First-Order Store and Visibility in Name-Passing Calculi

Daniel Hirschkoff
LIP, ENS de Lyon, France

Iwan Quémerais
LIP, ENS de Lyon, France

Davide Sangiorgi
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Abstract

The π-calculus is the paradigmatical name-passing calculus. While being purely name-passing, it allows the representation of higher-order functions and store.

We study how π-calculus processes can be controlled so that computations can only involve storage of first-order values. The discipline is enforced by a type system that is based on the notion of visibility, coming from game semantics. We discuss the impact of visibility on the behavioural theory. We propose characterisations of may-testing and barbed equivalence, based on (variants of) trace equivalence and labelled bisimilarity, in the case where computation is sequential, and in the case where computation is well-bracketed.

2012 ACM Subject Classification Theory of computation → Process calculi

Keywords and phrases process calculi, behavioural equivalence, type system

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.23

Related Version *Extended Version*: `arXiv.2504.17350` [4]

Funding Work partly supported by the MIUR-PRIN project "Resource Awareness in Programming: Algebra, Rewriting, and Analysis" (RAP, ID P2022HXNSC).

Acknowledgements We have benefited from discussions with G. Jaber and E. Prebet.

1 Introduction

The π-calculus is one of the best known process calculi. It has a handful of operators – parallelism, input, output, restriction being the main ones – with which it achieves an amazing expressive power. Yet, it has a rich algebraic theory and a wide spectrum of proof techniques. In the π-calculus, computation is communication, by means of message-passing synchronisations between processes. Names are the communication units; new names may be created, and may themselves be exchanged in communications. When used for modelling, π-calculus is normally typed, so to be able to support structured data values, beginning with basic first-order values such as integers and booleans.

The π-calculus has been advocated as a model to interpret, and give semantics to, languages with higher-order features. For instance, the representation of functions as π-calculus processes has been widely studied, beginning with Milner's seminal work [16]. Usually, higher-order languages include forms of *store*. It is therefore important to understand how expected semantic properties of store can be captured in a language for pure concurrency such as the π-calculus.

An aspect of store with strong semantic consequences is the distinction between higher-order and first-order store. The term "first-order store" refers to a memory model where variables can hold only basic data types such as integers and booleans. This contrasts with a

higher-order store, where variables can also store functions, or code, or references to these. Limiting the store to first-order data makes it easier to reason about program behaviours, and verify behavioural properties. It may also enhance security and predictability guarantees by preventing unintended behaviours associated with dynamic function storage and code injection. A striking example of a property that may be broken with higher-order store is termination. For instance, the simply-typed λ-calculus is strongly normalising and therefore terminating. This property is maintained by the addition of first-order store, but not with higher-order store, as functions may recursively reference and invoke each other through the store. In game semantics of higher-order languages, the distinction between first-order and higher-order store has led to studying the concept of *visibility*. Intuitively, visibility ensures us that each move in a computational game is justified by prior interactions. With the introduction of higher-order store, the visibility condition has to be relaxed, reflecting the added complexity in tracking information flow within a program.

The goal of this paper is precisely to understand the meaning of first-order store and visibility within the π-calculus. In functional languages based on the λ-calculus, imposing first-order store is simply determined by the constraint that all references may only store first-order values. In the π-calculus, in contrast, there is no explicit notion of reference, and therefore the meaning of "first-order store" requires some care. In π-calculi, a piece of store is usually represented by an output message. For instance $\overline{h}\langle 5\rangle$ may be thought of as the statement that name h currently contains 5. A process that inputs at h and then re-emits a message at h, such as $h(z).\,\overline{h}\langle z+1\rangle$, is a process that acts on such a store. In this case, h is a first-order name, hence h implements a piece of first-order store. An example of higher-order store is given by the process

$$P \stackrel{\text{def}}{=} a(b).\,((\overline{b}(d).\,!d.\,R) \mid c.\,\overline{b}(d').\,d'.\,R') \tag{1}$$

Here, $\overline{b}(d)$ is a bound output, and represents the output of a "new" name d, and similarly for $\overline{b}(d')$; whereas, e.g., $d.\,R$ and $c.\,Q$ (where Q is $\overline{b}(d').\,d'.\,R'$) represent inputs in which no value is exchanged (i.e., d and c carry `unit` values). In P, name a is higher-order, in that a carries a (pointer to a) service. In the input at a, the service name received in the input (the parameter b) is used after the input, but it is also stored, and used later when c is invoked. In other words, the visibility of c (the services that c may call when invoked) depends on interactions that have occurred earlier, at a. For this reason, P implements a form of higher-order store. When this does not happen, that is, the visibility of a higher-order name (the set of services that may be called when the name is invoked) may be determined at the point in which the name is created, the process implements a form of first-order store. (The word "service" is freely used, here and elsewhere, to indicate the input end of a name; in π-calculus, a name may have multiple input occurrences, in arbitrary syntactic positions within a process, in contrast with, say, the function declarations in functional languages, which are supposed to be unique.)

The amazing expressive power of the π-calculus also makes the contexts of the calculus very discriminating; as a consequence, behavioural equivalences, which are supposed to be preserved by the contexts of the calculus, are rather demanding relations. A well-established way of imposing constraints on the set of legal contexts, thus obtaining usefully coarser behavioural equivalences, is to adopt behavioural type systems. Types may be used in order to capture communication patterns on the usage of names, such as capabilities, linearity, sessions, and so on, e.g., [19, 11, 6, 1]. Type systems have also been designed to capture specific properties of processes, such as termination, deadlock-freedom, lock-freedom, e.g., [24, 12, 18].

A further step is then to tune the proof techniques of the π-calculus to such type systems, so to be able to actually prove the behavioural equalities that only hold in presence of types. For instance, in the case of may testing (or contextual equivalence) this may yield to refinements or modifications of the notion of trace equivalence; whereas in the case of barbed equivalence this may lead to refinements or modifications of the standard notion of labelled bisimilarity. The resulting proof techniques must be sound with respect to initial contextually-defined relations; ideally, they should also be complete.

In this paper we propose a type system that guarantees first-order store in π-calculus processes. The type system makes use of *visibility functions*, that is, partial finite functions that specify, for each higher-order name b that may occur free in a process, the set of services (i.e., higher-order names) that may be invoked when a call at b is made. In order to better understand the behavioural effects of visibility, we combine it first with *sequentiality* and then with *well-bracketing*. "Sequentiality" intuitively indicates the existence of a single thread of computation. That is, at any point there is a single process that is active and decides what the next computation step can be. In the encodings of the λ-calculus [16, 21], a process is active, i.e., it carries the thread, when it contains an unguarded output at a higher-order name. When encoding λ-calculi with first-order references, an active process may also expose the thread by exhibiting an input at a first-order name. We follow such a convention also in the paper: higher-order names carry the thread via outputs, first-order names via inputs (other choices technically would however be possible). Sequentiality allows us to expose, in isolation, the causality properties of service calls, within a standard (interleaving) semantics. These aspects would be partially or totally obscured by the presence of other concurrent threads.

In higher-order languages, the effects of having first-order store are particularly strong in absence of control operators; that is, using a terminology borrowed from game semantics, in a "well-bracketed" setting, where the call-return interaction behaviour between a term and its context follows a stack discipline. We therefore also consider the refinement of our visibility-based type system under the well-bracketing hypothesis. To see an example of the effect of visibility, consider the following well-bracketed example:

$$\boldsymbol{\nu} h\,(\,\overline{h}\langle 0\rangle \;\mid\; \overline{a}(c,q).\,(!c.\,h(z).\,\overline{h}\langle z+1\rangle \mid q(y).\,Q)\,)$$

Here, an external service a is called, exporting a local service c capable of incrementing a private reference h, and with a return channel q. If visibility holds, then the external environment cannot store c. Therefore, after providing an answer at q , the environment is unable to call c again, thus incrementing h, unless an access to c is explicitly provided again (by process Q). This property may be used to perform optimisations, possibly including garbage-collection of the service c if at some point c is not used anymore in Q.

We then study characterisations for both may testing and barbed equivalence, as special forms of trace equivalence and of labelled bisimilarity. Following the tradition of testing, the language includes special success signals, that are used by testers (i.e, processes running in parallel with the testees), and may be used at any time, to report success.

When developing such characterisations, some care is necessary so to distinguish between the visibility of the tested and of the tester processes, as they need not be the same. For this, we introduce visibility-constrained LTSs, in which process transitions are constrained by a visibility function, intuitively expressing the possibility for processes to perform a certain transition given a certain visibility function for the external observer. As a consequence of the transition, the visibility function may need to be updated, so to yield the visibility function for the next transition. In the case of well-bracketing, the constraints given by visibility have

to be coupled with those given by the stack names used to track well-bracketing (and again, a distinction has to be made between the stacks for the testers and those for the testees). Based on these LTSs, we introduce forms of trace equivalence and of labelled bisimilarity that are proved to be sound and complete, for may testing and barbed equivalence, both in the case of sequentiality and in that of well-bracketing.

We present the version of the π-calculus with higher-order and first-order names in Section 2. In Section 3, we define the type system implementing visibility for sequential processes. We define typed labelled transitions and present labelled characterisations of behavioural equivalences in Section 4. Section 5 is devoted to the extension with well-bracketing. For lack of space, the results about labelled bisimilarities are presented in [4].

2 Background: the π-Calculus Language

The π-calculus language that will be used is called, technically, Internal π (πI) [22] (all channels exchanged are fresh – no free object outputs), with the addition of *first-order values* (such as integers and booleans), and expressions that evaluate to first-order values, called *first-order expressions*. We do not specify what exactly first-order values and first-order expressions are. We simply assume that, in a closed process, a first-order expression evaluates to a first-order value. Moreover, only the output capability of names may be exported. This means that, for instance, in an input $a(b).P$ name b may only be used in output in P. This constraint is often followed in theory and applications of the π-calculus, as it simplifies various technical matters.

We assume that names are distinguished (partitioned) into *higher-order* names and *first-order* names, and *success names*. Higher-order names are the ordinary πI names; first-order names may only carry first-order values, and they may not be passed around (the second constraint simplifies the technical details, though it is not mandatory, see Section 3). In later sections, first-order names will be used to represent references. The fact that these names may only carry first-order values, together with constraints on visibility, will ensure that the calculus does not have (implicit or explicit forms of) higher-order store. The *success* names are used, in the tradition of testing equivalences, by tester processes so to report results of experiments on tested processes (thus, the processes compared in the examples of the paper do not contain success names).

The calculi in the paper will be typed. For simplicity we define our type systems as refinements of the most basic type system for π-calculus, namely Milner's *sorting* [15], in which names are partitioned into a collection of *types* (or sorts), and a sorting function maps types onto types. We assume that there is a sorting system under which all processes we manipulate are well-typed. Thus, the sorting system separates higher-order and first-order names; moreover we assume that first-order names are monadic, and that higher-order names carry a pair of a first-order value and a higher-order name. Thus $\overline{a}\langle v\rangle(b).P$ is the output of the first-order value v and of the fresh higher-order name b at the (higher-order) name a, with continuation P. When writing examples, we allow different sortings; for instance, writing $a.P$ and $\overline{b}.Q$ for inputs and outputs in which no value is exchanged.

The syntactic categories of the calculus are presented in Table 1. As usual, input and restriction are binding constructs, and give rise to the expected definitions of free and bound names. An expression or a process is *closed* if it does not have free (first-order) variables. It is intended that transitions are defined on closed processes.

The definition of structural congruence, written $\equiv$, and of the strong and weak (untyped) labelled transitions, written $\xrightarrow{\mu}$, $\Longrightarrow$, and $\stackrel{\hat{\mu}}{\Longrightarrow}$, are standard and are given in Appendix A (later we will introduce typed variants of the LTS). We note $\mathtt{fn}(P)$ (resp. $\mathtt{fn}(\mu)$) the set of free names of P (resp. μ). We sometimes abbreviate reductions $P \xrightarrow{\tau} P'$ as $P \longrightarrow P'$.

Table 1 The syntax of the calculus.

$$\begin{array}{rrrrr} [\textit{Higher-order names, } \texttt{HO}] & a,b,c & & [\textit{First-order expressions}] & e,f \\ [\textit{First-order names, } \texttt{FO}] & h,k & & [\textit{First-order variables}] & x,y \\ [\textit{Success names}] & \omega & & [\textit{First-order values}] & v,w \end{array}$$

$$\begin{array}{rrcl} [\textit{Inputs}] & \alpha & ::= & a(x,b) \mid h(x) \\ [\textit{Outputs}] & \overline{\alpha} & ::= & \overline{a}\langle e\rangle(b) \mid \overline{h}\langle e\rangle \mid \omega \\ [\textit{Processes}] & P,Q,R & ::= & \alpha.P \mid \overline{\alpha}.P \mid !\alpha.P \mid P \mid Q \mid \boldsymbol{\nu} a\, P \mid \boldsymbol{\nu} h\, P \\ & & & \mid \mathbf{0} \mid P+Q \mid \texttt{if } e=f \texttt{ then } P \texttt{ else } Q \end{array}$$

As behavioural equivalences we consider both may-testing and barbed equivalence. Both relations are contextual, in that they are defined by requiring a closure under all "observers". They both use "barbs" (or "success") signals; barbed equivalence, in addition, makes use of a bisimulation game on reductions. We write $\Downarrow_\omega$ to indicate the possibility of exhibiting an ω barb, possibly after some reductions; i.e., $P \Downarrow_\omega$ holds if there is P' with $P \Longrightarrow P'$ and P' has an unguarded occurrence of ω. Thus intuitively two processes are behaviourally equal if they can produce "the same barbs" under all legal observers running in parallel with them. As we are in a typed setting, observer and tested processes should have compatible typings. The meaning of "compatible" depends on the specific type system. For visibility, the meaning will be specified in Sections 3 and 5. We write $\Delta \vdash P$, and say that P is a Δ-*process*, if P is typable under Δ, and $\Delta \vdash P, Q$ if both $\Delta \vdash P$ and $\Delta \vdash Q$ hold. Behavioural equivalences are defined on closed processes; they are extended to open processes (i.e., processes with free first-order variables) in the usual manner, by considering all possible closing substitutions. We only consider *weak* behavioural equivalences (i.e., equivalences that abstract from internal moves of processes, the reductions), as these are, in practice, the most useful ones.

▶ **Definition 1** (May testing)**.** Suppose that Γ, Δ are compatible. Then two Δ-processes P and Q are *may testing equivalent at* Γ, written $P \cong^{\Gamma}_{\Delta} Q$, if for all Γ-process R and success name ω, we have $(R \mid P) \Downarrow_\omega$ iff $(R \mid Q) \Downarrow_\omega$.

▶ **Definition 2** (Barbed bisimilarity and equivalence)**.** *Barbed bisimilarity* is the largest symmetric relation $\dot{\approx}$ s.t. $P \dot{\approx} Q$ implies:

1. whenever $P \longrightarrow P'$, there exists Q' such that $Q \Longrightarrow Q'$ and $P' \dot{\approx} Q'$;
2. for each ω, if $P \Downarrow_\omega$ then also $Q \Downarrow_\omega$.

Suppose that Γ, Δ are compatible. Then two Δ-processes P and Q are *barbed equivalent at* Γ, written $P \approx^{\Gamma}_{\Delta} Q$, if for all R with $\Gamma \vdash R$ it holds that $P \mid R \dot{\approx} Q \mid R$.

We write $P \cong Q$ (resp. $P \approx Q$) when $P \cong^{\Gamma}_{\Delta} Q$ (resp. $P \approx^{\Gamma}_{\Delta} Q$) holds under any typing Δ for P, Q and any Γ compatible with Δ.

We now introduce some notations that will be useful below. In the paper we use partial functions whose codomains are finite powersets. If Δ is such a function, then $\Delta; i \mapsto A$ is the extension of Δ in which i is mapped onto A, assuming that i is not in $\texttt{dom}(\Delta) \cup \texttt{codom}(\Delta)$; conversely Δ^{-i} is the restriction of Δ in which the entry for i is erased, both in the domain and codomain of Δ. We also write $i \mapsto A, j$ as an abbreviation for $i \mapsto (A \cup \{j\})$; and $\Delta \lfloor i$ (resp. $\Delta \not\lfloor i$) means that Δ is (resp. is not) defined on i. Function $\Delta_1 \cap \Delta_2$ has domain $\texttt{dom}(\Delta_1) \cap \texttt{dom}(\Delta_2)$, and then is defined as $(\Delta_1 \cap \Delta_2)(p) = \Delta_1(p) \cap \Delta_2(p)$; and likewise for $\Delta_1 \cup \Delta_2$. We write $\Delta_1 \subseteq \Delta_2$ if $\texttt{dom}(\Delta_1) \subseteq \texttt{dom}(\Delta_2)$, and for each $i \in \texttt{dom}(\Delta_1)$, we have $\Delta_1(i) \subseteq \Delta_1(i)$.

3 A Type System for Visibility

Our type system for visibility is built on top of that for sequentiality [3]. While visibility could be defined as a standalone notion, the combination with sequentiality makes the behavioural effects sharper and easier to grasp.

Intuitively, sequentiality ensures us that at any time at most one interaction can occur; i.e., there is a single computation thread. A process that holds the thread, called *active*, decides what the next interaction can be. It does so by offering a single particle (input or output) that controls the thread. A given name may exercise the control on the thread either in output or in input. Following [3], we require that, in higher-order names, the thread is carried by outputs, whereas, in first-order names, it is carried by inputs. An input that carries the thread maintains the thread, whereas an output releases it. (These appear to be, practically, the most interesting combinations; other combinations could be adopted, with the expected modifications.) Thus, for instance, the following process P is well-typed:

$$P \stackrel{\text{def}}{=} \big(h(x).\,\overline{a}\langle x\rangle(b).\,(b(z,c').\,Q \mid \overline{h}\langle x\rangle)\big) \;\mid\; \overline{k}\langle v\rangle$$

The initial particles in P are the input at h and the output at k; however only the input at h carries the thread, as h and k are first-order. Underneath the input, the thread is maintained and released in the output at a (the thread will be acquired by the process in the external environment performing the input at a); later, when the input at b is performed, process Q obtains the thread back. Sequentiality however does not exclude non-determinism: e.g., an output particle may have the possibility of interacting with different inputs.

The type system for *visibility* intuitively allows us to specify which possible higher-order outputs can be performed by a process that becomes active. Type environments are partial functions, called *visibility functions*, acting on higher-order names and, for active processes, on the special token $*$, representing the active thread. (Later, when studying well-bracketing, type environments will have both a visibility function and a name stack.)

▶ **Definition 3** (Visibility function)**.** A *visibility function* is a finite partial function $\Delta : \mathtt{HO} \cup \{*\} \to \mathcal{P}_{\mathtt{fin}}(\mathtt{HO})$. Moreover the function should be *transitive*, that is, for all $o \in \mathtt{dom}(\Delta)$, if $\Delta \downharpoonright a$ and $a \in \Delta(o)$, then $\Delta(a) \subseteq \Delta(o)$.

To see the use of $*$, suppose $\Delta(*) = \{a, b\}$; this means that a Δ-process is active, and may only perform outputs at a and b, as in the following process:

$$\overline{a}\langle v\rangle(c).\,c(d).\,\overline{b}(w).\,\mathbf{0}$$

Name c is created in the output at a and therefore c inherits a's visibility.

Similarly, if $\Delta(d) = \{a, b\}$ then, in a Δ-process $d(x, c).\,P$, the continuation P of the input at d may only perform outputs at a, b or at c (as c is the higher-order name received in the input). The use of visibility functions also allows us to rule out implicit forms of higher order store; in the following example, p, q are higher order names used for continuations.

When extending Δ, we impose in $\Delta; a \mapsto A$ that $a \notin \mathtt{dom}(\Delta) \cup \mathtt{codom}(\Delta)$, that is, a is fresh for Δ. Similarly, $\Delta; * \mapsto A$ is defined only when $* \notin \mathtt{dom}(\Delta)$.

▶ **Example 4** (Implicit forms of higher-order store)**.** In the λ-calculus, if a term without higher-order references is well-bracketed then it necessarily obeys visibility. In our calculus, even though we do not have explicit higher-order references, we can simulate higher-order store. Consider

$$P \stackrel{\text{def}}{=} \overline{a}(b, p).\,b(n, c, q).\,\overline{q}\langle n\rangle.\,p(m).\,\overline{c}.\,\mathbf{0}$$

$$\texttt{O-HO}\ \frac{a \in \Delta(*) \quad \Delta^{-*}; b \mapsto \Delta(*), b \vdash P}{\Delta \vdash \overline{a}\langle e\rangle(b).\, P} \qquad \texttt{I-HO}\ \frac{\Delta; * \mapsto \Delta(a), b \vdash P}{\Delta \vdash a(x,b).\, P}$$

$$\texttt{O-FO}\ \frac{\Delta \vdash P \quad \Delta \not\downarrow *}{\Delta \vdash \overline{h}\langle e\rangle.\, P} \qquad \texttt{I-FO}\ \frac{\Delta \vdash P \quad \Delta \downarrow *}{\Delta \vdash h(x).\, P} \qquad \texttt{REP}\ \frac{\Delta \vdash \alpha.\, P \quad \Delta \not\downarrow *}{\Delta \vdash !\alpha.\, P} \qquad \texttt{SUCC}\ \frac{}{\Delta \vdash \omega.\, P}$$

$$\texttt{RES-HO}\ \frac{\Delta \vdash P}{\Delta^{-a} \vdash \nu a P} \qquad \texttt{RES-FO}\ \frac{\Delta \vdash P}{\Delta \vdash \nu h P} \qquad \texttt{NIL}\ \frac{\Delta \not\downarrow *}{\Delta \vdash \mathbf{0}} \qquad \texttt{SUM}\ \frac{\Delta \vdash P_1 \qquad \Delta \vdash P_2}{\Delta \vdash P_1 + P_2}$$

$$\texttt{PAR}\ \frac{\Delta_1 \vdash P_1 \qquad \Delta_2 \vdash P_2 \qquad (\Delta_1, \Delta_2) \in \texttt{split}(\Delta)}{\Delta \vdash P_1 \mid P_2} \qquad \texttt{IF}\ \frac{\Delta \vdash P_1 \quad \Delta \vdash P_2}{\Delta \vdash \texttt{if}\ e = f\ \texttt{then}\ P_1\ \texttt{else}\ P_2}$$

Figure 1 Typing rules for visibility.

In P the visibility $\Delta(p)$ is defined when the output at a is made, therefore $c \notin \Delta(p)$, and the final output at c breaks visibility; indeed, the input at b, which justifies the output at c, is answered by the output at q before the output at c.

Transitivity ensures us that the visibility functions, viewed as "named sets", represent transitive sets. Transitivity is needed to guarantee the behavioural expectations of visibility. If a process P calls a service a, with actual (higher-order) parameter b, then an output triggered by such a call (and possibly directed back to P) should be either at b or at a name in the visibility of a; however, the server a might call an external service c; this call should not allow an increase of the visibility associated to a; hence the transitivity constraint that the visibility of c is contained in that of a. Technically, transitivity is necessary to ensure preservation of typing under reduction (see Theorem 10 and Example 11).

▶ **Remark 5** (Transitivity and functional languages)**.** The issue of transitivity does not arise in the semantics of purely functional languages such as the λ-calculus [7]: visibility functions do appear, but the domain and codomain of such functions are disjoint (technically, in the game semantics terminology, the domain contains names introduced by Player, whereas the codomain contains names introduced by Opponent). This makes transitivity trivial.

In the typing of a parallel composition, the visibility function has to be split between the two components of the composition. This is important for elements of the domain of the function that are linear, and therefore may only appear in one, but not both, components; this is the case for $*$, and the linear continuation names that will be introduced in Section 5.

▶ **Definition 6** (Split)**.** Let Δ be a visibility function, we define $\texttt{split}(\Delta)$ as the set of pairs (Δ_1, Δ_2) such that $\Delta_1 \cup \Delta_2 = \Delta$ and $* \notin \texttt{dom}(\Delta_1) \cap \texttt{dom}(\Delta_2)$.

The typing rules are given in Table 1. We comment on a few of them. Rule `O-HO` specifies that an output at a `HO` name a owns the thread, and that name a must be in the current visibility. In the continuation P (which is not active) the visibility for b is determined by the current visibility $\Delta(*)$ together with b itself, to allow for recursive calls at b. Symmetrically, in rule `I-HO`, a process performing an input at a acquires the thread, with a visibility given by $\Delta(a)$ plus the received name b. A replicated input is well-typed only if it is not active (rule `REP`). When typing parallel composition, we use `split` to insure that at most one of the components holds the thread. We simply remove the name from Δ when typing a restriction (rule `RES-HO`). Visibility plays no role in rules `I-FO` and `O-FO`; the only checks made are given by sequentiality, concerning the respect of the thread.

▶ **Example 7.** Consider

$$P \stackrel{\text{def}}{=} \begin{array}{l} a(x,b).\,\overline{b}\langle x+1\rangle(d).\,a(x',b').\,\overline{b}\langle x+x'\rangle(d').\,\mathbf{0} \\ \mid\ a(x_1,b_1).\,\overline{b_1}\langle x_1\rangle(d_1).\,d_1(d_2).\,\overline{c}\langle x+1\rangle(c_1).\,\mathbf{0} \end{array}$$

The process has three inputs at a; the first two are sequential, and make no requirements on visibility, as their continuations only perform outputs at names received in the input; the third one, in addition, has the effect of creating a name d_1, whose input may activate an output at the free name c. This input requires that in the visibility Δ of P, we find $c \in \Delta(a)$.

▶ **Example 8.** (In this example, we ignore the first-order components of actions, as they are not relevant.) A process of the form

$$P \stackrel{\text{def}}{=} \overline{a}(b).\,(Q \mid c(d).\,\overline{b}(d').\,R)$$

is not typable: the initial visibility of c may not include b, which is created fresh by a; yet, P may perform an output at b after the input at c. Indeed, while P does not directly create a form of higher-order store, it may do so indirectly: the external environment, after completing the treatment of the initial call at a, is supposed to "forget" the higher-order name b. However, this is not the case, as the environment may regain access to b by invoking the (unrelated) name c. In contrast P would become typable (in the straightforward extension of our type system with polyadicity) if the initial output at a would be replaced by $\overline{a}(b,c)$: now c becomes a fresh name, initially unknown, and created in the initial output, together with b, so that the visibility for b and c can be the same and can include both names.

The following lemma shows that a visibility function may always be enlarged (both in the domain and codomain), as long as sequentiality (i.e., being defined on $*$) is respected.

▶ **Lemma 9** (Weakening). *Suppose $\Delta_1 \vdash P$, $\Delta_1 \subseteq \Delta_2$, and ($\Delta_2\downarrow * $ iff $\Delta_1\downarrow *$). Then $\Delta_2 \vdash P$.*

Some other technical lemmas that are used in proofs are presented in Appendix B.1.

▶ **Theorem 10** (Subject reduction). *Suppose $\Delta \vdash P$ and $P \xrightarrow{\mu} P'$. We have:*

- *if μ is a τ action or a first-order input, then $\Delta \vdash P'$; the same holds if μ is a first-order output and $\Delta \not\downarrow *$;*
- *if $\mu = \overline{a}\langle v\rangle(b)$ then $a \in \Delta(*)$ and $\Delta^{-*}; b \mapsto \Delta(*), b \vdash P'$;*
- *if $\mu = a\langle v\rangle(b)$ and $\Delta \not\downarrow *$, then $\Delta; * \mapsto \Delta(a), b \vdash P'$.*

▶ **Example 11.** Transitivity of visibility functions is necessary. Indeed, consider process $P \stackrel{\text{def}}{=} a.\,\overline{b}.\,Q \mid \overline{a}$, and the *non-transitive* function $\Delta = a \mapsto \{a,b\}, * \mapsto a$. We have $\Delta \vdash P$. However $P \xrightarrow{\tau} \overline{b}.\,Q$, with $\Delta \not\vdash \overline{b}.\,Q$, violating Subject Reduction (Theorem 10).

To adapt the generic definitions of behavioural equivalence in Section 2 to the type system for visibility, we only have to specify the meaning of compatibility between typing environments (which in our case are visibility functions). Compatibility means ensuring that processes typed under the two typing environments cannot both be active.

▶ **Definition 12.** Two typings Δ_1 and Δ_2 are *compatible* if $* \notin \mathtt{dom}(\Delta_1) \cap \mathtt{dom}(\Delta_2)$.

We discuss some examples of equivalences in which the behavioural constraints imposed on legal observers by visibility are essential: the equivalences would otherwise be broken.

▶ **Example 13.** Consider a process

$$P \stackrel{\text{def}}{=} \overline{a}(b,c).\,P' \qquad \text{for} \qquad P' \stackrel{\text{def}}{=} (b(d).\,R) \mid c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\overline{b}(d').\,Q$$

where we assume b does not occur in Q and R. Process P interrogates an external service a, providing access to local services b and c, and the derivative is P'. Now, P' may either be interrogated at b, or at c. In the former case, the output $\overline{b}(d').\,Q$ becomes dead code (as the environment may not install any input at b). In the latter case, the environment is again called at a; now, since visibility ensures us the environment cannot remember b, the input and the output at b in P' may only interact with each other; we can therefore inline the call at b. We can express this behaviour using expansion as follows:

$$P \cong \overline{a}(b,c).\Big(\begin{array}{rl} & b(d).\,(R \mid c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\mathbf{0}_*) \\ + & c(c_1).\,\overline{a}(b',c').\,c'(c_2).\,\tau.\,\boldsymbol{\nu} d'\,(Q \mid R\{^{d'}/_{d}\})\end{array}\Big)$$

where $\mathbf{0}_*$ indicates an active process that cannot perform any action (e.g., $\boldsymbol{\nu} b\,_2(\overline{b_2}.\,\mathbf{0})$). (This equality holds under any visibility, hence we use the symbol $\cong$.)

If we replaced the second output at a with an output at a different free name, say a', then the processes could still be typable, and the equalities above would still hold. However, the equalities would not hold if we replaced the second output at a with an output at c_1. Indeed, while the environment may not store b or c for using them in a later call at a (or at an unrelated name such as a'), the environment is allowed to use b and c within services that are created following the first call at a; for instance, the environment could be

$$a(b,c).\,\overline{c}(c_1).\,!c_1(c_1').\,\overline{b}(d'').\,\mathbf{0}$$

The following example illustrates the use of visibility to perform garbage-collection on names that will never be used. We refer to Appendix B.3 for the full definition of the corresponding pruning function.

▶ **Example 14.** Consider a typing environment Δ, a Δ-process

$$P \stackrel{\text{def}}{=} !a.\,Q_1 \mid !b.\,\overline{c}.\,!a.\,Q_2 \mid !d.\,\overline{a}.\,\mathbf{0}$$

and a visibility Γ compatible with Δ such that $\Gamma \cup \Delta$ is transitive. Γ is intended here to represent the observer's visibility. Suppose $a \notin \Gamma(*)$; that is, the observer is active but its current visibility does not include a (which may however be in the visibility of other names in Γ). We then have $P \cong^{\Delta}_{\Gamma} !b.\,\overline{c}.\,\mathbf{0}$. The pruning is possible because, intuitively, the transitivity of $\Delta \cup \Gamma$ ensures us that an observer, whenever active, cannot have a in its current visibility. For instance, if the observer interrogates b, then the transitivity of the visibility functions ensures us that $a \notin \Delta(b)$; then $c \in \Delta(b)$ (which is necessary for P to be typable) gives us also $a \notin \Gamma(c)$. Similarly, exploiting transitivity, we deduce that also $d \notin \Gamma(*)$, and that an observer may never be able to interrogate d; thus we can also remove the input on d.

4 Labelled Semantics

4.1 Trace-based proof techniques

The definitions of may-testing and barbed equivalence make use of contextual quantifications. We study here proof techniques for them, using labelled transitions, without such quantifications. These proof techniques will be expressed as forms of trace equivalence and labelled

$$\texttt{TrO-HO}\ \frac{P \xrightarrow{\overline{a}\langle v\rangle(b)} P' \quad a \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{\overline{a}\langle v\rangle(b)} [\Gamma; * \mapsto \Gamma(a), b]\, P'} \qquad \texttt{TrI-HO}\ \frac{P \xrightarrow{a\langle v\rangle(b)} P' \quad a \in \Gamma(*)}{[\Gamma]\, P \xrightarrow{a\langle v\rangle(b)} [\Gamma^{-*}; b \mapsto \Gamma(*), b]\, P'}$$

$$\texttt{TrO-FO}\ \frac{P \xrightarrow{\overline{h}\langle v\rangle} P' \quad \Gamma \downarrow *}{[\Gamma]\, P \xrightarrow{\overline{h}\langle v\rangle} [\Gamma]\, P'} \qquad \texttt{TrI-FO}\ \frac{P \xrightarrow{h\langle v\rangle} P' \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{h\langle v\rangle} [\Gamma]\, P'} \qquad \texttt{TrTau}\ \frac{P \xrightarrow{\tau} P' \quad \Gamma \not\downarrow *}{[\Gamma]\, P \xrightarrow{\tau} [\Gamma]\, P'}$$

Figure 2 Visible LTS.

bisimilarity. The definition of may-testing makes use of contextual quantifications. We study here proof techniques without such quantifications, as forms of trace equivalence. For lack of space, analogous proof techniques for barbed equivalence are only reported in [4]. The first step is to express the transitions for a process that are allowed by an observer subject to certain visibility constraints. We call these the *type-allowed transitions*. They are of the form $[\Gamma]\, P \xrightarrow{\mu} [\Gamma']\, P'$, to be read "an observer that has visibility Γ allows process P to make a transition μ and, as a result of the transition, the process becomes P' and the visibility of the observer becomes Γ'". Type-allowed transitions are defined by the rules in Figure 2. Intuitively, an observer typable under Γ can initially test inputs of processes at names in $\Gamma(*)$, if $\Gamma \downarrow *$, and outputs of processes in $\texttt{dom}(\Gamma)$ otherwise. The weak transitions $\Longrightarrow$ and $\xRightarrow{\mu}$, are defined in the usual manner from the strong one; for instance, $[\Gamma]\, P \xRightarrow{\mu} [\Gamma']\, P'$ holds if $[\Gamma]\, P \Longrightarrow [\Gamma]\, P''$, and $[\Gamma]\, P'' \xrightarrow{\mu} [\Gamma']\, P'''$, and $[\Gamma']\, P''' \Longrightarrow [\Gamma']\, P'$, for some P'' and P'''.

An action τ is called a *reduction* (or an *internal action*); the remaining actions are called *interactions*. We use $\mathcal{T}$ to range over *traces*, which are finite sequences of interactions $\mu_1 \dots \mu_n$ $(n \geq 0)$ in which the bound names are all distinct and different from the free names (the definition of bound and free names is as for processes, viewing a trace as a sequence of prefixes). A trace $\mu_1 \dots \mu_n$ is *trace of* P_0 *under visibility* Γ_0 if there are P_i, Γ_i with $[\Gamma_i]\, P_i \xRightarrow{\mu_i} [\Gamma_{i+1}]\, P_{i+1}$, for $0 \leq i < n$.

▶ **Definition 15** (Typed trace equivalence). Two Δ-processes P and Q are *trace equivalent at* Γ, written $P \eqsim^{\Gamma}_{\Delta} Q$, if Δ, Γ are compatible and P, Q have the same sets of traces under Γ.

We show the main compositionality lemmas, and a trace-based characterisation of may testing. Other technical lemmas needed in proofs may be found in Appendix B.2.

▶ **Lemma 16** (Restriction). *If* $P \eqsim^{\Gamma}_{\Delta} Q$ *then also* $\boldsymbol{\nu} a\, P \eqsim^{\Gamma^{-a}}_{\Delta^{-a}} \boldsymbol{\nu} a\, Q$.

▶ **Lemma 17** (Input). *If* $P \eqsim^{\Gamma^{-*}; b \mapsto \Gamma(*), b}_{\Delta; * \mapsto \Delta(a), b} Q$ *and then also* $a(x,b).\, P \eqsim^{\Gamma}_{\Delta} a(x,b).\, Q$.

▶ **Lemma 18** (Output). *If* $P \eqsim^{\Gamma; * \mapsto \Gamma(a), b}_{\Delta^{-*}; b \mapsto \Delta(*), b} Q$ *and then also* $\overline{a}\langle e\rangle(b).\, P \eqsim^{\Gamma}_{\Delta} \overline{a}\langle e\rangle(b).\, Q$.

▶ **Lemma 19** (Parallel composition). *Suppose* $P \eqsim^{\Gamma}_{\Delta} Q$ *and* $(\Gamma_1, \Gamma_2) \in \texttt{split}(\Gamma)$. *Then* $\Gamma_2 \vdash R$ *implies* $P \mid R \eqsim^{\Gamma_1}_{(\Delta \cup \Gamma_2)^+} Q \mid R$ *where* $(-)^+$ *indicates the transitive closure.*

In the proof of Lemma 19, a trace of $P \mid R$ is split into a trace for P and one for R. We have then to ensure that the condition of the lemma are maintained throughout the trace, notwithstanding modifications of the visibility functions. Some care is needed in case of synchronisations between P and R, where the typing of the external observer has to remain the same; we also use the restriction Lemma 16. Other cases also exploit the extension Lemma 36, and the transitivity of the visibility function, to make sure that the views on free names are not increased along the trace.

▶ **Theorem 20** (Characterisation of may testing using trace equivalence). *Suppose* $\Delta \vdash P, Q$ *then* $P \eqsim^{\Gamma}_{\Delta} Q$ *if and only if* $P \cong^{\Gamma}_{\Delta} Q$.

The sketch of the proofs is the same as for characterisations of may testing with trace equivalence in untyped calculi, exploiting substitutivity properties of trace equivalence in the case of Soundness, and reason by contradiction in the case of Completeness. In addition, one has to take care of the observer visibilities, how it changes, and use type-allowed transitions. A similar results of characterisation also hold for barbed equivalence, using a typed form of labelled bisimilarity, and are presented in [4].

We can use typed trace equivalence to prove equalities such as those in Examples 13 and 14; the proofs are simple, following the definition of the equivalence and exploiting type-allowed transitions. See also Appendix B.3 for the proof about the pruning function that generalises Example 14. (Similarly, the proofs can be carried out for barbed equivalence, using the labelled bisimilarity in [4].)

4.2 Visibility property on traces

We formalise the link between our type system and the notion of visibility in game semantics [7]. The reader not familiar with game semantics may safely skip the section. In game semantics, the notions of visibility and well-bracketing are defined based on a justification relation between moves performed by Player. These moves correspond to interactions on higher-order names in our setting, and the notion of justification can be formulated as follows.

▶ **Definition 21.** Let μ_1, μ_2 be two interactions at $\mathtt{HO}$ names. We say that μ_1 *justifies* μ_2, noted $\mu_1 \curvearrowleft \mu_2$, if: (i) either $\mu_1 = a\langle v\rangle(b)$ and $\mu_2 = \overline{b}\langle v'\rangle(c)$, or (ii) $\mu_1 = \overline{a}\langle v\rangle(b)$ and $\mu_2 = b\langle v'\rangle(c)$. When this is not the case, we write $\mu_1 \not\curvearrowleft \mu_2$.

$\mu_1 \curvearrowleft \mu_2$ captures the dependency induced by the fact that μ_2 performs an interaction at a name that has been previously introduced by μ_1. In contrast with the situation in game semantics, though, there is no notion of question and answer in our setting.

We can now define the notion of view. In the definition below, we allow a starting visibility A, because we want to consider processes that already have some knowledge about the observer (Opponent).

▶ **Definition 22** (View). Given $A \subseteq \mathtt{HO}$, we define the *view with starting visibility* A *of a trace* $\mathcal{T}$, written $\mathtt{view}_A(\mathcal{T})$, as follows:

$$\mathtt{view}_A(\varepsilon) = A \qquad \mathtt{view}_A(\mathcal{T}_1\mu_1\mathcal{T}_2\mu_2) = \mathtt{view}_A(\mathcal{T}_1) \cup \mathtt{nHO}(\mu_1) \cup \mathtt{nHO}(\mu_2) \text{ if } \mu_1 \curvearrowleft \mu_2$$

$$\mathtt{view}_A(\mu_0 \dots \mu_n) = A \cup \mathtt{nHO}(\mu_n) \text{ if } \forall 0 \leq i \leq n-1, \mu_i \not\curvearrowleft \mu_n$$

where $\mathtt{nHO}(\mu)$ denotes the set of higher-order names appearing in the action μ.

Views capture the idea that when an action is justified by another one, the names created in between must have been forgotten, otherwise this would mean that the process had the ability to store these names in order to reuse them. In the last clause in Definition 22, the action is justified by an action that happened before the beginning of the trace, therefore it has the starting visibility. In particular, $\mathtt{view}_\emptyset(\mathcal{T})$ yields the usual definition in game semantics.

▶ **Definition 23** (Respecting views). Consider $A \subseteq \mathtt{HO}$, a trace $(\mu_0 \dots \mu_n)$ *respect views under starting visibility* A if $\mathtt{fn}(\mu_0) \subseteq A$ and $\forall 1 \leq i \leq n, \mathtt{fn}(\mu_i) \subseteq \mathtt{view}_A(\mu_0 \dots \mu_{i-1})$

▶ **Lemma 24.** *Consider* Δ, Γ, P *such that* $\Delta \vdash P$, Γ *is compatible with* Δ, *and* $\Delta \cup \Gamma$ *is transitive. Let* $A = (\Gamma \cup \Delta)(*)$, *then every trace of* P *under visibility* Γ *respects views with starting visibility* A.

The idea of the proof is that for a trace $(\mu_0 \dots \mu_n)$ there exist $P_0, \dots, P_{n+1}$, $\Delta_0, \dots, \Delta_n$, $\Gamma_0, \dots, \Gamma_{n+1}$ such that $P_0 = P$, $\Delta_0 = \Delta$, $\Gamma_0 = \Gamma$ and for all $0 \leq i \leq n$, $[\Gamma_i]\, P_i \xrightarrow{\mu_i} [\Gamma_{i+1}]\, P_{i+1}$ and $\Delta_i \vdash P_i$. We show by induction that for each $1 \leq i \leq n$, $(\Gamma_i \cup \Delta_i)(*) \subseteq \mathtt{view}_A(\mu_0 \dots \mu_{i-1})$; it thus follows that the trace respects views under starting visibility A. The condition of $\Delta \cup \Gamma$ being transitive is crucial, otherwise $(\Gamma_i \cup \Delta_i)(*)$ could grow in unwanted ways.

5 Adding Well-Bracketing

5.1 Type system

In the previous sections we have studied visibility in connection with sequentiality. In this section we do the same for a refinement of sequentiality, namely well-bracketing. We begin by reviewing well-bracketing, following [3], and then outline how visibility is accommodated.

From sequentiality to well-bracketing. In languages without control operators, well-bracketing intuitively means that return-call interactions among terms follow a stack-based discipline. In a well-bracketed system, a service that is interrogated, say S, acquires the thread and is supposed to return a final result (unless the computation diverges) thus releasing the thread. During its computation, S may however interrogate another service, say T, which, upon completion of its computation, will return the result to S; service S must receive the answer from T before returning the final result. The implementation of this policy requires *continuation names*, that are used to handle the return messages. When a service is interrogated, it receives a fresh continuation name that is used once, in output, to deliver the final result. Accordingly, a client that has interrogated the service will use the continuation name in input to receive the result. Thus continuation names are *linear* [11] – they may only be used once. In contrast with [3], we do not require *receptiveness* [23] – the input-end of the name needs not be made available as soon as the name is created. Enforcing only linearity allows us to gain flexibility. To ensure the well-bracketing policy, and linearity of continuation names, the type system in [20] makes use of stacks.

▶ **Definition 25** ([20])**.** A stack, σ, is a sequence of input- and output- tagged continuation names, in which the input and output tags alternate:

$$\sigma ::= \sigma_{\mathtt{O}} \mid \sigma_{\mathtt{I}} \qquad \sigma_{\mathtt{O}} ::= p^{\mathtt{O}}, \sigma_{\mathtt{I}} \mid \emptyset \qquad \sigma_{\mathtt{I}} ::= p^{\mathtt{I}}, \sigma_{\mathtt{O}} \mid \emptyset$$

Moreover: a name may appear at most once with a given tag; and, if a name appears with both tags, then the input occurrence should immediately follow the output occurrence.

We write $p^{\mathtt{O}} \in \sigma$ (resp. $p^{\mathtt{I}} \in \sigma$) if there is an occurrence of $p^{\mathtt{O}}$ (resp. $p^{\mathtt{I}}$) in σ. We write $p \in \sigma$ if either $p^{\mathtt{O}} \in \sigma$ or $p^{\mathtt{I}} \in \sigma$.

Intuitively, a stack expresses the expected usage of the free continuation names in a process. For instance, if P can be typed using stack $p_1 : \mathtt{o}, p_2 : \mathtt{i}, p_3 : \mathtt{o}, p_3 : \mathtt{i}, p_4 : \mathtt{o}$, then $p_1, .., p_4$ are the free continuation names in P; among these, p_1 will be used first, in an output; then p_2 will be used, in an input interaction with the environment. P possesses both the output and the input capability on p_3, so it shall perform a reduction at p_3; the computation for P terminates with an output at p_4. This behaviour however concerns only the free continuation names of P: at any time when an output usage is expected, P may decide to create a new continuation name and send it out, maintaining its input end.

$$\texttt{O-ser}\ \frac{a \in A \quad \langle \Delta; b \mapsto A, b; p \mapsto A, b \mid p^{\mathrm{I}}, \sigma\rangle \vdash P}{\langle \Delta; * \mapsto A \mid \sigma\rangle \vdash \overline{a}\langle e\rangle(b,p).\, P} \qquad \texttt{O-con}\ \frac{p \in A \quad \langle \Delta; b \mapsto A, b \mid \sigma\rangle \vdash P}{\langle \Delta; * \mapsto A \mid p^{\mathrm{O}}, \sigma\rangle \vdash \overline{p}\langle e\rangle(b).\, P}$$

$$\texttt{I-ser}_1\ \frac{\langle \Delta; a \mapsto A; * \mapsto A, b, p \mid p^{\mathrm{O}}, \sigma\rangle \vdash P}{\langle \Delta; a \mapsto A \mid \sigma\rangle \vdash a(x,b,p).\, P} \qquad \texttt{I-ser}_2\ \frac{\langle \Delta; a \mapsto A; * \mapsto A, b, p \mid p^{\mathrm{O}}\rangle \vdash P}{\langle \Delta; a \mapsto A \mid \emptyset\rangle \vdash\, !a(x,b,p).\, P}$$

$$\texttt{I-con}\ \frac{\langle \Delta; * \mapsto A, b \mid q^{\mathrm{O}}\rangle \vdash P}{\langle \Delta; p \mapsto A \mid p^{\mathrm{I}}, q^{\mathrm{O}}\rangle \vdash p(x,b).\, P} \qquad \texttt{RES-ser}\ \frac{\langle \Delta \mid \sigma\rangle \vdash P}{\langle \Delta^{-a} \mid \sigma\rangle \vdash \nu a P}$$

$$\texttt{RES-con}_1\ \frac{\langle \Delta \mid \sigma_1, p^{\mathrm{O}}, p^{\mathrm{I}}, \sigma_2\rangle \vdash P}{\langle \Delta^{-p} \mid \sigma_1, \sigma_2\rangle \vdash \nu p P} \qquad \texttt{RES-con}_2\ \frac{\langle \Delta \mid \sigma\rangle \vdash P \quad p \notin \sigma}{\langle \Delta^{-p} \mid \sigma\rangle \vdash \nu p P}$$

$$\texttt{PAR}\ \frac{\begin{array}{c}\langle \Delta_1 \mid \sigma_1\rangle \vdash P_1 \quad (\Delta_1, \Delta_2) \in \texttt{split}(\Delta) \\ \langle \Delta_2 \mid \sigma_2\rangle \vdash P_2 \quad \sigma \in \texttt{inter}(\sigma_1, \sigma_2)\end{array}}{\langle \Delta \mid \sigma\rangle \vdash P_1 \mid P_2} \qquad \texttt{NIL}\ \frac{\Delta \not/ *}{\langle \Delta \mid \emptyset\rangle \vdash \mathbf{0}}$$

Figure 3 Typing rules with visibility and stack.

Visibility. We now discuss how to combine the visibility in Section 3 with well-bracketing. Higher-order names are partitioned into the set SER of *server names*, and the set CON of *continuation names*. Now a, b, c range over server names, while p, q, r range over continuation names. By convention, server names carry a triple consisting of a first-order value, a server name and a continuation; whereas continuations carry a first-order value and a server name. Accordingly, outputs on higher-order names are of the form $\overline{a}\langle n\rangle(b,p)$ and $\overline{p}\langle n\rangle(b)$, and similarly for inputs. First-order names are as before.

Visibility functions are then transitive partial functions from $\texttt{SER} \cup \texttt{CON} \cup \{*\}$ to $\mathcal{P}_{\texttt{fin}}(\texttt{SER} \cup \texttt{CON})$. These functions handle server and continuation names essentially like HO names in Section 3.

Typing environments include a stack; thus a typing environment is a pair $\langle \Delta \mid \sigma\rangle$ of a visibility function and a stack. In the typing rule for parallel composition, we rely on splitting (defined as in Section 3) as well as on $\texttt{inter}(\sigma_1, \sigma_2)$, the interleaving of stacks σ_1 and σ_2.

▶ **Definition 26** (Interleaving). We write $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$ if (i) σ_1 is a stack, and (ii) σ is an interleaving of σ_2 and σ_3 as by the following inductive rules:

1. $\emptyset \in \texttt{inter}(\emptyset; \emptyset)$
2. $p : \texttt{o}, \sigma_1 \in \texttt{inter}(\sigma_2; \texttt{p} : \texttt{o}, \sigma_3)$ if $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$
3. $p : \texttt{o}, \sigma_1 \in \texttt{inter}(\texttt{p} : \texttt{o}, \sigma_2; \sigma_3)$ if $\sigma_1 \in \texttt{inter}(\sigma_2; \sigma_3)$
4. the same as (2) and (3) with $p : \texttt{i}$ instead of $p : \texttt{o}$

If a name appears both in σ_2 and in σ_3 with the same tag, then $\texttt{inter}(\sigma_2; \sigma_3)$ is empty. Figure 3 gives the typing rules. The rules combine the handling of visibility and the update of the stack. For instance, in rule O-ser, newly created names b and p are added to the visibility when typing the continuation P, and p^{I} is added on top of the stack, meaning that P shall perform an input at p before interacting on other continuation names. The output capability on p is transmitted along the output at a; correspondingly, in rules $\texttt{I-ser}_1$ and $\texttt{I-ser}_2$, p^{O} is added to σ. In contrast with [3], we do not impose $\sigma = \emptyset$ in rule $\texttt{I-ser}_1$, as we do not enforce input receptiveness on continuation names. Performing inputs and outputs at continuation names has the effect of consuming the corresponding particle on top of the stack. The stack is empty when typing a replicated input, so to avoid duplication of continuation names. Rules SUCC, O-FO, I-FO, RES-FO, SUM and IF are like in Figure 1, and are omitted.

▶ **Theorem 27** (Subject reduction). *Suppose* $\langle \Delta \mid \sigma \rangle \vdash P$ *and* $P \xrightarrow{\mu} P'$. *We have:*

- *if* $\mu = \tau$, *then either* $\sigma = p^{\mathcal{O}}, p^{\mathcal{I}}, \sigma''$ *with* $p \in \mathtt{Ncon} \setminus \mathtt{fn}(P')$, *in which case* $\langle \Delta^{-p} \mid \sigma'' \rangle \vdash P'$, *or* $\langle \Delta \mid \sigma \rangle \vdash P'$;
- *if* $\mu = \overline{a}\langle v \rangle(b, p)$ *then* $a \in \Delta(*)$ *and* $\langle \Delta^{-*}; b \mapsto \Delta(*), b; p \mapsto \Delta(*), b \mid p^{\mathcal{I}}, \sigma \rangle \vdash P'$;
- *if* $\mu = \overline{p}\langle v \rangle(b)$ *then* $\sigma = p^{\mathcal{O}}, \sigma''$, $p \in \Delta(*)$ *and* $\langle \Delta^{-*}; b \mapsto \Delta(*), b \mid \sigma'' \rangle \vdash P'$;
- *if* $\mu = a\langle v \rangle(b, p)$ *and* $\Delta \not\downarrow *$ *then* $\langle \Delta; * \mapsto \Delta(a), b, p \mid p^{\mathcal{O}}, \sigma \rangle \vdash P'$;
- *if* $\mu = p\langle v \rangle(b)$ *and* $\Delta \not\downarrow *$ *then* $\sigma = p^{\mathcal{I}}, q^{\mathcal{O}}, \sigma''$ *and* $\langle \Delta^{-p}; * \mapsto \Delta(p), b \mid q^{\mathcal{O}}, \sigma'' \rangle \vdash P'$;
- *if* $\mu = h\langle v \rangle$ *or* $(\mu = \overline{h}\langle v \rangle$ *and* $\Delta \not\downarrow *)$ *then* $\langle \Delta \mid \sigma \rangle \vdash P'$.

As previously, we have to define compatibility of typing environments so to be able to use the contextual forms of behavioural equivalence from Section 2.

▶ **Definition 28.** $\langle \Delta_1 \mid \sigma_1 \rangle$ and $\langle \Delta_2 \mid \sigma_2 \rangle$ are compatible if Δ_1 and Δ_2 are compatible (as by Definition 12) and $\mathtt{inter}(\sigma_1, \sigma_2) \neq \emptyset$.

▶ **Example 29.** Consider the process

$$\begin{aligned} P \stackrel{\text{def}}{=} \boldsymbol{\nu} h \,(\ & \overline{h}\langle 5 \rangle \mid \\ & \overline{a}(b, q).\,(\ !b(p).\, h(x).\, (\overline{h}\langle x+1 \rangle.\, \mathbf{0} \mid \overline{p}.\, \mathbf{0}) \mid \\ & \qquad q.\, h(x).\, \overline{h}\langle x \rangle.\, \overline{c}(d, r).\, r.\, h(y).\, \overline{h}\langle y \rangle.\, \mathtt{if}\ x = y\ \mathtt{then}\ \overline{s}.\, \mathbf{0}\ \mathtt{else}\ \mathbf{0}_*)) \end{aligned}$$

Here h is used as an integer reference, initially set to 5. The process calls a service a, giving access to a name b that can be used to increment the reference. When the call at a returns (at q), the reference is read twice and, in between, a call at c is made. Without visibility, the two values read for h might be different: as b has been exported, the environment could freely use b and increment the reference in between the two reads. Visibility ensures us that, when the initial call at a is returned, name b is "forgotten" by the environment, which cannot therefore use it anymore. Thus, if Q is the process defined like P but with the last line replaced with $q.\, \overline{c}(d, r).\, r.\, \overline{s}.\, \mathbf{0}$, then we have $P \cong Q$. We refer to Appendix C.1 for an ML-like presentation of the example above, as well as for a longer example, also showing how re-entrant calls may sometimes make visibility boundaries hard to predict.

5.2 Labelled semantics

To represent the point of view of the observer, we use typing environments of the form $[\Gamma \mid \kappa]$, where Γ is a visibility function and κ is a *stack*. Figure 4 defines the typed transitions, written $[\Gamma \mid \kappa]P \xrightarrow{\mu} [\Gamma' \mid \kappa']P'$. Rules `TrTau`, `TrO-fo` and `TrI-fo` are similar to their counterpart in Figure 2, and are omitted.

A trace $\mu_1 \dots \mu_n$ is a *well-bracketed trace of* P_0 *under* $[\Gamma_0 \mid \kappa_0]$ if there are P_i, Γ_i, κ_i $(1 \leq i \leq n)$ with $[\Gamma_i \mid \kappa_i]P_i \stackrel{\mu_i}{\Longrightarrow} [\Gamma_{i+1} \mid \kappa_{i+1}]P_{i+1}$ for all $0 \leq i < n$.

▶ **Definition 30** (Typed trace equivalence). Two $\langle \Delta \mid \sigma \rangle$-processes P and Q are *trace equivalent at* $[\Gamma \mid \kappa]$, written $P \eqsim^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$, if $\langle \Delta \mid \sigma \rangle$ and $[\Gamma \mid \kappa]$ are compatible and P and Q have the same sets of well-bracketed traces under $[\Gamma \mid \kappa]$.

This typed version of trace equivalence enjoys properties, including substitutivity properties, similar to those of the trace equivalence in Section 4. We only report the characterisation result with respect to may testing.

▶ **Theorem 31** (Characterisation of may testing using trace equivalence). *Suppose* $\langle \Delta \mid \sigma \rangle \vdash P, Q$; *then* $P \eqsim^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$ *if and only if* $P \cong^{[\Gamma \mid \kappa]}_{\langle \Delta \mid \sigma \rangle} Q$.

The analogous result for typed bisimilarity is in Appendix C, together with results about the shape of type-allowed traces akin to those in Section 4.2.

$$\texttt{TrO-ser}\ \frac{P \xrightarrow{\overline{a}\langle v\rangle(b,p)} P' \quad a \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma \mid \kappa]P \xrightarrow{\overline{a}\langle v\rangle(b,p)} [\Gamma; * \mapsto \Gamma(a), b, p \mid p^{\texttt{O}}, \kappa]P'}$$

$$\texttt{TrO-con}\ \frac{P \xrightarrow{\overline{p}\langle v\rangle(b)} P' \quad p \in \texttt{dom}(\Gamma) \quad \Gamma \not\downarrow *}{[\Gamma \mid p^{\texttt{I}}, q^{\texttt{O}}, \kappa]P \xrightarrow{\overline{p}\langle v\rangle(b)} [\Gamma^{-p}; * \mapsto \Gamma(p), b \mid q^{\texttt{O}}, \kappa]P'}$$

$$\texttt{TrI-ser}\ \frac{P \xrightarrow{a\langle v\rangle(b,p)} P' \quad a \in \Gamma(*)}{[\Gamma \mid \kappa]P \xrightarrow{a\langle v\rangle(b,p)} [\Gamma^{-*}; p \mapsto \Gamma(*), b; b \mapsto \Gamma(*), b \mid p^{\texttt{I}}, \kappa]P'}$$

$$\texttt{TrI-con}\ \frac{P \xrightarrow{p\langle v\rangle(b)} P' \quad p \in \Gamma(*)}{[\Gamma \mid p^{\texttt{O}}, \kappa]P \xrightarrow{p\langle v\rangle(b)} [\Gamma^{-*}; b \mapsto \Gamma(*), b \mid \kappa]P'}$$

Figure 4 Visible LTS with stack.

6 Conclusions and Future Work

In this paper we have studied the meaning of visibility, originating from game-semantics interpretations of typed λ-calculi and reflecting absence of higher-order store, in a calculus of pure name-passing such as the π-calculus. In contrast with λ-calculi, in the π-calculus there is no explicit notion of store, and no unique identities (e.g., the input end of a name may have multiple occurrences, arbitrarily structured). These aspects determine considerable differences in the semantic analysis of visibility. For instance, in the π-calculus visibility has to be enforced by means of a dedicated type system; and the visibility functions have to satisfy special properties such as transitivity. We have highlighted such differences, and investigated the behavioural effects of visibility. In particular, we have shown full abstract characterisations of may testing and barbed equivalence, as forms of trace equivalence and labelled bisimilarity.

There are several directions for future work that we would like to pursue. In game semantics, further aspects related to first-order store have been studied; for instance allowing first-order references as values (that can be passed around) [13, 9], or allowing to store first-order references (i.e., references with types such as `int ref ref` [17, 10]). The distinction between first-order and higher-order state has also been studied in [14]. We would like to investigate the effects of such distinctions on our types and proof techniques.

We have worked with πI, a dialect of π-calculus in which all names exchanged are newly created. Thus πI lacks the free-output construct, where the communication of existing names is permitted. As shown in the literature [5, 25, 8], πI is closer to game semantics than the standard π-calculus. We would like to see whether our results can be adapted so to accommodate also the free output construct. While free output can be modelled in πI [2], it allows a direct representation of idioms such as "delegate to someone else the task of answering', or the expression of tail-call recursion.

We have studied visibility on top of type systems for sequentiality and well-bracketing. Allowing concurrency would probably require relaxing the definition of the visibility functions; we leave the details to future work.

References

1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

2 Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998. doi:10.1016/S0304-3975(97)00220-X.

3 Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On sequentiality and well-bracketing in the π-calculus. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, June 2021. doi:10.1109/lics52264.2021.9470559.

4 Daniel Hirschkoff, Iwan Quémerais, and Davide Sangiorgi. First-order store and visibility in name-passing calculi. *CoRR*, abs/2504.17350, 2025. Extended online version of this paper. doi:10.48550/arXiv.2504.17350.

5 Kohei Honda. Processes and games. In Fabio Gadducci and Ugo Montanari, editors, *Fourth International Workshop on Rewriting logic and Its Applications, WRLA2002, Pisa, Italy, 19-21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 40–69. Elsevier, 2002. doi:10.1016/S1571-0661(05)82528-9.

6 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.

7 Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings*, pages 348–374, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-72019-3_13.

8 Guilhem Jaber and Davide Sangiorgi. Games, Mobile Processes, and functions. In *CSL 2022 - 30th EACSL Annual Conference on Computer Science Logic*, pages 1–35, Göttingen, Germany, February 2022. URL: https://hal.science/hal-03407123.

9 Guilhem Jaber and Nikos Tzevelekos. Trace semantics for polymorphic references. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 585–594. ACM, 2016. doi:10.1145/2933575.2934509.

10 Ohad Kammar, Paul B. Levy, Sean K. Moss, and Sam Staton. A monad for full ground reference cells. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, Reykjavik, Iceland, 2017. IEEE. doi:10.1109/LICS.2017.8005109.

11 Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. doi:10.1145/330249.330251.

12 Naoki Kobayashi and Davide Sangiorgi. A hybrid type system for lock-freedom of mobile processes. *ACM Trans. Program. Lang. Syst.*, 32(5):16:1–16:49, 2010. doi:10.1145/1745312.1745313.

13 James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007. doi:10.1007/978-3-540-73420-8_58.

14 Paul-André Melliès and Nicolas Tabareau. An algebraic account of references in game semantics. In Samson Abramsky, Michael W. Mislove, and Catuscia Palamidessi, editors, *Proceedings of*

the 25th Conference on Mathematical Foundations of Programming Semantics, MFPS 2009, Oxford, UK, April 3-7, 2009, volume 249 of *Electronic Notes in Theoretical Computer Science*, pages 377–405. Elsevier, 2009. doi:10.1016/J.ENTCS.2009.07.099.

15 R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

16 Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992. doi:10.1017/S0960129500001407.

17 Andrzej S. Murawski and Nikos Tzevelekos. Algorithmic games for full ground references. *Formal Methods in System Design*, 52(3):277–314, 2018. doi:10.1007/s10703-017-0292-9.

18 Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 72:1–72:10. ACM, 2014. doi:10.1145/2603088.2603116.

19 Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.*, 6(5):409–453, 1996. doi:10.1017/S096012950007002X.

20 Enguerrand Prebet. *Typed Behavioural Equivalences in the Pi-Calculus*. Theses, Ecole normale supérieure de lyon - ENS LYON ; Università degli studi (Bologne, Italie), September 2022. URL: https://hal.science/tel-03920089.

21 Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Inf. Comput.*, 111(1):120–153, 1994. doi:10.1006/INCO.1994.1042.

22 Davide Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996. doi:10.1016/0304-3975(96)00075-8.

23 Davide Sangiorgi. The name discipline of uniform receptiveness. *Theor. Comput. Sci.*, 221(1-2):457–493, 1999. doi:10.1016/S0304-3975(99)00040-7.

24 Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong normalisation in the pi -calculus. *Inf. Comput.*, 191(2):145–202, 2004. doi:10.1016/J.IC.2003.08.004.

25 Nobuko Yoshida, Simon Castellan, and Léo Stefanesco. Game semantics: Easy as pi. *CoRR*, abs/2011.05248, 2020. arXiv:2011.05248.

A Section 2: Definition of the Operational Semantics

Structural congruence, written $\equiv$, is the smallest relation that is an equivalence relation, contains α-conversion, is preserved by all operators of the calculus, and satisfies the following axioms:

$$\frac{}{P \mid Q \;\equiv\; Q \mid P} \qquad \frac{}{P \mid (Q \mid R) \;\equiv\; (P \mid Q) \mid R} \qquad \frac{}{P \mid 0 \equiv P} \qquad \frac{}{P + Q \;\equiv\; Q + P}$$

$$\frac{}{P + (Q + R) \;\equiv\; (P + Q) + R} \qquad \frac{a \notin \mathtt{fn}(Q)}{(\boldsymbol{\nu} aP) \mid Q \;\equiv\; \boldsymbol{\nu} a(P \mid Q)} \qquad \frac{}{\boldsymbol{\nu} a \boldsymbol{\nu} b P \equiv \boldsymbol{\nu} b \boldsymbol{\nu} a P}$$

$$\frac{}{!\alpha.\, P \equiv !\alpha.\, P \mid \alpha.\, P}$$

Similar versions of axioms for restrictions of the form $\boldsymbol{\nu} h$ instead of $\boldsymbol{\nu} a$ have been omitted.

To define the labelled transition system (LTS), we introduce *actions*, defined by the following grammar:

$$\mu \quad ::= \quad \tau \;\mid\; a\langle v\rangle(b) \;\mid\; \overline{a}\langle v\rangle(b) \;\mid\; h\langle v\rangle \;\mid\; \overline{h}\langle v\rangle \;\mid\; \omega$$

Actions are *early* for first-order values, and *late* for the transmission of `HO` names. We write $\mathtt{bn}(\mu)$ (resp. $\mathtt{fn}(\mu)$) for the bound (resp. free) names of action μ, defined by saying that b is a bound name in actions $a\langle v\rangle(b)$ and $\overline{a}\langle v\rangle(b)$, and the other names in actions are free.

$$\frac{}{a(x,b).\,P \xrightarrow{a\langle v\rangle(b)} P[v/x]} \qquad \frac{e \Downarrow v}{\overline{a}\langle e\rangle(b).\,P \xrightarrow{\overline{a}\langle v\rangle(b)} P} \qquad \frac{}{h(x).\,P \xrightarrow{h\langle v\rangle} P[v/x]}$$

$$\frac{e \Downarrow v}{\overline{h}\langle e\rangle.\,P \xrightarrow{\overline{h}\langle v\rangle} P} \qquad \frac{}{\omega.\,P \xrightarrow{\omega} \mathbf{0}} \qquad \frac{\alpha.\,P \xrightarrow{\mu} Q}{!\alpha.\,P \xrightarrow{\mu} Q \mid !\alpha.\,P} \qquad \frac{P \xrightarrow{\mu} Q}{\boldsymbol{\nu} a\ P \xrightarrow{\mu} \boldsymbol{\nu} a\ Q}\, a \notin \mathtt{fn}(\mu)$$

$$\frac{P \xrightarrow{\mu} Q}{\boldsymbol{\nu} h\ P \xrightarrow{\mu} \boldsymbol{\nu} h\ Q}\, h \notin \mathtt{fn}(\mu) \qquad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{a\langle v\rangle(b)} P' \quad Q \xrightarrow{\overline{a}\langle v\rangle(b)} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu} b(P' \mid Q')}$$

$$\frac{P \xrightarrow{h\langle v\rangle} P' \quad Q \xrightarrow{\overline{h}\langle v\rangle} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\, \mathtt{fn}(Q) \cap \mathtt{bn}(\mu) = \emptyset$$

$$\frac{e \Downarrow v \quad f \Downarrow v \quad P \xrightarrow{\mu} P'}{\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\mu} P'} \qquad \frac{e \Downarrow v \quad f \Downarrow v' \quad v \neq v' \quad Q \xrightarrow{\mu} Q'}{\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q \xrightarrow{\mu} Q'}$$

Figure 5 Labelled transition semantics.
Symmetric versions of the rules for parallel composition and for sum have been omitted.

We suppose that any closed first-order expression e can be evaluated to some value v. We write $e \Downarrow v$ for the corresponding evaluation relation. We furthermore suppose that equality between values can be tested.

The LTS is defined, on closed processes, by the rules in Figure 5. Based on $\xrightarrow{\mu}$, we define the following relations: $\longrightarrow$ stands for $\xrightarrow{\tau}$, and $\Longrightarrow$ stands for the reflexive transitive closure of $\longrightarrow$. We write $\overset{\mu}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$, and $\overset{\hat{\mu}}{\Longrightarrow}$ is equal to $\overset{\mu}{\Longrightarrow}$ when $\mu \neq \tau$, while $\overset{\hat{\tau}}{\Longrightarrow} = \Longrightarrow$.

We write $P \downarrow_\omega$ if $P \xrightarrow{\omega}$, i.e., if P contains an unguarded occurrence of an ω prefix at toplevel. We write $P \not\downarrow_\omega$ when this is not the case.

B Additional Material for Section 3

B.1 Properties of the type system

▶ **Lemma 32.** *Suppose $\Delta \vdash P$ and $a \notin \mathtt{fn}(P)$. Then also $\Delta^{-a} \vdash P$.*

▶ **Lemma 33** (Invariance of typing w.r.t. structural congruence)**.** *If $\langle \Delta \mid \sigma\rangle \vdash P$ and $P \equiv P'$, then also $\langle \Delta \mid \sigma\rangle \vdash P'$.*

B.2 Properties of trace equivalence

▶ **Lemma 34** (Shrinking)**.** *Suppose $\Gamma' \subseteq \Gamma$, Δ and Γ' compatible and $P \mathrel{\dot{\simeq}}^{\Gamma}_{\Delta} Q$. Then also $P \mathrel{\dot{\simeq}}^{\Gamma'}_{\Delta} Q$.*

▶ **Definition 35** (Projection)**.** Let Γ be a visibility function and N a set of names. We define a visibility function as follows:

$$\mathtt{proj}(N,\Gamma) : \begin{array}{rcl} \mathtt{HO} \cup \{*\} & \to & \mathcal{P}(\mathtt{HO}) \\ a \in \mathrm{dom}(\Gamma) \cap N & \mapsto & \Gamma(a) \cap N \end{array}$$

▶ **Lemma 36** (Extension). *Consider* $\Delta, \Gamma, \Gamma', P, Q$ *such that* $\mathtt{proj}(\mathtt{fn}(\Gamma, P, Q), \Gamma') = \Gamma$ *and* Δ *and* Γ' *are compatible. If* $P \eqsim^{\Gamma}_{\Delta} Q$ *then* $P \eqsim^{\Gamma'}_{\Delta} Q$.

In the statement above, $\mathtt{fn}(\Gamma, P, Q)$ stands for the union of all names mentioned in Γ, together with the names occurring free in P and Q.

B.3 Pruning Processes using Types

Consider a visibility function Δ and $S \subseteq \mathtt{HO}$. We write $\alpha \frown S$ when α is an input of the form $a(x, b)$ and either $a \in S$ or ($a \in \mathtt{dom}(\Delta)$ and $\Delta(a) \cap S \neq \emptyset$).

We define inductively the pruning function on processes that removes all instances of the names in S as follows:

$$\mathtt{Pr}^{\Delta}_{S}(\alpha.\, P) = \begin{cases} \mathbf{0} \text{ if } \alpha \frown S \\ \alpha.\, \mathtt{Pr}^{\Delta}_{S}(P) \text{ otherwise} \end{cases} \qquad \mathtt{Pr}^{\Delta}_{S}(!\alpha.\, P) = \begin{cases} \mathbf{0} \text{ if } \alpha \frown S \\ !\alpha.\, \mathtt{Pr}^{\Delta}_{S}(P) \text{ otherwise} \end{cases}$$

$$\mathtt{Pr}^{\Delta}_{S}(\overline{\alpha}.\, P) = \overline{\alpha}.\, \mathtt{Pr}^{\Delta}_{S}(P) \qquad \mathtt{Pr}^{\Delta}_{S}(\boldsymbol{\nu} a\ P) = \boldsymbol{\nu} a\ \mathtt{Pr}^{\Delta}_{S}(P) \qquad \mathtt{Pr}^{\Delta}_{S}(\boldsymbol{\nu} h\ P) = \boldsymbol{\nu} h\ \mathtt{Pr}^{\Delta}_{S}(P)$$

$$\mathtt{Pr}^{\Delta}_{S}(P \mid Q) = \mathtt{Pr}^{\Delta}_{S}(P) \mid \mathtt{Pr}^{\Delta}_{S}(Q) \qquad \mathtt{Pr}^{\Delta}_{S}(P + Q) = \mathtt{Pr}^{\Delta}_{S}(P) + \mathtt{Pr}^{\Delta}_{S}(Q) \qquad \mathtt{Pr}^{\Delta}_{S}(\mathbf{0}) = \mathbf{0}$$

$$\mathtt{Pr}^{\Delta}_{S}(\omega) = \omega \qquad \mathtt{Pr}^{\Delta}_{S}(\mathtt{if}\ e = f\ \mathtt{then}\ P\ \mathtt{else}\ Q) = \mathtt{if}\ e = f\ \mathtt{then}\ \mathtt{Pr}^{\Delta}_{S}(P)\ \mathtt{else}\ \mathtt{Pr}^{\Delta}_{S}(Q)$$

▶ **Lemma 37** (Pruning). *Let* Γ, Δ, P *such* $\Delta \vdash P$ *and* Γ *and* Δ *are compatible and* $\Delta \cup \Gamma$ *is transitive. Then for all* $S \subseteq \mathtt{HO}$ *such that* $S \cap (\Gamma \cup \Delta(*)) = \emptyset$, $\Delta^{-S} \vdash \mathtt{Pr}^{\Delta}_{S}(P)$ *and* $P \eqsim^{\Gamma}_{\Delta} \mathtt{Pr}^{\Delta}_{S}(P)$.

We actually remove more than the names in S. Indeed, to preserve sequentiality, we cannot simply remove an output; we actually remove all inputs that may cause an output on the names in S.

The idea of the proof is to show that $S \cap (\Gamma \cup \Delta(*)) = \emptyset$ is an invariant throughout the trace; for this we use repeatedly transitivity of $\Gamma \cup \Delta$.

Proof. Let $\mu_1 \dots \mu_n$ a trace of P under Γ, let P_i, Γ_i be the processes and visibility functions given that trace and Δ_i the visibility functions given by subject reduction. We show by induction on i that there exists Γ'_i, Δ'_i such that $\Gamma'_i \supseteq \Gamma_i$ and $\Delta'_i \supseteq \Delta_i$ and $S \cap (\Gamma'_i \cup \Delta'_i)(*) = \emptyset$ and that Δ'_i and Γ'_i are compatible, have a transitive union and coincide on the intersection of their domains. The case $i = 0$ is immediate, let $0 \leq i < n$, we assume the induction hypothesis on i and show it for $i + 1$ by cases on μ_i :

- if $\mu_i = \tau$, or $\mu_i = h\langle v \rangle$ or $\mu_i = \overline{h}\langle v \rangle$ then $\Gamma_{i+1} = \Gamma_i$ and $\Delta_{i+1} = \Delta_i$, these cases are immediate by setting $\Gamma'_{i+1} = \Gamma'_i$ and $\Delta'_{i+1} = \Delta'_i$.
- if $\mu_i = \overline{a}\langle v \rangle(b)$ then $\Gamma_{i+1} = \Gamma_i; * \mapsto \Gamma_i(a), b$ and $\Delta_{i+1} = \Delta_i^{-*}; b \mapsto \Delta_i(*), b$. Let $\Gamma'_{i+1} = \Gamma'_i; * \mapsto \Delta'_i(*), b \supseteq \Gamma_{i+1}$ because $a \in \Delta_i(*)$ therefore $\Gamma_i(a) \subseteq \Delta_i(*)$. And let $\Delta'_{i+1} = \Delta_i'^{-*}; b \mapsto \Delta'_i(*), b$.
  Γ'_{i+1} and Δ'_{i+1} are compatible, their union is transitive and they coincide on the intersection of their domains.
  $(\Gamma'_{i+1} \cup \Delta'_{i+1})(*) = (\Gamma'_i \cup \Delta'_i)(*), b$ where b is fresh therefore $b \notin S$ and by induction hypothesis $S \cap (\Gamma'_{i+1} \cup \Delta'_{i+1})(*) = \emptyset$.
- if $\mu_i = a\langle v \rangle(b)$ the case is similar to the previous one, swapping Γ and Δ.

This ensures us that none of the μ_i are inputs or outputs on higher-order names in S. Moreover if $a \in \mathtt{dom}(\Delta)$ and $\Delta(a) \cap S \neq \emptyset$ then for all i, $a \notin (\Gamma_i \cup \Delta_i)(*)$ otherwise we would have $\emptyset \neq \Delta(a) \cap S = \Delta_i(a) \cap S \subseteq (\Gamma_i \cup \Delta_i)(*) \cap S = \emptyset$.
Therefore the trace $\mu_1 \dots \mu_n$ is also a trace of $\mathtt{Pr}^{\Delta}_{S}(P)$ under Γ. ◀

C Additional Material for Section 5

C.1 About Example 29

The process discussed in Example 29 can be seen as the π-calculus counterpart of the following ML program, which might be more readable for a reader familiar with functional languages:

```
let h = ref 0 in
let b () = h := !h +1 in
begin
   a b;
   let x = !h in
   c x;
   let y = !h in
   if x=y then loop() else ()
end
```

Re-entrant call

The following example first shows an equality similar to that of Example 29; and then develops the example so to show how re-entrant calls may sometimes break "expected" visibility boundaries.

We define two processes P_1 and P_2, which access a reference h (implemented using a first-order name), and can call a server b. When calling b, the processes first pass the name `incr` of a server that increments the value stored in h. Once the call on b is answered the processes read the value in h and make another call to b passing the identity function. After that, the processes read again the value of h. Finally, P_1 outputs the first value read in h while P_2 outputs the second one.

$$\begin{aligned} P_i \stackrel{\text{def}}{=} \overline{b}(\texttt{incr}, q_1).q_1.\, h(x_1).\, (\overline{h}\langle x_1\rangle \\ \mid\ \overline{b}(\texttt{id}, q_2).\\ q_2.\, h(x_2).\, (\overline{h}\langle x_2\rangle \\ \mid\ \overline{p}\langle x_i\rangle)) \end{aligned}$$

We first compare the processes

$$\boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ a(b,p).\, P_1 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}) \quad \text{and} \quad \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ a(b,p).\, P_2 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}),$$

where $Q_{\texttt{incr}}$ (resp. $Q_{\texttt{id}}$) is the encoding of function `incr` (resp. `id`).

Here, the only way to change the value stored in h is by calling `incr`, which is not visible by the observer after the input on q_1. These two processes are bisimilar under any visibility.

Then we compare the processes

$$\begin{aligned} Q_1 &\stackrel{\text{def}}{=} \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ !a(b,p).\, P_1 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}})) \\ \text{and} \quad Q_2 &\stackrel{\text{def}}{=} \boldsymbol{\nu} h(\overline{h}\langle 0\rangle \mid\ !a(b,p).\, P_2 \mid\ Q_{\texttt{incr}} \mid\ Q_{\texttt{id}}). \end{aligned}$$

We are be able to distinguish them using a re-entrant call :

$$\begin{aligned} R \stackrel{\text{def}}{=} \overline{a}(b,p).\, (b(\texttt{incr}, q).\, \overline{a}(d,r).\, (!d(-,s).\, \overline{\texttt{incr}}(t).\, t.\, \overline{s} \\ \mid\ r(n).\, \texttt{if}\ n = 1\ \texttt{then}\ \overline{q}\ \texttt{else}\ \omega) \\ \mid\ p(m)) \end{aligned}$$

The idea is to call a a second time when `incr` is still in the observer's visibility; this means that the new server passed to a has access to `incr` during both calls. With the observer R, only Q_2 will be able to send a success signal. Therefore, for any visibility function Γ such that $a \in \Gamma(*)$, Q_1 and Q_2 are not bisimilar under visibility Γ.

The processes Q_1 and Q_2 can be seen as the π-calculus counterparts of the following ML program. (This example, with re-entrant calls, has been inspired by similar examples in game semantics of functional languages):

```
let h = ref 0 in
let incr () = h := !h +1 in
let id x = x in
let a b =
   b incr;
   let x1 = !h in
   b id;
   let x2 = !h in
   xi
```

These programs can be distinguished by the following ML program that corresponds to the process R :

```
let b f =
   let d _ = f () in
   let n = a d in
   if n = 1 then () else ω
in
a b
```

The idea is the same as before: `d` calls `f` which will be `incr` in the first call to `b`; therefore at each call to `d` the reference is incremented. As a consequence, first program returns 1 while the second one returns 2.

C.2 Properties of traces: views and well-bracketing

Along the lines of the material presented in Section 4.2, we can reason about the traces produced by typable processes, based on the notion of justification. For this, we adapt Definition 22 to define views for traces. The notion of *respecting views* (Definition 23) is also adapted straightforwardly, which allows us to obtain the analogue of Lemma 24.

▶ **Lemma 38.** *Consider $\Delta, \Gamma, \sigma, \kappa, P$ such that $\langle \Delta \mid \sigma \rangle \vdash P$ and $[\Gamma \mid \kappa]$ is compatible with $\langle \Delta \mid \sigma \rangle$. Let $A = (\Gamma \cup \Delta)(*)$, then every trace of P under $[\Gamma \mid \kappa]$ respects views with starting visibility A.*

As highlighted in [3], continuation names allow us to formulate a property of well-bracketing for traces, enforced by our type system. In contrast with [3], however, we we do not impose input receptiveness on continuation names.

▶ **Definition 39** (Well-bracketing)**.** An action of the form $a\langle v \rangle(b, p)$ or $\overline{a}\langle v \rangle(b, p)$ is called a *question*. An action of the form $p\langle v \rangle(b)$ or $\overline{p}\langle v \rangle(b)$ is called an *answer*.

A trace $\mu_1 \dots \mu_n$ is *well-bracketed* if for all $i < j$, if μ_i is a question and μ_j is an answer with $\mu_i \not\frown \mu_k$ and $\mu_k \not\frown \mu_j$ for all $i < k < j$, then $\mu_i \curvearrowright \mu_j$.

A stack σ is *clean* if each name has at most one occurrence, with any tag, in σ. A trace $\mu_0 \dots \mu_n$ is a *typed trace emanating from* P_0 if there are $\Delta_1, \sigma_1, \dots, \Delta_n, \sigma_n, P_1, \dots, P_n$ such that for all $0 \leq j < n$ we are in one of the cases given by Theorem 27, which gives $\langle \Delta_{j+1} \mid \sigma_{j+1} \rangle$ when $P_j \xrightarrow{\mu_{j+1}} P_{j+1}$.

▶ **Lemma 40.** *Suppose $\langle \Delta_0 \mid \sigma_0 \rangle \vdash P_0$, where σ_0 is clean. All typed traces emanating from P_0 are well-bracketed.*