Process Types as a Descriptive Tool for Interaction[⋆]
Control and the Pi-Calculus

Kohei Honda[1⋆⋆], Nobuko Yoshida[2], and Martin Berger[3]

[1] Queen Mary, University of London
[2] Imperial College London
[3] University of Sussex

Abstract. We demonstrate a tight relationship between linearly typed π-calculi and typed λ-calculi by giving a type-preserving translation from the call-by-value $\lambda\mu$-calculus into a typed π-calculus. The $\lambda\mu$-calculus has a particularly simple representation as typed mobile processes. The target calculus is a simple variant of the linear π-calculus. We establish full abstraction up to maximally consistent observational congruences in source and target calculi using techniques from games semantics and process calculi.

1 Introduction

At TLCA 2001 [4] the authors started a research programme relating typed λ-calculi and typed π-calculi [5, 17, 18, 39, 40]. The rationale behind the programme has been twofold: first to demonstrate that functional computation can be decomposed into name-passing and thus be fruitfully understood as a constrained and well-behaved form of interaction. Secondly, the authors wanted their nascent investigations of the vast space of typing systems for interacting processes be guided by the λ-calculus community's insights into the nature of typing, one of the great contributions to computer science. A key aspect of our methodology for generalising λ-calculus types to interaction has been to study full-abstraction: Milner's encoding of untyped λ-calculus [24] is sound but not complete in the absence of types. This is because atomic β-reduction steps are decomposed into multiple name-passing interactions where intermediate steps can be observed. To achieve completeness, the ability to observe such intermediate steps must be ruled out by typing – in other words, only processes that interact 'functionally' should be well-typed. A key insight has been the usefulness of *linearly* typed π-calculi for the understanding of typed λ-calculi. Linearity and it's close cousin *affinity* yield full abstraction using proof techniques coming from games semantics, linear logic and process calculi. This paper continues our programme by studying the call-by-value $\lambda\mu$-calculus [26], henceforth $\lambda\mu$v-calculus, a λ-calculus with non-local control-flow manipulation (often referred to as "control" or "jumping"). The $\lambda\mu$v-calculus is a proof-calculus for classical propositional logic. Non-local control can be encoded in pure λ-calculus using continuation-passing style, but a direct representation of jumping is of interest, not just because of the connection with classical logic, and because it leads to more readable

⋆ Partially supported by EPSRC EP/K011715/1, EP/L00058X/1 and EP/K034413/1.
⋆⋆ http://mrg.doc.ic.ac.uk/kohei.html [29, 33]

programs in comparison with continuation-passing style, but also because jumping can be seen as a form of interaction that is less restrictive than the last-in/first-out calling discipline of functional computation.

Technically, we present a type-preserving translation from the $\lambda\mu$v-calculus into a typed π-calculus, and show that it is fully abstract up to maximally consistent observational congruences in source and target 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 [26].

There are different notions of control. In the π-calculus they can be represented as distinct forms of typed interaction. The $\lambda\mu$-calculus introduced by Parigot [27] provides unrestricted control, and was later studied in call-by-value form by Ong and Stewart [26]. Surprisingly, as this paper demonstrates, unrestricted control 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, for example, the simply typed λ-calculus full abstractly. We call the subset of the linear π-calculus corresponding to full control the π^{c}-calculus ("c" indicates control). The π^{c}-calculus is restricted in that each channel is used only for a unique stateless replicated input and for zero or more dual outputs. The π^{c}-calculus also precludes circular dependency between channels. In spite of its simplicity, both, call-by-value and call-by-name full control, are precisely embeddable into the π^{c}-calculus by changing the translation of types. Because the π^{c}-calculus is a proper subset of the linear π-calculus, many known results about the linear π-calculus 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 [24]. The tight operational and equational correspondence enables the use of typed π-calculi for investigating and analysing 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.

Section 2 summarises the π^{c}-calculus. Section 3 gives our encoding of the $\lambda\mu$v-calculus. Section 4 establishes full abstraction. The paper concludes with an outline of the call-by-name encoding, and discusses related work as well as open issues. Full proofs and additional discussions are delegated to [1].

2 Processes and Types

Processes. 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 [31] and has

(Structural Rules)

(S0) $P \equiv Q \quad \text{if } P \equiv_\alpha Q$

(S1) $P|\mathbf{0} \equiv P$ (S2) $P|Q \equiv Q|P$

(S3) $P|(Q|R) \equiv (P|Q)|R$

(S4) $(\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0}$

(S5) $(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P$

(S6) $(\boldsymbol{\nu}\, x)(P|Q) \equiv ((\boldsymbol{\nu}\, x)P)|Q \quad (x \notin \mathsf{fn}(Q))$

(S7) $\overline{x}(\vec{y})\overline{z}(\vec{w})P \equiv \overline{z}(\vec{w})\overline{x}(\vec{y})P \quad (x, z \notin \{\vec{w}\vec{y}\})$

(S8) $(\boldsymbol{\nu}\, z)\overline{x}(\vec{y})P \equiv \overline{x}(\vec{y})(\boldsymbol{\nu}\, z)P \quad (z \notin \{x\vec{y}\})$

(S9) $\overline{x}(\vec{y})(P|Q) \equiv (\overline{x}(\vec{y})P)|Q \quad (\{\vec{y}\} \cap \mathsf{fn}(Q) = \emptyset)$

(Reduction)

(Com$_!$)
$! \, x(\vec{y}).P \mid \overline{x}(\vec{y})Q \longrightarrow ! \, x(\vec{y}).P | (\boldsymbol{\nu}\, \vec{y})(P|Q)$

(Res)
$P \longrightarrow Q \implies (\boldsymbol{\nu}\, x)P \longrightarrow (\boldsymbol{\nu}\, x)Q$

(Par)
$P \longrightarrow P' \implies P|Q \longrightarrow P'|Q$

(Out)
$P \longrightarrow Q \implies \overline{x}(\vec{y})P \longrightarrow \overline{x}(\vec{y})Q$

(Cong)
$P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q$

Fig. 1. Reduction and Structural Rules.

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. The grammar of the calculus is given 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 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. $|$ is parallel composition, $! \, x(\vec{y}).P$ is replicated input, and $\overline{x}(\vec{y})P$ is asynchronous bound output, $(\boldsymbol{\nu}\, x)P$ is name hiding and $\mathbf{0}$ denotes nil. The full definition of the reduction rules and the structure rules is found in Figure 1 ($\longrightarrow$ is generated from the given rules; $\equiv$ is generated from the given rules together with the closure under all contexts). We write $\twoheadrightarrow$ for $\longrightarrow^* \cup \equiv$.

Types. First we introduce *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. 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.* The *dual of* τ, written $\overline{\tau}$, is defined as the result of dualising all modes in τ_i. For example, $(\overline{\tau_1}\, \overline{\tau_2})^{\mathbf{?}}$ 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: $\tau \odot \overline{\tau} = \overline{\tau} \odot \tau = \overline{\tau}$ and $\tau \odot \tau = \tau$ with $(\mathsf{md}(\tau) = \mathbf{?})$. 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. To guarantee the second condition, we introduce an *action type* ranged over by $A, B, C....$ The syntax is given as follows:

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

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.

Typing. The typing judgement are of the form $\vdash_\phi P \triangleright A$ which is read: *P has type A with mode ϕ* where *IO-modes*, $\phi \in \{\mathrm{I}, \mathrm{O}\}$, guarantees a restriction to a single thread. We present the typing system in Appendix A. 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}. 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 2.1. (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, taken modulo $\equiv$, that includes:

$$C[\overline{x}(\vec{y})P] | !x(\vec{y}).Q \searrow_r C[(\boldsymbol{\nu}\, \vec{y})(P|Q)] \mid !x(\vec{y}).Q \qquad (\boldsymbol{\nu}\, x)!x(\vec{y}).Q \searrow_g \mathbf{0}$$

where $C[\,\cdot\,]$ is an arbitrary (typed) context. We can immediately see that $\longrightarrow \subset \searrow$. Note $\searrow$ calculates *under* prefixes, which is unusual in process calculi. 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 2.2. *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.*

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 artefact of our restrictive typing discipline and does not hold in the untyped calculus. The notation was chosen to emphasise that Q in $P \searrow Q$ is 'smaller' or more reduced than P in a sense that can be made precise.

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 2.2 (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 2.1. *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; and (5) If $P \in \mathsf{NF}_e$ and $P \equiv Q$ then $Q \in \mathsf{NF}_e$. Clearly if P is typable and $P \not\searrow$ then $P \in \mathsf{NF}_e$.*

Contextual Congruence for π^{c}. The Church-Rosser property of typed processes, as stated in Proposition 2.2, 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:

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

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 2.2. $\cong_\pi$ is the maximum typed congruence satisfying: if $\vdash_{\mathrm{O}} P \cong_\pi Q \triangleright x : ()^{?}$, then $P \Downarrow_x$ iff $Q \Downarrow_x$.

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

Proposition 2.3. (1) $\searrow \subset \cong_\pi$; (2) $\cong_\pi$ *is a maximally consistent typed congruence;* (3) $\cong_\pi$ *is the unique maximally consistent congruence containing* $\searrow$.

Our choice of observable in π^{c}corresponds to the usual output-barbed congruence one considers in the untyped calculus. It is also *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 2.3 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.

3 Encoding

Call-by-value $\lambda\mu$-calculus. This section presents a type-preserving embedding of the call-by-value $\lambda\mu$-calculus by Ong and Stewart [26] in π^{c}. Apart from tractable syntactic properties of the calculus in comparison with its call-by-name counterpart, [26] showed how various control primitives of call-by-value languages (such as call/cc in ML) can be encoded in this calculus and its extension with recursion [26]. 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.

Types $(\alpha, \beta, \ldots)$ are those of simply typed λ-calculus with the atomic type $\bot$ (we can add other atomic types with appropriate values and operations on them). We use *variables* $(x, y, \ldots)$ and *control variables* (or *names*) $(a, b, \ldots)$. *Preterms* $(M, N, \ldots)$ and *values* $(V, W, \ldots)$ are generated from the grammar:

$$M, N ::= x \mid \lambda x^{\alpha}.N \mid MN \mid \mu a^{\alpha}.M \mid [a]M \qquad V, W ::= x \mid \lambda x^{\alpha}.N$$

Apart from 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 below:

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

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

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

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

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

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*. The reduction rules for the $\lambda\mu$v-calculus is given next:

(β_v) $(\lambda x.M)V \longrightarrow M\{V/x\}$

(ζ_{arg}) $V^{\alpha\Rightarrow\beta}(\mu a^{\alpha}.M) \longrightarrow \mu b.(M\{\,[b](V[\,\cdot\,])\,/\,[a][\,\cdot\,]\,\}$

(η_v) $\lambda x.(Vx) \longrightarrow V \quad (x \notin \mathsf{fv}(V))$

$(\zeta_{\text{fun},\bot})$ $(\mu a^{\alpha\Rightarrow\bot}.M)N \longrightarrow M\{\,[\,\cdot\,]N\,/\,[a][\,\cdot\,]\,\}$

$(\mu\text{-}\beta)$ $[b]\mu a.M \longrightarrow M\{b/a\}$

$(\zeta_{\text{arg},\bot})$ $V^{\alpha\Rightarrow\bot}(\mu a^{\alpha}.M) \longrightarrow M\{\,V[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}$

$(\mu\text{-}\eta)$ $\mu a.[a]M \longrightarrow M \quad (a \notin \mathsf{fn}(M))$

$(\bot)$ $V^{\bot\Rightarrow\beta}M \longrightarrow \mu b^{\beta}.M \quad (b \text{ fresh})$

(ζ_{fun}) $(\mu a^{\alpha\Rightarrow\beta}.M)N \longrightarrow \mu b.M\{\,[b]([\,\cdot\,]N)\,/\,[a][\,\cdot\,]\,\}$

$(\bot_{\bot})$ $V^{\bot\Rightarrow\bot}M \longrightarrow M$

We let $\beta \neq \bot$. In $(\zeta_{\text{arg}},\zeta_{\text{arg},\bot})$, $\alpha \neq \bot$. In the rules we include η_v-reduction, unlike [26]. Inclusion or non-inclusion does not affect the subsequent technical development. Some rules use substitution $M\{\,C[\,\cdot\,]\,/\,[a][\,\cdot\,]\,\}$ defined next, assuming the bound name convention. Note that substitution is applied in a nested fashion in the last line.

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

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. Given a $\lambda\mu$-term, $\Gamma \vdash M : \alpha; \Delta$, 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, 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 [25].

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

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

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

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

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

$$[\![x:\alpha]\!]_u \stackrel{\text{def}}{=} \begin{cases} \overline{u}\langle x^{\overline{\alpha^\circ}}\rangle & (\alpha \neq \bot) \\ \mathbf{0} & (\alpha = \bot) \end{cases} \qquad [\![\lambda x^\alpha.M:\alpha\Rightarrow\beta]\!]_u \stackrel{\text{def}}{=} \begin{cases} \overline{u}(c)!c(xz).[\![M:\beta]\!]_z & (\alpha \neq \bot, \beta \neq \bot) \\ \overline{u}(c)!c(z).[\![M:\beta]\!]_z & (\alpha = \bot, \beta \neq \bot) \\ \overline{u}(c)!c(x).[\![M:\bot]\!]_z & (\alpha \neq \bot, \beta = \bot) \\ \overline{u}(c)!c.[\![M:\bot]\!]_z & (\alpha = \bot, \beta = \bot) \end{cases}$$

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

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

Fig. 2. Encoding of $\lambda\mu$-terms.

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.

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).

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

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

   where $\prod_{1\leq i\leq n} P_i$ (or $\prod_i P_i$) stands for the n-fold parallel composition $P_1|\cdots|P_n$.
2. (free output) $\overline{x}\langle\vec{y}^{\vec{\tau}}\rangle \stackrel{\text{def}}{=} \overline{x}(\vec{z})\Pi[z_i \to y_i]^{\overline{\tau_i}}$ 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 2 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. We omit the type environment from the source term in Figure 2. 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 querying information) and at its free control variables (for returning values). Note both of them are now channel names.

Proposition 3.1. $\Gamma \vdash M:\alpha\,;\Delta$ *implies* $\vdash_{\bigcirc} [\![M:\alpha]\!]_u \triangleright (u:\alpha\cdot\Delta)^\bullet,\ \Gamma^\circ$.

In Proposition 3.1, 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 $\mathbf{!}$ to $\mathbf{?}$.

Example 3.2 (variable) As a simplest example, consider $[\![x : \bot]\!]_u \overset{\text{def}}{=} \mathbf{0}$. Since $(x : \bot)^\circ = (u : \bot)^\bullet = \emptyset$, we have $\vdash_{\bigcirc} [\![x : \bot]\!]_u \triangleright (u : \bot)^\bullet,\ (x : \bot)^\circ$ 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 3.3 (identity, 1) By closing x in Example 3.2, $[\![\lambda x^\bot.x : \bot \Rightarrow \bot]\!]_u \overset{\text{def}}{=} \overline{u}(c)!c.\mathbf{0}$. Since $(\bot \Rightarrow \bot)^\bullet = (()^{\mathbf{!}})^{\mathbf{?}}$, we have $\vdash_{\bigcirc} [\![\lambda x^\bot.x : \bot \Rightarrow \bot]\!]_u \triangleright u\!:\!(\bot \Rightarrow \bot)^\bullet$.

Example 3.4 (identity, 2) If $\alpha \neq \bot$, then $[\![\lambda x^\alpha.x : \alpha \Rightarrow \alpha]\!]_u \overset{\text{def}}{=} \overline{u}(c)!c(xz).\overline{z}\langle x^{\overline{\alpha^\circ}}\rangle$.

Example 3.5 (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 $\neg\alpha \overset{\text{def}}{=} \alpha \Rightarrow \bot$: $\aleph \overset{\text{def}}{=} \lambda z^{\neg\neg\alpha}.\mu a^\alpha.z(\lambda x^\alpha.[a]x)$. 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 3.6 (control operator, 3) The following is the well-known witness of Peirce's law, $((A \supset B) \supset A) \supset A$, and corresponds to `call/cc` 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 is simplified with $\searrow$ 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.

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

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.

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. Using the definitions above, the generation of reduction can be attributed to one of the following cases: (1) (β_v)-rule or (η_v)-rule; (2) one of the μ-reduction rules; or (3) 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.

$$M \overset{\text{def}}{=} (\mu a^{\alpha\Rightarrow\beta}.[a]\lambda y^{\alpha}.\mu e^{\beta}.[a]f)g \longrightarrow \mu b^{\beta}.[b](\lambda y.\mu e.[b](fg))g \overset{\text{def}}{=} M' \qquad (1)$$

The encoding into π^{c}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 $(\boldsymbol{\nu}\, n)(!n(y).\overline{c}\langle yu\rangle|\overline{n}\langle x\rangle)$:

$$[\![M : \alpha \Rightarrow \beta]\!]_u \;\overset{\text{def}}{=}\; (\boldsymbol{\nu}\, a)(\overline{a}(c)!c(ye).\overline{a}\langle f\rangle \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \qquad (2)$$

On the right of (2), 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 do not depend on whether the argument (g above) is a value or not, explaining the shape of (ζ_{fun}).

To see how M' in (1) results from M in the encoding, we "copy" replications to make these two redexes contiguous, obtaining:

$$(\boldsymbol{\nu}\, a)(\overline{a}(c)!c(ye).(\boldsymbol{\nu}\, a)(\overline{a}\langle f\rangle \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \mid !a(c).\overline{c}\langle\!\langle gu\rangle\!\rangle) \qquad (3)$$

This term is an intermediate form before reducing the mentioned two redexes in (2) and is behaviourally equivalent to (2) (even in the untyped weak bisimilarity). We observe:

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

so the intermediate form (3) is nothing but the encoding of M'. This also shows if we really reduce the two $\searrow$-redexes from (2), the result goes past (3). 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 (3).Based on this observation, we formally state the main result. Below size(M) is the size of M, which is inductively defined as: size$(x) = 1$, size$([a]M) = 1 +$ size(M), size$(\lambda x.M) =$ size$(M) + 1$, size$(\mu a.M) = 1 +$ size(M), size$(MN) =$ size$(M) +$ size(N). We use this index for maintaining the well-ordering on reduction. Below $\rightarrow_{\lambda\mu\mathrm{v}}$ is the reduction relation on $\lambda\mu$-terms presented in [26].

Proposition 3.2. *$M \rightarrow_{\lambda\mu\mathrm{v}} M'$ with M and M' typed implies either $[\![M:\alpha]\!] \equiv [\![M':\alpha]\!]$ such that size$(M) \gneqq$ size(M'), or $[\![M:\alpha]\!]_u \searrow^+ P$ such that $[\![M':\alpha]\!]_u \searrow^* P$. In particular, $\rightarrow_{\lambda\mu\mathrm{v}}$ on $\lambda\mu$-terms is strongly normalising.*

4 Decoding and Full Abstraction

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) [2, 4, 19], which are essentially a subset of $\lambda\mu$v-terms whose syntactic structures precisely correspond to their process representation.

First, *CNF preterms* $(\mathtt{N}, \ldots)$ and *CNF value preterms* $(\mathtt{U}, \ldots)$ are given by:

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

We further assume the following conditions on CNF preterms: (1) In $[a]\mathtt{N}$, $\mathtt{N}$ does not have form $\mu b^{\beta}.\mathtt{N}'$. (2) In $\mu a^{\alpha}.\mathtt{N}$, (a) if $\mathtt{N}$ is $[a]\mathtt{U}$ then $a \in \mathsf{fn}(\mathtt{U})$; and (b) if $\mathtt{N}$ is $\mathtt{let}\ x = y\mathtt{U}'\ \mathtt{in}\ \mathtt{N}'$ then $a \in \mathsf{fn}(\mathtt{U}')$. (3) In $\mu a^{\alpha}.[a]\mathtt{U}$, $a \in \mathsf{fn}(\mathtt{U})$. 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 $\mu a.\mathtt{let}\ x = y\mathtt{U}'\ \mathtt{in}\ \mathtt{N}'$ can be written $\mathtt{let}\ x = y\mathtt{U}'\ \mathtt{in}\ \mu a.\mathtt{N}'$.

Under these conditions, the set of CNFs are those which are typable by the following typing rules combined with those for the $\lambda\mu$-calculus except the rule for application.

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

In ($\bot$-const), 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

$$[\mathbf{0}]_u^{\Gamma;\,\Delta} \overset{\text{def}}{=} \mathtt{c}\!:\!\bot \qquad [\overline{u}(c)!c(xz).R]_u^{\Gamma;\,\Delta\cdot u:(\alpha\Rightarrow\beta)} \overset{\text{def}}{=} \lambda x^{\alpha}.[R]_z^{\Gamma\cdot x:\overline{\alpha};\,\Delta\cdot z:\beta}$$
$$[\overline{y}]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} \overset{\text{def}}{=} \mathtt{let}\ \_ = y\mathtt{c} \qquad [\overline{u}(c)!c(x).R]_u^{\Gamma;\,\Delta\cdot u:(\alpha\Rightarrow\bot)} \overset{\text{def}}{=} \lambda x^{\alpha}.[R]_m^{\Gamma\cdot x:\overline{\alpha};\,\Delta}$$
$$[\overline{u}(c)!c.R]_u^{\Gamma;\,\Delta\cdot u:(\bot\Rightarrow\bot)} \overset{\text{def}}{=} \lambda x^{\bot}.[R]_m^{\Gamma;\,\Delta} \quad [\overline{u}(c)!c(z).R]_u^{\Gamma;\,\Delta\cdot u:(\bot\Rightarrow\beta)} \overset{\text{def}}{=} \lambda x^{\bot}.[R]_z^{\Gamma;\,\Delta\cdot z:\beta}$$

$$[P_{\langle a\rangle}]_u^{\Gamma;\,\Delta\cdot a:\alpha} \overset{\text{def}}{=} [a][P_{\langle m/a\rangle}]_m^{\Gamma;\Delta\cdot a:\alpha\cdot m:\alpha} \qquad [\overline{y}(w)R]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} \overset{\text{def}}{=} \mathtt{let}\ \_ = y[\overline{c}(w)P]_c^{\Gamma;\Delta}$$

$$[\overline{y}(wz)(R\,|\,!z(x).Q)]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} \overset{\text{def}}{=} \mathtt{let}\ x^{\beta} = y[\overline{c}(w)R]_c^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta}\ \mathtt{in}\ [Q]_u^{(\Gamma\cdot x:\beta);\Delta}$$
$$[\overline{y}(z)!z(x).Q]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta;\Delta} \overset{\text{def}}{=} \mathtt{let}\ x^{\beta} = y\mathtt{c}\ \mathtt{in}\ [Q]_u^{\Gamma\cdot y:\alpha\Rightarrow\beta\cdot x:\beta;\Delta}$$
$$[P]_u^{\Gamma;\,\Delta\cdot u:\alpha} \overset{\text{def}}{=} \mu u^{\alpha}.[P]_m^{\Gamma;\,\Delta\cdot u:\alpha}\ \text{other cases}$$

Fig. 3. Decoding of $\lambda\mu$-typed processes ($\alpha,\beta \neq \bot$ and m,u are fresh)

a CNF value is either $\mathtt{c}$ (which is the sole case when it has a type $\bot$), a λ-abstraction, or a μ-abstraction followed by a λ-abstraction.

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.

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

For CNFs which are λ-abstraction, μ-abstraction and named terms, the mapping uses the same clauses as in Figure 2, replacing $[\![\,\cdot\,]\!]$ in the defining clauses with $(\,\cdot\,)^*$.

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

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

CNFs can also be directly encoded into π^{c}-processes, using the following rules combined with those for abstraction, naming and μ-abstraction given in Figure 2 (replacing $[\![\,\cdot\,]\!]$ with $\langle\,\cdot\,\rangle$ in each clause).

$$\langle \mathtt{let}\ x = y\mathtt{U}\ \mathtt{in}\ \mathtt{N}\!:\!\gamma\rangle_u \overset{\text{def}}{=} \begin{cases} \overline{y}(wz)(P|!z(x).\langle \mathtt{N}\!:\!\gamma\rangle_u) & (\mathtt{U} \neq \mathtt{c},\ \langle \mathtt{U}\rangle_c \overset{\text{def}}{=} \overline{c}(w)P) \\ \overline{y}(z)!z(x).\langle \mathtt{N}\!:\!\gamma\rangle_u & (\mathtt{U} = \mathtt{c}) \end{cases}$$
$$\langle \mathtt{let}\ \_ = y\mathtt{U}\!:\!\bot\rangle_u \overset{\text{def}}{=} \begin{cases} \langle \mathtt{U}\rangle_y & (\mathtt{U} \neq \mathtt{c}) \\ \overline{y} & (\mathtt{U} = \mathtt{c}) \end{cases} \qquad \langle \mathtt{c}\!:\!\bot\rangle_u \overset{\text{def}}{=} \mathbf{0}$$

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

Proposition 4.1. *1. Let $\Gamma \vdash \mathtt{N}\!:\!\alpha\,;\Delta$. Then $\vdash_{\bigcirc} \langle \mathtt{N}\rangle_u \triangleright (u\!:\!\alpha, \Delta)^{\bullet}, \Gamma^{\circ}$.*
2. Let $\Gamma \vdash \mathtt{N}\!:\!\alpha\,;\Delta$. Then $[\![\mathtt{N}^* : \alpha]\!]_u \searrow^* \langle \mathtt{N}\rangle_u \not\searrow$.

Definability. The decoding of π^{c}-processes (of encoded $\lambda\mu$v-types) to $\lambda\mu$v-preterms is written $[P]_u^{\Gamma\,;\,\Delta}$, which translates $P \in \mathsf{NF}_e$ such that $\vdash_{\bigcirc}\ P \triangleright \Gamma^{\circ}, \Delta^{\bullet}$ with $u \notin$

dom(Γ) to a $\lambda\mu$v-preterm M. Without loss of generality, we assume P does not contain redundant **0** or hiding. The mapping is defined inductively by the rules given in Figure 3. 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 4.2. *Let $\bot \notin$ image(Γ), $u \notin$ dom(Γ) 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 $u \notin$ dom(Δ) then $\Gamma \cdot x : \bot \vdash [P]_u^{\Gamma^\circ ; \Delta^\bullet} : \bot \,;\, \Delta_0$.*

Let us say $\Gamma \vdash M : \beta \,; \Delta$ with $u \notin \mathrm{dom}(\Gamma)$ *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 4.1. (definability) *Let $\vdash P \triangleright \Gamma^\circ \cdot \Delta^\bullet \cdot u : \alpha^\bullet \in \mathsf{NF}_e$ such that $\bot \notin$ image(Γ). Then $\Gamma \cdot x : \bot \vdash [P]_u : \alpha \,;\, \Delta$ defines P. Further if $\Gamma = \Delta = \emptyset$ and $P \not\equiv$ **0**, then there is a closed $\lambda\mu$v-term which defines P.*

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: (1) The congruence is closely related with (and possibly coincide with some of) the notions of equality over full controls, as studied by Laird [20], Selinger [34] and others; and (2) 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 4.1. *$\equiv_\bot$ is the smallest typed congruence on $\lambda\mu$v-terms which includes:*

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

For example, we have, under the environment $x : \bot, y : \bot$: $x \equiv_\bot y$. We also have: $\lambda x^\bot.\lambda y^\bot x \equiv_\bot \lambda x^\bot.\lambda y^\bot y$. We can easily check that, in the encoding, $\equiv_\bot$-related terms are always mapped to an identical process.

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

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

Definition 4.2. Define $\{W_i\}_{i\in\omega}$ by the following induction: $W_0 \overset{\mathrm{def}}{=} \lambda z^\bot.\mu u^{\bot\Rightarrow\bot}.z$ and $W_{n+1} \overset{\mathrm{def}}{=} \lambda z^\bot.\mu u^{\bot\Rightarrow\bot}.[w]W_n$. Let $\gamma = \bot \Rightarrow \bot \Rightarrow \bot$. We then define: $Obs \overset{\mathrm{def}}{=} \{W_0\} \cup \{\mu\, w^\gamma.[w]W_{n+1},\ n \in \mathbb{N}\}$ where we take terms up $\equiv_\bot$.

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 4.3. Define $\{P_i\}_{i\in\omega}$ as follows (below we use the same names for bound names for simplicity). $P_0 \stackrel{\text{def}}{=} \overline{w}(c)!c(u).\mathbf{0}$ and $P_{n+1} \stackrel{\text{def}}{=} \overline{w}(c)!c(u).P_n$. We set $Obs_\pi \stackrel{\text{def}}{=} \{\vdash_{\bigcirc} P_i \triangleright w : \gamma^\bullet\}_{i\in\omega}$, 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, $P'_1 \stackrel{\text{def}}{=} \overline{w}(c)!c(u).\overline{u}(c)!c(u).\mathbf{0}$ 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 $[\![\lambda z^\perp.\lambda x^\perp.x]\!]_w \stackrel{\text{def}}{=} P'_1$.

Proposition 4.3. *1.* $\vdash_{\bigcirc} P_i \cong_\pi P_j \triangleright w : \gamma^\bullet$ *for arbitrary* i *and* j.
2. If $\vdash_{\bigcirc} Q \triangleright w : \gamma^\bullet$, $Q \in \mathsf{NF}_e$ *and* $Q \cong_\pi P_i$ *then* $Q \in Obs_\pi$.

Processes in Obs_π have uniform behaviours: indeed they are closed under $\cong_\pi$. 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 4.4. We write $\Gamma \vdash M \cong_{\lambda\mu} N : \alpha\,;\Delta$ when, for each typed context $C[\,]^{\perp\Rightarrow\perp\Rightarrow\perp}_{\Gamma;\alpha;\Delta}$, we have: $\exists L.(C[M] \Downarrow L \in Obs)$ iff $\exists L'.(C[N] \Downarrow L' \in Obs)$.

Note that we treat all values in Obs as an identical observable. Immediately $\rightarrow_{\lambda\mu\mathrm{v}} \subset \cong_{\lambda\mu}$. We can now establish the full abstraction, following the standard routine. We start with the computational adequacy. We write $M \Downarrow L$ when $M \rightarrow^*_{\lambda\mu\mathrm{v}} L \not\rightarrow_{\lambda\mu\mathrm{v}}$.

Proposition 4.4. (computational adequacy) *Let* $M : \perp \Rightarrow \perp \Rightarrow \perp$ *be closed. Then* $\exists L.(M \Downarrow L \in Obs)$ *iff* $\exists P.([\![M]\!]_u \searrow^* P \in Obs_\pi)$.

Corollary 4.1. (soundness) $[\![M]\!]_u \cong_\pi [\![N]\!]_u$ *implies* $M \cong_{\lambda\mu} N$.

Proof. Assume $[\![M]\!]_u \cong_\pi [\![N]\!]_u$. We show, for each well-typed $C[\cdot]$, $\exists L.(C[M] \Downarrow L \in Obs)$ iff $\exists L'.(C[M] \Downarrow L' \in Obs)$. Let $C[\cdot]$ be well-typed. Now we reason:

$$\begin{array}{rcll} C[M] \Downarrow L \in Obs & \Rightarrow & [\![C[M]]\!]_v \Downarrow [\![L]\!]_v \in Obs_\pi & \text{(Proposition 4.4)} \\ & \Rightarrow & \exists O.[\![C[N]]\!]_v \searrow^* O \in Obs_\pi & ([\![C[M]]\!]_v \cong_\pi [\![C[N]]\!]_v) \\ & \Rightarrow & C[N]_v \searrow^* L' \in Obs & \text{(Proposition 4.4)} \end{array}$$

Theorem 4.2. (full abstraction) *Let* $\Gamma \vdash M_i : \alpha;\Delta$ *(*$i = 1,2$*). Then* $M_1 \cong_{\lambda\mu} M_2$ *if and only if* $[\![M_1]\!]_u \cong_\pi [\![M_2]\!]_u$.

Proof. Suppose $\emptyset \vdash M_1 \cong_{\lambda\mu} M_2 : \alpha; \emptyset$ but $\vdash_{\mathrm{O}} [\![M_1]\!]_u \not\cong_\pi [\![M_2]\!]_u \triangleright u : \alpha^\bullet$. By this, converting the observable $()^?$ to the convergence to Obs_π in $\gamma^\bullet$ with $\gamma = \bot \Rightarrow \bot \Rightarrow \bot$, there exists $\vdash_{\mathrm{I}} R \triangleright u : \overline{\alpha^\bullet}, v : \gamma^\bullet$ such that $R \in \mathsf{NF}_e$ and (say) $\exists P.\ (\boldsymbol{\nu}\, u)([\![M_1]\!]|R) \searrow^* P \in Obs_\pi$ and $\neg\exists P.\ (\boldsymbol{\nu}\, u)([\![M_2]\!]|R) \searrow^* P \in Obs_\pi$. Since $R \in \mathsf{NF}_e$, we can safely set $R \overset{\text{def}}{=} !u(c).R'$. Now take $\vdash_{\mathrm{I}} !u(cv).R' \triangleright u : (\alpha \Rightarrow \gamma)^\bullet$. By Theorem 4.1 (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 $\exists L'.\ LM_1 \Downarrow L' \in Obs$ and $\neg\exists L'.\ LM_2 \Downarrow L' \in Obs$, which contradicts the assumption. 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

This paper explored the connection between control and the π-calculus, first pointed out by Thielecke [35] who showed that the target of CPS-transform can be written down as name passing processes. This paper presented the typed π-calculus for full control, which arises as a subcalculus of the linear π-calculus [39] where all inputs are replicated. The main contribution of the present work 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.

Control and Name Passing (1). The notion of full control arises in several related contexts. Historical survey of studies of controls and continuations can be found in [28, 36]. 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 [27] 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.

Control and Name Passing (2). Another closely related context is the CPS transform [8, 12, 30]. 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 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:

$$\lambda k.(\lambda h.h\lambda x.\langle\!\langle x\rangle\!\rangle)(\lambda m.\langle\!\langle 1\rangle\!\rangle(\lambda n.mnk)) \qquad (4)$$

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 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 (4) is isomorphic to its process encoding via $[\![\cdot]\!]$ of Section 3, given as:

$$(\boldsymbol{\nu}\, h)([\![\lambda x.x]\!]_h \mid !h(n).(\boldsymbol{\nu}\, h')([\![1]\!]_{h'} \mid !h'(m).\overline{n}\langle mk\rangle)) \qquad (5)$$

In (5), 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 (4) are replaced by hiding. Setting $[\![1]\!]_{h'}$ to be $\overline{h'}\langle 1\rangle$ (regarding 1 as a specific name), we can see how (5) reduces precisely as (4) reduces modulo the book-keeping reductions. Sangiorgi [32] observed that we can regard (4) as terms in the applicable 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 λ-calculi, the main contribution of our 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, 30], on the one hand, and the encoding/decoding in Sections 3 and 4 in this paper on the other.

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 [35], Laird [21] and Selinger [34]). The present work may have two interests in this context.

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.

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 [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.

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, as discussed in [14].

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. One interesting aspect is the way non-determinism 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 mapping $[\alpha_1, ..., \alpha_n, \perp]^\circ \stackrel{\text{def}}{=} (\overline{\alpha_1^\circ}, ..., \overline{\alpha_n^\circ})^!$ is the standard Hyland-Ong encoding of call-by-name types [4] assuming the only atomic type is $\perp$. In the presence of control, we can simply augment this map with $\alpha^\bullet \stackrel{\text{def}}{=} (\alpha^\circ)^?$ 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.

Van Bakel and Vigliotti [37] present a different approach towards representing control as interaction. Milner's encoding of λ-calculus does not model reductions under λ-binders by matching reductions in the π-calculus. They consider a different encoding of a variant of the $\lambda\mu$-calculus with explicit substitution that preserves single-step explicit head reduction.

Finally, the idea of viewing non-local control-flow manipulation in λ-calculi as typed interacting processes in π^{c} has been fruitful for finding Hoare logics for languages with call/cc: [3] produces the first such logic by designing a Hennessy-Milner logic for a variant of π^{c} and then pushes that logic back to call-by-value PCF extended with with call/cc to obtain a conventional Hoare logic.

References

1. A full version of this paper. Doc Technical Report DTR 2014/2, Department of Computing, Imperial College London, April 2014.
2. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full Abstraction for PCF. *Info. & Comp.*, 163:409–470, 2000.
3. M. Berger. Program Logics for Sequential Higher-Order Control. In *Proc. FSEN*, pages 194–211, 2009.
4. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.
5. M. Berger, K. Honda, and N. Yoshida. Genericity and the pi-calculus. *Acta Inf.*, 42(2-3):83–141, 2005.
6. P.-L. Curien and H. Herbelin. Computing with Abstract Böhm Trees. In *Functional and Logic Programming*, pages 20–39. World Scientific, 1998.
7. P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. ICFP*, pages 233–243, 2000.
8. O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *MSCS*, 2(4):361–391, 1992.
9. P. 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. M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. Syntactic theories of sequential control. *TCS*, 52:205–237, 1987.
11. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2):235–271, 1992.
12. C. Führmann and H. Thielecke. On the call-by-value CPS transform and its semantics. *Inf. Comput.*, 188(2):241–283, 2004.
13. T. G. Griffin. A formulae-as-type notion of control. In *Proc. POPL*, pages 47–58, 1990.
14. K. Honda and O. Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.*, 411(22-24):2223–2238, 2010.

15. K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 151:393–456, 1995.
16. K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *TCS*, 221:393–456, 1999.
17. K. Honda and N. Yoshida. Noninterference through flow analysis. *JFP*, 15(2):293–349, 2005.
18. K. Honda and N. Yoshida. A uniform type structure for secure information flow. *TOPLAS*, 29(6), 2007.
19. J. M. E. Hyland and C. H. L. Ong. On full abstraction for PCF. *Inf. & Comp.*, 163:285–408, 2000.
20. J. Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.
21. J. Laird. A game semantics of linearly used continuations. In *Proc. FOSSACS*, number 2620 in LNCS, pages 313–327, 2003.
22. O. Laurent. Polarized games. In *Proc. LICS*, pages 265–274, 2002.
23. O. Laurent. Polarized proof-nets and $\lambda\mu$-calculus. *TCS*, 290(1):161–188, 2003.
24. R. Milner. Functions as Processes. *MSCS*, 2(2):119–141, 1992.
25. E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
26. C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proc. POPL*, pages 215–227, 1997.
27. M. Parigot. λ-μ-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *Proc. LPAR*, pages 190–201, 1992.
28. J. Reynolds. The discovers of continuations. *Lisp and Symbolic Computation*, 6:233–247, 1993.
29. E. Robinson. Kohei Honda. *Bulletin of the EATCS*, 112, February 2014.
30. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LFP*, pages 288–298, 1992.
31. D. Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *TCS*, 167(2):235–274, 1996.
32. D. Sangiorgi. From λ to π: or, rediscovering continuations. *MSCS*, 9:367–401., 1999.
33. V. Sassone. ETAPS Award: Laudatio for Kohei Honda. *Bulletin of the EATCS*, 112, February 2014.
34. P. Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *MSCS*, 11(2):207–260, 2001.
35. H. Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997.
36. H. Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, 1999.
37. S. van Bakel and M. G. Vigliotti. An Output-Based Semantics of $\Lambda\mu$ with Explicit Substitution in the π-Calculus. In *Proc. IFIP*, TCS'12, pages 372–387, 2012.
38. P. Wadler. Call-by-value is dual to call-by-name. In *Proc. ICFP*, pages 189–201, 2003.
39. N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the π-calculus. *Inf. Comput.*, 191(2):145–202, 2004.
40. N. Yoshida, K. Honda, and M. Berger. Linearity and bisimulation. *J. Log. Algebr. Program.*, 72(2):207–238, 2007.

A Typing

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. We define $A \asymp B$ iff: (1) whenever $x : \tau \in A$ and $x : \tau' \in B$, $\tau \odot \tau'$ is defined; and (2) 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.

Finally, the third condition, 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 $\odot$: $\mathtt{I} \odot \mathtt{I} = \mathtt{I}$ and $\mathtt{I} \odot \mathtt{O} = \mathtt{O} \odot \mathtt{I} = \mathtt{O}$. 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.

$$\text{(Zero)}\ \frac{-}{\vdash_{\mathtt{I}} 0 \triangleright \emptyset} \qquad \text{(Par)}\ \frac{\vdash_{\phi_i} P_i \triangleright A_i \quad (i = 1, 2) \qquad A_1 \asymp A_2 \quad \phi_1 \asymp \phi_2}{\vdash_{\phi_1 \odot \phi_2} P_1 | P_2 \triangleright A_1 \odot A_2} \qquad \text{(Res)}\ \frac{\vdash_{\phi} P \triangleright A \qquad \mathsf{md}(A(x)) = \mathbf{!}}{\vdash_{\phi} (\boldsymbol{\nu}\, x) P \triangleright A / x} \qquad \text{(Weak)}\ x \notin \mathsf{fn}(A)\ \frac{\vdash_{\phi} P \triangleright A \qquad \mathsf{md}(\tau) = \mathbf{?}}{\vdash_{\phi} P \triangleright A,\ x : \tau}$$

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

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. 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. In (Res), we do not allow $\mathbf{?}$ to be restricted since this action expects its dual server always exists in the environment. $A/\vec{y}$ means the result of deleting the nodes $\vec{x} : \vec{\tau}$ in A (and edges from/to deleted nodes). 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. ($\mathsf{In}^{\mathbf{!}}$) 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, $x : (\vec{\tau})^{\mathbf{!}} \to A$ simply stands for $x : (\vec{\tau})^{\mathbf{!}}$. ($\mathsf{Out}^{\mathbf{?}}$) 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.