Sequentiality and the π-Calculus ⋆

Martin Berger[†] Kohei Honda[†] Nobuko Yoshida[‡]

Abstract. We present a type discipline for the π-calculus which precisely captures the notion of sequential functional computation as a specific class of name passing interactive behaviour. The typed calculus allows direct interpretation of both call-by-name and call-by-value sequential functions. The precision of the representation is demonstrated by way of a fully abstract encoding of PCF. The result shows how a typed π-calculus can be used as a descriptive tool for a significant class of programming languages without losing the latter's semantic properties. Close correspondence with games semantics and process-theoretic reasoning techniques are together used to establish full abstraction.

1 Introduction

This paper studies a type discipline for the π-calculus which precisely captures the notion of sequential functional computation. The precision of the representation is demonstrated by way of a fully abstract encoding of PCF. Preceding studies have shown that while operational encodings of diverse programming language constructs into the π-calculus are possible, they are rarely fully abstract [28, 32]: we necessarily lose information by such a translation. The translation of a source term M will generally result in a process containing more behaviour than M. Type disciplines for the π-calculus with significant properties such as linearity and deadlock-freedom have been studied before [9, 16, 21, 22, 29, 30, 37], but, to our knowledge, no previous typing system for the π-calculus has enabled a fully abstract translation of functional sequentiality. The present work shows that a relatively simple typing system suffices for this purpose. Despite its simplicity, the calculus is general enough to give clean interpretations of both call-by-name and call-by-value sequentiality, offering a basic articulation of functional sequentiality without relying on particular evaluation strategies.

The core idea of the typing system is that *affineness* and *stateless replication* ensure deterministic computation. Sequentiality is guaranteed by controlling the number of threads through restricting the shape of processes. While the idea itself is simple, the result would offer a technical underpinning for the potential use of typed π-calculi as meta-languages for programming language study: having fully abstract descriptions in this setting means ensuring the results obtained in the meta-language to be transferable, in principle, to object languages. In a later exposition we wish to report how the proposed typed syntax can be a powerful tool for language analysis when coupled with process-theoretic reasoning techniques.

⋆ A short version appears in Proc. of *TLCA'01*, the 5th International Conference on Typed Lambda Calculi and Applications, LNCS, Springer, May, 2001.
[†]Queen Mary, University of London, U.K. [‡]University of Leicester, U.K.

From the viewpoint of the semantic study of sequentiality [6, 10, 27], our work positions sequentiality as a sub-class of the general universe of name passing interactive behaviour. This characterisation allows us to delineate sequentiality against the background of a broad computational universe which, among others, includes concurrency and non-determinism, offering a uniform basis on which various semantic findings can be integrated and extensions considered. A significant point in this context is the close connection between the presented calculus and game semantics [3, 20, 23]: the structure of interaction of typed processes (with respect to typed environments) precisely conforms to the intensional structures of games introduced in [23] and studied in e.g. [2, 11, 20, 25, 26]. It is notable that the type discipline itself does not mention basic notions in game semantics such as visibility, well-bracketing and innocence (although it does use a syntactic form of IO-alternation): yet they are derivable as operational properties of typed processes. We use this correspondence combined with process-theoretic reasoning techniques to establish full abstraction. While we expect a direct behavioural proof would be possible, the correspondence, in addition to facilitating the proof, offers deeper understanding of the present type discipline and game semantics.

We briefly give comparisons with related work. Hyland and Ong [24] presented a π-calculus encoding of innocent strategies of their games and show operational correspondence with a π-calculus encoding of PCF. Fiore and Honda [11] propose another π-calculus encoding for call-by-value games [20]. Our work, while being built on these preceding studies, is novel in that it puts forward a general type discipline where typability ensures functional sequentiality. In comparison with game semantics, our approach differs as it is based on a syntactic calculus representing a general notion of concurrent, communicating processes. In spite of the difference, our results do confirm some of the significant findings in game semantics, such as the equal status owned by call-by-name and call-by-value evaluation. From a different viewpoint, our work shows an effective way to apply game semantics to the study of basic typing systems for the π-calculus, in particular for the proof of full abstraction of encodings. Concerning the use of the π-calculus as the target language for translations, [28] was the first to point out the difficulty of fully abstract embeddings of functional sequentiality and [32] showed that the same problems arise even with the higher-order π-calculus. While some preceding work studies the significance of replication and linearity of channels [9, 16, 22, 29, 31, 34, 37], none offers a fully abstract interpretation of functional sequentiality.

In the remainder, Section 2 and 3 introduce the typed calculus. Section 4 analyses operational structures of typed terms. Based on them Section 5 establishes full abstraction. The technical details, including proofs omitted from the main sections of the paper, can be found in the full version [4].

Acknowledgements. We thank Makoto Hasegawa and Vasco Vasconcelos for their comments. The first two authors have been partially supported by an EPSRC grant (GR/N/37633).

2 Processes

2.1 Syntax

We use a variant of the π-calculus as our base syntax. As in typed λ-calculi, we start from the leanest untyped syntax. The following gives the reduction rule of the asynchronous version of the π-calculus, introduced in [8, 18]:

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

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

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

where the replicated process remains in the configuration after reduction.

Types for processes prescribe usage of names [29, 36]. To be able to do this with precision, it is important to control dynamic sharing of names. For this purpose it is essential to distinguish *free name passing* and *bound (private) name passing*: the latter allows tight control of sharing and can control name usage in more stringent ways. In the present study, using bound name passing alone is sufficient. Further, to have tractable inference rules, it is vital to specify bound names associated with the concerned output. Thus, instead of $(\boldsymbol{\nu}\,\vec{y})(\overline{x}\langle\vec{y}\rangle|P)$, we write $\overline{x}(\vec{y})\,P$, and replace (1) by the following reduction rule.

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

Here "$\overline{x}(\vec{y})\,Q$" indicates that $\overline{x}(\vec{y})$ is an asynchronous output exporting $\vec{y}$ which are local to Q. The rule corresponding to (2) is given accordingly. To ensure asynchrony of outputs, we add the following rule to the standard closure rules for $\mid$ and $(\boldsymbol{\nu}\,x)$.

$$P \;\longrightarrow\; P' \quad\Rightarrow\quad \overline{x}(\vec{y})\,P \;\longrightarrow\; \overline{x}(\vec{y})\,P' \tag{4}$$

Further, the following structural rules are added to allow inference of interaction under an output prefix.

$$\overline{x}(\vec{z})\,(P|Q) \equiv (\overline{x}(\vec{z})\,P)|Q \qquad \text{if } \mathsf{fn}(Q) \cap \{\vec{z}\} = \emptyset, \tag{5}$$

$$\overline{x}(\vec{z})\,(\boldsymbol{\nu}\,y)P \equiv (\boldsymbol{\nu}\,y)\overline{x}(\vec{z})\,P \qquad \text{if } y \notin \{x, \vec{z}\}. \tag{6}$$

By these rules we maintain the dynamics based on the original asynchronous calculus (up to the equation $\overline{x}(\vec{z})\,P \;\equiv\; (\boldsymbol{\nu}\,\vec{z})(\overline{x}\langle\vec{z}\rangle|P)$), while enabling output actions to be typed with the same ease as input actions. Name-passing calculi using only bound name passing, called πI-calculi, have been studied in [7, 33].

Another useful construct for typing is *branching*. Branching is similar to the "case" construct in typed λ-calculi and can represent both base values such as booleans or integers and conditionals. While binary branching has some merit, we use indexed branching because it simplifies the description of base value passing. The branching variant of the reduction (3) becomes:

$$x[\&_{i\in I}(\vec{y}_i).P_i] \mid \overline{x}\mathtt{in}_j(\vec{y}_j)\,Q \;\longrightarrow\; (\boldsymbol{\nu}\,\vec{y}_j)(P_j \mid Q) \qquad (7)$$

where we assume $j \in I$, with $I\,(\neq \emptyset)$ denoting a finite or countably infinite indexing set. Accordingly we define the rule for replicated branching. Branching constructs of this kind have been studied in tyco [35] and other calculi [12, 15, 17] (the corresponding type structure already appeared in Linear Logic [1, 13]).

Augmenting the original asynchronous syntax with bound output and branching, we now arrive at the following grammar.

$P ::= x(\vec{y}).P$	input	$\mid$	$P \mid Q$	parallel
$\mid\quad \overline{x}(\vec{y})\,P$	output	$\mid$	$(\boldsymbol{\nu}\,x)P$	hiding
$\mid\quad x[\&_{i\in I}(\vec{y}_i).P_i]$	branching input	$\mid$	$\mathbf{0}$	inaction
$\mid\quad \overline{x}\mathtt{in}_i(\vec{z})\,P$	selection	$\mid$	$!P$	replication

In $!P$ we require P to be either a unary or branching input. The bound/free names/variables are defined as usual and we assume the variable convention for bound names. The structural rules are standard except for the omission of $!P \equiv\, !P|P$ and the incorporation of (5) as well as (6) together with the corresponding rules for branching output. The reduction rules are as explained above, which also include variants of (3) and (7) for replicated branching inputs.

2.2 Examples

Henceforth we omit trailing zeros and null arguments and write $x[\&_i P_i]$ for $x[\&_i().P_i]$.

(i) $[\![\mathtt{n}]\!]_u \stackrel{\text{def}}{=} !\,u(a).\overline{a}\mathtt{in}_n$. Each time $[\![\mathtt{n}]\!]_u$ is invoked, it replies by telling its number, $\mathtt{n}$. Here a natural number becomes a *stateless server.*

(ii) $[\![\mathtt{succ}]\!]_u \stackrel{\text{def}}{=} !\,u(ya).\overline{y}(b)\,b[\&_n\,\overline{a}\mathtt{in}_{n+1}]$. $[\![\mathtt{succ}]\!]_x$ describes the behaviour of a successor function, which queries for its argument, a natural number as in (i) above, and returns its increment. This is another stateless server but this time it asks its client for an input.

(iii) $!u(xa).\overline{x}(zb)\,([\![\mathtt{1}]\!]_z \mid b[\&_i\overline{a}\mathtt{in}_i])$. This represents a type-2 functional $\lambda x.x1$: $(\mathsf{Nat}\Rightarrow\mathsf{Nat})\Rightarrow\mathsf{Nat}$. When the process is invoked, it queries for its argument (which is a function itself), that function then asks back for its own argument, to which $[\![\mathtt{1}]\!]_z$ replies. Finally the process receives, at b, an answer to its own question, based on which it answers to the initial question.

3 Typing

3.1 Action Modes

Functional computation is *deterministic.* There are two basic ways to realise this in interacting processes. One is to have (at most) one input and (at most) one output at a given channel (such a channel is called *affine*). Another is to have a unique stateless replicated input with zero or more dual outputs. These ideas have been studied in the past [13, 15, 16, 21, 22, 31, 34, 37]. To capture them in typing, we use the following *action modes*, denoted $p, q, \ldots$:

$!_1$	Affine input	$?_1$	Affine output
$!_\omega$	Replicated input	$?_\omega$	Output to replicated input

We also use $\perp$ to denote the presence of both input and output at an affine channel. In the table above, the mode on the left and that on the right in the same row are *dual* to each other, denoted $\overline{p}$ (for example, $\overline{!_1} = ?_1$).

3.2 Channel Types

Channel types indicate possible usage of channels. We use sorting [29] augmented with branching [1, 13, 15, 17, 35] and action modes. The grammar follows.

$$\alpha ::= \langle \tau,\, \overline{\tau} \rangle \qquad \tau_{\mathtt{I}} ::= (\vec{\tau})^{!_1} \mid (\vec{\tau})^{!_\omega} \mid [\&_{i\in I}\, \vec{\tau}_i]^{!_1} \mid [\&_{i\in I}\, \vec{\tau}_i]^{!_\omega}$$
$$\tau ::= \tau_{\mathtt{I}} \mid \tau_{\mathtt{O}} \qquad \tau_{\mathtt{O}} ::= (\vec{\tau})^{?_1} \mid (\vec{\tau})^{?_\omega} \mid [\oplus_{i\in I}\, \vec{\tau}_i]^{?_1} \mid [\oplus_{i\in I}\, \vec{\tau}_i]^{?_\omega}$$

In the first line $\overline{\tau}$ denotes the *dual* of τ, which is the result of dualising all action modes and exchanging $\oplus$ and $\&$. A type of form $\langle \tau, \overline{\tau} \rangle$ is called *pair type*, which we regard as a set. $[\&_{i\in I} \ldots]$ corresponds to branching and $[\oplus_{i\in I} \ldots]$ corresponds to selection. As an example of types, let $\mathsf{Nat}^\bullet \stackrel{\text{def}}{=} [\oplus_{i\in\mathbb{N}}]^{?_1}$ and $\mathsf{Nat}^\circ \stackrel{\text{def}}{=} (\mathsf{Nat}^\bullet)^{!_\omega}$. Then in $!a(x).\overline{x}\mathtt{in}_n$, x is used as $\mathsf{Nat}^\bullet$ while a is used as Nat°.

A further idea in functional computation is asking a question and receiving a unique answer [3, 23]. A type is *sequential* when for each subexpression:

(i) In $(\vec{\tau})^{!_\omega}$, if $\vec{\tau} \neq \varepsilon$ then there is a unique τ_i of mode $?_1$, while each τ_j $(i \neq j)$ is of mode $?_\omega$. Dually for $(\vec{\tau})^{?_\omega}$. The same applies to $[\&_{i\in I}\, \vec{\tau}_i]^{!_\omega}$ and $[\oplus_{i\in I}\, \vec{\tau}_i]^{?_\omega}$.
(ii) In $(\vec{\tau})^{!_1}$, each τ_i is of mode $?_\omega$, dually for $(\vec{\tau})^{?_1}$. The same applies to $[\&_{i\in I}\, \vec{\tau}_i]^{!_1}$ and $[\oplus_{i\in I}\, \vec{\tau}_i]^{?_1}$.

As an example, $(\overline{\mathsf{Nat}^\circ}\mathsf{Nat}^\bullet)^{!_\omega}$ is a sequential type for $[\![\mathsf{succ}]\!]_u$ in §2.2 (ii).

3.3 Action Types and IO-Modes

The sequents we use have the form $\Gamma \vdash_\phi P \triangleright A$. Γ is a *base*, i.e. a finite map from names to channel types, P is a process with type annotations on binding names, A is an *action type*, and ϕ is an *IO-mode*. Intuitively, an action type witnesses the real usage of channels in P with respect to their modes specified in Γ (thus controlling determinacy); an IO-mode ensures P contains at most one active thread (thus controlling sequentiality). Below in (i) we use a symmetric partial operator $\odot$ on action modes generated from $!_1 \odot ?_1 = \perp$, $?_\omega \odot ?_\omega = ?_\omega$ and $!_\omega \odot ?_\omega = !_\omega$. Thus, for example, $!_\omega \odot !_\omega$ is undefined. This partial algebra ensures that only one-one (resp. one-many) connection is possible at an affine (resp. replicated) channel.

(i) An *action type* assigns action modes to names. Each assignment is written px. $\mathsf{fn}(A)$ denotes the set of names in A. A partial operator $A \odot B$ is defined iff $p \odot q$ is defined whenever $px \in A$ and $qx \in B$; then we set $A \odot B = (A \backslash B) \cup (B \backslash A) \cup \{(p \odot q)x \mid px \in A, qx \in B\}$. We write $A \asymp B$ when $A \odot B$ is defined. The set of modes used in A is $\mathsf{md}(A)$.

(ii) An *IO-mode* is one of $\{\mathtt{I}, \mathtt{O}\}$. We set $\mathtt{I} \odot \mathtt{I} = \mathtt{I}$ and $\mathtt{I} \odot \mathtt{O} = \mathtt{O} \odot \mathtt{I} = \mathtt{O}$. Note $\mathtt{O} \odot \mathtt{O}$ is not defined. When $\phi_1 \odot \phi_2$ is defined we write $\phi_1 \asymp \phi_2$.

In IO-modes, $\mathtt{O}$ indicates a unique active output (consider it as a thread): thus $\mathtt{O} \not\asymp \mathtt{O}$ shows that we do not want more than one thread in a process.

3.4 Typing Rules

$$\text{(Zero)}\quad \frac{\Gamma\ \text{Sequential}}{\Gamma \vdash_{\mathtt{I}} \mathbf{0} \triangleright \emptyset}$$

$$\text{(Par)}\quad \frac{\Gamma \vdash_{\phi_i} P_i \triangleright A_i \quad (i=1,2) \qquad A_1 \asymp A_2 \quad \phi_1 \asymp \phi_2}{\Gamma \vdash_{\phi_1 \odot \phi_2} P_1 | P_2 \triangleright A_1 \odot A_2}$$

$$\text{(Res)}\quad \frac{\Gamma \cdot x : \alpha \vdash_{\phi} P \triangleright A \otimes px \qquad p \in \{\bot, !_\omega\}}{\Gamma \vdash_{\phi} (\boldsymbol{\nu}\, x : \alpha) P \triangleright A}$$

$$(\mathsf{In}^{!_1})\quad (C/\vec{y} = ?A)\quad \frac{\Gamma \vdash x : (\vec{\tau})^{!_1} \qquad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_{\mathtt{O}} P \triangleright C^{\text{-}x}}{\Gamma \vdash_{\mathtt{I}} x(\vec{y} : \vec{\tau}).P \triangleright A \otimes !_1 x}$$

$$(\mathsf{Out}^{?_1})\quad (C/\vec{y} = A \asymp ?_1 x)\quad \frac{\Gamma \vdash x : (\vec{\tau})^{?_1} \qquad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_{\mathtt{I}} P \triangleright C}{\Gamma \vdash_{\mathtt{O}} \overline{x}(\vec{y} : \vec{\tau}) P \triangleright A \odot ?_1 x}$$

$$(\mathsf{Weak}\text{-}\bot)\quad \frac{\Gamma \vdash x : !_1, ?_1 \qquad \Gamma \vdash_{\phi} P \triangleright A^{\text{-}x}}{\Gamma \vdash_{\phi} P \triangleright A \otimes \bot x}$$

$$(\mathsf{In}^{!_\omega})\quad (C/\vec{y} = ?_\omega A)\quad \frac{\Gamma \vdash x : (\vec{\tau})^{!_\omega} \qquad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_{\mathtt{O}} P \triangleright C^{\text{-}x}}{\Gamma \vdash_{\mathtt{I}} ! x(\vec{y} : \vec{\tau}).P \triangleright A \otimes !_\omega x}$$

$$(\mathsf{Out}^{?_\omega})\quad (C/\vec{y} = A \asymp ?_\omega x)\quad \frac{\Gamma \vdash x : (\vec{\tau})^{?_\omega} \qquad \Gamma \cdot \vec{y} : \vec{\tau} \vdash_{\mathtt{I}} P \triangleright C}{\Gamma \vdash_{\mathtt{O}} \overline{x}(\vec{y} : \vec{\tau}) P \triangleright A \odot ?_\omega x}$$

$$(\mathsf{Weak}\text{-}?_\omega)\quad \frac{\Gamma \vdash x : ?_\omega \qquad \Gamma \vdash_{\phi} P \triangleright A^{\text{-}x}}{\Gamma \vdash_{\phi} P \triangleright A \otimes ?_\omega x}$$

Fig. 1. Sequential Typing System

The typing rules are given in Figure 1. The rules for branching/selection are defined similarly and left to Appendix A. The following notation is used:

$?_\omega A$	A s.t. $\mathsf{md}(A) = \{?_\omega\}$	$A^{\text{-}x}$	A s.t. $x \notin \mathsf{fn}(A)$
$?A$	A s.t. $\mathsf{md}(A) = \{?_\omega, ?_1\}$	$A \otimes B$	$A \cup B$ s.t. $\mathsf{fn}(A) \cap \mathsf{fn}(B) = \emptyset$
$A/\vec{x}$	$A \backslash \{p\vec{x}\}$ s.t. $\{\vec{x}\} \subseteq \mathsf{fn}(A)$	$\Gamma \cdot \Delta$	$\Gamma \cup \Delta$ s.t. $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$

$\Gamma \vdash x : \tau$ denotes $x : \tau$ or $x : \langle \tau, \overline{\tau} \rangle$ in Γ, while $\Gamma \vdash x : p$ indicates $\Gamma \vdash x : \tau$ such that the mode of τ is p. Typed processes are often called *sequential processes*. The sequent $\Gamma \vdash_\phi P \triangleright A$ is often abbreviated to $\Gamma \vdash_\phi P$.

We briefly illustrate each typing rule. In (Zero), we start in $\mathtt{I}$-mode since there is no active output. In (Par), "$\asymp$" controls composability, ensuring that at most one thread is active in a given term. In (Res), we do not allow $?_1$, $?_\omega$ or $!_1$-channel to be restricted since these actions expect their dual actions exist in the environment (cf. [16, 19, 22]). ($\mathsf{In}^{!_1}$) ensures that x occurs precisely once (by $C^{\text{-}x}$) and no free input is suppressed under prefix (by $C/\vec{y} = ?A$). ($\mathsf{Out}^{?_1}$) also ensures an output at x occurs precisely once, but does not suppress the body by prefix since output is asynchronous (essentially the rule composes the output prefix and the body in parallel). (Weak-$\perp$) allows assigning the same type after a pair of dual affine channels disappears following an interaction. This is essential for subject reduction. ($\mathsf{In}^{!_\omega}$) is the same as ($\mathsf{In}^{!_1}$) except no free $?_1$-channels are suppressed (note that if a $?_1$-channel is under replication then it can be used more than once). ($\mathsf{Out}^{?_\omega}$) and (Weak-$?_\omega$) say $?_\omega$-channels occur zero or more times, and it does not suppress any actions. Finally, in ($\mathsf{Out}^{?_1}$) and ($\mathsf{Out}^{?_\omega}$), the premise must have $\mathtt{I}$-mode for otherwise we would end up with more than one thread. Note that, for input, we require the premise to be $\mathtt{O}$-mode. This together ensures single-threadedness to be invariant under reduction, as we discuss later.

3.5 Examples

The following examples indicate how the present type discipline imposes strong constraints on term structure.

(i) Given $\Gamma = a\!:\!()^{?_1} \cdot b\!:\!\langle ()^{!_1}, ()^{?_1}\rangle \cdot c\!:\!()^{?_1}$, we build sequential processes one by one, starting from inaction. (1) $\Gamma \vdash_{\mathtt{I}} \mathbf{0} \triangleright \emptyset$, (2) $\Gamma \vdash_{\mathtt{O}} \overline{a} \triangleright ?_1 a$, and
(3) $\Gamma \vdash_{\mathtt{I}} b.\overline{a} \triangleright ?_1 a \otimes !_1 b$. Then we have:

$$\Gamma \vdash_{\mathtt{O}} \overline{b} \mid b.\overline{a} \triangleright ?_1 a \otimes \perp b \quad \text{with} \quad ?_1 b \odot !_1 b = \perp b \quad \text{and} \quad \mathtt{O} \odot \mathtt{I} = \mathtt{O}$$

where "$\perp b$" means name b is no longer composable. Note for any ϕ, $\Gamma \not\vdash_\phi b.\overline{a} \mid b.\overline{c}$ since b is affine.

(ii) Given $\Gamma = a\!:\!()^{?_\omega} \cdot b\!:\!\langle ()^{!_\omega}, ()^{?_\omega}\rangle$, we have:
– $\Gamma \vdash_{\mathtt{O}} \overline{a} \mid !b.\overline{a} \triangleright ?_\omega a \otimes !_\omega b$ with $?_\omega a \odot ?_\omega a = ?_\omega a$ and $\mathtt{O} \odot \mathtt{I} = \mathtt{O}$; and
– $\Gamma \vdash_{\mathtt{O}} !b.\overline{a} \mid \overline{b} \triangleright ?_\omega a \otimes !_\omega b$ with $?_\omega b \odot !_\omega b = !_\omega b$.
However, for any ϕ, $\Gamma \not\vdash_\phi \overline{a} \mid !b.\overline{a} \mid \overline{b}$ since $\mathtt{O} \odot \mathtt{O}$ is undefined. This example shows control by modes is essential even if $?_\omega$-mode channel does not appear in parallel; we can check after one step interaction between $!b.\overline{a}$ and $\overline{b}$, two messages to a will appear in parallel.

(iii) For $[\![\mathtt{n}]\!]_u$ in Example 2.2 (i), we have $u\!:\!\mathsf{Nat}^\circ \vdash_{\mathtt{I}} [\![\mathtt{n}]\!]_u$ (see § 3.2 for Nat°).

(iv) For $[\![\mathsf{succ}]\!]_u$ in Example 2.2 (ii), we can derive $u\!:\!(\overline{\mathsf{Nat}^\circ}\mathsf{Nat}^\bullet)^{!_\omega} \vdash_{\mathtt{I}} [\![\mathsf{succ}]\!]_u$.

(v) For the process in Example 2.2 (iii), let $\tau \stackrel{\text{def}}{=} ((\mathsf{Nat}^\circ\overline{\mathsf{Nat}^\bullet})^{?_\omega}\mathsf{Nat}^\bullet)^{!_\omega}$. Then we have $u : \tau \vdash_{\mathtt{I}} !u(xa).\overline{x}(zb)\,([\![1]\!]_z \mid b[\&_{i\in\mathbb{N}}\overline{a}\mathtt{in}_i]) \triangleright !_\omega u$.

(vi) A *copy-cat* $[x \to y]^\tau \stackrel{\text{def}}{=} !x(a).\overline{y}(b)b.\overline{a}$ copies all behaviour starting at one channel to those starting at another. Let $\tau = (()^{?_1})^{!_\omega}$ and $\Gamma = x : \tau \cdot y : \overline{\tau}$.

Then (1) $\Gamma{\cdot}a:()^{?_1}{\cdot}b:()^{!_1} \vdash_{\mathtt{I}} b.\overline{a} \triangleright ?_1 a \otimes !_1 b$, (2) $\Gamma{\cdot}a:()^{?_1} \vdash_{\mathtt{O}} \overline{y}(b)b.\overline{a} \triangleright ?_1 a \otimes ?_\omega y$, with $(?_1 a \otimes !_1 b)/b = ?_1 a$, and (3) $\Gamma \vdash_{\mathtt{I}} [x \to y]^\tau \triangleright !_\omega x \otimes ?_\omega y$.

Taking for example $(\boldsymbol{\nu}\, x)(P|[x \to y]^\tau)$ with $P \stackrel{\text{def}}{=} \overline{x}(a)a.\overline{c}$, we can check that all actions of P are copied from x to y (this does not include c which is emitted by P).

(vii) Let $\Delta = x:\langle\tau,\overline{\tau}\rangle{\cdot}y:\langle\tau,\overline{\tau}\rangle{\cdot}z:\langle\tau,\overline{\tau}\rangle$ and $\tau = (()^{?_1})^{!_\omega}$. Then we have:

- connection of two links: $\Gamma \vdash_{\mathtt{I}} [x \to y]^\tau \mid [y \to z]^\tau \triangleright !_\omega x \otimes !_\omega y \otimes ?_\omega z$ with $!_\omega y \otimes ?_\omega y = !_\omega y$.
- links to a shared resource at z: $\Gamma \vdash_{\mathtt{I}} [x \to z]^\tau \mid [y \to z]^\tau \triangleright !_\omega x \otimes !_\omega y \otimes ?_\omega z$ with $?_\omega z \otimes ?_\omega z = ?_\omega z$.

However, for any ϕ and environent, $[x \to z]^\tau \mid [x \to y]^\tau$ which represents non-deterministic forwarding is untypable since $!_\omega x \odot !_\omega x$ is undefined.

(viii) Let $\rho \stackrel{\text{def}}{=} ([\oplus_{i\in\mathbb{N}}]^{?_1})^{!_\omega}$ and $\Omega_z^\rho \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, xy)([x \to y]^\rho|[y \to x]^\rho|\overline{x}(a)\, a[\&_{i\in\mathbb{N}}\overline{z}\mathtt{in}_i])$. Then $u:\rho \vdash_{\mathtt{I}} !u(z).\Omega_z^\rho \triangleright !_\omega u$. Unlike $[\![\mathtt{n}]\!]_u$, it returns nothing when asked, representing the undefined.

3.6 Basic Syntactic Properties

The type discipline satisfies the following standard properties. In (i) and (ii) below, the partial order $\le$ on bases is generated from set inclusion and the rule $\Gamma \le \Delta \;\Rightarrow\; \Gamma \cdot x : \tau \le \Delta \cdot x : \langle\tau,\overline{\tau}\rangle$. The order on action types is simply set inclusion. In (iii) we let $\twoheadrightarrow \stackrel{\text{def}}{=} \equiv \cup (\longrightarrow)^*$.

Proposition 1. (i) (weakening) *If $\Delta \le \Gamma$ and $\Delta \vdash_\phi P$ then $\Gamma \vdash_\phi P$.*

(ii) (minimal type) *A typable process has a minimum base and action type. Further, if $\Gamma \vdash_\phi P$ and $\Delta \vdash_\psi P$ then $\phi = \psi$.*

(iii) (subject reduction) *If $\Gamma \vdash_\phi P$ and $P \twoheadrightarrow Q$ then $\Gamma \vdash_\phi Q$.*

We say an occurrence (subterm) in a process is an *active input* (resp. *active output*) if it is an input-prefixed (resp. output-prefixed) term which neither occurs under an input prefix nor has its subject bound by an output prefix.

Proposition 2. (i) *Let $\Gamma \vdash_\phi P \triangleright A \otimes px$ such that $p \in \{!_\omega, !_1\}$. Then there is an active input with free subject x in P.*

(ii) *Let $\Gamma \vdash_\phi P$. (1) If $\phi = \mathtt{I}$ there is no active output in P; (2) If $\phi = \mathtt{O}$ there is a unique active output in P; and (3) In both cases, two input processes never share the same name for their subjects, either bound or free.*

Corollary 1. (determinacy) *If $\Gamma \vdash_\phi P$ and $P \longrightarrow Q_i$ $(i = 1, 2)$ then $Q_1 \equiv Q_2$ and $\phi = \mathtt{O}$.*

3.7 Contextual Equality

Corollary 1 suggests 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 say x is *active* when it is the free subject of an active input/output, e.g. x in $(\boldsymbol{\nu}\,\vec{w})(\overline{x}(\vec{y})P \mid R)$ assuming $x \notin \vec{w}$. We first define:

$$\Gamma \vdash_\phi P \Downarrow_x \quad \stackrel{\text{def}}{\Leftrightarrow} \quad \Gamma \vdash_\phi P \twoheadrightarrow P' \text{ with } x \text{ active in } P' \text{ and } \Gamma \vdash_\phi P \triangleright A \otimes \mathbf{?}_1 x.$$

Choosing only affine output as observables induces a strictly coarser (pre-)congruence than if we had also included non-affine output ($\mathbf{?}_\omega$-actions are not considered since, intuitively, they do not affect the environment). We can now define a typed equality. Below, a relation over sequential processes is *typed* if it relates only processes with identical base, action type and IO-mode. A relation $\cong \supseteq \equiv$ is a *typed congruence* when it is a typed equivalence closed under typed contexts and, moreover, it satisfies: if $\Gamma \geq \Delta$ and $\Delta \vdash_\phi P \cong Q$ then $\Gamma \vdash_\phi P \cong Q$.

Definition 1. $\cong_{\mathsf{seq}}$ is the maximum typed congruence on sequential processes such that: if $\Gamma \vdash_\phi P \cong_{\mathsf{seq}} Q$ and $\Gamma \vdash_\phi P \Downarrow_x$ then $\Gamma \vdash_\phi Q \Downarrow_x$.

4 Analysis of Sequential Interactive Behaviour

4.1 Preamble

The purpose of the rest of the paper is to demonstrate that our typed processes precisely characterise the notion of functional sequentiality. By functional sequentiality we mean the class of computational dynamics that is exhibited by, for example, call-by-name and call-by-value PCF. Concretely we show, via an interpretation $u : \alpha^\circ \vdash_{\mathtt{I}} [\![M_i : \alpha]\!]_u$ that, for a PCF term $\vdash M_i : \alpha$ $(i = 1, 2)$, we have $M_1 \cong M_2$ iff $u : \alpha^\circ \vdash_{\mathtt{I}} [\![M_1 : \alpha]\!]_u \cong_{\mathsf{seq}} [\![M_2 : \alpha]\!]_u$. Here $\cong$ is the standard contextual equality on PCF-terms [14]. To this end we first introduce typed transitions to give a tractable account of processes interacting in typed contexts (the latter, like the former, must be input-output alternating). We then show that these transitions satisfy central properties of the intensional structures of games introduced in [23], namely visibility, bracketing and innocence. In particular, by innocence, any sequential process is representable by the corresponding innocent function up to redundant τ-actions. Further, the typed behaviour of a composite process $P|Q$ is completely determined by that of P and Q. Finally we show, *à la* game semantics, that any difference between typed processes in $\cong_{\mathsf{seq}}$ can be detected by sequential "tester" processes whose graphs as innocent functions are finite. But finite processes in (the interpretation of) PCF types are in turn representable by PCF-terms up to $\cong$, leading to the completeness of the interpretation. Since soundness is easy by operational correspondence, this establishes full abstraction. In the following we illustrate key steps of reasoning to reach finite definability.

Note on terminology. In this section, correspondence with typed transition and intensional structures of games is a central topic. Since there is some difference in terminology between process calculi and game semantics, we list the correspondence for reference.

O's Question (OQ)	[	$\mathbf{!}_\omega$	P's Answer (PA)	]	$\mathbf{?}_1$
P's Question (PQ)	(	$\mathbf{?}_\omega$	O's Answer (OA)	)	$\mathbf{!}_1$

Note that "O" is usually used to indicate "Opponent" in game semantics, which corresponds to *input* in our (process-algebraic) terminology. To avoid confusion, we shall consistently use "input" and "output" rather than "Opponent" and "Player".

4.2 Typed Transitions

Let $P \stackrel{\text{def}}{=} !x(yz).\overline{y}(c)c.\overline{z}$ and $Q \stackrel{\text{def}}{=} \overline{x}(yz)(!y(c).\overline{c}|z.\overline{w})$. Then $P|Q$ is well-typed, and we have:

$$\begin{aligned} P \mid Q &\longrightarrow (\boldsymbol{\nu}\, yz)((P|\overline{y}(c)c.\overline{z}) \mid (!y(c).\overline{c}|z.\overline{w})) \\ &\longrightarrow (\boldsymbol{\nu}\, yzc)((P|c.\overline{z}) \mid (\overline{c}|z.\overline{w}|!y(c).\overline{c})) \\ &\longrightarrow (\boldsymbol{\nu}\, yzc)((P|\overline{z}) \mid (z.\overline{w}|!y(c).\overline{c})) \\ &\longrightarrow (\boldsymbol{\nu}\, yzc)(P \mid (\overline{w}|!y(c).\overline{c})). \end{aligned}$$

This example suggests that input and output alternate in typed interaction. Indeed this is the only way sequential processes interact: if P does an output and Q does an input, then the derivatives of P and Q should now be in I-mode and O-mode, respectively. If they interact again, input and output are reversed. Typed transitions are built on this idea.

First we generate *untyped transitions* $P \stackrel{l}{\longrightarrow} Q$, with labels τ, $x(\vec{y})$, $\overline{x}(\vec{y})$, $x\mathtt{in}_i(\vec{y})$ and $\overline{x}\mathtt{in}_i(\vec{y})$ by the following rules.

(In)	$x(\vec{y}).P \xrightarrow{x(\vec{y})} P$	(Out)	$\overline{x}(\vec{z})P \xrightarrow{\overline{x}(\vec{z})} P$
(Bra)	$x[\&_{i\in I}(\vec{y}_i).P_i] \xrightarrow{x\,\mathtt{in}_i(\vec{y}_i)} P_i$	(Sel)	$\overline{x}\,\mathtt{in}_i(\vec{z})P \xrightarrow{\overline{x}\mathtt{in}_i(\vec{z})} P$

The rules for replicated input are defined similarly. The contextual rules are standard except for closure under asynchronous output (we omit the corresponding rule for branching).

$$(\text{Out-}\xi) \quad P \stackrel{l}{\longrightarrow} P' \text{ with } \mathsf{fn}(l) \cap \{\vec{y}\} = \emptyset \quad \Rightarrow \quad \overline{x}(\vec{y})\, P \stackrel{l}{\longrightarrow} \overline{x}(\vec{y})\, P'$$

To turn this into typed transitions, we first restrict the transitions of a process of mode O to only τ-actions and outputs since (as discussed at the outset) the interacting party should always be in I-mode. Secondly, if a process has $\bot x$ (resp. $\mathbf{!}_\omega x$) in its action type, then both input and output at x (resp. output at x) are excluded since, again, such actions can never be observed in a typed context. It is easy to check that sequential processes are closed under the restricted transition relation. The resulting typed transitions are written:

$$\Gamma \vdash_\phi P \stackrel{l}{\longrightarrow} \Gamma \cdot \vec{y}\!:\!\vec{\tau} \vdash_\psi Q$$

where $\vec{y}\!:\!\vec{\tau}$ assigns names introduced in l as prescribed by Γ. Typed τ-transitions coincide with untyped τ-transitions, hence typing of transitions restricts only observability of actions, not computation. Basic properties of transitions follow.

Proposition 3. (i) (IO-alternation) *Let* $\Gamma \vdash_\phi P \stackrel{l_1 l_2}{\Longrightarrow} \Delta \vdash_\psi Q$. *Then* (1) $\phi = \psi$, *and* (2) l_1 *is input iff* l_2 *is output and vice versa.*

(ii) (determinacy) *If* $\Gamma \vdash_\phi P \stackrel{l}{\longrightarrow} \Delta \vdash_\psi Q_i$ $(i = 1, 2)$ *then* $Q_1 \equiv_\alpha Q_2$.

(iii) (unique output) *If* $\Gamma \vdash_{\mathtt{O}} P \stackrel{l_i}{\longrightarrow} P_i$ $(i = 1, 2)$ *then* $l_1 \equiv_\alpha l_2$.

As an example of typed transitions, let $\tau \stackrel{\text{def}}{=} (\overline{\mathsf{Nat}^\circ \mathsf{Nat}^\bullet})^{!_\omega}$. Then, using the notation in Examples 2.2 (ii), we have:

$$\begin{array}{rl}
x : \tau \vdash_{\mathtt{I}} [\![\mathtt{succ}]\!]_u \stackrel{u(ya)}{\longrightarrow} & u : \tau, y : \overline{\mathsf{Nat}^\circ}, a : \mathsf{Nat}^\bullet \vdash_{\mathtt{O}} \overline{y}(b)\, b[\&_{i \in \mathbb{N}} \overline{a}\mathtt{in}_{i+1}] \mid [\![\mathtt{succ}]\!]_u \\
\stackrel{\overline{y}(b)}{\longrightarrow} & u : \tau, y : \overline{\mathsf{Nat}^\circ}, a : \mathsf{Nat}^\bullet, b : \overline{\mathsf{Nat}^\bullet} \vdash_{\mathtt{I}} b[\&_{i \in \mathbb{N}} \overline{a}\mathtt{in}_{i+1}] \mid [\![\mathtt{succ}]\!]_u \\
\stackrel{b\mathtt{in}_j}{\longrightarrow} & u : \tau, y : \overline{\mathsf{Nat}^\circ}, a : \mathsf{Nat}^\bullet, b : \overline{\mathsf{Nat}^\bullet} \vdash_{\mathtt{O}} \overline{a}\mathtt{in}_{j+1} \mid [\![\mathtt{succ}]\!]_u \\
\stackrel{\overline{a}\mathtt{in}_{j+1}}{\longrightarrow} & u : \tau, y : \overline{\mathsf{Nat}^\circ}, a : \mathsf{Nat}^\bullet, b : \overline{\mathsf{Nat}^\bullet} \vdash_{\mathtt{I}} \mathbf{0} \mid [\![\mathtt{succ}]\!]_u
\end{array}$$

4.3 Visibility and Well-Bracketing

Let us write $\Gamma \vdash_\phi P \stackrel{l_1..l_n}{\Longrightarrow} \Delta \vdash_\psi Q$ if $\Gamma \vdash_\phi P \stackrel{\tau^*}{\longrightarrow}\stackrel{l_1}{\longrightarrow}\stackrel{\tau^*}{\longrightarrow} \ldots \stackrel{\tau^*}{\longrightarrow}\stackrel{l_n}{\longrightarrow}\stackrel{\tau^*}{\longrightarrow} \Delta \vdash_\psi Q$ with $l_i \neq \tau$ $(0 \leq i \leq n)$. For $i \lneq j$, we write $l_i \curvearrowright_{\mathtt{b}} l_j$ (read: l_i binds l_j) when the subject of l_j is bound by l_i (e.g. $x(y) \curvearrowright_{\mathtt{b}} \overline{y}\mathtt{in}_n$). Clearly, in typable processes, input only binds output and vice versa. $\curvearrowright_{\mathtt{b}}$ corresponds to justification of moves in games. Now we define the notion of *views* as follows. $\ulcorner l_1 ... l_n \urcorner^{\mathtt{O}}$ is defined first, with $s, t, \ldots$ ranging over sequences of labels.

$$\begin{array}{lll}
\ulcorner \epsilon \urcorner^{\mathtt{O}} & = \emptyset & \\
\ulcorner s \cdot l_n \urcorner^{\mathtt{O}} & = \{n\} \cup \ulcorner s \urcorner^{\mathtt{O}} & l_n \text{ output} \\
\ulcorner s \cdot l_n \urcorner^{\mathtt{O}} & = \{n\} & l_n \text{ input}, \forall i. i \not\curvearrowright_{\mathtt{b}} n \\
\ulcorner s_1 \cdot l_i \cdot s_2 \cdot l_n \urcorner^{\mathtt{O}} & = \{i, n\} \cup \ulcorner s_1 \urcorner^{\mathtt{O}} & l_n \text{ input}, i \curvearrowright_{\mathtt{b}} n
\end{array}$$

Input view, denoted $\ulcorner s \urcorner^{\mathtt{I}}$, is defined dually by exchanging Os and Is as well as input and output. We often confuse $\ulcorner s \urcorner^{\mathtt{O}}$ and $\ulcorner s \urcorner^{\mathtt{I}}$ with the corresponding sequences. We now define:

Definition 2. (visibility) Let $\Gamma \vdash_\phi P \stackrel{s}{\Longrightarrow} \Delta \vdash_\psi Q$. Then $s = l_1 \cdots l_n$ is *input-visible* if whenever l_{i+1} is input such that $l_j \curvearrowright_{\mathtt{b}} l_i$, we have $j \in \ulcorner l_1 \cdots l_i \urcorner^{\mathtt{I}}$. Dually we define output-visibility. We say $\Gamma \vdash_\phi P$ is *visible* if whenever $\Gamma \vdash_\phi P \stackrel{s}{\Longrightarrow}$ and s is input-visible then it is output-visible.

The first key result follows.

Proposition 4. $\Gamma \vdash_\phi P$ *is visible.*

The proof proceeds by first establishing that it suffices to consider only *well-knit* traces where the only free input (if any) is an initial one. We then use induction on the typing rules to show that well-knit traces are visible. The only non-trivial cases are input prefixes and parallel composition. For input prefixes we use Proposition 3.2 (i). For parallel composition, we use *composite transitions* of $\Gamma \vdash_\phi P|Q$ which record the transitions of P and Q contributing to the those of $P|Q$ as a whole. Such transitions can be written in a matrix with four rows. For example, a composite transition of a sequential process (omitting types) $!x(c).\overline{y}(e).e[\&_{i\in\mathbb{N}}\overline{c}\mathtt{in}_{i+1}] \mid !y(e).\overline{z}(e').e'[\&_{i\in\mathbb{N}}\overline{e}\mathtt{in}_i]$ is given as follows, writing P and Q for the first and second components of parallel composition:

$$\begin{array}{lllllll}
P\text{-visible}: & x(c) & & & & & \overline{c}\mathtt{in}_3 \\
P\text{-}\tau: & & \overline{y}(e) & & & e\mathtt{in}_2 & \\
Q\text{-}\tau: & & y(e) & & & \overline{e}\mathtt{in}_2 & \\
Q\text{-visible}: & & & \overline{z}(e') & e'\mathtt{in}_2 & &
\end{array}$$

If such a sequence is well-knit and input-visible in its observable part (i.e. the first and fourth rows), then it satisfies the *switching condition* [3, 23], i.e. the action of P (resp. Q) moving from one row to another is always an output. To establish this we use IO-modes of derivatives and input-visibility. Then output visibility is immediate using standard game semantics technique [20, 23, 26].

Next, *well-bracketing* [3, 23] says that later questions are always answered first, i.e. nesting of bracketing is always properly matched. Below, following the table in §4.1, we call actions of mode $\mathbf{!}_\omega$ and $\mathbf{?}_\omega$ *questions* while actions of mode $\mathbf{!}_1$ and $\mathbf{?}_1$ are *answers*.

Definition 3. Let $\Gamma \vdash_\phi P \stackrel{s}{\Longrightarrow} \Delta \vdash_\psi Q$ be input-visible. Then s is *well-bracketing* if, whenever $s' = s_0 \cdot l_i \cdot s_1 \cdot l_j$ for a prefix s' of s is such that (1) l_i is a question and (2) l_j is an answer free in $s_1 \cdot l_j$, we have $l_i \curvearrowright_{\mathtt{b}} l_j$.

Now we say $\Gamma \vdash_\phi P$ is *well-bracketing* if whenever $\Gamma \vdash_\phi P \stackrel{sl}{\Longrightarrow}$, s is well-bracketing and l is output, then sl is well-bracketing. Then we have:

Proposition 5. *$\Gamma \vdash_\phi P$ is well-bracketing.*

The proof uses induction on typing rules, noting that it suffices to consider well-knit sequences. The non-trivial cases are input by $\mathbf{!}_\omega$ and parallel composition. The former holds because a $\mathbf{!}_\omega$-prefix does not suppress a free output with action mode $\mathbf{?}_1$, while the latter follows from the switching condition [20, 23, 26].

Definition 4. (legal trace) Let $\Gamma \vdash_\phi P \stackrel{s}{\Longrightarrow}$. Then s is *legal* if it is both input-visible and well-bracketing.

4.4 Innocence

Innocence [23] says that a process does the same action whenever it is in the same "context", i.e. in the same output-view. To establish innocence of traces of typed processes we begin with the following lemma, proved by analysis of possible redexes relying on the shape of the syntax imposed by the type discipline.

Lemma 1. (permutation) *Let $\Gamma \vdash_{\mathtt{I}} P \stackrel{l_1 l_2 l_3 l_4}{\Longrightarrow} \Delta \vdash_{\mathtt{I}} Q$ such that $l_1 \not\curvearrowright_{\mathtt{b}} l_4$ and $l_2 \not\curvearrowright_{\mathtt{b}} l_3$. Then $\Gamma \vdash_{\mathtt{I}} P \stackrel{l_3 l_4 l_1 l_2}{\Longrightarrow} \Delta \vdash_{\mathtt{I}} Q$.*

By the above lemma and visibility, we can transform any transition of form $\Gamma \vdash_\phi P \stackrel{sl}{\Longrightarrow}$, with l output, to $\Gamma \vdash_\phi P \stackrel{tl}{\Longrightarrow}$ where $t = \ulcorner s \urcorner^{\mathsf{O}}$. Since an output is always unique (cf. Proposition 2 (ii)), we can now conclude:

Proposition 6. (innocence) *Let* $\Gamma \vdash_\psi P \stackrel{s_i l_i}{\Longrightarrow}$ $(i = 1, 2)$ *such that:* (1) *both sequences are legal;* (2) *both* l_1 *and* l_2 *are output; and* (3) $\ulcorner s_1 \urcorner^{\mathsf{O}} \equiv_\alpha \ulcorner s_2 \urcorner^{\mathsf{O}}$. *Then we have* $\ulcorner s_1 \urcorner^{\mathsf{O}} \cdot l_1 \equiv_\alpha \ulcorner s_2 \urcorner^{\mathsf{O}} \cdot l_2$.

Note that *contingency completeness* in [23] corresponds to the property that any legal trace ending in an output has a legal extension ending with an input, which is immediate by Proposition 3.2 (i) and typability of transitions. Therefore, up to redundant τ-actions, a sequential process is precisely characterised by the function mapping a set of output views to next actions. This is the *innocent function representation* of a sequential process.

It is now easy to see that well-knit legal traces of $\Gamma \vdash_\phi P_1 | P_2$ are uniquely determined by those of $\Gamma \vdash_{\psi_i} P_i$ $(i = 1, 2)$ in the same way that innocent strategies are composed in the appropriate category of games [23].

4.5 Factoring Observables

An important property of $\cong_{\mathsf{seq}}$ is that any violation of $\cong_{\mathsf{seq}}$ can be detected by a tester process which is finite in the sense that the cardinality of the graph of its induced innocent function is finite. In particular, for our full abstraction result, we need finite processes which are type-wise translatable to (the interpretation of) PCF terms. To this end, we first show that the congruence $\cong_{\mathsf{seq}}$ can be obtained by only closing terms under $|$, given an appropriate base (*Context Lemma*, cf. [27]). Then we use the following result to unfold replication.

Proposition 7. (open replication) *Assume* $\Gamma \vdash_\psi P_1 \mid P_2 \mid R$ *where* R *is a replication with subject* x. *Then* $\Gamma \vdash_\psi P_1 \mid P_2 \mid R \cong_{\mathsf{seq}} (P_1 \mid R) \mid (\boldsymbol{\nu}\, x)(P_2 \mid R)$.

The proof of Proposition 7 uses a bisimulation induced by the typed transition (which stays within $\cong$). We can then establish the following proposition where $\overline{\Gamma}$ denotes the result of dualising each type occurring in Γ.

Proposition 8. (finite testability) *Assume* $\Gamma \vdash_{\mathtt{I}} P_i \triangleright \mathbf{?}_\omega y_1 \otimes \cdots \mathbf{?}_\omega y_n \otimes \mathbf{!}_\omega z$ $(i = 1, 2)$ *such that* $\mathsf{fn}(\Gamma) = \{\vec{y}, z\}$. *Then* $\Gamma \vdash_{\mathtt{I}} P_1 \not\cong_{\mathsf{seq}} P_2$ *iff there exist finite* $\overline{\Gamma} \vdash_{\mathtt{I}} R_j \triangleright \mathbf{!}_\omega y_j$ $(1 \leq j \leq n)$ *and a finite* $\overline{\Gamma} \cdot x : \mathsf{Nat}^\bullet \vdash_{\mathtt{O}} S \triangleright \mathbf{?}_\omega z \otimes \mathbf{?}_1 x$ *such that* $(\Pi_j R_j | P_1 | S) \Downarrow_x$ *and* $(\Pi_j R_j | P_2 | S) \not\Downarrow_x$, *or its symmetric case.*

Towards the proof, we first take, by the Context Lemma mentioned above, a tester of form $\overline{\Gamma} \cdot x : \mathsf{Nat}^\bullet \vdash T'$ which, when composed with P_i, gives different observables. We then make, using Proposition 7, all shared replicated processes private to their "clients". This gives processes R'_i and S' which have the same types as R_i and S above. Finally, the shapes of types allow to consider processes R_i, P_i and S (to be precise by turning S to $u(x).S$) as strategies in games. We

$$
\begin{array}{ll}
\textbf{(Type)} \quad \mathsf{Nat}^\bullet \stackrel{\text{def}}{=} [\oplus_{i\in\mathbb{N}}]^{?_1} & [\alpha_1..\alpha_{n-1}\mathsf{Nat}]^\circ \stackrel{\text{def}}{=} (\overline{\alpha_1^\circ}..\overline{\alpha_{n-1}^\circ}\mathsf{Nat}^\bullet)^{!_\omega} \\
\textbf{(Base)} \quad \emptyset^\circ \stackrel{\text{def}}{=} \emptyset & (E\cdot x:\alpha)^\circ \stackrel{\text{def}}{=} E^\circ \cdot x : \overline{\alpha^\circ}
\end{array}
$$

(Terms) Below we set $\beta = [\alpha_1..\alpha_{n-1}\mathsf{Nat}]$.

$$
\begin{array}{l}
[\![x : \alpha]\!]_u \stackrel{\text{def}}{=} [u \to x]^{\alpha^\circ} \\
[\![\lambda x_0 : \alpha_0.M : \alpha_0 \Rightarrow \beta]\!]_u \stackrel{\text{def}}{=} !\, u(x_0 x_1..x_{n-1}z).(\boldsymbol{\nu}\, u')([\![M]\!]_{u'} \mid \mathsf{Arg}\langle u' x_1...x_{n-1}z\rangle^\beta) \\
[\![MN : \beta]\!]_u \stackrel{\text{def}}{=} !\, u(x_1..x_{n-1}z).(\boldsymbol{\nu}\, u' x_0)([\![M : \alpha \Rightarrow \beta]\!]_{u'} \mid [\![N : \alpha]\!]_{x_0} \mid \mathsf{Arg}\langle u' x_0...x_{n-1}z\rangle^{\alpha\Rightarrow\beta}) \\
[\![\mathtt{n} : \mathsf{Nat}]\!]_u \stackrel{\text{def}}{=} !\, u(z).\overline{z}\mathtt{in}_n \\
[\![\mathtt{succ}(M) : \mathsf{Nat}]\!]_u \stackrel{\text{def}}{=} !\, u(z).(\boldsymbol{\nu}\, x)([\![M]\!]_x \mid \overline{x}(y)y[\&_{n\in\mathbb{N}}\, \overline{z}\mathtt{in}_{n+1}]) \\
[\![\mathtt{pred}(M) : \mathsf{Nat}]\!]_u \stackrel{\text{def}}{=} !\, u(z).(\boldsymbol{\nu}\, x)([\![M]\!]_x \mid \overline{x}(y)y[\&_{n\in\mathbb{N}}\, \overline{z}\mathtt{in}_{n-1}]) \\
[\![\mathtt{ifzero}\ M\ \mathtt{then}\ N\ \mathtt{else}\ L : \beta]\!]_u \\
\qquad \stackrel{\text{def}}{=} !\, u(x_1..x_{n-1}z).(\boldsymbol{\nu}\, m)([\![M]\!]_m \mid \overline{m}(z')z'[\&_i(\boldsymbol{\nu}\, u')(P_i \mid \mathsf{Arg}\langle u' x_1..x_{n-1}z\rangle^\beta)]) \\
\qquad\qquad \text{where } P_0 \stackrel{\text{def}}{=} [\![N]\!]_{u'} \text{ else } P_i \stackrel{\text{def}}{=} [\![L]\!]_{u'}. \\
[\![\mu x : \alpha.M : \alpha]\!]_u \stackrel{\text{def}}{=} (\boldsymbol{\nu}\, m)([u \to m]^{\alpha^\circ} \mid [\![M : \alpha]\!]_m \mid [x \to m]^{\alpha^\circ}) \\
\mathsf{Arg}\langle x\vec{y}z\rangle^{[\vec{\alpha}\mathsf{Nat}]} \stackrel{\text{def}}{=} \overline{x}(\vec{y'}z')(\Pi_i[y'_i \to y_i]^{\alpha_i^\circ} \mid [z' \to z]^{\overline{\mathsf{Nat}^\bullet}})
\end{array}
$$

Fig. 2. Encoding of PCF

can now appeal to finite testability in games, cf. [23], from which, by retranslating finite innocent strategies to finite processes, we conclude that finite testers suffice. Alternatively we can directly reason at the level of the π-calculus and its typed transitions, showing that any behaviour characterised by a finite innocent function (which is enough for testability) is realisable by (typable) syntactic processes [5, 38].

5 Full Abstraction

5.1 Interpretation

We consider PCF with a single base type, Nat, without loss of generality. Let $\alpha ::= \mathsf{Nat} \mid \alpha \Rightarrow \beta$. We write $[\alpha_1..\alpha_n\mathsf{Nat}]$ $(n \le 0)$ for $\alpha_1 \Rightarrow (...(\alpha_n \Rightarrow \mathsf{Nat})..)$. Now the syntax of PCF terms are given by:

$$
\begin{array}{rcl}
M & ::= & x \mid \lambda x : \alpha.M \mid MN \mid n \mid \mathtt{succ}(M) \mid \mathtt{pred}(M) \\
& \mid & \mathtt{ifzero}\ M\ \mathtt{then}\ N\ \mathtt{else}\ L \mid \mu x : \alpha.M
\end{array}
$$

We omit operational semantics and the typing rules [14]. The mappings from PCF types and terms to π-types and terms, which are due to Hyland and Ong [24], are given in Figure 2. Copy-cat processes are given by $[x \to x']^{[\&_i(\vec{\tau}_i)]^{!_1}} \stackrel{\text{def}}{=} x[\&_i(\vec{y}_i).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$ and for replicated types: $[x \to x']^{[\&_i(\vec{\tau}_i)]^{!_\omega}}$

$\stackrel{\text{def}}{=} !x[\&_i(\vec{y_i}).\overline{x'}\mathtt{in}_i(\vec{y'}_i)\Pi_{ij}[y'_{ij} \to y_{ij}]^{\overline{\tau_{ij}}}]$. Copy-cats for unary types are special cases where the indexing sets are singletons. The interpretation of $[\alpha_1..\alpha_n\mathsf{Nat}]$ says a process, when asked for its value, asks back questions at types $\alpha_1,..,\alpha_n$, receives the results to these questions, and finally returns a natural number as the answer to the initial question.

5.2 Soundness

This is by the standard computational adequacy [27], which is proved by both-way operational correspondence, cf. [28]. Below let $\perp^{\mathsf{Nat}^\circ} \stackrel{\text{def}}{=} !\,u(z).\Omega_z^{\mathsf{Nat}^\circ}$ where Ω_u^τ is given in Example 3.5 (v).

Theorem 1. (computational adequacy) $M\!:\!\mathsf{Nat}\Downarrow$ *iff* $[\![M:\mathsf{Nat}]\!]_u \not\cong_{\mathsf{seq}} \perp^{\mathsf{Nat}^\circ}$.

Corollary 2. (soundness) $E \vdash M \cong N : \alpha$ *if* $E^\circ \cdot u\!:\!\alpha^\circ \vdash_{\mathtt{I}} [\![M\!:\!\alpha]\!]_u \cong_{\mathsf{seq}} [\![N\!:\!\alpha]\!]_u$

5.3 Completeness

Assume P is typed under (the interpretation of) a PCF-type and, moreover, it is finite, i.e. is representable by an innocent function. By [3, 23] or by a direct syntactic transformation, P can be mapped into a so-called *finite canonical-form*, which in turn is easily transformed to a standard PCF term without changing meaning in its interpretation up to $\cong_{\mathsf{seq}}$. Thus we obtain:

Theorem 2. (finite definability) *Let* $E^\circ\!\cdot\!u\!:\!\alpha^\circ \vdash_{\mathtt{I}} P \triangleright \mathbf{!}_\omega u$ *be finite. Then* $E^\circ\!\cdot\!u\!:\!\alpha^\circ \vdash [\![M\!:\!\alpha]\!]_u \cong_{\mathsf{seq}} P$ *for some* M.

This result indicates that, in essence, *only sequential functional behaviour inhabits each type.* Now suppose $\vdash M_1 \cong M_2 : \alpha$ but $u:\mathsf{Nat}^\circ \vdash_{\mathtt{I}} [\![M_1]\!]_u \not\cong_{\mathsf{seq}} [\![M_2]\!]_u$. Then the latter's difference is detectable by finite processes (Proposition 8). By Theorem 2 we can consider these finite testers as interpretations of PCF-terms so that we know, for example, $[\![C[M_1]:\mathsf{Nat}]\!]\Downarrow$ and $[\![C[M_2]:\mathsf{Nat}]\!]\Uparrow$. But this means, by Theorem 1, $C[M_1]:\mathsf{Nat}\Downarrow$ and $C[M_2]:\mathsf{Nat}\Uparrow$, contradicting our assumption. We have now reached the main result of the paper.

Theorem 3. (full abstraction) $E^\circ \cdot u\!:\!\alpha^\circ \vdash [\![M_1:\alpha]\!]_u \cong_{\mathsf{seq}} [\![M_2:\alpha]\!]_u$ *if and only if* $E \vdash M_1 \cong M_2 : \alpha$.

By replacing $\cong_{\mathsf{seq}}$ and $\cong$ with the corresponding precongruences, we similarly obtain inequational full abstraction. It is also notable that a fully abstract interpretation of call-by-value sequentiality is easily gotten by simply changing the interpretation of types. The following comes from [20]. (1) $\mathsf{Nat}^\star \stackrel{\text{def}}{=} [\oplus_{i\in\mathbb{N}}]^{\mathbf{?}_1}$ and $(A \Rightarrow B)^\star \stackrel{\text{def}}{=} ((A \Rightarrow B)^\diamond)^{\mathbf{?}_1}$; (2) $(\mathsf{Nat} \Rightarrow B)^\diamond \stackrel{\text{def}}{=} [\&_{i\in\mathbb{N}}B^\star]^{\mathbf{!}_\omega}$ and, when $A \neq \mathsf{Nat}$, $(A \Rightarrow B)^\diamond \stackrel{\text{def}}{=} (\overline{A^\diamond}B^\star)^{\mathbf{!}_\omega}$. For example, $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ is interpreted as $([\&_{i\in\mathbb{N}}[\oplus_{i\in\mathbb{N}}]^{\mathbf{?}_1}]^{\mathbf{!}_\omega})^{\mathbf{?}_1}$, where the function first signals itself, receives a natural number, then returns the result. Again the only inhabitants of $A^\diamond$ are easily the encodings of call-by-value PCF terms, from which we obtain full abstraction. The result also extends to recursive types [11]. Further, another change in interpretation of types allows us to fully abstractly capture the semantics of call-by-name PCF with observability at higher-order types. These results may suggest the power and flexibility of the present framework for the semantic analysis of sequentiality.

References

1. Abramsky, S., Computational interpretation of linear logic. *TCS*, Vol. 111, 1993.
2. Abramsky, S., Honda, K. and McCusker, G., A Fully Abstract Game Semantics for General References. *LICS*, 334-344, IEEE, 1998.
3. Abramsky, S., Jagadeesan, R. and Malacaria, P., Full Abstraction for PCF. *Info. & Comp.*, Vol. 163, 2000.
4. Berger, M. Honda, K. and N. Yoshida. *Sequentiality and the π-Calculus.* To appear as a QMW DCS Technical Report, 2001.
5. Berger, M. Honda, K. and N. Yoshida. *Genericity in the π-Calculus.* To appear as a QMW DCS Technical Report, 2001.
6. Berry, G. and Curien, P. L., Sequential algorithms on concrete data structures *TCS*, 20(3), 265-321, North-Holland, 1982.
7. Boreale, M. and Sangiorgi, D., Some congruence properties for π-calculus bisimilarities, *TCS*, 198, 159–176, 1998.
8. Boudol, G., Asynchrony and the pi-calculus, INRIA Research Report 1702, 1992.
9. Boudol, G., The pi-calculus in direct style, *POPL'97*, 228–241, ACM, 1997.
10. Curien, P. L., Sequentiality and full abstraction. Proc. of *Application of Categories in Computer Science*, LNM 177, 86–94, Cambridge Press, 1995.
11. Fiore, M. and Honda, K., Recursive Types in Games: axiomatics and process representation, *LICS'98*, 345-356, IEEE, 1998.
12. Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions, *ESOP'99*, LNCS 1576, 74–90, Springer, 1999.
13. Girard, J.-Y., Linear Logic, *TCS*, Vol. 50, 1–102, 1987.
14. Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.
15. Honda, K., Types for Dyadic Interaction. *CONCUR'93*, LNCS 715, 509-523, 1993.
16. Honda, K., Composing Processes, *POPL'96*, 344-357, ACM, 1996.
17. Honda, K., Kubo, M. and Vasconcelos, V., Language Primitives and Type Discipline for Structured Communication-Based Programming. *ESOP'98*, LNCS 1381, 122–138. Springer-Verlag, 1998.
18. Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. *ECOOP'91*, LNCS 512, 133–147, Springer-Verlag 1991.
19. Honda, K. Vasconcelos, V., and Yoshida, N. Secure Information Flow as Typed Process Behaviour, *ESOP '99*, LNCS 1782, 180–199, Springer-Verlag, 2000.
20. Honda, K. and Yoshida, N. Game-theoretic analysis of call-by-value computation. *TCS* Vol. 221 (1999), 393–456, North-Holland, 1999.
21. Kobayashi, N., A partially deadlock-free typed process calculus, *ACM TOPLAS*, Vol. 20, No. 2, 436–482, 1998.
22. Kobayashi, N., Pierce, B., and Turner, D., Linear Types and π-calculus, *POPL'96*, 358–371, ACM Press, 1996.

23. Hyland, M. and Ong, L., On Full Abstraction for PCF: I, II and III. 130 pages, 1994. To appear in *Info. & Comp.*
24. Hyland, M. and Ong, L., Pi-calculus, dialogue games and PCF, *FPCA'95*, ACM, 1995.
25. Laird, J., Full abstraction for functional languages with control, *LICS'97*, IEEE, 1997.
26. McCusker, G., Games and Full Abstraction for FPC. *LICS'96*, IEEE, 1996.
27. Milner, R., Fully abstract models of typed lambda calculi. *TCS*, 4:1–22, 1977.
28. Milner, R., Functions as Processes. *MSCS*, 2(2), 119–146, CUP, 1992.
29. Milner, R., Polyadic π-Calculus: a tutorial. *Proceedings of the International Summer School on Logic Algebra of Specification*, Marktoberdorf, 1992.
30. Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *LICS'93*, 187–215, IEEE, 1993.
31. Quaglia, P. and Walker, D., On Synchronous and Asynchronous Mobile Processes, *FoSSaCS 00*, LNCS 1784, 283–296, Springer, 2000.
32. Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms.* Ph.D. Thesis, University of Edinburgh, 1992.
33. Sangiorgi, D. π-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(2):235–271, North-Holland, 1996.
34. Sangiorgi, D., The name discipline of uniform receptiveness, *ICALP'97*, LNCS 1256, 303–313, Springer, 1997.
35. Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, 100–117. Springer, 1994.
36. Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, 524–538, Springer-Verlag, 1993.
37. Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, 371–387, Springer-Verlag, December, 1996.
38. N. Yoshida., Berger, M. and Honda, K., *Strong Normalisation in the π-Calculus*, To appear as a MCS Technical Report, University of Leicester, 2001.

A Typing Rules for Branching

$$(\mathsf{Bra}^{!_1})\quad (C_i/\vec{y}_i = \mathbf{?}A) \qquad\qquad (\mathsf{Sel}^{?_1})\quad (C_i/\vec{y}_i = A \asymp \mathbf{?}_1 x)$$

$$\frac{\Gamma \vdash x : [\&_{i\in I}\vec{\tau}_i]^{!_1} \qquad \Gamma\cdot\vec{y}_i : \vec{\tau}_i \vdash_{\mathtt{O}} P_i \triangleright C_i^{-x}}{\Gamma \vdash_{\mathtt{I}} x[\&_{i\in I}(\vec{y}_i : \vec{\tau}_i).P_i] \triangleright A \otimes \mathbf{!}_1 x} \qquad \frac{\Gamma \vdash x : [\oplus_{i\in I}\,\vec{\tau}_i]^{?_1} \qquad \Gamma\cdot\vec{y}_i : \vec{\tau}_i \vdash_{\mathtt{I}} P \triangleright C}{\Gamma \vdash_{\mathtt{O}} \overline{x}\mathtt{in}(\vec{y}_i : \vec{\tau}_i)P \triangleright A \odot \mathbf{?}_1 x}$$

(Bra$^{!_\omega}$) and (Sel$^{?_\omega}$) are similarly defined.