Math. Struct. in Comp. Science (1999), *vol.* 9, *pp.* 367–401. Printed in the United Kingdom

From λ to π; or, Rediscovering continuations

DAVIDE SANGIORGI

INRIA-Sophia Antipolis, 2004 Rue des Lucioles, B.P. 93,
06902 Sophia Antipolis, France.
Email: `davide.sangiorgi@inria.fr.`

Received 7 June 1998, revised 30 March 1999

We study the relationship between the encodings of the λ-calculus into π-calculus, the Continuation Passing Style (CPS) transforms, and the compilation of the Higher-Order π-calculus (HOπ) into π-calculus. We factorise the π-calculus encodings of (untyped as well as simply-typed) call-by-name and call-by-value λ-calculus into three steps: a CPS transform, the inclusion of CPS terms into HOπ and the compilation from HOπ to π-calculus. The factorisations are used both to derive the encodings and to prove their correctness.

1. Introduction

The λ-calculus describes *sequential* computations. The π-calculus naturally describes *parallel* computations. In the λ-calculus, computation is function application. In the π-calculus computation is process interaction: two π-calculus processes that know a given name can use it to interact with each other; names themselves may be exchanged in communications. By viewing function application as a special form of process interaction, the λ-calculus functions can be represented as π-calculus processes. Encodings of call-by-name and call-by-value λ-calculus evaluation strategies into π-calculus have been given by Milner (1992).

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* (CPS functions). Continuations are widely used: for programming, as an implementation technique and for giving denotational semantics. There is a very large literature of functional programming study in 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 λ-calculus in Plotkin's seminal paper Plotkin (1975).

The π-calculus is the paradigmatic *first-order* (or *name-passing*) process calculus. The *Higher-Order π-calculus* (HOπ) is a process calculus whose operators are the same as the π-calculus, but the objects exchanged in communications are built out of processes, rather than names. HOπ is therefore a *higher-order* (or *process-passing*) process calculus. Despite this difference, HOπ can be faithfully compiled down into π-calculus (Sangiorgi 1992a; Sangiorgi 1998a).

In this paper we study the relationship between the encodings of the λ-calculus into

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

Fig. 1. The derivatives of the π-calculus encodings of λN and λV.

π-calculus, the CPS transforms, and the compilation of HOπ into π-calculus. We factorise the π-calculus encodings of call-by-name and call-by-value λ-calculus into three steps: a CPS transform, the inclusion of CPS terms into HOπ and the compilation from HOπ to π-calculus. This programme is summarised in Figure 1, where: λN and λV are the call-by-name and call-by-value λ-calculi; $\mathscr{C}_{\mathrm{V}}$ and $\mathscr{C}_{\mathrm{N}}$ are the call-by-value and call-by-name CPS transforms of Plotkin (1975); CPS$_{\mathrm{V}}$ and CPS$_{\mathrm{N}}$ are languages of CPS λ-terms, which are the target languages of the two CPS transforms; $\mathscr{H}$ is the injection of these CPS languages into HOπ; and $\mathscr{C}$ is the compilation from HOπ to π-calculus. We work out a similar factorisation using uniform CPS transforms, which only differ in the clauses for application – the clauses for λ-abstraction and variables are the same. We thus derive π-calculus encodings which have the same uniformity properties, and which can be easily modified so as to obtain an encoding of the *call-by-need* strategy.

We then apply the same schema of Figure 1 to (simply-)*typed* λ-calculi. For this, we extend the translations of terms to translations of types. Understanding how λ-calculus types are transformed by the π-calculus encodings is not entirely obvious. In λ-calculi, types are assigned to terms, and provide an abstract view of their behaviour. In contrast, in the type systems for the π-calculus, types are assigned to names and hence reveal very little about the behavioural properties of the processes.

Several CPS transforms appear in the literature, and the schema of Figure 1 can probably be repeated for many of them. The CPS transforms of Plotkin (1975) are those that, in our view, yield the simplest and most robust π-calculus encodings. The π-calculus encodings and their correctness results that we present are not new (the encodings and the results about typed calculi have not been presented elsewhere, but they are hinted at in Pierce and Sangiorgi (1996)). What is new is the way in which the encodings and their correctness are derived.

We use CPS transforms to obtain π-calculus encodings of λ-calculi and prove their correctness. Sometimes it is also possible to go the other way round to exploit the factorisation to understand, and reason about, the CPS transforms. For instance, the adequacy of the CPS transforms can be derived from that of the π-calculus encodings.

Overview of paper. Sections 2–4 contain background material on the π-calculus, HOπ, and the compilation of HOπ into π-calculus. In Section 5 we review the syntax, the

Table 1. *The π-calculus*

Grammars:			
	$a, b, \ldots x, y, z$		*Names*
			Values
	v, w	$:= a$	names
			Processes
	P, Q, R	$:= 0$	nil
		$\mid\ P \mid P$	parallel
		$\mid\ \nu a\, P$	restriction
		$\mid\ !P$	replication
		$\mid\ a(x).\, P$	input
		$\mid\ \overline{a}v$	output

reduction rules, and the call-by-name and call-by-value reduction strategies for the untyped λ-calculus. In Sections 6 and 7 we develop Figure 1, for untyped λ-calculi, using the CPS transforms of Plotkin (1975). In Section 8 we do the same, but starting from uniform CPS transforms. In Section 9 we study typed λ-calculi.

2. The π-calculus

Syntax. The grammar of the π-calculus is given in Table 1. **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 $\overline{a}v$ emits value v at a. A parallel composition $P_1 \mid P_2$ is to run two processes in parallel. A restriction $\nu a\, P$ makes name a local, or private, to P. A replicated process $!P$ stands for a countable infinite number of copies of P. We use an *asynchronous* subset of the original π-calculus (Milner 1992), where outputs have no continuation and there are no operators of summation and matching, because it is sufficient for encoding the λ-calculus; in this paper 'π-calculus' will mean this calculus.

We use σ to range over substitutions; for any expression E, we write $E\sigma$ for the result of applying σ 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.

An input prefix $a(b).\, P$ and a restriction $\nu b\, P$ are binders for name b, and give rise in the expected way to the definitions of free and bound names, and the definition of α-conversion. Symbol '=' denotes equality up to α-conversion. We identify α-equivalent processes. In statements, we always assume that bound names of the expressions of the statements (processes, actions, *etc.*) are *fresh*, that is, they are different from the other bound and free names of the expressions in the statement; we say that a name a is *fresh for an expression* E to mean that a does not appear E. We write fn(E) and bn(E) for, respectively, the free and bound names of E, and n(E) for all names in E (free and bound). A *context* is a process expression with a single occurrence of a hole [·] in it. We abbreviate

$\nu a\, \nu b\, P$ as $(\nu a, b)\, P$. In $\nu b\,(\overline{a}b \mid b(y).\, Q)$ and $\nu b\,(\overline{a}b \mid !b(y).\, Q)$, due to the restriction on b, the output at a must be consumed before the inputs at b; to highlight the ordering, we abbreviate these expressions as $\overline{a}(b).\, b(y).\, Q$ and $\overline{a}(b).\, !b(y).\, Q$, respectively. We use a tilde to represent tuples of expressions. With some abuse of notation, we sometimes view $\tilde{E}$ both as a tuple and a set. We extend notations to tuples componentwise.

Reduction semantics. Following Milner (1991), the one-step reduction relation $\longrightarrow$ of the calculus exploits the auxiliary relation $\equiv$ of structural congruence to bring the participants of a potential communication into contiguous positions. The *structural congruence relation* $\equiv$ is the least congruence on processes that is closed under these rules:

1 Abelian monoid laws for parallel composition:
$P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, $P \mid \mathbf{0} \equiv P$;
2 $\nu p\, \mathbf{0} \equiv \mathbf{0}$, $\nu p\, \nu q\, P \equiv \nu q\, \nu p\, P$;
3 $(\nu p\, P) \mid Q \equiv \nu p\,(P \mid Q)$, if p not free in Q;
4 $!P \equiv P \mid !P$.

The one-step reduction $P \longrightarrow Q$ is the least relation closed under these rules:

$$\text{R-C{\small OM}}\frac{}{\overline{a}v \mid a(x).\, Q \longrightarrow P \mid Q\{v/x\}} \qquad \text{R-P{\small AR}}\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

$$\text{R-R{\small ES}}\frac{P \longrightarrow P'}{\nu a\, P \longrightarrow \nu a\, P'} \qquad \text{R-E{\small QV}}\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}.$$

For any name a, the observation predicate $\downarrow_a$ denotes the possibility for a process of immediately sending a message along a. Thus, $P \downarrow_a$ holds if P has a particle $\overline{a}v$ that is not underneath an input prefix, and not in the scope of a restriction on a. We write $\Longrightarrow$ for the reflexive and transitive closure of $\longrightarrow$, and $P \Downarrow_a$ if there is P' such that $P \Longrightarrow P' \downarrow_a$. We also write $P \Downarrow$ to mean that $P \Downarrow_a$ for some a.

Behavioural equivalences. We define behavioural equality using the notion of *barbed congruence* (Milner and Sangiorgi 1992). Barbed congruence can be defined on any calculus possessing a reduction relation $\longrightarrow$, and observability predicates $\downarrow_a$ for each channel a. Barbed congruence is the congruence induced by barbed bisimulation. The latter equates processes that can match each other's reductions and, at each step, are observable on the same channels. We define barbed congruence on a generic process calculus $\mathscr{L}$ that has reduction and observation relations as above. An $\mathscr{L}$ *relation* is a relation on the processes of $\mathscr{L}$.

Definition 2.1. (Strong barbed bisimilarity and congruence) An $\mathscr{L}$ relation $\mathscr{R}$ is an $\mathscr{L}$ *strong barbed bisimulation* if $P \mathrel{\mathscr{R}} Q$ implies

1 if $P \longrightarrow P'$, there is Q' such that $Q \longrightarrow Q'$ and $P' \mathrel{\mathscr{R}} Q'$,
2 for each channel a, $P \downarrow_a$ iff $Q \downarrow_a$,

and the converse of (1) on the transitions from Q. Two $\mathscr{L}$ processes P and Q are *strongly barbed bisimilar* (*in* $\mathscr{L}$), written $\mathscr{L} \rhd P \overset{\cdot}{\sim} Q$, if $P \mathrel{\mathscr{R}} Q$ for some $\mathscr{L}$ strong barbed bisimulation $\mathscr{R}$.

Two $\mathscr{L}$ processes P, Q are *strongly barbed congruent* (*in* $\mathscr{L}$), written $\mathscr{L} \rhd P \sim Q$, if, for each context C in $\mathscr{L}$, we have $\mathscr{L} \rhd C[P] \dot{\sim} C[Q]$.

When there is no ambiguity on what the calculus $\mathscr{L}$ is, we may abbreviate $\mathscr{L} \rhd P \dot{\sim} Q$ as $P \dot{\sim} Q$, and $\mathscr{L} \rhd P \sim Q$ as $P \sim Q$. The *weak* versions of barbed bisimilarity and congruence are defined by replacing the strong transitions $\longrightarrow$ with the weak transitions $\Longrightarrow$, and the predicates $\downarrow_a$ with $\Downarrow_a$. We write $\approx$ for weak barbed congruence; we sometimes just call it *barbed congruence*. Weak barbed congruence is the semantic equality we are mainly interested in.

2.1. *Extensions and types*

Some conventions and notation for typed calculi. In typed π-calculi, *type environments* are finite assignment of types to names. Γ, Δ range over type environments; S, T over types. A typing judgement $\Gamma \vdash P$ asserts that process P is well-typed under the type assumptions Γ, and $\Gamma \vdash v : T$ that value v has type T under the type assumptions Γ. (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 π-calculus in Section 3.) When we deal with several typed calculi, to avoid ambiguity, we sometimes add the name of the calculus to typing judgements; thus we write $\mathscr{L} \rhd \Gamma \vdash P$ to mean that the typing judgement $\Gamma \vdash P$ holds in the calculus $\mathscr{L}$.

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 π-calculus, subject types are also object types; in the Higher-Order π-calculus of Section 3, by contrast, object and subject types are distinct. In typed calculi, a restricted name is annotated with a type, as in $(\boldsymbol{\nu} a : T)P$, and $\bar{b}(a : T).P$; the type T is a subject type, which is omitted when not important. A type environment Γ is *closed* if Γ is an assignment of subject types to names. We write $\tilde{x} : T$ to mean that each $x \in \tilde{x}$ has type T; and $\Gamma \vdash P, Q$ to mean that both $\Gamma \vdash P$ and $\Gamma \vdash Q$ hold. Substitutions map names onto values of the same type.

Polyadicity. In the polyadic π-calculus, tuples of names can be communicated. For this, the following production is added to the grammar of values:

$$\begin{array}{rcl} & & \textit{Values} \\ v & := \langle \tilde{v} \rangle & \text{tuples.} \end{array}$$

Having added a constructor for values, we need a corresponding destructor. A destructor can be added as a new process construct. In the case of tuples, however, it is convenient to decompose a value by means of pattern matching in the input prefix, thus adopting the polyadic form of input $a(\tilde{x}).P$.

In an untyped polyadic π-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 π-calculus* the calculus with channel types, recursive types, and product types (Milner 1991; Vasconcelos and Honda 1993; Pierce and Sangiorgi 1996; Turner 1996). Channel types are the only subject types, and are of the form $\sharp T$; product types are of the form $T_1 \times \ldots \times T_n$ for $n \geqslant 0$; recursive types are of the form $\mu X.S$, where X is a type variable and X must be guarded in S, that is, underneath at least a type construct different from recursion. For instance $T \times S$ is the type of a pair, whose first component has type T and the second type S; a name a with type $\sharp(T_1 \times \ldots \times T_n)$ may only carry n-tuples 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 π-calculus generalise to the polyadic π-calculus in the obvious way.

i/o types. Frequently in π-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 the reasons why i/o types are useful is that they give rise to subtyping: type annotation i gives covariance, 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 $\sharp T < \mathsf{o}T$.

We call $\sharp$, i,o the i/o *tags*, and $\sharp T$, $\mathsf{i}\ T$ and $\mathsf{o}T$ the i/o *types*. For readability, we sometimes use brackets with i/o types, as in $\mathsf{o}(T)$. We also occasionally use these notations for i/o types:

— $(T)^-$ is the type obtained from T by cancelling the outermost i/o tag.
— $(T)^{\leftarrow\sharp}$ is the type obtained by replacing the outermost i/o tag in T with $\sharp$.
— For $I \in \{\sharp, \mathsf{o}, \mathsf{i}\}$, we write $T \frown IS$ 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.

Linearity. Further refinements of the i/o types are *linear types* (Kobayashi *et al.* 1996). We do not present this type system here, as it is 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.

Types and behavioural equivalences. 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 (Γ/Δ)-context is a context that, when filled in with a process obeying typing Δ, becomes a process obeying typing Γ. Below is the typed version of barbed congruence for a calculus $\mathscr{L}$.

Definition 2.2. (Strong barbed congruence) Let Δ be a typing, with $\mathscr{L} \triangleright \Delta \vdash P, Q$. We say that processes P, Q are *strongly barbed congruent (in $\mathscr{L}$) at* Δ, written $\mathscr{L}(\Delta) \triangleright P \sim Q$, if, for each closed type environment Γ and (Γ/Δ)-context C, we have $\mathscr{L} \triangleright C[P] \stackrel{\centerdot}{\sim} c[Q]$.

Again, *weak* barbed congruence at Δ is defined by replacing $\stackrel{\centerdot}{\sim}$ with $\stackrel{\centerdot}{\approx}$, and $\sim$ with $\approx$ in Definition 2.2. When there is no ambiguity on what the calculus $\mathscr{L}$ is, we may drop $\mathscr{L}$, and abbreviate $\mathscr{L}(\Delta) \triangleright P \approx Q$ as $P \approx_{\triangle} Q$. We write $P \approx_{\triangle} Q$ (or $\mathscr{L}(\Delta) \triangleright P \approx Q$) without recalling the assumption that P and Q are well-typed in Δ. We also omit Δ if it is clear, and just write $P \approx Q$. The same conventions apply to strong barbed congruence.

Here are some simple facts about behavioural equivalences of typed π-calculi that we shall use.

Lemma 2.3.

1 $\equiv \subseteq \sim_{\Gamma} \subseteq \approx_{\Gamma}$;
2 $(\nu x : T)(x(y).P \mid \bar{x}v) \approx_{\Gamma} (\nu x : T)(P\{v/y\})$;
3 $(\nu x : T)(!x(y).P \mid Q) \sim_{\Gamma} Q$, if x not free in Q.

Abstractions. In the encodings of the λ-calculus into the π-calculus described in the paper, the encoding of a λ-term is parametric on a name, that is, it is a function from names to π-calculus processes. We call such expressions *abstractions*, and use F, G to range over them. An abstraction with parameters $\tilde{a}$ and body P is written $(\tilde{a}).P$. An abstraction $(\tilde{a}).P$ is a binder for $\tilde{a}$ of the same nature as the input prefix $b(\tilde{a}).P$. Indeed, the input $b(\tilde{a}).P$ may be seen as constructed by juxtaposition of the name b and the abstraction $(\tilde{a}).P$. Accordingly, if F is the abstraction $(\tilde{a}).P$, we sometimes write bF for the process $b(\tilde{a}).P$. We use the following abbreviations for abstractions: if F is $(a).P$, then $b(\tilde{c})F$ stands for $b(\tilde{c}, a).P$, and F_c stands for $P\{c/a\}$ – the actual parameter c substitutes the formal parameter a in the body P of F. We extend typed barbed congruence to (monadic) abstractions; $F \approx_{\Gamma;T} G$ means that $Fc \approx_{\Gamma,c:T} Gc$, for all c such that $\Gamma, c : T \vdash Fc, Gc$.

3. The Higher-Order π-calculus

Starting from the constructs for concurrency of the π-calculus, we move to higher-order by allowing values built out of processes. We thus obtain the *Higher-Order π-calculus* (HOπ). As in the untyped π-calculus, names are the only values that are exchanged among processes, so in the untyped HOπ *abstractions*, that is, parametrised processes, are the only values. We have already introduced abstractions in the π-calculus, where we presented them as a convenient syntactic notation for representing the encodings of λ-terms. The role of abstractions is important in HOπ, 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 HOπ into π-calculus. Alternatively, the annotation could be in the binder of abstractions, as is common in λ-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.

Summarising, with respect to the grammar of the π-calculus, that of HOπ 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; in typed HOπ, the grammar of object types always include the production $T \to \diamond$ for arrow types. But, by contrast with π-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:

$$R\text{-}\textsc{App}\,\frac{}{((x).\,P)\langle v : T\rangle \longrightarrow P\{v/x\}}.$$

If P is a HOπ process and the proof of the derivation of $P \longrightarrow P'$ uses rule R-App, then we write $P \longrightarrow_\beta P'$, and say that P *β-reduces* to P'; $=_\beta$ is the equivalence induced by $\longrightarrow_\beta$. Barbed congruence is defined on HOπ as in Definition 2.2.

Remark 3.1. In HOπ, 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, as 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. HOπ 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 (which can be studied in the λ-calculus).

4. Compiling process-passing into name-passing

The process-passing features of HOπ can be faithfully coded up using name-passing. We show this for HO$\pi^{\to\diamond,\mu,\mathrm{unit}}$, the HO$\pi$ calculus in which the object types are arrow types, recursive types, unit type (in HO$\pi^{\to\diamond,\mu,\mathrm{unit}}$ there is also the channel type $\sharp T$, needed for typing channels; but channels may not be passed around as values); the target language is $\pi^{\mathsf{i/o},\mu,\mathrm{unit}}$, the π-calculus with $\mathsf{i/o}$ types, recursive types, unit type.

In the encoding, the communication of an abstraction v is translated as the communication of a private name that 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, if v is $(x).\,R$, process $\bar{a}\,v$ is translated into the process $\nu y(\bar{a}y \mid !y(x).\,\mathscr{C}[\![R]\!])$; a recipient of the pointer y can use it to activate as many copies of $\mathscr{C}[\![\mathscr{R}]\!]$ as needed. However, when translating $\bar{a}\,v$, if v is a name or the unit value, then v is also a π-calculus value and therefore v can be directly sent along a; in this case we say that v is π-transmittable. Output and application are translated in a similar way, which reveals the similarity between the two constructs. The compilation modifies types: a channel used in HOπ to exchange processes becomes, in π-calculus, a channel used for exchanging other names.

The compilation is defined on types, type environments, values and processes in Table 2. We only translate well-typed expressions. The translation of an expression is annotated with a typing environment and (for higher-order values) a type under which that expression is well-typed. Thus $\mathscr{C}[\![\mathscr{P}]\!]^{\Gamma}$ is defined if $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash P$, and $\mathscr{C}[\![v]\!]^{\Gamma;T}$ if $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash v : T$. These annotations are used for assigning the appropriate types to the names introduced by the compilation. Note that, by the notations for abstraction, an expression $\mathscr{C}[\![(x).\,P]\!]^{\Gamma;T} y$ is the result of applying the abstraction $\mathscr{C}[\![(x).\,P]\!]^{\Gamma;T}$ to name y, that is, the process $\mathscr{C}[\![P]\!]^{\Gamma,x:T}\{y/x\}$.

Proposition 4.1. For all P, v, Γ, T and a fresh for Γ, we have
1 $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash P$ implies $\pi^{\mathsf{i/o},\mu,\mathrm{unit}} \triangleright \mathscr{C}[\![\Gamma]\!] \vdash \mathscr{C}[\![P]\!]$
2 $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash v : T \to \diamond$ implies $\pi^{\mathsf{i/o},\mu,\mathrm{unit}} \triangleright \mathscr{C}[\![\Gamma]\!], a : \mathscr{C}[\![T]\!] \vdash C[\![v]\!]^{\Gamma;T\to\diamond} a$.

Lemma 4.2. (Adequacy of $\mathscr{C}$) Suppose $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash P$. Then $P \Downarrow_a$ iff $\mathscr{C}[\![P]\!]^{\Gamma} \Downarrow_a$, for all a.

Encoding $\mathscr{C}$ agrees with the behavioural equivalences of $\mathrm{HO}\pi$ and π-calculus. Full abstraction for similar encodings are studied in Sangiorgi (1992a), Sangiorgi (1998a); here we report two (much simpler) results that are sufficient for our needs in the paper:

Lemma 4.3. Suppose $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash P, Q$. If $P =_\beta Q$, then $\pi^{\mathsf{i/o},\mu,\mathrm{unit}}(\mathscr{C}[\![\Gamma]\!]) \triangleright \mathscr{C}[\![P]\!]^{\Gamma} \approx \mathscr{C}[\![Q]\!]^{\Gamma}$.

(In the above lemma, $\mathsf{i/o}$ types are necessary; the lemma is not true if all types $\mathsf{i}T$ and $\mathsf{o}T$ are replaced by the less informative channel type $\sharp T$.)

Lemma 4.4. Suppose $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}} \triangleright \Gamma \vdash P, Q$. If $\pi^{\mathsf{i/o},\mu,\mathrm{unit}}(\mathscr{C}[\![\Gamma]\!]) \triangleright \mathscr{C}[\![P]\!]^{\Gamma} \approx \mathscr{C}[\![Q]\!]^{\Gamma}$, then $\mathrm{HO}\pi^{\to\diamond,\mu,\mathrm{unit}}(\Gamma) \triangleright P \approx Q$.

4.1. *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 $\bar{x}\,w$ and $v\langle w : T\rangle$ is used at most once, then in the clauses of Table 2 no replications are needed before $y\,\mathscr{C}[\![w]\!]^{\Gamma;T}$. A linear type of $\mathrm{HO}\pi$ can be translated into a linear type of π-calculus. However, the linearity information can be ignored in the translation, without affecting the results of Lemmas 4.3 and 4.4. For this reason, in this paper we shall not use linear types in the π-calculus.

5. The λ-calculus

The basic operators of the λ-calculus, in its pure form, are λ-abstraction and application. Letting x and y range over the set of λ-calculus variables, the set Λ of λ-*terms* is defined by the grammar

$$M := x \mid \lambda x.\, M \mid M_1 M_2.$$

We omit the definitions of α-conversion, free variable, substitution, *etc.* We identify α-convertible terms, and therefore write $M = N$ if M and N are α-convertible. A λ-term is *closed* if it contains no free variables. The set of free variables of a term M is written

Table 2. *The compilation of HOπ into π-calculus (In this table we abbreviate all expressions $\mathscr{C}[\![E]\!]$ as $[\![E]\!]$.)*

Translation of types:

$$[\![\sharp T]\!] \stackrel{\text{def}}{=} \sharp[\![T]\!] \qquad [\![T \to \diamond]\!] \stackrel{\text{def}}{=} \mathsf{o}[\![T]\!]$$

$$[\![\text{unit}]\!] \stackrel{\text{def}}{=} \text{unit} \qquad [\![\mu X.\, T]\!] \stackrel{\text{def}}{=} \mu X.\, [\![T]\!] \qquad [\![X]\!] \stackrel{\text{def}}{=} X$$

Translation of type environments:

$$[\![\varnothing]\!] \stackrel{\text{def}}{=} \varnothing$$

$$[\![\Gamma, x : T]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!], x : [\![T]\!]$$

Translation of higher-order values:

$$[\![(x).\, P]\!]^{\Gamma;T} \stackrel{\text{def}}{=} (x).\, [\![P]\!]^{\Gamma,x:S} \quad \text{if } T = S \to \diamond$$

$$[\![y]\!]^{\Gamma;T} \stackrel{\text{def}}{=} (x).\, \bar{y}x$$

Translation of processes:
(we say that a value v is π-transmittable if v is a name or the unit value; and we assume that y is a fresh name)

$$[\![v\langle w : T\rangle]\!]^{\Gamma} \stackrel{\text{def}}{=} \begin{cases} [\![v]\!]^{\Gamma;T\to\diamond} w & \text{if } w \text{ is } \pi\text{—transmittable} \\ (\nu y : [\![T]\!]^{\leftarrow\sharp})([\![v]\!]^{\Gamma;T\to\diamond} y \mid !y[\![w]\!]^{\Gamma;T}) & \\ \qquad\qquad \text{if } w \text{ is not } \pi\text{—transmittable} & \end{cases}$$

$$[\![\bar{v}w]\!]^{\Gamma} \stackrel{\text{def}}{=} \begin{cases} \bar{v}w & \text{if } w \text{ is } \pi\text{—transmittable} \\ (\nu y : [\![T]\!]^{\leftarrow\sharp})(\bar{v}y \mid !y[\![w]\!]^{\Gamma;T}) & \\ \qquad \text{if } w \text{ is not } \pi\text{—transmittable and } \Gamma \vdash v : \sharp T & \end{cases}$$

$$[\![z(x).\, P]\!]^{\Gamma} \stackrel{\text{def}}{=} z[\![(x).\, P]\!]^{\Gamma;T} \quad \text{if } \Gamma \vdash z : \sharp T$$

$$[\![P \mid Q]\!]^{\Gamma} \stackrel{\text{def}}{=} [\![P]\!]^{\Gamma} \mid [\![Q]\!]^{\Gamma} \qquad [\![0]\!]^{\Gamma} \stackrel{\text{def}}{=} 0$$

$$[\![(\nu x : T)P]\!]^{\Gamma} \stackrel{\text{def}}{=} (\nu x : [\![T]\!])[\![P]\!]^{\Gamma,s:T} \qquad [\![!P]\!]^{\Gamma} \stackrel{\text{def}}{=} ![\![P]\!]^{\Gamma}$$

fv(M). The subset of Λ containing only the closed terms is Λ^0. To avoid too many brackets, we assume that application associates to the left, so that MNL should be read $(MN)L$, and that the scope of a λ extends as far as possible to the right, so that $\lambda x.\, MN$ should be 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 Barendregt (1984) and Hindley and Seldin (1986) in notation and terminology for the λ-calculus. The basic computational step of the λ-calculus is β-reduction:

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

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

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

Table 3. *The rules for a λ-calculus congruence*

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

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

The rules β, μ, ν, ξ define the *full β-reduction*, written $\longrightarrow_\beta$; its reflexive and transitive closure is $\Longrightarrow_\beta$. The $\lambda\beta$ *theory*, also called the *formal theory of β equality*, is defined by the axiom

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

plus the axiom and inference rules for congruence in Table 3. We write $\lambda\beta \vdash M = N$ if $M = N$ can be proved in the $\lambda\beta$ theory. We give names to some special λ-terms:

$$I \overset{\text{def}}{=} \lambda x.\,x \qquad \Omega \overset{\text{def}}{=} (\lambda x.\,xx)(\lambda x.\,xx).$$

Reduction strategies for the λ-calculus. A *reduction strategy* specifies which β-redexes in a term may be contracted. 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. A notion of reduction gives rise to a λ-theory when one adds the rules that turn the reduction relation into a congruence relation. The full β reduction is itself a strategy. Important strategies in programming languages are *call-by-name* and *call-by-value.*

Call-by-value. In the *call-by-value* (or *eager*) strategy, the argument N of a function is reduced to a *value* before the redex is contracted. The values of the untyped call-by-value λ-calculus are the functions (that is, the λ-abstractions) and, on open terms, 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 obtained by substitution from a variable are closed values):

$$\text{Values} \quad V := \lambda x.\,M \mid x \qquad M \in \Lambda.$$

The *one-step call-by-value reduction relation* $\longrightarrow_V \subseteq \Lambda \times \Lambda$ is defined by the rule μ plus the two rules:

$$\beta_V\,\frac{}{(\lambda x.\,M)V \longrightarrow_V M\{V/x\}} \qquad \nu_V\,\frac{N \longrightarrow_V N'}{(\lambda x.\,M)N \longrightarrow_V (\lambda x.\,M)N'}.$$

Examples of call-by-value reductions are

$$(\lambda x.\,I)\Omega \longrightarrow_V (\lambda x.\,I)\Omega \longrightarrow_V \ldots$$
$$(\lambda x.\,xx)(II) \longrightarrow_V (\lambda x.\,xx)I \longrightarrow_V II \longrightarrow_V I$$
$$(\lambda xy.\,x)z(II) \longrightarrow_V (\lambda y.\,z)(II) \longrightarrow_V (\lambda y.\,z)I \longrightarrow_V z.$$

Call-by-value is based on the β_V rule. The λ-theory induced by this rule is the *$\lambda\beta_V$ theory*, also called the *formal theory of β_V equality*, defined by the axiom

$$\beta_V \frac{}{(\lambda x.\, M)V = M\{V/x\}}$$

plus the inference rules in Table 3. A correct semantics of call-by-value should (at least) validate the equalities of $\lambda\beta_V$.

Call-by-name. The *call-by-name* strategy always choses the leftmost redex, but reductions stop when a constructor is at the top. On open terms, the call-by-name strategy also stops on terms with a variable in head position, that is, terms of the form $xM_1 \dots M_n$. This is 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* $\longrightarrow_N \subseteq \Lambda \times \Lambda$ is defined by the rules β and μ. Reduction is deterministic: the redex is always at the extreme left of a term. Examples of call-by-name reductions are

$$\begin{aligned}
&(\lambda x.\, I)\Omega \longrightarrow_N I\\
&(\lambda x.\, xx)(II) \longrightarrow_N (II)(II) \longrightarrow_N I(II) \longrightarrow_N II \longrightarrow_N I\\
&(\lambda xy.\, x)z(II) \longrightarrow_N (\lambda y.\, z)(II) \longrightarrow_N z.
\end{aligned}$$

Since call-by-name is based on the β rule, the λ-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 $\lambda\beta$.

Interpreting λ-calculi into π-calculus. All of the translations of λ-calculus strategies into π-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 β-reduction is modelled as an interaction between them.

— The encoding of a λ-term is parametric over a name. This name is used by (the translation of) the λ-term to interact with the environment, and corresponds to the continuation argument of a CPS function.

For simplicity, we adopt the convention that λ-calculus variables are also π-calculus names.

6. The interpretation of call-by-value

In this section we develop the left-hand part of the diagram of Figure 1. The π-calculus encoding of call-by-value λ-calculus (λV) is obtained in three steps, the first of which is the call-by-value CPS transform of Plotkin (1975).

Step 1: The call-by-value CPS transform. The *call-by-value CPS transform*, $\mathscr{C}_V$, transforms functions of λV into CPS functions. In its definition, the translation of values uses the auxiliary translation function $\mathscr{C}^*_V$ (which will be particularly useful when considering types). We call a term $\mathscr{C}^*_V[V]$ a *CPS-value*. The transform is presented in Table 4. 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 π-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 $\mathscr{C}_{\mathrm{V}}$ 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.)

First we explain informally how the CPS transform works. The CPS image of a λ-term L immediately needs a continuation. When a continuation is provided, L is reduced to a value, and this value (more precisely, its CPS-value) is passed to the continuation. Therefore if L is itself a value, it can be passed directly to the continuation. If, however, L is an application MN, the following happens. First M is evaluated with continuation $\lambda v. \mathscr{C}_{\mathrm{V}}[\![N]\!](\lambda w. vwk)$. When M becomes a function, say $\lambda x. M_1$, this function is passed to the continuation, and the body of the continuation is evaluated. This means evaluating N with continuation $\lambda w. vwk\{\mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]/v\}$. 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 $(vwk)\{\mathscr{C}^*_{\mathrm{V}}[V]/w\}\{\mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]/v\}$, that is $(\lambda x. \mathscr{C}_{\mathrm{V}}[\![M_1]\!])\mathscr{C}^*_{\mathrm{V}}[V]k$. This term reduces to $\mathscr{C}_{\mathrm{V}}[\![M_1]\!]\{\mathscr{C}^*_{\mathrm{V}}[V]/x\}k$, which is the same as $\mathscr{C}_{\mathrm{V}}[\![M_1\{V/x\}]\!]k$. The reduction of $\mathscr{C}_{\mathrm{V}}[\![M_1\{V/x\}]\!]k$ 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 λV 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 of the behaviour of application, we report below the details of how a β_{v} reduction

$$(\lambda x. M_1)V \longrightarrow_{\mathrm{V}} M_1\{V/x\}$$

is simulated. We use the call-by-name reduction $\longrightarrow_{\mathrm{N}}$ 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 6.3).

$$\begin{aligned}
& \mathscr{C}_{\mathrm{V}}[\![(\lambda x. M_1)V]\!]k \qquad\qquad (1)\\
&= \big(\lambda k. \mathscr{C}_{\mathrm{V}}[\![\lambda x. M_1]\!](\lambda v. \mathscr{C}_{\mathrm{V}}[\![V]\!](\lambda w. vwk))\big)k\\
&\longrightarrow_{\mathrm{N}} \mathscr{C}_{\mathrm{V}}[\![\lambda x. M_1]\!](\lambda v. \mathscr{C}_{\mathrm{V}}[\![V]\!](\lambda w. vwk))\\
&= \lambda h. h\mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1](\lambda v. \mathscr{C}_{\mathrm{V}}[\![V]\!](\lambda w. vwk))\\
&\longrightarrow_{\mathrm{N}} (\lambda v. \mathscr{C}_{\mathrm{V}}[\![V]\!](\lambda w. vwk))\mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]\\
&\longrightarrow_{\mathrm{N}} \mathscr{C}_{\mathrm{V}}[\![V]\!](\lambda w. \mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]wk)\\
&= \lambda h. h\mathscr{C}^*_{\mathrm{V}}[V](\lambda w. \mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]wk)\\
&\longrightarrow_{\mathrm{N}} (\lambda w. \mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]wk)\mathscr{C}^*_{\mathrm{V}}[V]\\
&\longrightarrow_{\mathrm{N}} \mathscr{C}^*_{\mathrm{V}}[\lambda x. M_1]\mathscr{C}^*_{\mathrm{V}}[V]k\\
&= (\lambda x. \mathscr{C}_{\mathrm{V}}[\![M_1]\!])\mathscr{C}^*_{\mathrm{V}}[V]k\\
&\longrightarrow_{\mathrm{N}} \mathscr{C}_{\mathrm{V}}[\![M_1]\!]\{\mathscr{C}^*_{\mathrm{V}}[V]/x\}k\\
&= \mathscr{C}_{\mathrm{V}}[\![M_1\{V/x\}]\!]k.
\end{aligned}$$

Table 4. *The call-by-value CPS transform* (*In this table we abbreviate* $\mathscr{C}_{\mathrm{V}}[\![M]\!]$ *as* $[\![M]\!]$ *and* $\mathscr{C}^*_{\mathrm{V}}[V]$ *as* $[V]$.)

Call-by-value values $V := \lambda x.\,M \mid x$

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

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 corresponds directly to the reduction on the source terms.

The call-by-value CPS transform maps λ-terms onto a subset of the λ-terms. The closure of that subset under β-conversion gives the language $\mathrm{CPS_V}$ of the call-by-value CPS:

$$\mathrm{CPS_V} \stackrel{\mathrm{def}}{=} \{A : \exists M \in \Lambda \text{ with } \mathscr{C}_{\mathrm{V}}[\![M]\!] \Longrightarrow_\beta A\}.$$

We call the terms of $\mathrm{CPS_V}$ the *CPS terms.*

The first theorem shows that on CPS terms, β- and β_{v}-redexes coincide.

Theorem 6.1. (Indifference of $\mathrm{CPS_V}$ on reductions) Let $M \in \mathrm{CPS_V}$ and let N be any subterm of M. For all N', we have $N \longrightarrow_{\mathrm{N}} N'$ iff $N \longrightarrow_{\mathrm{V}} N'$.

Proof. Below we shall give a grammar, Grammar (3), that generates all CPS terms. It is immediate to check that on terms generated by that grammar, β-redexes and β_{v}-redexes coincide, because all arguments of functions are values of λV (abstractions and variables). $\square$

Essentially as a consequence of Theorem 6.1, we obtain the following Theorem.

Theorem 6.2. (Indifference of $\mathrm{CPS_V}$ on λ-theories) For all $M, N \in \mathrm{CPS_V}$, we have $\lambda\beta \vdash M = N$ iff $\lambda\beta_{\mathrm{v}} \vdash M = N$.

Proof. The implication from right to left holds because a β_{v}-redex is also a β-redex. We consider the implication from left to right. Since full β-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 6.1, all β-redexes contracted in the reductions $L_1 \Longrightarrow_\beta L_3$ and $L_2 \Longrightarrow_\beta L_3$ are also β_{v}-redexes. Hence $\lambda\beta_{\mathrm{v}} \vdash L_1 = L_2$. $\square$

Remark 6.3. These indifference properties allow us to take either call-by-name or call-by-value as the reduction strategy and the λ-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 λ-term is correctly mimicked by its CPS image. The second shows that the CPS transform preserves β_{v}-conversion.

Theorem 6.4. (adequacy for $\mathscr{C}_V$) Let $M \in \Lambda^0$.

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

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 made clear by considering the open term xx: this term is not reducible (it is a V-nf), but its image $\mathscr{C}_V[\![xx]\!]k$ has 5 (administrative) reductions.

Theorem 6.5. (Validity of the β-theory for $\mathscr{C}_v$) Let $M, N \in \Lambda$. If $\lambda\beta_v \vdash M = N$, then $\lambda\beta \vdash \mathscr{C}_V[\![M]\!] = \mathscr{C}_V[\![N]\!]$.

Proof. [sketch] First one shows that if $M \Longrightarrow_V N$, then $\lambda\beta \vdash \mathscr{C}_V[\![M]\!] = C_V[\![N]\!]$. Then one concludes the proof using the fact that β_v is confluent. □

The converse of Theorem 6.5 fails. For instance, if

$$\begin{aligned} M &\stackrel{\text{def}}{=} \Omega y = (\lambda x.\, xx)(\lambda x.\, xx)y \\ N &\stackrel{\text{def}}{=} (\lambda x.\, xy)\Omega = (\lambda x.\, xy)((\lambda x.\, xx)(\lambda x.\, xx)), \end{aligned} \tag{2}$$

then $\lambda\beta \vdash \mathscr{C}_V[\![M]\!] = C_V[\![N]\!]$ holds, but $\lambda\beta_v \vdash M = N$ does not.

The statements on the correctness of the CPS transform complete Step 1 of the left-hand part of Figure 1.

Step 2: From CPS$_V$ to HOπ. The next step is to show that, modulo the different syntax for abstraction and application, the terms of CPS$_V$ are also terms of HOπ. To do this, we present a grammar that generates all CPS$_V$ terms, and show that the terms generated by this grammar are also terms of HOπ.

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 λ-terms under the CPS transform. CPS-values correspond, intuitively, to the values of λV; 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 (β-reductions) takes place.

$$\begin{array}{rl} \text{continuation variable} & k \\ \text{ordinary variables} & x, \ldots \\ \text{answers} & P := KV \mid VVK \mid AK \\ \text{CPS-values} & V := \lambda x.\, \lambda k.\, P \mid x \\ \text{continuations} & K := k \mid \lambda x.\, P \\ \text{principal terms} & A := \lambda k.\, P \end{array} \tag{3}$$

Remark 6.6. 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 HOπ below.

Remark 6.7. 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 α-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 (3) 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

$$\begin{aligned} T_V &\stackrel{\text{def}}{=} \mu X.\big((X \to (X \to \diamond) \to \diamond)\big) \\ T_K &\stackrel{\text{def}}{=} T_V \to \diamond \\ T_A &\stackrel{\text{def}}{=} T_K \to \diamond. \end{aligned} \tag{4}$$

The type judgements for the terms M generated by Grammar (3) are of the form

$$\Gamma \vdash M : T \tag{5}$$

where $T \in \{T_V, T_K, T_A, \diamond\}$ and Γ 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 9 that there is a general schema for translating type judgements on λ-terms to type judgements on the CPS-images of the λ-terms, and that the schema applies also to untyped λ-calculus.

Remark 6.8. To be precise, we should include some linear information in the types (4) in order to show the linear usage of continuations. In the step of encoding into the π-calculus, these linear information allow us to avoid some replication operators. We do not show the linear types for the following reasons: the linear information is already highlighted in (3) by the syntactic separation between continuation and ordinary variables; linearity is not fundamental (the simplification on replications in the π-calculus is pretty straightforward and could anyway be omitted), linear types are not needed in the final π-calculus encoding (linearity can be 'forgotten' when translating types, see the discussion in Section 4.1).

All CPS terms are indeed generated by Grammar (3):

Proposition 6.9. If $M \in \mathrm{CPS_V}$, then M is a principal term of Grammar (3).

Proof. The set of principal terms includes the set $\{\mathscr{C}_V[\![M]\!] : M \in \Lambda\}$ and is closed under β-conversion. $\square$

It is easy to see that the set of terms generated by Grammar (3) is, essentially, a subset of HOπ terms. To be precise, answers may be regarded as HOπ processes, and CPS-values, continuations, and principal terms as HOπ abstractions. Recall from Section 3 that the grammar of (polyadic) HOπ requires that abstractions be either variables or parametrised processes. CPS-values, continuations, and principal terms of Grammar (3) 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 4,

Table 5. *The encoding of λV into the π-calculus*

$$\begin{aligned}
\mathscr{V}[\![\lambda x.\, M]\!] &\stackrel{\text{def}}{=} (p).\, \overline{p}(y).\, !y(x)\mathscr{V}[\![M]\!] \\
\mathscr{V}[\![x]\!] &\stackrel{\text{def}}{=} (p).\, \overline{p}x \\
\mathscr{V}[\![MN]\!] &\stackrel{\text{def}}{=} (p).\, \nu q\Big(\mathscr{V}[\![M]\!]q \mid q(v).\, \nu r(\mathscr{V}[\![N]\!]r \mid r(w).\, \overline{v}\langle w, p\rangle)\Big)
\end{aligned}$$

$\mathscr{C}^*_{\mathrm{V}}[\lambda x.\, M]$ becomes $\lambda(x,k).\mathscr{C}_{\mathrm{V}}[\![M]\!]k$, 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 $\mathscr{H}$ from the terms of Grammar (3) to HOπ. On terms, modulo this uncurrying, the injection rewrites λ-abstractions into HOπ abstractions, and λ-applications into HOπ applications (with the obvious type annotations in applications, given by (4) and the translation of types). The injection is the identity on (uncurried) types; thus

$$\mathscr{H}[\![T_V]\!] \stackrel{\text{def}}{=} \mu X.\big((X \times (X \to \diamond)) \to \diamond\big). \tag{6}$$

We have $\mathscr{H}[\![\diamond]\!] \stackrel{\text{def}}{=} \diamond$, which explains the abuse of notation whereby the same symbol $\diamond$ is used for the type of answers of Grammar (3) and for the type of processes of HOπ. The image of $\mathscr{H}$ is a HOπ language that has recursive and product types.

The proofs of the following lemmas are straightforward.

Lemma 6.10. Suppose that M is a term generated by Grammar (3), and that $\Gamma \vdash M : T$ for Γ and T as in (5). Then $\mathscr{H}[\![\Gamma]\!] \vdash \mathscr{H}[\![M]\!] : \mathscr{H}[\![T]\!]$.

As a corollary of Lemma 6.10, if $M \in \Lambda$ with $\mathrm{fv}(M) \subseteq \widetilde{x}$, then

$$\widetilde{x} : \mathscr{H}[\![T_V]\!] \vdash \mathscr{H}[\![\mathscr{C}_{\mathrm{V}}[\![M]\!]]\!] : \mathscr{H}[\![T_A]\!] = (\mathscr{H}[\![T_V]\!] \to \diamond) \to \diamond. \tag{7}$$

We recall that if P, Q are HOπ processes, then $P \longrightarrow_\beta Q$ and $P =_\beta Q$ denote β-reduction and β-convertibility in HOπ, respectively (Section 3).

Lemma 6.11. For all terms M of Grammar (3), if $M \longrightarrow_{\mathrm{N}} M'$, then $\mathscr{H}[\![M]\!] \longrightarrow_\beta\ =_\beta \mathscr{H}[\![M']\!]$. Conversely, if $\mathscr{H}[\![M]\!] \longrightarrow_\beta P$, then there is M' such that $M \longrightarrow_{\mathrm{N}} M'$ and $P =_\beta \mathscr{H}[\![M]\!]$.

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

Corollary 6.12. For all terms M, N generated by Grammar (3), $\lambda\beta \vdash M = N$ implies $\mathscr{H}[\![M]\!] =_\beta \mathscr{H}[\![N]\!]$.

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 6.11, $\mathscr{H}[\![M]\!] =_\beta \mathscr{H}[\![L]\!] =_\beta \mathscr{H}[\![N]\!]$. □

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

Step 3: Compilation $\mathscr{C}$. The third and final step for the left-hand part of Figure 1 is from HOπ to π-calculus. This step is given by the compilation $\mathscr{C}$ from HOπ to π-calculus of Section 4.

Composing the steps. Composing the three steps, from λV to $\mathrm{CPS}_{\mathrm{V}}$, from $\mathrm{CPS}_{\mathrm{V}}$ to HOπ, and from HOπ to π-calculus, we obtain the encoding $\mathscr{V}$ of λV into π-calculus in Table 5. To be precise, the encoding $\mathscr{V}[\![M]\!]$ of a λ-term M is obtained as follows (we can omit the type environment index in $\mathscr{C}$, for M is untyped and therefore all free variables are translated in the same way):

$$\mathscr{V}[\![M]\!] \stackrel{\text{def}}{=} \mathscr{C}[\![\mathscr{H}[\![\mathscr{C}_{\mathrm{V}}[\![M]\!]]\!]]\!].$$

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

— *Location names* (p, q, r) that are used as arguments of the encodings of λ-terms. These names correspond to the continuation variable of the CPS Grammar (3).

— *Value names* (x, y) that are used to access values. These names correspond to the ordinary variables of the CPS Grammar (3).

(As $\mathscr{V}[\![M]\!]$ is an abstraction, we recall these notations for abstractions from Section 2.1: if $\mathscr{V}[\![M]\!]$ is $(q).\,Q$, then $y(x)\mathscr{V}[\![M]\!]$ is $y(x, q).\,Q$, and $\mathscr{V}[\![M]\!]r$ is $Q\{r/q\}$.) 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 N also 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 β_{v}-reduction of λV; the previous communications were 'administrative' (exactly as happens in the CPS transform – see Section 6). The second argument of the pair $\langle w, p\rangle$ is the location where the final result of MN will be delivered. The first argument is a pointer to the value that N has reduced to, or the value itself if this value is a variable.

Remark 6.13. In Table 5, the inputs at location names (q and r) are not replicated because, in the step of translation from HOπ to π-calculus, we take into account the linearity constraint on continuations (see Remark 6.7) and therefore adopt the optimisation of Section 4.1. As discussed in Section 4.1, linear types are, however, not needed in the π-calculus.

For the sake of readability, the translation of Table 5 is not annotated with types. Types, in particular i/o types, are, however, introduced in the compilation from HOπ to π-calculus, and are needed for the correctness of the compilation (Remark 6.17). The π-calculus translation of type $\mathscr{H}[\![T_V]\!]$ in (6) is the recursive type

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

We do not need the i tag on types, because both location and trigger names that are communicated may only be used by a recipient for sending. Below is the π-calculus translation of λV, including the type annotations for restricted names. We recall that if T is a π-calculus type, then $(T)^{\leftarrow\sharp}$ is the type obtained by replacing the outermost i/o tag in T with $\sharp$, possibly after unfolding T if its outermost construct is recursion.

$$\begin{aligned}
\mathscr{V}[\![\lambda x.\, M]\!] &\overset{\text{def}}{=} (p).\,\overline{p}(y : (\mathrm{Val})^{\rightarrow\sharp}).\, !y(x)\mathscr{V}[\![M]\!] \\
\mathscr{V}[\![x]\!] &\overset{\text{def}}{=} (p).\,\overline{p}x \\
\mathscr{V}[\![MN]\!] &\overset{\text{def}}{=} (p).\,(\nu q : \sharp(\mathrm{Val})) \\
&\quad \big(\mathscr{V}[\![M]\!]q \mid q(x).\,(\nu r : \sharp(\mathrm{Val}))\; (\mathscr{V}[\![N]\!]r \mid r(y).\,\overline{x}\langle y, p\rangle)\big)\,.
\end{aligned}$$

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

The translation of (7) into π-calculus gives the following lemma.

Lemma 6.14. Suppose $\mathrm{fv}(M) \subseteq \tilde{x}$. Then $\tilde{x} : \mathrm{Val},\, p : \mathsf{o}\,(\mathrm{Val}) \vdash \mathscr{V}[\![M]\!]p$.

From the correctness of the 3 steps from which encoding $\mathscr{V}$ has been derived, we get the two corollaries below.

Corollary 6.15. (Adequacy of $\mathscr{V}$) Let $M \in \Lambda^0$. Then $M \Downarrow_{\mathrm{V}}$ iff $\mathscr{V}[\![M]\!]p \Downarrow$, for any p.

Proof. The result follows from Theorem 6.4, Lemma 6.11 and the operational correctness of the encoding of HOπ into the π-calculus (Lemma 4.2 and its extensions, Section 4.1). □

Corollary 6.16. (Validity of $\lambda\beta_{\mathrm{v}}$ theory for $\mathscr{V}$) Suppose $\mathrm{fv}(M, N) \subseteq \tilde{x}$, and let $H \overset{\text{def}}{=} \tilde{x} : \mathrm{Val}; \mathsf{o}(\mathrm{Val})$. If $\lambda\beta_{\mathrm{v}} \vdash M = N$, then $\mathscr{V}[\![M]\!] \approx_H \mathscr{V}[\![N]\!]$.

Proof. The result follows from Theorem 6.5, Corollary 6.12 and (the appropriate extension of) Lemma 4.3. □

The counterexample to the converse of Theorem 6.5 implies also that the converse of Corollary 6.16 is false.

Remark 6.17. (Effect of i/o types on behavioural equivalences) In the relation of barbed congruence of Corollary 6.16, the presence of i/o types is important. The result would not hold if each i/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 $M \overset{\text{def}}{=} (\lambda x.\,(\lambda y.\,x))\lambda z.\,z$ and $N \overset{\text{def}}{=} \lambda y.\,(\lambda z.\,z)$. With a β_{v}-conversion, M reduces to N. However, without i/o types, $\mathscr{V}[\![M]\!]p$ and $\mathscr{V}[\![N]\!]p$ could be distinguished – (see Sangiorgi 1992a page 128).

By Corollary 6.16, we know that if $M \longrightarrow_{\mathrm{V}} M'$, then $\mathscr{V}[\![M]\!]$ and $\mathscr{V}[\![M']\!]$, are behaviourally indistinguishable. One might like, however, to see how the reduction of λV is simulated in the π-calculus:

$$\begin{aligned}
& \mathscr{V}[\![(\lambda x.\, M)\lambda y.\, N]\!]p && (8) \\
= \;& \nu q(\overline{q}(z).\, !z(x)\mathscr{V}[\![M]\!] \mid q(z).\, \nu r(\overline{r}(u).\, !u(y)\mathscr{V}[\![N]\!] \mid r(u).\, \overline{z}\langle u, p\rangle)) \\
\longrightarrow\sim\; & (\nu r, z)(!z(x)\mathscr{V}[\![M]\!] \mid \overline{r}(u).\, !u(y)\mathscr{V}[\![N]\!] \mid r(u).\, \overline{z}\langle u, p\rangle) \\
\longrightarrow\sim\; & (\nu u, z)(!z(x)\mathscr{V}[\![M]\!] \mid !u(y)\mathscr{V}[\![N]\!] \mid \overline{z}\langle u, p\rangle) \\
\longrightarrow\sim\; & (\nu u, z)(\mathscr{V}[\![M]\!]p\{u/x\} \mid !z(x)v[\![M]\!] \mid !u(y)\mathscr{V}[\![N]\!]) \\
\sim\; & (\nu x)\; (\mathscr{V}[\![M]\!]p) \mid !x(y)\, \mathscr{V}[\![N]\!])
\end{aligned}$$

Table 6. *The call-by-name CPS transform* (*In this table we abbreviate* $\mathscr{C}_{\mathrm{N}}[\![M]\!]$ *as* $[\![M]\!]$ *and* $\mathscr{C}^{*}_{\mathrm{N}}[V]$ *as* $[V]$.)

call-by-name values $V := \lambda x.\, M$

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

where the occurrences of $\sim$ indicate the application of the laws of structural equivalence and the laws of Lemma 2.3. We have that $(\nu x)(\mathscr{V}[\![M]\!]p \mid !x(y)\mathscr{V}[\![N]\!]) \approx \mathscr{V}[\![M\{N/x\}]\!]p$. The proof of this is simpler than, but similar to, the proof of Lemma 4.3.

Remark 6.18. The results on the CPS transform, notably Theorems 6.4 and 6.5, have been used to derive results about the correctness of the π-calculus encoding. But we can also use the factorisation of Figure 1 in the opposite direction. For instance, developing (8) above, we can give a direct proof of Corollary 6.15, and from this, Lemma 6.11, and Lemma 4.2, we can derive the adequacy of the CPS transform (Theorem 6.4). Similarly, from Corollary 6.16 and Lemma 4.4 we can prove a weaker version of Theorem 6.5, saying that if $\lambda\beta_{\mathrm{v}} \vdash M = N$, then $\mathscr{C}_{\mathrm{V}}[\![M]\!]$ and $\mathscr{C}_{\mathrm{V}}[\![N]\!]$ are behaviourally equivalent as terms of λN. (As behavioural equivalence on λN, we can take barbed congruence, using $\Downarrow_{\mathrm{N}}$ as the only observability predicate.)

7. The interpretation of call-by-name

In this section we develop a π-calculus encoding of call-by-name λ-calculus. The approach is similar to that for call-by-value in Section 6.

Step 1: The call-by-name CPS. Table 6 shows the call-by-name CPS transform of Plotkin (1975) – in fact, a rectified variant of it (see the notes in Section 10). How a β reduction

$$(\lambda x.\, M)N \longrightarrow_{\mathrm{N}} M\{N/x\}$$

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 7.1):

$$
\begin{aligned}
& \mathscr{C}_{\mathrm{N}}[\![(\lambda x.\, M)N]\!]k \\
\longrightarrow_{\mathrm{N}}\ & \mathscr{C}_{\mathrm{N}}[\![\lambda x.\, M]\!](\lambda v.\, v\mathscr{C}_{\mathrm{N}}[\![N]\!]k) \\
=\ & (\lambda k.\, (k\mathscr{C}^{*}_{\mathrm{N}}[\lambda x.\, M]))(\lambda v.\, v\mathscr{C}_{\mathrm{N}}[\![N]\!]k) \\
\longrightarrow_{\mathrm{N}}\ & (\lambda v.\, v\mathscr{C}_{\mathrm{N}}[\![N]\!]k)\mathscr{C}^{*}_{\mathrm{N}}[\lambda x.\, M]
\end{aligned}
$$

$$\begin{aligned} &\longrightarrow_{\mathrm{N}} \mathscr{C}^*_{\mathrm{N}}[\lambda x.\, M]\mathscr{C}_{\mathrm{N}}[\![N]\!]k \\ &= \quad (\lambda x. \mathscr{C}_{\mathrm{N}}[\![M]\!])\mathscr{C}_{\mathrm{N}}[\![N]\!]k \\ &\longrightarrow_{\mathrm{N}} \mathscr{C}_{\mathrm{N}}[\![M]\!]\{\mathscr{C}_{\mathrm{N}}[\![N]\!]/x\}k. \end{aligned}$$

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

Closing the images of the transform under β-reduction gives the language $\mathrm{CPS_N}$ of the call-by-name CPS:

$$\mathrm{CPS_N} \overset{\mathrm{def}}{=} \{A : \exists M \in \Lambda \text{ with } \mathscr{C}_{\mathrm{N}}[\![M]\!] \Longrightarrow_\beta A\}.$$

When there is no ambiguity, we call the terms of $\mathrm{CPS_N}$ the *CPS terms*.

The theorems below are the call-by-name versions of Theorems 6.1–6.5. 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 7.3 is slightly weaker than the corresponding result for call-by-value. The reason is that, as illustrated above, $\mathscr{C}_{\mathrm{N}}$ commutes with substitution only up to β-conversion. In contrast, Theorem 7.4 is stronger than the corresponding result for call-by-value: it asserts a logical equivalence rather than an implication.

Theorem 7.1. (Indifference of $\mathbf{CPS_N}$ on reductions) Let $M \in \mathrm{CPS_N}$ and let N be any subterm of M. For all N', we have $N \longrightarrow_{\mathrm{N}} N'$ iff $N \longrightarrow_{\mathrm{V}} N'$.

Theorem 7.2. (Indifference of $\mathbf{CPS_N}$ on $\boldsymbol{\lambda}$-theories) For all $M, N \in \mathrm{CPS_N}$, we have $\lambda\beta \vdash M = N$ iff $\lambda\beta_{\mathrm{V}} \vdash M = N$.

Theorem 7.3. (Adequacy of $\mathscr{C}_{\mathrm{N}}$) Let $M \in \Lambda^0$.

1 If $M \Longrightarrow_{\mathrm{N}} V$ where V is a call-by-name value, then there is an N-nf N such that $\mathscr{C}_{\mathrm{N}}[\![M]\!]k \Longrightarrow_{\mathrm{N}} N$ and $\lambda\beta \vdash N = k\mathscr{C}^*_{\mathrm{N}}[V]$.

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

Theorem 7.4. (Validity of the $\boldsymbol{\beta}$-theory for $\mathscr{C}_{\mathrm{N}}$) Let $M, N \in \Lambda$. Then $\lambda\beta \vdash M = N$ iff $\lambda\beta \vdash \mathscr{C}_{\mathrm{N}}[\![M]\!] = \mathscr{C}_{\mathrm{N}}[\![N]\!]$.

Step 2: From $\mathbf{CPS_N}$ to HO$\boldsymbol{\pi}$. The grammar below generates all terms of $\mathrm{CPS_N}$. The intuitive meaning of the various syntactic categories in the grammar is the same as for the call-by-value Grammar (3). 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 8). As in the call-by-value grammar, there is only one continuation variable k because continuations are used linearly.

Table 7. *The encoding of λN into the π-calculus*

$$
\begin{aligned}
\mathscr{N}[\![\lambda x.\,M]\!] &\overset{\text{def}}{=} (p).\,\overline{p}(v).\,v(x)\mathscr{N}[\![M]\!]\\
\mathscr{N}[\![x]\!] &\overset{\text{def}}{=} (p).\,\overline{x}p\\
\mathscr{N}[\![MN]\!] &\overset{\text{def}}{=} (p).\,\nu q\,\big(\mathscr{N}[\![M]\!]q \mid q(v).\,\nu x(\overline{v}\langle x,p\rangle.\,!x\,\mathscr{N}[\![N]\!])\big)
\end{aligned}
$$

$$
\begin{array}{rl}
\text{continuation variable} & k\\
\text{ordinary variables} & x,\ldots\\
\text{value variable} & v\\
\text{answers} & P \;:= P_1 \mid P_2\\
 & P_1 \;:= K\,V \mid VAK \mid AK\\
 & P_2 \;:= vAK\\
\text{CPS-values} & V \;:= \lambda x.\,\lambda k.\,P_1\\
\text{continuations} & K \;:= k \mid \lambda v.\,P_2\\
\text{principal terms} & A \;:= \lambda k.\,P_1 \mid x.
\end{array}
\tag{9}
$$

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

$$
\begin{aligned}
T_V &\overset{\text{def}}{=} \mu X.\,(((X \to \diamond) \to \diamond) \to ((X \to \diamond) \to \diamond))\\
T_K &\overset{\text{def}}{=} T_V \to \diamond\\
T_A &\overset{\text{def}}{=} T_K \to \diamond.
\end{aligned}
\tag{10}
$$

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

Proposition 7.5. If $M \in \text{CPS}_\text{N}$, then M is also a principal term of Grammar (9).

Proof. One can show that the set of principal terms includes the set $\{\mathscr{C}_\text{N}[\![M]\!] : M \in \Lambda\}$ and is closed under β-conversion. □

The injection $\mathscr{H}$, from the terms generated by Grammar (9) to HOπ, is defined as for the call-by-value Grammar (3), and similar results hold:

Lemma 7.6. Suppose M is a term generated by Grammar (9), and $\Gamma \vdash M : T$. Then also $\mathscr{H}[\![\Gamma]\!] \vdash \mathscr{H}[\![M]\!] : \mathscr{H}[\![T]\!]$.

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

$$
\widetilde{x} : \mathscr{H}[\![T_A]\!] \vdash \mathscr{H}[\![C_\text{N}[\![M]\!]]\!] : \mathscr{H}[\![T_A]\!] = (\mathscr{H}[\![T_V]\!] \to \diamond) \to \diamond.
\tag{11}
$$

Lemma 7.7. For all terms M of Grammar (9), we have if $M \longrightarrow_\text{N} M'$, then $\mathscr{H}[\![M]\!] \longrightarrow_\beta =_\beta \mathscr{H}[\![M]\!]'$. Conversely, if $\mathscr{H}[\![M]\!] \longrightarrow_\beta P$, then there is M' such that $M \longrightarrow_\text{N} M'$ and $P =_\beta \mathscr{H}[\![M']\!]$.

Corollary 7.8. For all terms M, N generated by Grammar (3), $\lambda\beta \vdash M = N$ implies $\mathscr{H}[\![M]\!] =_\beta \mathscr{H}[\![N]\!]$.

Step 3: Compilation $\mathscr{C}$. This step is given by compilation $\mathscr{C}$ of HOπ into π-calculus.

Composing the steps. Composing the three steps, from λN to CPS$_\mathrm{N}$, from CPS$_\mathrm{N}$ to HOπ, and from HOπ to π-calculus, we obtain the encoding of λN into π-calculus in Table 7.

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 (9). Trigger names are pointers to λ-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 (9)), and are the counterpart of the value variable of the CPS grammar.

The difference between call-by-value and call-by-name is evident in the encoding of a variable x: the corresponding π-calculus name x is used not as a value, as it was in the translation of call-by-value, but as a trigger for activating a term and providing it with a location. As a consequence, in the encoding of an application MN at location p, when M signals that it has become function, it receives a trigger for the argument N, together with the location p for interacting with the environment.

Remark 7.9. 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 the continuation variable of Grammar (9) that, in the compilation of HOπ to the π-calculus, enables us to adopt the optimisation of Section 4.1; we need a similar approval in the call-by-value encoding, see Remark 6.13.

In the encoding of Table 7, we have omitted type annotations. Below is the complete encoding, including types. The type

$$\mathrm{Trig} \overset{\mathrm{def}}{=} \mu X.\,(\mathsf{o}(\mathsf{o}\,\mathsf{o}X \times \mathsf{o}X)) \tag{12}$$

is the translation into π-calculus of the type T_V in (10).

$$\begin{aligned}
\mathscr{N}[\![\lambda x.\,M]\!] &\overset{\mathrm{def}}{=} (p).\,\overline{p}(v : (\mathrm{Trig})^{\leftarrow\sharp}).\,v(x)\mathscr{N}[\![M]\!] \\
\mathscr{N}[\![x]\!] &\overset{\mathrm{def}}{=} (p).\,\overline{x}p \\
\mathscr{N}[\![MN]\!] &\overset{\mathrm{def}}{=} (p).\,(\nu q : \sharp(\mathrm{Trig})) \\
&\quad \big(\mathscr{N}[\![M]\!]q \mid q(v).\,(\nu x : \sharp\mathsf{o}(\mathrm{Trig}))\ (\overline{v}\langle x, p\rangle.\ !x\,\mathscr{N}[\![N]\!])\big)\,.
\end{aligned} \tag{13}$$

The translation of (11) into π-calculus gives

Lemma 7.10. Suppose $\mathrm{fv}(M) \subseteq \widetilde{x}$. Then $\widetilde{x} : \mathsf{o}\,\mathsf{o}(\mathrm{Trig}),\ p : \mathsf{o}(\mathrm{Trig}) \vdash \mathscr{N}[\![M]\!]p$.

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

Corollary 7.11. (Adequacy of $\mathscr{N}$) Let $M \in \Lambda^0$. Then $M \Downarrow_\mathrm{N}$ iff $\mathscr{N}[\![M]\!]p \Downarrow$, for any p.

Table 8. *A uniform CPS transform* (*In this table we abbreviate* $\mathscr{C}_{\mathrm{U}}[\![M]\!]$ *as* $[\![M]\!]$.)

$$[\![x]\!] \overset{\mathrm{def}}{=} \lambda k.\, xk$$

$$[\![\lambda x.\, M]\!] \overset{\mathrm{def}}{=} \lambda k.\, k(\lambda x.\, [\![M]\!])$$

call-by-name application:

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

call-by-value application:

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

Proof. From Theorem 7.3, Corollary 7.7, and (the appropriate extension of) Lemma 4.2. □

Corollary 7.12. (Validity of $\lambda\beta$ theory for $\mathscr{N}$) Suppose $\mathrm{fv}(M,N) \subseteq \widetilde{x}$ and let $H \overset{\mathrm{def}}{=} \widetilde{x} : \mathsf{oo}(\mathrm{Trig}); \mathsf{o}(\mathrm{Trig})$. If $\lambda\beta \vdash M = N$, then $\mathscr{N}[\![M]\!] \approx_H \mathscr{N}[\![N]\!]$.

Proof. The result follows from Theorem 7.4, Lemma 6.12 and (the appropriate extension of) Lemma 4.3. □

By contrast, $\mathscr{N}$ does not validate rule η, namely

$$\lambda x.\, (Mx) = M \qquad \text{if } x \notin \mathrm{fv}(M). \tag{14}$$

Neither does the call-by-value encoding $\mathscr{V}$ satisfy η. These failures make sense, as η is not operationally valid in either call-by-name or call-by-value λ-calculus.

8. A uniform encoding

The differences between the definitions of application in the π-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 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 obtain the new encodings again by going through a CPS transform, an injection into HOπ and the compilation of HOπ into π-calculus. We begin with the CPS transform. Starting from the call-by-value and call-by-name transforms examined in the previous sections (Tables 4 and 6), 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

Table 9. *The uniform encoding of call-by-name, and call-by-value*

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

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

call-by-value application:

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

call-by-name application:

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

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 8. 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

continuation variable	k
ordinary variables	$x, \ldots$
value variables	$v, w, \ldots$
answers	$P := KV \mid VAK \mid AK$
CPS-values	$V := \lambda x.\, \lambda k.\, P \mid v$
continuations	$K := k \mid \lambda v.\, P$
principal terms	$A := \lambda k.\, P \mid x.$

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 HOπ. Applying the compilation of HOπ into π-calculus, we obtain the encodings given in Table 9. We omit the correctness results, which are similar to those in Sections 6 and 7.

8.1. *Call-by-need*

Call-by-need is an implementation technique for call-by-name, which (in languages without side effects) avoids the inefficiency problems caused by repeated evaluations of copies of the argument of a function: 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. Call-by-need is usually presented as a reduction strategy on graphs, where it is easy to represent *sharing* of subterms.

As call-by-need is an implementation technique for call-by-name, its theory is related to that of call-by-name. The sets of λ-terms that have a normal form under call-by-need

Table 10. *The* (*core*) *simply-typed* λ*-calculus*

Terms	$M := x \mid c \mid \lambda x : T.\, M \mid MN$	$c \in$ base constants
Types	$T := T_1 \to T_2 \mid t$	$t \in$ base types
Type environments	$\Gamma := \emptyset \mid \Gamma, x : T$	
Typing rules	$\dfrac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x : S.\, M : S \to T}$ $\quad \dfrac{\Gamma(x) = T}{\Gamma \vdash x : T}$	
	$\dfrac{\Gamma \vdash M : S \to T \quad \Gamma \vdash N : S}{\Gamma \vdash MN : T}$	

and call-by-name coincide; and two λ-terms are behaviourally equivalent in call-by-need iff they also behaviourally equivalent in call-by-name.

We can modify the clauses for application in Table 9 to obtain the clause for call-by-need application:

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

We can explain this clause as follows. When $\mathscr{U}[\![M]\!]q$ becomes a function it signals that it has 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 $\mathscr{U}[\![N]\!]r$ 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.

It is not by chance that the call-by-need encoding is best derived from the uniform encoding of Table 9, 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 show the correctness of the call-by-need encoding, for it would require formally introducing the call-by-need system. See the notes of Section 10 for references.

9. Interpreting typed λ-calculi

In this section we show that the encodings of the previous section can be extended to encodings of typed λ-calculi. To do this we have to define translations on types to match those on terms. We analyse the case of the *simply-typed* λ*-calculus* in detail, and discuss subtyping and recursive types. For studies of other type systems, see the notes of Section 10.

The simply typed λ-calculus is presented in Table 10. For simplicity, we use only constants of base types. As usual, the arrow type associates to the right, so $T \to S \to U$ should be read as $T \to (S \to U)$. 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 also contains the constants. We call the typed versions of λV and λN (*simply-*) *typed call-by-value* ($\lambda V^{\rightarrow}$) and (*simply-*) *typed call-by-name* ($\lambda N^{\rightarrow}$), respectively.

We add the same base constants and base types to HOπ and π-calculus, and repeat the diagram of Figure 1, this time for $\lambda V^{\rightarrow}$ and $\lambda N^{\rightarrow}$. We show how to extend the encodings of λV and λN (from Sections 6 and 7), and their correctness results, to take account of types. The encodings in Section 8 can be extended similarly.

Lemma 9.1. In the simply-typed λ-calculus, for every Γ and M there is at most one T such that $\Gamma \vdash M : T$.

9.1. *The interpretation of typed call-by-value*

We begin with the left-hand part of Figure 1, which concerns $\lambda V^{\rightarrow}$. We follow a schema similar to that of Section 6, pointing out the main additions. As is usual when translating typed calculi, the definition of $\mathscr{C}_{\mathrm{V}}$ (Table 4) 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 $\mathscr{C}_{\mathrm{V}}$ on types, explained below. We have to add a clause for constants:

$$\mathscr{C}^*_{\mathrm{V}}[c]^{\Gamma} \overset{\mathrm{def}}{=} c.$$

It is important to understand how the CPS transform acts on types. Recalling from Section 6 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 $\lambda V^{\rightarrow}$-terms as follows:

$$\begin{aligned} \mathscr{C}_{\mathrm{V}}[\![T]\!] &\overset{\mathrm{def}}{=} (\mathscr{C}^*_{\mathrm{V}}[T] \rightarrow \diamond) \rightarrow \diamond \\ \mathscr{C}^*_{\mathrm{V}}[t] &\overset{\mathrm{def}}{=} t \qquad\qquad \text{if } t \text{ is a base type} \\ \mathscr{C}^*_{\mathrm{V}}[S \rightarrow T] &\overset{\mathrm{def}}{=} \mathscr{C}^*_{\mathrm{V}}[S] \rightarrow \mathscr{C}_{\mathrm{V}}[\![T]\!]. \end{aligned} \tag{15}$$

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

$$\mathscr{C}^*_{\mathrm{V}}[S \rightarrow T] = \mathscr{C}^*_{\mathrm{V}}[S] \rightarrow \neg\neg\mathscr{C}^*_{\mathrm{V}}[T].$$

Type environments are modified accordingly:

$$\begin{aligned} \mathscr{C}_{\mathrm{V}}[\![\varnothing]\!] &\overset{\mathrm{def}}{=} \varnothing \\ \mathscr{C}_{\mathrm{V}}[\![\Gamma, x : S]\!] &\overset{\mathrm{def}}{=} \mathscr{C}_{\mathrm{V}}[\![\Gamma]\!], x : \mathscr{C}_{\mathrm{V}}[\![S]\!]. \end{aligned}$$

and similarly for $\mathscr{C}^*_{\mathrm{V}}$. The correctness of this translation of types is given in the following Theorem.

Theorem 9.2. (Correctness of call-by-value CPS on types) If $M \in \Lambda$, then

$$\Gamma \vdash M : T \quad \text{implies} \quad \mathscr{C}^*_{\mathrm{V}}[\Gamma] \vdash \mathscr{C}_{\mathrm{V}}[\![M]\!]^{\Gamma} : \mathscr{C}_{\mathrm{V}}[\![T]\!].$$

(It follows that for any value V,

$$\Gamma \vdash V : T \quad \text{implies} \quad \mathscr{C}^*_{\mathrm{V}}[\Gamma] \vdash \mathscr{C}^*_{\mathrm{V}}[V]^{\Gamma} : \mathscr{C}^*_{\mathrm{V}}[T],$$

which shows the agreement between the definitions of the auxiliary function $\mathscr{C}_{\mathrm{V}}^{*}$ on terms and on types.)

Remark 9.3. Schema (15) is also useful for understanding the types of the CPS images of the untyped λV in (4), because the untyped λ-calculus can be described as a typed λ-calculus in which all terms have the recursive type

$$T \overset{\text{def}}{=} \mu X.(X \to X). \tag{16}$$

To apply the type translation to (16), we just need to add the clauses for type variables and for recursion to those in (15):

$$\mathscr{C}_{\mathrm{V}}^{*}[X] \overset{\text{def}}{=} X \quad \mathscr{C}_{\mathrm{V}}^{*}[\mu X.T] \overset{\text{def}}{=} \mu X.\mathscr{C}_{\mathrm{V}}^{*}[T]. \tag{17}$$

The translation of type T in (16) is then

$$\mathscr{C}_{\mathrm{V}}^{*}[T] = \mu X.(X \to (X \to \diamond) \to \diamond),$$

which is precisely type T_V in (4). Moreover

$$\begin{aligned} \mathscr{C}_{\mathrm{V}}[\![T]\!] &= (\mathscr{C}_{\mathrm{V}}^{*}[T] \to \diamond) \to \diamond \\ &= (T_V \to \diamond) \to \diamond \\ &= T_A. \end{aligned}$$

The grammar of CPS terms is obtained from Grammar 3 by adding a production for constants to those defining CPS-values. The relationship between the CPS grammar and HOπ 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 HOπ. Thus Theorem 9.2 can also be read as a result about the encoding of $\lambda V^{\to}$ into HOπ.

Finally, we apply the compilation $\mathscr{C}$ of HOπ terms and types into π-calculus terms and types, and obtain the encoding of $\lambda V^{\to}$ into typed π-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 i/o tag, and therefore $(\mathscr{V}[\![T]\!])^{-}$ is $\mathsf{o}\mathscr{V}^{*}[T]$, the translation of Theorem 9.2 into π-calculus gives the following corollary.

Corollary 9.4. (Correctness of $\mathscr{V}$ on types) Let M be a term of a simply typed λ-calculus. Then

$$\Gamma \vdash M : T \quad \text{implies} \quad \mathscr{V}^{*}[\Gamma], p : (\mathscr{V}[\![T]\!])^{-} \vdash \mathscr{V}[\![M]\!]^{\Gamma} p.$$

The results for the encoding of untyped λV, namely validity of β_{v}-rule and adequacy (Corollary 6.16 and 6.15), remain valid for the typed calculus, with the necessary modifications to the statements to take account of types. For instance, Corollary 6.16 becomes the following Corollary.

Corollary 9.5. (Validity of β_{v} theory) Suppose that $\Gamma \vdash M : T$ and $\Gamma \vdash N : T$, and let $H \overset{\text{def}}{=} \mathscr{V}^{*}[\Gamma]; (\mathscr{V}[\![T]\!])^{-}$. If $\lambda\beta_{\mathrm{v}} \vdash M = N$ then $\mathscr{V}[\![M]\!]^{\Gamma} \approx_{H} \mathscr{V}[\![N]\!]^{\Gamma}$.

Table 11. *The encoding of $\lambda V^{\rightarrow}$ into π-calculus (In this table we abbreviate $\mathcal{V}[\![M]\!]$ as $[\![M]\!]$, and for a type or type expression E, we abbreviate $\mathcal{V}^*[E]$ as $[E]$ and $\mathcal{V}[\![E]\!]$ as $[\![E]\!]$.)*

Translation of types:

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

$$[t] \stackrel{\text{def}}{=} t \qquad\qquad t \in \text{base types}$$

$$[S \to T] \stackrel{\text{def}}{=} [S]^\frown [\![T]\!] = \mathsf{o}([S] \times \mathsf{o}[T])$$

Translation of type environments:

$$[\![\varnothing]\!] = [\varnothing] \stackrel{\text{def}}{=} \varnothing$$

$$[\![\Gamma, x : S]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!], x : [\![S]\!]$$

$$[\Gamma, x : S] \stackrel{\text{def}}{=} [\Gamma], x : [S]$$

Translation of terms:

$$[\![\lambda x : S.\, M]\!]^\Gamma \stackrel{\text{def}}{=} (p).\, \overline{p}(y : [S \to T]^{\leftarrow\sharp}).\, !y(x)[\![M]\!]^{\Gamma, x:S}$$

$$[\![x]\!]^\Gamma \stackrel{\text{def}}{=} (p).\, \overline{p}x$$

$$[\![c]\!]^\Gamma \stackrel{\text{def}}{=} (p).\, \overline{p}c$$

$$[\![MN]\!]^\Gamma \stackrel{\text{def}}{=}$$
$$(p).\, (\nu q : \sharp[S \to T])\, \big([\![M]\!]^\Gamma q \mid q(x).\, (\nu r : \sharp[S])([\![N]\!]^\Gamma r \mid r(y).\, \overline{x}\langle y, p\rangle)\big)$$

where in the clause for abstraction, T is the unique type such that $\Gamma \vdash \lambda x : S.\, M : T$, and, similarly, in the clause for application, $S \to T$ is the unique type such that $\Gamma \vdash M : S \to T$ (these types are unique by Lemma 9.1).

Remark 9.6. The types of π-calculus names used for the encoding of untyped λV in Section 6 agree with those used for $\lambda V^{\rightarrow}$ in this section, when we view the untyped λ-calculus as a typed λ-calculus where the only type is $\mu X.\, X \to X$ (Remark 9.3). From (17), the translation of recursive types and type variables of λV into π-calculus is

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

Therefore type Val of Section 6 is precisely $\mathcal{V}^*[\mu X.\, (X \to X)]$, so Lemma 6.14 can be presented thus: if $\text{fv}(M) \subseteq \widetilde{x}$, then

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

Remark 9.7. (Subtyping) In typed λ-calculi with subtyping, the arrow type is contravariant in the first argument and covariant in the second. The i/o tag o is a contravariant type constructor. In the translation of an arrow type $S \to T$, component $\mathcal{V}^*[S]$ is in contravariant position, because it is underneath an odd number of o tags; in contrast

$\mathscr{V}^*[T]$ is in covariant position, because it is underneath an even number of o tags. Therefore the π-calculus translation of types correctly explains the subtyping rule for arrow type. As a consequence, the translation of this section can be extended to one of $\lambda V^{\rightarrow}$ with subtyping.

9.2. *The interpretation of typed call-by-name*

The work presented for call-by value can be repeated for call-by-name. We only show how the CPS modifies types, and the analogue of Theorem 9.2 and Corollary 9.4.

$$\mathscr{C}_{\mathrm{N}}[\![T]\!] \stackrel{\mathrm{def}}{=} (\mathscr{C}^*_{\mathrm{N}}[T] \rightarrow \diamond) \rightarrow \diamond$$

$$\begin{aligned} \mathscr{C}^*_{\mathrm{N}}[t] &\stackrel{\mathrm{def}}{=} t && \text{if } t \text{ is a base type} \\ \mathscr{C}^*_{\mathrm{N}}[S \rightarrow T] &\stackrel{\mathrm{def}}{=} \mathscr{C}_{\mathrm{N}}[\![S]\!] \rightarrow \mathscr{C}_{\mathrm{N}}[\![T]\!]. \end{aligned}$$

Theorem 9.8. (Correctness of call-by-name CPS on types) Let $M \in \Lambda$. Then

$$\Gamma \vdash M : T \quad \text{implies} \quad \mathscr{C}_{\mathrm{N}}[\![\Gamma]\!] \vdash \mathscr{C}_{\mathrm{N}}[\![M]\!]^{\Gamma} : \mathscr{C}_{\mathrm{N}}[\![T]\!].$$

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

$$\Gamma \vdash V : T \quad \text{implies} \quad \mathscr{C}_{\mathrm{N}}[\![\Gamma]\!] \vdash \mathscr{C}^*_{\mathrm{N}}[V]^{\Gamma} : \mathscr{C}^*_{\mathrm{N}}[T].$$

The final encoding of types, type environments and terms of $\lambda N^{\rightarrow}$ into π-calculus is given in Table 12.

Corollary 9.9. (Correctness of $\mathscr{N}$ on types) Let M be a term of a simply typed λ-calculus. Then

$$\Gamma \vdash M : T \quad \text{implies} \quad \mathscr{N}[\![\Gamma]\!], p : (\mathscr{N}[\![T]\!])^{-} \vdash \mathscr{N}[\![M]\!]^{\Gamma} p.$$

10. Historical notes

The standard references on the classical theory of the λ-calculus (the *sensible theory*) are: Barendregt (1984), and Hindley and Seldin (1986). The textbooks: Gunter (1992), Hindley (1997), and Mitchell (1996) contain detailed presentations of typed λ-calculi and PCF.

The term 'continuation' is due to Strachey and Wadsworth (1974), who used them to give semantics to control jumps. See Reynolds (1993) for a history of the discovery of continuations and CPS transforms. For continuations in denotational semantics, see Gordon (1979), Schmidt (1986) or Tennent (1991). For continuations as a programming technique, see Friedman *et al.* (1992). For the use of CPS transform in compilers, see Appel (1992), where the language is ML, or Friedman *et al.* (1992), where the language is Scheme. The call-by-value CPS transform of Table 4 is due to Fischer (1972) (of which a more complete version is Fischer (1993)). The call-by-name CPS transform of Table 6 is that of Plotkin (1975), based on work by Reynolds (such as Reynolds (1972); however, we have adopted the rectification in the clause for variables due to Hatcliff and Danvy (1997) (Plotkin's translation for variables was $\mathscr{C}_{\mathrm{N}}[\![x]\!] \stackrel{\mathrm{def}}{=} x$; the rectification is necessary for the left-to-right implication of Theorem 7.4 to hold).

Table 12. *The encoding of $\lambda N^{\rightarrow}$ into the π-calculus (In this table we abbreviate $\mathcal{N}[\![M]\!]$ as $[\![M]\!]$, and for a type or type expression E, we abbreviate $\mathcal{N}^*[E]$ as $[E]$ and $\mathcal{N}[\![E]\!]$ as $[\![E]\!]$.)*

Translation of types:

$$\begin{aligned} [\![T]\!] &\overset{\text{def}}{=} \mathsf{oo}[T] \\ [t] &\overset{\text{def}}{=} t \qquad\qquad t \in \text{base types} \\ [S \to T] &\overset{\text{def}}{=} [\![S]\!] \frown [\![T]\!] = \mathsf{o}([\![S]\!] \times \mathsf{o}[T]) \end{aligned}$$

Translation of type environments:

$$\begin{aligned} [\![\emptyset]\!] = [\emptyset] &\overset{\text{def}}{=} \emptyset \\ [\![\Gamma, x : S]\!] &\overset{\text{def}}{=} [\![\Gamma]\!], x : [\![S]\!] \\ [\Gamma, x : S] &\overset{\text{def}}{=} [\Gamma], x : [S] \end{aligned}$$

Translation of terms:

$$\begin{aligned} [\![\lambda x : S.\, M]\!]^{\Gamma} &\overset{\text{def}}{=} (p).\, \overline{p}(v : [S \to T]^{\leftarrow\sharp}).\, v(x)[\![M]\!]^{\Gamma, x:S} \\ [\![x]\!]^{\Gamma} &\overset{\text{def}}{=} (p).\, \overline{x}p \\ [\![c]\!]^{\Gamma} &\overset{\text{def}}{=} (p).\, \overline{p}c \\ [\![MN]\!]^{\Gamma} &\overset{\text{def}}{=} \end{aligned}$$

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

where in the clause for abstraction, T is the unique type such that $\Gamma \vdash \lambda x : S.\, M : T$, and, similarly, in the clause for application, $S \to T$ is the unique type such that $\Gamma \vdash M : S \to T$.

Theorems 6.1–6.4 and 7.1–7.3, on CPS terms and CPS transforms, are proved in Plotkin (1975). (The assertions of these theorems is actually slightly different from Plotkin's, but the content and the proofs are similar.) Plotkin also presents counterexample (2) to the converse of Theorem 6.5. We did not find in the literature grammars for the terms of the CPS transforms (Grammars (3), (9), (14)) but grammars related to ours are found in Ogata (1998), which compares cut elimination of certain logics and CPS transforms, and Sabry and Felleisen (1993), which 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 8.

The relationship between types of λ-terms and types of their CPS images was first noted by Meyer and Wand (1985). Other important papers on types and CPS are Murthy (1992), Harper and Lillibridge (1993) and Harper *et al.* (1993). Theorem 9.2 is due to Meyer and Wand. The translation of arrow types is sometimes called 'double-negation construction' after Murthy (1992). The transformation of types for the call-by-name CPS and Theorem 9.8, are by Harper and Lillibridge (1993), 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* Hewitt (1977); Hewitt *et al.* 1973; Hewitt and Baker (1977). The analogy between Milner's encodings of λ-calculus into π-calculus and the CPS transforms was noticed by several people, and was first partly formalised by Boudol (1997) and Thielecke (1997). Boudol compares encodings of call-by-name and call-by-value λ-calculus into, respectively, the blue calculus and the π-calculus. He noticed that, for either strategy, if the CPS transform is composed with the encoding of (call-by-name) λ-calculus into the blue calculus, then the results can be read as the standard encoding of that λ-calculus strategy into the π-calculus. Thielecke introduces a CPS calculus, similar to the intermediate language in Appel's compiler Appel (1992). He shows that this CPS calculus has a simple translation into the π-calculus and that, if Plotkin's CPS transforms are formulated in the CPS calculus, their translations into the π-calculus yield an encoding similar to Milner's Milner (1992). In this paper we go further, in that the encodings of both call-by-name and call-by-value λ-calculus into the π-calculus are *factorised* using the CPS transforms, and the compilation $\mathscr{C}$ of HOπ into π-calculus, for both untyped and typed λ-calculi. One of the reasons for working out the factorisations in detail is to be able to *derive* the correctness of the π-calculus encodings (on terms as well as on types) from those of the CPS transforms and of the compilation of HOπ into π-calculus.

Translations of functions into process calculi have been given by Kennaway and Sleep (1982), Leth (1991), Thomsen (1990) and Boudol (1989). Robin Milner's work on functions as π-calculus processes Milner (1992) is a landmark in the area. Milner's encodings are, essentially, those of Sections 6 and 7. (In the encoding of call-by-value, our clause for variable is simpler, which is possible because we use `i/o` types – see Remark 6.17; in call-by-name, Milner's encoding does not have output particles at p and q of Table 7, which we need because we use asynchronous π-calculi and asynchronous behavioural equivalence; in a synchronous barbed congruence input prefixes are also observable.) The uniform encoding in Section 8 is from Ostheimer and Davie (1993). Niehren (1996) uses encodings of call-by-name, call-by-value and call-by-need λ-calculi into π-calculus to compare the time complexity of the strategies.

Call-by-need was proposed by Wadsworth (1971) as an implementation technique. Formalisations of call-by-need on a λ-calculus with a `let` construct or with environments include Ariola *et al.* (1995), Launchbury (1993), Purushothaman and Seaman (1992) and Yoshida (1993). A study of the correctness of the call-by-need encoding in Section 8 is in Brock and Ostheimer (1995). Encodings of graph reductions, related to call-by-need, into π-calculus are given in Boudol (1994) and Jeffrey (1993) but their correctness is not studied.

Turner first established a relationship between the types of λ-calculus terms and those of their encodings into π-calculus Turner (1996). He takes (variants of) Milner's encodings of the λ-calculus into the π-calculus and proves that for some of these encodings there is a correspondence between principal types of the λ-terms and principal types of the encoding π-calculus terms; the π-calculus used is the polyadic one, without `i/o` types, plus polymorphism. Turner also extends Milner's encodings to the polymorphic λ-calculus. Using `i/o` types, as we have done in the paper, the relationship between λ-calculus and

π-calculus types is clearer and sharper, and can be easily extended to other type systems. The work presented in Section 9 follows the schema of the interpretation Abadi and Cardelli's *types object calculus* Abadi and Cardelli (1996) into π-calculus in Sangiorgi (1992b). Encodings of simply-typed λ-calculi are also given by Kobayashi (1998), to illustrate the use of a type system guaranteeing absence of deadlocks.

An interesting question on encodings of λ-calculi into π-calculi is the characterisation of the equivalence induced on λ-terms by the encodings: when are the encoding of two λ-terms behaviourally equivalent π-calculus processes? The answer to this question is known for the call-by-name encodings (Sangiorgi 1992b; Sangiorgi 1995; Boudol and Laneve (1995)), but is unknown for call-by-value.

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

References

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*, Monographs in Computer Science, Springer-Verlag.

Ariola, Z., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need λ-calculus. In: *Proc. 22th POPL*, ACM Press.

Appel, A. (1992) *Compiling with Continuations*, Cambridge University Press.

Barendregt, H. (1984) The Lambda Calculus: Its Syntax and Semantics. *Studies in Logic* **103**, North Holland. (Revised edition).

Boudol, G. (1989) Towards a lambda calculus for concurrent and communicating systems. In: TAPSOFT '89. *Springer-Verlag Lecture Notes in Computer Science* **351** 149–161.

Boudol, G. (1994) Some chemical abstract machines. In: Proc. REX Summer School/Symposium 1993. *Springer-Verlag Lecture Notes in Computer Science* **803**.

Boudol, G. (1997) The pi-calculus in direct style. In: *Proc. 24th POPL*, ACM Press.

Boudol, G. and Laneve, C. (1995) λ-calculus, multiplicities and the π-calculus. Technical Report RR-2581, INRIA-Sophia Antipolis. To appear in 'Festschrift volume in honor of Robin Milner's 60th birthday', MIT Press.

Brock, S. and Ostheimer, G. (1995) Process semantics of graph reduction. In: Lee, I. and Smolka, S. A. (eds.) Proc. CONCUR '95. *Springer-Verlag Lecture Notes in Computer Science* **962** 471–485.

Fischer, M. J. (1972) Lambda-calculus schemata. In: *Proc. ACM conf. on Proving Assertions about Programs* 104–109.

Fischer, M. J. (1993) Lambda-calculus schemata. *Lisp and Symbolic Computation* **6** 259–288.

Friedman, D. P., Wand, M. and Haynes, C. T. (1992) *Essentials of Programming Languages*, McGraw-Hill Book Co., New York.

Gordon, M. J. C. (1979) *The denotational description of programming languages*, Springer-Verlag.

Gunter, C. A. (1992) *Semantics of Programming Languages*, MIT Press.

Harper, R., Duba, F. B. and MacQueen, D. (1993) Typing first-class continuations in ML. *Journal of Functional Programming* **3** (4) 465–484.

Harper, R. and Lillibridge, M. (1993) Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation* **6** 361–380.

Hatcliff, J. and Danvy, O. (1997) Thunks and the λ-calculus. *Journal of Functional Programming* **7** (3) 303–319.

Hewitt, C. (1977) Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence* **8** (3) 323–364.

Hewitt, C. and Baker, H. (1977) Laws for communicating parallel processes. In: *1977 IFIP Congress Proceedings*, IFIP 987–992.

Hewitt, C., Bishop, P., Greif, I., Smith, B., Matson, T. and Steiger, R. (1973) Actor induction and meta-evaluation. In: *ACM Symposium on Principles of Programming Languages*, ACM 153–168.

Hindley, J. R. (1997) *Basic simple type theory*, Cambridge University Press.

Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinators and λ-calculus*, Cambridge University Press.

Jeffrey, A. (1993) A chemical abstract machine for graph reduction. In: Proc. Ninth International Conference on the Mathematical Foundations of Programming Semantics (MFPS '93). *Springer-Verlag Lecture Notes in Computer Science* **802**.

Kennaway, J. R. and Sleep, M. R. (1982) Expressions as processes. In: *ACM Conference on LISP and Functional Programming*, ACM 21–28.

Kobayashi, N. (1998) A partially deadlock-free typed process calculus. *TOPLAS* **20** (2) 436–482. A preliminary version in *Twelfth Annual IEEE Symposium on Logic in Computer Science* 128–139.

Kobayashi, N., Pierce, B. C. and Turner, D. N. (1996) Linearity and the pi-calculus. In: *Proc. 23rd POPL*, ACM Press.

Launchbury, J. (1993) A natural semantics for lazy evaluation. In: *Proc. 20th POPL*, ACM Press.

Leth, L. (1991) *Functional Programs as Reconfigurable Networks of Communicating Processes*, Ph.D. thesis, Imperial College, University of London.

Meyer, A. R. and Wand, M. (1985) Continuation semantics in typed lambda-calculi. In: Parikh R. (ed.) Proceedings of the Conference on Logic of Programs. *Springer-Verlag Lecture Notes in Computer Science* **193** 219–224.

Milner, R. (1991) The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in: Bauer, F. L., Brauer, W. and Schwichtenberg, H. (eds.) *Logic and Algebra of Specification*, Springer-Verlag.

Milner, R. (1992) Functions as processes. *Mathematical Structures in Computer Science* **2** (2) 119–141.

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

Milner, R. and Sangiorgi, D. (1992) Barbed bisimulation. In: Kuich, W. (ed.) 19th ICALP. *Springer-Verlag Lecture Notes in Computer Science* **623** 685–695.

Mitchell, J. C. (1996) *Foundations for Programming Languages*, MIT Press.

Murthy, C. (1992) A computational analysis of Girard's translation and LC. In: *7th LICS Conf.*, IEEE Computer Society Press.

Niehren, J. (1996) Functional computation as concurrent computation. In: *Proc. 23rd POPL*, ACM Press.

Ogata, I. (1998) Cut elimination for classical proofs as continuation passing style computation. In: Proc. ASIAN '98. *Springer-Verlag Lecture Notes in Computer Science* (to appear).

Ostheimer, G. K. and Davie, A. J. T. (1993) Pi-calculus characterizations of some practical lambda-calculus reduction strategies. CS 93/14, University of St Andrews, Scotland.

Pierce, B. and Sangiorgi, D. (1996) Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science* **6** (5) 409–454. (An extended abstract in *Proc. LICS '93*, IEEE Computer Society Press.)

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

Purushothaman, S. and Seaman, J. (1992) An adequate operational semantics for sharing in lazy evaluation. In: Krieg-Brückner, B. (ed.) ESOP '92. *Springer-Verlag Lecture Notes in Computer Science* **582** 435–450.

Reynolds, J. C. (1972) Definitional interpreters for higher order programming languages. *ACM Conference Proceedings* 717–740.

Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbolic Computation* **6** 233–248.

Sabry, A. and Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* **6** 289–360.

Sangiorgi, D. (1992a) *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*, Ph.D. thesis CST-99-93, Department of Computer Science, University of Edinburgh.

Sangiorgi, D. (1992b) The lazy lambda calculus in a concurrency scenario. In: *7th LICS Conf.*, IEEE Computer Society Press 102–109.

Sangiorgi, D. (1995) 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.

Sangiorgi, D. (1998a) Asynchronous process calculi: the first-order and higher-order paradigms (tutorial). To appear in *Theoretical Computer Science*.

Sangiorgi, D. (1998b) An interpretation of typed objects into typed π-calculus. *Information and Computation* **143** (1) 34–73.

Schmidt, D. A. (1986) *Denotational Semantics – A methodology for language development*, Allyn and Bacon.

Strachey, C. and Wadsworth, C. P. (1974) Continuations: A mathematical semantics for handling full jumps. Technical Report Technical Monograph PRG-11, Oxford University Computer Laboratory.

Tennent, R. D. (1991) *Semantics of Programming Languages*, Prentice Hall, New York.

Thielecke, H. (1997) *Categorical Structure of Continuation Passing Style*, Ph.D. thesis, University of Edinburgh. Also available as technical report ECS-LFCS-97-376.

Thomsen, B. (1990) *Calculi for Higher Order Communicating Systems*, Ph.D. thesis, Department of Computing, Imperial College, University of London.

Turner, N. D. (1996) *The polymorphic pi-calculus: Theory and Implementation*, Ph.D. thesis, Department of Computer Science, University of Edinburgh.

Vasconcelos, V. T. and Honda, K. (1993) Principal typing schemes in a polyadic π-calculus. In: Best, E. (ed.) Proc. CONCUR '93. *Springer-Verlag Lecture Notes in Computer Science* **715**.

Wadsworth, C. P. (1971) *Semantics and pragmatics of the lambda calculus*, Ph.D. thesis, University of Oxford.

Yoshida, N. (1993) Optimal reduction in weak lambda-calculus with shared environments. In: *Proc. of FPCA'93, Functional Programming and Computer Architecture* 243–252.