Towards an algebraic theory of typed mobile processes*

Yuxin Deng[1] and Davide Sangiorgi[2]

[1] INRIA and Université Paris 7, France
[2] Università di Bologna, Italy

Abstract. The impact of types on the algebraic theory of the π-calculus is studied. The type system has capability types. They allow one to distinguish between the ability to read from a channel, to write to a channel, and both to read and to write. They also give rise to a natural and powerful subtyping relation.
Two variants of typed bisimilarity are considered, both in their late and in their early version. For both of them, proof systems that are sound and complete on the closed finite terms are given. For one of the two variants, a complete axiomatisation for the open finite terms is also presented.

1 Introduction

The π-calculus is the paradigmatic calculus for mobile processes. Its theory has been studied in depth [8, 12]. Relevant parts of it are the algebraic theory and the type systems. Most of the algebraic theory has been developed on the untyped calculus; the results include proof systems or axiomatisations that are sound and complete on finite processes for the main behavioral equivalences: late and early bisimilarity, late and early congruence [9, 5, 6], open bisimilarity [11], testing equivalence [1]. Much of the research on types has focused on their behavioral effects. For instance, modifications of the standard behavioral equivalences have been proposed so as to take types into account [10, 12].

In this paper, we study the impact of types on the algebraic theory of the π-calculus. Precisely, we study axiomatisations of the typed π-calculus. Although algebraic laws for typed calculi for mobility have been considered in the literature [12], we are not aware of any axiomatisation or proof system.

The type system that we consider has *capability types* (sometimes called I/O types) [10, 4]. These types allow us to distinguish, for instance, the capability of using a channel in input from that of using the channel in output. A capability type shows the capability of a channel and, recursively, of the channels carried by that channel. For instance, a type $a : \mathtt{iob}T$ (for an appropriate type expression T) says that channel a can be used only in input; moreover, any channel received at a may only be used in output — to send channels which can be used both in input and in output. Thus, process $a(x).\bar{x}b.b(y).\bar{b}y.\mathbf{0}$ (sometimes the trailing $\mathbf{0}$ is omitted) is well-typed under the type assignment $a : \mathtt{iob}T, b : \mathtt{b}T$. We recall

* Work supported by EU project PROFUNDIS

that $\bar{a}b.P$ is the output at a of channel b with continuation P, and that $a(x).P$ is an input at a with x a placeholder for channels received in the input whose continuation is P.

On calculi for mobility, capability types have emerged as one of the most useful forms of types, and one whose behavioral effects are most prominent. Capabilities are useful for protecting resources; for instance, in a client-server model, they can be used for preventing clients from using the access channel to the server in input and stealing messages to the server; similarly they can be used in distributed programming for expressing security constraints [4]. Capabilities give rise to *subtyping*: the output capability is contravariant, whereas the input capability is covariant. As an example, we show a subtyping relation in Fig. 1, where an arrow indicates the subtyping relation between two related types. The depth of nesting of capabilities is 1 for all types in diagram (a), and 2 for all types in diagram (b). (The formal definitions of types and subtyping relation will be given in Section 2.) Subtyping is useful when the π-calculus is used for object-

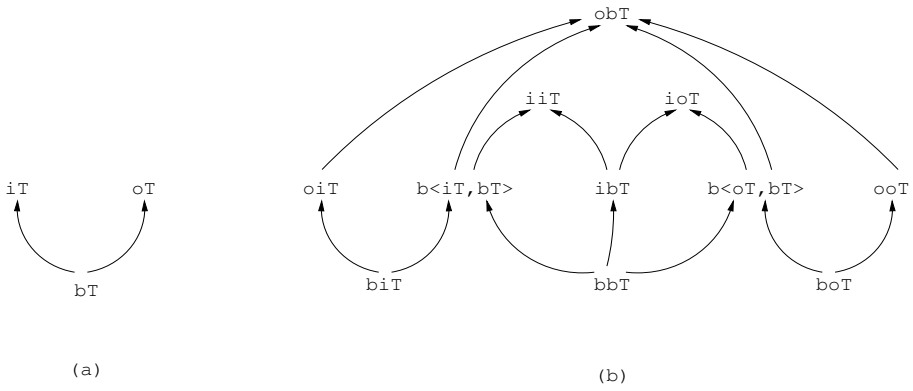

Fig. 1. Subtyping relation, with $T = \texttt{unit}$

oriented programming, or for giving semantics to object-oriented languages.

To see why the addition of capability types has semantic consequences, consider

$$P \stackrel{\text{def}}{=} \nu c\, \bar{b}c.a(y).(\overline{y} \mid c) \qquad Q \stackrel{\text{def}}{=} \nu c\, \bar{b}c.a(y).(\overline{y}.c + c.\overline{y})$$

These processes are not behaviorally equivalent in the untyped π-calculus. For instance, if the channel received at a is c, then P can terminate after 2 interactions with the external observer. By contrast, Q always terminates after 4 interactions with the observer. However, if we require that only the input capability of channels may be communicated at b, then P and Q are indistinguishable in any (well-typed) context. For instance, since the observer only receives the input capability on c, it cannot resend c along a: channels sent at a require at least the output capability (cf: the occurrence of $\overline{y}$). Therefore, in the typed

setting, processes are compared w.r.t. an observer with certain capabilities (i.e., types on channels). Denoting with Δ these capabilities, then typed bisimilarity between P and Q is written $P \sim_\Delta Q$.

In the untyped π-calculus, labelled transition systems are defined on processes; the transition $P \xrightarrow{\alpha} P'$ means that P can perform action α and then become P'. In the typed π-calculus, the information about the observer capabilities is relevant because the observer can only test processes on interactions for which the observer has all needed capabilities. Hence typed labelled transition systems are defined on configurations, and a configuration $\Delta \sharp P$ is composed of a process P and the observer capabilities Δ (we sometimes call Δ the external environment). A transition $\Delta \sharp P \xrightarrow{\alpha} \Delta' \sharp P'$ now means that P can evolve into P' after performing an action α allowed by the environment Δ, which in turn evolves into Δ'.

Capability types have been introduced in [10]. A number of variants and extensions have then been proposed. We follow Hennessy and Riely's system [4], in which, in contrast with the system in [10]: (i) there are partial meet and join operations on types; (ii) the typing rule for the *matching* construct (the construct used for testing equality between channels) is very liberal, in that it can be applied to channels of arbitrary types (in [10] only channels that possess both the input and the output capability can be compared). While (i) only simplifies certain technical details, (ii) seems essential. Indeed, the importance of matching for the algebraic theory of the π-calculus is well-known (it is the main reason for the existence of matching in the untyped calculus).

Typed bisimilarity and the use of configurations for defining typed bisimilarity have been introduced in [2]. We follow a variant of them put forward by Hennessy and Rathke [3], because it uses the type system of [4].

The main results in this paper are an axiomatisation and a proof system for typed bisimilarity ($\sim$). The axiomatisation is for all finite processes. The proof system has a simple correctness proof but only works on the closed terms. The bisimilarity $\sim$ is a variant of that in [3]. For the typed bisimilarity in [3] we provide a proof system for the closed terms, and an indirect axiomatisation of all terms that exploits the system of $\sim$. We have not been able to give a direct axiomatisation: the main difficulties are discussed in Section 5.1. All results are given for both the late and the early versions of the bisimilarities.

The axiomatisation and the proof systems are obtained by modifying some of the rules of the systems for the untyped π-calculus, and by adding a few new laws. The proofs of soundness and completeness, although follow the general schema of the proofs of the untyped calculus, have quite different details. An example of this is the treatment of fresh channels in input actions and the closure under injective substitutions, that we comment on below.

In the untyped π-calculus, the following holds:

$$\textit{If } P \sim Q \textit{ and } \sigma \textit{ is injective on } fn(P,Q), \textit{ then } P\sigma \sim Q\sigma.$$

Hence it is sufficient to consider all free channels in P, Q and *one* fresh channel when comparing the input actions of P and Q in the bisimulation game. This

result is crucial in the algebraic theory of untyped calculi. For instance, in the proof system for (late) bisimilarity the inference rule for input is:

If $P\{b/x\} = Q\{b/x\}$ for all $b \in fn(P, Q, c)$, where c is a fresh channel, then $a(x).P = a(x).Q$.

For typed bisimilarity the situation is different. Take the processes

$$P \stackrel{\text{def}}{=} a(x : \mathtt{ob}T).\bar{x}c.\bar{c} \qquad Q \stackrel{\text{def}}{=} a(x : \mathtt{ob}T).\bar{x}c$$

and compare them w.r.t. an observer Δ. Consider what happens when the variable x is replaced by a fresh channel b, whose type in Δ is S. By the constraint imposed by types, S must be a subtype of the type $\mathtt{ob}T$ for x (see Fig. 1 (b)). Now, different choices for S will give different results. For instance, if S is $\mathtt{ob}T$ itself, then the observer has no input capability on b, thus can not communicate with P and Q at b. That is, from the observer's point of view the output $\bar{b}c$ is not observable and the two derivative processes are equivalent. Similarly if S is $\mathtt{bo}T$ then the output $\bar{c}$ is not observable. However, if S is $\mathtt{bb}T$ then $\bar{b}c.\bar{c}$ is not equivalent to $\bar{b}c$, since all outputs become observable. This example illustrates the essential difficulties in formulating proof systems for typed bisimilarities:

1. Subtyping appears in substitutions and changes the original type of a variable into one of its subtypes.
2. The choice of this subtype is relevant for behavioral equivalence.
3. Different subtypes may be incompatible (have no common subtype) with one another (for instance, $\mathtt{bo}T$ and $\mathtt{bb}T$ in the example above; they are both subtypes of $\mathtt{ob}T$).

A consequence of (2) and (3), for instance, is that there is not a "best subtype", that is a single type with the property that equivalence under this type implies equivalence under any other types.

Another example of the consequences brought by types in the algebraic theory is the congruence rule for prefixes: we have to distinguish the cases in which the subject of the prefix is a channel from the case in which the subject is a variable. This is a rather subtle and technical difference, that is discussed in Section 3.

2 The typed π-calculus

In this section we review the π-calculus, capability types, and typed bisimilarity. We assume an infinite set of channels, ranged over by $a, b, \ldots$, and an infinite set of variables, ranged over by $x, y, \ldots$. We write $*$ for the unit value (we shall use `unit` as the only base type). Channels, variables and $*$ are the *names*, ranged over by $u, v, \ldots$. Below is the syntax of finite π-calculus processes.

$$P, Q ::= \mathbf{0} \mid \tau.P \mid u(x:T).P \mid \bar{u}v.P \mid P + Q \mid P \mid Q \mid (\nu a : T)P \mid \varphi PQ$$
$$\varphi ::= [u = v] \mid \neg\varphi \mid \varphi \vee \psi$$

Here φPQ is an if-then-else construct on the boolean condition φ. We omit the else branch Q when it is $\mathbf{0}$; also, $[u = v]$ is called a *match*, and $\neg[u = v]$, abbreviated as $[u \neq v]$, is called a *mismatch*. Binding names (in input and restriction) are annotated with their types. We write $fn(P)$ and $fv(P)$ for the set of free names and the set of free variables, respectively, in P. When φ has no variables, $[\![\varphi]\!]$ denotes the boolean value of φ. In the calculus, the distinction between channels and variables simplifies certain technical details; see for instance the discussion on the rules for substitutivity of prefixes in Section 3: the rules are different depending on whether the prefixes use channels or variables. (This is not the case in the untyped case: for instance, [9] does not distinguish between variables and channels, but it is quite straightforward to adapt the work to the case where there is such a distinction.)

We recall the capability types, as from [3, 4]. The subtyping relation $<:$ and the typing rules for processes are displayed in Table 1. We write $T :: \mathtt{TYPE}$ to mean that T is a legal type. There are three forms of types for channel names: $\mathtt{i}T, \mathtt{o}S$ and $\mathtt{b}\langle T, S\rangle$. They give names the ability to receive values of type T, send values of type S, or to do both. We often abbreviate $\mathtt{b}\langle T, T\rangle$ to $\mathtt{b}T$. We refer to [4] for the definition of the two partial operators meet ($\sqcap$) and join ($\sqcup$). Intuitively, the meet (resp. join) of two types is the union (resp. intersection) of their capabilities.

Types:

$$\frac{}{\mathtt{unit} :: \mathtt{TYPE}} \qquad \frac{T :: \mathtt{TYPE}}{\mathtt{i}T, \mathtt{o}T :: \mathtt{TYPE}} \qquad \frac{T, S :: \mathtt{TYPE} \quad S <: T}{\mathtt{b}\langle T, S\rangle :: \mathtt{TYPE}}$$

Subtyping:

$$\frac{}{T <: T} \qquad \frac{T <: T'}{\mathtt{i}T <: \mathtt{i}T'} \qquad \frac{T <: T'}{\mathtt{o}T' <: \mathtt{o}T}$$

$$\frac{T <: T'}{\mathtt{b}\langle T, S\rangle <: \mathtt{i}T'} \qquad \frac{T <: T'}{\mathtt{b}\langle S, T'\rangle <: \mathtt{o}T} \qquad \frac{T <: T' \quad S <: S'}{\mathtt{b}\langle T, S'\rangle <: \mathtt{b}\langle T', S\rangle}$$

Typing rules:

$$\frac{\Gamma(u) <: T}{\Gamma \vdash u : T} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P + Q} \qquad \frac{\Gamma, x : T \vdash P \quad \Gamma \vdash u : \mathtt{i}T}{\Gamma \vdash u(x : T).P}$$

$$\frac{}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma, a : T \vdash P}{\Gamma \vdash (\nu a : T)P} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash v : T \quad \Gamma \vdash u : \mathtt{o}T}{\Gamma \vdash \bar{u}v.P}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \tau.P} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q \quad n(\varphi) \subseteq dom(\Gamma)}{\Gamma \vdash \varphi\ P\ Q}$$

Table 1. Types and typing rules

We use Δ and Γ for type environments. A type environment Δ is a partial function from channels and variables to types; we write $\Delta_{\mathtt{c}}$ and $\Delta_{\mathtt{v}}$ for the

channel and variable parts of Δ, respectively. A type environment is undefined on infinitely many channels and variables (to make sure it can always be extended). We often view, and talk about, $\Delta_{\mathtt{c}}$ as a set of assignments of the form $a : T$. Similarly for $\Delta_{\mathtt{v}}$. We write $dom(\Delta)$ for the channels and variables on which Δ is defined ($dom(\Delta)$ can be infinite). Using the partial meet operation, we can extend a type environment Δ to $\Delta \sqcap u : T$, which is just $\Delta, u : T$ if $u \notin dom(\Delta)$, otherwise it differs from Δ at name u because the capability of this name is extended to be $\Delta(u) \sqcap T$ (if $\Delta(u) \sqcap T$ is undefined, then so is $\Delta \sqcap u : T$). When $dom(\Delta) \cap dom(\Delta') = \emptyset$, we use Δ, Δ' to represent the union of Δ and Δ'. Subtyping is extended to type environments, but only considering the types of channels. So $\Gamma <: \Delta$ means that $\Gamma_{\mathtt{v}} = \Delta_{\mathtt{v}}$, $dom(\Delta_{\mathtt{c}}) \subseteq dom(\Gamma_{\mathtt{c}})$ and $\Gamma_{\mathtt{c}}(a) <: \Delta_{\mathtt{c}}(a)$ for all $a \in dom(\Delta_{\mathtt{c}})$. The intuition is that channels are capabilities while variables are obligations of the environment. The environment is obligated to fill in the variables at the specified types. Therefore we cannot allow weakening on variables as this would weaken the obligations. On the other hand we cannot use contravariant weakening because then the process would not type. That's why variables are invariant. If $\Delta(u)$ is defined and takes the form $\mathtt{i}T$ or $\mathtt{b}\langle T, S\rangle$, then the predicate $\Delta(u)\downarrow_{\mathtt{i}}$ holds and we write $\Delta(u)_{\mathtt{i}}$ for T, otherwise we write $\Delta(u)\not\downarrow_{\mathtt{i}}$, indicating that Δ has no input capability on u. Similarly for $\Delta(u)_{\mathtt{o}}$ and $\Delta(u)\downarrow_{\mathtt{o}}$ (output capability). The typing rules follow [10, 4, 3].

Definition 1. *A configuration is a pair $\Delta \sharp P$ which respects some type environment Γ, i.e., $\Gamma <: \Delta$ and $\Gamma \vdash P$.*

The transition system for configurations is in Table 2. Bound names, names and the subject of a prefix α, written $bn(\alpha)$, $n(\alpha)$ and $subj(\alpha)$ respectively, are defined in the usual way. We identify terms up to alpha conversion and assume $bn(P) \cap dom(\Delta) = \emptyset$ for any configuration $\Delta \sharp P$. In the premise of rule Red, $P \xrightarrow{\tau} P'$ stands for the normal reduction relation of the untyped π-calculus. In rule Out, the process sends channel b to the environment, so the latter should be dynamically extended with the capability on b thus received. For this, we use the meet operator, and exploit the following property on types:

$$R <: T \text{ and } R <: S \text{ imply } T \sqcap S \text{ defined and } R <: T \sqcap S$$

for any type T, S and R. (This property does not hold for the capability types as in [10].)

$P \sim_{\Delta} Q$ reads "P and Q are bisimilar under type environment Δ". The type environment Δ is used as follows: $\Delta_{\mathtt{c}}$ shows the channels that are known to the external observer testing the processes in the bisimulation game, and the types with which the observer is allowed to use such channels. By contrast, $\Delta_{\mathtt{v}}$ shows the set of variables that may appear free in the processes and the types for these variables show how the observer can instantiate such variables (in closing substitutions). Therefore: the channels of $\Delta_{\mathtt{c}}$ are to be used by the observer, with the types indicated in $\Delta_{\mathtt{c}}$; the variables in $\Delta_{\mathtt{v}}$ are to be used by the processes, but the observer can instantiate them following the types indicated in $\Delta_{\mathtt{v}}$.

$$\textsf{Red}\ \frac{P \xrightarrow{\tau} P'}{\Delta \sharp P \xrightarrow{\tau} \Delta \sharp P'} \qquad \textsf{Out}\ \frac{\Delta(a)\downarrow_{\mathtt{i}}}{\Delta \sharp \bar{a}b.P \xrightarrow{\bar{a}b} \Delta \sqcap b : \Delta(a)_{\mathtt{i}} \sharp P}$$

$$\textsf{In}\ \frac{\Delta(a)\downarrow_{\mathtt{o}}}{\Delta \sharp a(x:T).P \xrightarrow{a(x:T)} \Delta, x : T \sharp P} \qquad \textsf{Open}\ \frac{\Delta \sharp P \xrightarrow{\bar{a}b} \Delta' \sharp P' \quad a \neq b}{\Delta \sharp (\nu b : T)P \xrightarrow{\bar{a}(b:T)} \Delta' \sharp P'}$$

$$\textsf{Res}\ \frac{\Delta \sharp P \xrightarrow{\alpha} \Delta' \sharp P' \quad a \notin n(\alpha)}{\Delta \sharp (\nu a : T)P \xrightarrow{\alpha} \Delta' \sharp (\nu a : T)P'} \qquad \textsf{Sum}\ \frac{\Delta \sharp P \xrightarrow{\alpha} \Delta' \sharp P'}{\Delta \sharp P + Q \xrightarrow{\alpha} \Delta' \sharp P'}$$

$$\textsf{True}\ \frac{[\![\varphi]\!] = \mathit{True} \quad \Delta \sharp P \xrightarrow{\alpha} \Delta' \sharp P'}{\Delta \sharp \varphi PQ \xrightarrow{\alpha} \Delta' \sharp P'} \qquad \textsf{False}\ \frac{[\![\varphi]\!] = \mathit{False} \quad \Delta \sharp Q \xrightarrow{\alpha} \Delta' \sharp Q'}{\Delta \sharp \varphi PQ \xrightarrow{\alpha} \Delta' \sharp Q'}$$

$$\textsf{Par}\ \frac{\Delta \sharp P \xrightarrow{\alpha} \Delta' \sharp P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{\Delta \sharp P \mid Q \xrightarrow{\alpha} \Delta' \sharp P' \mid Q}$$

Table 2. Typed transition system

A process is *closed* if it does not have free variables; similarly a type environment is closed if it is only defined on channels. Otherwise, processes and type environments are *open*. We first define $\sim_\Delta$ on the closed terms, then on the open terms. We write $|\alpha|$ for the action α where its type annotations have been stripped off. Bisimilarity is given in the "late" style [12]; we consider the "early" style in Section 5.2.

Definition 2. *A family of symmetric binary relations over closed terms, indexed by type environments, and written $\{\mathcal{R}_\Delta\}_\Delta$, is a* typed bisimulation *whenever $P\ \mathcal{R}_\Delta\ Q$ implies that, for two configurations $\Delta\sharp P$ and $\Delta\sharp Q$,*

1. *if $\Delta\sharp P \xrightarrow{\alpha} \Delta'\sharp P'$ and α is not an input action, then for some Q', $\Delta\sharp Q \xrightarrow{\beta} \Delta'\sharp Q'$, $|\alpha| = |\beta|$ and $P'\ \mathcal{R}_{\Delta'}\ Q'$.*
2. *if $\Delta\sharp P \xrightarrow{a(x:T)} \Delta'\sharp P'$, then for some Q', $\Delta\sharp Q \xrightarrow{a(x:S)} \Delta''\sharp Q'$ and for all b with $\Delta_{\mathsf{c}} \vdash b : \Delta(a)_{\mathtt{o}}$ it holds that $P'\{b/x\}\ \mathcal{R}_\Delta\ Q'\{b/x\}$.*

Two processes P and Q are typed Δ-bisimilar, written $P \sim_\Delta Q$, if there exists a typed bisimulation $\{\mathcal{R}_\Delta\}_\Delta$ such that $P\ \mathcal{R}_\Delta\ Q$.

The difference w.r.t. typed bisimilarity as in [2, 3] is that, in the input clause, the type environment Δ is not extended. In other words, the knowledge of the external observer does not change through interactions with the process in which the value transmitted is supplied by the observer itself (by contrast, the knowledge does change when the value is supplied by the process; cf. rule Out in Table 2). We discuss the alternative equivalence (where the environment can be extended) in Section 5.1.

Definition 3. *Two processes P and Q are bisimilar under the environment Δ, written $P \sim_\Delta Q$, if $fv(P, Q) \subseteq dom(\Delta_{\mathtt{v}})$ and, for all $\widetilde{b}$ with $\Delta_{\mathsf{c}} \vdash \widetilde{b} : \widetilde{T}$, it holds that $P\{\widetilde{b}/\widetilde{x}\}\ \sim_\Delta\ Q\{\widetilde{b}/\widetilde{x}\}$.*

Since all processes are finite, and we do not use recursive types, in $P \sim_\Delta Q$, the environment Δ can always be taken to be *finite* (i.e., defined only on a finite number of channels and variables): it is sufficient that Δ has enough names fresh w.r.t. P and Q, for all relevant types. This can be proved with a construction similar to that in Lemma 2. In the remainder of the paper all type environments are assumed to be finite.

3 Axioms for typed bisimilarity

The axiom system $\mathcal{A}$ for typed bisimilarity is in Table 3. Whenever we write $P =_\Delta Q$ it is intended that both $\Delta \sharp P$ and $\Delta \sharp Q$ are configurations. The rules are divided into seven groups, namely those for: substitutivity, sums, looking up the type environment, conditions, restrictions, the expansion law and alpha-conversion. The rules that are new or different w.r.t. those of the untyped π-calculus are marked with an asterisk. As in the untyped case [9], other rules involving conditions, such as

$$\mathbf{C8} \quad [\varphi \vee \psi]P =_\Delta \varphi P + \psi P$$

are derivable. **Tin*** shows that an input prefix is not observable if the observer has no output capability on the subject of the input. **Tout*** is the symmetric rule, for output. **Twea*** gives us weakening for type environments. **Tvar*** shows that a variable can only be instantiated with channels that in the type environment have types compatible with that of the variable. **Tpre*** is used to replace names underneath a match. (In the untyped setting, the rule has no side condition. Here we need one to ensure well-typedness of the process resulting from the substitution, since the names in the match can have arbitrary — and possibly unrelated — types. Similarly for the conditions on types in the expansion law **E**.) In **Ires***, different types T_1, T_2 are used for the processes in the conclusion. We cannot replace **Ires*** with two simpler rules such as

$$\begin{array}{l} \text{If } P =_\Delta Q \text{ then } (\nu a : T)P =_\Delta (\nu a : T)Q \\ (\nu a : T_1)P =_\Delta (\nu a : T_2)P, \end{array}$$

for equalities like

$$(\nu b : \mathtt{bi}T)\bar{a}b.b(x : \mathtt{i}T).\mathbf{0} =_{a:\mathtt{iob}T} (\nu b : \mathtt{bo}T)\bar{a}b.b(x : \mathtt{o}T).\mathbf{0}$$

could not be derived (due to the constraints given by the well-typedness of processes). Similarly for rule **Iin***.

To illustrate rule **Ipar*** we need to introduce some notations. We define the depth, $d(T)$, of a type T, to be the maximum number of nesting of capabilities in it. Let $\Gamma \vdash P$. Each name in P has a type, either recorded in the syntax of P or in Γ. If $T_1, \ldots, T_n$ are all such types, $d(\Gamma, P)$ is $max\{d(T_i) \mid 1 \leq i \leq n\}$. Now, if $\Delta \sharp P_i$ is a configuration, for $i = 1, 2$, then there are type environments Γ_i such that $\Gamma_i \lessdot \Delta$ and $\Gamma_i \vdash P_i$. In this case, we set $d(P_1, P_2, \Gamma_1, \Gamma_2)$ as $max\{d(\Gamma_1, P_1), d(\Gamma_2, P_2)\}$. There are only finitely many different types with

depth less than or equal to $d(P_1, P_2, \Gamma_1, \Gamma_2)$, say $S_1, \ldots, S_m$, and $\Delta_{\mathtt{v}}$ is defined on finitely many variables, say $x_1, \ldots, x_k$. We can pick up n fresh (hitherto unused) channels $a_{i1}, \ldots, a_{in}$ for each S_i, with n the larger value between k and the size of $P_1 + P_2$, and construct a type environment

$$Env(\Delta, P_1, P_2, \Gamma_1, \Gamma_2) = \{a_{ij} : S_{ij} \mid 0 < i \leq m, 0 < j \leq n\}.$$

Rule **Ipar*** says that if Δ cannot distinguish P from Q, then it cannot distinguish $P \mid R$ from $Q \mid R$ either, provided that: (i) Δ contains enough fresh channels; (ii) R requires no capabilities beyond the knowledge of Δ. Note that we cannot do without the first condition, i.e., the rule cannot be simplified as:

$$\textit{For any } \Delta, \textit{ if } P =_\Delta Q \textit{ and } \Delta \vdash R \textit{ then } P \mid R =_\Delta Q \mid R$$

which is unsound for $\sim$. The point is that when comparing $P \mid R$ and $Q \mid R$, the observer may first increase his knowledge by interacting with R, then distinguish P from Q by the new knowledge. For example, let $\Delta \stackrel{\text{def}}{=} a : \mathtt{b}T, e : \mathtt{b}T, b : T$ and

$$P \stackrel{\text{def}}{=} a(x : T).[x \neq b]\tau \qquad Q \stackrel{\text{def}}{=} a(x : T).\mathbf{0} \qquad R \stackrel{\text{def}}{=} (\nu c : T)\bar{e}c.$$

It is easy to see that $P \sim_\Delta Q$ and $\Delta \vdash R$ but $P \mid R \not\sim_\Delta Q \mid R$. After the interaction with R, the environment evolves into $\Delta, c : T$. Later the new channel c may be used to instantiate x, thus validating the condition $x \neq b$ and liberating the prefix τ.

Iin* and **Iout*** are the rules for substitutivity for input and output prefixes. In **Iin***, well-definedness of the configurations $\Delta \sharp a(x : T_1).P$ and $\Delta \sharp a(x : T_2).Q$ implies the condition: $\Delta(a)_{\mathtt{o}} <: T_i$ for $i = 1, 2$. (Similarly for rules **Iv1**, **Iin** and **Iin′** below.) In **Iout***, the observer knowledge of the type of b may increase when the processes emit b themselves (for the type under which b is emitted is composed with the possible type of b in Δ). Both in **Iin*** and in **Iout***, the free names of the input and output prefixes are channels rather than variables. Below we discuss:

1. the unsoundness of the rules in which (some or all) the channels are replaced by variables;
2. other rules, that are valid for variables;
3. why these other rules are not needed in the axiom system.

Intuitively the reason for (1) is the different usages of channels and variables that appear in a type environment: the information on channels tells us how these channels are to be used by the *external environment*, while the information on variables tells us how these variables are to be instantiated inside the *tested processes*.

To see that **Iin*** is unsound when the subject of the prefix is a variable, take $\Delta_{\mathtt{c}} \stackrel{\text{def}}{=} a : \mathtt{bo}T, b : \mathtt{o}T$ and $\Delta \stackrel{\text{def}}{=} \Delta_{\mathtt{c}}, x : \mathtt{b}\langle \mathtt{o}T, \mathtt{b}T \rangle$. Then we have

$$[y = b]\tau \sim_{\Delta, y:\Delta(x)_{\mathtt{o}}} \mathbf{0}$$

Iin*	If $P =_{\Delta,x:\Delta(a)_\circ} Q$ then $a(x:T_1).P =_\Delta a(x:T_2).Q$
Iout*	If $P =_{\Delta\sqcap b:\Delta(a)_\imath} Q$ then $\bar{a}b.P =_\Delta \bar{a}b.Q$
Itau	If $P =_\Delta Q$ then $\tau.P =_\Delta \tau.Q$
Isum	If $P =_\Delta Q$ then $P + R =_\Delta Q + R$
Ires*	If $P =_\Delta Q$ then $(\nu a:T_1)P =_\Delta (\nu a:T_2)Q \quad a \notin dom(\Delta)$
Icon	If $P =_\Delta Q$ then $\varphi P =_\Delta \varphi Q$
Ipar*	Assume $\Delta_0 \sharp P$ respects Γ_1, $\Delta_0 \sharp Q$ respects Γ_2, and $\Delta = \Delta_0, Env(\Delta_0, P, Q, \Gamma_1, \Gamma_2)$. If $P =_\Delta Q$ and $\Delta \vdash R$ then $P \mid R =_\Delta Q \mid R$

S1	$P + \mathbf{0} =_\Delta P$
S2	$P + P =_\Delta P$
S3	$P + Q =_\Delta Q + P$
S4	$P + (Q + R) =_\Delta (P + Q) + R$

Tin*	If $\Delta(a)\not\downarrow_\circ$ then $a(x:T).P =_\Delta \mathbf{0}$
Tout*	If $\Delta(a)\not\downarrow_\imath$ then $\bar{a}u.P =_\Delta \mathbf{0}$
Twea*	If $P =_\Delta Q$ and $\Delta <: \Delta'$ then $P =_{\Delta'} Q$
Tvar*	$[x \neq a_1] \cdots [x \neq a_m]P =_\Delta \mathbf{0}$ if $\{b \in dom(\Delta_\mathsf{c}) \mid \Delta(b) <: \Delta(x)\} \subseteq \{a_1, \cdots, a_m\}$
Tpre*	$[x = a]\alpha.P =_\Delta [x = a](\alpha\{a/x\}).P \quad$ if $\Delta(a) <: \Delta(x)$

C1	$\varphi\ P =_\Delta \psi\ P \quad$ if $\varphi \Longleftrightarrow \psi$
C2	$[a = b]P =_\Delta [a = b]Q \quad$ if $a \neq b$
C3	$\varphi\ P\ P =_\Delta P$
C4	$\varphi\ P\ Q =_\Delta \neg\varphi\ Q\ P$
C5	$\varphi(\psi P) =_\Delta [\varphi \wedge \psi]P$
C6	$\varphi\ (P_1 + P_2)\ (Q_1 + Q_2) =_\Delta \varphi\ P_1\ Q_1 + \varphi\ P_2\ Q_2$
C7	$\varphi\ (\alpha.P) =_\Delta \varphi\ (\alpha.\varphi P) \quad$ if $bn(\alpha) \cap n(\varphi) = \emptyset$

R1	$(\nu a:T)(\nu b:S)P =_\Delta (\nu b:S)(\nu a:T)P$
R2	$(\nu a:T)(P + Q) =_\Delta (\nu a:T)P + (\nu a:T)Q$
R3	$(\nu a:T)\alpha.P =_\Delta \alpha.(\nu a:T)P \quad$ if $a \notin n(\alpha)$
R4	$(\nu a:T)\alpha.P =_\Delta \mathbf{0} \quad$ if $subj(\alpha) = a$
R5	$(\nu a:T)[a = u]P =_\Delta \mathbf{0} \quad$ if $a \neq u$
R6	$(\nu a:T)[u = v]P =_\Delta [u = v](\nu a:T)P \quad$ if $a \neq u, v$

E Assume $P \equiv \Sigma_i \varphi_i \alpha_i.P_i$ and $Q \equiv \Sigma_j \psi_j \beta_j.Q_j$ where no α_i (resp. β_j) binds a name free in Q (resp. P). Let $\Gamma \vdash P \mid Q$. Then infer:

$$P \mid Q =_\Delta \textstyle\sum_i \varphi_i\alpha_i.(P_i \mid Q) + \sum_j \psi_j\beta_j.(P \mid Q_j) + \sum_{\alpha_i\ opp\ \beta_j} [\varphi_i \wedge \psi_j \wedge (u_i = v_j)]\tau.R_{ij}$$

where α_i *opp* β_j, u_i, v_j and R_{ij} are defined as follows:
1. α_i is $\bar{u}_i w$, β_j is $v_j(x:T)$ and $\Gamma(w) <: T$; then R_{ij} is $P_i \mid Q_j\{w/x\}$;
2. α_i is $\bar{u}_i(w:S)$, β_j is $v_j(x:T)$ and $S <: T$; then R_{ij} is $(\nu w:S)(P_i \mid Q_j\{w/x\})$;
3. the converse of (1) or (2).

A	$P =_\Delta Q \quad$ if P alpha-equivalent to Q

Table 3. The axiom system $\mathcal{A}$

because $\Delta(x)_{\mathtt{o}} = \mathtt{b}T$ and no c in Δ satisfies the condition $\Delta_{\mathtt{c}} \vdash c : \mathtt{b}T$ and can therefore instantiate y. However,

$$x(y : \mathtt{o}T).[y = b]\tau \not\sim_{\Delta} x(y : \mathtt{o}T).\mathbf{0}.$$

To see this, let us look at the possible closing substitutions. In $dom(\Delta_{\mathtt{c}})$, a is the only channel satisfying $\Delta_{\mathtt{c}} \vdash a : \Delta(x)$, and so the only substitution we need to consider is $\{a/x\}$. After applying this substitution, the resulting closed terms are not bisimilar:

$$a(y : \mathtt{o}T).[y = b]\tau \not\sim_{\Delta} a(y : \mathtt{o}T).\mathbf{0}$$

This holds because the observer can send b along a and, after the communication, y is instantiated to be b, thus validating the condition $y = b$ and liberating the prefix τ. When the subject of the prefix is a variable, the following rule is needed in place of **Iin***:

Iv1 If $P =_{\Delta, y:\Delta(x)_{\mathtt{i}}} Q$ then $x(y : T_1).P =_{\Delta} x(y : T_2).Q$

In rule **Iout***, both the subject and object of the output prefix are channels. The rule is also valid when the object is a variable. However, it is not valid if the subject is a variable. As a counterexample, let $\Delta_{\mathtt{c}} \stackrel{\text{def}}{=} a : \mathtt{i}T, b : \mathtt{bb}T$ and $\Delta \stackrel{\text{def}}{=} \Delta_{\mathtt{c}}, x : \mathtt{b}\langle \mathtt{i}T, \mathtt{b}T\rangle$. Then we have

$$a \sim_{\Delta \sqcap a:\mathtt{i}T} \mathbf{0}$$

but

$$\bar{x}a.a \not\sim_{\Delta} \bar{x}a.\mathbf{0}$$

because, under the substitution $\{b/x\}$,

$$\bar{b}a.a \not\sim_{\Delta} \bar{b}a.\mathbf{0}.$$

When the subject of the prefix is a variable, we need the following rule:

Iv2 If $P =_{\Delta \sqcap v:\Delta(x)_{\mathtt{o}}} Q$ then $\bar{x}v.P =_{\Delta} \bar{x}v.Q$

We show, by means of an example, why rules **Iin*** and **Iout*** are sufficient in the axiom system (rules **Iv1** and **Iv2** are derivable). Consider the equality

$$x(y : \mathtt{ii}T).y \sim_{\Delta} x(y : \mathtt{io}T).\mathbf{0}$$

where $\Delta \stackrel{\text{def}}{=} a : \mathtt{bib}T, b : \mathtt{ib}T, x : \mathtt{bib}T$. First, we infer

$$y =_{\Delta'} \mathbf{0} \quad \text{for } \Delta' = \Delta, y : \mathtt{ib}T \tag{1}$$

proceeding as follows (we only report the main rules used in the reasoning; rule **C8** is the derived rule given at the beginning of this section):

$$
\begin{array}{ll}
y =_{\Delta'} [y = b]y + [y \neq b]y & \textbf{C8} \\
=_{\Delta'} [y = b]y & \textbf{Tvar}* \\
=_{\Delta'} [y = b]b & \textbf{Tpre}* \\
=_{\Delta'} [y = b]\mathbf{0} & \textbf{Tin}*, \textbf{Icon} \\
=_{\Delta'} \mathbf{0} & \textbf{C3}
\end{array}
$$

Then we derive $x(y : \mathtt{ii}T).y =_{\Delta} x(y : \mathtt{io}T).\mathbf{0}$ in a similar way:

$$
\begin{array}{ll}
\quad x(y : \mathtt{ii}T).y & \\
=_{\Delta} [x = a]x(y : \mathtt{ii}T).y + [x \neq a]x(y : \mathtt{ii}T).y & \textbf{C8} \\
=_{\Delta} [x = a]x(y : \mathtt{ii}T).y & \textbf{Tvar}* \\
=_{\Delta} [x = a]a(y : \mathtt{ii}T).y & \textbf{Tpre}* \\
=_{\Delta} [x = a]a(y : \mathtt{io}T).\mathbf{0} & (\mathbf{1}), \textbf{Iin}*, \textbf{Icon} \\
=_{\Delta} x(y : \mathtt{io}T).\mathbf{0} & \textbf{Tpre}*, \textbf{Tvar}*, \textbf{C8}
\end{array}
$$

Theorem 1 (Soundness and Completeness of $\mathcal{A}$). *$\mathcal{A} \vdash P =_{\Delta} Q$ iff $P \sim_{\Delta} Q$.*

The schema of the completeness proof is similar to that for the untyped π-calculus [9]. The details, however, are quite different. An example of this is the manipulation of terms underneath input and output prefixes mentioned above. We discuss below another example, related to the issue of invariance of bisimilarity under injective substitutions.

In the untyped case, the process $x \mid \overline{a}$ is equal to $x.\overline{a} + \overline{a}.x + \tau$ when x is instantiated to a, to $x.\overline{a} + \overline{a}.x$ otherwise. This can be expressed by expanding the process by mean of conditions: that is, using conditions to make a case analysis on the possible values that the variable may take. Thus, $x \mid \overline{a}$ is expanded to $[x = a](x \mid \overline{a}) + [x \neq a](x \mid \overline{a})$. Now, underneath $[x = a]$ we know that x will be a, and therefore $x \mid \overline{a}$ can be rewritten as $x.\overline{a} + \overline{a}.x + \tau$, whereas underneath $[x \neq a]$ we know that x will not be a and therefore $x \mid \overline{a}$ can be rewritten as $x.\overline{a} + \overline{a}.x$. In general, the expansion of a process with a free variable x produces a summand $[x \neq a_1] \cdots [x \neq a_n]P$ where $a_1, \cdots, a_n$ are all channels (different from x) that appear free in P. The mismatch $[x \neq a_1] \cdots [x \neq a_n]$ tells us that x in P will be instantiated to a fresh channel, which is sufficient for all manipulations of P involving x, since bisimulation is invariant under injective substitutions. In the typed calculus, by contrast, knowing that x is fresh may not be sufficient: we may also need the information on the type with which x will be instantiated. This type may be different from the type T of x in the type environment: x could be instantiated to a fresh channel whose type is a *subtype* of T (the behavioral consequences of this type information can be seen in the example at the end of Section 5.1). We have therefore adopted a strategy different from that in the proof for untyped calculi: rather than manipulating processes that begin with "complete" sequences of mismatches — as in the untyped case — we try to cancel them, using rule **Tvar***; further, the conditional expansion of a process takes into account also the names that appear in the type environment.

4 A proof system for the closed terms

The system of Section 3 can be simplified if we limit ourselves to proving equalities on *closed* terms. With one caveat: the substitutivity rule for input is replaced by an inference rule, where (possibly several) instantiations of the bound variable of the input are considered. We call $\mathcal{P}$ the system of rules; it is presented in Table 4.

Rules **Tin*, Tout*, Iout*, Itau, Isum, Ires*, Ipar*, Twea*, S1-4, R1-4, E, A** as in Table 1, plus the following ones:

Iin If $P\{b/x\} =_\Delta Q\{b/x\}$ for all b s.t. $\Delta_c \vdash b : \Delta(a)_o$ then $a(x:T_1).P =_\Delta a(x:T_2).Q$.

Ca $\varphi\ P\ Q =_\Delta P$ if $[\![\varphi]\!] = \mathit{True}$
Cb $\varphi\ P\ Q =_\Delta Q$ if $[\![\varphi]\!] = \mathit{False}$

R $(\nu a:T)\mathbf{0} =_\Delta \mathbf{0}$

Table 4. The proof system $\mathcal{P}$ for the closed terms

Note that rules **Icon, Tvar*, Tpre*, R5-6** are not needed, and that the set of rules **C1-7** for conditions is cut down to just **Ca-b**. In the previous system $\mathcal{A}$, axiom **R** was redundant in view of **C3** and **R5**.

Theorem 2 (Soundness and completeness of $\mathcal{P}$). *$\mathcal{P} \vdash P =_\Delta Q$ iff $P \sim_\Delta Q$, where P and Q are closed.*

Also the proofs of soundness and completeness are simpler. The two main problems in the proofs of Section 3 (manipulations of conditions and of open terms underneath prefixes) do not arise now, because all terms are closed and because conditions are removed as soon as possible, by means of **Ca-b**, without any conditional expansion. Compared with proof systems for untyped π-calculus [9], **Tin*** and **Tout*** are the main differences.

5 Other equivalences

5.1 Hennessy and Rathke's typed bisimilarity

In the input clause of $\sim$ (Definition 2), the type environment Δ is not extended. By contrast, extensions are allowed in the bisimilarity used in [3]. We denote with $\asymp_\Delta$ the variant of $\sim_\Delta$ which allows extension; its definition is obtained from that of $\sim_\Delta$ by using the following input clause:

– if $\Delta \sharp P \stackrel{a(x:T)}{\longrightarrow} \Delta' \sharp P'$, then for some Q', $\Delta \sharp Q \stackrel{a(x:S)}{\longrightarrow} \Delta'' \sharp Q'$ and $\Delta, \Delta' \vdash b : \Delta(a)_\circ$ implies $P'\{b/x\}\ \mathcal{R}_{\Delta,\Delta'}\ Q'\{b/x\}$, for any channel b and closed type environment Δ' with $dom(\Delta') \cap (fn(P,Q) \cup dom(\Delta)) = \emptyset$.

Similarly, Δ can be extended in the definition on open terms.

In $\asymp_\Delta$, the environment collects the knowledge of the observer *relative* to the tested processes, in the sense that the environment only tells us what the observer knows of the free channels of the processes. In contrast, in $\sim_\Delta$, the environment collects the *absolute* knowledge of the observer, including information on channels that at present do not appear in the tested processes, but that might appear later — if the observer decides to send them to the processes. The main advantage of $\asymp_\Delta$ is that the environment is smaller. On the other hand, $\sim_\Delta$ allows us to express more refined interrogations on the equivalence of processes, for it gives us more flexibility in setting the observer knowledge. Indeed, while $\asymp$-equivalences can be expressed using $\sim$ (Lemma 1), the converse is false. For instance, the processes

$$P \stackrel{\text{def}}{=} a(x : \mathtt{bo}T).[x = y]\tau \qquad Q \stackrel{\text{def}}{=} a(x : \mathtt{bo}T).\mathbf{0}$$

are in the relation $\sim_\Delta$, for $\Delta \stackrel{\text{def}}{=} a : \mathtt{obo}T, b : \mathtt{bb}T, y : \mathtt{ob}T$. However, they are not in a relation $\asymp_\Gamma$, for any Γ: the observer can always create a new channel of type $\mathtt{bo}T$, and use it to instantiate both x and y, thus validating the condition $[x = y]$.

Lemma 1. *If $P \asymp_\Delta Q$ then $P \sim_\Delta Q$.*

We can derive a proof system for $\asymp$ with a simple modification of that for $\sim$ in Section 4. Let $\mathcal{P}'$ be the system obtained from $\mathcal{P}$ by replacing rule **Iin** with **Iin′**:

Iin′ If – $P\{b/x\} =_\Delta Q\{b/x\}$ for all b with $\Delta(b) <: \Delta(a)_\circ$, and
– given $c \notin fn(P,Q) \cup dom(\Delta)$,
$P\{c/x\} =_{\Delta,c:T} Q\{c/x\}$ for all $T <: \Delta(a)_\circ$,
then $a(x : T_1).P =_\Delta a(x : T_2).Q$.

The quantification on T in the premises is finite: any type has only finitely-many subtypes.

Theorem 3. *$\mathcal{P}' \vdash P =_\Delta Q$ iff $P \asymp_\Delta Q$, where P and Q are closed.*

By contrast, we have tried and failed to obtain the counterpart of Theorem 1 for $\asymp$. The encountered problem is discussed at the end of this subsection. Using Lemma 2, that relates $\asymp$ to $\sim$, we however obtain an indirect axiomatisation of $\asymp$.

We say that $P_1 \asymp_\Delta P_2$ *under* Γ_1, Γ_2 if $\Gamma_i <: \Delta$ and $\Gamma_i \vdash P_i$ $(i = 1, 2)$.

Lemma 2. *$P_1 \asymp_\Delta P_2$ under Γ_1, Γ_2 iff $P_1\ \sim_{\Delta, Env(\Delta, P_1, P_2, \Gamma_1, \Gamma_2)}\ P_2$.*

As a consequence of this lemma, we obtain the following theorem.

Theorem 4. *$P_1 \asymp_\Delta P_2$ under Γ_1, Γ_2 iff $\mathcal{A} \vdash P_1 =_{\Delta, Env(\Delta, P_1, P_2, \Gamma_1, \Gamma_2)} P_2$.*

Directly axiomatizing $\asymp$ appears far from straightforward due to complications entailed by subtyping. We consider an example. Let $T \stackrel{\text{def}}{=} \mathtt{unit}$ and

$$\begin{aligned}
\Delta &\stackrel{\text{def}}{=} a : \mathtt{obo}T, y : \mathtt{ob}T \\
R &\stackrel{\text{def}}{=} \tau.((\nu c : \mathtt{b}T)\bar{y}c.\bar{c} + a(x : \mathtt{bo}T).[x = y]\tau) \\
R_1 &\stackrel{\text{def}}{=} \tau.((\nu c : \mathtt{b}T)\bar{y}c.\mathbf{0} + a(x : \mathtt{bo}T).[x = y]\tau) \\
R_2 &\stackrel{\text{def}}{=} \tau.((\nu c : \mathtt{b}T)\bar{y}c.\bar{c} + a(x : \mathtt{bo}T).\mathbf{0}).
\end{aligned}$$

It holds that

$$R + R_1 + R_2 \asymp_\Delta R_1 + R_2.$$

Here y can be instantiated by channels with subtypes of $\mathtt{ob}T$, which can be seen in Fig. 1 (b). When y is instantiated by a channel with type $\mathtt{bo}T$, we can simulate R with R_1. For other subtypes of $\mathtt{ob}T$, we can simulate R with R_2. That is, we have two equivalent processes, say P and Q, with a free variable y, and the actions from a summand of P have to be matched by different summands of Q, depending on the types used to instantiate y. It appears hard to capture this relationship among terms using axioms involving only the standard operators of the π-calculus.

5.2 Early bisimilarity

All bisimilarities considered so far in the paper are in the "late" style [12]. As usual, the "early" versions are obtained by commuting the quantifiers in the input clause of bisimilarity. As in the untyped case, the difference between late and early equivalences is captured by the axiom **SP** [9]:

$$\begin{aligned}
\textbf{SP} \qquad & a(x : T_1).P + a(x : T_2).Q \\
=_\Delta\ & a(x : T_1).P + a(x : T_2).Q + a(x : T_3).([x = u]PQ)
\end{aligned}$$

All results in the paper also hold for the early versions of the equivalences, when rule **SP** is added.

6 Future work

As future work, the following problems need to be solved.

- Due to the difficulty discussed at the end of Section 5.1 we are only able to give an indirect axiomatisation of $\asymp$. We are not clear whether it is possible to directly axiomatize the equivalence in the language considered in the current paper.
- In our type system we allow matching names to have arbitrary types. It is not clear how to relax our use of matching. Limiting matching to names of compatible types would pose a problem for subject reduction. On the other

hand, allowing matching only on names with types of the form $\mathtt{b}T$, as in [10], would seem difficult, for matching plays an important role in axiomatisations. For example, one would not be able to rewrite $x \mid \bar{y}$ as $x.\bar{y} + \bar{y}.x + [x = y]\tau$ under the type environment $\Delta = x : \mathtt{i}T, y : \mathtt{o}T$. In [3], a particular typing rule for matching was presented, which allowed meet of types on successful matches. It might be interesting to know whether the presence of this typing rule would affect the validity of our proof systems.
- For the variant bisimilarity $\asymp$, as well as the typed bisimilarity defined in [12], there are results that relate them to contextual equivalences such as barbed equivalence. It would be interesting to see what kind of contextual equivalence (if any) corresponds to $\sim$.
- Another issue is axiomatisations of typed *weak* bisimilarities. In this case, however, types may not be so central, in that the addition of the usual tau laws [7] might be sufficient.

Acknowledgements We are grateful to Catuscia Palamidessi, Pierre-Louis Curien and the anonymous referees for comments on a preliminary version of the paper.

References

1. M. Boreale and R. De Nicola. Testing equivalences for mobile processes. *Journal of Information and Computation*, 120:279–303, 1995.
2. M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proceedings of LICS '98*. IEEE, Computer Society Press, 1998.
3. M. Hennessy and J. Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14:651–684, 2004.
4. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Journal of Information and Computation*, 173:82–120, 2002.
5. H. Lin. Symbolic bisimulation and proof systems for the π-calculus. Technical Report 7/94, School of Cognitive and Computing Sciences, University of Sussex, UK, 1994.
6. H. Lin. Complete inference systems for weak bisimulation equivalences in the π-calculus. *Journal of Information and Computation*, 180(1):1–29, 2003.
7. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
8. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
9. J. Parrow and D. Sangiorgi. Algebraic theories for name-passing calculi. *Journal of Information and Computation*, 120(2):174–197, 1995.
10. B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
11. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996.
12. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.