Control in the π-Calculus

Kohei Honda (kohei@dcs.qmul.ac.uk)
Queen Mary, Department of Computer Science

Nobuko Yoshida (yoshida@doc.ic.ac.uk)
Imperial College London, Department of Computing

Martin Berger (m.f.berger@sussex.ac.uk)
Department of Informatics, University of Sussex

Abstract. This paper presents a type-preserving translation from the call-by-value $\lambda\mu$-calculus ($\lambda\mu$v-calculus) into a typed π-calculus, and shows full abstraction up to maximally consistent observational congruences in both calculi. The $\lambda\mu$-calculus has a particularly simple representation as typed mobile processes where a unique stateless replicated input is associated to each name. The corresponding π-calculus is a proper subset of the linear π-calculus, the latter being able to embed the simply-typed λ-calculus fully abstractly. Strong normalisability of the $\lambda\mu$v-calculus is an immediate consequence of this correspondence and the strong normalisability of the linear π-calculus, using the standard argument based on simulation between the $\lambda\mu$v-calculus and its translation. Full abstraction, our main result, is proved via an inverse transformation from the typed π-terms which inhabit the encoded $\lambda\mu$v-types into the $\lambda\mu$v-calculus (the so-called definability argument), using proof techniques from games semantics and process calculi. A tight operational correspondence assisted by the definability result opens a possibility to use typed π-calculi as a tool to investigate and analyse behaviours of various control operators and associated calculi in a uniform setting, possibly integrated with other language primitives and operational structures.

Keywords: Control, Mobile Processes, Types, Semantics, Full Abstraction, Definability, the $\lambda\mu$-calculus, the π-calculus

1. Introduction

This paper presents a type-preserving translation from the call-by-value $\lambda\mu$-calculus ($\lambda\mu$v-calculus) [27] into a typed π-calculus, and shows it is fully abstract up to maximally consistent observational congruences in respective calculi. Full abstraction is proved via an inverse transformation from the typed π-terms which inhabit the $\lambda\mu$v-types into the $\lambda\mu$v-calculus [27] (the so-called definability argument), using proof techniques coming from games semantics and process calculi.

While there are different notions of control which would be represented as distinct forms of typed interactions in the π-calculus, surprisingly, full control, the $\lambda\mu$-calculus originally introduced by Parigot [28], the call-by-value version of which was later studied by Ong and Stewart [27], has a particularly simple representation as typed name passing processes; processes used for the embedding are exactly characterised as a proper subset of the linear π-calculus introduced in [39], with a clean characterisation in types and behaviour. The linear π-calculus can embed, among others, the simply typed λ-calculus full abstractly. The subset of the linear π-calculus which corresponds to full control, here called the π^{c}-calculus ("c" indicates control), is restricted in that each channel is used only for a unique stateless replicated input and for zero or more dual outputs, as well as precluding circular dependency between channels. In spite of its simplicity, both, call-by-value and call-by-name full control, are precisely embeddable into this π^{c}-calculus by changing translation of types. Because the π^{c}-calculus is a proper subset of the linear π-calculus, many of the known results about the linear π-calculus, as studied in [39], can be carried over to the π^{c}-calculus. This can be used for establishing properties of the $\lambda\mu$v-calculus. For example, strong normalisability of the π^{c}-calculus is an immediate consequence of the same property of the linear π-calculus, and that can be used for showing strong normalisability of the $\lambda\mu$v-calculus by a standard argument [24] based on a simulation in reduction. We believe a tight operational and equational correspondence assisted by formal embedding results including definability, as will be explored in the present paper, opens a possibility to use typed π-calculi as a tool to investigate and analyse behaviours of various control operators and calculi in a uniform setting, possibly integrated with other language primitives and operational structures. After studying the call-by-value $\lambda\mu$-calculus, we also demonstrate applicability of our framework by an embedding of the call-by-name $\lambda\mu$-calculus into the same π^{c}-calculus by changing translation of types.

In the rest of the paper, Section 2 introduces the π-calculus for control. Section 3 presents the embedding of the call-by-value $\lambda\mu$-calculus (the $\lambda\mu$v-calculus), shows it is type- and dynamics-preserving, and illustrates how the process embedding elucidates the behaviour of $\lambda\mu$v-terms using typical control operators. Section 4 presents the decoding of processes typed with the encoded $\lambda\mu$v-types, establishes definability using the decoding, and establishes the equational full abstraction of the process encoding of $\lambda\mu$v-terms. The paper concludes with discussions, where we outline the call-by-name encoding, related work, and open issues. Some proofs are delegated to the Appendix.

2. Processes and Types

2.1. Processes

The π-calculus used in this paper is a subset of the standard asynchronous π-calculus [14, 24, 25]. The following is the key reduction rule of this calculus.

$$x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \;\longrightarrow\; P\{\vec{v}/\vec{y}\} \tag{1}$$

Here $\vec{y} = y_1...y_n$ denotes a potentially empty vector, $\mid$ denotes parallel composition, $x(\vec{y}).P$ is an input (or a receptor), and $\overline{x}\langle\vec{v}\rangle$ is an asynchronous output (or a message). Operationally, this reduction represents the consumption of an asynchronous message by a receptor. The idea extends to a replicated receptor, $!\,x(\vec{y}).P$:

$$!\,x(\vec{y}).P \mid \overline{x}\langle\vec{v}\rangle \;\longrightarrow\, !\,x(\vec{y}).P \mid P\{\vec{v}/\vec{y}\}, \tag{2}$$

where the replicated process remains in the configuration after reduction, unlike in (1). The π-calculus used in this paper is the above π-calculus but without linear input prefixes: hence we only have (2) as the communication rule.

Types for processes prescribe usage of names. To be able to do this with precision, it is important to control dynamic sharing of names. For this purpose, it is useful to restrict name passing to *bound (private) name passing*, where only bound names are passed in interaction. This allows tighter control of sharing without losing essential expressiveness, making it easier to administer name usage in more stringent ways. The resulting calculus is an asynchronous version of the πI-calculus [32] and has expressive power equivalent to the calculus with free name passing (for the result in the typed setting, see [39]). In the present study, the restriction to bound name passing leads to, among others, a clean inverse transformation from the π-calculus into the $\lambda\mu$-calculus.

Syntactically we restrict an output to the form $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|P)$ (where names in $\vec{y}$ are pairwise distinct), which we henceforth write $\overline{x}(\vec{y})P$. For dynamics, we need several straightforward communication rules, only one of which is relevant later for the typed calculus. It is the rule corresponding to (2) which now has the following form by the restriction to bound output[1]:

$$!\,x(\vec{y}).P \,|\, \overline{x}(\vec{y})Q \;\longrightarrow\, !\,x(\vec{y}).P \,|\, (\boldsymbol{\nu}\,\vec{y})(P \,|\, Q) \tag{3}$$

[1] To be precise, we also need:

$$\overline{x}(\vec{y})(P|!x(\vec{v}).Q) \longrightarrow (\nu\vec{y})(P|!x(\vec{v}).Q|Q\{\vec{y}/\vec{v}\}).$$

to simulate the whole of the untyped reduction of the free name passing calculus (cf. [39, Proof of Proposition 4.2]). However, as we shall see soon, this rule does not apply to typable processes, hence can be ignored in the typed setting.

Note "$\overline{x}(\vec{y})Q$" indicates that $\overline{x}(\vec{y})$ is an asynchronous output exporting $\vec{y}$ which are originally local to Q. After communication, $\vec{y}$ are shared between P and Q. To ensure asynchrony of outputs, we add the following rule to the standard closure rules for $\mid$ and $(\boldsymbol{\nu}\, x)$.

$$P \longrightarrow P' \quad \Rightarrow \quad \overline{x}(\vec{y})\, P \longrightarrow \overline{x}(\vec{y})\, P'$$

Further, for the generation of the structural congruence used for defining the reduction, the following structural rules are added to the standard rules, allowing inference of interaction under an output prefix.

$$\overline{x}(\vec{z})\,(P|Q) \equiv (\overline{x}(\vec{z})\,P)|Q \qquad \text{if } \mathsf{fn}(Q) \cap \{\vec{z}\} = \emptyset, \tag{4}$$

$$\overline{x}(\vec{z})\,(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)\overline{x}(\vec{z})\,P \qquad \text{if } y \notin \{x, \vec{z}\}. \tag{5}$$

where $\mathsf{fn}(Q)$ means the set of free names in Q. By these rules, we maintain the same dynamics as in the original asynchronous calculus, projected onto the restricted and typed syntax. We show a simple example of reductions with bound output.

$$\begin{array}{rcl} !\, x(y).(\overline{y} \mid \overline{y}) \mid \overline{x}(y)!y.0 & \longrightarrow & !\, x(y).(\overline{y} \mid \overline{y}) \mid (\boldsymbol{\nu}\, y)(\overline{y} \mid \overline{y} \mid !y.0) \\ & \longrightarrow^2 & !\, x(y).(\overline{y} \mid \overline{y}) \mid (\boldsymbol{\nu}\, y)!y.0 \end{array}$$

Above $(\boldsymbol{\nu}\, y)!y.0$ is garbage in the sense that no further interaction is possible, so that process has the same meaning as 0 up to the standard observational congruences [14, 24, 25], for example untyped strong bisimilarity. Thus the final configuration above is behaviourally the same thing as having only a single replicated process $!\, x(y).(\overline{y} \mid \overline{y})$.

Let us present the formal grammar of the calculus below.

$$P \ ::= \ !\, x(\vec{y}).P \ \mid \ \overline{x}(\vec{y})P \ \mid \ P|Q \ \mid \ (\boldsymbol{\nu}\, x)P \ \mid \ \mathbf{0}$$

The initial x in $!\, x(\vec{y}).P$ and $\overline{x}(\vec{y})P$ is often called *subject*. We write $!x.P$ for $!x(\epsilon).P$ and $\overline{x}P$ for $\overline{x}(\epsilon)P$, where ϵ denotes the empty vector. $(\boldsymbol{\nu}\, x)P$ is name hiding and $\mathbf{0}$ denotes nil. The full definition of the reduction rules and the structure rules is found in Figure 2.1 ($\longrightarrow$ is generated from the given rules; $\equiv$ is generated from the given rules together with the closure under all contexts). We denote $\twoheadrightarrow$ as $\longrightarrow^* \cup \equiv$.

2.2. Types and Typing (1): Basic Idea

A central idea for precisely embedding functional computation in the π-calculus used in the present paper is to restrict process behaviour to a *deterministic, sequential* one. To realise this idea, the following three simple conditions are ensured by a type discipline.

- For each name there is a unique stateless replicated input with zero or more dual outputs.

(Structural Rules)

$$\begin{array}{llll}
(\mathsf{S0}) & P \equiv Q \quad \text{if } P \equiv_\alpha Q & & \\
(\mathsf{S1}) & P|\mathbf{0} \equiv P & (\mathsf{S2}) & P|Q \equiv Q|P \\
(\mathsf{S3}) & P|(Q|R) \equiv (P|Q)|R & & \\
(\mathsf{S4}) & (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & (\mathsf{S5}) & (\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P \\
(\mathsf{S6}) & (\boldsymbol{\nu}\, x)(P|Q) \equiv ((\boldsymbol{\nu}\, x)P)|Q & (x \notin \mathsf{fn}(Q)) & \\
(\mathsf{S7}) & \overline{x}(\vec{y})\overline{z}(\vec{w})P \equiv \overline{z}(\vec{w})\overline{x}(\vec{y})P & (x, z \notin \{\vec{w}\vec{y}\}) & \\
(\mathsf{S8}) & (\boldsymbol{\nu}\, z)\overline{x}(\vec{y})P \equiv \overline{x}(\vec{y})(\boldsymbol{\nu}\, z)P & (z \notin \{x\vec{y}\}) & \\
(\mathsf{S9}) & \overline{x}(\vec{y})(P|Q) \equiv (\overline{x}(\vec{y})P)|Q & (\{\vec{y}\} \cap \mathsf{fn}(Q) = \emptyset) &
\end{array}$$

(Reduction)

$$\begin{array}{rl}
(\mathsf{Com}_!) & !\, x(\vec{y}).P \mid \overline{x}(\vec{y})Q \longrightarrow !\, x(\vec{y}).P|(\boldsymbol{\nu}\, \vec{y})(P|Q) \\
(\mathsf{Res}) & P \longrightarrow Q \implies (\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)Q \\
(\mathsf{Par}) & P \longrightarrow P' \implies P|Q \longrightarrow P'|Q \\
(\mathsf{Out}) & P \longrightarrow Q \implies \overline{x}(\vec{y})P \longrightarrow \overline{x}(\vec{y})Q \\
(\mathsf{Cong}) & P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q
\end{array}$$

Figure 1. Reduction and Structural Rules .

- Channels have no circular dependency.
- Only one single thread (output) can be active at one time, at present and potentially.

For example, by the first condition,

$$P_1 \stackrel{\text{def}}{=} !\, b.\overline{a} \mid !\, b.\overline{c} \tag{6}$$

should be untypable because b is associated to two replicators, but

$$P_2 \stackrel{\text{def}}{=} !\, b.\overline{a} \mid \overline{b} \mid !\, c.\overline{b} \tag{7}$$

is typable since, while output at b appears twice, replicated input at b appears only once. Also by the second condition,

$$P_3 \stackrel{\text{def}}{=} !\, b.\overline{a} \mid !\, a.\overline{b} \tag{8}$$

is untypable: we can easily observe if we compose message $\overline{a}$ to the above process, then the computation does not terminate. Finally by the third condition, the following two processes

$$P_4 \stackrel{\text{def}}{=} \overline{a} \mid \overline{a} \quad \text{and} \quad P_5 \stackrel{\text{def}}{=} !b.(\overline{a} \mid \overline{c}) \tag{9}$$

are both untypable since two threads (outputs) are/would be running in parallel. But

$$P_6 \stackrel{\text{def}}{=} !a.\overline{b} \mid !b.\overline{c} \mid !e.\overline{c} \mid \overline{a} \tag{10}$$

is typable though c appears twice. The resulting typed processes may look too restricted: however it is sufficient to embed the full control fully abstractly. The three conditions informally specified above are formally guaranteed by a simple typing system, which we shall introduce in the next two paragraphs.

2.3. Types and Typing (2): Types

First we introduce the syntax of *channel types.* They indicate possible usage of channels.

$$\tau \ ::= \ (\vec{\tau})^p \qquad\qquad p \ ::= \ \mathbf{!} \ \mid \ \mathbf{?}$$

$\tau, \tau', ..$ (resp. $p, p', ..$) range over types (resp. modes). $\mathbf{!}$ and $\mathbf{?}$ are called *server* mode and *client* mode, respectively, and they are *dual* to each other. Here, by server we mean that the process is waiting with an input to be invoked. Conversely, a process is a client, if its next action is sending a message to a server. We write $\mathsf{md}(\tau)$ for the outermost mode of τ. For example, $\mathsf{md}((\tau_1\,\tau_2)^{\mathbf{!}}) = \mathbf{!}$. We write $()^p$ for $(\epsilon)^p$, which stands for a channel that carries no names. We further demand the following condition to hold for channel types.

DEFINITION 2.1. A channel type τ is *IO-alternating* if, for each of its subexpression $(\tau_1..\tau_n)^p$, if $p = \mathbf{!}$ (resp. $p = \mathbf{?}$) then each $\mathsf{md}(\tau_i) = \mathbf{?}$ (resp. $\mathsf{md}(\tau_i) = \mathbf{!}$). *Hereafter we assume all channel types we use are IO-alternating.*

For example, $(()^{\mathbf{?}})^{\mathbf{!}}$ is IO-alternating but $(()^{\mathbf{!}})^{\mathbf{!}}$ is not. The channel types combined with the IO-alternation in Definition 2.1 is a subset of channel types in [39], given by taking off the linear modes.

As a simple example, a type

$$(\tau_1\tau_2)^{\mathbf{!}}$$

indicates that a channel with this type should be used as a replicated input through which a process would input two channels typed as τ_1 and τ_2, respectively.

The *dual of* τ, written $\overline{\tau}$, is defined as the result of dualising all modes in τ_i. For example, $(\overline{\tau_1}\,\overline{\tau_2})^{?}$ is the dual of the above type.

To guarantee the uniqueness of a server (replicated) process, we introduce the partial operation $\odot$ on types, which is generated from:

$$\tau \odot \overline{\tau} = \overline{\tau} \odot \tau = \overline{\tau} \quad \text{and} \quad \tau \odot \tau = \tau \quad \text{with} \quad (\mathsf{md}(\tau) = \mathbf{?})$$

Note $\odot$ is indeed partial since it is not defined in other cases. This operation means that a server should be unique, but an arbitrary number of clients can request interactions. For example,

$$(()^{?})^{!} \odot (()^{!})^{?} = (()^{!})^{?} \odot (()^{?})^{!} = (()^{?})^{!} \qquad (()^{!})^{?} \odot (()^{!})^{?} = (()^{!})^{?}$$

This law can be used for prohibiting the process such as P_1 in (6) since the process becomes untypable due to the undefinedness of the operation $()^{!} \odot ()^{!}$.

To guarantee the second condition, we introduce an *action type* ranged over by $A, B, C....$ The syntax is given as follows:

$$A \;::=\; \emptyset \mid x\!:\!\tau \mid x\!:\!(\vec{\tau_1})^{!} \to y\!:\!(\vec{\tau_2})^{?} \mid A, B$$

The idea behind this definition is that action types are graphs where nodes are of the form $x : \tau$, provided names like x occur at most once. We write $|A|$ for the set of A's nodes. Edges, which are always from input-moded nodes to output-moded nodes, denote dependency between channels and are used to prevent vicious cycles between names. If A is such a graph and $x : \tau$ is one of its nodes, we also write $A(x) = \tau$. By $\mathsf{fn}(A)$ we denote the set of all names x such that $A(x) = \tau$ for some τ. Sometimes we also write $x : \tau \in A$ to indicate that $A(x) = \tau$. We write $\mathsf{md}(A) = p$ to indicate that $\mathsf{md}(A(x)) = p$ for all $x \in \mathsf{fn}(A)$. We write $x \to y$ if $x\!:\!\tau \to y\!:\!\tau'$ for some τ and τ', in a given action type. We compose two processes typed by A and B iff: (1) $A(a) \odot B(a)$ is defined for all $a \in \mathsf{fn}(A) \cap \mathsf{fn}(B)$; and (2) the composition creates no circularity between names. For example, the following two compositions satisfy these two clauses.

$$\begin{aligned}(x\!:\!\tau_1 \to y\!:\!\tau_2) \odot (y\!:\!\overline{\tau_2} \to z\!:\!\tau_3) \;&=\; (x\!:\!\tau_1 \to z\!:\!\tau_3), (y\!:\!\overline{\tau_2} \to z\!:\!\tau_3); \quad \text{and}\\ (x\!:\!\tau_1 \to y\!:\!\tau_2) \odot x\!:\!\overline{\tau_1} \odot x\!:\!\overline{\tau_1} \;&=\; x\!:\!\tau_1 \to y\!:\!\tau_2\end{aligned}$$

However the composition of $x\!:\!\tau_1 \to y\!:\!\tau_2$ and $y\!:\!\overline{\tau_2} \to x\!:\!\overline{\tau_1}$ is undefined as it induces circularity between names x and y. Formally, let us write $A \asymp B$ iff:

– whenever $x\!:\!\tau \in A$ and $x\!:\!\tau' \in B$, $\tau \odot \tau'$ is defined; and

– whenever $x_1 \to x_2$, $x_2 \to x_3$, $\ldots$, $x_{n-1} \to x_n$ alternately in A and B $(n \geq 2)$, we have $x_1 \neq x_n$.

Then $A \odot B$, defined iff $A \asymp B$, is the following action type.

- $x\!:\!\tau \in |A \odot B|$ iff either (1) $x\!:\!\tau$ occurs in either A or B, but not both ; or (2) $x\!:\!\tau' \in A$ and $x\!:\!\tau'' \in B$ and $\tau = \tau' \odot \tau''$.
- $x \to y$ in $A \odot B$ iff $x : \tau, y : \tau' \in |A \odot B|$ and $x = z_1 \to z_2$, $z_2 \to z_3, \ldots, z_{n-1} \to z_n = y$ $(n \geq 2)$ alternately in A and B.

We can easily check that $\odot$ is a symmetric and associative partial operation on action types with unit $\emptyset$. The restriction to A/B-alternation is because paths in an action type cannot have length exceeding 1.

Using this partial operation, P_3 in (8) becomes untypable since $!b.\overline{a}$ has a type $b : ()^{!} \to a : ()^{?}$, which is uncomposable with the type of $!a.\overline{b}$, $a : ()^{!} \to b : ()^{?}$.

Finally the third condition discussed at the outset of §2.2, the restriction to a single thread, is guaranteed by using *IO-modes*, $\phi \in \{\mathtt{I}, \mathtt{O}\}$, in the typing judgement. These IO-modes are given the following partial algebra, using the overloaded notation $\odot$:

$$\mathtt{I} \odot \mathtt{I} = \mathtt{I} \quad \text{and} \quad \mathtt{I} \odot \mathtt{O} = \mathtt{O} \odot \mathtt{I} = \mathtt{O}.$$

Among the two IO-modes, $\mathtt{O}$ indicates a unique active output: thus $\mathtt{O} \odot \mathtt{O}$ is undefined, which means that we do not want more than one active thread at the same time. We write $\phi_1 \asymp \phi_2$ if $\phi_1 \odot \phi_2$ is defined. IO-modes sequentialise the computation in our typed calculus. This makes reductions deterministic which in turn simplifies reasoning. But other than for simplicity, this restriction is not needed. Using this partial algebra, the type discipline which we formally introduce below says the process P_4 in (9) is untypable since each $\overline{a}$ should have mode $\mathtt{O}$, and we know $\mathtt{O} \not\asymp \mathtt{O}$. Similarly P_5 is untypable because its body, $\overline{b} \mid \overline{c}$, is untypable.

2.4. Types and Typing (3): Typing

The typing judgement takes the following form:

$$\vdash_{\phi} P \rhd A$$

which is read: *P has type A with mode ϕ*. We present the typing system in Figure 2. The rules are obtained just by restricting the typing system in [39] to the replicated fragment of the syntax we are now using. The resulting typed calculus is called π^{c}. Appendix C briefly discusses the key differences between π^{c} and the typed calculus of [39]. In the following, we briefly illustrate each typing rule.

- In (Zero), we start in $\mathtt{I}$-mode with empty type since there is no active output.

$$\textsf{(Zero)}\quad \frac{-}{\vdash_{\mathtt{I}} 0 \triangleright \emptyset} \qquad\qquad \textsf{(Par)}\quad \frac{\vdash_{\phi_i} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2 \quad \phi_1 \asymp \phi_2}{\vdash_{\phi_1 \odot \phi_2} P_1|P_2 \triangleright A_1 \odot A_2}$$

$$\textsf{(Res)}\quad \frac{\vdash_{\phi} P \triangleright A \qquad \mathsf{md}(A(x)) = \mathbf{!}}{\vdash_{\phi} (\boldsymbol{\nu}\, x)P \triangleright A/x} \qquad \textsf{(Weak)}\quad x \notin \mathsf{fn}(A) \quad \frac{\vdash_{\phi} P \triangleright A \qquad \mathsf{md}(\tau) = \mathbf{?}}{\vdash_{\phi} P \triangleright A,\ x:\tau} \qquad \textsf{(Weak-}io\textsf{)}\quad \frac{\vdash_{\mathtt{I}} P \triangleright A}{\vdash_{\mathtt{O}} P \triangleright A}$$

$$\textsf{(In}^{\mathbf{!}}\textsf{)}\quad x \notin \mathsf{fn}(A),\ \mathsf{md}(A) = \mathbf{?} \quad \frac{\vdash_{\mathtt{O}} P \triangleright \vec{y}:\vec{\tau},\ A}{\vdash_{\mathtt{I}} !x(\vec{y}).P \triangleright x:(\vec{\tau})^{\mathbf{!}} \rightarrow A} \qquad \textsf{(Out}^{\mathbf{?}}\textsf{)}\quad y_i:\tau_i \in A \quad \frac{\vdash_{\mathtt{I}} P \triangleright A \asymp x:(\vec{\tau})^{\mathbf{?}}}{\vdash_{\mathtt{O}} \overline{x}(\vec{y})P \triangleright A/\vec{y} \odot x:(\vec{\tau})^{\mathbf{?}}}$$

Figure 2. Typing for the π^{C}-Calculus.

- In (Par), "$\asymp$" controls composability, ensuring that at most one thread is active in a given term (by $\phi_1 \asymp \phi_2$) and uniqueness of replicated inputs and non-circularity (by $A_1 \asymp A_2$). The resulting type is given by merging two types.

  For example, if P has type $x : ()^{\mathbf{?}}$ with mode $\mathtt{O}$ and Q has type $x : ()^{\mathbf{!}}$ with mode $\mathtt{I}$, then $P \mid Q$ is typable with type $()^{\mathbf{?}} \odot ()^{\mathbf{!}} = ()^{\mathbf{!}}$ and $\mathtt{O} \odot \mathtt{I} = \mathtt{O}$. The restriction to composable types also guarantees that there's at most one server at each $\mathbf{!}$-moded type. For example $!x().P|!x().Q$ cannot be typable.

- In (Res), we do not allow $\mathbf{?}$ to be restricted since this action expects its dual server always exists in the environment. Hence $(\boldsymbol{\nu}\, x)\overline{x}$ is untypable. $A/\vec{y}$ means the result of deleting the nodes $\vec{x} : \vec{\tau}$ in A (and edges from/to deleted nodes). For example, $(\boldsymbol{\nu}\, x)!x.0$ is always typable with action type $x:\tau/x = \emptyset$.

- In (Weak), we weaken a $\mathbf{?}$-moded channel since this mode means zero or more output actions at a given channel. In (Weak-*io*), we turn the input mode into the output mode. This is intuitively because an output mode means there is at most one active thread, which includes the case when there is no thread. More technically, we need this weakening for subject reduction: even if we start from an active thread, a process can reach a configuration without any active thread, after a series of reductions.

– $(\mathsf{In}^{!})$ ensures non-circularity at x (by $x \notin \mathsf{fn}(A)$) and no free input occurrence under input (by $\mathsf{md}(A) = \mathbf{?}$). Then it records the causality from input to free outputs. If A is empty, $x : (\vec{\tau})^{!} \to A$ simply stands for $x : (\vec{\tau})^{!}$.

– $(\mathsf{Out}^{?})$ essentially the rule composes the output prefix and the body in parallel. In the condition, $y_i : \tau_i \in A$ means each $y_i : \tau_i$ appears in A. This ensures bound input channels $\vec{y}$ become always active after the message received. It also changes the mode to output, to indicate an active thread or server. Note that this rule does not suppress the body by prefix since output is asynchronous. We can also now see why we don't need a rule like that mentioned in Footnote 1 on Page 3 to deal with processes like $\overline{x}(\vec{y})(P|!x(\vec{v}).Q)$. Because if $\{\vec{y}\} \cap \mathsf{fn}(!x(\vec{v}).Q) = \emptyset$, we can simply use the structural rule $(\mathsf{S9})$

$$\overline{x}(\vec{y})(P|!x(\vec{v}).Q) \equiv \overline{x}(\vec{y})P|\ !x(\vec{v}).Q$$

and use $(\mathsf{Com}^{!})$. This covers all the cases because by typing $a \in \{\vec{y}\} \cap \mathsf{fn}(!x(\vec{v}).Q)$ is impossible, as any such a would have to be input moded (so it can be carried by x) and output moded (to be suppressed freely under $!x$) at the same time.

EXAMPLE 2.2. (copy-cat) As an example of the type inference, we take a copy-cat agent. Let $[x \to y] \stackrel{\text{def}}{=} !x(c).\overline{y}(c')!c'.\overline{c}$. This is a *copy-cat agent*, linking two locations x and y: when asked at x, it asks back at y. Then, on receiving the answer c' from y, forwards it back as an answer c to the initial question at x. Having this agent between two locations does not change the whole behaviour. For example:

$$\begin{array}{ll} & !y(z).\overline{z} \mid [x \to y] \mid \overline{x}(e)!e.\overline{e}' \\ \longrightarrow & !y(z).\overline{z} \mid [x \to y] \mid \overline{y}(c)(\boldsymbol{\nu}\, e)(!c.\overline{e} \mid !e.\overline{e}') \\ \longrightarrow & !y(z).\overline{z} \mid [x \to y] \mid (\boldsymbol{\nu}\, ce)(\overline{c} \mid !c.\overline{e} \mid !e.\overline{e}') \\ \longrightarrow^2 \sim & !y(z).\overline{z} \mid [x \to y] \mid \overline{e}' \end{array}$$

where $\sim$ is the standard strong bisimulation (note $(\boldsymbol{\nu}\, y)!y(\vec{z}).P \sim 0$). The above agent is the same as $!y(z).\overline{z}|\overline{y}(e)!e.\overline{e}' \longrightarrow\longrightarrow\sim\ !y(z).\overline{z}|\overline{e}'$, except for some internal reductions. Below we show a step by step derivation for the typing of $[x \to y]$ with $\tau \stackrel{\text{def}}{=} (()^{?})^{!}$.

1: $\vdash_{\mathtt{I}}\ 0 \triangleright \emptyset$
2: $\vdash_{\mathtt{O}}\ \overline{c} \triangleright c\colon ()^{?}$
3: $\vdash_{\mathtt{I}}\ !c'.\overline{c} \triangleright c'\colon ()^{!} \to c\colon ()^{?}$
4: $\vdash_{\mathtt{O}}\ \overline{y}(c')!c'.\overline{c} \triangleright y\colon\overline{\tau}, c\colon ()^{?}$ (by $(c'\colon ()^{!} \to c\colon ()^{?})/c' = c\colon ()^{?}$)
5: $\vdash_{\mathtt{I}}\ !x(c).\overline{y}(c')!c'.\overline{c} \triangleright x\colon\tau \to y\colon\overline{\tau}$ (by $(y\colon\overline{\tau}, c\colon ()^{?})/c = y\colon\overline{\tau}$)

In this derivation, the length of paths in action types does not exceed 1 even when the term gets bigger and bigger in size.

In the same way, the reader can also check P_2 in (7) and P_6 in (10) are typed as follows.

$$\vdash_{\mathtt{O}} P_2 \triangleright b\colon ()^{!} \to a\colon ()^{?}, c\colon ()^{!} \to a\colon ()^{?}$$
$$\vdash_{\mathtt{O}} P_6 \triangleright a\colon ()^{!} \to c\colon ()^{?}, b\colon ()^{!} \to c\colon ()^{?}, e\colon ()^{!} \to c\colon ()^{?}$$

2.5. Properties of π^{c}

The following substitution lemma, whose second clause shows ?-names can be coalesced together, is a basic observation needed to prove the correctness of the encoding.

LEMMA 2.3. (substitution lemma)

1. *If* $\vdash_{\phi} P \triangleright A$ *and* $y \notin \mathsf{fn}(A)$, *then* $\vdash_{\phi} P\{y/x\} \triangleright A\{y/x\}$.
2. *If* $\vdash_{\phi} P \triangleright A$ *such that* $A(x) = A(y)$ *and, moreover,* $\mathsf{md}(A(x)) = ?$, *then* $\vdash_{\phi} P\{z/xy\} \triangleright A\{z/xy\}$ *for fresh* z.

PROOF: By an easy induction on the rules in Figure 1. ■

The subject reduction of π^{c} is an immediate consequence of that in [39], since both the action types and the reduction of the present calculus are projection of those of the sequential linear π-calculus in [39, §5.3].

PROPOSITION 2.4. (Subject Reduction) *If* $\vdash_{\phi} P \triangleright A$ *and* $P \longrightarrow Q$ *then* $\vdash_{\phi} Q \triangleright A$.

In addition to the standard reduction, we define an extended notion of reduction, called *the extended reduction*, written $\searrow$, again precisely following [39]. We shall use this reduction extensively in the present study. While $\longrightarrow$ gives a natural notion of dynamics which makes sense in both sequential and concurrent computation, $\searrow$ extends $\longrightarrow$ by exploiting the stateless nature of π^{c}-processes. It offers a close correspondence with the reduction in the $\lambda\mu$v-calculus through the encoding.

For that reason $\searrow$ is useful for studying the correspondence between two calculi. Formally $\searrow$ is the least compatible relation, i.e. closed under typed context, over typed processes, taken modulo $\equiv$, that includes:

$$\begin{array}{rcl} C[\overline{x}(\vec{y})P] | !x(\vec{y}).Q & \searrow_r & C[(\boldsymbol{\nu}\,\vec{y})(P|Q)] \,|\, !x(\vec{y}).Q \\ (\boldsymbol{\nu}\,x)!x(\vec{y}).Q & \searrow_g & \mathbf{0} \end{array}$$

where $C[\,\cdot\,]$ is an arbitrary (typed) context. We can immediately see $\longrightarrow\subset\searrow$. Note $\searrow$ calculates *under* prefixes, which is unusual in process calculi. For example, we have

$$P_2 \;\longrightarrow\; !\,b.\overline{a} \,|\, \overline{a} \,|\, !\,c.\overline{b} \;\searrow\; !\,b.\overline{a} \,|\, \overline{a} \,|\, !\,c.\overline{a}$$

Another observation is that a given typed process in the π^{c}-calculus can have at most one redex for the standard reduction $\longrightarrow$ while it may have more than one redex for $\searrow$.

The extended reduction $\searrow$ is the exact image of extended reduction in [39] onto the present subcalculus, so that we immediately conclude, from the results in [39]:

PROPOSITION 2.5.

1. (Subject Reduction) *If* $\vdash_\phi P \triangleright A$ *and* $P \searrow Q$ *then* $\vdash_\phi Q \triangleright A$.
2. (CR) *If* P *is typable and* $P \searrow Q_i$ *(*$i = 1, 2$*) with* $Q_1 \not\equiv Q_2$, *we have* $Q_i \searrow^+ R$ *(*$i = 1, 2$*) for some* R.
3. (SN) *If* P *is typable then* P *does not have infinite* $\searrow$*-reductions.*

PROOF: By Proposition 5.5 in [39]. ∎

It may be useful to state at this point that, possibly contrary to what is suggested by the asymmetric notation, $\searrow$ does *not* introduce a new form of computation step, a new form of interaction. Instead, $P \searrow Q$ says that P and Q cannot be distinguished by *well-typed* observers. This indistinguishability is an artifact of our restrictive typing discipline and does not hold in the untyped calculus. The asymmetric notation was chosen to emphasise that Q in $P \searrow Q$ is smaller, in the sense of having fewer process constructors, than P. There are three further observations on the extended reduction. First, while we do not use the property directly in the present work, the convertibility induced by $\searrow$ (i.e. the typed congruent closure of $\searrow$) coincides with the weak bisimilarity $\approx$ [39, Theorem 4.1], because the transition relation is the faithful image of that of the pure sequential linear π-calculus in [39].

Second, Proposition 2.5 (3) indicates all π^{c}-processes are represented by their $\searrow$-normal forms, i.e. those π^{c}-processes which do not have a $\searrow$-redex, which own a very simple syntactic structure characterised inductively.

Finally, in the definition of $\searrow$ it is not necessary to cater for replicated inputs occurring freely under other input prefixes as that is impossible by typing. Similarly, any replicated input with free subject under an output can be put into parallel with that output by the structural rules in the typed setting.

DEFINITION 2.6. *Let the set* NF_e *of* π^{c}*-processes be generated by the following induction, assuming typability in each clause.*

1. $\mathbf{0} \in \mathsf{NF}_e$
2. *if* $P, Q \in \mathsf{NF}_e$ *and* P *and* Q *do not share a common free name of different polarities, then* $P|Q \in \mathsf{NF}_e$
3. $P \in \mathsf{NF}_e$ *then* $!x(\vec{y}).P \in \mathsf{NF}_e$
4. $\overline{x}(\vec{y})P \in \mathsf{NF}_e$ *if* $P \in \mathsf{NF}_e$ *and* $\overline{x}(\vec{y})P$ *is a prime output, where we call* $\overline{x}(\vec{y})P$ prime *if the initial* x *is its only free name not under input prefix.*
5. *If* $P \in \mathsf{NF}_e$ *and* $P \equiv Q$ *then* $Q \in \mathsf{NF}_e$.

PROPOSITION 2.7. *Let* P *be typable and* $P \not\searrow$. *Then* $P \in \mathsf{NF}_e$.

PROOF: By Proposition 3.3 in [39]. ■

2.6. Contextual Congruence for π^{c}

The Church-Rosser property of typed processes, as stated in Proposition 2.5, suggests that non-deterministic state change (which plays a basic role in e.g. bisimilarity and testing/failure equivalence) may safely be ignored in typed equality, so that a Morris-like contextual equivalence suffices as a basic equality over processes. Let us define:

$$P \Downarrow_x \text{ iff } P \twoheadrightarrow \overline{x}(\vec{y})Q \text{ for some } Q$$

Here $\twoheadrightarrow$ is the transitive and reflexive closure of $\longrightarrow$. We can now define a basic typed congruence. Below, a relation over typed processes is *typed* if it relates only processes with identical action type and IO-mode. If $\mathcal{R}$ is a typed relation and $\vdash_\phi P_{1,2} \triangleright A$ are related by $\mathcal{R}$ then we write $\vdash_\phi P_1 \mathcal{R} P_2 \triangleright A$ or, when no confusion arises, $P_1 \mathcal{R} P_2$. A relation is a *typed congruence* when (1) $\mathcal{R} \supseteq \equiv$, and (2) $\mathcal{R}$ is a typed equivalence relation closed under typed contexts (note we are taking $\equiv$ as if it were the α-equality: this is essentially because the notion of reduction depends on this relation, just as reduction in the λ-calculus depends on the α-equality).

DEFINITION 2.8. $\cong_\pi$ is the maximum typed congruence satisfying: if $\vdash_{\mathtt{0}} P \cong_\pi Q \triangleright x\!:\!()^{?}$, then $P \Downarrow_x$ iff $Q \Downarrow_x$.

Below a typed congruence is *maximally consistent* (cf. [3, 15]) if adding any additional equation to it leads to inconsistency, i.e. equations on all processes with identical typing.

PROPOSITION 2.9.

1. $\searrow \subset \cong_\pi$.
2. $\cong_\pi$ *is a maximally consistent typed congruence.*
3. $\cong_\pi$ *is the unique maximally consistent congruence containing* $\searrow$.

PROOF: See Appendix A.1. ■

REMARK 2.10. (observables in π^{c}) This choice, which also corresponds to the usual output-barbed congruence one considers in the untyped calculus, is *the* canonical choice for the calculus fragment under discussion, for the following reasons. **?**-actions are not considered as observables in linear/affine π-calculi [4, 39] since, intuitively, invoking replicated processes do not affect them. Proposition 2.9 suggests that the existence/non-existence of **?**-actions may be the only sensible way to obtain a non-trivial large equality in π^{c}, equationally justifying the use of **?**-actions as observables. The reader familiar with observational congruences in languages/calculi with full control may observe that this choice does conform to the fact that, in these languages/calculi, observational congruences do care about, albeit *a posteriori*, calls/jumps to procedures/names in the environment, cf. Section 5.

We list a basic characterisation of $\cong_\pi$. Below $|A|$ stands for the map from names to channel types underlying A; $\overline{(\ \cdot\ )}$ dualises such a map.

PROPOSITION 2.11. (context lemma) $\vdash_{\mathtt{0}} P_1 \cong_\pi P_2 \triangleright A$ *if and only if the following condition holds: for each* $\vdash_{\mathtt{I}} R \triangleright B$ *such that* $|B| = \overline{|A|} \cup \{x : ()^{?}\}$ *for some fresh* x*, we have* $P_1|R \Downarrow_x$ *iff* $P_2|R \Downarrow_x$.

PROOF: Standard (cf. [29]), listed in Appendix A.2 for reference. ■

REMARK 2.12. (context lemma) The context lemma such as Proposition 2.11 is a major tool for reasoning about process equivalences in many known typed and untyped π-calculi. It is also a natural counterpart of the standard context lemma for functions, where the main composition operator for functions, the application, is generalised to the main composition operator for communicating processes, concurrent composition. From a different viewpoint, the characterisation may be considered as a healthiness condition for contextual congruences since we may as well demand the sole source of distinguishability among processes should come from how they interact with other processes and, as a result, contributes to a basic notion of observables.

3. Encoding

3.1. Call-by-value $\lambda\mu$-calculus

In this section we present a type-preserving embedding of the call-by-value $\lambda\mu$-calculus by Ong and Stewart [27] in π^{c}. Apart from tractable syntactic properties of the calculus in comparison with its call-by-name counterpart, Ong and Stewart showed how various control primitives of call-by-value languages (such as call-cc in ML) can be encoded cleanly in this calculus and its extension with recursion [27]. The calculus represents full control in a call-by-value setting, just like the call-by-value λ-calculus with Felleisen's $\mathcal{C}$ operator.

We have decided to embed the full call-by-value $\lambda\mu$-calculus, rather than some simpler calculi, for example the $\lambda\mu$-calculus without $\perp$, for the following reasons.

- The use of $\perp$ is a good "stress test" to the embedding and the typed π-calculus (for example we want no closed processes to inhabit the $\perp$-type).
- Deriving $\perp$ is central to classical proofs (contraposition): the present paper shows that an appropriate embedding does offer a clean understanding of $\perp$-types (as lack of information).
- Subcalculi of $\lambda\mu$-calculus, such as the $\perp$-free fragment, and extension with constants and fixed-points can be embedded straightforwardly starting with the $\lambda\mu$-calculus.

Types $(\alpha, \beta, \ldots)$ are those of simply typed λ-calculus with the atomic type $\perp$ (we can add other atomic types with appropriate values and operations on them). We use *variables* $(x, y, \ldots)$ and *control variables*

(Id)
$$\frac{-}{\Gamma\cdot x\!:\!\alpha\vdash x\!:\!\alpha\,;\Delta}$$

(C-var)
$$\frac{\Gamma\cdot x\!:\!\alpha\cdot y\!:\!\alpha\vdash M\!:\!\beta\,;\Delta}{\Gamma\cdot z\!:\!\alpha\vdash M\{z/xy\}\!:\!\beta\,;\Delta}$$

(C-name)
$$\frac{\Gamma\vdash M\!:\!\beta\,;\Delta\cdot a\!:\!\alpha\cdot b\!:\!\alpha}{\Gamma\vdash M\{c/ab\}\!:\!\beta\,;\Delta\cdot c\!:\!\alpha}$$

(⇒-I)
$$\frac{\Gamma\cdot x\!:\!\alpha\vdash M\!:\!\beta\,;\Delta}{\Gamma\vdash\lambda x^{\alpha}.M\!:\!\alpha\Rightarrow\beta\,;\Delta}$$

(⇒-E)
$$\frac{\Gamma\vdash M\!:\!\alpha\Rightarrow\beta\,;\Delta\qquad\Gamma\vdash N\!:\!\alpha\,;\Delta}{\Gamma\vdash MN\!:\!\beta\,;\Delta}$$

(⊥-I)
$$\frac{\Gamma\vdash M\!:\!\alpha\,;\Delta\quad\alpha\neq\bot}{\Gamma\vdash[a]M\!:\!\bot\,;\Delta\cdot a\!:\!\alpha}$$

(⊥-E)
$$\frac{\Gamma\vdash M\!:\!\bot\,;\Delta\cdot a\!:\!\alpha}{\Gamma\vdash\mu a^{\alpha}.M\,:\,\alpha\,;\Delta}$$

Figure 3. Typing rules for the $\lambda\mu$-calculus.

(or *names*) $(a,b,\ldots)$. *Preterms* $(M,N,\ldots)$ and *values* $(V,W,\ldots)$ are generated from the grammar:

$$\begin{array}{rcl} M,N & ::= & x \mid \lambda x^{\alpha}.N \mid MN \mid \mu a^{\alpha}.M \mid [a]M \\ V,W & ::= & x \mid \lambda x^{\alpha}.N \end{array}$$

Apart from the standard variables, abstraction and application, we have a *named term* $[a]M$ and a *μ-abstraction* $\mu a.M$, both of which use names. The typing judgement has the form:

$$\Gamma\vdash M:\alpha;\Delta$$

where Γ is a finite map from variables to types, M is a preterm given above, and Δ is a finite map from names to non-$\bot$-types.

The typing rules are given in Figure 3. In the rules, we assume newly introduced names/variables in the conclusion are always fresh. The notation $\Gamma\cdot x\!:\!\tau$ indicates x is not in the domain of Γ. $M\{z/xy\}$ denotes the result of substituting z in M for both x and y, similarly for $M\{c/ab\}$. A typable preterm is called a *$\lambda\mu v$-term.*

We let $\beta \neq \bot$. In the last four rules, $M\{\, C[\,\cdot\,] \,/\, [a][\,\cdot\,] \,\}$ is the result of substituting $C[L_i]$ for each subterm of the shape $[a]L_i$ occurring in M assuming the bound name convention, see Appendix B.

$$\begin{array}{rl}
(\beta_v) & (\lambda x.M)V \longrightarrow M\{V/x\} \\
(\eta_v) & \lambda x.(Vx) \longrightarrow V \quad (\text{if } x \notin \mathsf{fv}(V)) \\
(\mu\text{-}\beta) & [b]\mu a.M \longrightarrow M\{b/a\} \\
(\mu\text{-}\eta) & \mu a.[a]M \longrightarrow M \quad (\text{if } a \notin \mathsf{fn}(M)) \\
(\zeta_{\text{fun}}) & (\mu a^{\alpha \Rightarrow \beta}.M)N \longrightarrow \mu b.M\{\, [b]([\,\cdot\,]N) \,/\, [a][\,\cdot\,] \,\} \\
(\zeta_{\text{fun},\bot}) & (\mu a^{\alpha \Rightarrow \bot}.M)N \longrightarrow M\{\, [\,\cdot\,]N \,/\, [a][\,\cdot\,] \,\} \\
(\zeta_{\text{arg}}) & V^{\alpha \Rightarrow \beta}(\mu a^{\alpha}.M) \longrightarrow \mu b.(M\{\, [b](V[\,\cdot\,]) \,/\, [a][\,\cdot\,] \,\} \quad \alpha \neq \bot \\
(\zeta_{\text{arg},\bot}) & V^{\alpha \Rightarrow \bot}(\mu a^{\alpha}.M) \longrightarrow M\{\, V[\,\cdot\,] \,/\, [a][\,\cdot\,] \,\} \quad \alpha \neq \bot \\
(\bot) & V^{\bot \Rightarrow \beta}M \longrightarrow \mu b^{\beta}.M \quad (b \text{ fresh}) \\
(\bot_\bot) & V^{\bot \Rightarrow \bot}M \longrightarrow M
\end{array}$$

Figure 4. Reduction rules for the $\lambda\mu$v-calculus. We have omitted the rules ensuring compatibility of $\longrightarrow$.

The reduction rules for the $\lambda\mu$v-calculus is given in Figure 4. In the rules we include η_v-reduction, unlike [27]. Inclusion or non-inclusion does not affect the subsequent technical development.

REMARK 3.1. The sequent for the $\lambda\mu$v-calculus in [27] has the form $\Gamma; \Delta \vdash M : \alpha$, which is natural from a logical viewpoint (one way to read this sequent is to regard Γ as positively assumed formulae and Δ as negatively assumed formulae, the latter to be discharged by the contraposition rule, cf. [28, 35]). We choose the present notation because it is close to its process representation, as we shall see soon.

3.2. Encoding (1): Types

The general idea of the encoding is simple, and closely follows the standard call-by-value encoding of the λ-calculus, due to Milner [24]. The reading is strongly operational, elucidating the dynamics of $\lambda\mu$-terms up to a certain level of abstraction. In brief, given a $\lambda\mu$-term:

$$\Gamma \vdash M : \alpha; \Delta, \tag{11}$$

its encoding considers Γ as the interaction points of the program/process where it queries the environment and gets information; while either at its main port, typed as α, or at one of the control variables given as Δ, the program/process would return a value: at which port it would return depends on how its sequential thread of control will proceed during execution. If Δ is empty, this reading precisely coincides with Milner's original one [24]. One of the distinguishing features of the π-calculus encodings of programming languages in general (including those for untyped calculi) and that of the present encoding in particular, is that the operational interpretation of this sort in fact obeys a clean and rigid type structure.

We start with the encoding of types, which we present using two maps, $\alpha^\bullet$ and α°. Intuitively α° maps α as a type for values; while $\alpha^\bullet$ maps α as a type for threads which may converge to values of type α or which may diverge, or "computation" in Moggi's terminology [26].

$$\alpha^\bullet \stackrel{\text{def}}{=} \begin{cases} \epsilon & (\alpha = \bot) \\ (\alpha^\circ)^? & (\alpha \neq \bot) \end{cases} \qquad (\alpha \Rightarrow \beta)^\circ \stackrel{\text{def}}{=} \begin{cases} (\beta^\bullet)^! & (\alpha = \bot) \\ (\overline{\alpha^\circ}\beta^\bullet)^! & (\alpha \neq \bot) \end{cases}$$

Note a type for computation is the lifting of a type for values. The encoding of $\bot$ indicates that we assume there is no (closed) value, or a proof without assumptions, inhabiting $\bot$. This leads to the degenerate treatment of $(\bot \Rightarrow \alpha)^\bullet$ since "asking at the assumed absurdity" does not make sense. By "degenerate" we mean that the argument in $(\bot \Rightarrow \alpha)$ is simply ignored.

EXAMPLE 3.2. As simple examples:

$$\begin{array}{rcl} (\bot \Rightarrow \bot)^\circ & \stackrel{\text{def}}{=} & ()^! \\ ((\bot \Rightarrow \bot) \Rightarrow \bot)^\circ & \stackrel{\text{def}}{=} & (()^?)^! \end{array}$$

Note if $\alpha \neq \bot$ we always have $(\alpha \Rightarrow \bot)^\circ = (\overline{\alpha^\circ})^!$ which corresponds to the standard translation, $\neg A \stackrel{\text{def}}{=} A \supset \bot$. As further examples:

$$\begin{array}{rcl} (\bot \Rightarrow (\bot \Rightarrow \bot))^\circ & \stackrel{\text{def}}{=} & ((()^!)^?)^! \\ ((\bot \Rightarrow \bot) \Rightarrow (\bot \Rightarrow \bot))^\circ & \stackrel{\text{def}}{=} & (()^?(()^!)^?)^! \end{array}$$

Following the mappings of types, the environments for variables and names are mapped as follows, starting from $\emptyset^\bullet \stackrel{\text{def}}{=} \emptyset$ and $\emptyset^\circ \stackrel{\text{def}}{=} \emptyset$.

$$(a\!:\!\alpha\cdot\Delta)^\bullet \stackrel{\text{def}}{=} \begin{cases} a\!:\!\alpha^\bullet \cdot \Delta^\bullet & (\alpha \neq \bot) \\ \Delta^\bullet & (\alpha = \bot) \end{cases} \quad (x\!:\!\alpha\cdot\Gamma)^\circ \stackrel{\text{def}}{=} \begin{cases} x\!:\!\overline{\alpha^\circ} \cdot \Gamma^\circ & (\alpha \neq \bot) \\ \Gamma^\circ & (\alpha = \bot) \end{cases}$$

The special treatment of $\bot$ follows the encoding of types above and reflects its special role in classical natural deduction. Simply put, if we have a proof whose conclusion is the falsity $\bot$, then it is given there, for its all usefulness, for the purpose of having a contradiction and negating a stipulated assumption. Operationally this suggests the proof whose (conclusion's) type is $\bot$ has nothing positive to communicate to the outside, which explains why the map for computation $(\ \cdot\ )^\bullet$ ignores the control channel of type $\bot$. Dually you get no information from the proof of type $\bot$, so querying at that environment port is insignificant, hence we ignore $\bot$-types in the negative positions.

3.3. Encoding (2): Terms

For the encoding of terms, we introduce the following notations, which we shall use throughout the paper. Below in (3) we use the notation from [12, Remark 15] in the context of CPS calculus (cf. Section 5).

NOTATION 3.3.

1. (copycat, cf. Ex. 2.2) Let τ be an input type. Then $[x \to y]^\tau$, *copy-cat of type* τ, is inductively defined by the following clause.

$$[x \to x']^{(\tau_1..\tau_n)^!} \stackrel{\text{def}}{=} !x(\vec{y}).\overline{x'}(\vec{y}')\Pi_{1\le i\le n}[y'_i \to y_i]^{\overline{\tau_i}}.$$

   where $\prod_{1\le i\le n} P_i$, which we often write $\prod_i P_i$, stands for the n-fold parallel composition $P_1|\cdots|P_n$.

2. (free output) We let $\overline{x}\langle\vec{y}^{\vec{\tau}}\rangle \stackrel{\text{def}}{=} \overline{x}(\vec{z})\Pi[z_i \to y_i]^{\overline{\tau_i}}$ with each τ_i having an output mode.

3. (substitution environment) $P\{x(\vec{y})\!=\!R\} \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)(P \mid !x(\vec{y}).R)$.

Figure 5 presents the encoding of terms. The encoding closely follows that of types, mapping a typing judgement $\Gamma \vdash M : \alpha\,;\Delta$ and a fresh name (called *anchor*) to a process. For brevity, we omit the type environment from the source term in Figure 5 (it suffices to assume, w.l.o.g., that all the free variables in a term are annotated with types). In each rule, we assume newly introduced names (among others an anchor) are always fresh.

The anchor u in $[\![M:\alpha]\!]_u$ represents the point of interaction which M may have as a process [24] or, more concretely, the channel through which the process returns the resulting value to the environment. The process $[\![M:\alpha]\!]_u$ may also have interactions at its free variables (for

$$[\![x:\alpha]\!]_u \stackrel{\text{def}}{=} \begin{cases} \overline{u}\langle x^{\overline{\alpha^\circ}}\rangle & (\alpha \neq \bot) \\ \mathbf{0} & (\alpha = \bot) \end{cases}$$

$$[\![\lambda x^\alpha.M:\alpha\Rightarrow\beta]\!]_u \stackrel{\text{def}}{=} \begin{cases} \overline{u}(c)!c(xz).[\![M:\beta]\!]_z & (\alpha \neq \bot, \beta \neq \bot) \\ \overline{u}(c)!c(z).[\![M:\beta]\!]_z & (\alpha = \bot, \beta \neq \bot) \\ \overline{u}(c)!c(x).[\![M:\bot]\!]_z & (\alpha \neq \bot, \beta = \bot) \\ \overline{u}(c)!c.[\![M:\bot]\!]_z & (\alpha = \bot, \beta = \bot) \end{cases}$$

$$[\![MN:\beta]\!]_u$$
$$\stackrel{\text{def}}{=} \begin{cases} [\![M:\alpha\Rightarrow\beta]\!]_m\{m(c)=([\![N:\alpha]\!]_n\{n(e)=\overline{c}\langle eu^{\overline{\alpha^\circ}\,\overline{\beta^\circ}}\rangle\})\} & (\alpha\neq\bot,\beta\neq\bot) \\ [\![M:\alpha\Rightarrow\beta]\!]_m\{m(c)=\overline{c}\langle u^{\overline{\beta^\circ}}\rangle\} & (\alpha=\bot,\beta\neq\bot) \\ [\![M:\alpha\Rightarrow\beta]\!]_m\{m(c)=[\![N:\alpha]\!]_u\} & (\alpha=\bot) \end{cases}$$

$$[\![[a]M:\bot]\!]_u \stackrel{\text{def}}{=} [\![M:\alpha]\!]_m\{a/m\}$$

$$[\![\mu a^\alpha.M:\alpha]\!]_u \stackrel{\text{def}}{=} [\![M:\bot]\!]_m\{u/a\}$$

Figure 5. Encoding of $\lambda\mu$-terms.

querying information) and at its free control variables (for returning values). Note both of them are now channel names.

The clauses in Figure 5 follows Milner's standard encoding [24] except for the treatment of:

(1) $\bot$-typed programs;

(2) $\bot$-typed types in abstraction and applications;

(3) Named terms and μ-abstraction.

For (1), we can check the encoding of a $\bot$-typed program never contains an anchor as its free name, so that the resulting process does not communicate at its principal port (anchor), but only interacts at its environments, possibly returning at one of its control variables.

(2) follows the treatment of $\bot$ in types: since $\bot$ is translated into an empty type which is inhabited by no processes except $\mathbf{0}$, there is no communication at $\bot$-typed channels (we can make the encoding more uniform by having an additional indirection: however the present treatment may be faithful to the intuitive understanding of $\bot$ as an empty type, apart from leading to a more terse encoding).

For (3), intuitively we interpret $[a]M$ as a process which jumps to a instead of to its anchor, still carrying the same value. Thus using the substitution makes sense. Note that M may already contain named terms of the form $[a]N_i$ as its subterm: in this case, the substitution leads to the collapse of names. Symmetrically, $\mu a.M$ redirects all jumps towards a to the anchor of the program, again using the substitution. As u is not used in $\llbracket [a]M \rrbracket_u$ we often write $\llbracket [a]M \rrbracket$ instead.

The following result comes from the precise correspondence between derivations of a typed $\lambda\mu$v-term and those of its encoding.

PROPOSITION 3.4. (type-preservation) $\Gamma \vdash M\!:\!\alpha\,;\Delta$ *implies* $\vdash_{\mathsf{O}} \llbracket M\!:\!\alpha \rrbracket_u \triangleright (u\!:\!\alpha \cdot \Delta)^{\bullet},\ \Gamma^{\circ}$.

PROOF: By rule induction of the typing rules in Figure 3. Variables are direct from the following standard results, cf. [39, Proposition 5.4].

1. *With* $\mathsf{md}(\tau) = \mathbf{!}$*, we have* $\vdash_{\mathsf{I}} [x \to y]^{\tau} \triangleright x\!:\!\tau \to y\!:\!\overline{\tau}$.

2. *With* $\mathsf{md}(\tau) = \mathbf{?}$*, we have* $\vdash_{\mathsf{O}} \overline{u}\langle x^{\tau}\rangle \triangleright u\!:\!(\overline{\tau})^{\mathbf{?}}, x\!:\!\tau$.

Each case of the λ-abstraction is direct from the induction hypothesis. For application, we first observe $\vdash_{\mathsf{O}} P \triangleright x\!:\!\overline{\tau}, \Gamma$ and $\vdash_{\mathsf{I}} !x(\vec{y}).R \triangleright x\!:\!\tau \to \Gamma$ implies $\vdash_{\mathsf{O}} P\{x(\vec{y}) = R\} \triangleright \Gamma$. Using this observation repeatedly, each clause of the encoding of the application is direct from the induction hypothesis. For illustration, we give a detailed example of the typing inference for $\llbracket M^{\alpha \Rightarrow \beta} N : \beta \rrbracket$, where $\alpha, \beta \neq \bot$.

$$\begin{array}{l}
\vdash_{\mathsf{O}} \overline{c}\langle eu\rangle \triangleright c : (\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}}, e : \overline{\alpha^{\circ}}, u : \beta^{\bullet} \\ \hline
\vdash_{\mathsf{I}} !n(e).\overline{c}\langle eu\rangle \triangleright n : (\overline{\alpha^{\circ}})^{\mathbf{!}} \to (c : (\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}}, u : \beta^{\bullet}) \\ \hline
\vdash_{\mathsf{O}} \llbracket N : \alpha \rrbracket_n | !n(e).\overline{c}\langle eu\rangle \triangleright n : (\overline{\alpha^{\circ}})^{\mathbf{!}} \to (c : (\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}}, u : \beta^{\bullet}), \Delta^{\bullet}, \Gamma^{\circ} \\ \hline
\vdash_{\mathsf{O}} (\nu n)(\llbracket N \rrbracket_n | !n(e).\overline{c}\langle eu\rangle) \triangleright c : (\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}}, u : \beta^{\bullet}, \Delta^{\bullet}, \Gamma^{\circ} \\ \hline
\vdash_{\mathsf{I}} !m(c).(\nu n)(\llbracket N \rrbracket_n | !n(e).\overline{c}\langle eu\rangle) \triangleright m : ((\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}})^{\mathbf{!}} \to (u : \beta^{\bullet}, \Delta^{\bullet}, \Gamma^{\circ}) \\ \hline
\vdash_{\mathsf{O}} \llbracket M \rrbracket_m | !m(c).(\nu n)(\llbracket N \rrbracket_n | !n(e).\overline{c}\langle eu\rangle) \triangleright m : ((\alpha^{\circ}\overline{\beta^{\bullet}})^{\mathbf{?}})^{\mathbf{!}} \to (u : \beta^{\bullet}, \Delta^{\bullet}, \Gamma^{\circ}) \\ \hline
\vdash_{\mathsf{O}} \llbracket MN \rrbracket_u \triangleright u : \beta^{\bullet}, \Delta^{\bullet}, \Gamma^{\circ}
\end{array}$$

For a named term, suppose $\Gamma \vdash [a]M : \bot \cdot \Delta, a : \alpha$ is inferred from $\Gamma \vdash M : \alpha{\cdot}\Delta$. Note that, in this case, $a \notin \mathsf{fn}(M)$ (later some names in M can be coalesced into a). By induction hypothesis we have $\vdash_{\mathsf{O}} \llbracket M \rrbracket_u \triangleright u : \alpha^{\bullet}, \Gamma^{\circ}, \Delta^{\bullet}$. By Lemma 2.3 (1) and noting we always have $\llbracket [a]M \rrbracket_u \overset{\text{def}}{=} \llbracket M \rrbracket_a$ whenever $a \notin \mathsf{fn}(M)$, we have $\vdash_{\mathsf{O}} \llbracket [a]M \rrbracket_u \triangleright a : \alpha^{\bullet}, \Gamma^{\circ}, \Delta^{\bullet}$. For name abstraction, suppose $\Gamma \vdash \mu a^{\alpha}.M : \alpha, \Delta$ is inferred from $\Gamma \vdash M : \bot, \Delta, a : \alpha$. By Lemma 2.3 (2) and noting $\llbracket \mu a.M \rrbracket_u \overset{\text{def}}{=} \llbracket M \rrbracket\{u/a\}$ we have $\vdash_{\mathsf{O}} \llbracket \mu a^{\alpha}.M \rrbracket_u \triangleright u\!:\!\alpha^{\bullet}, \Gamma^{\circ}, \Delta^{\bullet}$. Finally if a term is typed with (C-var) or (C-name) as the last rule of a type derivation, we can directly apply Lemma 2.3 (2). ∎

REMARK 3.5. In Proposition 3.4, the type of the term and the types of control names are both mapped with $(\ )^\bullet$, conforming to the shape of the sequent $\Gamma \vdash M : \alpha; \Delta$. In particular, there is no causality arrow in the types for translations of $\lambda\mu$-terms. This is because all types (including environments and types for names) are mapped to output types, and causality can only from **!** to **?**.

3.4. Examples of Encodings

A few examples of the encoded $\lambda\mu$v-terms follow, including representative control operators.

EXAMPLE 3.6. (variable) As a simplest example, consider

$$[\![x : \bot]\!]_u \stackrel{\text{def}}{=} \mathbf{0}.$$

Since $(x : \bot)^\circ = (u : \bot)^\bullet = \emptyset$, we have

$$\vdash_{\mathsf{O}} [\![x : \bot]\!]_u \triangleright (u : \bot)^\bullet,\ (x : \bot)^\circ$$

This encoding intuitively represents a trivial proof which assumes $\bot$ and concludes $\bot$, or, in the terminology of Linear Logic, the axiom link of the empty type.

EXAMPLE 3.7. (identity, 1) By closing x of the preceding example, we get:

$$[\![\lambda x^\bot.x : \bot \Rightarrow \bot]\!]_u \stackrel{\text{def}}{=} \overline{u}(c)!c.\mathbf{0}.$$

Since $(\bot \Rightarrow \bot)^\bullet = (()^!)^?$, we have

$$\vdash_{\mathsf{O}} [\![\lambda x^\bot.x : \bot \Rightarrow \bot]\!]_u \triangleright u{:}(\bot \Rightarrow \bot)^\bullet$$

At the end of this subsection, we explore inhabitants of this simple process type.

EXAMPLE 3.8. (identity, 2) If $\alpha \neq \bot$, then

$$[\![\lambda x^\alpha.x : \alpha \Rightarrow \alpha]\!]_u \stackrel{\text{def}}{=} \overline{u}(c)!c(xz).\overline{z}\langle x^{\overline{\alpha^\circ}}\rangle$$

EXAMPLE 3.9. (control operator, 1) The following term essentially corresponds to $\mathcal{C}$ in $\lambda\mathcal{C}v$ introduced by Felleisen and his colleagues [10, 11]. Logically it is a shortest proof of $\neg\neg A \supset A$. Below we let $\neg\alpha \stackrel{\text{def}}{=} \alpha \Rightarrow \bot$.

$$\aleph \stackrel{\text{def}}{=} \lambda z^{\neg\neg\alpha}.\mu a^\alpha.z(\lambda x^\alpha.[a]x)$$

Its direct encoding is, assuming $\alpha \neq \bot$:

$$[\![\aleph]\!]_u \overset{\text{def}}{=} \overline{u}(c)!c(za).(\boldsymbol{\nu}\, m)(\overline{m}\langle z\rangle \mid !m(z).(\boldsymbol{\nu}\, n)(\overline{n}(f)!f(x).\overline{a}\langle x\rangle \mid !n(f).\overline{z}\langle f\rangle))$$

which, through a couple of $\searrow$ uses, can be simplified into:

$$\overline{u}(c)!c(za).\overline{z}(f)!f(x).\overline{a}\langle x\rangle$$

This agent first signals itself: then it is invoked with a function in the environment (of type $\neg\neg\alpha$) as an argument and a continuation a (of type α), invokes the former with the identity agent (whose continuation is a) and a continuation a. Then if that function asks back at the identity with an argument, say x, then this x is returned to a as the answer to the initial invocation. Note how the π^{c}-translation makes explicit the operational content of the agent, especially when simplified using $\searrow$.

EXAMPLE 3.10. (control operator, 2) The origin of the following operator is also from the work by Felleisen and others [10, 11].

$$\mathcal{A}^a \overset{\text{def}}{=} \lambda x^\alpha.\mu b^\beta.[a]x$$

The direct encoding gives a $\searrow$-normal form, as follows (assuming α and β are non-trivial).

$$[\![\mathcal{A}^a]\!]_u \overset{\text{def}}{=} \overline{u}(c)!c(xz).\overline{a}\langle x^{\overline{\alpha^\circ}}\rangle$$

After signalling itself, the process receives an argument x and a continuation, but discarding that continuation and just sends out x to a.

EXAMPLE 3.11. (control operator, 3) The following is the well-known witness of Peirce's law, $((A \supset B) \supset A) \supset A$, and corresponds to `callcc` in Scheme.

$$\kappa \overset{\text{def}}{=} \lambda y^{(\alpha\Rightarrow\beta)\Rightarrow\alpha}.\mu a^\alpha.[a](y(\lambda x^\alpha.\mu b^\beta.[a]x)).$$

The direct encoding becomes:

$$[\![\kappa]\!]_u \overset{\text{def}}{=} \overline{u}(c)!c(za).(\boldsymbol{\nu}\, m)(\overline{m}\langle z\rangle | !m(z).(\boldsymbol{\nu}\, n)(\overline{n}(f)!f(xb).\overline{a}\langle x\rangle | !n(f).\overline{z}\langle fa\rangle))$$

which can be simplified with some uses of $\searrow$ into:

$$\overline{u}(c)!c(ya).\overline{y}(fa')(!f(xb).\overline{a}\langle x\rangle \mid [a' \to a])$$

The process first signals itself at u, then, when invoked with an argument y and a return point a, asks at y with an argument f and a new return point a'. Then whichever is invoked, it would return with the received value to the initial return point a. Note that the only difference from the encoding of $\aleph$ is whether, in addition to the invocation of the identity function at f, there is the possibility that the direct return comes from the environment: the difference, thus, is, in the standard execution, whether it preserves a current stack to forward the value from the environment or not.

EXAMPLE 3.12. (reasoning on inhabitants through encoding) The encoding offers a precise representation of the behaviour of $\lambda\mu$v-terms which not only is semantically faithful to the source language but also offers a tractable reasoning tool on properties of types and terms, such as type inhabitation. In the following we show an example reasoning which determines the inhabitants of $(\bot \Rightarrow \bot)^\bullet$ up to $\cong_\pi$. Through the definability result we shall discuss later, this in fact gives complete information about inhabitants in $\bot \Rightarrow \bot$ in the $\lambda\mu$v up to a contextual congruence. The following statement intuitively says $(\bot \Rightarrow \bot)^\bullet$ has a unique non-trivial inhabitant semantically.

REMARK 3.13. *If* $\vdash_{\mathtt{O}} P \triangleright u : (()^{!})^{?}$ *then either* $P \cong_\pi \overline{u}(c)!c.\mathbf{0}$ *or* $P \cong_\pi \mathbf{0}$.

We prove this in two steps.

LEMMA 3.14. *Let* $P \in \mathsf{NF}_e$. *Then*

1. *If* $\vdash_{\mathtt{I}} P \triangleright \emptyset$ *then* $P \cong_\pi 0$.
2. *If* $\vdash_{\mathtt{O}} P \triangleright x : ()^{?}$ *then* $P \cong_\pi \overline{x}$ *or* $P \cong_\pi 0$.
3. *If* $\vdash_{\mathtt{I}} P \triangleright u : (()^{?})^{!}, x : ()^{?}$ *or* $\vdash_{\mathtt{I}} P \triangleright u : (()^{?})^{!} \to x : ()^{?}$, *then, up to* $\cong_\pi$, *P is one of the following:* $!u(c).0, !u(c).\overline{c}, !u(c).\overline{x}$.

PROOF: Straightforward from the typing. ■

Now we are ready to establish the remark. First assume that P is in NF_e, for if not, by Proposition 2.7 and strong normalisation of the calculus and Lemma 2.5 we can always find a contextually congruent Q that is in NF_e. By Proposition 2.9, this does not change the relevant behaviour.

Now we proceed by induction of the derivation of $P \in \mathsf{NF}_e$. If $P = 0$ we are done. If $P = P_1|P_2$ with $P_i \in \mathsf{NF}_e$ then assume w.l.o.g. that $P_2 = 0$. We can make this assumption, because by IO-modes, only one of the P_i can be $\mathtt{O}$-moded, the other, here assumed to be P_2, must be $\mathtt{I}$. But this input cannot have a free name because of P's type. By Lemma 3.14 this means that $P_2 \cong_\pi 0$. Now the desired result follows by the

(IH). The case that $P \equiv Q \in \mathsf{NF}_e$ is also immediate by the (IH) and the fact that $\equiv \subseteq \cong_\pi$. This leaves the case that

$$P \equiv \overline{x}(c).P' \quad P' \in \mathsf{NF}_e, x \text{ is the only free name not under an input.}$$

Considering the typing judgement $\vdash_{\mathtt{O}} \overline{x}(c).P' \triangleright u : (()^{!})^{?}$, there are clearly only two non-trivial possibilities.

1. $\vdash_{\mathtt{I}} P' \triangleright c : ()^{!} \to u : (()^{!})^{?}$. In this case $P' \equiv !c.P''$ with $\vdash_{\mathtt{O}} P'' \triangleright \emptyset$. Hence by Lemma 3.14: $P' \equiv !c.0$.
2. $\vdash_{\mathtt{I}} P' \triangleright c : ()^{!}$. In this case $P \equiv \overline{u}(c)!c.P''$ with $\vdash_{\mathtt{O}} P'' \triangleright u : (()^{!})^{?}$. By (IH) now $P'' \cong_\pi 0$ or $P'' \cong_\pi \overline{u}(c)!c.0$.

In summary we know that the P with $\vdash_{\mathtt{O}} P \triangleright u : (()^{!})^{?}$ are given by

$$P_0 \cong_\pi 0 \qquad P_{n+1} \cong_\pi \overline{u}(c)!c.P_n.$$

It is notable that $\{P_i\}_{i>0}$ correspond to the series of closed $\lambda\mu$v-terms $\{M_i : \bot \Rightarrow \bot\}_{i\in\omega}$ where $M_i \stackrel{\text{def}}{=} \mu a.[a]V_{i-1}$ and V_i is defined by the following induction: $V_0 \stackrel{\text{def}}{=} \lambda x^{\bot}.x$ and $V_{n+1} \stackrel{\text{def}}{=} \lambda x^{\bot}.[a]V_n$. We now show $P_i \cong_\pi P_j$ for arbitrary $i, j > 0$ by mathematical induction, using:

– (base case) $P_1 \cong_\pi P_2$; and

– (inductive case) If $P_n \cong_\pi P_{n+1}$ then $P_{n+1} \cong_\pi P_{n+2}$ for each n.

For the base case, by Proposition 2.11 (context lemma), it suffices to consider composition with processes typed as:

$$\vdash_{\mathtt{I}} R \triangleright u : (()^{?})^{!}, x : ()^{?} \qquad \vdash_{\mathtt{I}} R \triangleright u : (()^{?})^{!} \to x : ()^{?}$$

and check the behaviour of a composite process at x. By Proposition 2.9 (1) again, we only have to consider such R in $\searrow$-normal forms. By typing and by the lack of $\searrow$-redex, R is one of $\{!u(c).\mathbf{0},\ !u(c).\overline{c},\ !u(c).\overline{x}\}$ up to $\equiv$, cf. Lemma 3.14. By inspecting reduction, we obtain:

$$P_{1,2}|!u(c).\mathbf{0} \not\Downarrow_x, \quad P_{1,2}|!u(c).\overline{c} \not\Downarrow_x, \quad P_{1,2}|!u(c).\overline{x} \Downarrow_x,$$

which shows $P_1 \cong_\pi P_2$, as required. The inductive case is direct from congruency of $\cong_\pi$.

3.5. Correspondence in Dynamics

The dynamics of $\lambda\mu$-calculi, including its call-by-name and call-by-value versions, has additional complexity due to the involvement of

μ-abstraction. Among others it becomes necessary to use a nested context substitution $M\{\,C[\cdot]\,/\,[a][\cdot]\,\}$ when μ-abstraction and application interact. In stark contrast, the dynamics of π^{c} is quite simple, both in its standard reduction and in its extended reduction: even in the latter (which includes the former), its two reduction rules, reproduced below, are quite simple.

$$\begin{array}{rcl} C[\overline{x}(\vec{y})P] \| !x(\vec{y}).Q & \searrow_r & C[(\boldsymbol{\nu}\,\vec{y})(P|Q)] \,|\, !x(\vec{y}).Q \\ (\boldsymbol{\nu}\,x)!x(\vec{y}).Q & \searrow_g & \mathbf{0} \end{array}$$

Since $\searrow_g$ is solely for removing unnecessary garbage processes, the true dynamics is born by $\searrow_r$, where we simply let two processes with shared channels and different polarities interact. In the following we analyse the dynamics of the $\lambda\mu$v-calculus through the embedding, using the interaction-oriented dynamics of π^{c}. The strong normalisability of $\lambda\mu$v-reduction is an immediate consequence of this analysis.

We need some preparations. First, for a $\lambda\mu$v-term which is also a value, the following construction is useful.

DEFINITION 3.15. (value mapping) Let $\Gamma \vdash V : \alpha; \Delta$. Then we set $[\![V]\!]^{\mathsf{val}}_m \stackrel{\text{def}}{=} P$ iff $[\![V]\!]_u \stackrel{\text{def}}{=} \overline{u}(m)P$.

Note $\overline{u}(m)[\![V]\!]^{\mathsf{val}}_m$ is identical with $[\![V]\!]_u$ up to alpha-equality. Note further, by typing, $[\![V]\!]^{\mathsf{val}}_m$ always has the form $!m(\vec{y}).P$. These observations are useful when we think about the encodings, especially when we apply extended reduction on them.

Second, we shall use the following specific instances of extended reduction in the following development. We write $C^{\text{-}x}[P_i]_i$ to stand for $C[P_1]...[P_n]$ $(n > 0)$ where none of the holes binds s and x does not occur freely in $C[\cdot]$.

PROPOSITION 3.16. *Below we assume typability of processes and programs. In (12f), we assume* $\Gamma \vdash V : \alpha; \Delta$ *such that* $x \notin \mathtt{dom}(\Gamma, \Delta)$. *In (12g), we assume* $\Gamma, x : \alpha \vdash M : \beta; \Delta$ *and* $\Gamma \vdash V : \alpha; \Delta$.

$$\begin{array}{rcll} (\overline{m}(\vec{y})P)\{m(\vec{y}) = Q\} & \searrow^{+} & (\boldsymbol{\nu}\,\vec{y})(P|Q). & \text{(12a)} \\ (\overline{m}(c)!c(\vec{y}).P)\{m(c) = \overline{c}(\vec{y})Q\} & \searrow^{+} & (\boldsymbol{\nu}\,\vec{y})(P|Q). & \text{(12b)} \\ (\boldsymbol{\nu}\,y)([x \to y]^{\tau} \mid !y(\vec{z}).P) & \searrow^{+} & !x(\vec{z}).P. & \text{(12c)} \\ !x(\vec{y}).P \,|\, \overline{x}\langle \vec{v}^{\vec{\tau}} \rangle & \searrow^{+} & !x(\vec{y}).P \,|\, P\{\vec{v}/\vec{y}\}. & \text{(12d)} \\ (\nu a)(C[\overline{x}\langle a^{\tau} \rangle] \| !a(\vec{v}).P) & \searrow^{+} & (\nu a)(C[\overline{x}(a)!a(\vec{v}).P] \| !a(\vec{v}).P) & \text{(12e)} \\ (\boldsymbol{\nu}\,x)([\![x : \alpha]\!]_u | [\![V]\!]^{\mathsf{val}}_x) & \searrow^{*} & [\![V]\!]_u & \text{(12f)} \\ (\boldsymbol{\nu}\,x)([\![M : \beta]\!]_u \,|\, [\![V]\!]^{\mathsf{val}}_x) & \searrow^{*} & [\![M\{V/x\} : \beta]\!]_u. & \text{(12g)} \\ C^{\text{-}x}[[\![M_i]\!]_x]_i\{x(v) = P\} & \searrow^{+} & C[[\![M_i]\!]_x\{x(v) = P\}]_i & \text{(12h)} \end{array}$$

PROOF: (12a) and (12b) are immediate. (12c) is by mechanical induction on τ, see [39, Proposition 5.4]. (12d) uses a form of substitution lemma, see Lemma 3.17 below. (12f) is direct from (12c) (the case $\alpha = \bot$ is also admissible, since in this case $[\![x : \alpha]\!]_u$ simply becomes the inaction). (12e) is by easy inductions on $C[\cdot]$ and τ. Finally, for (12g), we show by induction on M that $[\![M]\!]_u = C^{\text{-}x}[\overline{a_i}\langle x\rangle]_{i\in I}$ and $[\![M\{V/x\}]\!]_u = C^{\text{-}x}[\overline{a_i}(x)[\![V]\!]_x]_{i\in I}$ where the $i \in I$ enumerate all occurrences of x. Then we apply (12g) and $\searrow_g$. The proof of (12h) is by induction on the structure of $C[\cdot]$. ■

For the proof of (12d) we need the following result about the interplay between copy-cat processes and substitution (this is already known from [24] and, in the typed setting, [4]).

LEMMA 3.17. *Assuming typability, we have*

$$(\nu\vec{x})(P \mid \Pi_i[x_i \to y_i]) \searrow^* P\{\vec{y}/\vec{x}\}.$$

PROOF: By induction on P. By typing, x can occur in P only as outputs. Observing this, rather than showing induction, we present a key reasoning, using the process of the following shape:

$$P \equiv C[\overline{x}(a)Q]$$

where x does not occur in C (either bound or free) but can occur in Q. We reason:

$$\begin{array}{rcll}
(\nu x)(P \mid [x \to y]) & \equiv & (\nu x)(C[\overline{x}(a)Q] \mid !x(a).\overline{y}\langle a\rangle) & \\
& \searrow & (\nu x)(C[(\nu a)(\overline{y}\langle a\rangle \mid Q)] \mid !x(a).\overline{y}\langle a\rangle) & \\
& \equiv & C[(\nu a)(\overline{y}\langle a\rangle \mid Q)] \mid (\nu x)(!x(a).\overline{y}\langle a\rangle) & \\
& \equiv & C[(\nu a)(\overline{y}\langle a\rangle \mid Q)] & \\
& \overset{\text{def}}{=} & C[(\nu a)(\overline{y}(b)[b \to a] \mid Q)] & \\
& \equiv & C[\overline{y}(b)(\nu a)([b \to a] \mid Q)] & \\
& \searrow^* & C[\overline{y}(b)(\nu a)(Q\{b/a\})] & \text{(IH)} \\
& \equiv & C[\overline{y}(b)Q\{b/a\})] & \\
& \equiv & P\{y/x\} &
\end{array}$$

Noting P consists of parallel composition of (possibly nested) messages to x and other processes, this shows all occurrences of x can be replaced by y. ■

We are now ready to embark on the analysis of $\lambda\mu$v-reduction through its encoding into π^{c}. Suppose we have reduction $M \longrightarrow M'$ for a $\lambda\mu$v-term M. By Figure 4, Page 17, the generation of reduction can be attributed to one of the following cases.

- (β_v)-rule or (η_v)-rule;
- one of the μ-reduction rules; or
- one of the ζ-reduction rules.

Of those, ζ-reductions require the most attention. Instead of considering the general case (which we shall treat later), let us first take a look at the following concrete $\lambda\mu$v-reduction. Below f and g are typed as $\alpha \Rightarrow \gamma$ and α.

$$M \stackrel{\text{def}}{=} (\mu a^{\alpha\Rightarrow\beta}.[a]\lambda y^{\alpha}.\mu e^{\beta}.[a]f)g \longrightarrow \mu b^{\beta}.[b](\lambda y.\mu e.[b](fg))g \stackrel{\text{def}}{=} M' \quad (13)$$

The encoding into π^{c}-process elucidates ζ-reductions on the uniform basis of name passing interaction. Let us first encode M, writing $\overline{c}\langle\!\langle xu\rangle\!\rangle$ for $(\boldsymbol{\nu}\, n)(!n(y).\overline{c}\langle yu\rangle|\overline{n}\langle x\rangle)$ for brevity.

$$[\![M : \alpha \Rightarrow \beta]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, a)(\overline{a}(c)!c(ye).\overline{a}\langle f\rangle \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \quad (14)$$

On the right of (14), we find two $\searrow_r$-redexes (apart from in $\overline{c}\langle\!\langle gu\rangle\!\rangle$), two outputs and a shared input at a, which are ready to interact. Redexes for the ζ-reduction now arise explicitly as redexes for interactions. Note also these redexes does not depend on whether the argument (g above) is a value or not, directly explaining the shape of (ζ_{fun}) in Figure 4.

To see how M' in (13) results from M in the encoding, we "copy" replications to make these two redexes contiguous, obtaining:

$$(\boldsymbol{\nu}\, a)(\overline{a}(c)!c(ye).(\boldsymbol{\nu}\, a)(\overline{a}\langle f\rangle \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \quad (15)$$

This term is an intermediate form before reducing the mentioned two redexes in (14) and is behaviourally equivalent to (14) (even in the untyped weak bisimilarity). We observe:

$$[\![M' : \alpha\Rightarrow\beta]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, a)(\overline{a}(c)!c(ye).(\boldsymbol{\nu}\, a')(\overline{a'}\langle f\rangle \mid !a'(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle)$$

so the intermediate form (15) is nothing but the encoding of M'. This also shows if we really reduce the two $\searrow$-redexes from (14), the result goes past (15). In general, $M \longrightarrow M'$ does not imply $[\![M]\!]_u \searrow^{+} [\![M']\!]_u$ since $[\![M]\!]_u$ reduces a little further than $[\![M']\!]_u$. However $[\![M']\!]_u$ can catch up with the result by reducing the mentioned two redexes in (15).[2] Based on this observation, we formally state the main result.

[2] This observation suggests we may as well decompose $\searrow_r$ into two steps, one for copying the input to a position contiguous with an output and another for performing the reduction at that site, in order to obtain exact simulation. Since this is not necessary for obtaining the main technical results of the present paper, we use $\searrow$ (which is arguably a natural notion of reduction).

Below $\mathtt{size}(M)$ is the size of M, which is inductively defined as:

$$\begin{array}{rclrcl}
\mathtt{size}(x) &=& 1 & \mathtt{size}([a]M) &=& 1+\mathtt{size}(M)\\
\mathtt{size}(\lambda x.M) &=& \mathtt{size}(M)+1 & \mathtt{size}(\mu a.M) &=& 1+\mathtt{size}(M)\\
\mathtt{size}(MN) &=& \mathtt{size}(M)+\mathtt{size}(N) &&&
\end{array}$$

We use this index for maintaining the well-ordering on reduction. Below $\rightarrow_{\lambda\mu\mathtt{v}}$ is the reduction relation on $\lambda\mu$-terms presented in [27].

PROPOSITION 3.18. *Let M and M' be well-typed. Then $M \rightarrow_{\lambda\mu\mathtt{v}} M'$ implies either $[\![M:\alpha]\!] \equiv [\![M':\alpha]\!]$ such that $\mathtt{size}(M) \gneq \mathtt{size}(M')$, or $[\![M:\alpha]\!]_u \searrow^+ P$ such that $[\![M':\alpha]\!]_u \searrow^* P$.*

PROOF: We begin as follows: $(\beta_v)/(\eta_v)$-reductions are directly simulated by $\searrow$. For $(\beta\mathrm{v})$, if $\alpha, \beta \neq \bot$:

$$\begin{array}{rcl}
[\![(\lambda x^\alpha.M)V:\beta]\!]_u &\searrow^+& (\overline{m}(c)!c(xz).[\![M]\!]_z)\{m(c)=([\![V]\!]_v\{v(x)=\overline{c}\langle xu\rangle\})\}\\
&\searrow^+& (\boldsymbol{\nu}\, x)([\![M:\beta]\!]_u \mid [\![V:\alpha]\!]^{\mathsf{val}}_x)\\
&\searrow^+& [\![M\{V/x\}:\beta]\!]_u
\end{array}$$

The second and third reductions are respectively by (12c) and (12g). If $\alpha = \bot$ but $\beta \neq \bot$, then we can set $V = y$. Noting occurrences of $\bot$-typed variables are all eliminated in the encoding:

$$\begin{array}{rcl}
[\![(\lambda x^\alpha.M)y:\beta]\!]_u &\stackrel{\mathrm{def}}{=}& (\overline{m}(c)!c(z).[\![M]\!]_z)\{m(c)=\overline{c}\langle u\rangle\}\\
&\searrow^+& [\![M:\beta]\!]_u \;\equiv\; [\![M\{y/x\}:\beta]\!]_u.
\end{array}$$

The case for $\alpha = \beta = \bot$ is the degenerate case of the above reduction. For (η_v), assume $x \notin \mathsf{fv}(V)$ and $[\![V]\!]_m \stackrel{\mathrm{def}}{=} \overline{m}(c)!c(x'z').P$.

$$\begin{array}{rcl}
[\![\lambda x^\alpha.(Vx):\alpha\Rightarrow\beta]\!]_u &\searrow^+& \overline{u}(c)!c(xz).((\overline{m}(c)!c(x'z').P)\{m(c)=\overline{c}\langle xz\rangle\})\\
&\searrow^+& \overline{u}(c)!c(xz).P\{xz/x'z'\} \quad\equiv\quad [\![V:\alpha\Rightarrow\beta]\!]_u
\end{array}$$

Above we used Proposition 3.16. The μ-reductions become the structural equality:

$$\begin{array}{lrcl}
(\mu\text{-}\beta) & [\![[b]\mu a.M:\bot]\!]_u &\equiv& ([\![M:\bot]\!]\{m/a\})\{b/m\}\\
&&\equiv& [\![M:\bot]\!]\{b/a\}\\
&&\stackrel{\mathrm{def}}{=}& [\![M\{b/a\}:\bot]\!].\\
(\mu\text{-}\eta) & [\![\mu a.[a]M:\alpha]\!]_u &\equiv& ([\![M:\alpha]\!]_m\{a/m\})\{u/a\}\\
&&\stackrel{\mathrm{def}}{=}& [\![M:\alpha]\!]_u.
\end{array}$$

In this way the encoding shows the μ-reductions are "obviously right".

The only remaining cases are the general cases of ζ-reductions. Let $M \stackrel{\mathrm{def}}{=} C[[a]L_i]_{i\in I}$ where I enumerates all a-named subterms (with possible nesting) and let $[\![M:\bot]\!] \stackrel{\text{def}}{=} C'[[\![L_i]\!]_a]_i$ accordingly. Note all free occurrences of a in $[\![M\!:\!\bot]\!]$ are exhaustively mentioned in this way. Given $C[[a]L_i]_i$, we write $C[[b]L_iN]_i$ (say) to indicate the result of filling each hole with a new subterm similarly we write $C'[[\![L_i]\!]_a\{a(x)\!=\!R'\}]$. Here R' is just a placeholder. We show one case of (ζ_{fun}). Below we assume $\beta \neq \bot$ and we let either $R \stackrel{\text{def}}{=} [\![N]\!]_n\{n(e)\!=\!\overline{c}\langle eu\rangle\}$ (if $\alpha \neq \bot$) or $R \stackrel{\text{def}}{=} \overline{c}\langle u\rangle$ (if $\alpha = \bot$).

$$
\begin{array}{lrcll}
(\boldsymbol{\zeta_{\mathbf{fun}}}) & [\![(\mu a^{\alpha\Rightarrow\beta}.M)N]\!]_u & \stackrel{\text{def}}{=} & [\![M\!:\!\bot]\!]\{m/a\}\{m(c)\!=\!R\} & \\
 & & \equiv & [\![M\!:\!\bot]\!]\{a(c)\!=\!R\} & \\
 & & \equiv & C'^{\text{-}a}[[\![L_i]\!]_a]_i\,\{a(c)\!=\!R\} & \\
 & & \searrow^{+} & C'^{\text{-}a}[[\![L_i]\!]_a\{a(c)\!=\!R\}]_i & (16)\\
 & & \stackrel{\text{def}}{=} & C'^{\text{-}a}[[\![L_iN]\!]_u]_i & \\
 & & \equiv & C'^{\text{-}a}[[\![L_iN]\!]_n\{b/n\}]_i\{u/b\} & \\
 & & \stackrel{\text{def}}{=} & C'^{\text{-}a}[[\![[b](L_iN)]\!]]_i\{u/b\} & \\
 & & \stackrel{\text{def}}{=} & [\![C[[b](L_iN)]_i]\!]\{u/b\} & \\
 & & \stackrel{\text{def}}{=} & [\![\mu b.C[[b](L_iN)]]\!]_u &
\end{array}
$$

Here (16) is an application of Proposition 3.16 (12h).

The case when the target type is $\bot$ is similarly reasoned as follows, again relying on Proposition 3.16 (12h). Below let $R \stackrel{\text{def}}{=} [\![N]\!]_n\{n(e) = \overline{c}\langle e\rangle\}$ if $\beta \neq \bot$, or $R \stackrel{\text{def}}{=} \overline{c}$ if $\beta = \bot$. Further let $P_1 \searrow^{+}\!\swarrow P_2$ denote $P_i \searrow^{+} P'$ $(i = 1, 2)$ for some P'.

$$
\begin{array}{llcl}
(\boldsymbol{\zeta_{\mathbf{fun},\bot}}) & [\![(\mu a^{\alpha\Rightarrow\bot}.M)N]\!]_u & \stackrel{\text{def}}{=} & [\![M\!:\!\bot]\!]\{m/a\}\{m(c)\!=\!R\} \\
 & & \equiv & C'[[\![L_i]\!]_a]_i\,\{a(c)\!=\!R\} \\
 & & \searrow^{+}\!\swarrow & C'[[\![L_i]\!]_{l_i}\{l_i(c)\!=\!R\}]_i \\
 & & \stackrel{\text{def}}{=} & [\![C[(L_iN)]]\!]_u.
\end{array}
$$

Next we consider the case for $(\bot)$. Let $b, c \notin \mathsf{fv}(N)$ and $\beta \neq \bot$. Then we have:

$$
\begin{array}{rcl}
[\![(\lambda x^{\bot\Rightarrow\beta}.M)N]\!]_u & \stackrel{\text{def}}{=} & (\nu m)(\overline{m}(c)!c(r).[\![M]\!]_r \mid !m(c).[\![N]\!]_u) \\
 & \searrow & (\nu mc)(!c(r).[\![M]\!]_r \mid [\![N]\!]_n \mid !m(c).[\![N]\!]_u) \\
 & \searrow^{+} & [\![N]\!]_n \stackrel{\text{def}}{=} [\![\mu b^{\beta}.N]\!]_n,
\end{array}
$$

noting b is fresh in the last line. The case for $(\bot_\bot)$ is similar.

(ζ_{arg}) rules are treated in Appendix A.3, again clarifying the notion of redexes in these rules. ■

Proposition 3.18 implies an infinite $\lambda\mu$v-reduction sequence means either an infinite term-size (which is impossible) or an infinite $\searrow$-reduction sequence (which is again impossible). Thus we conclude:

COROLLARY 3.19. *$\to_{\lambda\mu\mathrm{v}}$ on $\lambda\mu$-terms is strongly normalising.*

4. Decoding and Full Abstraction

4.1. Canonical Normal Forms

In the previous section we have shown that types and dynamics of $\lambda\mu$v-terms are faithfully embeddable into π^{c}. In this section we show this embedding is as faithful as possible — if a process lives in the encoding of a $\lambda\mu$v-type, then it is indeed the image of a $\lambda\mu$v-term of that type. This result corresponds to the standard definability result in denotational semantics, and immediately leads to full abstraction for a suitably defined observational congruence for $\lambda\mu$v.

A key observation towards definability is that we can algorithmically translate back processes having the encoded $\lambda\mu$v-types into the original $\lambda\mu$v-terms. To study the decoding, it is convenient to introduce *canonical normal forms* (CNFs) [1, 4, 18], which are essentially a subset of $\lambda\mu$v-terms whose syntactic structures precisely correspond to their process representation.

First, *CNF preterms* ($\mathtt{N}, \ldots$), together with its subset *CNF value preterms* ($\mathtt{U}, \ldots$), are given by the following grammar.

$$\begin{aligned}
\mathtt{N} &::= \mathtt{c} \mid \lambda x^{\alpha}.\mathtt{N} \mid \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N} \mid \mathtt{let}\ \_ = y\mathtt{U} \mid [a]\mathtt{U} \mid \mu a^{\alpha}.\mathtt{N}\\
\mathtt{U} &::= \mathtt{c} \mid \lambda x^{\alpha}.\mathtt{N} \mid \mu a^{\alpha}.[a]\mathtt{U}
\end{aligned}$$

Some observations on the grammar above:

– $\mathtt{c}$ is a constant with type $\bot$, giving a pseudo inhabitant of this type. While we can always eliminate its occurrences from a typable CNF using variables of $\bot$-type, it allows us to have a direct correspondence between CNFs and $\lambda\mu$v-processes, as will be clarified in the proof of correctness of decoding (Lemma 4.6) later.

– Following [1, 4, 18], the let construct, $\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}$, is used for directly representing interactive behaviour of π^{c}-processes in the universe of $\lambda\mu$v. Intuitively, $\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}$ corresponds to the process which:

$$\textsf{(}\bot\textsf{-const)}\quad \frac{-}{\Gamma\cdot x\!:\!\bot \vdash \mathtt{c}\!:\!\bot\,;\Delta}$$

$$\textsf{(let)}\quad \frac{\Gamma\cdot x\!:\!\beta \vdash \mathtt{N}\!:\!\gamma\,;\Delta \qquad \Gamma \vdash y\mathtt{U}\!:\!\beta\,;\Delta \quad (\beta\neq\bot)}{\Gamma \vdash \mathtt{let}\ x^{\beta} = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}\!:\!\gamma\,;\Delta}
\qquad
\textsf{(let-}\bot\textsf{)}\quad \frac{\Gamma \vdash y\!:\!\alpha\Rightarrow\bot\,;\emptyset \qquad \Gamma \vdash \mathtt{U}\!:\!\alpha\,;\Delta}{\Gamma \vdash \mathtt{let}\ \_ = y\mathtt{U}\!:\!\bot\,;\Delta}$$

Figure 6. Typing rules for CNFs.

- outputs with a **?**-action at y carrying a name which is a subject of a "value" process U and a continuation; then
- inputs at that continuation with a formal parameter x, and behaves as N.

– `let _` $= y$U is the degenerate case of `let` $x = y$U `in` N when x is of type $\bot$.

We further assume the following conditions on CNF preterms.

1. In $[a]$N, N does not have form μb^{β}.N$'$.
2. In μa^{α}.N,
   a) if N is $[a]$U then $a \in \mathsf{fn}(\mathtt{U})$; and
   b) if N is `let` $x = y$U$'$ `in` N$'$ then $a \in \mathsf{fn}(\mathtt{U}')$.
3. In $\mu a^{\alpha}.[a]$U, $a \in \mathsf{fn}(\mathtt{U})$.

The conditions 1, 2-a and 3 are to avoid a μ-redex. The condition 2-b is to determine the shape of a normal form, since without this condition $\mu a.$`let` $x = y$U$'$ `in` N$'$ can be written `let` $x = y$U$'$ `in` $\mu a.$N$'$.

Using the CNF-preterms under these conditions, the set of CNFs are those which are typable by the typing rules in Figure 6 combined with those in Figure 3 except the rule for application. In ($\bot$-const) in Figure 6, c, which witnesses absurdity, is introduced only when $\bot$ is assumed in the environment (logically this says that we can say an absurd thing only when the environment is absurd). CNFs which are also CNF value preterms are called *CNF values.* Note a CNF value is either c (which is the sole case when it has a type $\bot$), a λ-abstraction, or a μ-abstraction followed by a λ-abstraction.

$$\langle \mathtt{c}\!:\!\bot\rangle_u \stackrel{\text{def}}{=} \mathbf{0}$$

$$\langle \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}\!:\!\gamma\rangle_u \stackrel{\text{def}}{=} \begin{cases} \overline{y}(wz)(P|!z(x).\langle \mathtt{N}\!:\!\gamma\rangle_u) & (\mathtt{U} \neq \mathtt{c},\\ & \quad \langle \mathtt{U}\rangle_c \stackrel{\text{def}}{=} \overline{c}(w)P) \\ \overline{y}(z)!z(x).\langle \mathtt{N}\!:\!\gamma\rangle_u & (\mathtt{U} = \mathtt{c}) \end{cases}$$

$$\langle \mathtt{let}\ \_ = y\mathtt{U}\!:\!\bot\rangle_u \stackrel{\text{def}}{=} \begin{cases} \langle \mathtt{U}\rangle_y & (\mathtt{U} \neq \mathtt{c}) \\ \overline{y} & (\mathtt{U} = \mathtt{c}) \end{cases}$$

Figure 7. Encoding of CNFs.

CNFs correspond to $\lambda\mu$v-terms as follows. In the first rule we assume x is chosen arbitrarily from variables assigned to $\bot$. Below in the first line, it is semantically (and logically) irrelevant which $\bot$-typed variable we choose: for example, we may assume there is a total order on names and choose the least one from the given environment.

$$(\Gamma\cdot x\!:\!\bot \vdash \mathtt{c}\!:\!\bot\,;\Delta)^* \stackrel{\text{def}}{=} \Gamma\cdot x\!:\!\bot \vdash x\!:\!\bot\,;\Delta$$
$$(\Gamma \vdash \mathtt{let}\ x^\beta = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}\!:\!\gamma\,;\Delta)^* \stackrel{\text{def}}{=} \Gamma \vdash (\lambda x.\mathtt{N}^*)(y\mathtt{U}^*)\!:\!\gamma\,;\Delta$$
$$(\Gamma \vdash \mathtt{let}\ \_ = y\mathtt{U}\!:\!\bot\,;\Delta)^* \stackrel{\text{def}}{=} \Gamma \vdash y\mathtt{U}^*\!:\!\bot\,;\Delta$$

For CNFs which are λ-abstraction, μ-abstraction and named terms, the mapping uses the same clauses as in Figure 5, replacing $[\![\cdot]\!]$ in the defining clauses with $(\cdot)^*$.

Via $(\ )^*$ we can encode CNFs to processes:

$$\Gamma \vdash \mathtt{N} : \alpha; \Delta \quad \mapsto \quad \Gamma \vdash \mathtt{N}^* : \alpha; \Delta \quad \mapsto \quad \vdash_{\mathsf{0}} [\![(\mathtt{N} : \alpha)^*]\!]_u \triangleright (u\!:\!\alpha, \Delta)^\bullet, \Delta^\circ$$

CNFs can also be directly encoded into π^{c}-processes, using the rules in Figure 7 combined with those for abstraction, naming and μ-abstraction given in Figure 5 (replacing $[\![\cdot]\!]$ with $\langle\cdot\rangle$ in each clause).

PROPOSITION 4.1. *Let* $\Gamma \vdash \mathtt{N}\!:\!\alpha\,;\Delta$*. Then* $\vdash_{\mathsf{0}} \langle\mathtt{N}\rangle_u \triangleright (u\!:\!\alpha, \Delta)^\bullet, \Gamma^\circ$.

PROOF: By induction on the typing rules for CNFs. The only interesting case is when $\mathtt{N} \stackrel{\text{def}}{=} \mathtt{let}\ \_ = y\mathtt{U}$, where we use the translation of negation by implication. By induction hypothesis, we assume:

$$\vdash_{\mathsf{0}} \langle\mathtt{U}\rangle_u \triangleright (u\!:\!\alpha, \Delta)^\bullet, \Gamma^\circ. \tag{17}$$

If $\alpha = \bot$, we have $\mathtt{U} = \mathtt{c}$, in which case, observing $\overline{(\bot \Rightarrow \bot)^\circ} = ()^?$:

$$\vdash_{\mathsf{0}} \overline{y} \triangleright (\Delta)^\bullet, (\Gamma, y\!:\!\bot \Rightarrow \bot)^\circ.$$

If $\alpha \neq \bot$, using the induction hypothesis (17) and noting $\alpha^\bullet = (\alpha^\circ)^? = (\alpha \Rightarrow \bot)^\circ$, we obtain:

$$\vdash_{\mathtt{O}} \langle \mathtt{U} \rangle_y \triangleright (\Delta)^\bullet, (\Gamma, y{:}\alpha \Rightarrow \bot)^\circ.$$

For other cases, λ-abstraction, μ-abstraction and named terms are as in Proposition 3.4. The case $\mathtt{N} = \mathtt{c}$ is immediate. The case $\langle \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}{:}\gamma \rangle_u$ is direct from the induction hypothesis. ■

Two process encodings of CNFs coincide up to $\searrow$.

PROPOSITION 4.2. *Let* $\Gamma \vdash \mathtt{N}{:}\alpha\,;\Delta$. *Then* $[\![\mathtt{N}^* : \alpha]\!]_u \searrow^* \langle \mathtt{N} \rangle_u \not\searrow$.

PROOF: Let $\mathtt{N}$ be a CNF. We first show $\langle \mathtt{N} \rangle_u \not\searrow$, that is $\langle \mathtt{N} \rangle_u \in \mathsf{NF}_e$, by induction on $\mathtt{N}$. We use the inductive characterisation of $\searrow$-normal forms in Proposition 2.7.

1. $\langle \mathsf{c} \rangle \stackrel{\text{def}}{=} \mathbf{0} \in \mathsf{NF}_e$.

2. $\langle \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}{:}\gamma \rangle_u \stackrel{\text{def}}{=} \overline{y}(wz)(P|!z(x).\langle \mathtt{N} \rangle_u) \in \mathsf{NF}_e$ where $\mathtt{U} \neq \mathtt{c}$, $\langle \mathtt{U} \rangle_c \stackrel{\text{def}}{=} \overline{c}(w)P \in \mathsf{NF}_e$ (hence $P \in \mathsf{NF}_e$) and $\langle \mathtt{N} \rangle_u \in \mathsf{NF}_e$. We must show that $\overline{y}(wz)...$ is a prime output. For this we need to establish that every free name, distinct from wz, in P is under an input prefix. But as $\overline{c}(w)P \in \mathsf{NF}_e$ we know $\overline{c}(w)P$ to be a prime output. Hence P is of the form $!w(\vec{a}).P'$. This guarantees that $\overline{y}(wz)...$ is prime. This also means that free names in P' as well as in $\langle N \rangle_u$ are output-moded, hence of the same polarity. By construction w and z are freshly chosen, so cannot occur as outputs in the relevant terms. Hence P and $!z(x).\langle \mathtt{N} \rangle_u$ have no names of different polarities in common and their parallel composition is indeed in NF_e.

3. $\langle \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N} : \gamma \rangle_u \stackrel{\text{def}}{=} \overline{u}(z)!z(x).\langle \mathtt{N} \rangle_u \in \mathsf{NF}_e$ if $\mathtt{U} = \mathtt{c}$ and $\langle \mathtt{N} \rangle_u \in \mathsf{NF}_e$.

4. $\langle \mathtt{let}\ \_ = y^{\alpha \Rightarrow \bot}\mathtt{U} : \bot \rangle_u \stackrel{\text{def}}{=} \langle \mathtt{U} \rangle_y \in \mathsf{NF}_e$ with $\alpha \neq \bot$ directly from induction hypothesis.

5. $\langle \mathtt{let}\ \_ = y^{\bot \Rightarrow \bot}\mathtt{U}{:}\bot \rangle_u \stackrel{\text{def}}{=} \overline{y} \in \mathsf{NF}_e$.

Next we show $[\![\mathtt{N}^* : \alpha]\!]_u \searrow^* \langle \mathtt{N} \rangle_u$, remembering $\searrow$ is compatible.

1. $[\![\mathsf{c}^*]\!]_u \stackrel{\text{def}}{=} \mathbf{0} \stackrel{\text{def}}{=} \langle \mathsf{c} \rangle$.

2. Let $\alpha \neq \bot$. By induction hypothesis we set $[\![\mathtt{U}^*]\!]_n \Downarrow_e \langle \mathtt{U} \rangle_n \stackrel{\text{def}}{=} \overline{n}(w)P$ and $[\![\mathtt{N}^*]\!]_u \Downarrow_e \langle \mathtt{N} \rangle_u$. Using Proposition 3.16 the second step we obtain the following derivation.

$$\begin{aligned}
[\![(\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N})^*]\!]_u &\stackrel{\text{def}}{=} [\![(\lambda x.\mathtt{N}^*)(y\mathtt{U}^*)]\!]_u \\
&\stackrel{\text{def}}{=} (\boldsymbol{\nu}\, f)(\,\overline{f}(z)!z(xv).[\![\mathtt{N}^*]\!]_v \mid \\
&\qquad !f(z).(\boldsymbol{\nu}\, g)\,(\,(\boldsymbol{\nu}\, l)(\bar{l}\langle y\rangle \mid !l(c).(\boldsymbol{\nu}\, n)([\![\mathtt{U}^*]\!]_n | !n(w).\bar{c}\langle wg\rangle)) \mid \\
&\qquad\qquad !g(x).\overline{z}\langle xu\rangle)) \\
&\searrow^+ \ \overline{y}(wz)(P | !z(x).\langle \mathtt{N} \rangle_u)
\end{aligned}$$

3. The reduction when $U = \mathtt{c}$ is very similar to the last.

4. If $\mathtt{U} \neq \mathtt{c}$, then $[\![(\mathtt{let}\ \_ = y^{\alpha \Rightarrow \bot}\mathtt{U})^*]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\overline{m}\langle y\rangle | !m(c).[\![\mathtt{U}^*]\!]_c)$ which, combined with induction hypothesis, reduces to $\langle \mathtt{U} \rangle_y$.

5. if $\mathtt{U} = \mathtt{c}$, then $[\![(\mathtt{let}\ \_ = y^{\bot \Rightarrow \bot}\mathtt{U})^*]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)(\overline{m}\langle y\rangle | !m(c).\overline{c}) \searrow^+ \overline{y}$.

The cases for λ-abstraction, μ-abstraction and the named terms are direct from the induction hypothesis, noting $\searrow$ is closed under type-correct name substitution. ■

We present CNFs corresponding to the $\lambda\mu$v-terms discussed in Examples 3.6-3.11. We also list the corresponding π^{c}-process in each case, adumbrating the definability arguments in the next subsection.

EXAMPLE 4.3. (CNFs)

1. A variable $x^\bot$ is $\mathbf{0}$ as a process; which becomes $\mathtt{c}$ in CNF.

2. For $x^{\bot \Rightarrow \bot}$, its process representation is $\overline{u}(c)!c.\mathbf{0}$, which becomes $\lambda y^\bot.\mathtt{let}\ \_ = x\mathtt{c}$ in CNF (which is just the η-expansion of x).

3. In general, the CNF corresponding to a variable x^α, written $\eta\text{-}\mathsf{ex}(x^\alpha)$, is inductively given as: $\eta\text{-}\mathsf{ex}(x^\bot)$ is given in 1 above; $\eta\text{-}\mathsf{ex}(x^{\alpha \Rightarrow \beta})$ is given in 2 above if $\alpha = \beta = \bot$; and if else

   a) $\lambda y^\alpha.\mathtt{let}\ z = xy\ \mathtt{in}\ \eta\text{-}\mathsf{ex}(z^\beta)$, if $\alpha, \beta \neq \bot$;

   b) $\lambda y^\bot.\mathtt{let}\ z = x\mathtt{c}\ \mathtt{in}\ \eta\text{-}\mathsf{ex}(z^\beta)$, if $\alpha = \bot$ and $\beta \neq \bot$; and

   c) $\lambda y^\alpha.\mathtt{let}\ \_ = xy$, if $\alpha \neq \bot$ and $\beta = \bot$.

4. A CNF corresponding to the identity $\lambda x^\alpha.x$ is $\lambda x^\alpha.\eta\text{-}\mathsf{ex}(x^\alpha)$.

5. Recall, from Example 3.9. $\aleph \stackrel{\text{def}}{=} \lambda z^{\neg\neg\alpha}.\mu a^{\alpha}.z(\lambda x^{\alpha}.[a]x)$. Its process representation is

$$\overline{u}(c)!c(za).\overline{z}(f)!f(x).\overline{a}\langle x^{\overline{\alpha^\circ}}\rangle$$

while its corresponding CNF becomes, using $\eta\text{-}\mathsf{ex}(x)$ above:

$$\lambda z^{\neg\neg\alpha}.\mu a^{\alpha}.\mathtt{let}\ \_ = z(\lambda x^{\alpha}.[a]\eta\text{-}\mathsf{ex}(x^{\alpha}))$$

Note there is a close correspondence in syntactic structures.

6. Recall, from Example 3.10. $\mathcal{A}^{a} \stackrel{\text{def}}{=} \lambda x^{\alpha}.\mu b^{\beta}.[a]x$. Its process encoding is $\overline{u}(c)!c(xz).\overline{a}\langle x^{\overline{\alpha^\circ}}\rangle$ while its CNF becomes $\lambda x^{\alpha}.\mu b^{\beta}.[a]\eta\text{-}\mathsf{ex}(x^{\alpha})$.

7. Recall, from Example 3.11,

$$\kappa \stackrel{\text{def}}{=} \lambda y^{(\alpha\Rightarrow\beta)\Rightarrow\alpha}.\mu a^{\alpha}.[a](y(\lambda x^{\alpha}.\mu b^{\beta}.[a]x)).$$

Its process representation is

$$\overline{u}(c)!c(ya).\overline{y}(fc')(!f(xb).\overline{a}\langle x^{\overline{\alpha^\circ}}\rangle \mid !c'(z).\overline{a}\langle z^{\overline{\alpha^\circ}}\rangle)$$

while its CNF is

$$\lambda y^{(\alpha\Rightarrow\beta)\Rightarrow\alpha}.\mu a^{\alpha}.\mathtt{let}\ z^{\beta} = y(\lambda x^{\alpha}.\mu b^{\beta}.[a]\eta\text{-}\mathsf{ex}(x^{\alpha}))\ \mathtt{in}\ [a]\eta\text{-}\mathsf{ex}(z^{\alpha}).$$

There is a close syntactic correspondence again.

4.2. Definability

The decoding of π^{c}-processes (of encoded $\lambda\mu\mathrm{v}$-types) to $\lambda\mu\mathrm{v}$-preterms is written $[P]_{u}^{\Gamma;\Delta}$, which translates $P \in \mathsf{NF}_e$ such that $\vdash_{\mathsf{0}} P \triangleright \Gamma^{\circ}, \Delta^{\bullet}$ with $u \notin \mathtt{dom}(\Gamma)$ to a $\lambda\mu\mathrm{v}$-preterm M. Without loss of generality, we assume P does not contain redundant $\mathbf{0}$ or hiding. The mapping is defined inductively by the rules given in Figure 8. In the second last line, $P_{\langle a\rangle}$ indicates P is a prime output with subject a, whereas $P_{\langle m/a\rangle}$ is the result of replacing the subject a in $P_{\langle a\rangle}$ with m.

PROPOSITION 4.4. *The map is well-defined and total on* NF_e*-processes* $\vdash_{\mathsf{0}} P \triangleright \Gamma^{\circ}, \Delta^{\bullet}$.

PROOF: The map is well-defined since each rule decreases the size of processes. This is indirect in the last two rules, for which we observe:

1. If the second last rule is applicable then one of the second to the fifth rules becomes applicable to the result of the mapping.

$$\begin{array}{rcll}
[\mathbf{0}]_u^{\Gamma;\,\Delta} & \stackrel{\text{def}}{=} & \mathtt{c}\!:\!\bot & u \notin \mathtt{dom}(\Delta) \\
[\overline{u}(c)!c(xz).R]_u^{\Gamma;\,\Delta\cdot u:(\alpha\Rightarrow\beta)} & \stackrel{\text{def}}{=} & \lambda x^{\alpha}.[R]_z^{\Gamma\cdot x:\overline{\alpha};\,\Delta\cdot z:\beta} & u \notin \mathsf{fn}(R) \\
[\overline{u}(c)!c(z).R]_u^{\Gamma;\,\Delta\cdot u:(\bot\Rightarrow\beta)} & \stackrel{\text{def}}{=} & \lambda x^{\bot}.[R]_z^{\Gamma;\,\Delta\cdot z:\beta} & u \notin \mathsf{fn}(R) \\
[\overline{u}(c)!c(x).R]_u^{\Gamma;\,\Delta\cdot u:(\alpha\Rightarrow\bot)} & \stackrel{\text{def}}{=} & \lambda x^{\alpha}.[R]_m^{\Gamma\cdot x:\overline{\alpha};\,\Delta} & u \notin \mathsf{fn}(R) \\
[\overline{u}(c)!c.R]_u^{\Gamma;\,\Delta\cdot u:(\bot\Rightarrow\bot)} & \stackrel{\text{def}}{=} & \lambda x^{\bot}.[R]_m^{\Gamma;\,\Delta} & u \notin \mathsf{fn}(R) \\
[\overline{y}(wz)(R\,|\,!z(x).Q)]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} & \stackrel{\text{def}}{=} & & \\
\multicolumn{3}{r}{\mathtt{let}\ x^{\beta}\!= y[\overline{c}(w)R]_c^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta}\mathtt{in}\ [Q]_u^{(\Gamma\cdot x:\beta);\Delta}} & u \notin \mathsf{fn}(R) \\
[\overline{y}(z)!z(x).Q]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} & \stackrel{\text{def}}{=} & \mathtt{let}\ x^{\beta}\!= y\mathtt{c}\ \mathtt{in}\ [Q]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta\cdot x:\beta;\Delta} & \\
[\overline{y}(w)R]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} & \stackrel{\text{def}}{=} & \mathtt{let}\ \_ = y[\overline{c}(w)P]_c^{\Gamma;\Delta} & u \notin \mathsf{fn}(R) \\
[\overline{y}]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} & \stackrel{\text{def}}{=} & \mathtt{let}\ \_ = y\mathtt{c} & \\
[P_{\langle a\rangle}]_u^{\Gamma;\,\Delta\cdot a:\alpha} & \stackrel{\text{def}}{=} & [a][P_{\langle m/a\rangle}]_m^{\Gamma;\Delta\cdot a:\alpha\cdot m:\alpha} & u \notin \mathtt{dom}(\Delta) \\
[P]_u^{\Gamma;\,\Delta\cdot u:\alpha} & \stackrel{\text{def}}{=} & \mu u^{\alpha}.[P]_m^{\Gamma;\,\Delta\cdot u:\alpha} & \text{other cases}
\end{array}$$

We assume $\alpha, \beta \neq \bot$ and m fresh.

Figure 8. Decoding of $\lambda\mu$-typed processes.

2. If the last rule is applicable, then the map translates:
    a) $[\mathbf{0}]_u^{\Gamma;\Delta\cdot u:\alpha}$ such that $u \in \mathtt{dom}(\Delta)$ into $\mu u.[\mathbf{0}]_m^{\Gamma;\Delta\cdot u:\alpha}$, to which the first rule is applicable.
    b) $[\overline{u}(c)S]_u^{\Gamma;\Delta}$ such that $u \in \mathsf{fn}(R)$ into $\mu u.[\overline{u}(c)S]_m^{\Gamma;\Delta}$, to which the second last rule becomes applicable.
    c) $[\overline{y}(wz)(R|!z(x).Q)]_u^{\Gamma;\Delta}$ (resp. $[\overline{y}(w)R]_u^{\Gamma;\Delta}$) such that $u \in \mathsf{fn}(R)$ into $\mu u.[\overline{y}(wz)(R|!z(x).Q)]_m^{\Gamma;\Delta}$ (resp. $\mu u.[\overline{y}(w)R]_m^{\Gamma;\Delta}$), to which the sixth or the eighth rule becomes applicable.

Totality on NF_e is immediate from the rules except for parallel composition. But if $P \in \mathsf{NF}_e$ and, by typing, P is output-moded, up to redundant Os and hiding, P must be a prime output. Hence NF_e processes of translated $\lambda\mu$-type do not contain non-trivial outermost parallel composition. ■

For typability, write $\mathtt{im}(\Gamma)$ for the image of Γ:

PROPOSITION 4.5. *Let $\bot \notin \mathtt{im}(\Gamma)$, $u \notin \mathtt{dom}(\Gamma)$ and $P \in \mathsf{NF}_e$. Then $\vdash P \triangleright \Gamma^{\circ} \cdot \Delta^{\bullet}$ implies, with x fresh:*

1. *if* $\Delta = \Delta_0 \cdot u{:}\alpha$ *then* $\Gamma \cdot x{:}\bot \vdash [P]_u^{\Gamma^\circ;\Delta^\bullet} : \alpha\,; \Delta_0$ *and*

2. *if* $u \notin \mathtt{dom}(\Delta)$ *then* $\Gamma \cdot x{:}\bot \vdash [P]_u^{\Gamma^\circ;\Delta^\bullet} : \bot\,; \Delta_0$.

Proof: Immediate by inspecting each rule (the additional $x : \bot$ becomes necessary only for $\mathtt{c} : \bot$, which can be eliminated whenever the occurrence(s) of $\mathtt{c}$ are placed under the scope of a λ-abstraction of a $\bot$-typed variable). ■

We now prove the decoding of π^{c}-processes is the inverse of the encoding of CNFs.

LEMMA 4.6. *If* $\vdash P \triangleright \Gamma^\circ \cdot \Delta^\bullet \in \mathsf{NF}_e$ *s.t.* $u \notin \mathtt{dom}(\Gamma)$ *then* $\langle [P]_u^{\Gamma^\circ;\Delta^\bullet} \rangle_u \equiv P$. *Conversely, if* $\Gamma \vdash \mathtt{N} : \beta\,;\Delta$ *s.t.* $u \notin \mathtt{dom}(\Gamma)$ *then* $[\langle \mathtt{N} \rangle_u]_u^{\Gamma^\circ;\Delta^\bullet} \equiv_\alpha \mathtt{N}$.

Proof: Let $\vdash P \triangleright \Gamma^\circ \cdot \Delta^\bullet \in \mathsf{NF}_e$ with $u \notin \mathtt{dom}(\Gamma)$. We first show $\langle [P]_u^{\Gamma^\circ;\Delta^\bullet} \rangle_u \equiv P$ by rule induction on rules in Figure 8. All rules except the last two rules are immediate by comparing the defining clauses of Figure 8, on the one hand, and those of Figure 7 and Figure 5 (the latter excepting the application). We now reason for the last two rules. In the following, $P \overset{[\,]_u}{\longmapsto} Q$ means an application of $[\,]_u$ to P results to Q. Similarly for $\overset{\langle\,\rangle_u}{\longmapsto}$.

For the second last rule, assuming $\vdash P \triangleright \Gamma^\circ \cdot \Delta^\bullet \cdot a : \beta^\bullet$:

$$P_{\langle a \rangle} \overset{[\,]_u}{\longmapsto} [a][P_{\langle m/a \rangle}]_m \overset{\langle\,\rangle_u}{\longmapsto} \langle [P_{\langle m/a \rangle}]_m \rangle_m \{a/m\} \overset{\text{def}}{=} (P_{\langle m/a \rangle})\{a/m\} \equiv P$$

For the last rule, we have:

$$P \overset{[\,]_u}{\longmapsto} \mu u^\beta.[P]_m \overset{\langle\,\rangle_u}{\longmapsto} \langle \mu u^\beta.[P]_m \rangle_u \overset{\text{def}}{=} \langle [P]_m \rangle_m \equiv P,$$

as required.

For the other direction, assume $\Gamma \vdash M : \beta\,;\Delta$ and $u \notin \mathtt{dom}(\Gamma)$. We show $[\langle \mathtt{N} \rangle_u]_u^{\Gamma^\circ;\Delta^\bullet} \equiv_\alpha \mathtt{N}$ by induction on the rules in Figures 7 and 5. Again all cases are easy except named terms and μ-abstraction. For the former, assuming $\Gamma \vdash [a]\mathtt{U} : \bot\,;\Delta$:

$$\begin{aligned} [a]\mathtt{U}{:}\bot &\overset{\langle\,\rangle_u}{\longmapsto} \langle \mathtt{U}{:}\beta \rangle_m \{a/m\} \overset{[\,]_u}{\longmapsto} [a]([\langle \mathtt{U}{:}\beta \rangle_m \{a/m\}_{\langle m/a \rangle}]_m) \\ &\equiv_\alpha \quad [a]\mathtt{U}{:}\bot \end{aligned}$$

For μ-abstraction, let $\Gamma \vdash \mu a^\beta.\mathtt{N}' : \beta\,;\Delta$. Note `N'` has type $\bot$. Hence $\mathtt{N}'$ has the shape of either $\mathtt{c}$, $\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}'$ with $\mathtt{N}''$ of $\bot$ type, $\mathtt{let}\ \_ = y\mathtt{U}$ or $[a]\mathtt{U}$. If $\mathtt{N}' \overset{\text{def}}{=} \mathtt{c}$, then we have:

$$\Gamma \vdash \mu a^\beta.\mathtt{c} : \beta; \Delta \quad \overset{\langle\,\rangle_u}{\longmapsto} \quad \vdash_{\mathtt{0}} \mathbf{0}\{u/a\} \triangleright \Gamma^\circ, \Delta^\bullet \quad \overset{[\,]_u^{\Gamma;\Delta}}{\longmapsto} \quad \mu a^\beta.\mathtt{c}$$

In the second transformation above, we map $[\mathbf{0}]_u^{\Gamma;\Delta\cdot u:\beta}$ such that $u \notin$ $\mathtt{dom}(\Delta)$ into $\mu u.[\mathbf{0}]_m^{\Gamma;\Delta\cdot u:\beta}$ to which the first rule in Figure 8 is applicable.

If $\mathtt{N}' \overset{\text{def}}{=} [a]\mathtt{U}$ with $a \in \mathsf{fn}(\mathtt{U})$ (for example $\mathtt{U}$ would be $\lambda x.\mathtt{N}''$), by definition we assume $a \in \mathsf{fn}(\mathtt{U})$. Then we have:

$$\begin{aligned}\mu a^{\beta}.[a]\mathtt{U}\!:\!\beta &\overset{\langle\rangle_u}{\longmapsto} \langle [a]\mathtt{U}\!:\!\bot\rangle\{u/a\}\\ &\overset{[]_u}{\longmapsto} \mu u^{\beta}.[\langle [a]\mathtt{U}\!:\!\bot\rangle\{u/a\}]_m \;\equiv_\alpha\; \mu a^{\beta}.[a]\mathtt{U}\!:\!\beta\end{aligned}$$

If $\mathtt{N}' \overset{\text{def}}{=} \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}'$ and $a \in \mathsf{fn}(\mathtt{U})$, we have, noting $a \in \mathsf{fn}(\mathtt{U})$,

$$\begin{aligned}(\mu a^{\beta}.\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N})\!:\!\beta &\overset{\langle\rangle_u}{\longmapsto} \langle(\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N})\!:\!\bot\rangle\{u/a\}\\ &\overset{[]_u}{\longmapsto} \mu u^{\beta}.[\langle(\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N})\!:\!\bot\rangle\{u/a\}]_m\\ &\equiv_\alpha (\mu a^{\beta}.\mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N})\!:\!\beta.\end{aligned}$$

The case when $\mathtt{N}' \overset{\text{def}}{=} \mathtt{let}\ \_ = y\mathtt{U}$ with $a \in \mathsf{fn}(\mathtt{U})$ is the same. ■

Let us say $\Gamma \vdash M\!:\!\beta\,;\Delta$ with $u \notin \mathtt{dom}(\Gamma)$ *defines* $\vdash P \triangleright \Gamma^{\circ}\cdot\Delta^{\bullet} \in \mathsf{NF}_e$ *at* u iff $[\![M\!:\!\beta]\!]_u \searrow^* P$. A $\lambda\mu$v-term is *closed* if it contains neither free names nor free variables. We can now establish the definability.

THEOREM 4.7. (definability) *Let* $\vdash P \triangleright \Gamma^{\circ}\cdot\Delta^{\bullet}\cdot u:\alpha^{\bullet} \in \mathsf{NF}_e$ *such that* $\bot \notin \mathtt{im}(\Gamma)$. *Then* $\Gamma\cdot x\!:\!\bot \vdash [P]_u\!:\!\alpha\ ;\ \Delta$ *defines* P. *Further if* $\Gamma = \Delta = \emptyset$ *and* $P \not\equiv \mathbf{0}$, *then there is a closed* $\lambda\mu v$*-term which defines* P.

PROOF: The first half is immediate from Proposition 4.2 and Lemma 4.6. The latter half is by inspecting by induction that the given condition implies all occurrences of control constants can be replaced by (bound) variables. ■

4.3. Full Abstraction

To prove full abstraction, our first task is to define a suitable observational congruence in the $\lambda\mu$v-calculus. There can be different notions of observational congruences for the calculus; here we choose a large, but consistent congruence. This equality is defined solely using the terms and dynamics of the calculus; yet, as we shall illustrate later, its construction comes from an analysis of $\lambda\mu$v-terms' behaviour through their encoding into π^{c}-processes and the process equivalence $\cong_\pi$. The analysis is useful since the notion of observation in pure $\lambda\mu$v-calculus may not be too obvious, while $\cong_\pi$ is based on a clear and simple idea of observables. Two further observations on the induced congruence:

– The congruence is closely related with (and possibly coincide with some of) the notions of equality over full controls, as studied by Laird [19, 20], Selinger [34] and others.

– If we extend $\lambda\mu$v with sums or non-trivial atomic types, and define the congruence based on the convergence to distinct normal forms of these types, then the resulting congruence restricted to the pure $\lambda\mu$v-calculus is precisely what we obtain by the present congruence,

DEFINITION 4.8. *$\equiv_\perp$ is the smallest typed congruence on $\lambda\mu v$-terms which includes:*

1. $\Gamma \vdash M \equiv_\perp N : \beta; \Delta$ when $M \equiv_\alpha N$.

2. $\Gamma \vdash M \equiv_\perp N : \beta; \Delta$ when $N \overset{def}{=} M\{y/x\}$ where $\Gamma(x) = \Gamma(y) = \perp$.

For example, we have, under the environment $x{:}\perp, y{:}\perp$:

$$x \equiv_\perp y$$

We also have:

$$\lambda x^\perp.\lambda y^\perp x \equiv_\perp \lambda x^\perp.\lambda y^\perp y$$

We can easily check that, in the encoding, $\equiv_\perp$-related terms are always mapped to an identical process.

CONVENTION 1. *Henceforth we always consider $\lambda\mu v$-terms and CNFs up to $\equiv_\perp$.*

We can now define observables, which is an infinite series of closed terms of the type $\perp \Rightarrow \perp \Rightarrow \perp$.

DEFINITION 4.9. Define $\{W_i\}_{i\in\omega}$ by the following induction.

$$\begin{aligned}
W_0 &\overset{\text{def}}{=} \lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.z \\
W_1 &\overset{\text{def}}{=} \lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.[w]\lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.z \\
W_2 &\overset{\text{def}}{=} \lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.[w]\lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.[w]\lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.z \\
&\vdots \\
W_{n+1} &\overset{\text{def}}{=} \lambda z^\perp.\mu u^{\perp\Rightarrow\perp}.[w]W_n \quad .
\end{aligned}$$

Let $\gamma = \perp \Rightarrow \perp \Rightarrow \perp$. We then define:

$$Obs \overset{\text{def}}{=} \{W_0\} \cup \{\mu\, w^\gamma.[w]W_{n+1},\ n \in \mathbb{N}\}$$

where we take terms up $\equiv_\perp$.

All terms in *Obs* are closed $\rightarrow_{\lambda\mu\mathrm{v}}$-normal forms of type γ (W_0 can also be written as $\mu w.[w]W_0$, but is treated separately since $\mu w.[w]W_0$ is not a normal form).

To illustrate the choice of *Obs*, we show below the π-calculus representation of W_0, $\mu\, w.[w]W_1$, $\mu\, w.[w]W_2$, $\ldots$ through $[\![\cdot]\!]_u$, which is in fact the origin of *Obs*.

DEFINITION 4.10. *Define $\{P_i\}_{i\in\omega}$ as follows (below we use the same names for bound names for simplicity).*

$$\begin{aligned}
P_0 &\overset{def}{=} \overline{w}(c)!c(u).\mathbf{0}\\
P_1 &\overset{def}{=} \overline{w}(c)!c(u).\overline{w}(c)!c(u).\mathbf{0}\\
P_2 &\overset{def}{=} \overline{w}(c)!c(u).\overline{w}(c)!c(u).\overline{w}(c)!c(u).\mathbf{0}\\
&\vdots\\
P_{n+1} &\overset{def}{=} \overline{w}(c)!c(u).P_n.
\end{aligned}$$

We set $Obs_\pi \overset{def}{=} \{\, \vdash_{\mathtt{O}} P_i \triangleright w : \gamma^\bullet \,\}_{i\in\omega}$, *taking processes modulo* $\equiv$.

Note each P_i only outputs at w (if ever) at any subsequent invocation, even though an output at any one of the bound names (u above) is well-typed. For example, $P_1' \overset{\mathrm{def}}{=} \overline{w}(c)!c(u).\overline{u}(c)!c(u).\mathbf{0}$ has type $w : \gamma^\bullet$ but differs from P_1 by outputting at the bound u when it is invoked the second time. One can check P_0 is the smallest (w.r.t. process size, i.e. number of constructors) non-trivial inhabitant of this type: in particular it is smaller than $[\![\lambda z^\perp.\lambda x^\perp.x]\!]_w \overset{\mathrm{def}}{=} P_1'$.

LEMMA 4.11. *Let* $\vdash_{\mathtt{I}} R \triangleright w : \overline{\gamma^\bullet} \rightarrow v : ()^{\mathbf{?}}$ *and* $R \in \mathsf{NF}_e$. *Then either* $R \equiv !w(c).\mathbf{0}$, $R \equiv !w(c).\bar{c}$ *or* $R \equiv !w(c).\overline{v}$.

PROOF: By $R \in \mathsf{NF}_e$ it should start from the input at w, hence it has shape $!w(c).R'$. By $R' \in \mathsf{NF}_e$ and by its typing it can only be either $\mathbf{0}$, an output at c (in which case there is no subsequent action), or an output at v (ditto). This concludes the proof. ■

Processes in Obs_π have uniform behaviours: indeed they are closed under $\cong_\pi$.

PROPOSITION 4.12.

1. $\vdash_{\mathtt{O}} P_i \cong_\pi P_j \triangleright w : \gamma^\bullet$ *for arbitrary i and j.*
2. *If* $\vdash_{\mathtt{O}} Q \triangleright w : \gamma^\bullet$, $Q \in \mathsf{NF}_e$ *and* $Q \cong_\pi P_i$ *then* $Q \in Obs_\pi$.

PROOF: For (1), it suffices to show $P_0 \cong_\pi P_1$ (cf. Example 3.12). By Proposition 2.11 (context lemma), we restrict the differentiating contexts for these processes to the shape of $(\boldsymbol{\nu}\, w)(R \mid [\,\cdot\,])$ such that $\vdash R \triangleright w : \overline{\gamma^\bullet} \to a : ()^?$. By Lemma 4.11 we only have to check three cases for R. No output is observed from either $C[P_0]$ or $C[P_1]$ when we take $R \stackrel{\text{def}}{=} !w(c).\mathbf{0}$, similarly when $R \stackrel{\text{def}}{=} !w(c).\overline{c}$, while if $R \stackrel{\text{def}}{=} !w(c).\overline{v}$ then both $C[P_0] \Downarrow_v$ and $C[P_1] \Downarrow_v$. For (2), we show any Q which is outside of Obs_π is observationally different from each P_i. Let each u_i and c_i be distinct names and $u_0 \stackrel{\text{def}}{=} w$. Then we generate the set $\mathbf{G}'_\pi$ of processes as follows.

1. $\mathbf{0} \in \mathbf{G}'_\pi$.

2. $\overline{u_i} \in \mathbf{G}'_\pi$.

3. If $S \in \mathbf{G}'_\pi$ then $\overline{w}(c_i)!c_i(u_i).S \in \mathbf{G}'_\pi$.

Here $u_0, u_i, \ldots$ are pairwise distinct and are different from w. Then we set $\mathbf{G}_\pi \stackrel{\text{def}}{=} \{S \mid S \in \mathbf{G}'_\pi,\ \vdash_{\mathsf{0}} S \triangleright w : \gamma^\bullet\}$. Note $Obs_\pi \subset \mathbf{G}_\pi$. Further, by typing, $\mathbf{G}_\pi$ exhausts all typable ENFs under $w : \gamma^\bullet$, because it can only start at w or if not it is 0; and if it starts from w it can only have the form $\overline{w}(c_i)!c_i(u_i).P$. Then P can only be 0, or if not it has an output at w or u_i, and if it has an output at u_i then it ends there by typing, if it has an output at w then we repeat the argument, hence done.

Now assume $Q \in \mathbf{G}_\pi \backslash Obs_\pi$. Then Q has at some height output at a name other than w. If it is the n-th input of Q, let:

$$R \stackrel{\text{def}}{=} !w(c).\overline{c}(u_1)!u_1.\overline{c}(u_2)....\overline{c}(u_n)!u_n(c).\overline{v}$$

and construct $C[\cdot] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, w)(R \mid [\cdot])$, which differentiates Q from each process in Obs_π, since $C[Q] \Downarrow_v$ while $C[P] \not\Downarrow_v$ for each $P \in Obs_\pi$ (as a concrete example, take $Q \stackrel{\text{def}}{=} \overline{w}(c)!c(u)\overline{u}(e)!e.\mathbf{0}$, then, setting $C[\,\cdot\,]$ to be the above mentioned context with $R \stackrel{\text{def}}{=} !w(c).\overline{c}(u)!u.\overline{a}$, we can observe $C[Q]$ outputs at a, but $C[P_i]$ does not for any i). ■

These observations motivate the following definition. Below $C[\,\cdot\,]^{\beta}_{\Gamma;\alpha;\Delta}$ is a typed context whose hole takes a term typed as $\alpha;\Delta$ under the base Γ and which returns a closed term of type β.

DEFINITION 4.13. *We write* $\Gamma \vdash M \cong_{\lambda\mu} N : \alpha\,;\Delta$ *when, for each typed context* $C[\;]^{\bot\Rightarrow\bot\Rightarrow\bot}_{\Gamma;\alpha;\Delta}$*, we have:*

$$\exists L.(C[M] \Downarrow L \in Obs) \quad \textit{iff} \quad \exists L'.(C[N] \Downarrow L' \in Obs).$$

Note that we treat all values in *Obs* as an identical observable. Immediately $\rightarrow_{\lambda\mu\mathsf{v}} \subset \cong_{\lambda\mu}$. The following result is crucial for full abstraction.

LEMMA 4.14. *Recall* $P_0 = \overline{w}(c)!c(u).\mathbf{0}$ *from Definition 4.10. Let* $\vdash L : \bot \Rightarrow \bot \Rightarrow \bot$ *be a normal form w.r.t.* $\lambda\mu_\nu$*-reduction such that* $\vdash_{\mathsf{O}} [\![L]\!]_w \cong_\pi P_0 \rhd w : \gamma^\bullet$. *Then* $L \in$ *Obs.*

PROOF: We generate the set $\mathbf{G}'$ of typed $\lambda\mu\mathsf{v}$-terms as follows. Below $\Gamma^\bot$ indicates the codomain of Γ is restricted to $\bot$, similarly for $\Delta^{\gamma, gamma'}$ where $\gamma' = \bot \Rightarrow \bot$.

1. $\Gamma^\bot, x : \bot \vdash x : \bot; \Delta^{\gamma,\gamma'} \in \mathbf{G}'$.
2. If $\Gamma^\bot, x : \bot \vdash M : \alpha; \Delta^{\gamma,\gamma'} \in \mathbf{G}'$, then $\Gamma \vdash \lambda x^\bot.M : \bot \Rightarrow \alpha; \Delta \in \mathbf{G}'$.
3. If $\Gamma^\bot \vdash M : \gamma; \Delta^{\gamma,\gamma'} \cdot a : \gamma \in \mathbf{G}'$, then $\Gamma \vdash [a]M : \bot; \Delta \in \mathbf{G}'$.
4. If $\Gamma^\bot \vdash M : \gamma'; \Delta^{\gamma,\gamma'} \cdot a : \gamma' \in \mathbf{G}'$, then $\Gamma \vdash [a]M : \bot; \Delta \in \mathbf{G}'$.
5. If $\Gamma^\bot \vdash M : \gamma; \Delta^{\gamma,\gamma'} \cdot a : \bot \Rightarrow \bot \in \mathbf{G}'$, then $\Gamma \vdash [a]M : \bot; \Delta \in \mathbf{G}'$.
6. If $\Gamma^\bot \vdash M : \bot; \Delta^{\gamma,\gamma'} \cdot a : \gamma' \in \mathbf{G}'$, and, moreover, it is not the case that M has form $[a]M'$ such that $a \notin \mathsf{fn}(M')$, then $\Gamma \vdash \mu a^{\gamma'}.M : \bot \Rightarrow \bot; \Delta \in \mathbf{G}'$.
7. If $\Gamma^\bot \vdash M : \bot; \Delta^{\gamma,\gamma'} \cdot a : \gamma \in \mathbf{G}'$, and, moreover, it is not the case that M has form $[a]M'$ such that $a \notin \mathsf{fn}(M')$, then $\Gamma \vdash \mu a^{\gamma}.M : \bot \Rightarrow \bot; \Delta \in \mathbf{G}'$.

Then we set

$$\mathbf{G} \stackrel{\text{def}}{=} \{\vdash M : \gamma \mid M \in \mathbf{G}'\}.$$

That is, $\mathbf{G}$ is the set of closed terms of type γ in $\mathbf{G}'$. By induction on the restricted syntactic shape of normal forms, we can show $\mathbf{G}$ enumerates all closed normal forms of type γ. Clearly all members of $\mathbf{G}$ are either of the form $\lambda x^\bot.M$ or $\mu w^\gamma[w]M$. We now show that $[\![\lambda x^\bot.M]\!]_w \not\cong_\pi P_0$. This is because

$$[\![\lambda x^\bot.M]\!]_w = \overline{w}(c)!c(z).\overline{z}(d).R \qquad \text{for some } R.$$

But P_0 does not have an output after doing the first input while $[\![\lambda x^\bot.M]\!]_w$ has. Hence there is a context $C[\cdot]$ such that

$$C[P_0] \not\Downarrow_a, \text{but} \qquad C[[\![\lambda x^\bot.M]\!]_w] \Downarrow_a .$$

for some fresh a. The remaining elements of $\mathbf{G}$ are of the form

$$\mu w^\gamma.[w]W_{n+1} \qquad (n \geq 0)$$

where the W_{n+1} are as in Definition 4.9. This means that the translation of the terms $\mu w^{\gamma}.[w]W_{n+1}$ gives the P_{n+1} from Definition 4.10. Now the result follows from Proposition 4.12.

■

We can now establish the full abstraction, following the standard routine. We start with the computational adequacy. Below and henceforth we write $M \Downarrow L$ when $M \to^{*}_{\lambda\mu\mathsf{v}} L \not\to_{\lambda\mu\mathsf{v}}$.

PROPOSITION 4.15. (computational adequacy) *Let* $M : \bot \Rightarrow \bot \Rightarrow \bot$ *be closed. Then* $\exists L.(M \Downarrow L \in \mathit{Obs})$ *iff* $\exists P.(\llbracket M \rrbracket_u \searrow^{*} P \in \mathit{Obs}_\pi)$.

PROOF: By our enumeration of normal forms of type γ as **G** in the previous proof, we know $M \Downarrow L$ implies $\llbracket L \rrbracket_u$ is a $\searrow$-normal form. Now assume $M \Downarrow L \in \mathit{Obs}$. Then by Proposition 3.18 and because $\llbracket L \rrbracket_u \in \mathsf{NF}_e$ we conclude $\llbracket M \rrbracket_u \searrow^{*} \llbracket L \rrbracket_u \in \mathit{Obs}_\pi$. On the other hand, suppose $\llbracket M \rrbracket_u \searrow^{*} P \in \mathit{Obs}_\pi$ and, by Corollary 3.19, $M \Downarrow L$. Then $\llbracket M \rrbracket_u \cong_\pi P \in \mathit{Obs}_\pi$ by Proposition 2.9 (1) while $\llbracket M \rrbracket_u \searrow^{*} \llbracket L \rrbracket_u$ by Proposition 3.18 and $\llbracket L \rrbracket \not\searrow$. Hence $L \in \mathit{Obs}$ by Lemma 4.14. ■

COROLLARY 4.16. (soundness) $\llbracket M \rrbracket_u \cong_\pi \llbracket N \rrbracket_u$ *implies* $M \cong_{\lambda\mu} N$.

PROOF: Assume $\llbracket M \rrbracket_u \cong_\pi \llbracket N \rrbracket_u$. We show, for each well-typed $C[\cdot]$, $\exists L.(C[M] \Downarrow L \in \mathit{Obs})$ iff $\exists L'.(C[M] \Downarrow L' \in \mathit{Obs})$. Let $C[\cdot]$ be well-typed. By assumption and congruency of $\cong_\pi$:

$$\llbracket C[M] \rrbracket_v \cong_\pi \llbracket C[N] \rrbracket_v \qquad (*)$$

Now we reason:

$$\begin{array}{lcll} C[M] \Downarrow L \in \mathit{Obs} & \Rightarrow & \llbracket C[M] \rrbracket_v \Downarrow \llbracket L \rrbracket_v \in \mathit{Obs}_\pi & \text{(Proposition 4.15)} \\ & \Rightarrow & \exists O.\llbracket C[N] \rrbracket_v \searrow^{*} O \in \mathit{Obs}_\pi & (*) \\ & \Rightarrow & C[N]_v \searrow^{*} L' \in \mathit{Obs} & \text{(Proposition 4.15)} \end{array} .$$

■

THEOREM 4.17. (full abstraction) *Let* $\Gamma \vdash M_i : \alpha; \Delta$ $(i = 1, 2)$. *Then* $M_1 \cong_{\lambda\mu} M_2$ *if and only if* $\llbracket M_1 \rrbracket_u \cong_\pi \llbracket M_2 \rrbracket_u$.

PROOF: Suppose

$$\emptyset \vdash M_1 \cong_{\lambda\mu} M_2 : \alpha; \emptyset \qquad \text{(18a)}$$

but

$$\vdash_{\mathsf{O}} \llbracket M_1 \rrbracket_u \not\cong_\pi \llbracket M_2 \rrbracket_u \triangleright u : \alpha^{\bullet}. \qquad \text{(18b)}$$

By (18b) and by Proposition 2.11 (context lemma for $\cong_\pi$), converting the observable $()^{?}$ to the convergence to Obs_π in $\gamma^{\bullet}$ with $\gamma = \bot \Rightarrow$

$\bot \Rightarrow \bot$ in the obvious way, there exists $\vdash_{\mathtt{I}} R \triangleright u : \overline{\alpha^\bullet}, v : \gamma^\bullet$ such that $R \in \mathsf{NF}_e$ and (say)

$$\exists P.\ (\boldsymbol{\nu}\, u)([\![M_1]\!]|R) \searrow^* P \in Obs_\pi \tag{18c}$$

and

$$\neg\exists P.\ (\boldsymbol{\nu}\, u)([\![M_2]\!]|R) \searrow^* P \in Obs_\pi. \tag{18d}$$

Since $R \in \mathsf{NF}_e$, we can safely set $R \overset{\text{def}}{=} !u(c).R'$. Now take

$$\vdash_{\mathtt{I}} !u(cv).R' \triangleright u : (\alpha \Rightarrow \gamma)^\bullet. \tag{18e}$$

By Theorem 4.7 (definability), we can find L such that $\vdash L : \alpha \Rightarrow \gamma$ where $[\![L]\!]_u \cong_\pi !u(cv).R'$. Since $[\![LM_i]\!]_u \searrow^+ (\boldsymbol{\nu}\, u)([\![M_i]\!]|R)$, we conclude

$$\exists L'.\ LM_1 \Downarrow L' \in Obs \ \wedge\ \neg\exists L'.\ LM_2 \Downarrow L' \in Obs, \tag{18f}$$

which contradicts (18a). Since precisely the same argument holds when Γ and Δ are possibly non-empty in $\Gamma \vdash M_{1,2} : \alpha; \Delta$ by closing them by λ/μ-abstractions, we have now established the full abstraction. ■

5. Discussion

5.1. Control and Name Passing (1)

This paper presents the typed π-calculus for full control, which arises as a typed π-calculus, i.e. as a subcalculus of the linear π-calculus [39], namely one that only uses replicated inputs. The connection between control and the π-calculus is first pointed out by Thielecke in his thesis [36], where he has shown that the target of CPS-transform can be written down as name passing processes. The main contribution of the present work in this context is the use of a duality-based type structure in the π-calculus, by which the embedding of control constructs in processes becomes semantically exact. Hoping the present work can serve as a starting point of a fruitful dialogue between the studies on control operators and those on theories of typed processes, this section concludes the paper with discussions on related work and further topics.

The notion of full control arises in several related contexts. Historical survey of studies of controls and continuations can be found in [30, 37]. Here we pick up three strands of research to position the present work in a historical context. In one strand, notions of control operators have been formulated and studied as a way to represent jumps and other non-trivial manipulation of control flows as an extension of the λ-calculus and related languages. Among many works, Felleisen and others [10, 11] studied syntactic and equational properties of control operators in the context of the call-by-value λ-calculus, clarifying their status. Griffin [13] shows a correspondence between the λ-calculus with control operators, classical proofs and the CPS transform. Finally Parigot [28] introduced the $\lambda\mu$-calculus, the calculus without control operators but which manipulates names, as term-representation of classical natural deduction proofs. The control-operator-based presentation and name-based presentation, which are shown to be equivalent by de Groote [9], elucidate statics and dynamics of full control in different ways: the latter gives a more fine-grained picture while the former often offers a more condensed representation. In this context, the present work shows a further decomposition (and arguably simpler presentation) of the dynamics of full control on the uniform basis of name passing interaction.

5.2. Control and Name Passing (2)

Another closely related context is the CPS transform [8, 12, 31]. In this line of studies, the main idea is to represent the dynamics of the λ-calculus, or procedural calls, in a way close to implementations. Consider for example the following reduction:

$$(\lambda x.x)1 \longrightarrow_\beta 1$$

To model implemented execution of this reduction, we elaborate each term with a continuation to which the resulting value should be returned. We write this transformation $\langle\!\langle M\rangle\!\rangle$. In the above example, $\langle\!\langle \lambda x.x\rangle\!\rangle \overset{\text{def}}{=} \lambda h.h(\lambda x.\langle\!\langle x\rangle\!\rangle)$ (which receives a next continuation and "sends out" its resulting value to that continuation, with $\langle\!\langle x\rangle\!\rangle = \lambda k.kx$); whereas $\langle\!\langle 1\rangle\!\rangle \overset{\text{def}}{=} \lambda h'.h'1$. The term $(\lambda x.x)1$ as a whole is transformed as follows:

$$\lambda k.(\lambda h.h\lambda x.\langle\!\langle x\rangle\!\rangle)(\lambda m.\langle\!\langle 1\rangle\!\rangle(\lambda n.mnk)) \tag{19a}$$

This transformation may need some illustration. Assume first we apply to the above abstraction the ultimate continuation k (to which the result of evaluating the whole term should jump), marking the start of computation. Write M for $\lambda x.x$ and N for 1. After the continuation k is fed to the left-hand side, we first give $\langle\!\langle M\rangle\!\rangle$ its next continuation $(\lambda m.\langle\!\langle N\rangle\!\rangle(\lambda n.mnk))$, to which the result of evaluating M, say V, is fed, replacing m, then we send $\langle\!\langle N\rangle\!\rangle$ its continuation $\lambda n.Vnk$, to which the result of evaluating $\langle\!\langle N\rangle\!\rangle$ is fed, replacing n, so that finally the "real" computation VW can be performed, to whose result the ultimate continuation k is applied.

As may be seen from the example above, the CPS transform can be seen as a way to mimic the operational idea of "jumping to an address with a value" solely using function application and abstraction. This representation is useful to connect the procedural calls in high-level languages to their representation at an execution level (the textbook by Appel [2] gives a lucid account on this topic). The representation is somewhat shy about the use of "names" by abstracting them immediately after their introduction, partly because this is the only way to use the notion in the world of pure functions (note in $(\lambda h.hM)V$, the bound h in fact names V). This however does not prevent us from observing (19a) is isomorphic to its standard process encoding via $[\![\cdot]\!]$ of Section 3, given as follows.

$$(\boldsymbol{\nu}\, h)([\![\lambda x.x]\!]_h \,|\, !h(n).(\boldsymbol{\nu}\, h')([\![1]\!]_{h'} \,|\, !h'(m).\overline{n}\langle mk\rangle)) \qquad (19b)$$

In (19b), k, h, h' are all channel names at which processes interact: the input/output polarities make it clear what is named (used as replicated inputs) and to which it is jumping (used as outputs, i.e. subscripts of the encoding). The "book-keeping" abstractions of h and h' in (19a) are replaced by hiding. Setting $[\![1]\!]_{h'}$ to be $\overline{h'}\langle 1\rangle$ (regarding 1 as a specific name), we can see how (19b) reduces precisely as (19a) reduces modulo the book-keeping reductions. Sangiorgi [33] observed that we can regard (19a) as terms in the applicative part of the higher-order π-calculus (a variant of π-calculus which processes communicate not only names but also terms) and that the translation from a λ-term to its process representation can be factored into the former's CPS transformation and its encoding into the π-calculus.

In the context of these studies, where the control is studied purely in the context of the standard λ-calculi, the main contribution of the present work may lie in identifying the precise realm of typed processes which, when it is used for the encoding of λ-terms, gives exactly the same equational effect as the standard CPS transform embedded in the λ-calculus. As we have shown in [4, 39], the encoding of the λ-calculi into the linear/affine-π-calculi [4] results in full abstraction. π^{c} offers a refined understanding on CPS-transform, with precisely the same induced equivalence. As related points, we have suggested possible relationship between existing CPS transformations/inversions [8, 12, 31], on the one hand, and the encoding/decoding in Sections 3 and 4 in this paper on the other.

5.3. Control and Name Passing (3)

There are many studies of semantics and equalities in calculi with full control, notably those which aim to investigate appropriate algebraic structures of suitable categories (for example those by Thielecke [36], Laird [21] and Selinger [34]). The present work may have two interests in this context.

First, the basis of the observational equivalence for $\lambda\mu$v-terms, the behavioural equivalence over π^{c}-processes, has very simple operational content, while inducing the equality closely related with those studied in the past. Among others we believe that $\cong_{\lambda\mu}$ coincides with the call-by-value, total and extensional version of Laird's games for control [16, 20] (it is easy to check all terms in *Obs* are equated in such a universe). We also suspect it is very close to the equality induced by the call-by-value part of Selinger's dualised universe [34] (for the same reason), though details are to be checked. The combination of clear observational scenario and correspondence with good denotational universes is one of the notable aspects of the use of the π-calculus.

Second and relatedly, as a tool for representation, typed π-calculi such as π^{c} may have a distinct merit which may complement other tools, such as categorical and logical. The process representation elucidates behaviours of programs with control, as shown in Sections 3.4 and 3.5. Here two aspects of typed processes are of paramount importance: In one view, π^{c} as a syntax offers a direct way to understand the observational/compositional behaviour of a program with control. We have already discussed this aspect in Section 6.1, with many examples and formal results attesting the use of π^{c} as a syntactic tool.

In another and related view, we may consider processes in π^{c} as name passing transition systems (or name passing synchronisation trees). As such, a process identifies meaning of a denoted program as an abstract entity. The rich repertoire of powerful reasoning techniques developed for π-calculi is now freely available; further this representation has enriching connection with studies on game-based semantics, most notably games for control studied by Laird [21]. Indeed, Laird's work may be regarded as a characterisation of dynamic interaction structure of π^{c} (or, to be precise, its affine extensions), where the lack of well-bracketing corresponds to the coalescing of linear actions into replicated actions. Another intensional structure in close connection is *Abstract Böhm Trees* studied by Curien and Herbelin [5, 6]. We expect the variant of these structures for full control to have a close connection to name passing transition of π^{c}.

It is also notable that representation of programs and other algorithmic entities as name passing transition, together with basic operators such as parallel composition, hiding and prefixing, is not limited to the control nor to sequential computation.

In summary, the interest in the use of π^{c} in the present enquiry is in the powerful analytical apparatus name passing processes offer for understanding contextual behaviour of processes, especially when combined with a strong notion of types, in the setting when the notion of observables itself is not clear in the source calculus. It is an interesting subject of study how the similar enterprise may be applicable in different kinds of typed calculi, for example those programming languages which include both purely functional behaviours and imperative features, as originally studied by Felleisen and others [10]. The use of typed name passing processes for analysing control may allow us to position various findings on syntax and semantics of diverse notions of control in a broad universe of typed name passing processes. This leads to further development and applications of these findings in both theoretical and practical settings.

5.4. Control as Proofs and Control as Processes

The present work has a close connection with recent studies on control from a proof-theoretic viewpoint, notably Polarised Linear Logic by Laurent [22, 23] and $\overline{\lambda}\mu\tilde{\mu}$-calculus by Curien and Herbelin [7]. The type structures for the linear/affine π-calculi are based on duality, here arising in a simplest possible way, as mutually dual input and output modes of channel types. This duality has a direct applicability for analysis of processes and programs, as may be seen in the new flow analysis we have recently developed for typed π-calculi [17]. This duality allows a clean decomposition of behaviours in programming languages into name passing interaction, and is in close correspondence with polarity in Polarised Linear Logic by Laurent [22, 23]. Laurent and the first author recently obtained a basic result on the relationship between π^{c} and Polarised Linear Logic, which will be discussed elsewhere.

In a different context, Curien and Herbelin [7] presents $\overline{\lambda}\mu\tilde{\mu}$, a calculus for control, based on Gentzen's LK, in which a strong notion of duality elucidates the distinction between the call-by-name and call-by-value evaluations in the setting of full control. The formal relationship between their calculus on the one hand and π^{c} on the other is currently under study. One interesting aspect is the way nondeterminism arises in their calculus, which suggests an intriguing connection between the dynamics of their calculus and name passing processes. From the same viewpoint, the connection with a recent work by Wadler [38] on duality and λ-calculi is an interesting subject for further studies.

The present study concentrates on the call-by-value encoding of the $\lambda\mu$-calculus. As in the λ-calculus [4], we can similarly embed the call-by-name $\lambda\mu$-calculus into π^{c} by changing the encoding of types (hence terms). The following is the standard Hyland-Ong encoding of call-by-name types [4] assuming the only atomic type is $\bot$.

$$[\alpha_1, ..., \alpha_n, \bot]^\circ \stackrel{\text{def}}{=} (\overline{\alpha_1^\circ}, ..., \overline{\alpha_n^\circ})^{\mathbf{!}}$$

In the presence of control, we can simply augment this map with:

$$\alpha^\bullet \stackrel{\text{def}}{=} (\alpha^\circ)^{\mathbf{?}}$$

which says: "a program may jump to a continuation" (this corresponds to the "player first" in Laurent's games [22]). This determines, together with the one given in [4], the encoding of programs. We strongly believe the embedding is fully abstract, though details are to be checked.

There are a few studies (for example [34]) on conjunction and disjunction in the $\lambda\mu$-calculus. By moving to classical logics, not only negation but also these connectives (especially disjunction) bear a new significance. The encoding can be extended to these connectives keeping the syntax of π^{c} as in Section 2 (i.e. without introducing branching and selections constructs [39]). One interesting observation is that a natural encoding of the disjunction type gives rise to an encoding of terms in processes which is directly based on the standard idiom for representing the choice in unary communication.

Appendix

A. Proofs

A.1. Proof of Proposition 2.9

(1) is by precisely the same argument as in [39], establishing the convertibility from $\searrow$ coincides with the untyped weak bisimilarity $\approx$ and that $\approx$ is a congruence which respects the observability condition in Definition 2.8, hence is a subcongruence of $\cong_\pi$. For (2), let $\cong'_\pi$ be a typed congruence which is strictly greater than $\cong_\pi$. By the defining condition of $\cong_\pi$, we have $P_1 \Downarrow_x$ but $P_2 \not\Downarrow_x$ for some $P_1 \cong'_\pi P_2$. This implies $Q \cong'_\pi \mathbf{0}$ for an arbitrary typed process Q of $\mathtt{o}$ [because: take a typed context $C_i[\ \cdot\ ] \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, x)(!x.[\cdot] \,|\, P_i)$ with $i = 1, 2$, then $Q \cong'_\pi C_2[Q] \cong'_\pi C_1[Q] \cong'_\pi \mathbf{0}$]. This also means an arbitrary input process $!x(\vec{y}).Q$ is equated with $!x(\vec{y}).\mathbf{0}$. Let $\vdash_\phi P_{1,2} \triangleright A$ which we safely assume to be without $\searrow$-redexes by (1) above, nor output processes by the above argument. Since input processes in P_1 and P_2 are precisely paired by their subjects which are equated by $\cong'_\pi$, we conclude $P_1 \cong'_\pi P_2$. These arguments also establish (3).

A.2. Proof of Proposition 2.11 (context lemma)

The "only if" direction is immediate by the defining condition of $\cong_\pi$. For the "if" direction, assume $\vdash_{\mathsf{O}} P_{1,2} \triangleright A$ and $P_1 \not\cong_\pi P_2$. Then for some typed context $C[\ \cdot\ ]$ of type $x : ()^{?}$, we have (say) $C[P_1] \Downarrow_x$ but *not* $C[P_2] \Downarrow_x$. Note this means P_1 contributes to the output at x, since if not we should also have had $C[P_2] \Downarrow_x$. This is only possible if P_1 (or its residual) is not under a prefix. Thus we know:

$$C[P_1] \ \longrightarrow^* \ C'_r[P_1] \ \longrightarrow^* \ \overline{x}|S$$

where $C'_r[\cdot]$ is of the form $(\boldsymbol{\nu}\, \mathsf{fn}(A)\vec{z})([\cdot] \,|\, R)$. Note we have

$$C[P_2] \ \longrightarrow^* \ C'_r[P_2] \not\Downarrow_x .$$

Thus we know $P_1|(\boldsymbol{\nu}\, \vec{z})R \Downarrow_x$ while $P_2|(\boldsymbol{\nu}\, \vec{z})R \not\Downarrow_x$, as required.

A.3. The remaining cases for Proposition 3.18

We prove the cases for (ζ_{arg}) rules in [27]. The shape of these rules (in particular why we have values on the right-hand side) is clearly explained by the encoding. In the following proof we often omit principal ports of $\perp$-typed terms, observing $u \notin [\![M : \perp]\!]_u$ for each $\Gamma \vdash M : \perp\,; \Delta$. Throughout we assume newly introduced names are fresh. Let $\alpha \neq \perp$. First let $\beta \neq \perp$ and $[\![V : \alpha \Rightarrow \beta]\!]_m \stackrel{\mathrm{def}}{=} \overline{m}(c)!c(e'u').R$ and $R' \stackrel{\mathrm{def}}{=} R\{eu/e'u'\}$. The reason why $V(\mu a^\alpha.M)$ should be a redex, comes from the following initial reduction from $[\![V(\mu a^\alpha.M)]\!]_u$. Below we use $m \notin \mathsf{fn}(R)$ since V is a value.

$$\begin{array}{ll}
\multicolumn{2}{l}{[\![V : \alpha \Rightarrow \beta]\!]_m\{m(c) = ([\![\mu a.M]\!]_n\{n(e) = \overline{c}\langle eu\rangle\})\}} \\
\stackrel{\mathrm{def}}{=} & (\nu m)([\![V]\!]_m \mid !m(c).([\![\mu a.M]\!]\{n(e) = \overline{c}\langle eu\rangle\})) \\
\searrow^{+} & (\boldsymbol{\nu}\, c)(!c(e'u').R \mid (\boldsymbol{\nu}\, a)([\![\mu a.M]\!]_n\{n(e) = \overline{c}\langle eu\rangle\})) \\
\stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\, a)([\![M]\!]_a \mid !a(e).\overline{c}\langle eu\rangle)\{c(e'u') = R\}
\end{array}$$

Note we moved the argument to the "body" of the configuration, placing R in the "environment" part of the whole configuration. In retrospect, the reason why $V(\mu a^\alpha.M)$ is regarded as a redex is because, when the left-hand side is a value, this transformation is possible. We now give the formal simulation, writing ${}^{+}\!\swarrow$ for the inverse of $\searrow^{+}$.

$$\begin{array}{llll}
(\boldsymbol{\zeta_{\mathrm{arg}}}) & [\![V(\mu a^\alpha.M)]\!]_u & \searrow^{+} & [\![M : \perp]\!]\{a(e) = R'\} \\
 & & \searrow^{+}\!\swarrow & C'[[\![L_i]\!]_a\{a(e) = R'\})]_i \\
 & & {}^{+}\!\swarrow & [\![C[[b](VL_i)]_i : \perp]\!]\{u/b\} \\
 & & \stackrel{\mathrm{def}}{=} & [\![\mu b.C[[b](VL_i)]]\!]_u .
\end{array}$$

This derivation is very similar to that of ζ_{fun} on Page 30 in that it uses an appropriate context $C'[[\![L_i]\!]_a]_i$ that enumerates all (possibly nested) occurrences of $[a]\cdot$ and then applies Proposition 3.16 (12h). Finally with $[\![V:\alpha\Rightarrow\bot]\!]_m \stackrel{\mathrm{def}}{=} \overline{m}(c)!c(e').R$ and $R' \stackrel{\mathrm{def}}{=} R\{e/e'\}$,

$$\begin{array}{llll} (\zeta_{\mathbf{arg},\bot}) & [\![V(\mu a^{\alpha}.M)]\!] & \searrow\overset{+}{}\swarrow & C'[[\![L_i]\!]_a\{a(e)=R'\})]_i \\ & & \overset{+}{\swarrow} & [\![C[(VL_i)]_i:\bot]\!]. \end{array}$$

B. Auxiliary Definitions

Below we define the $\lambda\mu$-substitution $M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}$, assuming the bound name convention.

$$\begin{array}{rcl} x\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\} & \stackrel{\mathrm{def}}{=} & x \\ (\lambda x.M)\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\} & \stackrel{\mathrm{def}}{=} & \lambda x.(M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}) \\ MN\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\} & \stackrel{\mathrm{def}}{=} & (M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\})(N\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}) \\ (\mu b^{\alpha}.M)\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\} & \stackrel{\mathrm{def}}{=} & \mu b^{\alpha}.(M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}) \\ ([a']M)\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\} & \stackrel{\mathrm{def}}{=} & \begin{cases} [a'](M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}) & (a\neq a') \\ [a'](C[M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}]) & (a=a') \end{cases} \end{array}$$

Observe the substitution is applied in a nested fashion in the last line.

C. A Brief Comparison between $\pi^{\mathsf{C}}$and π^{L}

The typing system presented in this paper is derived from, and very similar to that of [39]'s linear π-calculus π^{L}. We shall now briefly describe the differences between the two systems. For this purpose it may be instructive two consider two servers:

$$\vdash_{\mathtt{I}} !x(v_1...v_m w).P \rhd x:(\tau_1^{?}...\tau_m^{?}\tau^{\uparrow})^{!}$$

typable in π^{L}, and a second one

$$\vdash_{\mathtt{I}} !x(u_1...u_n).Q \rhd x:(\tau_1^{?}...\tau_n^{?})^{!}$$

which would be typed in π^{C}. Here $\uparrow$ indicates a *linear* output. For simplicity we assume that x is the only free name in both cases. Whenever we invoke the linear server $!x(v_1...v_m w).P$, we pass $m+1$ channels, the first m of which are used by P as "arguments" in the standard sense. They can be used for recursively querying other processes – which are servers again – about the data. The last, $m + 1$-th name is the *unique* return channel, used linearly. The typing system in π^{L} guarantees that P will eventually return its result on this channel and will do so exactly once. Hence the invoker can force the server to return the result on a given channel. The call-return sequences in π^{L} are highly structured: if P invokes another server, that other server will return to P first, and only then can P return to its caller. In other words, π^{L} enforces a stack-like calling discipline.

This is quite different in π^{C}. The invoker passes n arguments and all that the typing system guarantees is that processes will terminate. However, we have no distinction between "returning" and "invoking an argument", these two concepts are fused in π^{C}: the arguments passed to the server are points where the computation may continue; the invoked server P will invoke at one of its arguments, or it will invoke many of them, or none at all (but it will not diverge). This is the key difference: π^{L} guarantees a unique return channel, while π^{C} only allows the invoker to supply a set of possible argument channels where the computation proceeds.

References

1. Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full Abstraction for PCF. *Info. & Comp.*, 163:409–470, 2000.
2. Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.
3. Henk Barendregt. *The Lambda Calculus.* North Holland, 1985.
4. Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, 2001.
5. Pierre-Louis Curien. Abstract Böhm Tree. *Mathematical Structures in Computer Science*, 8(6):559–591, 1998.
6. Pierre-Louis Curien and Hugo Herbelin. Computing with Abstract Böhm Tree. pages 20–39. World Scientific, 1998.
7. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proc. ICFP*, pages 233–243, 2000.
8. Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *MSCS*, 2(4):361–391, 1992.
9. Philippe de Groote. On the Relation between the lambda-mu-calculus and the Syntactic Theory of Sequential Control. In *Proc. LPAR*, pages 31–43, 1994.
10. Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. Syntactic theories of sequential control. *TCS*, 52:205–237, 1987.
11. Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2):235–271, 1992.
12. Carsten Führmann and Hayo Thielecke. On the call-by-value CPS transform and its semantics. *Inf. Comput.*, 188(2):241–283, 2004.
13. Timothy G. Griffin. A formulae-as-type notion of control. In *Proc. POPL*, pages 47–58, 1990.

14. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, number 512 in LNCS, pages 133–147, 1991.
15. Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *TCS*, 151:393–456, 1995.
16. Kohei Honda and Nobuko Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
17. Kohei Honda and Nobuko Yoshida. Noninterference through flow analysis. *Journal of Functional Programming*, To appear.
18. J. Martin E. Hyland and C. H. Luke Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
19. James Laird. Full abstraction for functional languages with control. In *Proc. LICS*. IEEE, 1997.
20. James Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.
21. James Laird. A game semantics of linearly used continuations. In *Proc. FOSSACS*, number 2620 in LNCS, pages 313–327, 2003.
22. Olivier Laurent. Polarized games. In *Proc. LICS*, pages 265–274, 2002.
23. Olivier Laurent. Polarized proof-nets and $\lambda\mu$-calculus. *TCS*, 290(1):161–188, 2003.
24. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
25. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
26. Eugenio Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
27. C.-H. Luke Ong and Charles A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proc. POPL*, pages 215–227, 1997.
28. Michel Parigot. $\lambda - \mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *Proc. LPAR*, pages 190–201, 1992.
29. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
30. John Reynolds. The discovers of continuations. *Lisp and Symbolic Computation*, 6:233–247, 1993.
31. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LFP*, pages 288–298, 1992.
32. Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *TCS*, 167(2):235–274, 1996.
33. Davide Sangiorgi. From λ to π: or, rediscovering continuations. *MSCS*, 9:367–401., 1999.
34. Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *MSCS*, 11(2):207–260, 2001.
35. Charles Stewart. *On the formulae-as-types correspondence on classical logic.* PhD thesis, Oxford University, 1999.
36. H. Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, 1997.
37. Hayo Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, 1999.
38. Philip Wadler. Call-by-value is dual to call-by-name. In *Proc. ICFP*, pages 189–201, 2003.
39. Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the π-calculus. *Inf. Comput.*, 191(2):145–202, 2004.