Assigning Types to Processes

(Extended Abstract)

NOBUKO YOSHIDA
Dept. of Mathematics and Computer Science
University of Leicester

MATTHEW HENNESSY
School of Cognitive and Computing Sciences
Universsiry of Sussex

ABSTRACT

In wide area distributed systems it is now common for *higher-order code* to be transferred from one domain to another; the receiving host may initialise parameters and then execute the code in its local environment. In this paper we propose a fine-grained typing system for a higher-order π-calculus which can be used to control the effect of such migrating code on local environments. Processes may be assigned different types depending on their intended use. This is in contrast to most of the previous work on typing processes where all processes are typed by a unique constant type, indicating essentially that they are well-typed relative to a particular environment.

Our process type takes a form of an *interface* limiting the resources to which it has access, and the types at which they may be used. Allowing resource names to appear both in process types and process terms, as interaction ports, complicates the typing system considerably. For the development of a coherent typing system, we use a kinding technique, similar to that used by the subtyping of the system F, and the order-theoretic property (*finite bounded completeness*) of our subtyping relation.

Various examples of this paper illustrate the usage of our fine-grained typing system for distributed systems. As a specific application we define a new typed behavioural equivalence for the higher-order π-calculus. The expressiveness of our types enables us to state and prove interesting identities between typed processes.

Corresponding Author: Nobuko Yoshida. Address: Department of Mathematics and Computer Science, University of Leicester, University Road, Leicester, LE1 7RH, UK. Tel: +44 116 252 3895 (direct), Fax: +44 116 252 3604 or 3915. E-Mail: `ny11@mcs.le.ac.uk`.

1 Introduction

Background In the distributed computing environments nowadays, it is common for *higher-order code* to be transferred from one domain to another [13, 23, 25]. This is recognised as very dangerous and various schemes have been put forward to ensure the integrity of systems in the presence of such operations. In this paper we propose a new *subtyping system* which can be used to control the effect of migrating code on local environments. Our investigation is in terms of a higher-order π-calculus in which values, including process terms, can be exchanged along communication channels [3, 29, 35]. We believe that our typing system based on fine-grained process types can be readily adapted to related location based distributed calculi such as those presented in [6, 4, 9, 11, 15, 33].

Higher-Order Processes The language we consider, $\lambda\pi_v$, is essentially a call-by-value λ-calculus augmented with the π-calculus primitives [22]. For example,

$$c?(x{:}\tau)\; f\, x \tag{1}$$

is a process which inputs a value of type τ on channel c and applies to it the function f. This process will be well-typed only in an environment in which the channel c has the capability to input values of type τ, written $c : (\tau)^{\mathtt{I}}$, and f denotes a function of type $(\tau \to \mathtt{proc})$; here, as in [11, 26, 35], we use `proc` to denote the type of processes.

As usual we allow as values arbitrary abstractions, but much of the descriptive power of $\lambda\pi_v$ comes from the ability to form values by abstracting over processes. For example $(\mathtt{unit} \to \mathtt{proc})$ is the type of *thunked* processes; we use this type so frequently that we will abbreviate it to $\langle\mathtt{proc}\rangle$. Values of this type, of the form $\lambda(x{:}\mathtt{unit}).P$, will also be abbreviated to $\langle P\rangle$. Such values can be exchanged between processes and subsequently executed, by applying the function $\lambda y.y()$; again for the sake of clarity we use *run* to denote this function. So in (1) above, if τ is the thunked type $\langle\mathtt{proc}\rangle$ and f is the function *run*, the process may input a thunked process $\langle P\rangle$ on channel c and execute it.

In papers such as [3, 26, 35, 27, 11] typing systems have been suggested which ensure that programs written in $\lambda\pi_v$-related languages are well-behaved. The main judgements normally take the form

$$\Gamma \vdash P : \mathtt{proc}$$

indicating that the term P is a well-typed process relative to the typing environment Γ. Here Γ is a mapping from channel names or variables to input/output capabilities or value types; $\Gamma(c)$ determines the type of values which channel c may transmit/receive. Thus the process above, (1), will be well-typed in any environment Γ which allows c the input capability $\langle\mathtt{proc}\rangle$, assuming of course that f is also a well-typed expression of type $\langle\mathtt{proc}\rangle \to \mathtt{proc}$.

However such typing offers limited control to programs over the code which they download for execution. To emphasise this point let us consider an example. First we define the abstraction Fw (called *forwarder*)

$$\lambda x\, \lambda y\; (*\, x?(z{:}\mathtt{int})\; y!\langle z\rangle)$$

which repeatedly inputs some value of a type `int` on channel x and outputs it immediately on y. If the channels a, b are assigned suitable types then both the values $\langle\mathsf{Fw}(ab)\rangle$ and $\langle\mathsf{Fw}(ba)\rangle$ have type $\langle\mathtt{proc}\rangle$ and thus may be sent along channel c to a process such as

$$c?(x:\tau)\;\mathit{run}\;x \tag{2}$$

But accepting these processes for execution confers on the incoming code very different *access rights*. In the first case the incoming code is allowed to read from channel a and write to channel b while in the second case these rights are reversed. Typing systems in which code can only be assigned the undifferentiated type `proc` does not provide any mechanism for limiting the effect of executing incoming code.

Typing processes In this paper we extend the typing systems of [35, 26, 15] by allowing processes to have types which bound the resources which they may use. The basic idea is straightforward. For processes in $\lambda\pi_v$ we will allow judgements of the form

$$\Gamma \vdash P : [\Delta]$$

where Δ is a finite environment, mapping channel names to capabilities. Intuitively this means that relative to Γ the term P denotes a well-defined process which uses *at most* the resources in the domain of Δ; moreover their use is in accordance with the capabilities given in Δ. Thus the type $[\Delta]$ may be viewed as a process *interface*.

For example let Δ_{ab}, Δ_{ba} denote the environments

$$\{a{:}(\mathtt{int})^{\mathtt{I}}, b{:}(\mathtt{int})^{\mathtt{O}}\}, \qquad \{b{:}(\mathtt{int})^{\mathtt{I}}, a{:}(\mathtt{int})^{\mathtt{O}}\}$$

respectively where $(\mathtt{int})^{\mathtt{I}}$ and $(\mathtt{int})^{\mathtt{O}}$ represent the input and output capabilities of a type `int`. Then, for a suitable Γ we will be able to derive the judgements

$$\Gamma \vdash \mathsf{Fw}(ab) : [\Delta_{ab}] \qquad \text{and} \qquad \Gamma \vdash \mathsf{Fw}(ba) : [\Delta_{ba}]$$

These more discriminating types for processes allows processes to be, in turn, more discriminating in the type of values which they will accept. Thus

$$c?(x{:}\langle\Delta_{ab}\rangle)\;\mathit{run}\;x$$

indicates that it is only willing to accept processes for execution if they at most read from resource a and write to b. Let us denote $c!\langle P\rangle$ for an output process which sends a thunked process $\langle P\rangle$ to a channel c. Thus, for example, a process

$$c!\langle\mathsf{Fw}(ab)\rangle \;\mid\; c?(x{:}\langle\Delta_{ab}\rangle)\;\mathit{run}\;x$$

is well-typed while

$$c!\langle\mathsf{Fw}(ba)\rangle \;\mid\; c?(x:\langle\Delta_{ab}\rangle)\;\mathit{run}\;x$$

is not; the process $\langle\mathsf{Fw}(ba)\rangle$ is not acceptable along c as it does not conform to the *interface* decreed by the host, Δ_{ab}.

This ability to constrain the effect of imported code means that hosts processes can maintain the consistency of local resources. For example consider the following example: nesting process types give even further control over code behaviour.

$$*\mathtt{req}?(y:\langle\Delta_c\rangle)(run\ y \mid c?(x:\langle\Delta_a\rangle)(run\ x \mid a?(z:\mathtt{int})P))$$

where $\langle\Delta_a\rangle$ denotes the type $\langle a:(\mathtt{int})^{\mathtt{O}}\rangle$ and $\langle\Delta_c\rangle$ denotes $\langle c:\langle\Delta_a\rangle^{\mathtt{O}}\rangle$. The annotated types ensure that

- the code downloaded on the request channel `req` can only access the resource c
- c can only be used to transmit code which can at most access resource a
- all communications to a will be serviced by $a?(z:\mathtt{int})P$.

Channel abstractions In $\lambda\pi_v$ processes are also allowed to download *abstracted code*; code in which resource parameters may be instantiated by the host process before the code is executed. A simple example is the abstraction Fw used above. Consider the server

$$*\, s?(z)\, z!\langle\mathsf{Fw}\rangle$$

which continually supplies the abstraction Fw to requesting clients. A specific client, such as R defined by

$$s!\langle c\rangle\ c?(y)\ (y\ a\ b),$$

can download the code Fw and instantiate it with particular channels, such as a, b. Thus in the presence of the server R will evolve to a process which should have a type of the form

$$[a:(\mathtt{int})^{\mathtt{I}}, b:(\mathtt{int})^{\mathtt{O}}, \ldots]$$

Other processes which instantiate Fw differently will evolve to processes with different types. For example S defined by

$$s!\langle c\rangle\ c?(y)\ (y\ b\ a),$$

will evolve to a process with a type

$$[b:(\mathtt{int})^{\mathtt{I}}, a:(\mathtt{int})^{\mathtt{O}}, \ldots]$$

However it is difficult to see how to give a type to the abstraction Fw which ensures that R and S are assigned such types. Within our current system of types it would be natural to assign to Fw a functional type of the form

$$(\mathtt{int})^{\mathtt{I}} \to (\mathtt{int})^{\mathtt{O}} \to \pi$$

for some process type π. If π is the undifferentiated type `proc` then both R and S would inherit this uninformative type. Otherwise π must assign some definite capabilities to a and b and assuming that typing is preserved under Subject Reduction these capabilities would be inherited by R and S. That is, they would have the same capabilities on the two resources a and b, contrary to our requirements.

Our solution is to introduce a new form of *dependent* functional type

$$(x:\sigma) \to \rho$$

Here σ is a channel type and we allow the type ρ to contain occurrences of the channel variable x. (These occurrences of x in ρ are bound occurrences in the dependent type $(x:\sigma) \to \rho$.) Thus the abstraction Fw will be assigned the type

$$(x:(\mathtt{int})^{\mathtt{I}}) \to (\mathtt{y}:(\mathtt{int})^{\mathtt{O}}) \to [\mathtt{x}:(\mathtt{int})^{\mathtt{I}}, \mathtt{y}:(\mathtt{int})^{\mathtt{O}}]$$

where the result type of the process depends on the type of the abstracted variables.

Results In this paper we formulate a fine-grained subtyping system for the higher-order π-calculus following the basic ideas mentioned above. This results in a simple, but non-trivial, extension of the IO-subtyping for π-calculus [26] to the higher-order setting. The main technical results can be summarised as follows.

1. First, we design a novel notion of *type* for the higher-order π-calculus, in which process have types corresponding to *interfaces*. We also define a subtyping partial order $\leq$ over the set of types which satisfies *finite bounded completeness*, [15, 35].

   Because type variables appear both in program terms and in types (in particular the types of processes), the formal definition of what constitutes a valid term and a valid type are interdependent and both in turn require a careful definition of even a valid typing environment. An analogous, albeit somewhat simpler, situation arises in subtyping for the polymorphic λ-calculus [7]. We clarify technical similarities and differences between the two typing systems. Although some techniques developed for the λ-calculus are useful in the current setting, novel concepts are required for process types.
2. Second, we propose a typing system in which many higher-order π-calculus processes can be assigned non-trivial *interface* types. We prove its soundness (Subject Reduction Theorem) and also establish a Type Safety Theorem, which ensures that no well-typed process can input higher-order code which does not conform the local interface of that process.
3. Finally we define a new typed behavioural equivalence for the higher-order π-calculus. Our fine-grained types enable us to establish general, and useful, process identities. Their application is illustrated on an example distributed system.

Outline of the Paper Section 2 introduces the types, syntax and a reduction semantics of $\lambda\pi_v$. Section 3 proposes the new typing system of $\lambda\pi_v$, explaining the various technical points in the formulation of the inference rules and demonstrates its expressiveness by typing the example in Section 2. In Section 4, we prove Subject Reduction and Type Safety Theorems. Section 5 briefly studies the typed behavioural equalities. Finally Section 6 concludes the paper with an extension of this system, further issues and related work. Due to space limitations, the proofs and many examples, including a further application to a distributed higher-order π-calculus, are left to the full version of this paper [36], available from: `http://www.mcs.le.ac.uk/~nyoshida/paper.html`.

(Type) $\alpha, \beta, \gamma, \ldots$

$$\begin{array}{lrcl}
\text{Term:} & \rho & ::= & \pi \mid \sigma_H \\
\text{Base:} & \sigma_G & ::= & \texttt{unit} \mid \texttt{nat} \mid \cdots \\
\text{Process:} & \pi & ::= & [\Delta] \mid \texttt{proc} \\
\text{HO Value:} & \sigma_H & ::= & \sigma_G \mid \sigma_H \to \rho \mid (x{:}\sigma) \to \rho \\
\text{Channel:} & \sigma & ::= & \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \\
\text{Value:} & \tau & ::= & \sigma_H \mid \sigma \\
\text{Sort:} & S & ::= & (\tau_1, \ldots, \tau_n) \mid \top \mid \bot
\end{array}$$

(Environment)

$$\begin{array}{l}
\text{Channel:} \\
\quad \Delta \;::=\; \emptyset \mid \Delta, u{:}\sigma \\
\text{General:} \\
\quad \Gamma \;::=\; \emptyset \mid \Gamma, x{:}\tau \mid \Gamma, a{:}\sigma
\end{array}$$

(Abbreviation)

$$\begin{array}{ll}
\text{input only:} & S^{\mathtt{I}} \overset{\text{def}}{=} \langle S, \bot \rangle \\
\text{output only:} & S^{\mathtt{O}} \overset{\text{def}}{=} \langle \top, S \rangle \\
\text{input/output:} & S^{\mathtt{IO}} \overset{\text{def}}{=} \langle S, S \rangle \\
 & \\
\text{thunk type:} & \langle \Delta \rangle \overset{\text{def}}{=} \texttt{unit} \to [\Delta]
\end{array}$$

FIGURE 1. Types

(Term)

$$\begin{array}{rcll}
P, Q, \ldots & ::= & V & \text{value} \\
 & \mid & \mathbf{0} & \text{nil} \\
 & \mid & P \mid P & \text{parallel} \\
 & \mid & u!\langle V_1, \ldots, V_n \rangle P & \text{output} \\
 & \mid & u?(x_1 : \tau_1, \ldots, x_n : \tau_n) P & \text{input} \\
 & \mid & *P & \text{replicator} \\
 & \mid & (\nu\, a{:}\sigma) P & \text{restriction} \\
 & \mid & P P & \text{application}
\end{array}$$

(Identifier)

$$\begin{array}{rcll}
u, v, w, \ldots & ::= & l & \text{literal} \\
 & \mid & x, y, z, \ldots & \text{variable} \\
 & \mid & a, b, c, \ldots & \text{channel}
\end{array}$$

(Value)

$$\begin{array}{rcll}
V, W, \ldots & ::= & u, v, w, \ldots & \text{identifier} \\
 & \mid & \lambda(x{:}\tau) P & \text{abstraction}
\end{array}$$

(Literal)

$$\begin{array}{rcll}
l, l', \ldots & ::= & () & \text{unit} \\
 & \mid & 1, 2, 3, \ldots & \text{number}
\end{array}$$

(**Abbreviation**)

$$\begin{array}{ll}
\langle P \rangle \overset{\text{def}}{=} \lambda(x{:}\texttt{unit}) P & \text{thunk} \\
\mathit{run} \overset{\text{def}}{=} \lambda(x{:}\texttt{unit} \to \pi)\, x() & \text{run}
\end{array}$$

FIGURE 2. Syntax

2 A Higher-order Process Language

Types The collection of types is a straightforward extension of that from [35]; a process type, ranged over by π, can either be the constant `proc`, as in [35], or take the form $[\Delta]$, where Δ is an *environment*. The formal definition is given in Figure 1; this assumes a set of base types such as `unit` and `nat`, an infinite set of channel or resource *names* $\mathbf{N}$, ranged over by $a, b, \ldots$, and an infinite set of *variables* $\mathbf{V}$, ranged over by $x, y, \ldots$ For the sake of clarity we will sometimes use $X, Y, \ldots$ as variables, whenever we intend them to be substituted specifically by higher order values rather than channels.

Channel types are as in [35], in turn an elaboration of the IO-types of [15, 26]; they take the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$, a pair consisting of an *input sort* $S_{\mathtt{I}}$ and an *output sort* $S_{\mathtt{O}}$; these input/output sorts are in turn either a general value type or $\top$, denoting the highest capability, or $\bot$, denoting the lowest; as explained in [35] the representation of IO-types as a tuple [16, 15] makes the integration with the arrow types of the λ-calculus more natural. Moreover the IO-types of [26] can also be represented as a special case of our IO-types, using the abbreviations given in Figure 1.

There are three kinds of value types: base types, channel types as already explained or HO-value types, ranged over by σ_H. These can be formed using either of the functional type constructors, $\sigma_H \to \rho$ or $(x{:}\sigma) \to \rho$, where ρ in turn is either a HO-value type or a process type. As already explained process types can either be the constant `proc`, (also denoted *ok* in [26]) or a type environment $[\Delta]$ where Δ is a mapping from $\mathbf{N} \cup \mathbf{V}$ to channel types; the formation rules for environments are also given in Figure 1.

EXAMPLE 2.1. (Types)

(1) The empty process, with no capabilities, has the type $[\ ]$.

(2) A process which can output `nat` at a and input `bool` at b has the type $[a{:}(\texttt{nat})^{\mathtt{O}}, b{:}(\texttt{bool})^{\mathtt{I}}]$.

(3) A higher order process which can output a thunked values of type (2) at c has the type $[c{:}\langle a{:}(\texttt{nat})^{\mathtt{O}}, b{:}(\texttt{bool})^{\mathtt{I}} \rangle^{\mathtt{O}}]$.

(4) A higher order identity function over thunked values of type (2) has the type $\langle a{:}(\texttt{nat})^{\mathtt{O}}, b{:}(\texttt{bool})^{\mathtt{I}} \rangle \to \langle a{:}(\texttt{nat})^{\mathtt{O}}, b{:}(\texttt{bool})^{\mathtt{I}} \rangle$.

(5) A dependent function which is applied to some name a and constructs a process of type $[b{:}(\texttt{nat})^{\mathtt{I}}, a{:}(\texttt{nat})^{\mathtt{O}}]$ has the type $(x{:}(\texttt{nat})^{\mathtt{O}}) \to [b{:}(\texttt{nat})^{\mathtt{I}}, x{:}(\texttt{nat})^{\mathtt{O}}]$.

Syntax The syntax for terms in the language $\lambda\pi_{\mathrm{v}}$ is given in Figure 2. It is essentially the same as that used in [35] except that we use the more expressive types, from Figure 1. We use the standard notational conventions, for example ignoring trailing occurrences of the empty process $\mathbf{0}$ and omitting type annotations unless they are relevant.

Nevertheless the use of more expressive types changes considerably the nature of the language as both resource names and resource variables may appear in types. This complicates the notions of free names $\mathsf{fn}(P)$ and free variables $\mathsf{fv}(P)$ of terms, which is required for the definition of substitution; this

(Reduction)

$$(\beta) \quad (\lambda(x{:}\tau)P)\,V \longrightarrow P\{V/x\}$$

$$(\mathrm{app}_r) \quad \frac{Q \longrightarrow Q'}{PQ \longrightarrow PQ'} \qquad (\mathrm{app}_l) \quad \frac{P \longrightarrow P'}{PV \longrightarrow P'V}$$

$$(\mathrm{com}) \quad a?(x_1{:}\tau_1,...,x_n{:}\tau_n)P \mid a!\langle V_1,...,V_n\rangle Q \longrightarrow P\{V_1,...,V_n/x_1,...,x_n\} \mid Q$$

$$(\mathrm{par}) \quad \frac{P \longrightarrow P'}{P\,|\,Q \longrightarrow P'\,|\,Q} \qquad (\mathrm{res}) \quad \frac{P \longrightarrow P'}{(\nu a{:}\sigma)P \longrightarrow (\nu a{:}\sigma)P'}$$

$$(\mathrm{str}) \quad \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

(Structure Equivalence)

- $P \equiv Q$ if $P \equiv_\alpha Q$.
- $P\,|\,Q \equiv Q\,|\,P \quad (P\,|\,Q)\,|\,R \equiv P\,|\,(Q\,|\,R) \quad P\,|\,\mathbf{0} \equiv P$ $\quad *P \equiv P\,|*P$
- $(\nu a{:}\sigma)\mathbf{0} \equiv \mathbf{0} \quad (\nu a{:}\sigma)P\,|\,Q \equiv (\nu a{:}\sigma)(P\,|\,Q)$ if $a \notin \mathsf{fn}(Q)$ $\quad (\nu a{:}\sigma)(\nu b{:}\sigma')P \equiv (\nu b{:}\sigma')(\nu a{:}\sigma)P$ if $a \notin \mathsf{fn}(\sigma')$ and $b \notin \mathsf{fn}(\sigma)$

FIGURE 3. Reduction

in turn is central to the reduction semantics. The formal definitions are available in the Appendix of [36], but as an example $\mathsf{fv}(u?(x_1:\tau_1,...,x_n:\tau_n)P) = \mathsf{fv}(u) \cup \mathsf{fv}(\tau_1) \cup ... \cup \mathsf{fv}(P) - \{x_1,...,x_n\}$.

Reduction Semantics The term P is called a *program* if it contains no free variables, i.e. $\mathsf{fv}(P) = \emptyset$. The reduction semantics is given in terms of a binary relation

$$P \longrightarrow Q$$

between programs and follows the standard approach from [22, 26, 29]; the formal definition is given in Figure 3 and should be understandable to those familiar with either the π-calculus or the λ-calculus. It uses the standard structural equivalence $\equiv$ of the π-calculus; the axioms for $\equiv$ is also given in Figure 3 and it is also assumed to be preserved by the parallel composition " $|$ " and restriction operator "ν". It has an associated reduction rule (str). We also use $\longrightarrow\!\!\!\!\rightarrow$ to denote the multi-step reduction. The main rules are β-reduction, (β), and communication (com). Both these require a definition of substitution of values for variable $P\{V/x\}$, which we have yet to define. Complications arise when the value to be substituted V is a channel name and is best explained with an example:

$$\lambda(x:(\mathtt{int})^{\mathtt{IO}}).\lambda(Y:\langle x:(\mathtt{int})^{\mathtt{I}}, a:(\mathtt{int})^{\mathtt{O}}\rangle) \quad (\mathit{run}\ Y\,|\,x!\langle\mathbf{0}\rangle\,|\,a?(z)\ r!\langle z\rangle) \qquad (3)$$

This function first takes some channel, say b, then takes a thunked process with a b-capability and an a-capability, sets it running and interacts with it via a and b. Suppose that it is applied to the specific channel b. Intuitively this means substituting the value b for free occurrences of the bound variable x in the body of the function. If the substitution ignores types in the body of the function, as is standard, we get

$$\top\{v/x\} = \top \qquad \bot\{v/x\} = \bot$$
$$\sigma_G\{v/x\} = \sigma_G \qquad \mathtt{proc}\{v/x\} = \mathtt{proc}$$
$$\begin{array}{rcl}\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle\{v/x\} & = & \langle S_{\mathtt{I}}\{v/x\}, S_{\mathtt{O}}\{v/x\}\rangle,\\ (\tau_1,...,\tau_n)\{v/x\} & = & (\tau_1\{v/x\},...,\tau_n\{v/x\})\\ (\sigma_H \to \rho)\{v/x\} & = & \sigma_H\{v/x\} \to \rho\{v/x\}\\ ((y{:}\sigma) \to \rho)\{v/x\} & = & (y{:}\sigma\{v/x\}) \to \rho\{v/x\} \ \ (x \neq y)\\ [\Delta]\{v/x\} & = & \sqcup[w\{v/x\}{:}\sigma\{v/x\}] \ \ (w{:}\sigma \in \Delta)\end{array}$$

FIGURE 4. Name Substitution into Types

$$\lambda(Y:\langle x:(\mathtt{int})^{\mathtt{I}}, a:(\mathtt{int})^{\mathtt{O}}\rangle)(\mathit{run}\ Y\,|\,b!\langle\mathbf{0}\rangle\,|\,a?(z)\ r!\langle z\rangle)$$

which is not even a program; it contains an occurrence of the free variable x. The proper definition of reduction requires that b is also substituted into the types occurring in the body of the function, to give the program

$$\lambda(Y:\langle b:(\mathtt{int})^{\mathtt{I}}, a:(\mathtt{int})^{\mathtt{O}}\rangle)(\mathit{run}\ Y\,|\,b!\langle\mathbf{0}\rangle\,|\,a?(z)\ r!\langle z\rangle)$$

This also makes sense as this now constrains the function to be only applied to processes which have an appropriate b-capability.

The formal definition of value substitution into terms, $P\{V/x\}$, is defined inductively on the structure of terms. The following is one instance for an input process:

$$u?(x{:}\tau)\ P\{V/x\} \stackrel{\mathrm{def}}{=} u\{V/x\}?(x{:}\tau\{V/x\})\ P\{V/x\}$$

However this instance uses the substitution of values into types, $\tau\{V/x\}$. If V is anything other than a channel name or variable this is the identity. So we need only define the substitution $\tau\{v/x\}$, where v is a channel or channel variable; this is defined in Figure 4. We will see later that when these substitutions are made to well-defined terms and types, in well-defined environments, the results will also be well-defined.

The definition of substitution into types is for the most part straightforward, with one exception which can be explained using the example function (3) above. If this is applied to the name a we would expect to get the result

$$\lambda(Y{:}\langle a:(\mathtt{int})^{\mathtt{IO}}\rangle)(\mathit{run}\ Y\,|\,a!\langle\mathbf{0}\rangle\,|\,a?(z)\ r!\langle z\rangle)$$

since a is allowed to have an input/output capability $(\mathtt{int})^{\mathtt{IO}}$ in the body. In other words the substitution of the name a for x in the type $\langle x:(\mathtt{int})^{\mathtt{I}}, a:(\mathtt{int})^{\mathtt{O}}\rangle$ should be $\langle a:(\mathtt{int})^{\mathtt{IO}}\rangle$. This is reflected in the final clause in Figure 4:

$$[\Delta]\{v/x\} = \sqcup[w\{v/x\}{:}\sigma\{v/x\}] \ \text{ with } \ w{:}\sigma \in \Delta$$

Here $\sqcup$ is an operator on types which intuitively acts like a (partial) least upper bound with respect to, yet to be defined, a subtyping order on types. The following are simple examples of $\sqcup$ on process types, which may be sufficient to read this paper; roughly speaking, $\sqcup$ calculates the union of the

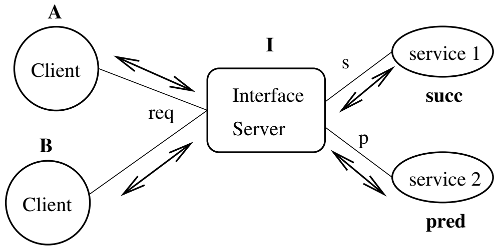

FIGURE 5. Interface Server and Distributed Services

accessibility rights of two processes. (The formal definitions are available in the Appendix in [36].)

$$[a\!:\!(\mathtt{int})^{\mathtt{I}}] \sqcup [b\!:\!(\mathtt{int})^{\mathtt{O}}] = [a\!:\!(\mathtt{int})^{\mathtt{I}}, b\!:\!(\mathtt{int})^{\mathtt{O}}] \quad \text{and}$$
$$[a\!:\!(\mathtt{int})^{\mathtt{I}}] \sqcup [a\!:\!(\mathtt{int})^{\mathtt{O}}] = [a\!:\!(\mathtt{int})^{\mathtt{IO}}]$$

Now we can analyse the substitution on types in the above example by the following equations.

$$[x\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}]\{b/x\}$$
$$= [x\{b/x\}:(\mathtt{int})^{\mathtt{I}}] \sqcup [a\!:\!(\mathtt{int})^{\mathtt{O}}] = [b\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}]$$
$$[x\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}]\{a/x\}$$
$$= [x\{a/x\}:(\mathtt{int})^{\mathtt{I}}] \sqcup [a\!:\!(\mathtt{int})^{\mathtt{O}}] = [a\!:\!(\mathtt{int})^{\mathtt{IO}}]$$

The properties of $\sqcup$ and $\sqcap$ will be discussed in the next section, when we consider well-typed programs. Note that in general $\sqcup$ is a partial operator and therefore apriori substitution is not always defined. However we will see that in properly typed environments it is always well-defined.

EXAMPLE 2.2. (Interface server and mobile client code) A (specific) compute service is a process which given some data and a return address, applies the specified operation to the data and returns the result to the address. A given name a, let $\mathbf{Succ}(a)$ represent the process $*a?(y,z)\, z!\langle \mathtt{succ}(y)\rangle$, which we write as

$$\mathbf{Succ}(a) \Longleftarrow\ *a?(y,z)\ z!\langle \mathtt{succ}(y)\rangle$$

This represents a service (for `succ`) situated at a. A similar one for the predecessor function, `pred` is defined as:

$$\mathbf{Pred}(a) \Longleftarrow\ *a?(y,z)\ z!\langle \mathtt{pred}(y)\rangle$$

A client may wish a number of operations to be performed on given data, perhaps in a particular sequence, with some data for later operations depending on results produced by earlier operations. This situation is given diagrammatically in Figure 5. Here there is an Interface (I) between clients and the collection of services or operations on offer. Now a higher-order *script* is sent to the interface. This script is executed locally by the interface, which interacts as necessary with the various services. This protocol puts the computational onus on the server and avoids repeated interactions between clients and services. The server may then be defined by:

$$\mathbf{Serv}_I(\mathrm{req}, s, p) \Longleftarrow\ *\mathrm{req}?(X)\ X\,s\,p$$

It takes in a script X, a process parameterised on service ports, and applies it to the actual port names of the two services, in this case s and p. Note that these actual names are not known to clients, thereby, in this case, affording some security protection to the server from clients; all interaction between clients and the server is through the interface name req.

We give two examples of clients requesting services. Client (A) wants to increment a number k twice, whereas the Client (B) wants to evaluate the successor and the predecessor of two different numbers n and m on parallel.

$$\mathbf{C}_A(\mathrm{req}) \Longleftarrow \mathrm{req}!\langle \lambda(s,p)\ ((\nu\, c) s!\langle k,c\rangle c?(z)\ s!\langle z,c\rangle \mathtt{Fw}(c r_A))\rangle$$
$$\mathbf{C}_B(\mathrm{req}) \Longleftarrow$$
$$\mathrm{req}!\langle \lambda(s,p)\ (\nu\, cc')(s!\langle n,c\rangle \mathtt{Fw}(c\ r_{1B}) \mid p!\langle m,c'\rangle \mathtt{Fw}(c'\ r_{2B}))\rangle$$

The forwarders in their bodies are used to relay the final results to each client on their result channels, r_A for client A and r_{1B}, r_{2B} for client B respectively.

Putting the clients and server together we have the following parallel composition (Figure 5):

$$\mathbf{C}_A(\mathrm{req}) \mid \mathbf{C}_B(\mathrm{req}) \mid \mathbf{Serv}_I(\mathrm{req}, s, p) \mid \mathbf{Succ}(s) \mid \mathbf{Pred}(p)$$

After certain amount of reductions, $k+2$ is returned to Client A on the channel r_A, and $n+1$ and $m-1$ are returned to r_{1B} and r_{2B}, respectively.

3 The Fine-Grained Typing System

3.1 Well-formed Types and Environments

In Figure 6 we present a formal system with three forms of judgements, all interrelated:

$\Gamma \ \vdash \mathtt{Env}$	Γ is a well-formed environment
$\Gamma \ \vdash \alpha : \mathtt{tp}$	α is a well-formed type in Γ
$\Gamma \ \vdash \alpha \leq \alpha'$	α is less than α' in Γ

For convenience we use $\Gamma \vdash \mathtt{J}$ as a shorthand for any of the three allowed forms of judgement. The first is designed to ensure that an identifier can only be used in the construction of a type if it has already been *declared* in the environment. For example one can not deduce

$$y : \langle x\!:\!(\mathtt{nat})^{\mathtt{O}}\rangle,\ x\!:\!(\mathtt{nat})^{\mathtt{O}} \vdash \mathtt{Env}$$

because the variable x is used in the type associated with y before being introduced. However if they are interchanged then this does constitute a valid environment:

$$x\!:\!(\mathtt{nat})^{\mathtt{O}},\ y : \langle x\!:\!(\mathtt{nat})^{\mathtt{O}}\rangle \vdash \mathtt{Env}$$

This emphasises the fact that our typing system will *not* have an interchange rule. In general being able to form a judgement of the form

$$\Gamma,\ x:\tau,\ y:\tau',\ \Gamma' \vdash \mathtt{J}$$

will not necessarily imply

$$\Gamma,\ y:\tau',\ x:\tau,\ \Gamma' \vdash \mathtt{J}$$

When constructing well-formed environments only types which are currently well-formed may be used. This is the

Well-formed Environment

$$\text{(e-nil)}\quad \emptyset \vdash \mathtt{Env} \qquad\qquad \text{(e-val)}\quad \frac{\Gamma \vdash \tau : \mathtt{tp} \quad u \notin \mathsf{dom}(\Gamma)}{\Gamma,\, u{:}\tau \vdash \mathtt{Env}}$$

Well-formed Types

$$\text{(t-base)}\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \top,\ \bot,\ \sigma_G,\ \mathtt{proc},\ [\ ] : \mathtt{tp}} \qquad \text{(t-sort)}\ \frac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash (\tau_1, \ldots, \tau_n) : \mathtt{tp}} \qquad \text{(t-chan)}\ \frac{\Gamma \vdash S_{\mathtt{I}} \geq S_{\mathtt{O}}}{\Gamma \vdash \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle : \mathtt{tp}}$$

$$\text{(t-abs}_H\text{)}\ \frac{\Gamma \vdash \sigma_H : \mathtt{tp}, \quad \rho : \mathtt{tp}}{\Gamma \vdash \sigma_H \to \rho : \mathtt{tp}} \qquad \text{(t-abs}_N\text{)}\ \frac{\Gamma, x{:}\sigma \vdash \rho : \mathtt{tp}}{\Gamma \vdash (x{:}\sigma) \to \rho : \mathtt{tp}} \qquad \text{(t-proc)}\ \frac{\forall u \in \mathsf{dom}(\Delta).\ \Gamma \vdash \Gamma(u) \leq \Delta(u)}{\Gamma \vdash [\Delta] : \mathtt{tp}}$$

Subtyping

$$\text{(s-id)}\ \frac{\Gamma \vdash \alpha : \mathtt{tp}}{\Gamma \vdash \alpha \leq \alpha} \qquad \text{(s-sort)}\ \frac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash \bot \leq (\tau_1, \ldots, \tau_n) \leq \top} \qquad \text{(s-base)}\ \frac{\Gamma \vdash [\Delta] : \mathtt{tp}}{\Gamma \vdash [\Delta] \leq \mathtt{proc}} \qquad \text{(s-chan)}\ \frac{\begin{array}{c}\Gamma \vdash S_{\mathtt{I}1} \leq S_{\mathtt{I}2},\ \ S_{\mathtt{O}2} \leq S_{\mathtt{O}1} \\ \Gamma \vdash \langle S_{\mathtt{I}i}, S_{\mathtt{O}i} \rangle : \mathtt{tp} \quad (i = 1, 2)\end{array}}{\Gamma \vdash \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle \leq \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle}$$

$$\text{(s-abs}_H\text{)}\ \frac{\Gamma \vdash \sigma'_H \leq \sigma_H, \quad \rho \leq \rho'}{\Gamma \vdash \sigma_H \to \rho \leq \sigma'_H \to \rho'} \qquad \text{(s-abs}_N\text{)}\ \frac{\Gamma \vdash \sigma_2 \leq \sigma_1 \quad \Gamma, x{:}\sigma_1 \vdash \rho_1 \leq \rho_2}{\Gamma \vdash (x{:}\sigma_1) \to \rho_1 \leq (x{:}\sigma_2) \to \rho_2} \qquad \text{(s-proc)}\ \frac{\begin{array}{c}\Gamma \vdash [\Delta_1] : \mathtt{tp} \\ \forall u \in \mathsf{dom}(\Delta_2).\ \Gamma \vdash \Delta_1(u) \leq \Delta_2(u)\end{array}}{\Gamma \vdash [\Delta_2] \leq [\Delta_1]}$$

FIGURE 6. Well-formed Types and Subtyping

purpose of the second form of judgement. So for example we can not deduce

$$\Gamma,\, y : \langle y{:}(\mathtt{nat})^{\mathtt{O}} \rangle \vdash \mathtt{Env}$$

To do so we would need to be able to deduce

$$\Gamma \vdash \langle y{:}(\mathtt{nat})^{\mathtt{O}} \rangle : \mathtt{tp}$$

This in turn is not possible, basically because y is not in the domain of Γ.

The rules for valid type formation are very straightforward; one is only constrained to use identifiers which are already declared in the current environment. In (t-chan), the condition $S_{\mathtt{I}} \geq S_{\mathtt{O}}$ is necessary to ensure that readers of a channel always receive at most the capabilities given by a sender. More details may be found in [15, 35]. There are only two novelties. In the formation rule for dependent types, (t-abs$_N$) the bound variable x is allowed to be used in the construction of the result type ρ. Secondly the rule (t-proc) ensures a process always has a type Δ which does not exceed the current environment Γ.

Subtyping also plays a role in the formation of environments. For example we can not deduce

$$a{:}(\mathtt{nat})^{\mathtt{O}},\ y : \langle a{:}(\mathtt{nat})^{\mathtt{IO}} \rangle \vdash \mathtt{Env}$$

because the capability associated with a when forming the type associated with y is not a subtype of that associated with a in the current environment. For no Γ can we deduce

$$\Gamma \vdash (\mathtt{nat})^{\mathtt{O}} \leq (\mathtt{nat})^{\mathtt{IO}}$$

The rules for subtyping are a straightforward extension of those given in [35, 26, 15], apart from the necessity to only use identifiers declared in the current environment. Both function types are contravariant in their first arguments and covariant in their second. Similarly in (s-chan) channel types are covariant in the input capability and contravariant in the output. Again the only real novelty is the sub-typing rule for process types, (s-proc); this means the the ordering of process types is *contravariant* w.r.t. the ordering of [35, 26].

A series of consistency lemmas about this system of judgements follows. They are invariably deduced by induction on the derivations in the standard manner. Remember informally $X, Y, Z, \ldots$ denote variables with higher-order types, as opposed to channel types.

LEMMA 3.1.

1. (Renaming) *Suppose* $u \notin \mathsf{fv}(\Gamma, v, \tau, \Gamma')$. Then we have $\Gamma,\, u{:}\tau,\, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\, v{:}\tau,\, \Gamma'\{v/u\} \vdash \mathtt{J}\{v/u\}$.
2. (Implied judgement) $\Gamma,\, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma \vdash \mathtt{Env}$ *and* $\Gamma,\, u{:}\tau,\, \Gamma' \vdash \mathtt{Env}$ *implies* $\Gamma \vdash \tau : \mathtt{tp}$.
3. (HO-bound change) $\Gamma,\, X : \sigma_H,\, \Gamma' \vdash \mathtt{J}$ *and* $\Gamma \vdash \sigma'_H : \mathtt{tp}$ *imply* $\Gamma,\, X : \sigma'_H,\, \Gamma' \vdash \mathtt{J}$.
4. (Weakening) *Assume* $\Gamma,\, x{:}\tau \vdash \mathtt{Env}$ *and* $u \notin \mathsf{dom}(\Gamma')$. *Then* $\Gamma,\, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\, u{:}\tau,\, \Gamma' \vdash \mathtt{J}$.
5. (Multiple weakening) *Assume* $\Gamma,\, \Gamma'' \vdash \mathtt{Env}$ *and* $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Gamma'') = \emptyset$. *Then* $\Gamma,\, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\, \Gamma'',\, \Gamma' \vdash \mathtt{J}$.
6. (Bound weakening) *Assume* $\Gamma \vdash \tau' \leq \tau$. *Then* $\Gamma, u{:}\tau, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, u{:}\tau', \Gamma' \vdash \mathtt{J}$.
7. (Implied judgement) $\Gamma \vdash \alpha \leq \alpha'$ *implies* $\Gamma \vdash \alpha{:}\mathtt{tp}$ *and* $\Gamma \vdash \alpha'{:}\mathtt{tp}$.
8. (HO narrowing) $\Gamma, X{:}\sigma_H, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma' \vdash \mathtt{J}$.
9. (Exchange) *Assume* $\Gamma, x' : \tau' \vdash \mathtt{Env}$. *Then* $\Gamma, u : \tau, u' : \tau', \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, u' {:} \tau', u {:} \tau, \Gamma' \vdash \mathtt{J}$. □

Note that in general we can not replace $X : \sigma_H$ with $u{:}\sigma$ in the statements (3) and (8) above since the channel u may appear freely in Γ and $\mathtt{J}$. This underlines the major technical difference between our system and the system $F_{<:}$, [7]; several lemmas for type variables in $F_{<:}$ [7], do not hold for channels in our system.

We now turn our attention to the partial meet and join operators, $\sqcap$, $\sqcup$ which play a crucial role in our definition of substitution,

DEFINITION 3.2. (FBC, cf. [15, 35]) We say that a partial order $(\mathbf{S}, \sqsubseteq)$ is *finite bounded complete* (FBC) for every finite nonempty subset $S \leq \mathbf{S}$, if S has a lower bound then S has a greatest lower bound. □

PROPOSITION 3.3. *Under an arbitrary well-formed environment, the subtyping relation over types is a partial order and finite bounded complete.* □.

The proof of this proposition, available in [36], is constructive; we give an inductive definition of the required partial meet operation $\sqcap$, which requires a simultaneous definition of a partial join operation $\sqcup$, already discussed above.

The next Lemma will be important in the proof of Subject Reduction. It shows that, under certain circumstances, a variable can be replaced by an identifier throughout a judgement, although this replacement may also change the environment of the judgement.

LEMMA 3.4. (Name substitution) *Suppose* $\Gamma \vdash \Gamma(u) \leq \sigma$. *Then* $\Gamma, x\!:\!\sigma, \Gamma' \vdash J$ *implies* $\Gamma, \Gamma'\{u/x\} \vdash J\{u/x\}$. □.

Finally we conclude this section with the following property, which relates to the construction of an efficient algorithm of the typing system (cf. [26, 35]).

PROPOSITION 3.5. (decidability) $\Gamma \vdash J$ *is decidable. Moreover, if* $\Gamma \vdash \alpha_i : \mathtt{tp}$ $(i = 1,2)$, *then* $\Gamma \vdash \alpha_1 \sqcup \alpha_2 : \mathtt{tp}$ *and* $\Gamma \vdash \alpha_1 \sqcap \alpha_2 : \mathtt{tp}$ *are decidable.* □.

3.2 Type Inference

The typing system is given in Figure 7. The judgements are of the form:

$$\Gamma \vdash P\!:\!\alpha \qquad \text{a term } P \text{ has a type } \alpha \text{ under } \Gamma$$

For convenience the inference rules in Figure 7 are divided into three groups. The first, **(Common)**, are elementary, although the subsumption rules (SUB$_H$) and (SUB$_N$) will play a major role in type inferences. The second, **(Function)**, are inherited from typing systems for the polymorphic λ-calculus. Here we have two forms of functional types, each with its introduction and elimination rules. The novelty occurs with abstraction over channel variables. Intuitively if a term P has a type ρ, then a channel abstraction $\lambda(x\!:\!\sigma)P$ is a function which becomes $P\{a/x\}$ when it is applied to a name a with a type σ. Therefore, we will bind free occurrences of x in ρ in the abstraction rule (ABS$_N$). The corresponding elimination (APP$_N$) allows dynamic channel instantiation into types during β-reduction. If a term P has a type $(x\!:\!\sigma) \to \rho$, we can apply a name a whose type is less than σ to P. Then a is substituted for x in ρ in (APP$_N$).

The final group, **(Process)**, are based on the IO-Typing systems from [26, 15, 35]. However many of the rules are sufficient novel to warrant detailed explanation.

The *empty* rule, (NIL)**:** A process type $[\Delta]$ of represents an upper bound on the interface or interaction points of a process. Since an empty process $\mathbf{0}$ has no interaction point, under any environment Γ it is typed as:

$$\Gamma \vdash \mathbf{0} : [\,]$$

The *parallel* rule, (PAR): To infer $\Gamma \vdash P_1 \,|\, P_2 : \pi$ it is sufficient to infer $\Gamma \vdash P_i : \pi$ for each of the individual processes. However, in the presence of subsumption, there is a much more informative derived version of this rule, which will be frequently used:

$$\frac{\Gamma \vdash P_1 : \pi_1,\ \ \Gamma \vdash P_1 : \pi_1}{\Gamma \vdash P_1 \,|\, P_2 : \pi_1 \sqcup \pi_2}$$

A meta-result about our typing system ensures that, relative to a particular environment, the join of two process types always exists.

The *output* rule, (OUT): Under what circumstances can we conclude $\Gamma \vdash a!\langle V\rangle P : \pi$? We require at least the following:

- The residual P should have the required type, $\Gamma \vdash P : \pi$
- The value V should have a type appropriate to the channel a. That is, there should be some value type τ such that $\Gamma \vdash V : \tau$ and
- the channel a should have the output capability at the type τ. However this capability on a should be available from the overall interface of the process, π. This can be represented by the judgement $\Gamma \vdash [a\!:\!(\tau)^{\mathtt{O}}] \leq \pi$.

However there may be a further requirement. If the value being output is actually a channel, say b with a type $\tau = \sigma$, then the capability being exported must also be available from the process interface $\Gamma \vdash [b\!:\!\sigma] \leq \pi$.

The general statement of the rule, for multiple output values, is given in Figure 7; it uses the notation

$$\pi \vdash_{\Gamma} u : \sigma$$

to mean that, relative to the environment Γ the interface, or process type, π can provide at least the capability σ at u, that is $\Gamma \vdash [u\!:\!\sigma] \leq \pi$.

As an example let Δ_{ab} be the environment which maps b to the type $(\mathtt{int})^{\mathtt{O}}$ and a to the type $\langle\Delta_b\rangle^{\mathtt{IO}}$, allowing it to transmit thunked values of type Δ_b; this is the process type which maps b to the same type $(\mathtt{int})^{\mathtt{O}}$. Then with the output rule, together with (NIL), we can establish

$$\Delta_{ab} \vdash b!\langle 1\rangle\,\mathbf{0} : [\Delta_b]$$

and therefore

$$\Delta_{ab} \vdash a!\langle b!\langle 1\rangle\,\mathbf{0}\rangle\,\mathbf{0} : [a\!:\!\langle\Delta_b\rangle^{\mathtt{O}}]$$

$$
\begin{array}{llll}
\textbf{(Common)} & (\textsc{Val})\ \dfrac{\vdash \Gamma, u\!:\!\tau, \Gamma' : \mathtt{Env}}{\Gamma, u\!:\!\tau, \Gamma' \vdash u : \tau} & & (\textsc{Con})\ \dfrac{\vdash \Gamma : \mathtt{Env}}{\Gamma \vdash 1 : \mathtt{nat}}\quad \text{etc.} \\[2ex]
 & (\textsc{Sub}_H)\ \dfrac{\Gamma \vdash P : \rho \quad \Gamma \vdash \rho \leq \rho'}{\Gamma \vdash P : \rho'} & & (\textsc{Sub}_N)\ \dfrac{\Gamma \vdash u : \sigma \quad \Gamma \vdash \sigma \leq \sigma'}{\Gamma \vdash u : \sigma'} \\[3ex]
\textbf{(Function)} & (\textsc{Abs}_H)\ \dfrac{\Gamma, X\!:\!\sigma_H \vdash P : \rho}{\Gamma \vdash \lambda(X\!:\!\sigma_H)P : \sigma_H \to \rho} & & (\textsc{App}_H)\ \dfrac{\Gamma \vdash P : \sigma_H \to \rho \quad \Gamma \vdash Q : \sigma_H}{\Gamma \vdash PQ : \rho} \\[2ex]
 & (\textsc{Abs}_N)\ \dfrac{\Gamma, x\!:\!\sigma \vdash P : \rho}{\Gamma \vdash \lambda(x\!:\!\sigma)P : (x\!:\!\sigma) \to \rho} & & (\textsc{App}_N)\ \dfrac{\Gamma \vdash P : (x\!:\!\sigma) \to \rho \quad \Gamma \vdash u : \sigma}{\Gamma \vdash Pu : \rho\{u/x\}} \\[3ex]
\textbf{(Process)} & (\textsc{Nil})\ \dfrac{\vdash \Gamma : \mathtt{Env}}{\Gamma \vdash \mathbf{0} : [\ ]} & (\textsc{Par})\ \dfrac{\Gamma \vdash P_{1,2} : \pi}{\Gamma \vdash P_1 \,|\, P_2 : \pi} & (\textsc{Rep})\ \dfrac{\Gamma \vdash P : \pi}{\Gamma \vdash *P : \pi} \quad (\textsc{Res})\ \dfrac{\Gamma, a\!:\!\sigma \vdash P : \pi}{\Gamma \vdash (\nu a\!:\!\sigma)P : \pi/a} \\[3ex]
 & (\textsc{Out})\ \dfrac{\begin{array}{l}\pi \vdash_\Gamma u : (\tau_1, \ldots, \tau_n)^{\mathtt{O}} \qquad \Gamma \vdash P : \pi \\ \Gamma \vdash V_i : \tau_i \quad \tau_i = \sigma_i \Rightarrow \pi \vdash_\Gamma V_i : \sigma_i\end{array}}{\Gamma \vdash u!\langle V_1, \ldots, V_n \rangle P : \pi} & & (\textsc{In})\ \dfrac{\begin{array}{l}\pi \vdash_\Gamma u : (\tau_1, \ldots, \tau_n)^{\mathtt{I}} \\ \Gamma, x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n \vdash P : \pi, x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n\end{array}}{\Gamma \vdash u?(x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n)P : \pi}
\end{array}
$$

FIGURE 7. Typing System for $\lambda\pi_{\mathrm{v}}$

The *input* rule, (IN)**:** The rule for prefixing is a straightforward generalisation of that in [35]:

$$\frac{\pi \vdash_\Gamma u : (\tau)^{\mathtt{I}} \qquad \Gamma, x\!:\!\tau \vdash P : \pi, x\!:\!\tau}{\Gamma \vdash u?(x\!:\!\tau)\, P : \pi}$$

To deduce that the process $a?(x\!:\!\tau)\ P$ has the interface π we need to establish two facts:

- The interface π can provide the correct capability for the channel u; that is $\pi \vdash_\Gamma u : (\tau)^{\mathtt{I}}$.
- The residual P, having input a value for the variable x, has the augmented interface $\pi, x\!:\!\tau$; however this can be established in the environment Γ augmented by x; that is $\Gamma, x\!:\!\tau \vdash P : \pi, x\!:\!\tau$.

Here we are using a notation defined by the following rules:

$$
\begin{array}{rcll}
\pi, x\!:\!\sigma & \stackrel{\text{def}}{=} & \pi \sqcup [x\!:\!\sigma] & \text{and} \\
\pi, x\!:\!\sigma_H & \stackrel{\text{def}}{=} & \pi &
\end{array}
$$

Note first in the above definition "$\pi, x : \tau$" denotes "π" if τ is not a channel type. In (IN), by the first sequence in the antecedent and (Implied judgement), Lemma 3.1, we know π is well-formed under Γ. Hence automatically x does not occur in π. From this, if π takes the form $[\Delta]$ for some Δ, then $x \notin \mathsf{fv}([\Delta])$, and P has a type $[\Delta], x\!:\!\sigma \stackrel{\text{def}}{=} [\Delta, x\!:\!\sigma]$ in the second assumption.

As an example let Δ_c be the environment which maps c to the capability $(\mathtt{int}^{\mathtt{O}})^{\mathtt{IO}}$. Then we one can easily check that

$$\Delta_c \vdash c?(z : (\mathtt{int})^{\mathtt{O}})\, z!\langle 1 \rangle : [\Delta_c]$$

It may seem strange that this process has been typed to have at most a capability on the channel c; obviously when it receives an input on c it will immediately gain some other capability. But this input will be sent by some other process, in the presence of which the interface will be increased appropriately.

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\top/a = \top,\ \ \bot/a = \bot,\ \sigma_G/a = \sigma_G,\ \ \mathtt{proc}/a = \mathtt{proc}} \\
\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle/a & = & \langle S_{\mathtt{I}}/a, S_{\mathtt{O}}/a \rangle \\
(\tau_1, \ldots, \tau_n)/a & = & (\tau_1/a, \ldots, \tau_n/a) \\
(\sigma_H \to \rho)/a & = & \sigma_H/a \to \rho/a \\
((x\!:\!\sigma) \to \rho)/a & = & (x\!:\!\sigma/a) \to \rho/a \\
[\Delta]/a & = & [\{u\!:\!(\sigma/a) \mid u\!:\!\sigma \in \Delta \wedge u \neq a\}]
\end{array}
$$

FIGURE 8. Name Erasing from Types

For example let Δ_{cd} be the extension of Δ_c which maps d to the output capability, $(\mathtt{int})^{\mathtt{O}}$. Then we have

$$\Delta_{cd} \vdash c!\langle d \rangle : [\Delta_{cd}]$$

Now we can use the rule (PAR), or rather its derived variant, (together with a version of Multiple Weakening) to deduce

$$\Delta_{cd} \vdash (c?(z : (\mathtt{int})^{\mathtt{O}})\ z!\langle 1 \rangle \ \mid c!\langle d \rangle) : [\Delta_{cd}]$$

The *restriction* rule, (RES)**:** The restriction operator $(\nu a)-$ reduces the interface of a process. For example in an appropriate environment the process $a!\langle 1 \rangle$ can be assigned the process type, or interface, $[a : (\mathtt{nat})^{\mathtt{O}}]$. When we restrict the channel a, to obtain the process $(\nu a)a!\langle 1 \rangle$, all a capabilities will be removed from the interface; the restricted process has the empty interface $[\,]$. The general rule is formulated as (RES) where π/a denotes the result of erasing all occurrences of a from π. This erasure operator on types is defined formally in Figure 8. For example, $[a : (\mathtt{nat})^{\mathtt{O}}]/a = [\ ]$, and $[b : \langle a : (\mathtt{nat})^{\mathtt{O}} \rangle^{\mathtt{O}}]/a = [b : \langle\ \rangle^{\mathtt{O}}]$. Hence, in appropriate environments, $(\nu a)a!\langle 1 \rangle$ has a type $[\ ]$ and $(\nu a)b!\langle a!\langle 1 \rangle \rangle$ has a type $[b : \langle\ \rangle^{\mathtt{O}}]$.

EXAMPLE 3.6. (Interface server and mobile client code) We revisit Example 2.2. First Sucsessor service is annotated as:

$$\mathbf{Succ}(a) \Longleftarrow\ *\,a?(y{:}\mathtt{int}, z{:}(\mathtt{int})^{\mathtt{O}})\ z!\langle \mathtt{succ}(y)\rangle$$

where types ensure that this process only receives the output capability on the return channel z. Let Δ_r, be an environment defining these return channels; in this case it maps r_A, r_{1B} and r_{2B} to the same type $(\mathtt{int})^{\mathtt{O}}$. Let σ_s be a type $(\mathtt{int}, (\mathtt{int})^{\mathtt{O}})^{\mathtt{O}}$ and τ_{sc} be a type for scripts:

$$(s{:}\sigma_s) \to (p{:}\sigma_s) \to [\,s{:}\sigma_s, p{:}\sigma_s, \Delta_r\,]$$

So these are abstractions which, when applied to appropriate names, generate processes which can at most use those names for output together with the return channels, also for output only. Using subsumption we can form the judgement

$$s{:}\sigma_s, p{:}\sigma_s, \Delta_r \vdash P_A : [\,s{:}\sigma_s, p{:}\sigma_s, \Delta_r\,]$$

where P_A denotes the body of the script sent by $\mathbf{C}_A$, namely $(\nu c)s!\langle k,c\rangle c?(z)\ s!\langle z,c\rangle \mathtt{Fw}(cr_A)$. By channel abstraction we therefore have

$$\Delta_r \vdash \lambda(s{:}\sigma_s, p{:}\sigma_s).P_A : \tau_{\mathrm{sc}}$$

That is the value sent by the client is indeed typed as a script.

Now let Δ_{cl} denote the environment $\{\mathrm{req} : (\tau_{\mathrm{sc}})^{\mathtt{O}},\ \Delta_r\}$. Then we can form the judgement

$$\Delta_{\mathrm{cl}} \vdash \mathbf{C}_A(\mathrm{req}) : [\Delta_{\mathrm{cl}}]$$

and a similar judgement can be made for $\mathbf{C}_B$.

This judgement gives detailed information about the resources known to the clients. For example it says that the clients do not need to know the locations of the actual interfaces of the various services; indeed it only needs to know that of the server, req, together with the return channels.

Typing the server is slightly different. Here we need to let Δ_{serv} be

$$\{\mathrm{req} : (\tau_{\mathrm{sc}})^{\mathtt{I}},\ \Delta_r, s : \sigma_s,\ p : \sigma_s\}.$$

Then the reader can check that

$$\Delta_{\mathrm{serv}} \vdash \mathbf{Serv}_I(\mathrm{req}, s, p) : [\Delta_{\mathrm{serv}}]$$

Thus the server requires knowledge of the locations of the service points, but needs only to be able to send data to them. It also only send capabilities on the return channels. Note also that if we only have a constant process type `proc`, as in the previous typing system of the process calculi [3, 35, 26, 11], then the interface server could input any function $\lambda s.\lambda p.Q$, where Q an arbitrary process via "req"; such incoming code may harm local resources. □

4 Type Soundness

4.1 Subject Reduction

The results in Lemma 3.1 have natural generalisations to our typing system, $\Gamma \vdash P : \alpha$. The details are available in [36], but here we show one instance, Channel Narrowing.

LEMMA 4.1. *Assume* $a \notin \mathsf{fn}(P)$. *Then* $\Gamma, a{:}\sigma, \Gamma' \vdash J : \alpha$ *implies* $\Gamma, \Gamma'/a \vdash J : \alpha/a$, *and* $\Gamma, a{:}\sigma, \Gamma' \vdash P : \alpha$ *implies* $\Gamma, \Gamma'/a \vdash P : \alpha/a$. □

$$\begin{array}{rl} a?(x_1{:}\tau_1,\ldots,x_n{:}\tau_n)P \xrightarrow{\Gamma,\pi}_{err} & \text{if } \Gamma \not\vdash [\,a : (\tau_1,\ldots,\tau_n)^{\mathtt{I}}\,] \leq \pi. \\ a!\langle V_1,\ldots,V_n\rangle P \xrightarrow{\Gamma,\pi}_{err} & \text{if no } \tau_i \text{ s.t. } \Gamma \vdash V_i : \tau_i \\ & \text{and } \Gamma \vdash [\,a{:}(\tau_1,\ldots,\tau_n)^{\mathtt{O}}\,] \leq \pi. \end{array}$$

$$\frac{P \xrightarrow{(\Gamma,a:\sigma),\pi}_{err}}{(\nu\, a{:}\sigma)P \xrightarrow{\Gamma,(\pi/a)}_{err}} \qquad \frac{P \xrightarrow{\Gamma,\pi} \text{ or } Q \xrightarrow{\Gamma,\pi}}{P\,|\,Q \xrightarrow{\Gamma,\pi}_{err}} \qquad \frac{P \xrightarrow{\Gamma,\pi}_{err}}{*\,P \xrightarrow{\Gamma,\pi}_{err}}$$

FIGURE 9. Run-time errors

The following result which states, informally, that well-typedness is preserved by substitution of appropriate values for variables, is the key result underlying Subject Reduction. This also gurantees that the substitution given by $\sqcup$ is always defined if P is well-typed.

LEMMA 4.2. (Substitution Lemma) *Assume* $\Gamma \vdash V{:}\tau$. *Then* $\Gamma, x{:}\tau, \Gamma' \vdash P : \alpha$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash P\{V/x\} : \alpha\{V/x\}$.

THEOREM 4.3. **(Subject Reduction)**

- $\Gamma \vdash P{:}\pi$ *and* $P \equiv P'$ *imply* $\Gamma \vdash P'{:}\pi$.
- *If* $\Gamma \vdash P : \rho$ *and* $P \longrightarrow P'$, *then* $\Gamma \vdash P' : \rho$. □.

4.2 Type Safety

Our typing system is an extension of that for the λ-calculus from [12] and that for the π-calculus from [26]; consequently it guarantees the absence of the typical run-time errors associated with these languages.

Intuitively $\Gamma \vdash P : \pi$ should mean that, assuming the environment Γ, the process P satisfies the *interface* π. If π is the undifferentiated type `proc` then, viewed as an interface, it provides no information. However if it has the form $[\Delta]$ this means that P can use *at most* the resources mentioned in Δ; moreover these resources can only be used according to the capabilities they are assigned in Δ. A simple formalisation of this intuitive idea is given in Figure 9, using a unary predicate $P \xrightarrow{\Gamma,\pi}_{err}$. The first two clauses are the most significant. The first says that, relative to Γ, P violates the interface π if it can input on the channel a but the interface π does not assign any input capability to a; the second is similar, but for output. Combining these rules, we can also derive the following communication runtime error between input and output:

$$a?(x_1{:}\tau_1,\ldots,x_n{:}\tau_n)P \mid a!\langle V_1,\ldots,V_n\rangle Q \xrightarrow{\Gamma,\pi}_{err}$$

if no τ_i' such that $\Gamma \vdash \tau_i' \leq \tau_i$ and $\Gamma \vdash V_i : \tau_i'$. The meaning of the above error is easily understood when we consider the following example:

$$a?(x{:}\langle\Delta\rangle)\,P\,|\,a!\langle R\rangle\ Q \xrightarrow{\Gamma,\pi}_{err} \quad \text{if } \Gamma \not\vdash R : [\Delta].$$

This says if the input process gets the process R which does not conform the interface "Δ", then a runtime error occurs.

THEOREM 4.4. **(Type Safety)** *If* $\Gamma \vdash P : \pi$ *then* $P \not\xrightarrow{\Gamma,\pi}_{err}$ $\square$.

5 Typed Behavioural Semantics

Types constrain the behaviour of processes and their environments and consequently have an impact on when their behaviour should be deemed to be equivalent. Typed behavioural equivalences have already been investigated for various process calculi in papers such as [20, 26, 27, 34]. We contend that the existence of the fine-grained process types facilitates the development of typed behavioural theories; more importantly they enable us to state and prove very general theorems about these equivalences which are extremely useful for establishing identities. Due to space limitations in this section we merely outline our results in this area.

A family of relations $\mathbf{R}$ over process terms, ranged over by closed type environments and process types, is said to be a *typed relation* if $P_1\, \mathbf{R}^{\Gamma}_{\pi}\, P_2$, implies $\Gamma \vdash P_{1,2} : \pi$. Note that the relation is parameterised by not only environment Γ but also interface π. The following definition uses the notation from [17, 26, 34, 3].

DEFINITION 5.1. Let $\sim^{\Gamma}_{\pi}$ denote the largest typed relation which:

- is a (typed) congruence
- is closed under (strong) reductions
- satisfies $P_1\ \mathbf{R}^{\Gamma}_{\pi}\, P_2$ implies P_i have the same (strong) input/output potentials.

Let $\approx^{\Gamma}_{\pi}$ denote the corresponding *weak* relation. $\square$

We can establish expected identities such as:

PROPOSITION 5.2. (1) (Garbage collection) $P \approx^{\Gamma}_{[\,]} \mathbf{0}$.
(2) (Beta-reduction, c.f. [3], p.10) $(\lambda x.P)V \approx^{\Gamma}_{\pi} P\{V/X\}$. $\square$

Note (1) is a more general garbage collection law of P than the standard one defined with the condition $\mathsf{fn}(P) = \emptyset$; for example, we can statically check $(\nu\, a)(a?(X)\ \mathbf{0} \,|\, a!\langle b?(Y)\ P\rangle)$ is equivalent to $\mathbf{0}$. Next we show our fine-grained types also allow us to give a simple, and clean, definition of *triggers* which have proven to be very important in the theory of the higher-order processes, as well as π-calculus [21, 26, 29].

DEFINITION 5.3. We say a is (only) used as a *trigger* in P under Γ if there exists Δ such that $\Gamma \vdash P : [\Delta]$, and if $a{:}\sigma \in \Delta$, then $\sigma = S^0$. $\square$

Note that this definition is considerably simpler than that given in [26] (c.f. Definition 5.3.1).

PROPOSITION 5.4. (Higher-order replication) *Assume a is only used as the trigger in P, Q and R under* Γ. *Then:*

$$\begin{aligned}&(a{:}\sigma)(P \,|\, Q \,|\, *a?(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).R)\\ &\quad \sim^{\Gamma}_{\pi} (a{:}\sigma)(P \,|\, *a?(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).R)\\ &\qquad\qquad |\,(a{:}\sigma)(Q \,|\, *a?(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).R) \qquad \square\end{aligned}$$

Again this is a non-trivial generalisation of the corresponding result in in [29], Theorem 4.3.3; there "a" is only allowed to appear syntactically as output subjects. The following forwarder equality is one useful extension from [21, 9] to the higher-order setting.

PROPOSITION 5.5. (Forwarder) *Assume a is used as a trigger in P, b is different from a and* `Fw` *is the forwarder defined in Section 1. Then:* $(a{:}\sigma)(P \,|\, \mathtt{Fw}(ab)) \approx^{\Gamma}_{\pi} P\{b/a\}$. $\square$.

Finally we illustrate the usefulness of these general identities by applying them to the analysis of the system discussed in Example 3.6. Let **Sys** denote:

$$(\nu\, \mathrm{req}, s, p)(\mathbf{C}_A(\mathrm{req}) \,|\, \mathbf{C}_B(\mathrm{req}) \,|\, \mathbf{Serv}_I(\mathrm{req}, s, p) \,|\, \mathbf{Succ}(s) \,|\, \mathbf{Pred}(p))$$

Intuitively in this closed system the two clients make three specific requests of the distributed server. So we have:

$$\begin{aligned}\mathbf{Sys} \sim^{\Gamma}_{\pi}\ & (\nu\, \mathrm{req}, s, p)(\mathbf{C}_A(\mathrm{req}) \,|\, \mathbf{Serv}_I(\mathrm{req}, s, p) \,|\, \mathbf{Succ}(s) \,|\, \mathbf{Pred}(p))\\ & |\,(\nu\, \mathrm{req}, s, p)(\mathbf{C}_B(\mathrm{req}) \,|\, \mathbf{Serv}_I(\mathrm{req}, s, p) \,|\, \mathbf{Succ}(s) \,|\, \mathbf{Pred}(p))\\ \approx^{\Gamma}_{\pi}\ & r_A!\langle k+2\rangle \,|\, r_{B1}!\langle n+1\rangle \,|\, r_{B2}!\langle m-1\rangle\end{aligned}$$

The first equation is an instance of our Replication theorem (since by τ_{sc} which annotates variable X in $\mathbf{Serv}_I(\mathrm{req}, s, p)$, we can check s and p are used as a trigger in the server). Note that s and p appear as values, hence they may be used both as output subjects and objects in the interface server; therefore our generalised version of this theorem is required. The second equation requires Proposition 5.2.

Further investigation of our typed equivalences is an interesting research topic, particularly in its application to the refinement of the context equality of [29].

6 Conclusion

This Section concludes the paper with extensions, further issues and related work. For more details, see [36].

Distributed Higher-order π-calculus Type Safety Theorem means that our typing system can be used to ensure various kinds of *host security*; that is, protecting hosts from untrusted imported code. In the full version of this paper, [36], we discuss this issue more explicitly, by extending our typing system to the distributed version of $\lambda\pi_{\mathrm{v}}$, given in [35]. Once more the expressiveness of our fine-grained types means that, for example, channel locality, [4], can easily be enforced; specifically there is no requirement to annotate higher-order values as being *sendable*. A similar problem appears with type systems for the λ-calculus involving arrow and reference types, [24]. We hope that an extension of our scheme to higher-order functional shared variables (i.e. passing environments as `com` types) will also be useful in this setting.

More type constructors One beneficial point of our typing system is that it is relatively straightforward to extend our set of types with many of the standard constructs from the literature for both the λ-calculus and the π-calculus; these include recursive types [5, 32, 26], record types [12, 31, 10], polymorphic types [7, 27], and dynamic types [2, 28]. A particularly useful extension would be *linear/affine types*, as in the π-calculus and the linear λ-calculus [20, 6, 16, 34]. This would allow, for example, host sites to further control the access to resources by mobile code, by restricting the number of times a channel may be used. An extension of our capability based typing systems to more advanced distributed primitives, especially to constructs involving security [1, 14] would be more challenging.

Type limitations One limitation of our typing system is that, while name variables in types can be abstracted by channel dependency types $(x{:}\sigma) \to \rho$ of the channel λ-abstraction $\lambda(x{:}\sigma)\,P$, a similar abstraction is not allowed when we bound name variables by input prefix $a?(x{:}\sigma)\;P$. The result is that there is a loss of information in many of the types we can assign to processes. An example is the process $a?(x)\;b!\langle x!\langle v\rangle\rangle$. In the current system this is assigned a process type in which b has the output capability of the type $\langle \mathtt{proc}\rangle$. There is a similar loss of information in typing restricted processes, $(\nu\,a)P$. For example the process $(\nu\,a)b!\langle a!\langle 1\rangle\rangle$ can be assigned, in an appropriate environment, the type $[b : \langle\;\rangle^{\mathtt{O}}]$ which intuitively says that b can output a (thunked) process which has the empty interface. Here we may want to say that b can output a (thunked) process which contains some *unknown* channel name of type $(\mathtt{nat})^{\mathtt{O}}$, and the input type associated with b should be able to accommodate such constraints. Some form of existential quantification over types may be appropriate but integrating such a construct into the type language is also a non-trivial task.

Related work We have already made reference to the extensive literature on typing for the π-calculus and related processes. In developing our fine-grained type system we have been guided by the polymorphic λ-calculus [12, 7], where type variables play an important role; as with our channel names they may appear, and be bound, both in terms and types. However there is an essential technical difference between our use of channels and the use of type variables in the polymorphic λ-calculus. Name instantiation can result in dynamic changes to the types annotating a term. Names are exchanged as values between processes but they also appear as interaction points in the types of processes. An essential difference is that their type variables are instantiated by types (say `int`) whereas in our case channel variables occurring in types are instantiated by *channels*, not by types. This feature necessitated the development of new concepts of well-formed type, subtyping, well-formed substitution, etc., independent of those developed for the polymorphic λ-calculus.

Pierce and Sangiorgi [27] recently proposed a polymorphic π-calculus with an existential quantifier over types and used a refined typed behavioural equivalence to reason about concurrent abstract data types. Since their polymorphic types are based on those of the polymorphic λ-calculus (that is they abstract over *type variables* via the operator ∃), they are quite different from ours. In particular they do not address the issue of assigning fine-grained types to processes.

For sequential computations, Tofte and Talpin have developed the effect typing system [30] and this was recently applied to Facile by Kirli [18]. Again her typing system is different from ours since she adds the original effect system to the arrow types; hence all processes have the unique constant type *Unit* and a channel cannot carry nested effects. More precisely, the region types in [30] are used to represent the region allocation of values during β-reduction, while our process types are used to represent interaction effects between concurrent processes. Hence an integration of the effect typing system of the λ-calculus and the IO-subtyping system of the π-calculus would have difficulty in expressing the kind of constraints guaranteed by our typing system.

De Nicola, Ferrari and Pugliese studied a subtyping system for a language based on Linda [8], and showed that it can be used to control the mobility of mobile agents. In their language, each located process is equipped with different capabilities (read, input, our, eval and newloc) rather than the unique process type, which is similar to our framework. However their calculus is based on CCS rather than the π-calculus and our form of process types based on IO-subtyping and λ-subtyping are not considered in their formulation.

References

[1] Abadi, M., Secrecy by Typing in Security Protocols, *TACS'97*, LNCS 1281, pp.611-638, Springer-Verlag, 1997.

[2] Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G., Dynamic Typing in a Statically Typed Language. *TOPLAS*, 13(2), pp.237-268, 1991.

[3] Amadio, R., Translating Core Facile, ECRC Research Report 944-3, 1994.

[4] Amadio, R., Boudol, G. and Lhoussain, C., The receptive distributed π-calculus, *FSTTCS'99*, LNCS, Springer-Verlag, 1999.

[5] Amadio, R. and Cardelli, L., Subtyping Recursive Types, *TOPLAS*, 15(4), pp.575–631, 1993.

[6] Cardelli, L. and Gordon, A., Typed Mobile Ambients, *POPL'99*, pp.79–92, ACM Press, 1999.

[7] Cardelli, L., Martini, S., Mitchell, J. and Scedrov, A., An extension of system F with subtyping, *Info. & Comp.*, 109(1):4-56, February 1994.

[8] De Nicola, R., Ferrari, G. and Pugliese, R., Klaim: a Kernel Language for Agents Interaction and Mobility, IEEE Trans. on Software Engineering, Vol.24(5), 1998.

[9] Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., and Rémy, D., A Calculus for Mobile Agents, *CONCUR'96*, LNCS 1119, pp.406–421, Springer-Verlag, 1996.

[10] Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions. *ETAPS'99*, LNCS, Springer, 1999.

[11] Giacalone, A., Mistra, P. and Prasad, S., Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming, *ICALP'90*, LNCS 443, pp.765–780, Springer-Verlag, 1990.

[12] Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.

[13] Sun Microsystems Inc., Java home page. http://www.javasoft.com/.

[14] Heintze, N. and Riecke, J., The SLam Calculus: Programming with Secrecy and Integrity, *POPL'98*, pp.365-377. ACM Press, 1998.

[15] Hennessy, M. and Riely, J., Resource Access Control in Systems of Mobile Agents, CS Report 02/98, University of Sussex, http://www.cogs.susx.ac.uk, 1998.

[16] Honda, K., Composing Processes, *POPL'96*, pp.344-357, ACM, 1996.

[17] Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *TCS*, pp.437–486, No.151, North-Holland, 1995.

[18] Kirli, D., A static type system for detecting potentially transmissible functions, ECOOP Workshop MOS'99, Available from: http://cuiwww.unige.ch/˜ecoopws, 1999.

[19] Knabe, F.C., Language Support for Mobile Agents, PhD Thesis, Carnegie Mellon University, Dec. 1995.

[20] Kobayashi, N., Pierce, B. and Turner, D., Linearity and the pi-calculus, *POPL'96*, ACM Press, 1996.

[21] Merro, M. and Sangiorgi, D., On asynchrony in name-passing calculi, *ICALP'98*, LNCS 1443, pp.856–867, Springer-Verlag, 1998.

[22] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Infor. & Comp.*, 100(1), pp.1–77, 1992.

[23] Necula, G., Proof-carrying code. *POPL'96*, ACM, 1996.

[24] O'Hearn, P., Power, J., Takeyama, M., and Tennent, D., Syntactic Control of Interference Revised, *MFPS'97*, *ENCS*, Elsevier, 1997.

[25] ObjectSpace Inc. ObjectSpace Voyager home page, http://www.objectspace.com/voyager.

[26] Pierce, B.C. and Sangiorgi. D, Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.

[27] Pierce, B.C. and Sangiorgi. D, Behavioral Equivalence in the Polymorphic Pi-calculus. *POPL'97*, ACM Press, 1997.

[28] Riely, J. and Hennessy, M., Trust and partial typing in open systems of mobile agents. *POPL'99*, ACM Press, 1999.

[29] Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, University of Edinburgh, CST-99-93, 1993.

[30] Tofte, M. and Talpin, J.-P., Region-based memory management, *Info. & Comp.*, 132(2)109–176, 1997.

[31] Vasconcelos, V., Typed concurrent objects. *ECOOP'94*, LNCS 821, pp.100–117. Springer-Verlag, 1994.

[32] Vasconcelos, V. and Honda, K., Principal Typing Scheme for Polyadic π-Calculus. *CONCUR'93*, LNCS 715, pp.524-538, Springer, 1993.

[33] Vasconcelos, V., Lopes, L. and Silva, F., Distribution and Mobility with Lexical Scoping in Process Calculi, 3rd *HLCL*, *ENTCS* 16(3), Elsevier, 1998.

[34] Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, pp. 371–386, Springer-Verlag, 1996.

[35] Yoshida, N. and Hennessy, M., Subtyping and Locality in Distributed Higher Order Processes. *CONCUR'99*, LNCS 1664, pp.557–573, Springer-Verlag, 1999. CS Technical Report 01/99, University of Sussex, Available at: http://www.cogs.susx.ac.uk, April, 1999.

[36] Yoshida, N. and Hennessy, M., The full version of this paper, CS Technical Report 02/99, University of Sussex, Available at: http://www.cogs.susx.ac.uk and http://www.mcs.le.ac.uk/˜nyoshida, Nov, 1999.