{\Longrightarrow} \gtrsim \mathbf{0}$.*
4. *If $M \Longrightarrow xM_1 \dots M_n$ where $n > 0$, then $[\![M]\!]p \overset{\overline{x}(q)}{\Longrightarrow}\gtrsim \mathcal{O}_n\langle q, p, [\![M]\!]_1, \dots, [\![M_n]\!]\rangle$.*

**Proof:** (1) is by induction on the number of reductions and Lemma 13.13(1). The other assertions are consequences of (1) and Lemma 13.13(2-4). ■

**Lemma 13.15** *Suppose* $[\![M]\!]p \xrightarrow{\mu} P$.

1. *If* $\mu = \tau$, *then there is* $N$ *such that* $M \longrightarrow N$ *and* $P \gtrsim [\![N]\!]p$.
2. *If* $\mu$ *is an output at* $p$, *then* $\mu = \overline{p}(v)$ *and there are* $x, N$ *such that* $M = \lambda x.\, N$ *and* $P = v(x)\, [\![N]\!]$.
3. *If* $\mu$ *is a free output, then there is* $x$ *such that* $\mu = \overline{x}\langle p\rangle$ *and* $P = \mathbf{0}$.
4. *otherwise, there are* $x$ *and* $M_1, \dots, M_n$, $n > 0$, *such that* $\mu = \overline{x}(q)$, $M = xM_1 \dots M_n$ *and* $P \sim \mathcal{O}_n\langle q, p, [\![M]\!]_1, \dots, [\![M_n]\!]\rangle$.

**Proof:** For each $M$ and $p$, $[\![M]\!]p$ has only one possible transition. The assertions follow from Lemma 13.13. ■

**Proposition 13.16 (operational correspondence on transitions of $[\![M]\!]p$)** *Suppose* $[\![M]\!]p \stackrel{\mu}{\Longrightarrow} P$:

1. *If* $\mu = \tau$, *then there is* $N$ *such that* $M \Longrightarrow N$ *and* $P \gtrsim [\![N]\!]p$.
2. *If* $\mu$ *is an output at* $p$, *then* $\mu = \overline{p}(v)$ *and there are* $x, N$ *such that* $M \Longrightarrow \lambda x.\, N$ *and* $P \gtrsim v(x)\, [\![N]\!]$.
3. *If* $\mu$ *is a free output, then there is* $x$ *such that* $\mu = \overline{x}\langle p\rangle$ *and* $P \gtrsim \mathbf{0}$.
4. *otherwise, there are* $x$ *and* $M_1, \dots, M_n$, $n > 0$, *such that* $\mu = \overline{x}(q)$, $M \Longrightarrow xM_1 \dots M_n$, *and* $P \gtrsim \mathcal{O}_n\langle q, p, [\![M]\!]_1, \dots, [\![M_n]\!]\rangle$.

**Proof:** By induction on the length of the transition of $[\![M]\!]p$. For the base case, note that for each $M$ and $p$, process $[\![M]\!]p$ has only one possible transition. The assertions follow by Lemma 13.13. ■

We need a few lemmas before tackling the full abstraction theorems. Lemma 13.17 shows a decomposition property for ground bisimilarity; Lemmas 13.18 and 13.19 show properties of the processes $\mathcal{O}_n\langle r_o, r_n, F_1, \dots, F_n\rangle$, introduced in Definition 13.10 to represent the encoding of $\lambda$-terms with a variable in head position.

For a process $P$, we let $\mathcal{N}_P$ be the set of names along which $P$ can perform an action, i.e.,

$$\mathcal{N}_P = \{a \ : \ \text{for some and } P' \text{ and } \mu \text{ with subject } a,\ P \stackrel{\mu}{\Longrightarrow} P'\}\,.$$

**Lemma 13.17**

1. *Suppose* $\mathrm{fn}(P_1, P_2) \cap (\mathcal{N}_{Q_1} \cup \mathcal{N}_{Q_2}) = \emptyset$. *Then* $P_1 \mid Q_1 \approx P_2 \mid Q_2$ *implies* $P_1 \approx P_2$.
2. *Suppose* $x, q \notin \mathrm{fn}(F, G)$. *Then* $!x\, F \approx\, !x\, G$ *implies* $Fq \approx Gq$.

**Proof:** We first prove (1). The relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{(P_1, P_2) \; : \; P_1 \mid Q_1 \approx P_2 \mid Q_2 \quad \text{for some } Q_1, Q_2 \text{ with } \mathrm{fn}(P_1, P_2) \cap (\mathcal{N}_{Q_1} \cup \mathcal{N}_{Q_2}) = \emptyset \; \}$$

is a ground bisimulation. The proof is straightforward, for if $\mathrm{fn}(P_1, P_2) \cap (\mathcal{N}_{Q_1} \cup \mathcal{N}_{Q_1}) = \emptyset$, no interaction between $P_i$ and $Q_i$ is possible, $i = 1, 2$. Moreover, if all bound names of actions of $P_1$ and $P_2$ are fresh, then the side condition of $\mathcal{R}$ is preserved.
Now assertion (2). We have to show that $Fq \approx Gq$. We can assume, without loss of generality, that $q \neq x$. Since $!x\,F \approx\, !x\,G$ and $!x\,F \xrightarrow{x(q)} Fq \mid !x\,F$, we have, for some $P$,

$$!x\,G \overset{x(q)}{\Longrightarrow} P \approx Fq \mid !x\,F \,. \tag{46}$$

Since $x$ does not occur in $Gq$, no interaction between $Gq$ and $!x\,G$ may have occurred; therefore $P$ is of the form $P_G \mid !x\,G$, for some $P_G$ such that

$$Gq \longrightarrow \; \Longrightarrow P_G. \tag{47}$$

Thus (46) can be written $P_G \mid !x\,G \approx Fq \mid !x\,F$. From this we get

$$P_G \approx Fq \tag{48}$$

using the assertion (1) of the lemma, since $\mathcal{N}_{!x\,G} = \mathcal{N}_{!x\,F} = \{x\}$ and $x$ is not free in $Fq$ and $P_G$. Similarly, (exchanging $F$ and $G$), we can derive, for some $P_F$,

$$Fq \longrightarrow \; \Longrightarrow P_F \,, \quad \text{and} \quad P_F \approx Gq \,. \tag{49}$$

Now, we exploit (47-49) to show that $Fq \approx Gq$: For this, we take

$$\mathcal{R} \stackrel{\text{def}}{=} \{(Fq, Gq), (Gq, Fq)\} \cup \approx$$

and show that it is a ground bisimulation. Suppose $Fq \xrightarrow{\mu}\longrightarrow Q_F$. We show how $Gq$ can match this move: Since $Fq \approx P_G$ we have $P_G \overset{\widehat{\mu}}{\Longrightarrow} Q_G \approx Q_F$. Therefore, using (47), $Gq \longrightarrow \; \Longrightarrow P_G \overset{\widehat{\mu}}{\Longrightarrow} Q_G \approx Q_F$, which closes the bisimulation. Similarly $Fq$ can match an action of $Gq$. ■

**Lemma 13.18** *If* $\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle \approx \mathcal{O}_m\langle r_0, r_m, G_1, \ldots, G_m\rangle$ *then* $n = m$.

**Lemma 13.19** $\mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle \approx \mathcal{O}_n\langle r_0, r_n, G_1, \ldots, G_n\rangle$ *iff* $(F_i)q \approx (G_i)q$ *for all* $1 \leq i \leq n$ *and for some fresh* $q$.

**Proof:** The implication from right to left follows from congruence properties of $\approx$. For the converse we proceed by induction on $n$. We only consider the inductive case. Let

$$P \stackrel{\text{def}}{=} \mathcal{O}_n\langle r_0, r_n, F_1, \ldots, F_n\rangle, \qquad Q \stackrel{\text{def}}{=} \mathcal{O}_n\langle r_0, r_n, G_1, \ldots, G_n\rangle.$$

By Lemma 13.11(2), their first two transitions are

$$P \xrightarrow{r_0(v_1)} \xrightarrow{(\boldsymbol{\nu} x_1, r_1)\overline{v}_1\langle x_1, r_1\rangle} \sim \mathcal{O}_{n-1}\langle r_1, r_n, F_2, \ldots, F_n\rangle \mid !x_1\,F_1,$$

$$Q \xrightarrow{r_0(v_1)} \xrightarrow{(\boldsymbol{\nu} x_1, r_1)\overline{v}_1\langle x_1, r_1\rangle} \sim \mathcal{O}_{n-1}\langle r_1, r_n, G_2, \ldots, G_n\rangle \mid !x_1\,G_1.$$

From this, we can derive $P \approx Q$ implies $\mathcal{O}_{n-1}\langle r_1, r_n, F_2, \ldots, F_n\rangle \mid !x_1\, F_1 \approx \mathcal{O}_{n-1}\langle r_1, r_n, G_2, \ldots, G_n\rangle \mid !x_1\, G_1$. Let $P_1 \stackrel{\text{def}}{=} \mathcal{O}_{n-1}\langle r_1, r_n, F_2, \ldots, F_n\rangle$ and $Q_1 \stackrel{\text{def}}{=} \mathcal{O}_{n-1}\langle r_1, r_n, G_2, \ldots, G_n\rangle$. The only actions that $!x_1\, F_1$ and $!x_1\, G_1$ can perform are at $x_1$ and, by Lemma 13.11(2), the only actions that $P_1$ and $Q_1$ can perform are at $r_1$. Since $r_1$ is not free in $!x_1\, F_1$ and $!x_1\, G_1$, and $x_1$ is not free in $P_1$ and $Q_1$, using Lemma 13.17(1) twice we infer

$$!x_1\, F_1 \approx !x_1\, G_1 \text{ and } P_1 \approx Q_1$$

From the first by Lemma 13.17(2) we get $(F_1)q \approx (G_1)q$. From the second by the induction hypothesis we get $(F_i)q \approx (G_i)q$, for $2 \le i \le n$. ■

We now ready to prove that open applicative bisimilarity is the same as the local structure of the $\pi$-interpretation.

**Theorem 13.20 (soundness w.r.t. $\approx_\lambda^{\mathsf{open}}$)** *$M =_\pi N$ implies $M \approx_\lambda^{\mathsf{open}} N$.*

**Proof:** By Definition 11.5 and Corollary 6.23, we have to prove that $[\![M]\!] \approx [\![N]\!]$ implies $M \approx_\lambda^{\mathsf{open}} N$. We prove that

$$\mathcal{R} \stackrel{\text{def}}{=} \{(M, N) \ : \ [\![M]\!] \approx [\![N]\!],\ \}$$

is an open applicative bisimulation. First, suppose $M \Longrightarrow \lambda x.\, M'$. We have to find $N'$ such that $N \Longrightarrow \lambda x.\, N'$ and $(M', N') \in \mathcal{R}$. From $M \Longrightarrow \lambda x.\, M'$ and Proposition 13.14(2), we get, for arbitrary names $p$ and $q$,

$$[\![M]\!]p \stackrel{\overline{p}(v)}{\Longrightarrow} \stackrel{v(x,q)}{\Longrightarrow} \gtrsim [\![M']\!]q\,.$$

Since $[\![M]\!]p \approx [\![N]\!]p$, there is $P''$ such that

$$[\![N]\!]p \stackrel{\overline{p}(v)}{\Longrightarrow} \stackrel{v(x,q)}{\Longrightarrow} P'' \approx [\![M']\!]q\,. \tag{50}$$

We can decompose $[\![N]\!]p \stackrel{\overline{p}(v)}{\Longrightarrow} \stackrel{v(x,q)}{\Longrightarrow} P''$ into $[\![N]\!]p \Longrightarrow\stackrel{\overline{p}(v)}{\longrightarrow} P' \stackrel{v(x,q)}{\Longrightarrow} P''$, for some $P'$. Then, using Proposition 13.16(1-2), we infer that there are $N'$ and $N''$ such that $N \Longrightarrow \lambda x.\, N'$ and $N' \Longrightarrow N''$ with $P' \gtrsim v(x)\, [\![N']\!]$ and

$$P'' \gtrsim [\![N'']\!]q\,. \tag{51}$$

Moreover, by Proposition 13.14(1),

$$[\![N']\!]p \gtrsim [\![N'']\!]p\,. \tag{52}$$

Since $\lesssim \subseteq \approx$, we can combine (50), (51) and (52) to derive $[\![M']\!]q \approx [\![N']\!]q$; hence $(M', N') \in \mathcal{R}$, as required.
Now suppose $M \Longrightarrow xM_1 \ldots M_n$. We suppose $n > 0$; the case $n = 0$ is simpler. We have to find $N_1, \ldots, N_n$ such that $N \Longrightarrow xN_1 \ldots N_n$ and $M_i \;\mathcal{R}\; N_i$, for all $i$. From Proposition 13.14(4), we get

$$[\![M]\!]p \stackrel{\overline{x}(q)}{\Longrightarrow} \gtrsim \mathcal{O}_n\langle q, p, [\![M_1]\!], \ldots, [\![M_n]\!]\rangle$$

and, from $[\![M]\!]p \approx [\![N]\!]p$ and Proposition 13.16(4), for some $m$ and $N_1, \ldots, N_m$,

$$[\![N]\!]p \stackrel{\overline{x}(q)}{\Longrightarrow} \gtrsim \mathcal{O}_m\langle q, p, [\![N_1]\!], \ldots, [\![N_m]\!]\rangle \tag{53}$$

with $\mathcal{O}_n\langle q, p, [\![M_1]\!], \dots, [\![M_n]\!]\rangle \approx \mathcal{O}_m\langle q, p, [\![N_1]\!], \dots, [\![N_m]\!]\rangle$. From this and Lemmas 13.18 and 13.19 we infer that $m = n$ and that $[\![M_i]\!] \approx [\![N_i]\!]$, for all $i$. Moreover, from (53) and Proposition 13.16(4) we infer that

$$N \Longrightarrow x N_1 \dots N_n .$$

Hence $(M_i, N_i) \in \mathcal{R}$. ∎

**Theorem 13.21 (completeness w.r.t. $\approx_\lambda^{\mathsf{open}}$)** *$M \approx_\lambda^{\mathsf{open}} N$ implies $M =_\pi N$.*

**Proof:** We show that

$$\mathcal{R} \stackrel{\text{def}}{=} \bigcup_p \{([\![M]\!]p, [\![N]\!]p) \ : \ M \approx_\lambda^{\mathsf{open}} N\}$$

is closed under substitutions and is a ground bisimulation up-to context and up-to $\gtrsim$. By Theorem B.8, this implies $\mathcal{R} \subseteq \approx$.
First, we show that $\mathcal{R}$ is closed under substitutions. The free names of a process $[\![M]\!]p$ are $\{p\} \cup \mathrm{fv}(M)$. Therefore, for each name substitution $\sigma$ there is a variable substitution $\rho$ such that

$$([\![M]\!]p)\sigma = [\![M\rho]\!]p\sigma .$$

Since by Lemma 13.6 $\approx_\lambda^{\mathsf{open}}$ is closed under variable substitutions, $\mathcal{R}$ is closed under name substitutions.
Now we show that $\mathcal{R}$ is a ground bisimulation up-to context and up-to $\gtrsim$. Let $([\![M]\!]p, [\![N]\!]p) \in \mathcal{R}$ and suppose that $[\![M]\!]p \xrightarrow{\mu} P$. By Proposition 13.16, there are four cases to consider, according to the form of $\mu$. We only show the argument for the case when $\mu$ is a bound output $\overline{x}(q)$ (this case needs both up-to context and up-to $\gtrsim$; the other cases are simpler because they only need only up-to $\gtrsim$).
By Proposition 13.16(4), and there are $x$ and $M_1, \dots, M_n$ such that $M = xM_1 \dots M_n M$ and $P \gtrsim \mathcal{O}_n\langle q, p, [\![M_1]\!], \dots, [\![M_n]\!]\rangle$. Since $M \approx_\lambda^{\mathsf{open}} N$, there are $N_1, \dots, N_n$ such that

$$N \Longrightarrow x N_1 \dots N_n \tag{54}$$

and

$$M_i \approx_\lambda^{\mathsf{open}} N_i, \text{for all } 1 \leq i \leq n. \tag{55}$$

Now, from (54), by Proposition 13.14 we get

$$[\![N]\!]p \stackrel{\overline{x}(q)}{\Longrightarrow} \gtrsim \mathcal{O}_n\langle q, p, [\![N_1]\!], \dots, [\![N_n]\!]\rangle$$

and from (55) we get

$$([\![M_i]\!]r, [\![N_i]\!]r) \in \mathcal{R}$$

for all $r$. Summarising, we have obtained that

$$\begin{array}{rcl} [\![M]\!]p \stackrel{\overline{x}(q)}{\Longrightarrow} & \gtrsim & \mathcal{O}_n\langle q, p, [\![M_1]\!], \dots, [\![M_n]\!]\rangle \, , \\ [\![N]\!]p \stackrel{\overline{x}(q)}{\Longrightarrow} & \gtrsim & \mathcal{O}_n\langle q, p, [\![N_1]\!], \dots, [\![N_n]\!]\rangle \, , \text{ and} \\ ([\![M_i]\!]r, [\![N_i]\!]r) & \in & \mathcal{R}, \quad \text{for all } r \text{ and } 1 \leq i \leq n. \end{array}$$

This is enough, because $\mathcal{R}$ is a ground bisimulation up-to context and up-to $\gtrsim$. ∎

It is worth stressing that without the "up-to context and up-to $\gtrsim$" technique the proof of Theorem 13.21 would be much longer.

Theorems 13.20 and 13.21 prove Theorem 13.8. We summarise the results of Theorems 13.7 and 13.8:

**Corollary 13.22** *For all $M, N \in \Lambda$, it holds that*

$$M =_\pi N \text{ iff } M \approx_\lambda^{\mathtt{open}} N \text{ iff } LT(M) = LT(N).$$

**Remark 13.23 (other call-by-name encodings)** The local structure of the $\pi$-interpretation (Theorem 13.8) does not change if the call-by-name encoding is taken to be that in Table 7 (that has a replication in from of the abstraction) or that in Table 8. The proofs are similar to those given in this section. ■

## 13.4 Böhm Trees

The (informal) definition of Böhm Tree (BT) is obtained from that of LT (Definition 13.2) by replacing clauses (1) and (2) with the clause:

$$BT(M) = \bot \qquad \text{if } M \in PO_n,\ 0 \leq n \leq \omega.$$

We briefly discuss how to obtain BTs from an encoding of the $\lambda$-calculus into $\pi$-calculus. The definition of BT is based on the notion of head normal form (rather than on weak head normal form as in the case of LTs) since only the terms with head normal form have BTs different from $\bot$. To define a $\pi$-calculus semantics that captures BT-equality we therefore need the following two modifications:

1. The $\pi$-calculus encoding of a $\lambda$-term should allow reductions under $\lambda$ (rule $\xi$).
2. The behavioral equivalence chosen for the $\pi$-calculus should be divergence-sensitive, that is regard divergence of a $\pi$-calculus process as disastrous. This is necessary because the computation under the '$\lambda$' may never terminate (this happens in terms that have a whnf but no hnf, like $\lambda x.\,\Omega$ and $\Xi$).

To satisfy (1), we can take the $\pi$-calculus encoding of strong call-by-name, in Section 9. For (2), we can take a divergence-sensitive bisimilarity [Wal90], or must-testing [Hen88]. In this way, the $\pi$-interpretation validates equations such as $\lambda x.\,\Omega = \Omega$ and $\Xi = \Omega$, that are at the basis of the difference between LTs and BTs.

## 13.5 Local structure of the call-by-value encoding

Proceeding as for the call-by-name encoding, one can show that also the call-by-value encoding of Table 3 is sound, but not complete, w.r.t. the operational equivalence of $\lambda V$. For closed $\lambda$-terms $M, N$, if $\mathcal{V}[\![M]\!]$ and $\mathcal{V}[\![N]\!]$ are behaviourally indistinguishable as $\pi$-calculus processes, then $M$ and $N$ are operationally equivalent in $\lambda V$ (defined as for call-by-name, Definition 11.1). But the converse may fail. For a counterexample, take $M \stackrel{\text{def}}{=} \lambda x.\,x$ and $N \stackrel{\text{def}}{=} \lambda x.\,\lambda y.\,(xy)$ (note: this counterexample is again obtained using the $\eta$-rule). Behavioural equivalence in call-by-value is different from call-by-name: for instance in call-by-name $(\lambda x.\, I)\Omega$ and $I$ are semantically the same, whereas in call-by-value $(\lambda x.\, I)\Omega$ is the same as $\Omega$.

No characterisation is known of the equivalence on $\lambda$-terms induced by the call-by-value encoding $\mathcal{V}$.

# 14 Notes

The $\lambda$-calculus was introduced by Church [Chu32, Chu41] who was hoping to use it, on the one hand, to produce a foundation for logic and mathematics and, on the other hand, to understand mathematically the notion of function. While the first goal failed, the second one has been a remarkable success. Church (and Kleene, who proved important results on $\lambda$-definability) realised that the $\lambda$-calculus captures the notion of function that is 'effectively computable', and conjectured that the class of effectively computable functions should coincide with the class of $\lambda$-definable functions. This proposal, referred to as *Church's thesis*, is now generally accepted.

The $\lambda$-calculus was redescovered by computer scientists in the 60's, mainly thanks to Böhm, McCarthy, Scott, and especially Landin, as a basis for programming languages. It has been important to understand essential programming constructs such as parametrisation mechanisms (those, for instance, of procedures) and types. In particular, it has served as a mathematical foundation for the class of languages known as *functional languages*.

The standard references on the classical theory of the $\lambda$-calculus (the *sensible theory*) are [Bar84, HS86]. The textbooks Gunter [Gun92], Hindley [Hin97], and Mitchell [Mit96] contain detailed presentations of typed $\lambda$-calculi and PCF.

PCF was introduced by Plotkin [Plo77]. In the same paper, he shows a sequentiality lemma for PCF from which the non-definability of parallel operators, such as forms of 'parallel-or' and 'parallel-if-then-else', can be derived. The classical sequentiality theorem for the untyped $\lambda$-calculus is due to Berry [Ber78]. A recent treatwise on sequentiality in the $\lambda$-calculus is Bethke and Klop [BK98]. The standardisation and normalisation theorems of the $\lambda$-calculus are proved in [CF58].

*Call-by-need* was proposed by Wadsworth [Wad71] as an implementation technique. Formalisations of call-by-need on a $\lambda$-calculus with a `let` construct or with environments include Ariola et al. [AFM$^+$95], Launchbury [Lau93], Purushothaman and Seaman [PS92], Yoshida [Yos93].

The term "continuation" is due to Strachey and Wadsworth [SW74], who used them to give semantics to control jumps. See Reynold [Rey93] for a history of the discovery of continuations and CPS transforms. For continuations in denotational semantics, see Gordon [Gor79], Schmidt [Sch86], or Tennent [Ten91]. For continuations as a programming technique, see [FWT92]. For the use of CPS transform in compilers, see Appel [App92], where the language is ML, or [FWT92], where the language is Scheme. Hatcliff and Danvy [HD94] present a unifying account of various CPS transforms based on Moggi's computational metalanguage [Mog91]. The encoding of $\lambda N$ into $\lambda V$ using thunks in Exercise 6.17, and its relationship to the CPS transform $\mathcal{C}_{\mathtt{N}}$, are studied by Danvy and Hatcliff [DH92, HD97]. The idea of using thunks for implementing call-by-name dates back to Ingerman [Ing61].

The call-by-value CPS transform of Table 2 is due to Fischer [Fis72] (of which a more complete version is [Fis93]). The call-by-name CPS transform of Table 4 is that of Plotkin [Plo75], based on work by Reynolds (such as [Rey72]); however, we have adopted the rectification in the clause for variables due to Hatcliff and Danvy [HD97] (Plotkin's translation for variables was $\mathcal{C}_{\mathtt{N}}[\![x]\!] \stackrel{\text{def}}{=} x$; the rectification is necessary for the left-to-right implication of Theorem 6.4 to hold).

Theorems 5.1-5.4 and 6.1-6.3, on CPS terms and CPS transforms, are proved by Plotkin [Plo75]. (The assertions of these theorems is actually slightly different from Plotkin's, but the content and the proofs are similar.) Plotkin also presents counterexample (14) to Theorem 5.5. The terms $M$ and $N$ used in the proof of Theorem 12.15 (non-completeness of the call-by-name encoding) are the same used by Plotkin [Plo75] to prove the non-completeness of the call-by-name CPS transform (Table 4) as a transformation of $\lambda N$ into $\lambda V$. A CPS transform for call-by-need is studied by Okasaki, Lee and Tarditi [OLT94], using a $\lambda$-calculus extended with mutable references as target language. We did not find in the literature grammars for the terms of the CPS transforms (grammars (15), (23), (29)); but related to our grammars are Ogata's, who compares cut elimination of certain logics and CPS transforms, and Sabry and Felleisen's [SF93], who gives the language of an optimised version of Fischer's call-by-value CPS. We also did not find in the literature the uniform CPS transform of Table 6.

The relationship between types of $\lambda$-terms and types of their CPS images was first noticed by Meyer and Wand [MW85]. Other important papers on types and CPS are [Mur92, HL93, HDM93]. Theorem 10.2 is due to Meyer and Wand. The translation of arrow types is sometimes called "double-negation construction" after Murthy [Mur92]. The transformation of types for the call-by-name CPS in (36), and Theorem 10.10, are by Harper and Lillibridge [HL93], who follow what Meyer and Wand had done for call-by-value.

Connections among functions, continuations and message-passing are already well visible, though (as far as we know) not formally stated, in Carl Hewitt's works on *actors* [Hew77, HBG$^+$73, HB77]. A function is represented as an actor that accepts messages containing an argument for the function and a continuation to which the result of the function should be sent. This is the same idea that we used for representing functions as $\pi$-calculus processes in the paper.

The analogy between Milner's encodings of $\lambda$-calculus into $\pi$-calculus and the CPS transforms was noticed by several people, and was first partly formalised by Boudol [Bou97] and Thielecke [Thi97]. Boudol compares encodings of call-by-name and call-by-value $\lambda$-calculus into, respectively, the blue calculus and the $\pi$-calculus. He noticed that, for either strategy, if the CPS transform is composed with the encoding of (call-by-name) $\lambda$-calculus into the blue calculus, then the results can be read as the standard encoding of that $\lambda$-calculus strategy into the $\pi$-calculus. Thielecke introduces a CPS calculus, similar to the intermediate language in Appel's compiler [App92]. He shows that this CPS calculus has a simple translation into the $\pi$-calculus and that, if Plotkin's CPS transforms are formulated in the CPS calculus, then their translations into the $\pi$-calculus yield an encoding similar to Milner's [Mil92]. In this paper we go further, in that the encodings of both call-by-name and call-by-value $\lambda$-calculus into the $\pi$-calculus are *factorised* using the CPS transforms and the compilation $\mathcal{C}$ of HO$\pi$ into $\pi$-calculus, for both untyped *and* typed $\lambda$-calculi. One of the reasons for working out the factorisations in detail is to be able to *derive* the correctness of the $\pi$-calculus encodings (on terms as well as on types) from those of the CPS transforms and of the compilation of HO$\pi$ into $\pi$-calculus.

Translations of functions into process calculi have been given by Kennaway and Sleep [KS82], Leth [Let91], Thomsen [Tho90], Boudol [Bou89]. Robin Milner's work on functions as $\pi$-calculus processes [Mil92] is a landmark in the area.

Milner considers the call-by-name and parallel call-by-value strategies (his encodings are, respectively, that in Table 8 and a variant of that in 3). Milner proves the operational correpondence between reductions in the $\lambda$-terms and in their process encoding (the analogue of Lemma 5.18 and Exercise 6.19); he also proves that, in both cases, the encoding is operationally sound but not complete.

Regarding the call-by-value encoding, (variants of) Corollary 5.15 and Lemma 5.19 are by Milner [Mil92]; Corollary 5.16 is by Sangiorgi and Pierce [PS96]; Remark 5.17 is from [San92]. Regarding the call-by-name encoding, Corollary 6.12 and 6.13 are by Milner [Mil92, Mil91]. The remainder of Section 6 is new. The uniform encoding in Section 7 is from [OD93]. A study of the correctness of the call-by-need encoding in Table 7 is in [BO95]. Encodings of graph reductions, related to call-by-need, into $\pi$-calculus are given in [Bou94, Jef93] but their correctness is not studied. Niehren [Nie96] uses encodings of call-by-name, call-by-value and call-by-need $\lambda$-calculi into $\pi$-calculus to compare the time complexity of the strategies. The encoding of Table 8 is precisely Milner's original encoding of $\lambda N$. The optimisation (30) is from [MS98]. An encoding of the $\xi$ rule into $\pi$-calculus was first given in [San96], where the target calculus is $\pi$I; the encoding presented in Section 9 is from [MS98].

Turner first established a relationship between the types of $\lambda$-calculus terms and those of their encodings into $\pi$-calculus [Tur96]. He takes (variants of) Milner's encodings of the $\lambda$-calculus into the $\pi$-calculus and proves that for some of these encodings there is a correspondence between principal types of the $\lambda$-terms and principal types of the encoding $\pi$-calculus terms; the $\pi$-calculus type system used is (the structural version of) Milner's sorting plus polymorphism. Turner also extends Milner's encodings to the polymorphic $\lambda$-calculus. Using $\mathsf{i}/\mathsf{o}$ types, as we have done in the paper, the relationship between $\lambda$-calculus and $\pi$-calculus types is clearer and sharper, and can be easily extended to other type systems. The work presented in Section 10 follows the schema of the interpretation Abadi and Cardelli's *typed object calculus* [AC96] into $\pi$-calculus in [San98]. Encodings of simply-typed $\lambda$-calculi are also given by Kobayashi [Kob98], to illustrate the use of a type system guaranteeing absence of deadlocks.

*Applicative bisimulation* was introduced by Abramsky [Abr89], inspired by the work of Milner and Park in concurrency [Par81, Mil89]. Since Abramsky's work, the idea of applicative bisimulation has been applied to a variety of higher-order sequential languages; see [Gor95, Pit97] for surveys.

Open applicative bisimulation coincides with the equivalence induced by Ong's *lazy PSE ordering* [Ong88]; however, a conceptual difference between the two is the emphasis that Ong's preorder places on $\eta$-expansion.

The operational and denotational theory of $\lambda N$ (that they call the *lazy $\lambda$-calculus*) has been extensively studied by Abramsky and Ong [Abr87, Abr89, Ong88, AO93]. The canonical model of $\lambda N$ mentioned in Section 11 was defined by Abramsky, as the solution to a certain domain equation [Abr87, Abr89]. He and Ong then proved that this model is sound but not complete [Abr87, Ong88, AO93].

The *internal choice* operator in Section 12 is inspired by an operator introduced by De Nicola and Hennessy [DH87] in a CCS-like process calculus. Studies of format of operators in concurrency include [DS85, BIM88, ABV94, GV92], and in functional languages [Blo90, How96, San97a].

The Lévy-Longo Trees were introduced by Longo [Lon83]—where they were simply called trees—developing an original idea by Levy [Lév76]. They were called Lévy-Longo Trees by Ong [Ong88]. Böhm Trees are studied in depth in [Bar84]. For models of the $\lambda$-calculus, such as $P\omega$, $T^\omega$, $D_A$, $D_\infty$, and free lazy models, see [Bar84, HS86, Ong88].

Most of the results in Sections 11-13 are from [San94],[San92, chapter 6] (some of the results in [San92, chapter 6] have also appeared in [San95a]). The grammar of well-formed operators in [San94] is more generous. Section 13.4 (Böhm Trees) is from [San95b].

The equivalence on $\lambda$-terms induced by their encoding into the $\pi$-calculus (Definition 11.5) is largely independent of the choice of the behavioural equivalence on $\pi$-calculus processes. We have adopted bisimilarity which is widely accepted as the finest extensional behavioural equivalence one would like to impose on processes; at the opposite extreme, as the coarsest equivalence, one normally places *Morris's context-equivalence* (defined as barbed congruence but without the clause on $\tau$ moves). On processes encoding $\lambda$-terms, the main forms of $\pi$-calculus bisimilarity that appear in the literature (ground, late, early, open, and their asynchronous variants) coincide. They also coincide with may-testing equivalence. The latter fact intuitively holds because the encoded lazy $\lambda$-terms are deterministic, and is formally proved in [BL95].

Other characterisations of Lévy-Longo Trees as *Morris's context-equivalence* of extended $\lambda$-terms include: Boudol and Laneve [BL96], who use *lambda calculus with multiplicities*, a form of $\lambda N$ where arguments of functions have a multiplicity and reduction, although deterministic, may introduce deadlock; and Dezani-Ciancaglini, Tiuryn and Urzyczyn [DCTU97] who use the *concurrent $\lambda$-calculus*, a form of $\lambda N$ extended with forms of call-by-value application, and operators for nondeterminism and parallelism.

*Grammars:*

| | | | |
|---|---|---|---|
| $a, b, \ldots x, y, z$ | | | *Names* |
| | | | *Values* |
| $v, w$ | $:=$ | $a$ | names |
| | | | *Processes* |
| $P, Q, R$ | $:=$ | $\mathbf{0}$ | nil |
| | $\vert$ | $P \mid P$ | parallel |
| | $\vert$ | $\boldsymbol{\nu} a\, P$ | restriction |
| | $\vert$ | $\mathit{Pref}$ | prefixing |
| | $\vert$ | $!P$ | replication |
| | $\vert$ | $[a = b]P$ | matching |
| | $\vert$ | $\mathit{Pref} + \mathit{Pref}$ | summation |
| | | | *Prefixes* |
| $\mathit{Pref}$ | $:=$ | $a(x).\, P$ | input |
| | $\vert$ | $\overline{a}\langle v\rangle.\, P$ | output |
| | $\vert$ | $\tau.\, P$ | silent prefix |

Table 13: The $\pi$-calculus

# APPENDIXES: background on the $\pi$-calculus

## A The $\pi$-calculus

### A.0.1 Processes

The grammar of the $\pi$-calculus is in Table 13. $\mathbf{0}$ denotes the inactive process. An input-prefixed process $a(x).\, P$ waits for a value $v$ at $a$ and then continues as $P\{^{v}\!/\!_{x}\}$. An output process $\overline{a}\langle v\rangle.\, P$ emits value $v$ at $a$ and continues as $P$. A $\tau$-prefixed process $\tau.\, P$ can evolve internally to $P$. A parallel composition $P_1 \mid P_2$ is to run two processes in parallel. A restriction $\boldsymbol{\nu} a\, P$ makes name $a$ local, or private, to $P$. A replicated process $!P$ stands for a countable infinite number of copies of $P$. Matching $[a = b]P$ is to test for the equality between the names $a$ and $b$. Summation expresses a choice between two behaviours (it is also called *guarded* summation, as the summands are guarded by a prefix).

| | |
|---|---|
| $P\{a = (b).\,Q\}$ | local environment (Section B.0.9) |
| $\simeq^{\mathsf{c}}, \approxeq^{\mathsf{c}}$ | strong and weak barbed congruence (Sections B.0.4, B.0.6, B.1.5) |
| $\sim, \approx$ | strong and weak ground bisimilarity (Section B.0.5, B.0.6, B.1.5) |
| $\lesssim$ | expansion (Section B.0.7) |
| $\xrightarrow{\alpha}, \longrightarrow, \longrightarrow_{\mathtt{d}}$ | strong transitions (Section A.0.2) |
| $\stackrel{\alpha}{\Longrightarrow}, \Longrightarrow$ | weak transitions (Section A.0.2) |
| $\mu$ | ground action (Section B.0.5) |
| $\sharp T$ | channel type (Section B.1.2) |
| $\langle \widetilde{T} \rangle$ | product type (Section B.1.2) |
| $\mathtt{unit}, \star$ | unit type and unit value (Section B.1.1) |
| $\mu X.\, T$ | recursive types (Section B.1.2) |
| $\mathsf{i}\,T, \mathsf{o}\,T, \sharp\,T$ | $\mathsf{i}/\mathsf{o}$ types (Section B.1.3) |
| $S < T$ | subtype relation (Section B.1.3) |
| $(T)^{-}, (T)^{\leftarrow\sharp}, T \frown S$ | auxiliary notation for $\mathsf{i}/\mathsf{o}$ types (Section B.1.3) |
| $Fc, b(\widetilde{c})\,F, b\,F, F \approxeq^{\mathsf{c}}_{\Gamma;T} G$ | notation for abstractions (Section B.1.7) |

Table 14: Main notations for the $\pi$-calculus

We use $\sigma$ to range over substitutions; for any expression $E$, we write $E\sigma$ for the result of applying $\sigma$ to $E$, with the usual renaming convention to avoid captures. We assign sum and parallel composition the lowest precedence among the operators. Substitutions have precedence over the operators of the language.

In an input $a(b).\,P$ and an output $\overline{a}\langle b\rangle.\,P$ name $a$ is the *subject* and name $b$ the *object* of the prefix. An input prefix $a(b).\,P$ and a restriction $\boldsymbol{\nu}bP$ are binders for name $b$, and give rise in the expected way to the definitions of free and bound names, and of $\alpha$-conversion. Symbol '=' denotes equality up to $\alpha$-conversion. We identify $\alpha$-equivalent processes. In statements, we always assume that bound names of the expressions of the statements (processes, actions, etc.) are *fresh*, i.e., they are different from the other bound and free names of the expressions in the statement; and we say that a name $a$ is *fresh for an expression $E$* to mean that $a$ does not appear in $E$. We write $\mathrm{fn}(E)$ and $\mathrm{bn}(E)$ for respectively the free and bound names of $E$, and $\mathrm{n}(E)$ for all names in $E$ (free and bound). A *context* is a process expression with a single occurrence of a hole $[\cdot]$ in it; a context is obtained by replacing an occurrence of $\mathbf{0}$ in a process with $[\cdot]$. We abbreviate $\boldsymbol{\nu}a\,\boldsymbol{\nu}b\,P$ as $(\boldsymbol{\nu}a, b\,)P$. We write $a.\,P$ and $\overline{a}.\,P$ when the value communicated along name $a$ is not important. We omit the trailing $\mathbf{0}$ in prefixed processes $\pi.\,\mathbf{0}$. Finally, $\overline{a}(b).\,P$ (*bound output prefix*) abbreviates $\boldsymbol{\nu}b\,(\overline{a}\langle b\rangle.\,P)$.

We use a tilde to represent tuples of expressions. With some abuse of notation, we sometimes view $\widetilde{E}$ both as a tuple and a set. We extend notations to tuples componentwise.

Table 14 collects some notations for the $\pi$-calculus (for behavioural equivalences, types) that are discussed in the sections below.

### A.0.2 Operational semantics

The operational semantics of processes is given by means of a labeled transition system, which expresses the internal steps that a process can make and the communications with other processes in which it can engage. (This is the standard transition system of the $\pi$-calculus, in the early style.) The set of rules is given in Table 15, plus the symmetric variants of COM and PAR and SUM. Transitions are of the form $P \xrightarrow{\alpha} P'$, where the label $\alpha$ ranges over *actions* of the following forms:

| | |
|---|---|
| $\tau$ | interaction |
| $a\langle v\rangle$ | input of $v$ at $a$ |
| $(\boldsymbol{\nu}\widetilde{b}\,)\overline{a}\langle v\rangle$ | output of $v$ at $a$, extruding bound names $\widetilde{b}$ |

In the case of input and output, $a$ is the *subject* of the action. Input and output actions describe possible interactions between $P$ and its environment, while a $\tau$ action represents an internal action in which one subprocess of $P$ communicates with another. We write

$$P \xrightarrow{a(x)} P',$$

and call it *bound input action*, to mean that $P \xrightarrow{a\langle x\rangle} P'$ and $x$ is not a free name of $P$; in this case, $x$ is a bound name of the action, and $a$ a free name. We sometimes abbreviate $P \xrightarrow{\tau} P'$ as $P \longrightarrow P'$, and call it a *reduction*. We also write $P(\longrightarrow)^n P'$ to mean that $P$ becomes $P'$ by performing $n$ $\tau$-transitions. *Weak transitions* are defined as usual: relation $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$; and $\stackrel{\alpha}{\Longrightarrow}$ stands for $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$.

We write $P \longrightarrow_{\mathtt{d}} P'$ if the reduction $P \longrightarrow P'$ is *deterministic* i.e. $P \longrightarrow P'$ is the only immediate possible transition for $P$. We write $P \longrightarrow_{\mathtt{d}}^n P'$ if $P$ evolves to $P'$ by performing $n$ deterministic reductions.

### A.0.3 The asynchronous $\pi$-calculus

A common subcalculus of the $\pi$-calculus is the *asynchronous $\pi$-calculus* ($\pi_{\mathrm{a}}$) [HT91, Bou92, ACS96], which has no operators of summation and matching, and all output prefixes are of the form $\overline{a}\langle b\rangle$ (they have no continuation). The encodings of the $\lambda$-calculus in this paper are written in $\pi_{\mathrm{a}}$. The processes of $\pi_{\mathrm{a}}$ enjoy some interesting behavioural properties, some of which are reported below.

# B Behavioural equivalences

### B.0.4 Barbed congruence

We define behavioural equality using the notion of barbed congruence [MS92]. This is a bisimulation-based equivalence that can be defined uniformly on different calculi. Barbed congruence can be defined on any calculus possessing a reduction relation (the $\tau$ steps of the $\pi$-calculus) and an observability predicate $\downarrow_a$, for each channel $a$, which detects the possibility for a process of accepting a communication at $a$ with the external environment. In the $\pi$-calculus, $P \downarrow_a$ holds if there are a derivative $P'$ and an input or output action $\alpha$ with subject $a$ such that $P \xrightarrow{\alpha} P'$. Barbed congruence is the congruence induced by

$$\textsc{Alpha}\ \frac{P\ \alpha\text{-convertible to } P' \qquad P' \xrightarrow{\alpha} P''}{P \xrightarrow{\alpha} P''}$$

$$\textsc{Inp}\ \frac{}{a(x).\,P \xrightarrow{a\langle v\rangle} P\{v/x\}} \qquad \textsc{Par}\ \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2}\,\mathrm{bn}(\alpha) \cap \mathrm{fn}(P_2) = \emptyset$$

$$\textsc{Out}\ \frac{}{\overline{a}\langle v\rangle.\,P \xrightarrow{\overline{a}\langle v\rangle} P} \qquad \textsc{Com}\ \frac{P_1 \xrightarrow{(\boldsymbol{\nu}\widetilde{b}\,)\overline{a}\langle v\rangle} P_1' \quad P_2 \xrightarrow{a\langle v\rangle} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} \boldsymbol{\nu}\widetilde{b}\,(P_1' \mid P_2')}\ \widetilde{b} \cap \mathrm{fn}(P_2) = \emptyset$$

$$\textsc{Mat}\ \frac{P \xrightarrow{\alpha} P'}{[a = a]P \xrightarrow{\alpha} P'} \qquad \textsc{Sum}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\textsc{Rep}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} !P \mid P'} \qquad \textsc{Res}\ \frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu}c\,P \xrightarrow{\alpha} \boldsymbol{\nu}c\,P'}\ c \notin (\mathrm{fn}(\alpha) \cup \mathrm{bn}(\alpha))$$

$$\textsc{Tau}\ \frac{}{\tau.\,P \xrightarrow{\tau} P} \qquad \textsc{Open}\ \frac{P \xrightarrow{(\boldsymbol{\nu}\widetilde{b}\,)\overline{a}\langle v\rangle} P'}{\boldsymbol{\nu}c\,P \xrightarrow{(\boldsymbol{\nu}\widetilde{b},c\,)\overline{a}\langle v\rangle} P'}\ c \in \mathrm{fn}(v) - \widetilde{b}$$

Table 15: Transition rules for the $\pi$-calculus.

---

barbed bisimulation. The latter equates processes that can match each other's reductions and, at each step, are observable on the same channels.

Below, we define barbed congruence on a generic process calculus $\mathcal{L}$ that has reduction and observation relations as above. An $\mathcal{L}$ *relation* is a relation on the processes of $\mathcal{L}$.

**Definition B.1 (strong barbed bisimilarity and congruence)** *A $\mathcal{L}$ relation $\mathcal{R}$ is an $\mathcal{L}$* strong barbed bisimulation *if $P \mathrel{\mathcal{R}} Q$ implies:*

1. *if $P \xrightarrow{\tau} P'$ then there is $Q'$ such that $Q \xrightarrow{\tau} Q'$ and $P' \mathrel{\mathcal{R}} Q'$,*
2. *for each channel $a$, $P \downarrow_a$ implies $Q \downarrow_a$,*

*and the converse, on the transitions from $Q$.*

*Two $\mathcal{L}$ processes $P$ and $Q$ are* strongly barbed bisimilar (in $\mathcal{L}$)*, written $\mathcal{L} \triangleright P \stackrel{\centerdot}{\sim} Q$, if $P \mathrel{\mathcal{R}} Q$ for some $\mathcal{L}$ strong barbed bisimulation $\mathcal{R}$.*

*Two $\mathcal{L}$ processes $P, Q$ are* strongly barbed congruent (in $\mathcal{L}$)*, written $\mathcal{L} \triangleright P \simeq^{\mathrm{c}} Q$, if, for each context $C$ in $\mathcal{L}$, it holds that $\mathcal{L} \triangleright C[P] \stackrel{\centerdot}{\sim} C[Q]$.*

When there is no ambiguity on what the calculus $\mathcal{L}$ is, we may abbreviate $\mathcal{L} \triangleright P \stackrel{\centerdot}{\sim} Q$ as $P \stackrel{\centerdot}{\sim} Q$, and $\mathcal{L} \triangleright P \simeq^{\mathrm{c}} Q$ as $P \simeq^{\mathrm{c}} Q$. (When $P, Q$ are $\pi$-calculus processes, $P \simeq^{\mathrm{c}} Q$ means $\pi \triangleright P \simeq^{\mathrm{c}} Q$, even if both $P$ and $Q$ are processes of the asynchronous $\pi$-calculus—which is a subcalculus of the $\pi$-calculus).

### B.0.5 Ground bisimilarity

The main inconvenience of barbed congruence is that it uses quantification over contexts in the definition, and this can make proofs of process equalities heavy.

Against this, it is important to find proof techniques, especially formulations of bisimulation whose definition does not use context quantification. One such formulation is *ground bisimilarity* below. Its input clause only checks bound input actions; therefore processes need not be tested on names that they already possess. We call $\tau$-actions, output actions and bound input actions the *ground actions*, ranged over by $\mu$.

**Definition B.2 (strong ground bisimilarity)** *A relation $\mathcal{R}$ on $\pi$-calculus processes is a* ground bisimulation *if $P \mathcal{R} Q$ implies, for all ground actions $\mu$:*

- *if $P \xrightarrow{\mu} P'$ then there is $Q'$ such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$,*

*and the converse, on the transitions from $Q$.*

*Two $\pi$-calculus processes $P$ and $Q$ are* strongly ground bisimilar*, written $P \sim Q$, if $P \mathcal{R} Q$, for some ground bisimulation $\mathcal{R}$.*

(In the clause of ground bisimulation, we omit the requirement "bound names of $\mu$ fresh for $Q$", since we work up-to $\alpha$-conversion). On processes of the asynchronous $\pi$-calculus, ground bisimulation implies barbed congruence:

**Theorem B.3** *Let $P, Q \in \pi_{\mathrm{a}}$. Then $P \sim Q$ implies $\pi \triangleright P \simeq^{\mathrm{c}} Q$.*

(On processes of the asynchronous $\pi$-calculus, ground bisimilarity coincides with early bisimilarity [San95a].)

### B.0.6 Weak equivalences

The weak versions of the above equivalences are defined by replacing the transition $Q \xrightarrow{\tau} Q'$ with the weak transition $Q \Longrightarrow Q'$, and, if $\mu$ is an input or an output action, the strong transition $Q \xrightarrow{\mu} Q'$ with the weak transition $Q \stackrel{\mu}{\Longrightarrow} Q'$; also, in Definition B.1, the predicate $Q \downarrow_a$ with $Q \Downarrow_a$, where $\Downarrow_a \stackrel{\text{def}}{=} \Longrightarrow \downarrow_a$.

We write $\approxeq^{\mathrm{c}}$ for weak barbed congruence; and $\approx$ for weak ground bisimilarity; we sometimes just call them *barbed congruence*, and *ground bisimulation.* Theorem B.3 also holds in the weak case:

**Theorem B.4** *Let $P, Q \in \pi_{\mathrm{a}}$. Then $P \approx Q$ implies $\pi \triangleright P \approxeq^{\mathrm{c}} Q$.*

Weak barbed congruence is the semantic equality on the $\pi$-calculus we are mainly interested in; other relations, like strong bisimilarities, ground bisimilarity, and expansion (defined below), will serve as auxiliary to it.

**Lemma B.5** *Suppose $P, Q \in \pi_{\mathrm{a}}$. Then $\pi \triangleright P \approxeq^{\mathrm{c}} Q$ implies $\pi_{\mathrm{a}} \triangleright P \approxeq^{\mathrm{c}} Q$.*

**Proof:** $\pi_{\mathrm{a}}$ is a subcalculus of $\pi$, therefore all contexts of $\pi$ are also contexts of $\pi_{\mathrm{a}}$. ■

### B.0.7 The expansion relation

The expansion relation [AKH92, SM92], written $\lesssim$, is an asymmetric variant of $\approx$ which takes into account the number of $\tau$-actions performed by processes. Thus, $P \lesssim Q$ holds if $P \approx Q$ and $Q$ has at least as many $\tau$-moves as $P$.

**Definition B.6 (expansion)** *A relation $\mathcal{R}$ on $\pi$-calculus processes is an* expansion *if $P \mathcal{R} Q$ implies:*

1. *If $P \xrightarrow{\mu} P'$ then there is $Q'$ such that $Q \stackrel{\mu}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$;*
2. *if $Q \xrightarrow{\tau} Q'$, then either $P \mathcal{R} Q'$, or there is $P'$ such that $P \xrightarrow{\tau} P'$ and $P' \mathcal{R} Q'$;*
3. *if $Q \xrightarrow{\mu} Q'$, where $\mu$ is an output or a bound input action, then there is $P'$ such that $P \xrightarrow{\mu} P'$ and $P' \mathcal{R} Q'$.*

*We say that $Q$* expands *$P$, written $P \lesssim Q$, if $P \mathcal{R} Q$ for some expansion $\mathcal{R}$.*

**Lemma B.7** *Let $P, Q$ be $\pi$-calculus processes. Then $P \lesssim Q$ implies $P \approx Q$.*

### B.0.8 Techniques of "bisimulation up to"

We shall use a proof technique for bisimulation in order to reduce the size of the relations to exhibit. In the bisimilarity clause, this technique allows us to manipulate the derivatives of two processes with the *expansion relation* and to cancel a common context. The technique uses the notions of *ground bisimulation up-to context and up-to* $\gtrsim$ and of relation *closed under substitutions.* A *polyadic* context may contain an arbitrary number of different holes, and, moreover, each of these holes may appear more than once.

- The definition of *ground bisimulation up-to context and up-to* $\gtrsim$ is like that of (weak) ground bisimulation, except that the clause $P' \mathcal{R} Q'$ is relaxed to:

  there are a polyadic context $C$ and processes $\widetilde{P''}$ and $\widetilde{Q''}$ s.t. $P' \gtrsim C[\widetilde{P''}]$, $Q' \gtrsim C[\widetilde{Q''}]$ and $\widetilde{P''} \mathcal{R} \widetilde{Q''}$.

- A relation $\mathcal{R}$ is *closed under substitutions* if $P \mathcal{R} Q$ implies $P\sigma \mathcal{R} Q\sigma$, for all substitution $\sigma$.

**Theorem B.8** *If $\mathcal{R}$ is is closed under substitutions and is a ground bisimulation up-to context and up-to $\gtrsim$, then $\mathcal{R} \subseteq \approx$.*

### B.0.9 Some laws for the $\pi$-calculus

**Lemma B.9** 1. *Abelian monoid laws for parallel composition: $P \mid Q \sim Q \mid P$, $P \mid (Q \mid R) \sim (P \mid Q) \mid R$, $P \mid \mathbf{0} \sim P$;*

2. *$\boldsymbol{\nu} a\, \mathbf{0} \sim \mathbf{0}$, $\boldsymbol{\nu} a\, \boldsymbol{\nu} b\, P \sim \boldsymbol{\nu} b\, \boldsymbol{\nu} a\, P$;*
3. *$(\boldsymbol{\nu} a\, P) \mid Q \sim \boldsymbol{\nu} a\, (P \mid Q)$, if $a$ not free in $Q$;*

We report some distributivity laws for private replications, i.e., for systems of the form

$$\boldsymbol{\nu} a\,(P \mid {!a(x)}.\,Q)$$

in which $a$ may occur free in $P$ and $Q$ only in output subject position. One should think of $Q$ as a private resource of $P$, for $P$ is the only process who can access $Q$; indeed $P$ can activate as many copies of $Q$ as needed. We write such a system as $P\{a = (x).\,Q\}$. The notation $\{a = (x).\,Q\}$ has syntactic precedence over the operators of the calculus.

**Theorem B.10 (replication theorems)** *Suppose $a$ occurs free in $P, R, Q$ only in output subject position. Then:*

1. $(P \mid R)\{a = (x).\,Q\} \sim P\{a = (x).\,Q\} \mid R\{a = (x).\,Q\}$;
2. $(!P)\{a = (x).\,Q\} \sim {!P}\{a = (x).\,Q\}$;
3. $(b(y).\,P)\{a = (x).\,Q\} \sim b(y).\,P\{a = (x).\,Q\}$, *if $y$ is not free in $Q$ and $a \neq b$;*
4. $(\boldsymbol{\nu} c\, P)\{a = (x).\,Q\} \sim \boldsymbol{\nu} c\, P\{a = (x).\,Q\}$, *if $c$ is not free in $a(x).\,Q$;*
5. $P\{a = (x).\,Q\} \sim P$, *if $a$ is not free in $P$;*
6. $\overline{a}\langle d\rangle\{a = (x).\,Q\} \gtrsim (Q\{d/x\})\{a = (x).\,Q\}$.

## B.1 Extensions and types

### B.1.1 Some conventions and notation for typed calculi

In typed $\pi$-calculi, *type environments*, ranged over by $\Gamma$, are a finite assignment of types to names. A typing judgement $\Gamma \vdash P$ asserts that process $P$ is well-typed under the type assumptions $\Gamma$, and $\Gamma \vdash v : T$ that value $v$ has type $T$ under the type assumptions $\Gamma$. (We may regard $\Gamma \vdash P$ as an abbreviation for $\Gamma \vdash P : \diamond$, where $\diamond$ is the type of all processes; we will use $\diamond$ for defining the *Higher-Order $\pi$-calculus*, in Section C.) When we deal with several typed calculi, to avoid ambiguity, we sometimes add the name of the calculus to typing judgements; thus we write $\mathcal{L} \triangleright \Gamma \vdash P$ to mean that the typing judgement $\Gamma \vdash P$ holds in the calculus $\mathcal{L}$.

We distinguish between *object* (or *value*) types and *subject* types. The former are the types of the values that may be communicated; the latter are the types of the names along which these values are communicated. In the $\pi$-calculus, subject types are also object types; in the Higher-Order $\pi$-calculus of Section C, by contrast, object and subject types are distinct. In typed calculi, a restricted name is annotated with a type, as in $(\boldsymbol{\nu} a : T)\, P$, $\overline{b}(a : T).\,P$, and $P\{a : T = (x).\,Q\}$; the type $T$ is a subject type; it is omitted when not important. A type environment $\Gamma$ is *closed* if $\Gamma$ is an assignment of subject types to names.

We write $\widetilde{x} : T$ to mean that each $x \in \widetilde{x}$ has type $T$; and $\Gamma \vdash P, Q$ to mean that both $\Gamma \vdash P$ and $\Gamma \vdash Q$ hold. Substitutions are performed only between names of the same type.

### B.1.2 Polyadicity

In the polyadic $\pi$-calculus tuples of names can be communicated. For this, the following production is added to the grammar of values:

$$\begin{array}{rcll} & & \textit{Values} & \\ v & := & v_1, \ldots, v_n & \text{tuples} \end{array}$$

Having added a constructor for values, we need a corresponding destructor. Usually a destructor is added as a new operator to the grammar for processes. In the case of tuples, however, it is convenient to decompose a value by means of pattern matching in input prefix, thus adopting the polyadic form of input $a(\widetilde{x}).\,P$.

In an untyped polyadic $\pi$-calculus there can be run-time errors due to arity mismatch on communication. These errors can be avoided by assigning types to names. We call *polyadic $\pi$-calculus* the calculus with channel types, recursive types, and product types [Mil91, VH93, PS96, Tur96]. Channel types are of the form $\sharp\, T$; product type of the form $\langle T_1, \ldots, T_n \rangle$ for $n \geq 0$; recursive types of the form $\mu X.\, S$, where $X$ is a type variable. For instance: $\langle T, S \rangle$ is the type of a pair, whose first component has type $T$ and the second type $S$; a name $a$ with type $\sharp\, \langle T_1, \ldots, T_n \rangle$ may only carry $n$-uples of values, where the $i$-th value has type $T_i$. Throughout the paper, we regard a recursive type as an abbreviation for its infinite unfolding.

Notations, definitions and results for the monadic $\pi$-calculus in this section generalise to the polyadic $\pi$-calculus in the obvious way.

### B.1.3 $\mathsf{i}/\mathsf{o}$ types

Frequently in $\pi$-calculus, one wishes to distinguish between the capability of using a name in input, or in output. For this, channel types are refined by introducing the types $\mathsf{i}\, T$ and $\mathsf{o}\, T$, with the following informal meaning:

- $\mathsf{o}\, T$ is the type of a name that may be used only in output and that carries values of type $T$;
- $\mathsf{i}\, T$ is the type of a name that may be used only in input and that carries values of type $T$.

The channel type $\sharp\, T$ gives, by contrast, both the input and the output capabilities. For instance, a type $a : \sharp\, (\mathsf{i}\, S \times \mathsf{o}\, T)$ says that name $a$ can be used *both* in input *and* in output, and that any message at $a$ carries a pair of names; moreover, the first component of the pair can be used by the recipient *only to input* values of type $S$, the second *only to output* values of type $T$. One of reasons why $\mathsf{i}/\mathsf{o}$ types are useful is that they give rise to subtyping: type annotation $\mathsf{i}$ gives covariance, $\mathsf{o}$ gives contravariance, and $\sharp$ gives invariance. Moreover, since a tag $\sharp$ gives more freedom in the use of a name, for each type $T$ we have $\sharp\, T < \mathsf{i}\, T$ and $\sharp\, T < \mathsf{o}\, T$.

Subtyping judgements are of the form $\Sigma \vdash S < T$, where $\Sigma$ represents the subtyping assumptions (this is written $S < T$ when the subtyping assumptions are empty). As an example, here are the subtyping rule for the $\mathsf{i}/\mathsf{o}$ type constructs:

$$\frac{\Sigma \vdash S < T \qquad \Sigma \vdash T < S}{\Sigma \vdash \sharp\, S < \sharp\, T}$$

$$\frac{I \in \{\sharp, \mathsf{i}\} \qquad \Sigma \vdash S < T}{\Sigma \vdash I\,S < \mathsf{i}\,T}$$

$$\frac{I \in \{\sharp, \mathsf{o}\} \qquad \Sigma \vdash T < S}{\Sigma \vdash I\,S < \mathsf{o}\,T}$$

We call $\sharp$, $\mathsf{i}, \mathsf{o}$ the $\mathsf{i}/\mathsf{o}$ *tags*, and $\sharp\,T$, $\mathsf{i}\,T$ and $\mathsf{o}\,T$ the $\mathsf{i}/\mathsf{o}$ *types*. For readability, we sometimes use brackets with $\mathsf{i}/\mathsf{o}$ types, as in $\mathsf{o}\,(T)$. We also occasionally use these notations for $\mathsf{i}/\mathsf{o}$ types:

- $(T)^-$ is the type obtained from $T$ by cancelling the outermost $\mathsf{i}/\mathsf{o}$ tag.
- $(T)^{\leftarrow\sharp}$ is the type obtained by replacing the outermost $\mathsf{i}/\mathsf{o}$ tag in $T$ with $\sharp$.
- for $I \in \{\sharp, \mathsf{o}, \mathsf{i}\}$, we write $T \frown I\,S$ for $I\,(S \times T)$.

For instance, $(\mathsf{i}\,S)^-$ is $S$, and $(\mathsf{i}\,S)^{\leftarrow\sharp}$ is $\sharp\,S$. Both for $(T)^-$ and for $(T)^{\leftarrow\sharp}$, we unfold $T$ first if its outermost construct is recursion.

**Lemma B.11** *If* $\Gamma \vdash P$ *and* $p$ *is not free in* $P$*, then for all* $S$ *also* $\Gamma, p : S \vdash P$*.*

**Lemma B.12** *If* $\Gamma, p : S \vdash P$ *and* $T < S$ *then also* $\Gamma, p : T \vdash P$*.*

### B.1.4 Linearity and receptiveness

Further refinements of the $\mathsf{i}/\mathsf{o}$ types are the type systems for *linearity* [KPT96] and for *receptiveness* [San97b]. We do not present these type systems here, as they are mentioned in very few places in this paper. We only recall that linearity allows us to say that a name may be used to perform a reduction at most once.

### B.1.5 Types and behavoural equivalences

The use of types affects contextually-defined forms of behavioural equivalence like barbed congruence, for in a typed calculus the processes being compared must obey the same typing and the contexts in which they are tested must be compatible with this typing. A $(\Gamma/\Delta)$-context is a context that, when filled in with a process obeying typing $\Delta$, becomes a process obeying typing $\Gamma$.

We define typed barbed congruence on a generic typed process calculus $\mathcal{L}$ that has reduction and observation relations as above, and typing judgements for processes of the form $\mathcal{L} \triangleright \Gamma \vdash P$; we assume that contexts, closed typing, etc. are defined in $\mathcal{L}$ as in $\pi$-calculus. A *typed* $\mathcal{L}$ *relation* is a set of triples $(\Delta\,;\,P\,;\,Q)$ where $\Delta$ is a closed typing and $\mathcal{L} \triangleright \Delta \vdash P, Q$.

**Definition B.13 (strong barbed congruence)** *Let* $\Delta$ *be a typing, with* $\mathcal{L} \triangleright \Delta \vdash P, Q$*. We say that processes* $P, Q$ *are* strongly barbed congruent (in $\mathcal{L}$) at $\Delta$, *written* $\mathcal{L}(\Delta) \triangleright P \simeq^{\mathrm{c}} Q$*, if, for each closed type environment* $\Gamma$ *and* $(\Gamma/\Delta)$-context $C$*, we have* $\mathcal{L} \triangleright C[P] \stackrel{\centerdot}{\sim} C[Q]$*.*

Again, *weak* barbed congruence at $\Delta$ is defined by replacing $\stackrel{\centerdot}{\sim}$ with $\stackrel{\centerdot}{\approx}$ and $\simeq^{\mathrm{c}}$ with $\approxeq^{\mathrm{c}}$ in Definition B.13.

When there is no ambiguity on what the calculus $\mathcal{L}$ is, we may drop $\mathcal{L}$, and abbreviate $\mathcal{L}(\Delta) \triangleright P \approxeq^{\mathsf{c}} Q$ as $P \approxeq^{\mathsf{c}}_{\Delta} Q$. We write $P \approxeq^{\mathsf{c}}_{\Delta} Q$ (or $\mathcal{L}(\Delta) \triangleright P \approxeq^{\mathsf{c}} Q$) without recalling the assumption that $P$ and $Q$ are well-typed in $\Delta$. In the case of the plain polyadic $\pi$-calculus, without $\mathsf{i}/\mathsf{o}$ types, we shall also drop the type environment index $\Delta$, and just write $P \approxeq^{\mathsf{c}} Q$; the omitted type environment will always be very simple, and will be clear from the context. The same conventions apply for strong barbed congruence.

Generalising the definitions of ground bisimulation and expansion to the polyadic $\pi$-calculus is straightforward—one just has to make sure that the names received by the processes in input actions have the appropriate types. Adopting the same convention as for barbed congruence, we write $P \approx Q$ and $P \lesssim Q$ for ground bisimulation and expansion on polyadic processes, omitting the type environment index.

We also use ground bisimulation and expansion on processes of the polyadic $\pi$-calculus with $\mathsf{i}/\mathsf{o}$ types, by ignoring the $\mathsf{i}/\mathsf{o}$ types as follows (as an example, we take weak ground bisimulation). If $P, Q$ are such processes, then we set $P \approx Q$ if $P_{\sharp} \approx Q_{\sharp}$, where: $P_{\sharp}$ and $Q_{\sharp}$ are obtained from $P$ and $Q$ by replacing each $\mathsf{i}/\mathsf{o}$ type $I\,T$ (for $I \in \{\mathsf{i}, \mathsf{o}, \sharp\}$) by the channel type $\sharp\, T$; and $P_{\sharp} \approx Q_{\sharp}$ is ground bisimulation of the plain polyadic $\pi$-calculus.

If $P$ and $Q$ are processes of the asynchronous $\pi$-calculus, $\Gamma$ a typing assumption that may contain $\mathsf{i}/\mathsf{o}$ types, and $\Gamma \vdash P, Q$ then $P \approx Q$ implies $P \approxeq^{\mathsf{c}}_{\Gamma} Q$.

A useful law of the asynchronous $\pi$-calculus, for whose validity $\mathsf{i}/\mathsf{o}$ types are necessary, is Lemma B.14 below. In this lemma, we use special processes called links. A link between a name $a$ and a name $b$, written $a \multimap b$, behaves like an ephemeral one-place buffer from $a$ to $b$: it receives names at $a$ and emits them at $b$. Names $a$ and $b$ must be of the same type. The definition is

$$a \multimap b \stackrel{\text{def}}{=} a(\widetilde{u}).\, \overline{b}\langle \widetilde{u} \rangle$$

Lemma B.14 says that if only the output capability of names may be communicated along a name $a$ (this is the hypothesis $a : \mathsf{o}\,\mathsf{o}\, S$; because of subtyping it also covers the case $a : \sharp\, \mathsf{o}\, S$), then sending a known name $b$ along $a$ is the same as sending a fresh name $c$ that is linked to $b$.

**Lemma B.14** *Let $\pi_{\mathrm{a}}^{\mathsf{i}/\mathsf{o},\times,\mu}$ be the polyadic asynchronous $\pi$-calculus with $\mathsf{i}/\mathsf{o}$, and suppose $\pi_{\mathrm{a}}^{\mathsf{i}/\mathsf{o},\times,\mu} \triangleright \Gamma \vdash \overline{a}\langle b \rangle$ and $\pi_{\mathrm{a}}^{\mathsf{i}/\mathsf{o},\times,\mu} \triangleright \Gamma \vdash a : \mathsf{o}\,\mathsf{o}\, S$, for some $S$. Then $\pi_{\mathrm{a}}^{\mathsf{i}/\mathsf{o},\times,\mu}(\Gamma) \triangleright \overline{a}\langle b \rangle \approxeq^{\mathsf{c}} (\boldsymbol{\nu} c : \sharp\, S)\, (\overline{a}\langle c \rangle \mid !c \multimap b)$.*

### B.1.6 The strong replication theorems

Using $\mathsf{i}/\mathsf{o}$ types, it is possible to formulate more powerful versions of the replication Theorem B.10, called the *strong replication theorems* [PS96, San98]. The condition "$a$ occurs free in $P, R, Q$ only in output subject position" of Theorem B.10 is relaxed by requiring, roughly, that the typing of processes $P, R, Q$ only requires the output capability (i.e., a type of the form $\mathsf{o}\, T$) on $a$. This means that these processes, as those of Theorem B.10, may not use $a$ in input; but, by contrast with those of Theorem B.10, these processes may communicate $a$, in which case the constraint on the output capability on $a$ is transmitted to the recipients of $a$.

### B.1.7 Abstractions

In the encodings of the $\lambda$-calculus into the $\pi$-calculus described in the paper, the encoding of a $\lambda$-term is parametric on a name, that is, is a function from names to $\pi$-calculus processes. We call such expressions *abstractions*, and use $F, G$ to range over them. An abstraction with parameters $\widetilde{a}$ and body $P$ is written $(\widetilde{a}).\,P$. An abstraction $(\widetilde{a}).\,P$ is a binder for $\widetilde{a}$ of the same nature as the input prefix $b(\widetilde{a}).\,P$. Indeed, the input $b(\widetilde{a}).\,P$ may be seen as constructed by juxtaposition of the name $b$ and the abstraction $(\widetilde{a}).\,P$. Accordingly, if $F$ is the abstraction $(\widetilde{a}).\,P$, we sometimes write $b\,F$ and $P\{b = F\}$ for the processes $b(\widetilde{a}).\,P$ and $P\{b = (\widetilde{a}).\,P\}$, respectively. We use the following abbreviations for abstractions: if $F$ is $(a).\,P$ then $b\langle\widetilde{c}\rangle\,F$ and $P\{b = (\widetilde{c})\,F\}$ stand for $b(\widetilde{c}, a).\,P$ and $P\{b = (\widetilde{c}, a).\,P\}$ respectively, $Fc$ stands for $P\{c/a\}$ — the actual parameter $c$ substitutes the formal parameter $a$ in the body $P$ of $F$ — and $F\{b = (\widetilde{c}).\,Q\}$ stands for $(a).\,(P\{b = (\widetilde{c}).\,Q\})$.

We extend behavioural equivalences to abstractions; if $\asymp$ is a behavioural equivalence defined on processes, then $F \asymp G$ means that $Fc \asymp Gc$, for all $c$; in typed $\pi$-calculi $F \asymp_{\Gamma;T} G$ means that $Fc \asymp_{\Gamma,c:T} Gc$, for all $c$ such that $\Gamma, c : T \vdash Fc, Gc$ (as usual we omit typing indexes in the plain polyadic $\pi$-calculus)

### B.1.8 The delayed input

In the asynchronous $\pi$-calculus, input prefix is the only syntactic construct for sequentialising process actions. Consider an input

$$a(x).\,(\overline{b}\langle v\rangle \mid \overline{x}\langle w\rangle)$$

It is reasonable that $\overline{x}\langle w\rangle$ should execute after the input $a(x)$ because the input binds $x$, and this creates a dependency between the two prefixes. By contrast, the other output $\overline{b}\langle v\rangle$ has no syntactic dependency on the input. One could therefore argue that the temporal dependency of $\overline{b}\langle v\rangle$ on input $a(x)$ is not justified. Accordingly, one might wish to replace the $\pi$-calculus input prefix $a(x).\,P$ with a *delayed input prefix* $a(x){:}P$ whose SOS rules include:

$$\textsc{Dinp1}\,\frac{}{a(x){:}P \xrightarrow{a\langle b\rangle} P\{b/x\}} \qquad \textsc{Dinp2}\,\frac{P \xrightarrow{\mu} P'}{a(x){:}P \xrightarrow{\mu} a(x){:}P'}\,x \notin \mathrm{n}(\mu)$$
$$\textsc{Dcom1}\,\frac{P \xrightarrow{\overline{a}(b)} P'}{a(x){:}P \xrightarrow{\tau} \boldsymbol{\nu} b\,(P'\{b/x\})} \qquad \textsc{Dcom2}\,\frac{P \xrightarrow{\overline{a}\langle b\rangle} P'}{a(x){:}P \xrightarrow{\tau} P'\{b/x\}}\,b \neq x \tag{56}$$

Rule Dinp1 is analogous to the rule for the ordinary input; Dinp2 allows the continuation of the delayed input to interact with the environment. Rules Dcom1 and Dcom2 allow the continuation to interact with the prefix itself. The delayed prefix is attractive because it allows more parallelism in processes. However, the sequentialisation forced by the ordinary prefix is useful both for expressing many interesting behaviours, and for having simple algebraic laws in axiomatisations. Moreover, in some cases a delayed prefix can be coded up, as we are going to show. For this, we make use of the link processes, introduced in B.1.5. We show how to code a delayed prefix $a(x){:}P$ of the asynchronous $\pi$-calculus, under the assumption that $P$ only possesses the output capability on $x$. This condition on the usage of names received in an input is often used (for instance, in most of the encodings of higher-order communications and of functions in this paper). Consider the following equality:

$$a(x):P = \boldsymbol{\nu} x\ (a(y).\, (!x \multimap y) \mid P) \tag{57}$$

Under the hypothesis that $P$ is asynchronous and possesses only the output capability on $x$, the two processes are behaviourally indistinguishable [MS98]. Lemma B.15 is about this claim, under the additional hypothesis that $P$ cannot export $x$. With this hypothesis, rules (56) completely define the behaviour of the delayed input construct; without it, other rules are needed and the proof of the correctness of (57) is more complex.

**Lemma B.15** *Let $P$ be a process of the asynchronous $\pi$-calculus, and suppose that $x$ appears free in $P$ only in output subject position. Assume that the behaviour of the delayed input construct is defined by rules (56). Then*

$$a(x):P \approx \boldsymbol{\nu} x\ (a(y).\, (!x \multimap y) \mid P)$$

*where $y$ is fresh for $P$ and $x$.*

A similar encoding can be given of a delayed prefix $a(x):P$ under the hypothesis that $P$ only possesses the input capability on $x$: we just invert the direction of the link:

$$\boldsymbol{\nu} y\ (a(y).\, (!y \multimap x) \mid P) \tag{58}$$

However, the correctness of this transformation is more delicate, and requires additional hypotheses on the contexts in which the processes can be used. The context must be composed of processes that are asynchronous, and the context should keep only the output capability on the names that are sent at $a$. These conditions can be nicely formalised using a behavioural equivalence for asynchronous calculi and type systems. We omit the details. Both in (57) and in (58), the replication in front of the link can be eliminated if $x$ is used linearly in $P$.

# C The Higher-Order $\pi$-calculus

Starting from the basic constructs for concurrency of the $\pi$-calculus we move to higher-order, by allowing values built out of processes. We call this higher-order calculus the *Higher-Order $\pi$-calculus* (HO$\pi$).

Passing a process is like passing a parametreless procedure. The recipient of a process can do nothing with it but execute it, possibly several times. Procedures gain great utility if they can be *parametrised* so that, when invoked, some arguments may be supplied. In the same way a process-passing calculus gains power if communication of parametrised processes is allowed. We call parametrised processes *abstractions*. We have already introduced abstractions in the $\pi$-calculus, where we presented them as a convenient syntactic notation for representing the encodings of $\lambda$-terms. The role of abstractions is important in HO$\pi$, as they can be used as values and exchanged in communications.

When an abstraction $(x).\,P$ is applied to an argument $w$ it yields the process $P\{^{w}\!/\!_{x}\}$. Application is the destructor for abstractions. The application of an abstraction $v$ to a value $v'$ is written $v\langle v' : T\rangle$, where $T$ is the type of $v'$. (The type annotation is to ensure that the typing derivation of a process is unique, which will facilitate the encoding of HO$\pi$ into $\pi$-calculus. Alternatively, the annotation could be in the binder of abstractions, as common in $\lambda$-calculi; our choice keeps the syntax of abstraction closer to that of input.) At the level of types, adding parametrisation means adding arrow types, so that an abstraction $(x).\,P$ has type $T \to \diamond$, where $\diamond$ is the type behaviour, that is the type of all processes.

A process, before being communicated, must be converted into a value, a *process value*. A process value is the special case of abstraction in which the argument has unit type. Therefore process values are expressions $(x).\,P$ of type $\mathtt{unit} \to \diamond$. We write `unit` for the unit type, and $\star$ for the unit value.

**Example C.1** *Here are a process $Q$ that is willing to send a process $P$ along a channel $a$ and then continues as $Q'$, and a process $R$ that is willing to receive and execute what $Q$ is sending on $a$:*

$$\begin{array}{rcl} Q & \stackrel{def}{=} & \overline{a}\langle (x).\,P\rangle.\,Q' \\ R & \stackrel{def}{=} & a(y).\,y\langle\star\rangle \end{array}$$

*These processes can interact as follows:*

$$\begin{array}{rl} & Q \mid R \\ \xrightarrow{\tau} & Q' \mid ((x).\,P)\langle\star\rangle \\ \xrightarrow{\tau} & Q' \mid P\{\star/x\} \end{array}$$

Summarising, with respect to the grammar of the $\pi$-calculus, that of HO$\pi$ has the following additional productions: the grammar of values has the production $(x).\,P$ for abstractions; the grammar of processes the production $v\langle w : T\rangle$ for application; that of types the production $T \to \diamond$ for arrow types. But, by contrast with $\pi$-calculus, subject types are not also object types. Both in $v\langle w : T\rangle$ and in $T \to \diamond$, type $T$ is an object type. The operational semantics has an additional rule for when an abstraction meets an application:

$$\text{R-A{\scriptsize PP}}\,\frac{}{((x).\,P)\langle v : T\rangle \xrightarrow{\tau} P\{v/x\}}$$

In this paper, we use extensions of this calculus with unit types, recursive types, and product types.

In HO$\pi$, no expression *reduces* to a value, therefore we need not specify a reduction strategy for value expressions. The right-hand side of an arrow type is always the behaviour type $\diamond$; allowing nesting of arrow types on the right, like in $T_1 \to T_2 \to \diamond$ (which would mean allowing nesting of abstractions such as $(x).\,(y).\,P$), would lead us towards issues of reduction strategies. HO$\pi$ does not have reduction to values because it is conceived for studying and understanding basic issues of processes that may exchange higher-order values; reduction strategies for value expressions is an orthogonal issue (that can be studied with the $\lambda$-calculus).

Barbed congruence is defined on HO$\pi$ as by Definition B.13.

If $P$ is a HO$\pi$ process, and the proof of the derivation of $P \xrightarrow{\tau} P'$ uses rule R-APP, then we write $P \longrightarrow_\beta P'$, and say that $P$ *$\beta$-reduces* to $P'$; $=_\beta$ is the equivalence induced by $\longrightarrow_\beta$.

# D Compiling process-passing into name-passing

The process-passing features of HO$\pi$ can be faithfully coded up using name-passing. In the encoding, the communication of an abstraction $v$ is translated as the communication of a private name which acts as a pointer to (the translation of) $v$ and which the recipient can use to trigger a copy of (the translation of) $v$, with appropriate arguments. For instance a process $\overline{a}\langle(x).\, R\rangle.\, \mathbf{0}$ is translated into the process $\boldsymbol{\nu} y\, (\overline{a}\langle y\rangle \mid !y(x).\, \mathcal{C}[\![R]\!])$; a recipient of the pointer $y$ can use it to activate as many copies of $\mathcal{C}[\![R]\!]$ as needed. However, when translating $\overline{a}\langle v\rangle.\, P$, if $v$ is a name or the unit value, then $v$ is also a $\pi$-calculus value and therefore $v$ can be directly sent along $a$; in this case we say that $v$ is $\pi$-transmittable.

In the translation of an application $v\langle w\rangle$, the function $v$ is located at some fresh name $y$; the function is activated by receiving its argument at $y$. Again, if the argument $w$ is $\pi$-transmittable, then $v$ is applied directly to $w$. The translation of application is very close to that of output, which reveals the similarity between the two constructs.

The compilation modifies types: a name used in HO$\pi$ to exchange processes becomes, in $\pi$-calculus, a name used for exchanging other names. Let HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}}$ the HO$\pi$ calculus in which the type constructs for values are arrow types, recursive types, unit types (in HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}}$ there is also the channel type $\sharp\, T$, needed for typing channels; but channels may not be passed around as values); and let $\pi^{\mathsf{i/o},\mu,\mathtt{unit}}$ be the $\pi$-calculus with $\mathsf{i/o}$ types, recursive types, unit types. The compilation from HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}}$ to $\pi^{\mathsf{i/o},\mu,\mathtt{unit}}$ is defined on types, type environments, values and processes in Table 16. The translation of a process is annotated with a typing environment, and that of a higher-order value with a typing environment and a type. These annotations are used for assigning the appropriate types to the names introduced by the compilation. (Intuitively, reading $\mathcal{C}[\![P]\!]^{\Gamma}$ think that HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash P$, and reading $\mathcal{C}[\![v]\!]^{\Gamma;T}$ think that HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash v\ : T$.) Note that, by the notations for abstraction, an expression $\mathcal{C}[\![(x).\, P]\!]^{\Gamma;T} y$ is the result of applying the abstraction $\mathcal{C}[\![(x).\, P]\!]^{\Gamma;T}$ to name $y$, that is the process $\mathcal{C}[\![P]\!]^{\Gamma,x:T}\{y/x\}$.

**Proposition D.1** *For all $P, v, \Gamma, T$ and $p$ fresh for $\Gamma$, we have:*

1. *HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash P$ iff $\pi^{\mathsf{i/o},\mu,\mathtt{unit}} \triangleright \mathcal{C}[\![\Gamma]\!] \vdash \mathcal{C}[\![P]\!]$;*
2. *HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash v\ : T \rightarrow \diamond$ iff $\pi^{\mathsf{i/o},\mu,\mathtt{unit}} \triangleright \mathcal{C}[\![\Gamma]\!], p\ :\ \mathcal{C}[\![T]\!] \vdash \mathcal{C}[\![v]\!]^{\Gamma;T\rightarrow\diamond} p$.*

**Lemma D.2 (adequacy of $\mathcal{C}$)** *Suppose HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash P$. Then $P \Downarrow_a$ iff $\mathcal{C}[\![P]\!]^{\Gamma} \Downarrow_a$, for all $a$.*

Encoding $\mathcal{C}$ agrees with the behavioural equivalences of HO$\pi$ and $\pi$-calculus. For instance, it is fully abstract for barbed congruence [San92, San97b]. However, stating and proving full abstraction requires more than $\mathsf{i/o}$ types (for instance receptiveness types); we therefore only present some simpler results, that are sufficient for our needs in the paper:

**Lemma D.3** *Suppose HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash P, Q$. If $P =_{\beta} Q$, then $\pi^{\mathsf{i/o},\mu,\mathtt{unit}}(\mathcal{C}[\![\Gamma]\!]) \triangleright \mathcal{C}[\![P]\!]^{\Gamma} \approxeq^{\mathsf{c}} \mathcal{C}[\![Q]\!]^{\Gamma}$.*

**Lemma D.4** *Suppose HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}} \triangleright \Gamma \vdash P, Q$. If $\pi^{\mathsf{i/o},\mu,\mathtt{unit}}(\mathcal{C}[\![\Gamma]\!]) \triangleright \mathcal{C}[\![P]\!]^{\Gamma} \approxeq^{\mathsf{c}} \mathcal{C}[\![Q]\!]^{\Gamma}$, then HO$\pi^{\rightarrow\diamond,\mu,\mathtt{unit}}(\Gamma) \triangleright P \approxeq^{\mathsf{c}} Q$.*

*In this table we abbreviate all expressions $\mathcal{C}[\![E]\!]$ as $[\![E]\!]$:*

*Translation of types:*

$$\begin{array}{rcl@{\qquad}rcl} [\![\sharp\, T]\!] & \stackrel{\mathrm{def}}{=} & \sharp\, [\![T]\!] & [\![T \to \diamond]\!] & \stackrel{\mathrm{def}}{=} & \mathtt{o}\, [\![T]\!] \\ [\![\mathtt{unit}]\!] & \stackrel{\mathrm{def}}{=} & \mathtt{unit} & [\![\mu X\,.\,T]\!] & \stackrel{\mathrm{def}}{=} & \mu X\,.\,[\![T]\!] \end{array}$$

*Translation of type environments:*

$$\begin{array}{rcl} [\![\emptyset]\!] & \stackrel{\mathrm{def}}{=} & \emptyset \\ [\![\Gamma, x \,:\, T]\!] & \stackrel{\mathrm{def}}{=} & [\![\Gamma]\!], x \,:\, [\![T]\!] \end{array}$$

*Translation of higher-order values:*

$$\begin{array}{rcll} [\![(x).\,P]\!]^{\Gamma;T} & \stackrel{\mathrm{def}}{=} & (x).\,[\![P]\!]^{\Gamma,x:S} & \text{if } T = S \to \diamond \\ [\![y]\!]^{\Gamma;T} & \stackrel{\mathrm{def}}{=} & (x).\,\overline{y}\langle x\rangle & \text{if } \Gamma \vdash y \,:\, T \end{array}$$

*Translation of processes:*
(we say that a value $v$ is $\pi$-transmittable if $v$ is a name or $v = \star$; and we assume that $y$ is a fresh name)

$$[\![\overline{x}\langle w\rangle.\,P]\!]^{\Gamma} \stackrel{\mathrm{def}}{=} \begin{cases} \overline{x}\langle w\rangle.\,[\![P]\!]^{\Gamma} & \text{if } w \text{ is } \pi\text{-transmittable} \\ (\boldsymbol{\nu} y : ([\![T]\!])^{\leftarrow\sharp})\,(\overline{x}\langle y\rangle.\,[\![P]\!]^{\Gamma} \mid !y\,[\![w]\!]^{\Gamma;T}) & \\ & \text{if } w \text{ is not } \pi\text{-transmittable} \end{cases}$$
$$\text{if } \Gamma \vdash x \,:\, \sharp\, T$$

$$[\![v\langle w \,:\, T\rangle]\!]^{\Gamma} \stackrel{\mathrm{def}}{=} \begin{cases} [\![v]\!]^{\Gamma;T\to\diamond} w & \text{if } w \text{ is } \pi\text{-transmittable} \\ (\boldsymbol{\nu} y : [\![T]\!]^{\leftarrow\sharp})\,([\![v]\!]^{\Gamma;T\to\diamond} y \mid !y\,[\![w]\!]^{\Gamma;T}) & \\ & \text{if } w \text{ is not } \pi\text{-transmittable} \end{cases}$$

$$\begin{array}{rcl@{\qquad}rcl} [\![z(x).\,P]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & z\,[\![(x).\,P]\!]^{\Gamma;T} \quad \text{if } \Gamma \vdash z \,:\, \sharp\, T & & & \\ [\![\tau.\,P]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & \tau.\,[\![P]\!]^{\Gamma} & [\![\mathbf{0}]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & \mathbf{0} \\ [\![P \mid Q]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & [\![P]\!]^{\Gamma} \mid [\![Q]\!]^{\Gamma} & [\![P + Q]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & [\![P]\!]^{\Gamma} + [\![Q]\!]^{\Gamma} \\ [\![(\boldsymbol{\nu} x : T)\,P]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu} x : [\![T]\!])\,[\![P]\!]^{\Gamma,x:T} & [\![!P]\!]^{\Gamma} & \stackrel{\mathrm{def}}{=} & ![\![P]\!]^{\Gamma} \end{array}$$

Table 16: The compilation $\mathcal{C}$ of $\mathrm{HO}\pi^{\to\diamond,\mu,\mathtt{unit}}$ into the $\pi$-calculus

### D.0.9 Extensions

The compilation and the above results can be extended to calculi with richer type structures, for instance with products and linearity. If we have linear types, so that we know that the argument $w$ of expressions $\overline{x}\langle w\rangle. P$ and $v\langle w : T\rangle$ is used at most once, then in the clauses of Table 16 no replications is needed before $y\,\mathcal{C}[\![w]\!]^{\Gamma;T}$.

# References


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

[Abr89] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[ABV94] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, 1994.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer Verlag, 1996.

[ACS96] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous $\pi$-calculus. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[AFM$^+$95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need $\lambda$-calculus. In *Proc. 22th POPL*. ACM Press, 1995.

[AKH92] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica*, 29:737–760, 1992.

[AO93] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[App92] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Bar77] H. Barendregt. The type free lambda calculus. In J. Barwise, editor, *Handbook of Mathematical Logic*, Studies in Logic 90, pages 1092–1132. North Holland, 1977.

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

[Ber78] G. Berry. Séquentialité de l'évaluation formelle des $\lambda$-expressions. In B. Robinet, editor, *Program Transformations, Proc. 3rd Int. Coll. on Programming*, pages 67–80, 1978.

[Ber81] G. Berry. Some syntactic and categorical constructions of lambda calculus models. Rapport de Recherche 80, Institute National de Recherche en Informatique et en Automatique (INRIA), 1981.

[BIM88] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be traced: preliminary report. In *Conference Record of the 15th ACM Symposium on Principle of Programming Languages (POPL)*, pages 229–239. ACM Press, 1988.

[BK98] I. Bethke and J.W. Klop. Sequentiality in the lambda-calculus and combinatory logic. Draft, 1998.

[BL95] G. Boudol and C. Laneve. $\lambda$-calculus, multiplicities and the $\pi$-calculus. Technical Report RR-2581, INRIA-Sophia Antipolis, 1995. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[BL96] G. Boudol and C. Laneve. The discriminating power of the $\lambda$-calculus with multiplicities. *Information and Computation*, 126(1):83–102, 1996.

[Blo90] B. Bloom. Can LCF be topped? Flat lattice models of typed $\lambda$-calculus. *Information and Computation*, 87(1/2):263–300, 1990.

[BO95] S. Brock and G. Ostheimer. Process semantics of graph reduction. In I. Lee and S.A. Smolka, editors, *Proc. CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*, pages 471–485, Philadelphia, Pennsylvania, 1995. Springer Verlag.

[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.

[Bou92] G. Boudol. Asynchrony and the $\pi$-calculus. Technical Report RR-1702, INRIA-Sophia Antipolis, 1992.

[Bou94] G. Boudol. Some chemical abstract machines. In *Proc. REX Summer School/Symposium 1993*, volume 803 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Bou97] G. Boudol. The pi-calculus in direct style. In *Proc. 24th POPL*. ACM Press, 1997.

[CF58] H.B . Curry and R. Feys. *Combinatory Logic(Vol I)*. North Holland, Amsterdam, 1958.

[Chu32] A. Church. A set of postulates for the foundations of logic. *Ann. of Math.*, 33:346–366, 1932.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[DCTU97] M. Dezani-Ciancaglini, J. Tiuryn, and P. Urzyczyn. Discrimination by parallel observers. In *Proceedings, Twelth Annual IEEE Symposium on Logic in Computer Science*, pages 396–407, Warsaw, Poland, 1997. IEEE Computer Society Press.

[DH87] R. De Nicola and M. Hennessy. CCS without $\tau$'s. In *Proc. Tapsoft '87*, volume 249 of *Lecture Notes in Computer Science*, pages 138–152, 1987.

[DH92] O. Danvy and J. Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, Bordeaux, France, 1992.

[DS85] R. De Simone. Higher level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[Fis72] M. J. Fischer. Lambda-calculus schemata. In *Proc. ACM conf. on Proving Assertions about Programs*, pages 104–109, 1972.

[Fis93] M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6:233–248, 1993.

[FWT92] D. P. Friedman, M. Wand, and Haynes C. T. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., 1992.

[Gor79] M.J.C. Gordon. *The denotational description of programming languages.* Springer Verlag, 1979.

[Gor95] A. D. Gordon. Bisimilarity as a theory of functional programming: mini-course. Notes Series BRICS-NS-95-3, BRICS, Department of Computer Science, University of Aarhus, July 1995.

[Gun92] C. A. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, Moss, 1992.

[GV92] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[HB77] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992. IFIP, 1977.

[HBG+73] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor induction and meta-evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 153–168. ACM, 1973.

[HD94] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proc. 21st POPL*, pages 458–471. ACM Press, 1994.

[HD97] J. Hatcliff and O. Danvy. Thunks and the $\lambda$-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

[HDM93] R. Harper, F. B. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, Mass., 1988.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial intelligence*, 8(3):323–364, 1977.

[Hin97] J.R. Hindley. *Basic simple type theory.* Cambridge University Press, 1997.

[HL93] R. Harper and M. Lillibridge. Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, 6:361–380, 1993.

[How96] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

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

[HT91] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communications. In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *ECOOP '91 Workshop on Object Based Concurrent Programming,* Geneva, Switzerland, 1991, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer Verlag, 1991.

[Ing61] P. Z. Ingerman. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[Jef93] A. Jeffrey. A chemical abstract machine for graph reduction. In *Proc. Ninth International Conference on the Mathematical Foundations of Programming Semantics (MFPS'93)*, volume 802 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[Kob98] N. Kobayashi. A partially deadlock-free typed process calculus. *TOPLAS*, 20(2):436–482, 1998. A preliminary version in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 128–139, 1997.

[KPT96] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL.* ACM Press, 1996.

[KS82] J.R. Kennaway and M.R. Sleep. Expressions as processes. In *ACM Conference on LISP and Functional Programming*, pages 21–28. ACM, 1982.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th POPL.* ACM Press, 1993.

[Let91] L. Leth. *Functional Programs as Reconfigurable Networks of Communicating Processes.* Ph.D. thesis, Imperial College, University of London, 1991.

[Lév76] J.J. Lévy. An algebraic interpretation of the $\lambda\beta\kappa$-calculus; and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97–114, 1976.

[Lon83] G. Longo. Set theoretical models of lambda calculus: Theory, expansions and isomorphisms. *Annales of Pure and Applied Logic*, 24:153–188, 1983.

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

[Mil91] R. Milner. The polyadic $\pi$-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[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.

[MS98] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *25th ICALP*, volume 1443 of *Lecture Notes in Computer Science.* Springer Verlag, 1998.

[Mur92] C. Murthy. A computational analysis of Girard's translation and LC. In *7th LICS Conf.* IEEE Computer Society Press., 1992.

[MW85] A. R. Meyer and M. Wand. Continuation semantics in typed lambda-calculi. In Rohit Parikh, editor, *Proceedings of the Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224, Brooklyn, NY, 1985. Springer Verlag.

[Nie96] J. Niehren. Functional computation as concurrent computation. In *Proc. 23th POPL.* ACM Press, 1996.

[OD93] G.K. Ostheimer and A.J.T. Davie. Pi-calculus characterizations of some practical lambda-calculus reduction strategies. CS 93/14, University of St Andrews, Scotland, October 1993.

[OLT94] C. Okasaki, P. Lee, and D Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, January 1994.

[Ong88] L. Ong. *The Lazy Lambda Calculus: an Investigation into the Foundations of Functional Programming.* PhD thesis, University of London, 1988. Also Prize Fellowship Dissertation, Trinity College, Cambridge, 256 pp.

[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.

[Pit97] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

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

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

[PS92] S. Purushothaman and J. Seaman. An adequate operational semantics for sharing in lazy evaluation. In Bernd Krieg-Brückner, editor, *ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 435–450. Springer Verlag, 1992.

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.

[Rey72] J. C. Reynolds. Definitional interpreters for higher order programming languages. *ACM Conference Proceedings*, pages 717–740, 1972.

[Rey93] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6:233–248, 1993.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[San94] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

[San95a] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[San95b] D. Sangiorgi. Lévy-Longo Trees and Böhm Trees from encodings of $\lambda$-calculus into $\pi$-calculus. Notes, 1995.

[San96] D. Sangiorgi. $\pi$-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.

[San97a] D. Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In *Proc. 24th POPL*, pages 428–441. ACM Press, 1997.

[San97b] D. Sangiorgi. The name discipline of receptiveness. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997. To appear in TCS.

[San98] D. Sangiorgi. An interpretation of typed objects into typed $\pi$-calculus. *Information and Computation*, 143(1):34–73, 1998.

[Sch86] D. A. Schmidt. *Denotational Semantics–A methodology for language development*. Allyn and Bacon, 1986.

[SF93] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6:289–360, 1993.

[SM92] D. Sangiorgi and R. Milner. The problem of "Weak Bisimulation up to". In W.R. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer Verlag, 1992.

[SW74] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report Technical Monograph PRG-11, Oxford University Computer Laboratory, 1974.

[Ten91] R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.

[Thi97] H. Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

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

[Tur96] N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[VH93] V.T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic $\pi$-calculus. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.

[Wad71] C. P. Wadsworth. *Semantics and pragmatics of the lambda calculus.* PhD thesis, University of Oxford, 1971.

[Wal90] D. Walker. Bisimulation and divergence. *Information and Computation*, 85(2):202–241, 1990.

[Yos93] N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. In *Proc. of FPCA'93, Functional Programming and Computer Architecture*, pages 243–252, 1993.