Full Abstraction in a Subtyped pi-Calculus with Linear Types

Romain Demangeon and Kohei Honda

Queen Mary, University of London

Abstract. We introduce a concise pi-calculus with directed choices and develop a theory of subtyping. Built on a simple behavioural intuition, the calculus offers exact semantic analysis of the extant notions of subtyping in functional programming languages and session-based programming languages. After illustrating the idea of subtyping through examples, we show type-directed embeddings of two known subtyped calculi, one for functions and another for session-based communications. In both cases, the behavioural content of the original subtyping is precisely captured in the fine-grained subtyping theory in the pi-calculus. We then establish full abstraction of these embeddings with respect to their standard semantics, Morris's contextual congruence in the case of the functional calculus and testing equivalence for the concurrent calculus. For the full abstraction of the embedding of the session-based calculus, we introduce a new proof method centring on non-deterministic computational adequacy and definability. Partially suggested by a technique used by Quaglia and Walker for their full abstraction result, the new proof method extends the framework used in game-based semantics to the May/Must equivalences, giving a uniform proof method for both deterministic and non-deterministic languages.

1 Introduction

A subtyping is a form of polymorphism where we can assign to a program a type which is more inclusive than the original type of the program, called subsumption. This notion of inclusion forms a partial order on types, where "more inclusive" may most simply be interpreted as having more inhabitants satisfying the type specification. In the standard subtyping theories, this inclusiveness is structurally calculable from the construction of types, such as through the well-known variance rule for arrow types, records and variants, cf. [2]. The notion of subtyping plays a key role in the practice of programming languages [20].

In this paper we study a simple theory of subtyping for interacting processes and show that it subsumes extant notions of subtyping in programming languages through encoding. We first introduce a concise pi-calculus with directed choices and linear types, and develop a theory of subtyping purely based on these choices. The resulting calculus is called $\pi^{\{1,\overline{1},\omega\}}$ for brevity. After illustrating a simple behavioural intuition behind the subtyping theory through examples, we show that the calculus offers exact semantic analysis of the existing notions of subtyping in functional programming languages and session-based programming languages. First we introduce type-directed embeddings of two known subtyped calculi, one for functions [22] (which uses Milner's encoding [16] but with novelty in the treatment of sums) and another for session-based communications [25,13]. In both cases, the behavioural content of the original subtyping is precisely captured in the fine-grained subtyping theory in the pi-calculus.

We then establish full abstraction of each embedding with respect to a standard semantics of the target calculus, Morris's contextual congruence in the case of the functional calculus and the testing/failure equivalence for the concurrent calculus. The full abstraction, together with the concision of the encoding, may offer an exact interactional elucidation of these existing subtyping notions. For the full abstraction of the embedding of the session-based calculus, we introduce a new proof method centring on non-deterministic computational adequacy and definability. Partially suggested by a technique used by Quaglia and Walker for their full abstraction of the polyadic synchronous π-calculus in the monadic asynchronous π-calculus [23], as well as by those from game-based semantics [15], the new proof method is uniform (the method for non-deterministic languages specialises the one for the traditional, deterministic languages), has generality (the type structure of a meta calculus, here the linear subtyped π-calculus, can be disjoint from that of an object language, here the λ-calculus and the session calculus), and is generic (as far as some key properties hold for adequacy and definability, it automatically gives full abstraction).

We summarise some of the main technical contributions of the work.

1. A concise subtyped π-calculus with linear typing ($\pi^{\{1,\overline{1},\omega\}}$), giving rise to a simple and general theory of subtyping, whose key properties we establish.
2. Type-directed embeddings of a call-by-value λ-calculus with record and variant subtyping and a concurrent calculus with session subtyping in $\pi^{\{1,\overline{1},\omega\}}$, obtaining full abstractions. For the latter we use a new proof method centring on non-deterministic computational adequacy and definability.

To our knowledge, this is the first full abstraction results for these subtyped calculi in the π-calculus: further, the corresponding results have not been known in game-based semantics (which is in close corresponding with the π-calculus, cf. [12,14,9]). The semantically sound encoding of the session calculus itself looks new. In another vein, this may be the first full abstraction result for the interactional representation of a non-trivial, fully non-deterministic concurrent calculus.

In the rest of the paper, Section 2 introduces $\pi^{\{1,\overline{1},\omega\}}$ and develops the theory of subtyping, illustrating its intuition through examples and establishing its key properties. Section 3 fully abstractly embeds $\lambda^{\Pi,\Sigma,\sqsubseteq}$ in $\pi^{\{1,\overline{1},\omega\}}$. Section 4 fully abstractly embeds the session-calculus from [13] with subtyping in $\pi^{\{1,\overline{1},\omega\}}$. Section 5 discusses related works. The full proofs can be found in [7].

2 A Concise, Subtyped π-Calculus

In the following, we use the shortcut $\widetilde{e}$ for a vector $(e_1, \ldots, e_k)$ for some integer k.

Processes and Reduction. We use $a, b, c, \ldots, u, v, \ldots, x, y$ to denote names, or channels, $X, Y, \ldots$ for agent variables, and $l, \ldots$ for labels. Syntax for processes of $\pi^{\{1,\overline{1},\omega\}}$, our linear-affine π-calculus (cf. [3,28]), is given by the following grammar.

$$P ::= (P \mid P) \mid \mathbf{0} \mid X\langle\widetilde{v}\rangle \mid (\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle \mid \overline{u} \oplus^m l\langle\widetilde{v}\rangle \mid u\&^m_{i\in I}\{l_i(\widetilde{x_i}).P_i\} \mid (\nu u)\ P$$

where $m ::= 1 \mid \overline{1} \mid \omega$ is called a *mode*. The mode can be either linear 1, affine $\overline{1}$ or replicated ω. We use a standard recursion $(\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle$. The two prefixes of our calculus are the asynchronous output (or *selection*) $\overline{u} \oplus^m l\langle\widetilde{v}\rangle$ which is the output of the values $\widetilde{v}$ on the channel u as well as selecting the label l, and the input (or *choice*) $u\&^m_{i\in I}\{l_i(\widetilde{x_i}).P_i\}$ which offers on channel u several branches to choose from, labelled by the l_is, each with continuation P_i. We use a standard structural congruence $\equiv$ on $\pi^{\{1,\overline{1},\omega\}}$, described in Figure 1.

$$P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3) \qquad P \mid \mathbf{0} \equiv P$$

$$(\nu a)(\nu b)\ P \equiv (\nu b)(\nu a)\ P \qquad (\nu a)\ (P_1 \mid P_2) \equiv ((\nu a)\ P_1) \mid P_2 \text{ if } a \text{ not free in } P_2$$

$$(\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle \equiv P\{\widetilde{v}/\widetilde{x}\}\{\mu X(\widetilde{x}).P/X\}$$

Fig. 1. Structural congruence rules

$$(\mathbf{cong})\frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

$$(\mathbf{comm})\frac{}{\mathbf{E}[\overline{u} \oplus^{1,\overline{1}} l_j\langle\widetilde{v}\rangle \mid u\&^{1,\overline{1}}_{i\in I}\{l_i(\widetilde{x_i}).P_i\}] \to \mathbf{E}[P_j\{\widetilde{v}/\widetilde{x_i}\}]}$$

$$(\mathbf{trig})\frac{}{\mathbf{E}[\overline{u} \oplus^{\omega} l_j\langle\widetilde{v}\rangle \mid u\&^{\omega}_{i\in I}\{l_i(\widetilde{x_i}).P_i\}] \to \mathbf{E}[P_j\{\widetilde{v}/\widetilde{x_i}\} \mid u\&^{\omega}_{i\in I}\{l\}_i(\widetilde{x_i}).P_i]}$$

Fig. 2. Reduction rules

The rules to generate the reduction relation $\to$ are given in Figure 2, using *evaluation contexts* (*à la Wright-Felleisen*) given by: $\mathbf{E} ::= [\,] \mid \mathbf{E} \mid P \mid (\nu u)\ \mathbf{E}$. Henceforth $\to^+$ (resp. $\to^*$) denotes the transitive (resp. reflexive-transitive) closure of $\to$. We also use $\not\to$ to notify that a process cannot reduce further.

Types. Types consist of base types (integers, booleans and unit), the choice and selection types (together called *interaction types*), and recursive types. The choice types can be seen as a generalisation of input types (as used in [3]); the selection types as a generalisation of output types. Their branching structure plays a key role in our subtyping theory. The syntax for types is given by the following grammar:

$$T \ ::= \ \&^m_{i\in I}\{l_i(\widetilde{T_i})\} \ \mid \ \oplus^m_{i\in I}\{l_i(\widetilde{T_i})\} \ \mid \ \mu t.T \ \mid \ t \ \mid \ \mathsf{uc} \ \mid \ \mathtt{N} \ \mid \ \mathtt{B} \ \mid \ \bigstar$$

where $\mathtt{N}$, $\mathtt{B}$ and $\bigstar$ are the types for integers, booleans and the unit, respectively. We assume recursive types are contractive (type variables occur guarded) [20]. *Closed types* are types without free type variables. uc, never occurring in another type, means a pair of dual linear-affine channels are present and is now"uncomposable". Types are considered up to the standard tree isomorphisms.

If T has form $\&_{i\in I}^{m}\{l_i(\widetilde{T_i})\}$ (resp. $\oplus_{i\in I}^{m}\{l_i(\widetilde{T_i})\}$), the *mode* of T, $\mathtt{mod}(T)$, is m.

For brevity we shall often use shortcuts for prefixes and types when their branching is reduced to a single branch. Below we set $x_1 = x$, $P_1 = P$ and $l_1 = l^{\mathsf{one}}$ (l^{one} is a distinguished label we fix).

$$u(x).P = u\&_{i\in\{1\}}^{1}\{l_i(x_i).P_i\} \qquad !u(x).P = u\&_{i\in\{1\}}^{\omega}\{l_i(x_i).P\}$$

$$\overline{u}\langle v\rangle = u \oplus^m l^{\mathsf{one}}\langle v\rangle \text{ when } u \text{ has type } \oplus_{i\in\{1\}}^{m}\{l_i(T_i)\}$$

$$\uparrow^m (T) = \oplus_{i\in\{1\}}^{m}\{l_i(T)\} \qquad \downarrow^m (T) = \&_{i\in\{1\}}^{m}\{l_i(T)\}$$

Example 1 (Intuitive meaning of types)

1. $\uparrow^1 (\downarrow^1 (\mathtt{N}))$ indicates a behaviour at an output channel, through which a process surely sends a channel exactly once; and through that channel, surely receiving a natural number (considered to be a constant channel) exactly once.
2. $\&_{i\in\{1,2\}}^{1}\{l_1(\mathtt{N}), l_2(\uparrow^1 (\downarrow^1 (\mathtt{N})))\}$ indicates a behaviour of exactly once receiving one of the two options, l_1 and l_2, in the former with an integer, in the latter with a channel which the process will use as specified in 1 above.
3. $\&_{i\in\{1,2\}}^{\overline{1}}\{l_1(\mathtt{N}), l_2(\uparrow^1 (\downarrow^1 (\mathtt{N})))\}$ is the same behaviour as 2 above, except it receives an initial option at most once, by the modality $\overline{1}$.

The interaction types form a self-contained universe in that base types can be considered as syntactic sugar, through encodings. There are several faithful (and semantically isomorphic) encodings, of which we present a simple and convenient one. We write T°, if T is a base type, for the encoding of T as an interaction type.

$$\mathtt{N}^\circ \overset{\text{def}}{=} \downarrow^\omega (\oplus_{i\in\mathtt{N}}^{1}\{\mathbf{i}()\}) \qquad \mathtt{B}^\circ \overset{\text{def}}{=} \downarrow^\omega (\oplus_{i\in\mathtt{B}}^{1}\{\mathbf{i}()\}) \qquad \bigstar^\circ \overset{\text{def}}{=} \downarrow^\omega (\oplus_{i\in\bigstar}^{1}\{\mathbf{i}()\})$$

where we use the labels which represent natural numbers, booleans and the single element of the unit. Each describes a behaviour which can be enquired about its content and responds with one. We then extend $(\ )^\circ$ so that when a base type is used for output, we use the above encoding, e.g. $(\uparrow (\mathtt{B}))^\circ = \uparrow (\mathtt{B}^\circ)$, and for input, its dual, e.g. $(\downarrow (\mathtt{B}))^\circ = \downarrow (\overline{\mathtt{B}^\circ})$, where $\overline{T}$ is defined below.

A key idea in interaction types is duality, defined co-inductively [21] to capture recursion. Note by taking recursive types modulo their tree isomorphism, we can safely regard each closed type as either a base, choice or selection type.

Definition 2. *A relation over closed types $\mathcal{R}$ is* duality *if $T_1\ \mathcal{R}\ T_2$ implies:*

1. *either $T_1 = T_2$ where $T_1 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$,*
2. *or (a) $T_1 = \&_{i\in I}^{m}\{l_i(\widetilde{T_i^1})\}$, (b) $T_2 = \oplus_{i\in I}^{m}\{l_i(\widetilde{T_i^2})\}$ and (c) $\forall i \in I, T_i^1\ \mathcal{R}\ T_i^2$*
3. *or (a) $T_1 = \oplus_{i\in I}^{m}\{l_i(\widetilde{T_i^1})\}$ (b) $T_2 = \&_{i\in I}^{m}\{l_i(\widetilde{T_i^2})\}$ and (c) $\forall i \in I, T_i^1\ \mathcal{R}\ T_i^2$*

There is the largest duality relation denoted $\bowtie$.

When we encode away base types through $(\ )^\circ$, we dispense with the first clause.

Fact and Definition 3 (Dualisation). *The duality* $\bowtie$ *defines a total involution (i.e. a symmetric total function), which we write* $\overline{T}$, *called the* dual *of* T.

A generalisation of duality is *coherence*, which we introduce below. Intuitively, coherence specifies when two types can match: this is not only when two types are dual, but also when one offers more choices than the options the other wishes to select. This intuition is the basis of the whole subtyping theory.

Definition 4. *A relation* $\mathcal{R}$ *over closed types is a coherence if* $T_1\ \mathcal{R}\ T_2$ *implies:*

1. *either* $T_1 = T_2$ *where* $T_1 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$.
2. *or (a)* $T_1 = \&^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, *(b)* $T_2 = \oplus^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$, *and (c)* $\forall j \in J, T^1_j\ \mathcal{R}\ T^2_j$ *where* $J \subseteq I$.
3. *or (a)* $T_1 = \oplus^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, *(b)* $T_2 = \&^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$, *and (c)* $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$ *where* $I \subseteq J$

There is the largest coherence relation noted $\asymp$.

Note $\bowtie\ \subsetneq\ \asymp$. Non-trivial inclusion among base types can be incorporated into coherence: however how we can do so is already in $\asymp$, through the encoding discussed above. We shall come back to this point later.

Typing. The typing rules for $\pi^{\{1,\overline{1},\omega\}}$ use typing contexts and compatibility over them, which we introduce below.

A *typing context* Γ is a partial map from names to types and from process variable to vectors of types. When $\Gamma(a)$ is undefined, we write $\Gamma, a{:}T$ to denote the map Γ' defined by $\Gamma'(a) = T$, $\Gamma'(u) = \Gamma(u)$ for $u \neq a$ and $\Gamma'(Y) = \Gamma(Y)$ for all Y. $\Gamma, X{:}\widetilde{T}$ is defined the same way. We write $\emptyset$ for the empty typing context.

Definition 5. *We define* compatibility $\odot$ *as a partial symmetric function over pair of typing contexts generated from:*

$$\begin{array}{ll}
\emptyset \odot \emptyset = \emptyset & \\
(\Gamma_1, X : \widetilde{T}) \odot \Gamma_2 = \Gamma_1 \odot \Gamma_2, X : \widetilde{T} & (\Gamma_1, u : \mathsf{uc}) \odot \Gamma_2 = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} & \textit{if } T_1 \asymp T_2 \textit{ and } \forall i, \mathtt{mod}(T_i) = 1 \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : \mathsf{uc} & \textit{if } T_1 \asymp T_2 \textit{ and } \forall i, \mathtt{mod}(T_i) = \overline{1} \\
(\Gamma_1, u : T_1) \odot (\Gamma_2, u : T_2) = \Gamma_1 \odot \Gamma_2, u : T_1 & \textit{if } T_1 \asymp T_2, \mathtt{mod}(T_1) = \mathtt{mod}(T_2) = \omega \\
& \quad \textit{and } \ T_1 = \&^\omega_{i\in I}\{l_i(\widetilde{T_i})\}
\end{array}$$

Compatibility stipulates, through its partiality, when a parallel composition of two typed processes is allowed. Typing rules are presented in Figure 3.
In (**Cho**), we assume no 1 types occur in Γ when $m = 1$; and no 1 and $\overline{1}$ types occur when $m = \omega$. We observe:

$$(\mathbf{Nil})\frac{}{\emptyset \vdash_\pi \mathbf{0}} \qquad (\mathbf{Rec})\frac{\varGamma, X : \widetilde{T}, \widetilde{x} : \widetilde{T} \vdash_\pi P \qquad \varGamma(\widetilde{v}) = \widetilde{T}}{\varGamma \vdash_\pi (\mu X(\widetilde{x}).P)\langle\widetilde{v}\rangle}$$

$$(\mathbf{Res})\frac{\varGamma, u : T \vdash_\pi P \qquad T = \mathsf{uc} \text{ or } \&^{\omega}_{i\in I}\{l_i(T_i)\}}{\varGamma \vdash_\pi (\nu u)\ P} \qquad (\mathbf{Par})\frac{\varGamma_1 \vdash_\pi P_1 \qquad \varGamma_2 \vdash_\pi P_2}{\varGamma_1 \odot \varGamma_2 \vdash_\pi P_1 \mid P_2}$$

$$(\mathbf{Var})\frac{}{X : \widetilde{T}, \widetilde{v} : \widetilde{T} \vdash_\pi X\langle\widetilde{v}\rangle} \qquad (\mathbf{Sel})\frac{j \in I}{u : \oplus^{m}_{i\in I}\{l_i(\overline{\widetilde{T_i}})\}, \widetilde{v_j} : \widetilde{T_j}, \varGamma \vdash_\pi \overline{u} \oplus^{m} l_j\langle\widetilde{v_j}\rangle}$$

$$(\mathbf{Cho})\frac{(\varGamma, \widetilde{x_i} : \widetilde{T_i} \vdash_\pi P_i)_{i\in I}}{\varGamma, u : \&^{m}_{i\in I}\{l_i(\widetilde{T_i})\} \vdash_\pi u\&^{m}_{i\in I}\{l_i(\widetilde{x_i}) : P_i\}}$$

Fig. 3. Typing rules for $\pi^{\{1,\overline{1},\omega\}}$

Proposition 6 (Subject Reduction). *If $\varGamma \vdash_\pi P$ and $P \to P'$, then $\varGamma \vdash_\pi P'$.*

Example 7. *As an example of the expressive power, we present an easy way to encode references (or* states*):*

$$\mathbf{Mem} = !ref(cell, val).cell\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r, new). & (\overline{r}\langle\,()\rangle \mid \overline{ref}\langle cell, new\rangle) \\ \mathtt{get}(s). & (\overline{s}\langle val\rangle \mid \overline{ref}\langle cell, val\rangle)\end{matrix}\right\}$$

Here **Mem** *is a server that create memory cells. Clients can interact with a cell either to fetch the value store inside or to update it with a new value. Consider:*

$$\mathbf{E}_1 = \mathbf{Mem} \mid \overline{ref}\langle cell_1, 0\rangle \mid \overline{ref}\langle cell_2, 3\rangle$$

which reduces in two steps to:

$$\mathbf{Mem} \mid cell_1\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r_1, new_1). & (\overline{r_1}\langle\,()\rangle \mid \overline{ref}\langle cell_1, new_1\rangle) \\ \mathtt{get}(s_1). & (\overline{s_1}\langle 0\rangle \mid \overline{ref}\langle cell_1, 0\rangle)\end{matrix}\right\}$$

$$\mid cell_2\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(r_2, new_2). & (\overline{r_2}\langle\,()\rangle \mid \overline{ref}\langle cell_2, new_2\rangle) \\ \mathtt{get}(s_2). & (\overline{s_2}\langle 3\rangle \mid \overline{ref}\langle cell_2, 3\rangle)\end{matrix}\right\}$$

creating two memory cells, one called $cell_1$ containing 0 *and one called $cell_2$ containing* 3*. Each cell offers an input with two labels,* `set` *and* `get`*. The former waits for a return channel r, returns the current value through r and construct the cell again, the latter waits for a return channel r and a new value new, reconstructs the updated cell with the new value and return an acknowledgement through r. The process* **Mem** *can be typed by giving to ref the type:*

$$T_{ref} = \downarrow^{\omega} (\&^{\overline{1}}_{l\in\{\mathtt{set},\mathtt{get}\}}\left\{\begin{matrix}\mathtt{set}(\uparrow^{\overline{1}}(\bigstar), \mathtt{N}) \\ \mathtt{get}(\uparrow^{\overline{1}}(\mathtt{N}))\end{matrix}\right\}, \mathtt{N})$$

As another example showing concurrency, consider:

$$\mathbf{Ex} = (\nu ans_1, ans_2)\ (\mathbf{E}_1 \mid \overline{cell_1}\mathtt{set}\langle 1, ans_1\rangle \mid ans_1() \mid \overline{cell_1}\mathtt{get}\langle ans_2\rangle \mid ans_2(x)$$

When reducing **Ex***, x can be instantiated either by* 1 *or* 0 *depending on which output on* $cell_1$ *occurs first.*

Subtyping theory. As hinted in Definition 4, the framework of $\pi^{\{1,\overline{1},\omega\}}$ lets us define a notion of subtyping: informally, a type will be a subtype of another if it "offers more choices" (as input) or if it "selects among less options" (as an output). Intuitively, a type which is ready to receive no less labels, and which may potentially sends no more labels, (in other words, a type representing a more gentle behaviour), is a subtype of another.

Definition 8. *A relation $\mathcal{R}$ is a* subtyping relation *if when $T_1 \mathrel{\mathcal{R}} T_2$ then:*

1. *either $T_1 = T_2 \in \{\mathtt{N}, \mathtt{B}, \bigstar\}$,*
2. *or (a) $T_1 = \&^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, (b) $T_2 = \&^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$ and (c) $\forall j \in J, T^1_j \mathrel{\mathcal{R}} T^2_j$, where $J \subseteq$.*
3. *or (a) $T_1 = \oplus^m_{i\in I}\{l_i(\widetilde{T^1_i})\}$, (b) $T_2 = \oplus^m_{j\in J}\{l_j(\widetilde{T^2_j})\}$, and (c) $\forall i \in I, T^1_i \mathrel{\mathcal{R}} T^2_i$, where $I \subseteq J$.*

$\sqsubseteq$ is the largest (for $\subseteq$) subtyping relation.

The subsumption is admissible in our typing system. Below we write $\Gamma \sqsubseteq \Gamma'$ to denote that the two typing environments have same domain and that for each a (resp. X) in the domain of Γ, $\Gamma(a) \sqsubseteq \Gamma'(a)$ (resp. $\Gamma(X) \sqsubseteq \Gamma'(X)$).

Proposition 9 (Subsumption). *If $\Gamma \vdash_\pi P$ and $\Gamma \sqsubseteq \Gamma'$ then $\Gamma' \vdash_\pi P$.*

That is, if P satisfies Γ and if Γ' is more inclusive as a specification then P also satisfies Γ' (noting Γ in $\Gamma \vdash_\pi P$ specifies the behaviour of P, cf. Example 1).

Now it is known in the literature that subtyping is closely related to composability of types, cf. [11], which we may call *compatibility.* In brief, T_2 *has more compatibility than* T_1 if it is coherent with every type with which T_1 is coherent: it is more composable. In the following we show compatibility and subtyping coincide in our theory, showing its consistency as well as giving useful theoretical tools.

Definition 10. *For T_1 and T_2 closed, $T_1 \leq^{\mathtt{comp}} T_2$ when $\forall T, T_1 \asymp T \Rightarrow T_2 \asymp T$.*

Below Propositions 11 relates together duality, subtyping and coherence, using which we show coincidence, Proposition 12.

Proposition 11. *(1) $T \asymp \overline{T}$. (2) $T_1 \asymp T_2$ iff $T_2 \asymp T_1$. (3) $T_1 \sqsubseteq T_2$ iff $\overline{T_2} \sqsubseteq \overline{T_1}$. (4) If $T'_1 \sqsubseteq T_1$, $T'_1 \asymp T'_2$, $T'_2 \sqsubseteq T_2$ then $T_1 \asymp T_2$. (5) $T_1 \sqsubseteq T_2$ iff $T_1 \asymp \overline{T_2}$*

Proposition 12 (Coincidence). *$T_1 \sqsubseteq T_2$ iff $T_2 \leq^{\mathtt{comp}} T_1$.*

Proof (Sketch). First we show $\leq^{\mathtt{comp}}$ is a subtyping relation coinductively by inspecting the shape of T_2 and Def. 4 gives information on T_1. Then we prove $T \asymp T_1$ implies $T \asymp T_2$ by the form of T_2, using Definition 8. □

Example 13. *To illustrate our subtyping, recall T_{ref} from section 2 inhabited by* **Mem***. An obvious subtype of T_{ref} is the following T'_{ref}:*

$$T'_{ref} = \downarrow^{\omega} (\overline{\&}^{1}_{l\in\{\texttt{set},\texttt{get},\texttt{del}\}}\{ \begin{array}{l} \texttt{set}(\uparrow^{\overline{1}}(\bigstar), \texttt{N}) \\ \texttt{get}(\uparrow^{\overline{1}}(\texttt{N})) \\ \texttt{del}(\uparrow^{\overline{1}}(\bigstar)) \end{array} \}, \texttt{N})$$

Since this is a subtype of T_{ref}, there are some processes which live (are typed by) $ref : T_{ref}$ as well as by $ref : T'_{ref}$. The following is such a process:

$$\mathbf{Mem}' =$$

$$!ref'(cell, val).cell\overline{\&}^{1}_{l\in\{\texttt{set},\texttt{get},\texttt{del}\}}\{ \begin{array}{ll} \texttt{set}(r, new). & (\overline{r}\langle\,()\rangle \mid \overline{ref'}\langle cell, new\rangle) \\ \texttt{get}(s). & (\overline{s}\langle val\rangle \mid \overline{ref'}\langle cell, val\rangle) \\ \texttt{del}(t). & (\overline{t}\langle\,()\rangle) \end{array} \}$$

The process **Mem′** *performs the same role as* **Mem** *but offer one additional choice, the label* `del` *allows one to delete a memory cell, preventing further interactions with this cell to be performed. One can notice that in every process containing* **Mem** *under $ref : T_{ref}$, it can be replaced with* **Mem′***: as far as the supertype T_{ref} goes, they have the same behaviour.*

3 Embedding Functional Subtyping

A subtyped call-by-value functional calculus. For brevity we consider a typed, PCF-like, call-by-value λ-calculus with $\bigstar$ as its base type, which we call $\lambda^{\Pi,\Sigma,\sqsubseteq}$. The syntax of terms contains both products (or *records*) $\{l_i.M_i\}_{i\in I}$ and projections, and sums (or *variants*) and case-branches, and is given, along with the syntax of types, by the following grammar:

$$M ::= M\ M \mid x \mid () \mid \lambda x.M \mid \{l_i.M_i\}_{i\in I} \mid M.l$$

$$\mid\ \ \texttt{inj}_l(M) \mid \texttt{case}\ M\ \texttt{of}\ [l_i(x_i).M_i]_{i\in I} \mid \texttt{Y}\ V$$

$$T ::= \bigstar \mid \texttt{N} \mid T \to T \mid \Pi\{l_i : T_i\}_{i\in I} \mid \Sigma[l_i : T_i]_{i\in I}$$

The subtyping relation $T_1 \sqsubseteq T_2$ on $\lambda^{\Pi,\Sigma,\sqsubseteq}$ types is defined as the largest reflexive and transitive relation satisfying:

1. If $T_1 = T_a \to T_b$, then $T_2 = T'_a \to T'_b$ and $T_a \sqsubseteq T'_a$, $T'_b \sqsubseteq T_b$.
2. If $T_1 = \Pi\{l_i : T_i\}_{i\in I}$, then $T_2 = \Pi\{l_i : T'_i\}_{i\in J}$, $J \subseteq I$ and $\forall i \in J, T_i \sqsubseteq T'_i$.
3. If $T_1 = \Sigma[l_i : T_i]_{i\in I}$, then $T_2 = \Sigma[l_j : T'_j]_{j\in J}$, $I \subseteq J$ and $\forall i \in I, T_i \sqsubseteq T'_i$.

The typing system and the call-by-value reduction rules are completely standard: for reference, below we only list the key rule for subtyping, the subsumption, and leave the rest in [7].

$$(\mathbf{Sub})\frac{\Gamma \vdash M : T_1 \qquad T_1 \sqsubseteq T_2}{\Gamma \vdash M : T_2}$$

Encoding of types. We present the encoding of $\lambda^{\Pi,\Sigma,\sqsubseteq}$ into our calculus $\pi^{\{1,\overline{1},\omega\}}$. Definition 14 presents how we encode $\lambda^{\Pi,\Sigma,\sqsubseteq}$ types into $\pi^{\{1,\overline{1},\omega\}}$ types. A notable point of this encoding is the encoding of the *arrow* type, which is decomposed in such a way that function application is seen as a choice between the possible arguments. As a result, the type $\mathtt{B} \to T$, for instance, is encoded into $\uparrow^{\overline{1}} (\&^{\omega}_{l \in \{\mathtt{true},\mathtt{false}\}}\{\mathtt{true}(T);\mathtt{false}(T)\})$. This means that a function having booleans as domain can be seen as being composed of two terms, one associated to the argument `true`, the other for `false`.

Definition 14 (Encoding of types)

$$\langle \bigstar \rangle = \uparrow^{\overline{1}} (\bigstar) \qquad\qquad \langle \Pi\{l_i : T_i\}_{i\in I}\rangle = \uparrow^{\overline{1}} (\&^{\omega}_{i\in I}\{l_i(\langle T_i\rangle)\})$$

$$\langle \Sigma[l_i : T_i]_{i\in I}\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(\downarrow^{\omega} (\langle T_i\rangle))\}$$

$$\langle T_1 \to T_2\rangle = \uparrow^{\overline{1}} (\&^{\omega}_{i\in I}\{l_i(\overline{T_i}, \langle T_2\rangle)\}) \quad \textit{if } \langle T_1\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(T_i)\}$$

The last line is well-defined since $\langle T_1\rangle$ is always an output type. In the first line, we can encode $\bigstar$ (of $\pi^{\{1,\overline{1},\omega\}}$) by $(\ )^{\circ}$ in Section 2, similarly any base types.

Encoding of terms. Figure 4 gives the encoding of terms following that of types and using a return channel u and an environment ζ (required to remember encoding of variables). The former is standard [24]. The latter may be notable, coming from our arrow type encoding which forces us to remember the association between a branch label and a variable. An environment ζ maps $\lambda^{\Pi,\Sigma,\sqsubseteq}$ each variable to a *choice* (l_i, x_i), a pair composed of one label and one $\pi^{\{1,\overline{1},\omega\}}$ variable.

A brief illustration of three key cases: when encoding an application $M\ N$ on u, one compute the encoding on M and N with two new return channels (respectively m and n), then the address of the function y is caught on m and the possible arguments are decomposed into an address x_i and a label l_i, fetched on n. Then both of them are sent to the function together with the return channel u. Symmetrically, to encode the abstraction $\lambda x.M$, we create a new channel c, send it on the return channel of the function, then we wait for a label l_i which will determine which branch of the function is chosen, an argument x_i and a return channel m, and we proceed to the execution of the encoding of M, with a new environment where the variable x is associated with the choice (l_i, x_i). Finally, to encode a variable x on the return channel u, we fetch in the environment the choice (l_i, x_i) associated with x and send it on the channel u.

As our encoding makes use of environment, we formally define the encoding for typing contexts accordingly.

$$\langle \Gamma, x : T\rangle^{\zeta, x\mapsto(l_i,x_i)} = \langle \Gamma\rangle^{\zeta}, x_i : T_i \text{ if } \langle T\rangle = \oplus^{m}_{i\in I}\{l_i(T_i)\}$$

A variable environment ζ is *reasonable w.r.t. a term* M when ζ maps every free variable of M to a choice.

Proposition 15. *If* $\Gamma \vdash M : T$, *then* $\langle \Gamma\rangle^{\zeta}, u : \langle T\rangle \vdash_{\pi} \langle M\rangle^{\zeta}_{u}$ *for all reasonable* ζ.

$$\langle ()\rangle^\zeta_u = \overline{u}\langle ()\rangle$$

$$\begin{aligned}\langle M\ N\rangle^\zeta_u =&\ (\nu m,n)\ (\langle M\rangle^\zeta_m \mid \langle N\rangle^\zeta_n \mid m(y).n\&^{\overline{1}}_{i\in I}\{l_i(x_i).y\oplus^\omega l_i\langle x_i,u\rangle\})\\ &\text{if } M \text{ has type } T\to T' \text{ with } \langle T\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(T_i)\}\\ \langle \lambda x.M\rangle^\zeta_u =&\ (\nu c)\ (\overline{u}\langle c\rangle.c\&^\omega_{i\in I}\{l_i(x_i,m).\langle M\rangle^{\zeta,x\mapsto(l_i,x_i)}_m\})\\ &\text{if } \lambda x.M \text{ has type } T\to T' \text{ with } \langle T\rangle = \oplus^{\overline{1}}_{i\in I}\{l_i(T_i)\}\end{aligned}$$

$$\langle x\rangle^{\zeta.x\mapsto(l_i,x_i)}_u = \overline{u}\oplus^{\overline{1}} l_i\langle x_i\rangle \qquad \langle\{l_i : M_i\}_{i\in I}\rangle^\zeta_u = (\nu p)\ \overline{u}\langle p\rangle.p\&^\omega_{i\in I}\{l_i().\langle M_i\rangle^\zeta_u\}$$

$$\langle M.l\rangle^\zeta_u = (\nu m)\ (\langle M\rangle^\zeta_m \mid m(y).\overline{y}\oplus^\omega l\langle\rangle)$$

$$\langle \mathtt{inj}_l(M)\rangle^\zeta_u = (\nu m)\ (\langle M\rangle^\zeta_m \mid m\&^{\overline{1}}_{j\in J}\{l'_j(y_j).(\nu c)\ (\overline{u}\oplus^{\overline{1}} l\langle c\rangle \mid !c(n).\overline{n}\oplus^\omega l'_j\langle y_j\rangle)\})$$

$$\langle \mathtt{case}\ M\ \mathtt{of}\ [l_i(x_i).M_i]_{i\in I}\rangle^\zeta_u =$$

$$(\nu m)\ (\langle M\rangle^\zeta_m \mid m\&^{\overline{1}}_{i\in I}\{l_i(x_i).(\nu p)\ (\overline{x_i}\langle p\rangle \mid p\&^\omega_{j\in J}\{l'_j(y_j).\langle M_i\rangle^{\zeta,x_i\mapsto(l'_j,y_j)}_u\})\})$$

$$\langle \mathtt{Y}\ V\rangle^\zeta_u = \mu X(x).((\nu p,m)\ \langle V\rangle^\zeta_p \mid (X\langle m\rangle) \mid m(f).p(a).\overline{f}\langle a,x\rangle)\langle u\rangle$$

Fig. 4. Encoding for λ-terms

Example 16. *Consider the following $\lambda^{\Pi,\Sigma,\sqsubseteq}$ term:*

$$F^{opt} = \lambda x.\mathtt{case}\ x\ \mathtt{of}\ \left[\begin{matrix}\mathtt{s}(x_1) & (F\ x_1)\\ \mathtt{n}(x_2) & 0\end{matrix}\right]_{l\in\{\mathtt{s},\mathtt{n}\}}$$

$F^{opt} : (\mathtt{s} : \mathtt{N}) + (\mathtt{n} : ()) \to \mathtt{N}$ is a partial version of the function $F : \mathtt{N}\to\mathtt{N}$. If its argument is an actual value $\mathtt{inj}_{\mathtt{s}}(3)$ it will apply F to it. And if its argument is undefined $\mathtt{inj}_{\mathtt{n}}(())$ then it will return 0. With $\langle\mathtt{N}\rangle = \uparrow^{\overline{1}}(\mathtt{N})$ and $\langle\bigstar\rangle = \uparrow^{\overline{1}}(\bigstar)$, its encoding $\langle F^{opt}\rangle^\emptyset_u$ is given by:

$$(\nu v)\ \overline{u}\langle v\rangle.v\&^\omega_{l\in\{\mathtt{s},\mathtt{n}\}}\left\{\begin{matrix}\mathtt{s}(x,b).\ (\nu c)\ (\overline{c}\oplus^{\overline{1}}\mathtt{s}\langle x\rangle \mid \textbf{Caseof})\\ \mathtt{n}(x,b).\ (\nu c)\ (\overline{c}\oplus^{\overline{1}}\mathtt{n}\langle x\rangle \mid \textbf{Caseof})\end{matrix}\right\}$$

$$\textbf{Caseof}\ \textit{being}\ c\&^{\overline{1}}_{l\in\{\mathtt{s},\mathtt{n}\}}\left\{\begin{matrix}\mathtt{s}(x_1).\ (\nu p_1)\ (\overline{x_1}\langle p_1\rangle \mid !p_1(n).\textbf{App})\\ \mathtt{n}(x_2)\ \ (\nu p_2)\ (\overline{x_1}\langle p_2\rangle \mid !p_2.\overline{b}\langle 0\rangle)\end{matrix}\right\}$$

$$\textbf{App}\ \textit{being}\ (\nu f,a)\ (\langle F\rangle^\emptyset_f \mid \overline{a}\langle n\rangle \mid f(y).a(z).\overline{y}\langle z,b\rangle)$$

If we look through the several indirections induced by the encoding, we can notice that the choice induced by the option type will be translated as two choices, one in the abstraction encoding and one in the case-construct encoding.

Functional subtyping through encoding. The following proposition relates the subtyping for $\lambda^{\Pi,\Sigma,\sqsubseteq}$ with the subtyping for $\pi^{\{1,\overline{1},\omega\}}$. It is proved by showing that the relation $\mathcal{R}$, defined by $T_a\mathcal{R}T_b$ when there exist T_1, T_2 s.t. $\langle T_1\rangle = T_a$, $\langle T_2\rangle = T_b$, and $T_1\ \mathcal{R}\ T_2$, is a subtyping according to Definition 8.

Proposition 17 *If $T_1 \sqsubseteq T_2$, then $\langle T_1 \rangle \sqsubseteq \langle T_2 \rangle$.*

The base type encoding in $\pi^{\{1,\overline{1},\omega\}}$ through the operator $(\ )^\circ$ in section 2 allows us to extend the above result to $\lambda^{\Pi,\Sigma,\sqsubseteq}$ with non-trivial subtyping on base types, e.g. with the type $\mathtt{R}$ of reals. Indeed, for $\mathtt{N} \sqsubseteq \mathtt{R}$, we get $\langle \mathtt{N} \rangle^\circ \sqsubseteq \langle \mathtt{R} \rangle^\circ$, by $\uparrow^{\overline{1}} (\downarrow^\omega (\oplus^1_{i \in \mathbb{N}}\{\mathbf{i}()\})) \sqsubseteq \uparrow^{\overline{1}} (\downarrow^\omega (\oplus^1_{i \in \mathbb{R}}\{\mathbf{i}()\}))$.

Full abstraction. In order to obtain a full abstraction result we restrict our π-calculus, imposing *sequentiality* to typed processes by controlling the number of activities compositionally as in [3]. We derive a definability result [7], that is, every sequential process typable with the encoding a λ-typing context is equivalent to the encoding of a term typable with that context. Using as the equivalence on the functional side $\simeq_\lambda$ Morris congruence [17] and as the equivalence of the concurrent side $\simeq_\pi$ the standard reduction-closed barbed congruence [24], we obtain:

Theorem 18. *Let M_1, M_2 be two $\lambda^{\Pi,\Sigma,\sqsubseteq}$ terms, then $M_1 \simeq_\lambda M_2$ if and only if $\langle M_1 \rangle^\emptyset_u \simeq_\pi \langle M_1 \rangle^\emptyset_u$ under sequential typing.*

4 Embedding Communication Subtyping

This section is dedicated to the study of the encoding of a session-based concurrent calculus π^{session} with synchronous interactions based on session types. Session types abstract protocols of communicating processes as types, and ensure their sound communication behaviour through the associated type discipline, see [8] for a survey.

Our presentation of π^{session} follows [13]. In this language, names are divided into channels $u, v, \ldots$ and sessions $s, k, \ldots$; we use $a, b, c, \ldots$ to denote names of any kind. Syntax for terms is given by the following grammar:

$$P ::= u(x).P \mid \overline{u}(x).P \mid k!l\langle v \rangle.P \mid k?\{l_i(x_i).P_i\}_{i \in I} \mid \text{if } e \text{ then } P \text{ else } P$$

$$\mid (\nu a)P \mid P|P \mid (\mu X(\widetilde{x}).P)\langle \widetilde{v} \rangle \mid X\langle \widetilde{v} \rangle \mid \mathbf{0}$$

The definition of evaluation contexts is straightforward. For brevity, we do not include delegation, though its encoding follows that of shared name passing. The semantics is generated from the following two base rules:

$$\frac{}{\mathbf{E}[u(x_1).P_1 \mid \overline{u}(x_2).P_2] \to \mathbf{E}[(\nu s)\ (P_1\{s/x_1\} \mid P_2\{s/x_2\})]}$$

$$\frac{}{\mathbf{E}[s?\{(x_i).P_i\}_{i \in I} \mid s!l_j\langle v \rangle.P] \to \mathbf{E}[P_j\{v/x_j\} \mid P]}$$

We use binary session types given as follows.

$$T, S ::= \&^{\mathsf{ses}}_{i \in I}\{l_i(T_i).S_i\} \mid \oplus^{\mathsf{ses}}_{i \in I}\{l_i(T_i).S_i\} \mid \mathsf{end} \mid \downarrow^{\mathsf{ses}} (S) \mid \uparrow^{\mathsf{ses}} (S) \mid \mu S.S \mid \mathtt{N} \mid \bigstar$$

As in $\pi^{\{1,\overline{1},\omega\}}$, we use shortcuts: when I is a singleton $\{1\}$, we use $?(T_1).S_1$ (resp. $!(T_1).S_1$) to denote $\&^{\mathsf{ses}}_{i \in I}\{l_i(T_i).S_i\}$ (resp. $\oplus^{\mathsf{ses}}_{i \in I}\{l_i(T_i).S_i\}$). Compatibility and duality for π^{session} are the straightforward adaptation of those of $\pi^{\{1,\overline{1},\omega\}}$.

$$(\mathbf{SOut})\frac{\Gamma(u) = \uparrow^{\mathsf{ses}} (\overline{T}) \quad \Gamma, x : \overline{T} \vdash_{\mathsf{ses}} P}{\Gamma \vdash_{\mathsf{ses}} \overline{u}(x).P} \qquad (\mathbf{SIn})\frac{\Gamma(u) = \downarrow^{\mathsf{ses}} (T) \quad \Gamma, x : T \vdash_{\mathsf{ses}} P}{\Gamma \vdash_{\mathsf{ses}} u(x).P}$$

$$(\mathbf{SCho})\frac{(\Gamma, x_i : T'_i, s : S_i \vdash_{\mathsf{ses}} P_i)_{i\in I} \qquad J \subseteq I \qquad \forall j \in J. T'_j \sqsubseteq T_j}{\Gamma, s : \&^{\mathsf{ses}}_{i\in J}\{l_i(T_i).S_i\} \vdash_{\mathsf{ses}} s?\{l_i(x_i).P_i\}_{i\in I}}$$

$$(\mathbf{SSel})\frac{\Gamma(v) = T'_j \qquad \Gamma, s : S_j \vdash_{\mathsf{ses}} P \qquad T'_j \sqsubseteq \overline{T_j}}{\Gamma, s : \oplus^{\mathsf{ses}}_{i\in I}\{l_i(\overline{T_i}).S_i\} \vdash_{\mathsf{ses}} s!l_j\langle v\rangle.P} \qquad (\mathbf{SPar})\frac{\Gamma_1 \vdash_{\mathsf{ses}} P_1 \mid \Gamma_2 \vdash_{\mathsf{ses}} P_2}{\Gamma_1 \odot \Gamma_2 \vdash_{\mathsf{ses}} P_1 \mid P_2}$$

Fig. 5. Some of the main typing rules for π^{session}

Some of the key typing rules are given in Figure 5. In session types, the type of a session name s in the prefix $s!l\langle v\rangle.P$ gives information not only on the type of v but also on how the session s will be used in the continuation P.

The encodings. The encoding of π^{session} terms into $\pi^{\{1,\overline{1},\omega\}}$ given in Figure 6. The main points are that, in $\pi^{\{1,\overline{1},\omega\}}$, we lack both synchronous outputs and the way to ensure that sessions behave correctly, that is, how the names in subject positions in later prefixes of the same session, are used following the stipulated protocol, i.e. its session type.

First, to encode the synchronous outputs, we use an administrative synchronisation on a linear name. This is standard [23]: we encode $\overline{a}(v).P$ into $(\nu v, c)\ \overline{a}\langle v, c\rangle \mid c.P$. The new name c is output with the value v and the synchronising party will emit it after inputting the message, thus activating the guarded continuation P. Then, to make sure that sessions are encoded following their protocols, we proceed as follows: if a session name s has type $?(S_1).S_2$, we create a new name k that is given type $[\![S_2]\!]$; and replace in the continuations the name s by k. This gives an equivalent process, and the type of k is now the encoding of the new type S_2 of the session s, after one communication step. The encoding is given by Figure 7.

Soundness of the encoding is stated in the following proposition and proved by induction on the typing derivation.

Proposition 19. *If* $\Gamma \vdash_{\mathsf{ses}} P$ *then* $[\![\Gamma]\!] \vdash_{\pi} [\![P]\!]$.

$$[\![(\nu a)\ P]\!] = (\nu a)\ [\![P]\!] \qquad [\![u(x).P]\!] = u(x,c).(\overline{c} \mid [\![P]\!])$$

$$[\![\overline{u}(x).P]\!] = (\nu x, c)\ (\overline{u}\langle x, c\rangle \mid c.[\![P]\!])$$

$$[\![k!l\langle e\rangle.P]\!] = (\nu c)\ (k \oplus^{\overline{1}} l\langle e, c\rangle \mid c(s).[\![P]\!]\{s/k\})$$

$$[\![k?\{l_i(x_i).P_i\}_{i\in I}]\!] = (\boldsymbol{\nu} s)\ k\&^{\overline{1}}_{i\in I}\{l_i(x_i, c).([\![P_i]\!]\{s/k\} \mid \overline{c}\langle s\rangle)\}$$

Fig. 6. Encoding of π^{session} terms

$$[\![\oplus^{\mathsf{ses}}_{i\in I}\{l_i(T_i).S_i\}]\!] = \oplus^{\omega}_{i\in I}\{l_i([\![T_i]\!], \downarrow^1([\![S_i]\!]))\}$$

$$[\![\&^{\mathsf{ses}}_{i\in I}\{l_i(T_i).S_i\}]\!] = \oplus^{\omega}_{i\in I}\{l_i([\![T_i]\!], \uparrow^1(\overline{[\![S_i]\!]}))\} \qquad [\![\downarrow^{\mathsf{ses}}(S)]\!] = \downarrow^{\overline{1}}([\![S]\!], \uparrow^1(\bigstar))$$

$$[\![\uparrow^{\mathsf{ses}}(S)]\!] = \uparrow^{\overline{1}}([\![S]\!], \downarrow^1(\bigstar)) \qquad [\![\bigstar]\!] = \bigstar \qquad [\![\mathtt{N}]\!] = \mathtt{N}$$

Fig. 7. Encoding of π^{session} types

Example 20. *As an example, consider this toy π^{session} process:*

$$\mathbf{S} \;=\; \overline{a}(x).x?(z_2).x!\langle z_2\rangle \mid a(y).(y!\langle 0\rangle.y?(z_1)$$

$\mathbf{S}$ is composed of two subprocesses, one initiates a new session through channel a, then receives and emits, the other behaves dually. Its encoding is given by:

$$\begin{aligned}[\![\mathbf{S}]\!] = (\nu x, c)(\overline{a}\langle x, c\rangle \mid c.(\nu k_2)\ (x(z_2, c_3).(\overline{c_3}\langle k_2\rangle \mid \overline{k_2}\langle z_2, c_4\rangle \mid c_4)))\\ \mid a(y, c_0).(\overline{c_0} \mid (\nu c_1)\ (\overline{y}\langle 0, c_1\rangle \mid c_1(k_1).(\nu c_2)\ k_1(z_1, c_2).\overline{c_2}))\end{aligned}$$

First, a session initialisation takes place on x (after y has been instantiated to x with a "channel" synchronisation), but a new name k_2 is later created and transmitted in order to continue the session.

For subtyping on session types, we can closely follow $\pi^{\{1,\overline{1},\omega\}}$: a relation over types $\mathcal{R}$ in π^{session} is a *subtyping relation* if, whenever $S_1\ \mathcal{R}\ S_2$, we have:

1. either $S_1 = S_2 = \mathtt{N}$ or $S_1 = S_2 = \bigstar$,
2. or (a) $S_1 = \downarrow^{\mathsf{ses}}(S_1)$ (resp. $\uparrow^{\mathsf{ses}}(S_1)$), (b) $S_2 = \downarrow^{\mathsf{ses}}(S_2)$ (resp. $\uparrow^{\mathsf{ses}}(S_2)$), (c) $S_1\ \mathcal{R}\ S_2$
3. or (a) $S_1 = \&^{\mathsf{ses}}_{i\in I}\{l_i(T^1_i).S^1_i\}$, (b) $S_2 = \&^{\mathsf{ses}}_{j\in J}\{l_j(T^2_j).S^2_j\}$, (c) $I \subseteq J$, (c) $\forall j \in J, T^1_j\ \mathcal{R}\ T^2_j$, (d) $\forall j \in J, S^1_j\ \mathcal{R}\ S^2_j$
4. or (a) $S_1 = \oplus^{\mathsf{ses}}_{i\in I}\{l_i(T^1_i).S^1_i\}$, (b) $S_2 = \oplus^{\mathsf{ses}}_{j\in J}\{l_j(T^2_j).S^2_j\}$, (c) $J \subseteq I$, (d) $\forall i \in I, T^1_i\ \mathcal{R}\ T^2_i$, (e) $\forall i \in I, S^1_i\ \mathcal{R}\ S^2_i$

Then $\sqsubseteq$ is the largest subtyping relation. We can then show the subsumption is admissible in the typing rules for π^{session}[1].

Using the fact that branching session prefixes are encoded into branching $\pi^{\{1,\overline{1},\omega\}}$ prefixes, we prove the following proposition, relating subtyping in π^{session} with subtyping in $\pi^{\{1,\overline{1},\omega\}}$.

Proposition 21 *If $S_1 \sqsubseteq S_2$, then $[\![S_1]\!] \sqsubseteq [\![S_2]\!]$*

[1] In the subtyping, a carried type for a shared channel is covariant in both output and input: this is because we choose each carried name to be typed as the dual of how the other party should use it, following $\pi^{\{1,\overline{1},\omega\}}$. We can use the standard format through dualisation.

For full abstraction, we use a testing (may-must) equivalence based on [6], both for $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$ processes. A maximal reduction sequence starting form P is a sequence $(P_i)_{i\leq n}$ with $n \in \mathbb{N} \cup \{\omega\}$ such that $P_0 = P$, $\forall i$, $P_i \to P_{i+1}$ and $P_n \not\to$.

Definition 22 *We define the barbs for $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$ as follows:*

- *If $P \in \pi^{\{1,\overline{1},\omega\}}$, $P \Downarrow \overline{a}$ when $P \equiv (\nu\widetilde{c})\ (\overline{a} \oplus^m l\langle v\rangle \mid P_1)$ and $a \notin \widetilde{c}$.*
- *If $P \in \pi^{\mathtt{session}}$,*
  - *$P \Downarrow \overline{u}$ when $P \equiv (\nu\widetilde{c})\ (\overline{u}(v).P_2 \mid P_1)$ and $u \notin \widetilde{c}$*
  - *and $P \Downarrow \overline{s}$ when $P \equiv (\nu\widetilde{c})\ (s!l\langle v\rangle.P_2 \mid P_1)$ and $s \notin \widetilde{c}$.*

We define the may observation for $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$ as: $P \Downarrow^{\mathtt{may}}_{\overline{a}}$ when there exists R, $P \to^* R$, $R \not\to$ and $R \Downarrow \overline{a}$. We also define the must observation for $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$ as $P \Downarrow^{\mathtt{must}}_{\overline{a}}$ when for all maximal reduction sequences $(P_i)_{i\leq n}$, $n \in \mathbb{N} \cup \{\omega\}$ starting from P, $P_j \Downarrow \overline{a}$.

Using Definition 22, we define may, must and testing barbed equivalences (considering only observables from processes), denoted $\sim_{\mathtt{may}}$, $\sim_{\mathtt{must}}$ and $\sim_{\mathtt{test}}$; and the corresponding congruences (considering testers), denoted $\simeq_{\mathtt{may}}$, $\simeq_{\mathtt{must}}$ and $\simeq_{\mathtt{test}}$. In both cases, testing is the conjunction of may and must.

Definition 23. *$P \sim_{\mathtt{may}} Q$ if for all a, $P \Downarrow^{\mathtt{may}}_{\overline{a}}$ implies $Q \Downarrow^{\mathtt{may}}_{\overline{a}}$ and $P \to P'$ implies there exists Q' s.t. $Q \to^* Q'*$ and $P' \sim_{\mathtt{may}} Q'$.*

$P \sim_{\mathtt{must}} Q$ if for all a, $P \Downarrow^{\mathtt{must}}_{\overline{a}}$ implies $Q \Downarrow^{\mathtt{must}}_{\overline{a}}$ and $P \to P'$ implies there exists Q' s.t. $Q \to^* Q'*$ and $P' \sim_{\mathtt{must}} Q'$. Then, $P \sim_{\mathtt{test}} Q$ when $P \sim_{\mathtt{may}} Q$ and $P \sim_{\mathtt{must}} Q$.

For $\pi^{\{1,\overline{1},\omega\}}$ and $\pi^{\mathtt{session}}$, we define $P \simeq_{\mathtt{may}} Q$ (resp. $P \simeq_{\mathtt{must}} Q$) if for all $R \in \pi^{\{1,\overline{1},\omega\}}$, $(R \mid P) \sim_{\mathtt{may}} (R \mid Q)$ (resp. $(R \mid P) \sim_{\mathtt{must}} (R \mid Q)$). Then, $P \simeq_{\mathtt{test}} Q$ when $P \simeq_{\mathtt{may}} Q$ and $P \simeq_{\mathtt{must}} Q$.

Full abstraction. Lemma 24 is crucial, it shows how the original process and its encoding are able to simulate each other. The main difficulty is that the encoding introduces communications on new linear names.

Lemma 24. *Below let P be a well-typed $\pi^{\mathtt{session}}$-term.*

1. *If $P \to P'$, then there exists R, $[\![P]\!] \to R$ and $R \to^+ [\![P']\!]$*
2. *If $[\![P]\!] \to R$, then there exists P', $P \to P'$ and $R \to [\![P']\!]$.*
3. *If $(Q_i)_{i\leq n}$ is a maximal reduction sequence starting from $[\![P]\!]$, there exists a maximal reduction sequence $(P_i)_{i\leq n}$ starting from P and a strictly increasing function $\phi : \mathbb{N} \to \mathbb{N}$ s.t. $[\![P_i]\!] \equiv Q_{\phi(i)}$*

The proof of 3 above concerns infinite reduction sequences. In this case, one has first to prove that such a reduction sequence contains an infinite number of non-linear reduction steps.

Lemma 25 (Definability). *For all $P \in \pi^{\{1,\overline{1},\omega\}}$, $\Gamma \in \pi^{\mathtt{session}}$ s.t. $[\![\Gamma]\!] \vdash_\pi P$, exists $R \in \pi^{\mathtt{session}}$ s.t. $\Gamma \vdash_{\mathtt{ses}} R$ and $P \simeq_{\mathtt{test}} [\![R]\!]$.*

A key idea of the proof is decoding P into a corresponding $\pi^{\mathtt{session}}$-process. First, for every e.g. output at a hidden name say a which is not hereditarily typed in $[\![\Gamma]\!]$, we replace it by an encoding of a session type, similarly for a hidden input. Then all names in P are now typed with the encoding of session types, without changing behaviour. We now use induction on typing rules for $\pi^{\{1,\overline{1},\omega\}}$ to extract the shape of encoded processes from P by induction on the size of P.

Now the computational adequacy lemma concludes the full-abstraction proof.

Lemma 26 (Adequacy). *For all $P \in \pi^{\mathtt{session}}$ s.t. $\Gamma \vdash_{\mathtt{ses}} P$, $P \sim_{\mathtt{test}} [\![P]\!]$.*

Theorem 27 (Full abstraction) *Let P, Q be two $\pi^{\mathtt{session}}$ processes s.t. $\Gamma \vdash_{\mathtt{ses}} P$ and $\Gamma \vdash_{\mathtt{ses}} Q$, then $P \simeq_{\mathtt{test}} Q$ if and only if $[\![P]\!] \simeq_{\mathtt{test}} [\![Q]\!]$.*

Proof (Sketch). For both implications, we present only the 'may' case, the 'must' one is very similar.

– We take a tester K, from Lemma 26, $(P \mid K) \sim_{\mathtt{may}} [\![P \mid K]\!]$. By hypothesis, $[\![P \mid K]\!] \sim_{\mathtt{may}} [\![Q \mid K]\!]$ and by Lemma 26, $[\![Q \mid K]\!] \sim_{\mathtt{may}} Q \mid K$, hence done.
– We take a tester R, from Lemma 25, we get K s.t. $R \simeq_{\mathtt{may}} [\![K]\!]$ and thus $(R \mid [\![P]\!]) \sim_{\mathtt{may}} [\![K \mid P]\!]$. By Lemma 26, $[\![K \mid P]\!] \sim_{\mathtt{may}} (K \mid P)$. By hypothesis, $(K \mid P) \sim_{\mathtt{test}} (K \mid Q)$. By Lemma 26, $(K \mid Q) \sim_{\mathtt{test}} [\![K \mid Q]\!]$ and then, $R \mid [\![P]\!] \sim_{\mathtt{test}} R \mid [\![Q]\!]$.

5 Related Work and Further Topics

In [21], the authors present both the idea of distinguishing output and input types in the π-calculus, and the use of subtyping for controlling the right to perform actions on a given channel. They show that this notion of subtyping allows them to state an operational correspondence between λ-terms and the second encoding into the π-calculus proposed in [16]. The work in [5] addresses the question of the subtyping in a semantic way: a type T_1 is a subtype of a type T_2 if the interpretation of T_2 (the set of all processes that can be typed with T_2) is included into the interpretation of T_1. They prove that their definition of subtyping is decidable and present a π-calculus with dynamic type-checks. The work in [10] first introduces to session types a notion of subtyping based on branching. Through dualisation, the session subtyping we treated in the present paper is essentially identical. The present work has embedded the session subtyping in a more fine-grained linear typing, and has shown that it leads to not only the embedding of the subtyping but also semantic full abstraction, which may shed light on this ordering relation and its theory. The author of [18] studies the semantics of session-types, which contains a notion of subtyping for sessions, called *subsession*, a session S_1 is a subsession of S_2 when for every session S, if $(S_1 \mid S)$ must reach a successful state (according to a *must* semantics similar to the one we use) implies that $(S_2 \mid S)$ must reach a successful state. This notion of subtyping is more related to what we called *compatibility ordering* in Section 2.

Our full abstraction proof for the $\lambda^{\Pi,\Sigma,\sqsubseteq}$ embedding follows game semantics except the subtyping and, in the context of the π-calculus, the one found in [3], which uses both a restriction to a sequential π-calculus and the use of a finite definability lemma.

Our full abstraction proof for the $\pi^{\mathtt{session}}$ embedding is inspired by the one in [23] where the authors prove the full abstraction of a polyadic, output-synchronous π-calculus into a monadic, output-asynchronous one. They use a computational adequacy lemma stating that a process and its encoding are barbed bisimilar. Together with a definability result, it leads to the full abstraction result, with a barbed congruence as equivalence. As a comparison, in [23], the meta-calculus (monadic asynchronous π) is a sub-calculus of the target calculus (polyadic synchronous π), which may have facilitated the proof based on barbed bisimilarity. On the other hand, barbed congruence is in terms of the branching structure finer than the testing equivalence. It is an interesting future topic how we can obtain a similar result for such finer equivalences in the present setting.

The calculus we propose in this paper can be easily extended to accommodate more features. For instance, we believe we can adapt a standard definition of polymorphic types [26] in order to include it into our subtyping framework. This opens the possibility of studying the encoding of subtyped System F [4], together with the polymorphic subtypes, into our calculus. We are also interested in showing that our subtyping theory can accommodate objects. Though an encoding of object-oriented calculi into the π-calculus have already been proposed (see [27] for instance) and subtyping for objects is well-studied [1], we believe the analysis as we have carried out in this work will shed new light on their nature. Finally, we are interested in studying how to accommodate other definitions for subtyping for sessions, as the one presented in [19].

Acknowledgements. We thank the CONCUR referees for their useful comments. This work is supported by EPSRC grants EP/F002114/1 and EP/G015481/1.

References

1. Abadi, M., Cardelli, L.: A semantics of object types. In: LICS, pp. 332–341. IEEE Computer Society, Los Alamitos (1994)
2. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. 15(4), 575–631 (1993)
3. Berger, M., Honda, K., Yoshida, N.: Genericity and the pi-calculus. Acta. Inf. 42(2-3), 83–141 (2005)
4. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. 17(4), 471–522 (1985)
5. Castagna, G., Nicola, R.D., Varacca, D.: Semantic subtyping for the pi-calculus. Theor. Comput. Sci. 398, 217–242 (2008)
6. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: Arvind, V., Ramanujam, R., (eds.) FST TCS 1998. LNCS, vol. 1530, pp. 90–102. Springer, Heidelberg (1998)

7. Demangeon, R., Honda, K.: Full abstraction in a subtyped pi-calculus with linear types, long version (2011) (in preparation), `http://perso.ens-lyon.fr/romain.demangeon/subtyping_long.pdf`
8. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and Session Types: An Overview. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 1–28. Springer, Heidelberg (2010)
9. Fiore, M.P., Honda, K.: Recursive types in games: Axiomatics and process representation. In: LICS, pp. 345–356 (1998)
10. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta. Inf. 42(2-3), 191–225 (2005)
11. Honda, K.: Composing processes. In: POPL, pp. 344–357 (1996)
12. Honda, K.: Processes and games. Electr. Notes Theor. Comput. Sci. 71 (2002)
13. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
14. Honda, K., Yoshida, N.: Game-theoretic analysis of call-by-value computation. Theor. Comput. Sci. 221(1-2), 393–456 (1999)
15. Laird, J.: A game semantics of the asynchronous *pi*-calculus. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 51–65. Springer, Heidelberg (2005)
16. Milner, R.: Functions as processes. Mathematical Structures in Computer Science 2(2), 119–141 (1992)
17. Morris, J.H.: Lambda-Calculus Models of Programming Languages. PhD Thesis, M.I.T (1968)
18. Padovani, L.: Session types at the mirror. In: ICE, pp. 71–86 (2009)
19. Padovani, L.: Fair subtyping for multi-party session types. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 127–141. Springer, Heidelberg (2011)
20. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
21. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science 6(5), 409–453 (1996)
22. Pierce, B.C., Steffen, M.: Higher-order subtyping. Theor. Comput. Sci. 176(1-2), 235–282 (1997)
23. Quaglia, P., Walker, D.: Types and full abstraction for polyadic *pi*-calculus. Inf. Comput. 200(2), 215–246 (2005)
24. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
25. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
26. Turner, N.: The polymorphic pi-calculus: Theory and Implementation. PhD thesis, Department of Computer Science, University of Edinburgh (1996)
27. Walker, D.: Objects in the pi-calculus. Inf. Comput. 116(2), 253–271 (1995)
28. Yoshida, N., Berger, M., Honda, K.: Strong Normalisation in the Pi-Calculus. Information and Computation 191(2), 145–202 (2004)