Assigning Types to Processes

Nobuko Yoshida

Department of Mathematics and Computer Science, University of Leicester, Leicester, LE1 7RH, England.

and

Matthew Hennessy

Computer Science, School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, England.

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 fine-grained typing facilitates the management of access rights and provides host protection from potentially malicious behaviour.

Our process type takes the 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 order-theoretic properties of our subtyping relation.

Various examples of this paper illustrate the usage of our fine-grained process types in distributed systems.

Key Words: concurrency theory, the higher-order π-calculus, type theory, access control

CONTENTS

1. INTRODUCTION

Background. In distributed computing environments nowadays, it is common for *higher-order code* to be transferred from one domain to another [15, 34, 35]. This is recognised as 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, 41, 50]. We believe that our typing system can be readily adapted to related location-based distributed calculi such as those in [6, 12, 28, 17, 40, 47].

Higher-Order Processes. The language we consider, $\lambda\pi_v$, is essentially a call-by-value λ-calculus [39] augmented with the π-calculus primitives [33]. *Values* may be sent and received along communication channels, as in the π-calculus, but functions may also be applied to them, as in the λ-calculus. Thus

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

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 [13, 37, 50], we use $\mathtt{proc}$ 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 $\ulcorner\mathtt{proc}\urcorner$. Values of this type, of the form $\lambda(x\!:\!\mathtt{unit}).P$, will also be abbreviated to $\ulcorner P\urcorner$. 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 "$c?(x\!:\!\tau)\ f\ x$" in (1) above, if τ is the thunked type $\ulcorner\mathtt{proc}\urcorner$ and f is the function *run*, the process may input a thunked process $\ulcorner P\urcorner$ on channel c and execute it.

In papers such as [3, 32, 37, 38, 46, 50] 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 "$c?(x\!:\!\tau)\ f\ x$" in (1), will be well-typed in any environment Γ which allows c to input a value of type $\ulcorner\mathtt{proc}\urcorner$, assuming of course that f is also a well-typed expression of type $\ulcorner\mathtt{proc}\urcorner \to \mathtt{proc}$; for example $\Gamma \supset \{c\!:\!(\ulcorner\mathtt{proc}\urcorner)^{\mathtt{I}}, f\!:\!\ulcorner\mathtt{proc}\urcorner \to \mathtt{proc}\}$.

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* in [23])

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

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

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

But accepting these processes for execution confers on the incoming code 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 $\mathtt{proc}$ do not provide any mechanism for limiting the effect of incoming code.

Typing Processes. In this paper we extend the typing systems of [50, 37, 17] 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 $\mathtt{int}$. Then, for a suitable Γ we will be able to derive the judgements

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

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

$$c?(x : \ulcorner \Delta_{ab} \urcorner)\;\; run\ x$$

indicates that it is only willing to accept processes for execution if they at most read from resource a and write to b. Thus, for example, a process

$$c!\langle \ulcorner \mathsf{Fw}(ab) \urcorner \rangle \;\;|\;\; c?(x : \ulcorner \Delta_{ab} \urcorner)\;\; run\ x$$

is well-typed while

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

is not; the (thunked) process $\ulcorner \mathsf{Fw}(ba) \urcorner$ 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 host processes can, for example, maintain the consistency of local resources. As a simple example consider the process

$$c?\,(x : \ulcorner\Delta_a\urcorner)\,(run\,x \mid Q) \tag{3}$$

where $\ulcorner\Delta_a\urcorner$ denotes $\ulcorner a : (\mathtt{int})^{\mathtt{O}}\urcorner$. This process knows that no matter what code is downloaded the only local read from the resource a will be carried out by the term Q. For example if Q is $\mathsf{Fw}(ab)$ then, regardless of what code is downloaded, all values sent to a will be forwarded to b.

The ability to nest these process types gives even further fine-grained control over code behaviour. For example consider

$$*\mathtt{req}?(y : \ulcorner\Delta_c\urcorner)\,(run\,y \mid c?(x : \ulcorner\Delta_a\urcorner)\,(run\,x \mid a?(z : \mathtt{int})P)) \tag{4}$$

where $\ulcorner\Delta_c\urcorner$ denotes the type $\ulcorner c : (\ulcorner\Delta_a\urcorner)^{\mathtt{O}}\urcorner$. The annotated types ensure that

- the code downloaded on the request channel $\mathtt{req}$ can only access the resource c
- c can only be used to transmit code which can at most access resource a

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 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 $\mathtt{proc}$ 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

$$\Pi(x\!:\!\sigma)\rho$$

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 $\Pi(x\!:\!\sigma)\rho$.) Thus the abstraction Fw will be assigned the type

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

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

Summary of 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 [37] to a higher-order setting. The main technical results can be summarised as follows.

First, we design a novel notion of *type* for the higher-order π-calculus, in which processes have types corresponding to *interfaces*. 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 is 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.

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.

Finally we show in this paper that a typing system based on these ideas can be developed for $\lambda\pi_{\mathrm{v}}$ and moreover it can typecheck many sophisticated instances of programs involving code abstraction and mobility.

Outline of the Paper. This paper is a revised and extended version of [51, 52]; it includes detailed definitions and proofs, as well as more examples omitted from [52].

Section 2 introduces the types and syntax of $\lambda\pi_{\mathrm{v}}$, together with a reduction semantics for the language; it also contains two example descriptions of systems which illustrate the tractability of higher-order code mobility in $\lambda\pi_{\mathrm{v}}$. We also explain the order theoretic property FBC (finite-bounded completeness) over the subtyping relation which will be required to ensure that type inference is coherent. Section 3 proposes the new typing system of $\lambda\pi_{\mathrm{v}}$, explaining the various technical points in the formulation of the inference rules. Since channels appear freely in the process types, we use a kinding technique analogous to the subtyping of system F [7]. Section 4 demonstrates its expressiveness by typing the exam-

TABLE 1

Types

(Type)

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

(Abbreviation)

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

(Environment)

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

ples introduced in Section 2. In Section 5 we discuss the soundness of the typing system. We prove a Subject Reduction Theorem and an elementary Type Safety Theorem. Finally Section 6 concludes the paper with a discussion of the further results, the limitations of our typing system, further issues and related work. Various auxiliary definitions and many of the proofs are routinely relegated to the appendix.

2. A HIGHER-ORDER PROCESS LANGUAGE

In this section we give the syntax and reduction semantics of $\lambda\pi_{\mathrm{v}}$.

Types. The collection of types we use is a straightforward extension of that from the simply typed λ-calculus [14] and IO-subtyping of the π-calculus [19, 17, 37, 50]; a process type, ranged over by π, can either be the constant $\mathtt{proc}$, as in [50], or take the form $[\Delta]$, where Δ is a *channel environment*. The formal definition is given in Table 1; this assumes a set of base types such as $\mathtt{unit}$ and $\mathtt{nat}$, an infinite set of channel or resource *names* $\mathbf{N}$, ranged over by $a, b, ...$, and an infinite set of *variables* $\mathbf{V}$, ranged over by $x, y, ...$

Channel types we use here are as in [50], which is a simplification of the input/output capabilities of [17] (in turn a *strict* generalisation of [37], see Remark in Section 4); they take the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$, a pair consisting of an *input capability* $S_{\mathtt{I}}$ and an *output capability* $S_{\mathtt{O}}$; these input/output capabilities are in turn either a vector of general value types, or $\top$, denoting then the highest capability, or $\bot$, denoting the lowest. They are used to control value passing as follows.

- The input capability $S_{\mathtt{I}}$ grants permission to a process to receive values from a channel and use them with *at most* the permissions specified in $S_{\mathtt{I}}$; fewer permissions may be used but $S_{\mathtt{I}}$ represents an upper limit.
- The output capability $S_{\mathtt{O}}$ grants permission for a process to send values on a channel, as long as the process has *at least* the permissions specified in $S_{\mathtt{O}}$; more permissions may be sent but here $S_{\mathtt{O}}$ represents a lower limit.

The IO-types of [37] can be represented as a special case of our IO-types, using the abbreviations given in Table 1.

TABLE 2
Syntax

(Term)			**(Identifier)**		
$P,Q,\ldots$ ::=	V	value	$u,v,w,\ldots$ ::=	l	literal
$\mid$	$\mathbf{0}$	nil	$\mid$	$x,y,z,\ldots$	variable
$\mid$	$P \mid P$	parallel	$\mid$	$a,b,c,\ldots$	channel
$\mid$	$u!\langle V_1,\ldots,V_n\rangle P$	output	**(Value)**		
$\mid$	$u?(x_1:\tau_1,\ldots,x_n:\tau_n)P$	input	$V,W,\ldots$ ::=	$u,v,w,\ldots$	identifier
$\mid$	$*P$	replicator	$\mid$	$\lambda(x:\tau)P$	abstraction
$\mid$	$(\nu a:\sigma)P$	restriction			
$\mid$	$P\,P$	application			

(**Literal**)			(**Abbreviations**)	
$l,l',\ldots$ ::=	$()$	unit	$\ulcorner P\urcorner \stackrel{\text{def}}{=} \lambda(x:\texttt{unit})P$	thunk
$\mid$	$1,2,3,\ldots$	numbers	$run \stackrel{\text{def}}{=} \lambda(x:\texttt{unit}\to\pi)\,x\,()$	run
$\mid$	$\texttt{succ},\ldots$	functions		

There are three kinds of value types: base types, channel types as will be 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 $\Pi(x:\sigma)\rho$, where ρ in turn is either a HO-value type or a process type. As already explained process types can either be the constant $\texttt{proc}$, (also denoted *ok* in [37]) or a type channel environment $[\Delta]$ where Δ is a mapping from $\mathbf{N}\cup\mathbf{V}$ to channel types; we also need a mapping from $\mathbf{N}\cup\mathbf{V}$ to value types, ranged over by Γ; the formation rules for environments are also given in Table 1.

EXAMPLE 2.1. (Types)

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

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

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

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

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

It should be emphasised that, despite these examples, the formation rules allow the construction of many meaningless types, in particular process types. In the next section we will introduce judgements which will constrain their formation, giving rise to well-formed types, and their use in well-formed environments.

Syntax.
The syntax for terms in the language $\lambda\pi_v$ is given in Table 2. It is essentially an integration of the λ-calculus and the π-calculus, except that we use the more expressive

TABLE 3

Free Names and Variables

(Free Names)

Terms:

$\mathsf{fn}(\mathbf{0}) = \mathsf{fn}(l) = \mathsf{fn}(x) = \emptyset \quad \mathsf{fn}(a) = \{a\}$

$\mathsf{fn}(P \mid Q) = \mathsf{fn}(PQ) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)$

$\mathsf{fn}(*P) = \mathsf{fn}(P)$

$\mathsf{fn}(u?(x_1:\tau_1,\ldots,x_n:\tau_n)P)$
$= \mathsf{fn}(u) \cup \mathsf{fn}(\tau_1) \cup \ldots \cup \mathsf{fn}(P)$

$\mathsf{fn}(u!\langle v_1,\ldots,v_n\rangle P)$
$= \mathsf{fn}(u) \cup \mathsf{fn}(v_1) \cup \ldots \cup \mathsf{fn}(v_n) \cup \mathsf{fn}(P)$

$\mathsf{fn}((\nu\, a:\sigma)P) = \mathsf{fn}(\sigma) \cup \mathsf{fn}(P) \setminus \{a\}$

$\mathsf{fn}(\lambda(x:\tau)P) = \mathsf{fn}(\tau) \cup \mathsf{fn}(P)$

Types and Environments:

$\mathsf{fn}(\top) = \mathsf{fn}(\bot) = \mathsf{fn}(\mathtt{unit})$
$= \mathsf{fn}(\sigma_B) = \mathsf{fn}(\mathtt{proc}) = \emptyset$

$\mathsf{fn}(\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle) = \mathsf{fn}(S_{\mathtt{I}}) \cup \mathsf{fn}(S_{\mathtt{O}})$

$\mathsf{fn}((\tau_1,\ldots,\tau_n)) = \mathsf{fn}(\tau_1) \cup \ldots \cup \mathsf{fn}(\tau_n)$

$\mathsf{fn}(\sigma_H \to \rho) = \mathsf{fn}(\sigma_H) \cup \mathsf{fn}(\rho)$

$\mathsf{fn}(\Pi(x:\sigma)\rho) = \mathsf{fn}(\sigma) \cup \mathsf{fn}(\rho)$

$\mathsf{fn}([\Delta]) = \mathsf{fn}(\Delta)$

$\mathsf{fn}(\Gamma) = \{\mathsf{fn}(u) \cup \mathsf{fn}(\tau) \mid u:\tau \in \Gamma\}$

(Free Variables)

$\mathsf{fv}(x) = \{x\},\ \mathsf{fv}(a) = \emptyset$

$\mathsf{fn}(u?(x_1:\tau_1,\ldots,x_n:\tau_n)P)$
$= \mathsf{fn}(u) \cup \mathsf{fn}(\tau_1) \cup \ldots \cup (\mathsf{fn}(P) \setminus \{x_1,\ldots,x_n\})$

$\mathsf{fv}((\nu\, a:\sigma)P) = \mathsf{fv}(\sigma) \cup \mathsf{fv}(P)$

$\mathsf{fv}(\lambda(x:\tau)P) = \mathsf{fv}(\tau) \cup \mathsf{fv}(P) \setminus \{x\}$

$\mathsf{fv}(\Pi(x:\sigma)\rho) = \mathsf{fv}(\sigma) \cup \mathsf{fv}(\rho) \setminus \{x\}$

Other rules are given by replacing $\mathsf{fn}(\ )$ by $\mathsf{fv}(\ )$.

types, from Table 1. From the λ-calculus, there are values, consisting of basic values and abstractions, together with application; from the π-calculus we have input and output on communication channels, dynamic channel creation, iteration and the empty process. 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. We will also assume syntax for some simple functions for manipulating values of the basic datatypes, such as `succ`.

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 and free variables, which is required for the definition of substitution; this in turn is central to the reduction semantics. For completeness we give definitions of $\mathsf{fn}(P), \mathsf{fn}(\alpha)$ the free names occurring in the term P and type α, and $\mathsf{fv}(P), \mathsf{fv}(\alpha)$, corresponding free variables in Table 3, which is defined in Table 3.

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

$$P \longrightarrow Q$$

TABLE 4

Reduction

(Reduction)

$$(\beta) \quad (\lambda(x{:}\tau)P)\,V \longrightarrow P\{V/x\} \qquad (\mathrm{app}_r)\ \frac{Q \longrightarrow Q'}{P\,Q \longrightarrow P\,Q'} \qquad (\mathrm{app}_l)\ \frac{P \longrightarrow P'}{P\,V \longrightarrow P'\,V}$$

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

$$(\mathrm{par})\ \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad (\mathrm{res})\ \frac{P \longrightarrow P'}{(\nu a{:}\sigma)P \longrightarrow (\nu a{:}\sigma)P'} \qquad (\mathrm{str})\ \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

(Structure Equivalence)

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

between programs and follows the standard approach from [33, 37, 41]; the formal definition is given in Table 4 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 Table 4 and it is also assumed to be preserved by the parallel composition " $\mid$ " and restriction operator "ν". It has an associated reduction rule (str). We will assume, without comment, some standard reduction rules for the built-in function symbols, such as `succ`. We also use $\longrightarrow\!\!\!\!\rightarrow$ to denote the multi-step reduction. Table 4 also contains the standard congruence rules for reduction, (app), (par), and (res). However the main rules are β-reduction, (β), and communication (com). Both rules require a definition of substitution of values for variable $\{V/x\}$, which we have yet to define. Complications arise when the value V to be substituted is a channel name and is best explained with an example:

$$\lambda(x:(\mathtt{int})^{\mathtt{IO}}).\lambda(y:\ulcorner x:(\mathtt{int})^{\mathtt{I}},a:(\mathtt{int})^{\mathtt{O}}\urcorner)(run\ y \mid x!\langle 1\rangle \mid a?(z)\ r!\langle z\rangle) \tag{5}$$

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

$$\lambda(y:\ulcorner x:(\mathtt{int})^{\mathtt{I}},a:(\mathtt{int})^{\mathtt{O}}\urcorner)(run\ y \mid b!\langle 1\rangle \mid a?(z)\ r!\langle z\rangle)$$

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

$$\lambda(y:\ulcorner b:(\mathtt{int})^{\mathtt{I}},a:(\mathtt{int})^{\mathtt{O}}\urcorner)(run\ y \mid b!\langle 1\rangle \mid a?(z)\ r!\langle z\rangle)$$

TABLE 5

Substitution into Types

$$\top\{V/x\} = \top,\ \ \bot\{V/x\} = \bot,\ \sigma_B\{V/x\} = \sigma_B,\ \ \mathtt{proc}\{V/x\} = \mathtt{proc}$$

$$\begin{array}{rcll}
\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle\{V/x\} & = & \langle S_{\mathtt{I}}\{V/x\}, S_{\mathtt{O}}\{V/x\}\rangle, & \\
(\tau_1,\ldots,\tau_n)\{V/x\} & = & (\tau_1\{V/x\},\ldots,\tau_n\{V/x\}) & \\
(\sigma_H \to \rho)\{V/x\} & = & \sigma_H\{V/x\} \to \rho\{V/x\} & \\
(\Pi(y\!:\!\sigma)\rho)\{V/x\} & = & \Pi(y\!:\!\sigma\{V/x\})\rho\{V/x\} & \text{with } x \neq y \\
[\Delta]\{V/x\} & = & \sqcup[w\{V/x\}\!:\!\sigma\{V/x\}] & \text{with } w\!:\!\sigma \in \Delta
\end{array}$$

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:

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

However this instance uses the substitution of values into types, $\rho\{V/x\}$; this is defined in Table 5. We will see later that when these substitutions are made to well-defined terms and types, in well-defined environments, the results will be well-defined. We will also see that x appears in a type if and only if it has a channel type. Note that substitution into types is essentially different from that employed in the standard polymorphic λ-calculus [7] and the polymorphic π-calculus [38]; there type variables are instantiated by *types* (say $\mathtt{int}$) whereas here channel variables occurring in types are instantiated by *channels*, not by types.

This definition of substitution into types is for the most part straightforward, with one exception which can be explained using the example function (5) above. If this is applied to the name a we would expect to get the result

$$\lambda(y\!:\!\ulcorner a : (\mathtt{int})^{\mathtt{IO}}\urcorner)(run\ y \,|\, a!\langle 1\rangle \,|\, a?(z)\ r!\langle z\rangle)$$

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 $\ulcorner x : (\mathtt{int})^{\mathtt{I}}, a : (\mathtt{int})^{\mathtt{O}}\urcorner$ should be $\ulcorner a : (\mathtt{int})^{\mathtt{IO}}\urcorner$. This is reflected in the final clause in Table 5:

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

Here $\sqcup$ is an operator on types which intuitively acts like a (partial) least upper bound with respect to a yet to be defined subtyping order on types. The formal definition of $\sqcup$ and the associated greatest lower bound operator $\sqcap$ is given in Table 1 in the appendix. 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 accessibility of two processes.

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

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

$$\begin{array}{lcll}
[x\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}]\{b/x\} & = & [x\{b/x\}\!:\!(\mathtt{int})^{\mathtt{I}}] \sqcup [a\!:\!(\mathtt{int})^{\mathtt{O}}] & \\
 & = & [b\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}] & \text{and} \\
[x\!:\!(\mathtt{int})^{\mathtt{I}}, a\!:\!(\mathtt{int})^{\mathtt{O}}]\{a/x\} & = & [x\{a/x\}\!:\!(\mathtt{int})^{\mathtt{I}}] \sqcup [a\!:\!(\mathtt{int})^{\mathtt{O}}] & \\
 & = & [a\!:\!(\mathtt{int})^{\mathtt{IO}}] &
\end{array}$$

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 a priori substitution is not always defined. However we will see in the next section that in properly typed environments it is always well-defined.

EXAMPLE 2.2. (Compute server, cf. [50]) 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. To keep matters simple we use integers as data and assume that we have a literal, such as `succ`, representing the successor function, for the operation. Then for some given name a, let **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. It receives a value on y to be processed together with a return channel z to which the processed data is to be sent. It then calculates the successor of y and then returns it along the return channel.

A *successor-server*, **sServ**, is a process which, on requests, sends to the client the parameterised code, which the client can initialise locally to provide the service:

$$\mathbf{sServ}(\mathrm{req}) \Longleftarrow\ *\mathrm{req}?(r)\ r!\langle \lambda(x)\ \mathbf{Succ}(x)\rangle$$

Here the process receives a request on the channel req, in the form of a return channel r, to which the abstraction $\lambda(x)$ **Succ**(x) is sent. A client can now download this code and initialise it using a locally generated channel a which will act as the request channel for data processing:

$$\mathbf{Client}(\mathrm{req}) \Longleftarrow (\nu r)\ \mathrm{req}!\langle r\rangle\ r?(y)\ (\nu a)(\ ya\ | a!\langle 1,c_1\rangle \,|\, a!\langle 2,c_2\rangle \,|\, a!\langle 3,c_3\rangle \,|\, \cdots)$$

Now **sServ**(req) | **Client**(req) can be reduced as follows:

$$\mathbf{sServ}(\mathrm{req})\ |\ c_1!\langle 2\rangle \,|\, c_2!\langle 3\rangle \,|\, \cdots\ |\ (\nu a)(\mathbf{Succ}(a) \,|\, a!\langle n,c_n\rangle \,|\, \cdots) \twoheadrightarrow \cdots$$

This interaction offers protection for the client against the server; the local service point, a, is not revealed to the server. It also economises on client/server interaction, in favour of local communication at the client site. If the remote access is expensive or unreliable, this represents a gain in efficiency or reliability.

EXAMPLE 2.3. (Interface server and mobile client code) A more general form of compute server is given diagrammatically in Figure 1. Here there is an Interface (I) between clients and the collection of services or operations on offer. 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. Now

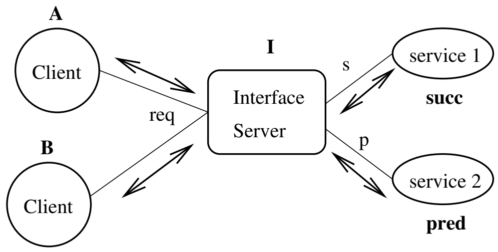

FIG. 1. Interface Server and Distributed Services

instead of a client interacting directly with the services a *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.

For the sake of simplicity suppose there are only two services available, `succ` as before and a similar one for the predecessor function, `pred`:

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

The server at the interface I may then be defined by:

$$\mathbf{Server}_I(\mathrm{req},s,p) \quad \Longleftarrow \quad *\mathrm{req}?(x)\ x\,s\,p$$

It takes in a script x, a process parameterised on service ports, and applies it to the actual port names of the two local 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 req and the result channels r_A, r_B and r_C.

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.

$$\begin{array}{lcl}\mathbf{Client}_A(\mathrm{req}) & \Longleftarrow & \mathrm{req}!\langle\lambda(s,p)\ ((\nu\, c)s!\langle k,c\rangle c?(z)\ s!\langle z,c\rangle\, \mathsf{Fw}(cr_A))\rangle \\ \mathbf{Client}_B(\mathrm{req}) & \Longleftarrow & \mathrm{req}!\langle\lambda(s,p)\ (\nu\, cc')(s!\langle n,c\rangle\, \mathsf{Fw}(c\ r_{1B}) \mid p!\langle m,c'\rangle\, \mathsf{Fw}(c'\ r_{2B}))\rangle\end{array}$$

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 1):

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

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

TABLE 6

Well-formed Types and Subtyping

(Well-formed Environment)

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

(Well-formed Types)

(t-base) $\dfrac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \top : \mathtt{tp},\ \ \bot : \mathtt{tp},\ \ \sigma_B : \mathtt{tp},\ \ \mathtt{proc} : \mathtt{tp},\ \ [\,] : \mathtt{tp}}$ (t-sort) $\dfrac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash (\tau_1, \ldots, \tau_n) : \mathtt{tp}}$

(t-abs$_H$) $\dfrac{\Gamma \vdash \sigma_H : \mathtt{tp}, \quad \rho : \mathtt{tp}}{\Gamma \vdash \sigma_H \to \rho : \mathtt{tp}}$ (t-abs$_N$) $\dfrac{\Gamma, x : \sigma \vdash \rho : \mathtt{tp}}{\Gamma \vdash \Pi(x : \sigma)\rho : \mathtt{tp}}$

(t-proc) $\dfrac{\forall u \in \mathsf{dom}(\Delta).\ \Gamma \vdash \Gamma(u) \le \Delta(u)}{\Gamma \vdash [\Delta] : \mathtt{tp}}$ (t-chan) $\dfrac{\Gamma \vdash S_{\mathtt{I}} \ge S_{\mathtt{O}}}{\Gamma \vdash \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle : \mathtt{tp}}$

(Subtyping)

(s-id) $\dfrac{\Gamma \vdash \alpha : \mathtt{tp}}{\Gamma \vdash \alpha \le \alpha}$

(s-sortb) $\dfrac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash \bot \le (\tau_1, \ldots, \tau_n) \le \top}$

(s-sort) $\dfrac{\Gamma \vdash \tau_i \le \tau_i'}{\Gamma \vdash (\tau_1, \ldots, \tau_n) \le (\tau_1', \ldots, \tau_n')}$

(s-abs$_H$) $\dfrac{\Gamma \vdash \sigma_H' \le \sigma_H, \quad \rho \le \rho'}{\Gamma \vdash \sigma_H \to \rho \le \sigma_H' \to \rho'}$

(s-abs$_H$) $\dfrac{\Gamma, x : \sigma \vdash \rho_1 \le \rho_2}{\Gamma \vdash \Pi(x : \sigma)\rho_1 \le \Pi(x : \sigma)\rho_2}$

(s-chan) $\dfrac{\Gamma \vdash \langle S_{\mathtt{I}i}, S_{\mathtt{O}i} \rangle : \mathtt{tp} \quad (i = 1, 2) \qquad \Gamma \vdash S_{\mathtt{I}1} \le S_{\mathtt{I}2},\ \ S_{\mathtt{O}2} \le S_{\mathtt{O}1}}{\Gamma \vdash \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle \le \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle}$

(s-base) $\dfrac{\Gamma \vdash [\Delta] : \mathtt{tp}}{\Gamma \vdash [\Delta] \le \mathtt{proc}}$

(s-proc) $\dfrac{\Gamma \vdash [\Delta_1] : \mathtt{tp} \qquad \forall u \in \mathsf{dom}(\Delta_2).\ \Gamma \vdash \Delta_1(u) \le \Delta_2(u)}{\Gamma \vdash [\Delta_2] \le [\Delta_1]}$

---

3. THE FINE-GRAINED TYPING SYSTEM

3.1. Well-formed Types and Environments

We present a formal system with three forms of judgements, all interrelated:

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

$$y : \ulcorner x : (\mathtt{nat})^{\mathtt{O}} \urcorner,\ x : (\mathtt{nat})^{\mathtt{O}} \vdash \mathtt{Env}$$

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:

$$x : (\mathtt{nat})^{\mathtt{O}},\ y : \ulcorner x : (\mathtt{nat})^{\mathtt{O}} \urcorner \vdash \mathtt{Env}$$

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

$$\Gamma,\ y:\ulcorner y\!:\!(\mathtt{nat})^{\mathtt{O}}\urcorner \vdash \mathtt{Env}$$

To do so we would need to be able to deduce

$$\Gamma \vdash \ulcorner y\!:\!(\mathtt{nat})^{\mathtt{O}}\urcorner : \mathtt{tp}$$

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

Subtyping judgements are in turn required in the deduction of well-formed types. Intuitively types may be viewed as *sets of capabilities* and the judgement $\Gamma \vdash\ \alpha \leq \alpha'$ may be read as α having at least as many capabilities as α'; in other words a value at type α may safely be used in a context in which one of type α' is required.

Subtyping judgements are required, for example, to ensure that $[\Delta]$ is a well-formed process type, relative to an environment Γ. That is to conclude

$$\Gamma \vdash [\Delta] : \mathtt{tp}$$

we would need to ensure that Δ does not associate with an identifier more capabilities than are available in the overall environment; we need to ensure

$$\Gamma \vdash \Gamma(u) \leq \Delta(u)$$

for every identifier u in the domain of Δ.

Finally it is easy to realise that subtyping judgements will rely on well-formed type judgements, as, in a given environment, we should only be able to compare well-formed types.

The rules for each of the three forms of judgements are given in Table 6. The rules for valid type formation are 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 the capabilities received by readers of a channel, $S_{\mathtt{I}}$, are always a subset of the capabilities given by a sender, $S_{\mathtt{O}}$. Intuitively, similar constraints exist in the typing of function application; there is an analogy between the typing of $(\lambda x.P)V$ and that of $a?(x)\ P \,|\, a!\langle V\rangle Q$.

Then there are only two novelties from the system in [50]. 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; this introduces the notions of free and bound variables in types, and as usual types will be identified up to α-conversion. 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

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

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 [50, 37, 17], 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. In (s-chan), channel types are covariant in the input capability and contravariant in the output. Intuitively, this follows from the interpretation of input/output capabilities given on page 6.

• If $\tau_1 \leq \tau_2$ then, intuitively, τ_2 contains fewer permissions than τ_1. So an identifier with type $(\tau_1)^{\mathtt{I}}$ may be used as one at type $(\tau_2)^{\mathtt{I}}$; values received from the latter can be used with *at most* the permissions in τ_2, which the type $(\tau_1)^{\mathtt{I}}$ ensures.

• Dually whenever $\tau_2 \leq \tau_1$ an identifier with type $(\tau_1)^{\mathtt{O}}$ can be used at type $(\tau_2)^{\mathtt{O}}$; a sender may always send *more* capabilities than are specified in the type.

More detailed discussion about the subtyping ordering on channel types can be found in [17, 50, 37].

Finally the real novelty from [50] is the sub-typing rule for process types, (s-proc); this means the ordering of process types is *contravariant* w.r.t. the ordering of [50, 37].

There is almost an endless series of consistency lemmas which one may prove about this system of judgements. They are invariably deduced by induction on the derivations in the standard manner.

LEMMA 3.1.

1. (Renaming) *Suppose* $v \notin \mathsf{fv}(\Gamma, u, \tau, \Gamma')$. *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-variable) $\Gamma,\ x:\sigma_H,\ \Gamma' \vdash \mathtt{J}$ *implies* $x \notin \mathsf{fv}(\Gamma, \sigma_H, \mathtt{J}, \Gamma')$.

4. (HO-bound change) $\Gamma,\ x:\sigma_H,\ \Gamma' \vdash \mathtt{J}$ *and* $\Gamma \vdash \sigma'_H : \mathtt{tp}$ *imply* $\Gamma,\ x:\sigma'_H,\ \Gamma' \vdash \mathtt{J}$.

5. (Weakening) *Assume* $\Gamma,\ x:\tau \vdash \mathtt{Env}$ *and* $u \notin \mathsf{dom}(\Gamma')$. *Then* $\Gamma,\ \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\ u:\tau,\ \Gamma' \vdash \mathtt{J}$.

6. (Multiple weakening) *Assume* $\Gamma,\ \Gamma'' \vdash \mathtt{Env}$ *and* $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Gamma'') = \emptyset$. *Then* $\Gamma,\ \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\ \Gamma'',\ \Gamma' \vdash \mathtt{J}$.

7. (Bound weakening) *Assume* $\Gamma \vdash \tau' \leq \tau$. *Then* $\Gamma, u:\tau, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, u:\tau', \Gamma' \vdash \mathtt{J}$.

8. (Implied judgement) $\Gamma \vdash \alpha \leq \alpha'$ *implies* $\Gamma \vdash \alpha : \mathtt{tp}$ *and* $\Gamma \vdash \alpha' : \mathtt{tp}$.

9. (HO narrowing) $\Gamma, x:\sigma_H, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma' \vdash \mathtt{J}$.

10. (Exchange) *Assume* $\Gamma, u':\tau' \vdash \mathtt{Env}$. *Then* $\Gamma, u:\tau, u':\tau', \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, u':\tau', u:\tau, \Gamma' \vdash \mathtt{J}$.

PROOF. See Appendix A.2.1. □

Note that in general we can not replace $x:\sigma_H$ with $u:\sigma$ in the statements (4) and (9) above since the channel u may appear freely in Γ and $\mathtt{J}$. This underlines the major technical difference between our system of types and those in the system $F_{<:}$, [7]. Intuitively, here free names behave both as free term variables (since they appear as free names in terms and are *used* as communication ports) and free type variables (since they appear freely in process types). Hence several lemmas for type variables in $F_{<:}$ [7], such as the Bound Change and Narrowing, do not hold for channels in our system.

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

DEFINITION 3.1. (FBC, cf. [17, 50]) We say that a partial order $(\mathbf{S}, \sqsubseteq)$ is *finite bounded complete* (FBC) if for every finite nonempty subset $S \subseteq \mathbf{S}$, if S has a lower bound then S has a greatest lower bound.

PROPOSITION 3.1. *Under an arbitrary well-formed environment, the subtyping relation over types is a partial order and finite bounded complete.*

PROOF. To be precise let $\mathtt{Ty}_\Gamma = \{\alpha \mid \Gamma \vdash \alpha : \mathtt{tp}\}$. We show that, up to α-equivalence, $\mathtt{Ty}_\Gamma$ is a partial order which is finite bounded complete. The proof is given in Appendix A.2.2. □

As an immediate corollary we have the following useful properties of process types:

COROLLARY 3.1. (Process types) *Assume* $\Gamma \vdash \pi_i : \mathtt{tp}$ *with* $i = 1, 2$. *Then:*

1. $\pi_1 \sqcup \pi_2$ *and* $\pi_1 \sqcap \pi_2$ *are always defined and* $\Gamma \vdash \pi_1 \sqcup \pi_2 : \mathtt{tp}$, $\pi_1 \sqcap \pi_2 : \mathtt{tp}$
2. $\pi_1 \sqcup \pi_2 = \mathtt{proc}$ *implies either* $\pi_1 = \mathtt{proc}$ *or* $\pi_2 = \mathtt{proc}$, *and* $\pi_1 \sqcap \pi_2 = \mathtt{proc}$ *implies* $\pi_1 = \mathtt{proc}$ *and* $\pi_2 = \mathtt{proc}$.

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

PROOF. By induction on the derivation of the judgement $\Gamma, x{:}\sigma, \Gamma' \vdash \mathtt{J}$, and a case analysis of then last rule employed. For the most part this proceeds as a standard Substitution Lemma for a typing system (see [50]). But because names may now appear in types there are some novel cases, which we emphasise here.

Suppose the last rule employed is (e-val). Then there are two cases depending on whether x is used or not. Then we have $\Gamma, x:\sigma, \Gamma'', v:\tau \vdash \mathtt{Env}$ because $\Gamma, x:\sigma, \Gamma'' \vdash v:\tau$. Applying induction to this latter derivation, we have $\Gamma, \Gamma''\{u/x\} \vdash \tau\{u/x\} : \mathtt{tp}$. Note $v \notin \mathsf{dom}(\Gamma, \Gamma''\{u/x\})$. Hence by applying (e-val) to the above, we have $\Gamma, \Gamma''\{u/x\}, v : \tau\{u/x\} \vdash \mathtt{Env}$.

Next suppose $x = v$. Then the judgement has the form $\Gamma, x:\sigma \vdash \mathtt{Env}$ and we have to prove $\Gamma \vdash \mathtt{Env}$. But this follows from Part (2) of Lemma 3.1.

If the judgement is derived from (t-base), then it is obvious. The cases when the final judgment is derived from any of the rules (t-sort), (t-chan), (s-id), (s-sortb), (s-sort), (s-chan), (s-abs$_H$), (s-abs$_N$), or (t-abs$_N$) are straightforward applications of the inductive hypothesis. As an example we consider the possibility, when the judgement has the form $\Gamma, x:\sigma, \Gamma' \vdash \Pi(y:\sigma_0)\rho : \mathtt{tp}$, and is derived from $\Gamma, x:\sigma, \Gamma', y:\sigma_0 \vdash \rho : \mathtt{tp}$ with $x \neq y$ by (t-abs$_N$). By the inductive hypothesis, we have $\Gamma, \Gamma'\{u/x\}, y:\sigma_0\{u/x\} \vdash \rho\{u/x\} : \mathtt{tp}$. Then application of (t-abs$_N$) gives $\Gamma, \Gamma'\{u/x\} \vdash \Pi(y:\sigma_0\{u/x\})\rho\{u/x\} : \mathtt{tp}$. Since $\Pi(y:\sigma_0\{u/x\})\rho\{u/x\} = (\Pi(y:\sigma_0)\rho)\{u/x\}$, we are done.

There are three remaining cases, involving interface types, (s-base), (s-proc) and (t-proc). The first follows easily by induction but the other two require attention. As an example we consider the last.

Here the judgement has the form $\Gamma, x:\sigma, \Gamma' \vdash [\Delta] : \mathtt{tp}$. This is interesting only when $u, x \in \mathsf{dom}(\Delta)$. Set $\Delta = \{v_1:\sigma_1, ..., v_n:\sigma_n, x:\sigma_x, u:\sigma_u\}$ with $n \geq 0$. Then by formulation of (t-proc) rule, we know:

$$\Gamma, x:\sigma, \Gamma' \vdash \Gamma(u) \leq \sigma_u \qquad \text{and} \qquad \Gamma, x:\sigma, \Gamma' \vdash \sigma \leq \sigma_x$$

First we note that the results in Lemma 3.1 can be used to ensure that $x \notin \mathsf{fn}(\sigma, \Gamma(u))$; the essential reason is the formation rule for well-formed environments, (e-val). Hence we have $\sigma\{u/x\} = \sigma$ and $\Gamma(u)\{u/x\} = \Gamma(u)$. Since the above two sequents are derived using sub-derivations of $\Gamma, x:\sigma, \Gamma' \vdash [\Delta] : \mathtt{tp}$ (see Lemma A.1), we can apply the inductive hypothesis, respectively. Hence we have:

$$\Gamma, \Gamma'\{u/x\} \vdash \Gamma(u) \leq \sigma_u\{u/x\} \quad (\star) \qquad \text{and} \qquad \Gamma, \Gamma'\{u/x\} \vdash \sigma \leq \sigma_x\{u/x\} \quad (\star\star)$$

Next by applying (Multiple weakening), Lemma 3.1 to the assumption $\Gamma \vdash \Gamma(u) \leq \sigma$, we have

$$\Gamma, \Gamma'\{u/x\} \vdash \Gamma(u) \leq \sigma$$

Now by transitivity, $(\star\star)$ and the above judgement imply:

$$\Gamma, \Gamma'\{u/x\} \vdash \Gamma(u) \leq \sigma_x\{u/x\}$$

Then by FBC, $(\star)$ and the above judgement imply $\sigma_u\{u/x\} \sqcap \sigma_x\{u/x\}$ is always defined, and

$$\Gamma, \Gamma'\{u/x\} \vdash \Gamma(u) \leq \sigma_u\{u/x\} \sqcap \sigma_x\{u/x\}$$

For $v_i \in \mathsf{dom}(\Delta) \setminus \{u, x\}$, by the inductive hypothesis, we also have:

$$\Gamma, \Gamma'\{u/x\} \vdash \Gamma(v_i) \leq \sigma_i\{u/x\} : \mathtt{tp}$$

Hence

$$\begin{aligned} [\Delta]\{u/x\} &= [v_1\{u/x\}:\sigma_1\{u/x\}] \sqcup \cdots \sqcup [u:\sigma_u\{u/x\}] \sqcup [x\{u/x\}:\sigma_x\{u/x\}] \\ &= [v_1:\sigma_1\{u/x\}, ..., v_n:\sigma_n\{u/x\}, u:(\sigma_u\{u/x\} \sqcap \sigma_x\{u/x\})] \end{aligned}$$

is well-formed under $\Gamma, \Gamma'\{u/x\}$. $\square$

Finally we conclude this subsection with the following property, which relates to the construction of an efficient algorithm of the typing system (cf. [46, 37, 50]). Since our system does not include recursive types, it is mechanical to show that in any derivation rule the complexity of the hypothesis is strictly smaller than that of the conclusion (see Appendix A.2 for details). Then the following proposition is straightforward (cf. Lemma 4.2 and Corollary 4.3 in [8]).

PROPOSITION 3.2. (Decidability)

1. *It is decidable whether, given an environment* Γ*, the judgement* $\Gamma \vdash \mathtt{Env}$ *can be derived.*
2. *It is decidable whether, given an environment* Γ *and type expressions* α *and* β*, the judgements* $\Gamma \vdash \alpha : \mathtt{tp}$ *and* $\Gamma \vdash \alpha \leq \beta$ *are derivable.*
3. *It is decidable whether, given an environment* Γ *and type expressions* α *and* β*, the type expressions* $\alpha \sqcup \beta$ *and* $\alpha \sqcap \beta$ *exist such that the judgements* $\Gamma \vdash \alpha \sqcup \beta : \mathtt{tp}$ *and* $\Gamma \vdash \alpha \sqcap \beta : \mathtt{tp}$ *are derivable.*

3.2. Type Inference

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

$\Gamma \vdash P : \alpha$ a term P has a type α under the environment Γ

which uses a subsidiary judgement for identifiers:

$\Gamma \vdash u : \sigma$ a name u has a type σ under the environment Γ

The latter essentially looks up the type associated with u in the environment Γ (see rule (VAL)), although further inferences can be made using the subsumption rule ($\textsc{Sub}_N$).

For convenience the inference rules in Table 7 are divided into three groups. The first, **(Common)**, are elementary, although the subsumption rules ($\textsc{Sub}_H$) and ($\textsc{Sub}_N$) will play a major role in type inference. 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 ($\textsc{Abs}_N$):

$$\frac{\Gamma, x:\sigma \vdash P : \rho}{\Gamma \vdash \lambda(x:\sigma)P : \Pi(x:\sigma)\rho}$$

The corresponding elimination ($\textsc{App}_N$) allows dynamic channel instantiation into types during β-reduction. If a term P has a type $\Pi(x:\sigma)\rho$, we can apply to P a name a with type σ. Then a is substituted for x in ρ.

$$\frac{\Gamma \vdash P : \Pi(x:\sigma)\rho, \quad \Gamma \vdash a : \sigma}{\Gamma \vdash Pa : \rho\{a/x\}}$$

TABLE 7
Typing System for $\lambda\pi_v$

(Common)

$$(\text{VAL})\ \frac{\Gamma, u:\tau, \Gamma' \vdash \mathtt{Env}}{\Gamma, u:\tau, \Gamma' \vdash u:\tau} \qquad (\text{CON})\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash 1:\mathtt{nat}}\ \text{etc.}$$

$$(\text{SUB}_H)\ \frac{\Gamma \vdash P:\rho \quad \Gamma \vdash \rho \leq \rho'}{\Gamma \vdash P:\rho'} \qquad (\text{SUB}_N)\ \frac{\Gamma \vdash u:\sigma \quad \Gamma \vdash \sigma \leq \sigma'}{\Gamma \vdash u:\sigma'}$$

(Function)

$$(\text{ABS}_H)\ \frac{\Gamma, x:\sigma_H \vdash P:\rho}{\Gamma \vdash \lambda(x:\sigma_H)P:\sigma_H \to \rho} \qquad (\text{APP}_H)\ \frac{\Gamma \vdash P:\sigma_H \to \rho \quad \Gamma \vdash Q:\sigma_H}{\Gamma \vdash PQ:\rho}$$

$$(\text{ABS}_N)\ \frac{\Gamma, x:\sigma \vdash P:\rho}{\Gamma \vdash \lambda(x:\sigma)P:\Pi(x:\sigma)\rho} \qquad (\text{APP}_N)\ \frac{\Gamma \vdash P:\Pi(x:\sigma)\rho \quad \Gamma \vdash u:\sigma}{\Gamma \vdash Pu:\rho\{u/x\}}$$

(Process)

$$(\text{NIL})\ \frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \mathbf{0}:[\ ]} \qquad (\text{PAR})\ \frac{\Gamma \vdash P_1:\pi \quad \Gamma \vdash P_2:\pi}{\Gamma \vdash P_1 \,|\, P_2:\pi} \qquad (\text{REP})\ \frac{\Gamma \vdash P:\pi}{\Gamma \vdash *P:\pi} \qquad (\text{RES})\ \frac{\Gamma, a:\sigma \vdash P:\pi}{\Gamma \vdash (\nu a:\sigma)P:\pi \setminus a}$$

$$(\text{OUT})\ \frac{\pi \vdash_\Gamma u:(\tau_1,\ldots,\tau_n)^{\mathtt{O}} \quad \pi \vdash_\Gamma V_i:\tau_i \quad \Gamma \vdash P:\pi}{\Gamma \vdash u!\langle V_1,\ldots,V_n\rangle P:\pi}$$

$$(\text{IN})\ \frac{\pi \vdash_\Gamma u:(\tau_1,\ldots,\tau_n)^{\mathtt{I}} \quad \Gamma, x_1:\tau_1,\ldots,x_n:\tau_n \vdash P:\pi, x_1:\tau_1,\ldots,x_n:\tau_n}{\Gamma \vdash u?(x_1:\tau_1,\ldots,x_n:\tau_n)P:\pi}$$

In (OUT,IN), $\pi \vdash_\Gamma V:\tau$ means $\Gamma \vdash [V:\tau] \leq \pi$ if τ is a channel type; else $\Gamma \vdash V:\tau$.

In (IN), $\pi, x:\tau$ is defined as: (1) $\pi, x:\sigma \stackrel{\text{def}}{=} \pi \sqcup [x:\sigma]$, and (2) $\pi, x:\sigma_H \stackrel{\text{def}}{=} \pi$.

As an example of the use of this rule consider the channel abstraction $P \equiv \lambda(x:(\mathtt{nat})^{\mathtt{O}})(x!\langle 1\rangle \,|\, b?(y:\mathtt{nat})\ \mathbf{0})$ which in an appropriate environment can be assigned the type $\rho = \Pi(x:(\mathtt{nat})^{\mathtt{O}})[x:(\mathtt{nat})^{\mathtt{O}}, b:(\mathtt{nat})^{\mathtt{I}}]$. We examine the following two applications to P.

$$Pc \longrightarrow c!\langle 1\rangle \ |\ b?(x:\mathtt{nat})\ \mathbf{0} \qquad \text{and} \qquad Pb \longrightarrow b!\langle 1\rangle \ |\ b?(x:\mathtt{nat})\ \mathbf{0}$$

The former process will be assigned the type $[c:(\mathtt{nat})^{\mathtt{O}}, b:(\mathtt{nat})^{\mathtt{I}}]$. But to calculate the type of the latter recall that the general definition of substitution $\rho\{a/x\}$ was defined using the partial join operator $\sqcup$ in Table 5. Thus by definition the latter has the type

$$[x:(\mathtt{nat})^{\mathtt{O}}]\{b/x\} \sqcup [b:(\mathtt{nat})^{\mathtt{I}}] = [b:(\mathtt{nat})^{\mathtt{O}}] \sqcup [b:(\mathtt{nat})^{\mathtt{I}}] = [b:(\mathtt{nat})^{\mathtt{IO}}]$$

The final group, **(Process)**, are based on the IO-Typing systems from [37, 17, 50]. However many of the rules are sufficiently novel to warrant detailed explanation.

The empty *rule,* (NIL)*:*. A process type $[\Delta]$ 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:

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

A meta-result about our typing system ensures that, from the hypotheses we can conclude $\Gamma \vdash \pi_i : \mathtt{tp}$. Thus from Corollary 3.1 we know that $\pi_1 \sqcup \pi_2$ exists and $\Gamma \vdash \pi_i \leq \pi_1 \sqcup \pi_2$. So two instances of subsumption and one of (PAR) justifies this derived rule.

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 Table 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 us define the environment

$$\Delta_{ab} = a\!:\!(\ulcorner b\!:\!(\mathtt{int})^{\mathtt{O}}\urcorner)^{\mathtt{IO}}, b\!:\!(\mathtt{int})^{\mathtt{O}}$$

in which a has type to transmit thunked values of type $b\!:\!(\mathtt{int})^{\mathtt{O}}$. Then with the output rule, together with (NIL) and the abstraction rules, we can first establish

$$\Delta_{ab} \vdash b!\langle 1 \rangle : [b\!:\!(\mathtt{int})^{\mathtt{O}}]$$

and therefore

$$\Delta_{ab} \vdash a!\langle \ulcorner b!\langle 1 \rangle \urcorner \rangle : [a\!:\!(\ulcorner b\!:\!(\mathtt{int})^{\mathtt{O}}\urcorner)^{\mathtt{O}}]$$

The input *rule,* (IN)*:.* The rule for prefixing is a straightforward generalisation of that in [50]:

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

TABLE 8

Name Erasing from Types

$$\top \setminus a = \top,\ \ \bot \setminus a = \bot,\ \sigma_B \setminus a = \sigma_B,\ \ \texttt{proc} \setminus a = \texttt{proc}$$

$$\begin{array}{rcl} \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \setminus a & = & \langle S_{\mathtt{I}} \setminus a, S_{\mathtt{O}} \setminus a \rangle \\ (\tau_1, \ldots, \tau_n) \setminus a & = & (\tau_1 \setminus a, \ldots, \tau_n \setminus a) \\ (\sigma_H \to \rho) \setminus a & = & \sigma_H \setminus a \to \rho \setminus a \\ (\Pi(x{:}\sigma)\rho) \setminus a & = & \Pi(x{:}\sigma \setminus a)\rho \setminus a \\ [\Delta] \setminus a & = & [\{u{:}(\sigma \setminus a) \mid u{:}\sigma \in \Delta \ \wedge\ u \neq a\}] \end{array}$$

To deduce that the process $u?(x{:}\tau)\ P$ 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 "$\pi, x{:}\tau$" defined in Table 7. Note "$\pi, x{:}\tau$" denotes "π" if τ is not a channel type. In (IN), by the first sequent 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 $((\texttt{int})^{\mathtt{O}})^{\mathtt{IO}}$. Then one can easily check that

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

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

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

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, $[a{:}(\texttt{nat})^{\mathtt{O}}]$. 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

$$\frac{\Gamma, a{:}\sigma \vdash P : \pi}{\Gamma \vdash P : \pi \setminus a}$$

where $\pi\setminus a$ denotes the result of erasing all occurrences of a from π. This erasure operator on types is defined formally in Table 8. For example, $[a:(\mathtt{nat})^{\mathtt{O}}]\setminus a = [\;]$, and $[b:(\ulcorner a:(\mathtt{nat})^{\mathtt{O}}\urcorner)^{\mathtt{O}}]\setminus a = [b:(\ulcorner\;\urcorner)^{\mathtt{O}}]$. Hence, in appropriate environments, $(\nu a)a!\langle 1\rangle$ has a type $[\;]$ and $(\nu a)b!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle$ has a type $[b:(\ulcorner\;\urcorner)^{\mathtt{O}}]$.

The main property of this operator on types is given by the following Lemma, which is easily established:

LEMMA 3.3. (Channel narrowing) $\Gamma, a:\sigma, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma'\setminus a \vdash \mathtt{J}\setminus a$. $\square$

4. EXAMPLES

In this section we give some examples of the use of the type system, which we hope will also elucidate the various rules.

EXAMPLE 4.1. (cf. §1.3. TYPING PROCESSES) As the first example, let us consider the following process.

$$P \stackrel{\text{def}}{=} (\nu c)(c!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle \mid c?(x:\ulcorner a:\tau\urcorner)\; \mathit{run}\; x) \tag{6}$$

where $\tau = (\mathtt{int})^{\mathtt{O}}$. After one step in the reduction via the private name c, the thunked process $\ulcorner a!\langle 1\rangle\urcorner$ is unfrozen by the function *run*. Hence P should be typed by *at least* the interface $[a:\tau]$. We will give a detailed derivation of a typing of P, using rules in Tables 6 and 7.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \tau:\mathtt{tp}}{a:\tau\vdash \mathtt{Env}}\;(\text{e-val})}{a:\tau\vdash \ulcorner a:\tau\urcorner:\mathtt{tp}}\;(\text{t-abs}_H)}{a:\tau, x:\ulcorner a:\tau\urcorner\vdash \mathtt{Env}}\;(\text{e-val})}{a:\tau, x:\ulcorner a:\tau\urcorner\vdash x:\ulcorner a:\tau\urcorner}\;(\textsc{Val})}{a:\tau, x:\ulcorner a:\tau\urcorner\vdash \mathit{run}\; x:[a:\tau]}\;(\textsc{Abs}_H,\textsc{App}_H)$$

Let us define $\sigma_c = (\ulcorner a:\tau\urcorner)^{\mathtt{IO}}$. By a similar derivation, we can infer $a:\tau\vdash\sigma_c:\mathtt{tp}$. Hence by (IN) together with (Bound Weakening), Lemmas 3.1 and 5.1, we have

$$a:\tau, c:\sigma_c\vdash c?(x:\ulcorner a:\tau\urcorner)\;\mathit{run}\;x : [c:(\ulcorner a:\tau\urcorner)^{\mathtt{I}}, a:\tau]$$

Similarly, we can derive

$$a:\tau, c:\sigma_c\vdash c!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle : [c:(\ulcorner a:\tau\urcorner)^{\mathtt{O}}]$$

Then we use ($\textsc{Sub}_H$) and (PAR) to obtain:

$$a:\tau, c:\sigma_c\vdash c!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle \mid c?(x:\ulcorner a:\tau\urcorner)\;\mathit{run}\;x : [c:\sigma_c, a:\tau]$$

By $[c:\sigma_c, a:\tau]\setminus c = [a:\tau]$, an application of (RES) gives:

$$a:\tau\vdash P:[a:\tau] \tag{7}$$

On the other hand, a similar process whose body is $\mathbf{0}$ is typed as follows.

$$a:\tau\vdash(\nu c)(c!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle \mid c?(x:\ulcorner a:\tau\urcorner)\;\mathbf{0}) : [\;] \tag{8}$$

Note that both processes in (7) and (8) have the same set of free names, that is are typed with respect to the same environment, but have different external effects.

EXAMPLE 4.2. (cf. §1.4. CHANNEL ABSTRACTION) As the second example let us consider the simple forwarder, already discussed in the Introduction, this time with type annotations:

$$\mathsf{Fw} \Longleftarrow \lambda(x:(\mathtt{int})^{\mathtt{I}})\,\lambda(y:(\mathtt{int})^{\mathtt{O}})\,(*\,x?(z{:}\mathtt{int})\,y!\langle z\rangle)$$

An application of the rule (OUT) gives the judgement

$$x:(\mathtt{int})^{\mathtt{I}},\ y:(\mathtt{int})^{\mathtt{O}}, z:\mathtt{int} \vdash y!\langle z\rangle : [\Delta_{xy}]$$

where Δ_{xy} denotes the interface $\{x{:}(\mathtt{int})^{\mathtt{I}},\ y{:}(\mathtt{int})^{\mathtt{O}}\}$. An application of the input rule (IN), followed by an application of (REP) now gives

$$x:(\mathtt{int})^{\mathtt{I}},\ y:(\mathtt{int})^{\mathtt{O}} \vdash *\,x?(z{:}\mathtt{int})\,y!\langle z\rangle : [\Delta_{xy}]$$

Now we may apply the channel abstraction rule (ABS_N) twice to obtain the following type for the forwarder:

$$\vdash \mathsf{Fw} : \Pi(x{:}(\mathtt{int})^{\mathtt{I}})\Pi(y{:}(\mathtt{int})^{\mathtt{O}})[\Delta_{xy}]$$

Let us now see how we can use this typing to assign a type to the process R, which is also discussed in the Introduction:

$$R \Longleftarrow\ s!\langle c\rangle\ c?(y:\tau_{\mathrm{fw}})\ (y\,a\,b)$$

For convenience τ_{fw} denotes the type assigned to the forwarder and let us define

$$\Delta_R \stackrel{\mathrm{def}}{=} \{a{:}(\mathtt{int})^{\mathtt{I}}, b{:}(\mathtt{int})^{\mathtt{O}}, c{:}(\tau_{\mathrm{fw}})^{\mathtt{IO}}, s{:}((\tau_{\mathrm{fw}})^{\mathtt{O}})^{\mathtt{O}}\}$$

Note that types ensure that c can be only used as a *trigger* in the sense of [37] on the receiver side. Then two applications of the rule (APP_N) gives

$$\Delta_R,\ y:\tau_{\mathrm{fw}} \vdash y\,a\,b\ : [\Delta_{ab}]$$

where $[\Delta_{ab}]$ is the obvious instantiation of the type $[\Delta_{xy}]$, namely $[a:(\mathtt{int})^{\mathtt{I}},\ b:(\mathtt{int})^{\mathtt{O}}]$. We can also derive

$$\Delta_R,\ y:\tau_{\mathrm{fw}} \vdash [\Delta_{ab}] \leq [\Delta_R]$$

and therefore using subsumption we have

$$\Delta_R,\ y:\tau_{\mathrm{fw}} \vdash y\,a\,b\ : [\Delta_R].$$

An application of (IN) gives

$$\Delta_R \vdash c?(y:\tau_{\mathrm{fw}})\,(y\,a\,b)\ : [\Delta_R]$$

and finally an application of the output rule gives

$$\Delta_R \vdash R : [\Delta_R]$$

Note that the companion term discussed in the Introduction

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

can be typed in a slightly modified environment, where the capabilities of a and b are interchanged.

EXAMPLE 4.3. (Typed compute server) We now revisit Example 2.2. The process **Succ**(a) receives a pair of values for input on a, an integer y to be calculated and a channel z on which the resulting integer is returned. So we annotate the code with types as follows:

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

where the types ensure that this process only receives the output capability on the return channel z. For convenience let $\sigma_s^{\mathtt{I}}$ denote the type $(\mathtt{int},(\mathtt{int})^{\mathtt{O}})^{\mathtt{I}}$. Then the reader can check that

$$\Gamma, x\!:\!\sigma_s^{\mathtt{I}} \vdash \mathbf{Succ}(x) : [x\!:\!\sigma_s^{\mathtt{I}}]$$

for any Γ such that $\Gamma \vdash \mathtt{Env}$. An application of (ABS$_N$) from Figure 7, then gives:

$$\Gamma \vdash \quad \lambda(x\!:\!\sigma_s^{\mathtt{I}})\,\mathbf{Succ}(x) : \Pi(x\!:\!\sigma_s^{\mathtt{I}})[x\!:\!\sigma_s^{\mathtt{I}}]$$

which means that when x is instantiated by a channel a whose capability is *dominated by* (that is, less than) $\sigma_s^{\mathtt{I}}$, it becomes a process which can *only* offer input at a.

To proceed with the typing of an annotated version of the server **sServ** let τ_λ denote this abstracted type, $\Pi(x\!:\!\sigma_s^{\mathtt{I}})[x\!:\!\sigma_s^{\mathtt{I}}]$.

Then an application of (OUT) followed by one of IN gives

$$\mathrm{req}\!:\!((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}} \vdash \mathrm{req}?(r:(\tau_\lambda)^{\mathtt{O}})\ r!\langle \lambda(x:\sigma_s^{\mathtt{I}})\ \mathbf{Succ}(x)\rangle : [\mathrm{req}\!:\!((\tau_\lambda)^{\mathtt{O}})^{\mathtt{I}}]$$

Note this code is simply a version of **sServ**(req), where the bound variables are annotated with types.

Now let us examine the client **Client**. For convenience let P denote the body $(\,y\,a \mid a!\langle 1, c_1\rangle \mid \cdots)$, Δ_c the environment $\{c_1\!:\!(\mathtt{int})^{\mathtt{O}}, c_2\!:\!(\mathtt{int})^{\mathtt{O}}, \ldots\}$, and $\sigma_s^{\mathtt{IO}}$ the type $(\mathtt{int},(\mathtt{int})^{\mathtt{O}})^{\mathtt{IO}}$. An application of (SUB$_H$) and (PAR) enables us to deduce

$$\Delta_c, y\!:\!\tau_\lambda,\ a\!:\!\sigma_s^{\mathtt{IO}} \vdash P : [a\!:\!\sigma_s^{\mathtt{IO}}, \Delta_c]$$

However we require a more general environment. Let Δ denote Δ_c, $\mathrm{req}\!:\!((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}}$. Then noting $r, a \notin \mathsf{fn}(\Delta)$, by (IN), we can also derive

$$\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}},\ y\!:\!\tau_\lambda,\ a\!:\!\sigma_s^{\mathtt{IO}} \vdash P : [\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}},\ a\!:\!\sigma_s^{\mathtt{IO}}]$$

An application of (RES) now gives

$$\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}},\ y\!:\!\tau_\lambda \vdash (\nu\, a\!:\!\sigma_s^{\mathtt{IO}})P : [\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}}]$$

An application of (IN), followed by (OUT), gives

$$\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}} \vdash \mathrm{req}!\langle r\rangle\, r?(y:\tau_\lambda)\,(\nu\, a\!:\!\sigma_s^{\mathtt{IO}})P : [\Delta,\ r:(\tau_\lambda)^{\mathtt{IO}}]$$

and with one final application on of (RES) we obtain

$$\Delta \vdash (\nu r:(\tau_\lambda)^{\mathtt{IO}})\,\mathrm{req}!\langle r\rangle\, r?(y:\tau_\lambda)\,(\nu a:\sigma_s^{\mathtt{IO}})P : [\Delta]$$

This code is an annotated version of **Client**(req).

We can now type the combined system. By (s-proc) in Figure 6, we know:

$$\begin{array}{lcl} \Delta \vdash [\mathrm{req}:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{I}}] & \leq & [\mathrm{req}:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}},\Delta_c] \text{ and} \\ \Delta \vdash [\mathrm{req}:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{I}},\Delta_c] & \leq & [\mathrm{req}:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}},\Delta_c] \end{array}$$

Hence applying SUB_N to **sServ**(req) and **Client**(req), we have:

$$\Delta \vdash \mathbf{sServ}(\mathrm{req}) \mid \mathbf{Client}(\mathrm{req}) : [\mathrm{req}:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}},\Delta_c]$$

Observe that the type $\tau_\lambda \stackrel{\text{def}}{=} \Pi(x:\sigma_s^{\mathtt{I}})[x:\sigma_s^{\mathtt{I}}]$ which annotates y in the client prevents dangerous code being input via r.

If we only have a constant process type `proc`, as in the previous typing system of the process calculi [50, 37, 13], then the client could input any function $\lambda(x:\sigma)Q$, where Q is an arbitrary process; such incoming code may harm the client's resources.

EXAMPLE 4.4. (Interface server and mobile client code) We revisit Example 2.3, where clients send scripts to a general interface which acts as an interface for a suite of services; results are returned on channels owned by the clients and embedded in the scripts.

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 $\sigma_s^{\mathtt{O}}$ be a type $(\mathtt{int},(\mathtt{int})^{\mathtt{O}})^{\mathtt{O}}$ and τ_{sc} be a type for scripts:

$$\Pi(s:\sigma_s^{\mathtt{O}})\Pi(p:\sigma_s^{\mathtt{O}})[s:\sigma_s^{\mathtt{O}},p:\sigma_s^{\mathtt{O}},\Delta_r]$$

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.

Let us define P_A denotes the body of the script sent by the client $\mathbf{Client}_A$.

$$P_A \stackrel{\text{def}}{=} (\nu c)s!\langle k,c\rangle c?(z)\; s!\langle z,c\rangle\, \mathsf{Fw}(cr_A)$$

Then using subsumption we can form the judgement

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

By channel abstraction we therefore have

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

That is the value sent by $\mathbf{Client}_A$ can indeed have script type τ_{sc}.

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

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

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

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

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

Then the reader can check that

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

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 sends capabilities on the return channels. Note also that if we only have a constant process type $\mathtt{proc}$, then the interface server could input any function $\lambda s.\lambda p.Q$, where Q is an arbitrary process via "req"; such a code may harm resources in the clients and remote services.

We end this section with a brief comparison of our type system with those of [50, 37]. It is easy to see that in our system judgements of the form

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

plays very much the same role as the well-typing judgements, such as

$$\Gamma \vdash P : \circ$$

from [37]. Indeed the more usual input/ouput judgement rules

$$\frac{\Gamma \vdash u : (\tau_1, \ldots, \tau_n)^{\mathtt{I}} \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash P : \mathtt{proc}}{\Gamma \vdash u?(x_1 : \tau_1, \ldots, x_n : \tau_n)P : \mathtt{proc}} \quad \text{and} \quad \frac{\Gamma \vdash u : (\tau_1, \ldots, \tau_n)^{\mathtt{O}} \qquad \Gamma \vdash V_i : \tau_i}{\Gamma \vdash u!\langle V_1, \ldots, V_n\rangle P : \mathtt{proc}}$$

are easily derivable in our system, showing that our typing system is as least as general. However, even for first-order systems, our use of IO-types, where input and output capabilities on a channel may be different [19, 17], gives us more typable terms than [37], as we shall now demonstrate.

Remark. (Comparison with IO-types) For convenience, let us define $\tau_{\mathtt{O}} = ()^{\mathtt{O}}$ and $\tau_{\mathtt{IO}} = ()^{\mathtt{IO}}$ and consider the following processes.

$$\begin{array}{lcl} P & \stackrel{\mathrm{def}}{=} & (\nu b : (\tau_{\mathtt{O}})^{\mathtt{IO}}, c : \tau_{\mathtt{O}})(a!\langle b\rangle \mid b?(x)\ x!\langle\rangle \mid b!\langle c\rangle) \\ Q & \stackrel{\mathrm{def}}{=} & (\nu d : (\tau_{\mathtt{IO}})^{\mathtt{IO}}, e : \tau_{\mathtt{IO}})(a!\langle d\rangle \mid d?(x)\ (x!\langle\rangle \mid x?()\ \mathbf{0}) \mid d!\langle e\rangle) \\ R & \stackrel{\mathrm{def}}{=} & a?(y)\ (\nu f : \tau_{\mathtt{IO}})(y?(z)\ z!\langle\rangle \mid y!\langle f\rangle) \end{array}$$

The process R provides, on channel a, a service for the processes P and Q, although their type requirements are slightly different.

Let Γ be the type environment $a:(\langle(\tau_{\mathtt{O}}),(\tau_{\mathtt{IO}})\rangle)^{\mathtt{IO}}$; note that this is well defined since $\tau_{\mathtt{IO}} \lneq \tau_{\mathtt{O}}$. Then

$$\Gamma \vdash P \,|\, Q \,|\, R : \mathtt{proc}.$$

However the type $(\langle(\tau_{\mathtt{O}}),(\tau_{\mathtt{IO}})\rangle)^{\mathtt{IO}}$ is not allowed in [37]. If we assign any type to a from those in [37], such as $((\tau_{\mathtt{IO}})^{\mathtt{IO}})^{\mathtt{IO}}$ or $((\tau_{\mathtt{O}})^{\mathtt{IO}})^{\mathtt{IO}}$ then we will be able to type P and R, or Q and R, but *not* the entire system $P\,|\,Q\,|\,R$.

Hence our subtyping ordering based on [17] is not only natural to integrate with functional subtypes, but also strictly more general than that of [37] even in the pure polyadic π-calculus.

5. TYPE SOUNDNESS

In this section we show some technical properties of our typing system. The main results are a Subject Reduction Theorem and a simple form of Type Safety.

5.1. Subject Reduction

Lemma 3.1 has natural generalisations to our typing system. These are included in the following Lemma:

LEMMA 5.1.

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

PROOF. See Appendix A.3. □

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 guarantees that substitution, which uses $\sqcup$ in its definition, is always defined when applied to well-typed terms; again it may be viewed as a generalisation of Lemma 3.2:

LEMMA 5.2. (Substitution Lemma) *Assume* $\Gamma \vdash V : \tau$. *Then*

1. $\Gamma, x{:}\tau, \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash \mathtt{J}\{V/x\}$ *and*

2. $\Gamma, x\!:\!\tau, \Gamma' \vdash P : \alpha$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash P\{V/x\} : \alpha\{V/x\}$.

PROOF. Note that if τ is not a channel type then $\Gamma'\{V/x\}$, $\mathtt{J}\{V/x\}$ and $\alpha\{V/x\}$ are simply Γ', $\mathtt{J}$ and α respectively. Hence in this case (1) is trivial and the proof of (2) is also straightforward as it is similar to the corresponding proofs in [46, 37, 17]. However, when x is a channel variable this substitution, of a channel name for a channel variable, may change the structures of Γ', $\mathtt{J}$ and α and the proof is more delicate. In this case, the proof of (1) coincides with Lemma 3.2 since the assumption $\Gamma \vdash u : \sigma$ implies $\Gamma \vdash \Gamma(u) \leq \sigma$ by (Sub$_N$); the proof of (2) is left to Appendix A.3. $\Box$

As an aside this Lemma means that if P is typed under a well-formed environment Γ, then the result of a well-typed substitution is always defined and the result is always uniquely determined.

To prove Subject Reduction theorem, we also need to prove that typing is closed under the structural rules for terms. Here again the order-theoretic property, FBC, plays an essential role.

PROPOSITION 5.1. $\Gamma \vdash P\!:\!\pi$ *and* $P \equiv P'$ *imply* $\Gamma \vdash P'\!:\!\pi$.

PROOF. By induction on the derivation of $P \equiv P'$. The only difficult case is the scope opening axiom

$$(\nu a\!:\!\sigma)P|Q \equiv (\nu a\!:\!\sigma)(P|Q) \text{ if } a \notin \mathsf{fn}(Q)$$

First suppose $\Gamma \vdash (\nu a:\sigma)P|Q : \pi$. We prove by induction on the derivation of this judgement that $\Gamma \vdash (\nu a\!:\!\sigma)(P|Q) : \pi$.

If the last rule used in this derivation is subsumption then we may use induction, followed by an instance of subsumption, to obtain the required judgement. So we may assume that an instance of the rule (PAR) was used. Then we have that $\Gamma \vdash Q : \pi$ and $\Gamma, a\!:\!\sigma \vdash P : \pi'$ for some π' such that $\pi' \setminus a = \pi$. Then we have $\Gamma, a\!:\!\sigma \vdash Q : \pi$ by (Weakening) in Lemma 5.1. Note, however, we can not apply (SUB$_N$) to this sequent in order to get $\Gamma, a\!:\!\sigma \vdash Q : \pi'$ directly since we do not in general have that $\Gamma \vdash \pi' \setminus a \leq \pi'$ (for example if we let π' denote $[b : (\ulcorner a\!:\!\sigma \urcorner)^{\mathtt{IO}}]$ it is easy to find an instance of Γ, Q such that $\Gamma, a:\sigma \vdash \pi' : \mathtt{tp}$ and $\Gamma \vdash Q : \pi' \setminus a$ but $\Gamma, a\!:\!\sigma \nvdash Q : \pi'$).

However, by Corollary 3.1 we know that $\pi' \setminus a \sqcup \pi'$ is always defined, and we have $\pi' \leq (\pi' \setminus a \sqcup \pi')$ and $\pi' \setminus a \leq (\pi' \setminus a \sqcup \pi')$. So we may apply subsumption and the rule PAR to obtain $\Gamma, a\!:\!\sigma \vdash P|Q : (\pi' \setminus a \sqcup \pi')$. A simple calculation gives $(\pi' \setminus a \sqcup \pi') \setminus a = \pi' \setminus a = \pi$ and so we obtain, by the rule (RES) the required $\Gamma \vdash (\nu a\!:\!\sigma)(P|Q) : \pi$.

The converse direction, proving $\Gamma \vdash (\nu a\!:\!\sigma)(P|Q) : \pi$ implies $\Gamma \vdash (\nu a\!:\!\sigma)P|Q : \pi$, is similar, although Channel narrowing, Lemma 5.1 is required. $\Box$

THEOREM 5.1. **(Subject Reduction)**

$$\text{If } \Gamma \vdash P : \rho \text{ and } P \longrightarrow P', \text{ then } \Gamma \vdash P' : \rho.$$

PROOF. The proof is by induction on the derivation of $P \longrightarrow P'$. The only non-trivial cases are applications of the rules (β), (com) and (str) respectively. The last follows from the previous proposition and so we consider the first two.

Before considering the rule (β) directly we first show that

$$\Gamma \vdash v : \sigma \text{ and } \Gamma \vdash \lambda(x\!:\!\sigma)P : \Pi(x\!:\!\sigma)\rho \text{ implies } \Gamma \vdash P\{v/x\} : \rho\{v/x\} \qquad (\star)$$

This is proved by induction on the derivation of the latter judgement. There are two cases.

First suppose the judgement was derived using the rule ($\textsc{Abs}_N$). Then we have $\Gamma, x : \sigma \vdash P : \rho$. In this case we may apply (Substitution lemma), Lemma 5.2, directly, to obtain the required result.

The second case is where the judgement was derived using subsumption, the rule ($\textsc{Sub}_H$). In this case we know $\Gamma \vdash \lambda(x\!:\!\sigma)P : \Pi(x\!:\!\sigma)\rho'$ and $\Gamma \vdash \Pi(x\!:\!\sigma)\rho' \leq \Pi(x\!:\!\sigma)\rho$. So we may apply induction to obtain $\Gamma \vdash P\{v/x\} : \rho'\{v/x\}$. However we also know that $\Gamma, x : \sigma \vdash \rho' \leq \rho$ and so we apply (Name substitution), Lemma 3.2 to obtain $\Gamma \vdash \rho'\{v/x\} \leq \rho\{v/x\}$. The required result now follows by an application of subsumption.

Now let us consider the rule (β). In fact we only consider the interesting case, when the abstraction is over a channel type, $\Gamma \vdash (\lambda(x\!:\!\sigma)P)\,v : \rho$. The case when the abstraction is over a higher-order value type is standard by (Substitution Lemma). Hence we prove, by induction on the derivation of this judgement, that $\Gamma \vdash P\{v/x\} : \rho$. There are two cases.

In the first, the derivation uses the rule ($\textsc{Abs}_N$). Here we know $\Gamma \vdash \lambda(x\!:\!\sigma)P : \Pi(x\!:\!\sigma)\rho'$, $\Gamma \vdash v : \sigma$ and ρ has the form $\rho'\{v/x\}$. So $\Gamma \vdash P\{v/x\} : \rho$ follows directly from an application of ($\star$).

In the second case the derivation uses an instance of subsumption. So $\Gamma \vdash (\lambda(x\!:\!\sigma)P)\,v : \rho'$ for some ρ' such that $\Gamma \vdash \rho' \leq \rho$. By induction we have $\Gamma \vdash P\{v/x\} : \rho'$ an instance of subsumption gives the required result.

Now let us consider the rule (com). For simplicity we only treat the monadic case; the extension to the polyadic case is straightforward. Assume

$$a?(x\!:\!\tau)\,P \,|\, a!\langle V\rangle Q \longrightarrow P\{V/x\} \,|\, Q$$

and the hypothesis is $\Gamma \vdash a?(x\!:\!\tau)\,P \,|\, a!\langle V\rangle Q : \pi$. We need to show $\Gamma \vdash P\{V/x\} \,|\, Q : \pi$. It is straightforward, using the hypothesis, to show $\Gamma \vdash Q : \pi$ and so we concentrate on proving $\Gamma \vdash P\{V/x\} : \pi$. If π is the undifferentiated type $\mathtt{proc}$ the proof follows standard lines. So we assume it has the form $[\Delta]$.

There are two cases. First we consider τ is channel type σ. Analysing the hypothesis we obtain

$$\begin{array}{lll} \Gamma, x\!:\!\sigma \vdash P : [\Delta_1, x\!:\!\sigma] & \text{with } \Gamma, x\!:\!\sigma \vdash \ [a\!:\!(\sigma)^{\mathtt{I}}] \ \leq [\Delta_1] \leq [\Delta] & x \notin \mathsf{fv}(\Delta_1) \\ \Gamma \vdash Q : [\Delta_2] & \text{with } \Gamma \vdash [a\!:\!(\sigma')^{\mathtt{O}}, V\!:\!\sigma'] \leq [\Delta_2] \leq [\Delta] & \\ \Gamma \vdash V : \sigma'. & & \end{array}$$

Noting $x \notin \mathsf{fv}(\sigma)$, we can apply (Channel narrowing), Lemma 3.3, to obtain $\Gamma \vdash [a\!:\!(\sigma)^{\mathtt{I}}] \leq [\Delta_1]$. Then we have: $\Gamma \vdash \Gamma(a) \leq \Delta(a) \leq \Delta_1(a) \leq (\sigma)^{\mathtt{I}}$ and $\Gamma \vdash \Gamma(a) \leq \Delta(a) \leq \Delta_2(a) \leq (\sigma')^{\mathtt{O}}$, which imply $\Gamma \vdash \sigma' \leq \sigma$.

Using subsumption we then have $\Gamma \vdash V : \sigma$ and so we can apply (Substitution Lemma), Lemma 5.2, to obtain $\Gamma \vdash P\{V/x\} : [\Delta_1, x\!:\!\sigma]\{V/x\}$. By calculation this type is $[\Delta_1] \sqcup [V\!:\!\sigma]$ and we have $\Gamma \vdash [\Delta_1] \sqcup [V\!:\!\sigma] \leq [\Delta_1] \sqcup [V\!:\!\sigma'] \leq [\Delta_1] \sqcup [\Delta_2] \leq [\Delta]$. Hence by subsumption we have the required $\Gamma \vdash P\{V/x\} : [\Delta]$.

Next suppose τ is a higher-order type σ_H. In this case, we obtain the hypothesis

TABLE 9

Run-time errors

$$a?(x_1:\tau_1,...,x_n:\tau_n)P \xrightarrow{\Gamma,\pi}_{\mathrm{err}} \quad \text{if } \Gamma \not\vdash [\,a:(\tau_1,...,\tau_n)^{\mathtt{I}}\,] \leq \pi.$$

$$a!\langle V_1,...,V_n\rangle P \xrightarrow{\Gamma,\pi}_{\mathrm{err}} \quad \text{for all } \tau_i \text{ s.t. } \Gamma \vdash V_i:\tau_i,\ \Gamma \not\vdash [\,a:(\tau_1,...,\tau_n)^{\mathtt{O}}\,] \leq \pi.$$

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

$$\begin{array}{ll} \Gamma,x:\sigma_H \vdash P:[\,\Delta_1\,] & \text{with } \Gamma,x:\sigma_H \vdash\ [\,a:(\sigma_H)^{\mathtt{I}}\,]\ \leq [\,\Delta_1\,] \leq [\,\Delta\,] \\ \Gamma \vdash Q:[\,\Delta_2\,] & \text{with } \Gamma \vdash [\,a:(\sigma'_H)^{\mathtt{O}}\,] \leq [\,\Delta_2\,] \leq [\,\Delta\,] \\ \Gamma \vdash V:\sigma'_H. & \end{array}$$

This time we apply (HO-variable), Lemma 3.1 to obtain $\Gamma \vdash [\,a:(\sigma_H)^{\mathtt{I}}\,] \leq [\,\Delta_1\,]$. The rest of the reasoning is just similar to the above case, noting $\Gamma \vdash P\{V/x\}:[\,\Delta_1\,]$ and $V \notin \mathsf{fn}([\Delta]) \cup \mathsf{fv}([\Delta])$. □

5.2. Type Safety

Our typing system is an extension of that for the λ-calculus from [14] and that for the π-calculus from [37]; consequently it guarantees the absence of the typical run-time errors associated with these languages. Rather than duplicate the formulation of these kinds of errors, which involves the development of a complicated *tagging* notation, here we concentrate on the novel run-time type errors which our typing system can catch.

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 Table 9, using a unary predicate $P \xrightarrow{\Gamma,\pi}_{\mathrm{err}}$. 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 processes:

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

if there is 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:

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

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

THEOREM 5.2. **(Type Safety)** *If* $\Gamma \vdash P:\pi$ *then* $P \twoheadrightarrow\not\xrightarrow{\Gamma,\pi}_{\mathrm{err}}$

PROOF. By the Subject Reduction Theorem, we only have to prove $P \stackrel{\Gamma,\pi}{\not\longrightarrow}_{\mathrm{err}}$. We prove the contrapositive, $P \stackrel{\Gamma,\pi}{\longrightarrow}_{\mathrm{err}}$ implies we can not derive $\Gamma \vdash P : \pi$, by induction on the derivation of $P \stackrel{\Gamma,\pi}{\longrightarrow}_{\mathrm{err}}$. The last three cases follow immediately by induction. For the first case it is sufficient to remark that the hypotheses, $\Gamma \vdash [a : (\tau_1, ..., \tau_n)^{\mathtt{I}}] \leq \pi$ for no type τ_i ensures that the required type judgements can not be formed. The output case is just similar. □

We conclude this section with a simple but interesting application of the two theorems. Let us define an input predicate $P \downarrow a^{\mathtt{I}}$ which means P can immediately perform input on the name a as follows:

$$a?(x_1,...,x_n)\, P \downarrow a^{\mathtt{I}} \qquad \frac{P \downarrow a^{\mathtt{I}} \text{ or } Q \downarrow a^{\mathtt{I}}}{(P\,|\,Q) \downarrow a^{\mathtt{I}}} \qquad \frac{P \downarrow a^{\mathtt{I}} \quad a \neq b}{(\nu b)P \downarrow a^{\mathtt{I}}} \qquad \frac{P \downarrow a^{\mathtt{I}}}{*P \downarrow a^{\mathtt{I}}}$$

We also write $P \Downarrow a^{\mathtt{I}}$ if there exists Q such that $P \twoheadrightarrow Q$ and $Q \downarrow a^{\mathtt{I}}$. Similarly we can define the output predicate $P \Downarrow a^{\mathtt{O}}$. Then we have the following proposition:

PROPOSITION 5.2.

- *Suppose* $\Gamma \vdash P : [\Delta]$ *and* $a \notin \mathsf{dom}(\Delta)$. *Then* $\neg P \Downarrow a^{\mathtt{I}}$ *and* $\neg P \Downarrow a^{\mathtt{O}}$.
- *Suppose* $\Gamma \vdash P : [\Delta]$ *and* $\Delta(a) = (\tau_1, ..., \tau_n)^{\mathtt{I}}$ *for some* τ_i. *Then* $\neg P \Downarrow a^{\mathtt{O}}$.
- *Suppose* $\Gamma \vdash P : [\Delta]$ *and* $\Delta(a) = (\tau_1, ..., \tau_n)^{\mathtt{O}}$ *for some* τ_i. *Then* $\neg P \Downarrow a^{\mathtt{I}}$.

PROOF. Because of the Subject Reduction Theorem, it is sufficient to prove the results for the predicates $\downarrow a^{\mathtt{I}}$, rather than $\Downarrow a^{\mathtt{I}}$. These may be proved by induction on the derivation of $P \downarrow a^{\mathtt{I}}$ and $P \downarrow a^{\mathtt{O}}$. □

Now let us revisit the example (3) in Section 1

$$c?\,(x{:}\ulcorner\Delta_a\urcorner)\;(run\,x\;|\,Q) \tag{9}$$

where $\ulcorner\Delta_a\urcorner$ denotes $\ulcorner a : (\mathtt{int})^{\mathtt{O}}\urcorner$. We wish to know whatever code is downloaded via c, after arbitrary computation by itself, it still, at most, uses a to write to Q. This assumes that the process which supplies the downloaded code is typeable in a common environment with (9). So suppose Γ is such that

$$\Gamma \vdash c?\,(x : \ulcorner\Delta_a\urcorner)\;(run\,x\;|\,Q)\,|\,c!\langle\ulcorner R\urcorner\rangle : \pi$$

Because this is derivable one can show that, regardless of π, $\Gamma(c)$ must have the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle$ where

- $S_{\mathtt{I}} \leq (\ulcorner\Delta_a\urcorner)$ and
- $\Gamma \vdash \ulcorner R\urcorner : \tau$ for some τ such that $(\tau) \leq S_{\mathtt{O}}$.

However because of the well-formedness of types we know $\Gamma \vdash S_{\mathtt{O}} \leq S_{\mathtt{I}}$, and therefore by subsumption, we have $\Gamma \vdash R : [\Delta_a]$. Hence by Proposition 5.2, for all b, $\neg R \Downarrow b^{\mathtt{I}}$; moreover if $R \Downarrow c^{\mathtt{O}}$, then $c = a$. Thus by static type-checking, we can ensure the receiver always gets safe incoming code, such as $\ulcorner R\urcorner$, ensuring that it conforms to a predetermined access policy, such as the interface type $\ulcorner\Delta_a\urcorner$.

6. CONCLUSION

This paper developed a new type inference system for a higher-order π-calculus in which the types of processes take the form of an *interface*; a mapping between channel names and input/output capabilities. Both the operational semantics of our calculus and its typing system are simple and straightforward extensions of the semantics and type constructions for the π-calculus (e.g. [19, 46, 37, 38, 50]) and λ-calculus (e.g. [7, 14]).

In the literature on higher-order typed process calculi [19, 37, 38, 46], including extensions to mobile processes [13, 12, 40, 17, 42, 50], all processes are typed by a unique constant type, indicating essentially that they are well-typed relative to a particular environment. In contrast, our fine-grained subtypes can be used to manage access rights of distributed code and to provide host protection from mobile code, as we have shown by examples of server-clients models. More generally in our language we can view the process type of some higher-order code as a *proof* that it will respect the constraints of a particular interface. We believe our typing system offers a basic starting point for the study of various static analysis techniques associated with code mobility, such as those found in [34].

Distributed Higher-Order π-Calculus. The 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 Technical Report associated with this paper, [51], we discuss this issue more explicitly, by extending our typing system to the distributed version of $\lambda\pi_v$, given in Section 6 in [50]. Once more the expressiveness of our fine-grained types means that, for example, channel locality, [50], 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, [36]. We hope that an extension of our scheme to higher-order functional shared variables (i.e. passing environments as $\mathtt{com}$ types) will also be useful in this setting.

More Type Constructors. We believe that it will be relatively straightforward to extend our set of types with many of the standard constructs from the literature; these include recursive types [4, 46, 37], record types [14, 45], polymorphic types [7, 38], and dynamic types [2, 40]. A particularly useful extension would be *linear/affine types*, as in the π-calculus and the linear λ-calculus [30, 6, 19, 49]. For example, in the higher-order input process in (4) of Section 1, if a is a linear name, then we can guarantee that all communications to a will be serviced by the code $a?(z:\mathtt{int})P$. Further, 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 maybe used. An extension of our capability based typing systems to more advanced distributed primitives, especially to constructs involving security [1, 18, 22, 48] would be more challenging.

Typed Behavioural Equality. 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 [30, 37, 38, 49]. Similar techniques could be applied to our language, resulting in a new typed equivalence, where equalities are influenced by the presence of fine-grained process types. Investigation of such equivalences is an interesting research topic, particularly in its application to the refinement of the context equality of [41]. It is also practically meaningful to establish a bisimulation technique in our subtyping calculus by extending methods for HOπ-calculus in [41] and those for CML in [26]; we leave this for future work.

Type Checking. For a practical use of a typing system, it is essential that we can check the well-typedness of a system P against a type environment Γ. For this purpose, we can construct an equivalent typing system to $\vdash$ without (SUB_H,SUB_N) in Figure 7, to obtain a syntax directed system (in type reconstruction we use the partial meet operator to obtain a process type as mentioned in § 3.2). Using this system, since the subtyping relation $\leq$ is always decidable (Proposition 3.2), we can obtain an algorithm to check the typability of P against Γ, as well as an algorithm to compute ρ such that $\Gamma \vdash P : \rho$.

Related to type inference algorithms for the π-calculus in general, Vasco and Honda [46] showed there is a principal typing scheme for Milner's simple sorting with recursive types [32] and analysed its time complexity. Igarashi and Kobayashi [25] proposed a type reconstruction algorithm of a linear and IO-subtyping of the π-calculus by using CSP (constraint satisfaction problems) techniques. More exactly, given P, their type reconstruction generates a set of inequalities between the number of usage of channels which all possible Γ such that $\Gamma \vdash P : \texttt{proc}$ satisfy. (It's complexity remains open). However it is still unknown whether there is a minimal principal typing system (in the sense of [27, 9]) for the IO-subtyping system, even in the π-calculus [19, 17, 37]. More systematic study of principal typings for various typing systems of the π-calculus and the Higher-Order π-calculus [46, 37, 45, 38, 30, 19, 42, 49], as well as related complexity analysis, is still required.

Type Limitations. One limitation of our typing system is that, while name variables in types can be abstracted by channel dependency types $\Pi(x{:}\sigma)\rho$ of the channel λ-abstraction $\lambda(x{:}\sigma)\,P$, a similar abstraction is not allowed when we bind 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. A typical example is the process $a?(x)\ b!\langle\ulcorner x!\langle v\rangle\urcorner\rangle$. In the current system this can only be assigned a process type in which b has the capability to output values of the undifferentiated type $\ulcorner\texttt{proc}\urcorner$.

Clearly some form of channel abstraction would be needed to give a more informative type but it is difficult to see how this might be formulated. One problem here is that, unlike β-application, value reception is nondeterministic. In the composed term

$$a?(x)\ b!\langle\ulcorner x!\langle v\rangle\urcorner\rangle \mid a!\langle c\rangle \mid a!\langle d\rangle \tag{10}$$

the particular channel, c or d which is bound to x depends on which message is delivered to the waiting process. Indeed the residual, after receiving an input, may take one of the (incomparable) types $[\,b{:}(\ulcorner c{:}\sigma\urcorner)^{\mathtt{o}}\,]$ or $[\,b{:}(\ulcorner d{:}\sigma\urcorner)^{\mathtt{o}}\,]$.

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 $[\,b : (\ulcorner\ \urcorner)^{\mathtt{o}}\,]$ which intuitively says that b can output a (thunked) process which has the empty interface. This type is of limited interest when used in context. For example consider

$$(\nu\, a)b!\langle\ulcorner a!\langle 1\rangle\urcorner\rangle \,|\, b?(x{:}\tau)\ \mathit{run}\ x$$

Here essentially the only possibility for τ is the type $\ulcorner\texttt{proc}\urcorner$.

Here we should be able to say that b can output a (thunked) process which contains some *unknown* channel name of type $(\texttt{nat})^{\mathtt{o}}$, 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 current type language is 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 [14, 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: channel instantiation in our system can result in dynamic changes to the types annotating a term. Channels are exchanged as values between processes but they also appear as interaction points in the types of processes. On the other hand, type variables of the polymorphic λ-calculus 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 in the context of the λ-calculus.

Pierce and Sangiorgi [38] recently proposed a polymorphic π-calculus 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 $\exists$), 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 [44], and Tang and Jouvelot developed its subtyping system [43]. This was recently applied to Facile by Kirli [28] and to a concurrent logic programming called HCAL in [31] by Kobayashi, Nakade and Yonezawa. It is natural to imagine a correspondence between their region polymorphic types and our process types. However, the typing system in [28] is different from ours since it simply adds the original effect system to the functional types; hence every process has a constant type *Unit* and channels cannot represent a general form of nested effects such as $\Pi(x:\sigma)[...,x:\sigma_1,b:(\ulcorner..[..,x:\sigma_2,.].,x:\sigma_3..\urcorner)^{\mathtt{O}},...]$ in our system. With respect to the typing system in [31], one may think a specific form of the first order dependency type such as $\Pi(x:\sigma)[x:\sigma,...]$ in our system corresponds to their region polymorphic type $\forall l.(\sigma \xrightarrow{b} \mathtt{proc})$ where $b = \{l_1 : (I_1, O_1), ..., l_n : (I_n, O_n)\}$ is a mapping from a set of regions to IO-capability (see [31] for the detailed definitions). In their system, since a channel can only refer to a region name, it is impossible to represent a type dependent on an individual channel like ours. This feature emphasises an essential difference between the two systems. Let us consider the following example

$$(\lambda x.(x(c) \mid ((\lambda z.x(c))\ulcorner x(d)\urcorner)))f \tag{11}$$

where $f = \lambda x.x!\langle 1\rangle$. Note that after β-reduction, this process will eventually give an output effect only to c, not to d. In our system, under an appropriate environment, one can type this process by a process type $[c:(\mathtt{int})^{\mathtt{O}}]$, which specifies the exact external effect (see Example 4.1 in Section 4 for a similar example); based on this property, one can ensure access control of resources, as proved in Proposition 5.2 in Section 5. In the system in [31], c and d in (11) should belong to the same region, hence one cannot determine exactly on which channels this process has an effect (i.e. either d and c, d only, c only , or none). On the other hand, one may be able to assign more fine-grained type to b in (10) in the previous paragraph if c and d belong to the same region. As a conclusion, it is unclear what kind of behavioural property their typing system can ensure, and moreover, similarly to [28], it is difficult to represent our nested higher-order type in a simple and natural way using their formation.

As a summary, the effect types in [44, 43] are used to represent the region allocation or effects 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 still 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 [11], and showed that it is used to control the mobility of agents. In their language, each located process is equipped with different capabilities (read, input, out, 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.

APPENDIX A

A.1. AUXILIARY DEFINITIONS

In Table 1, we define the partial join and meet operator of the partial order $\leq$ defined in Table 6.

A.2. PROOFS OF SECTION 3

Many of the proofs in this appendix use a measure on the judgements defined in Table 6. This measure is defined so that in any judgement, the complexity of the hypothesis is strictly smaller than that of the conclusion [8].

1. The size of type expression α, $\mathrm{size}_\Gamma(\alpha)$ is defined as follows.

   (a) $\mathrm{size}_\Gamma(\top) = \mathrm{size}_\Gamma(\bot) = \mathrm{size}_\Gamma(\mathtt{proc}) = 2$.

   (b) $\mathrm{size}_\Gamma(\sigma_H \to \rho) = \mathrm{size}_\Gamma(\sigma_H) + \mathrm{size}_\Gamma(\rho) + 1$.

   (c) $\mathrm{size}_\Gamma(\Pi(x{:}\sigma)\rho) = \mathrm{size}_\Gamma(\sigma) + \mathrm{size}_\Gamma(\rho) + 1$.

   (d) $\mathrm{size}_\Gamma((\tau_1,...,\tau_n)) = \Sigma_{1\leq i\leq n}\,\mathrm{size}_\Gamma(\tau_i) + 2$.

   (e) $\mathrm{size}_\Gamma(\langle S_\mathtt{I}, S_\mathtt{O}\rangle) = \mathrm{size}_\Gamma(S_\mathtt{I}) + \mathrm{size}_\Gamma(S_\mathtt{O}) + 2$.

   (f) $\mathrm{size}_\Gamma([\Delta]) = \Sigma_{u\in\mathsf{dom}(\Delta)}\,\mathrm{size}_\Gamma(\Gamma(u)) + \Sigma_{u\in\mathsf{dom}(\Delta)}\,\mathrm{size}_\Gamma(\Delta(u)) + 2$.

2. The size of environments, $\mathrm{size}_\mathtt{e}(\Gamma)$, is defined as follows.

   (a) $\mathrm{size}_\mathtt{e}(\emptyset) = 0$.

   (b) $\mathrm{size}_\mathtt{e}(\Gamma, u{:}\tau) = \mathrm{size}_\mathtt{e}(\Gamma) + \mathrm{size}_\Gamma(\tau) + 1$.

3. The size of a subtyping, kinding and environment judgement $\mathtt{J}$, $\mathrm{size}_\mathtt{j}(\mathtt{J})$, is defined as follows.

   (a) $\mathrm{size}_\mathtt{j}(\Gamma \vdash \mathtt{Env}) = \mathrm{size}_\mathtt{e}(\Gamma) + 1$.

   (b) $\mathrm{size}_\mathtt{j}(\Gamma \vdash \tau : \mathtt{tp}) = \mathrm{size}_\mathtt{e}(\Gamma) + \mathrm{size}_\Gamma(\tau)$.

   (c) $\mathrm{size}_\mathtt{j}(\Gamma \vdash \alpha \leq \beta) = \mathrm{size}_\mathtt{e}(\Gamma) + \mathrm{size}_\Gamma(\alpha) + \mathrm{size}_\Gamma(\beta)$.

Then we have the following lemma, whose proof is straightforward.

TABLE 1

Partial Meet and Join Operators

(o-base)	$\alpha \sqcap \alpha \stackrel{\text{def}}{=} \alpha \sqcup \alpha \stackrel{\text{def}}{=} \alpha$
(o-$\top,\bot$)	$\bot \sqcap S \stackrel{\text{def}}{=} S \sqcap \bot \stackrel{\text{def}}{=} \bot, \top \sqcup S \stackrel{\text{def}}{=} S \sqcup \top \stackrel{\text{def}}{=} \top,$ $\bot \sqcup S \stackrel{\text{def}}{=} S \sqcup \bot \stackrel{\text{def}}{=} \top \sqcap S \stackrel{\text{def}}{=} S \sqcap \top \stackrel{\text{def}}{=} S.$
(o-vec)	$(\tau_1,..,\tau_n) \sqcap (\tau'_1,..,\tau'_n) \stackrel{\text{def}}{=} (\tau_1 \sqcap \tau'_1,..,\tau_n \sqcap \tau'_n)$ $(\tau_1,..,\tau_n) \sqcup (\tau'_1,..,\tau'_n) \stackrel{\text{def}}{=} (\tau_1 \sqcup \tau'_1,..,\tau_n \sqcup \tau'_n)$
(o-abs$_H$)	$(\sigma_H \to \rho) \sqcap (\sigma'_H \to \rho') \stackrel{\text{def}}{=} \sigma_H \sqcup \sigma'_H \to \rho \sqcap \rho'$ $(\sigma_H \to \rho) \sqcup (\sigma'_H \to \rho') \stackrel{\text{def}}{=} \sigma_H \sqcap \sigma'_H \to \rho \sqcup \rho'$
(o-abs$_N$)	$(\Pi(x\!:\!\sigma)\rho) \sqcap (\Pi(x\!:\!\sigma)\rho') \stackrel{\text{def}}{=} \Pi(x\!:\!\sigma)(\rho \sqcap \rho')$ $(\Pi(x\!:\!\sigma)\rho) \sqcup (\Pi(x\!:\!\sigma)\rho') \stackrel{\text{def}}{=} \Pi(x\!:\!\sigma)(\rho \sqcup \rho')$
(o-chan)	$\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcup \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle S_{\mathtt{I}} \sqcup S_{\mathtt{I}}, S_{\mathtt{O}} \sqcap S'_{\mathtt{O}} \rangle$ $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}} \rangle \stackrel{\text{def}}{=} \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}} \rangle$ if $S_{\mathtt{I}} \geq S'_{\mathtt{O}}$ and $S'_{\mathtt{I}} \geq S_{\mathtt{O}}$ else undefined.
(o-cenv)	$\Gamma_1 \sqcap \Gamma_2 \stackrel{\text{def}}{=} \Gamma_1 \setminus \mathsf{dom}(\Gamma_2) \cup \Gamma_2 \setminus \mathsf{dom}(\Gamma_1) \cup \{u\!:\!(\Gamma_1(u) \sqcap \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$ $\Gamma_1 \sqcup \Gamma_2 \stackrel{\text{def}}{=} \{u\!:\!(\Gamma_1(u) \sqcup \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$
(o-proc)	$[\Delta_1] \sqcap [\Delta_2] \stackrel{\text{def}}{=} [\Delta_1 \sqcup \Delta_2] \quad [\Delta_1] \sqcup [\Delta_2] \stackrel{\text{def}}{=} [\Delta_1 \sqcap \Delta_2]$ $\mathtt{proc} \sqcup \pi = \mathtt{proc} \quad \mathtt{proc} \sqcap \pi = \pi$

For sort types (but not value, term or channel types) we can ensure that both $\sqcap$ and $\sqcup$ are total; in all cases of $S \sqcap S'$ (respectively $S \sqcup S'$) not covered by the above clauses (for example if they are structurally dissimilar or do not satisfy the IO constraint), then we set $S \sqcap S' = \bot$ (respectively $S \sqcup S' = \top$).

LEMMA A.1. (Well-formedness)

1. *For every* $\mathtt{J}$ *in Table 6,* $size_{\mathtt{j}}(\emptyset \vdash \mathtt{Env}) \leq size_{\mathtt{j}}(\mathtt{J})$.
2. *If* $\dfrac{\mathtt{J}_1,\ \mathtt{J}_2,\ \ldots,\ \mathtt{J}_n}{\mathtt{J}}$ *in Table 6, then* $size_{\mathtt{j}}(\mathtt{J}_i) \lneq size_{\mathtt{j}}(\mathtt{J})$.

This size function is the measure we refer to when talking of a "smaller" size of judgement. Note that we can similarly check $\sqcup$ and $\sqcap$ given in Table 1 are well-formed by defining $\mathrm{size}_{\mathtt{j}}(\Gamma \vdash \alpha \sqcup \beta) = \mathrm{size}_{\mathtt{j}}(\Gamma \vdash \alpha \sqcap \beta) = \mathrm{size}_{\mathtt{e}}(\Gamma) + \mathrm{size}_{\Gamma}(\alpha) + \mathrm{size}_{\Gamma}(\beta)$.

A.2.1. Proof of Lemma 3.1

All proofs are by induction on the size of the judgements and a case analysis of the last rule employed. The proofs are straightforward and as an example we consider (Implied judgement).

Implied Judgement: $\Gamma \vdash \alpha \leq \alpha'$ *implies* $\Gamma \vdash \alpha\!:\!\mathtt{tp}$ *and* $\Gamma \vdash \alpha'\!:\!\mathtt{tp}$.

The case (s-id) is trivial.
The case (s-sort) is proved by applying (t-sort).
The case (s-chan) is direct by definition.

Assume $\alpha = \sigma_H \to \rho$ and $\alpha' = \sigma'_H \to \rho'$, and the last rule used is (s-abs$_H$). Then we have $\Gamma \vdash \sigma'_H \leq \sigma_H$ and $\Gamma \vdash \rho \leq \rho'$. By induction, we have $\Gamma \vdash \sigma_H : \mathtt{tp}$, $\Gamma \vdash \sigma'_H : \mathtt{tp}$, $\Gamma \vdash \rho : \mathtt{tp}$, and $\Gamma \vdash \rho' : \mathtt{tp}$. Hence application of (t-abs$_H$) gives $\Gamma \vdash \sigma_H \to \rho : \mathtt{tp}$ and $\Gamma \vdash \sigma'_H \to \rho' : \mathtt{tp}$.

Suppose $\alpha = \Pi(x{:}\sigma_1)\rho_1$ and $\alpha' = \Pi(x{:}\sigma_2)\rho_2$, and the last rule used is (s-abs$_N$). Then we have $\Gamma, x{:}\sigma \vdash \rho_1 \leq \rho_2$. By induction, we have $\Gamma, x{:}\sigma \vdash \rho_1 : \mathtt{tp}$ and $\Gamma, x{:}\sigma \vdash \rho_2 : \mathtt{tp}$. Then applying (t-abs$_N$) to each sequent, we have $\Gamma \vdash \Pi(x{:}\sigma)\rho_1 : \mathtt{tp}$ and $\Gamma \vdash \Pi(x{:}\sigma)\rho_2 : \mathtt{tp}$.

The case (s-base) is trivial.

Suppose $\alpha = [\Delta_1]$ and $\alpha' = [\Delta_2]$, and the last rule used is (s-proc). (In this proof, we assume $\leq$ forms a partial order on the smaller size of types as inductive hypothesis). First we note that $\Gamma \vdash [\Delta_1] : \mathtt{tp}$ is direct from the assumption of (s-proc), from which we have $\Gamma \vdash \Gamma(w) \leq \Delta_1(w)$ for all $w \in \mathsf{dom}(\Delta_1)$. Also by assumption, for all $u \in \mathsf{dom}(\Delta_2)$, we have $\Gamma \vdash \Delta_1(u) \leq \Delta_2(u)$. Then, by $\mathsf{dom}(\Delta_2) \subseteq \mathsf{dom}(\Delta_1)$, and transitivity, for all u such that $u \in \mathsf{dom}(\Delta_2)$, we have $\Gamma \vdash \Gamma(u) \leq \Delta_2(u)$. Now an application (t-proc) gives $\Gamma \vdash [\Delta_2] : \mathtt{tp}$.

A.2.2. Proof of Proposition 3.1

In this subsection, we prove Proposition 3.1, that is the subtyping relation over types is a partial order, up to α equivalence, and is finite bounded complete, relative to an arbitrary well-formed environment. It will be useful to note that

- $\Gamma \vdash \mathtt{proc} \leq \alpha$ implies α is $\mathtt{proc}$
- $\Gamma \vdash \top \leq \alpha$ implies α is $\top$ and $\Gamma \vdash \alpha \leq \bot$ implies α is $\bot$
- $\Gamma \vdash \alpha \leq \sigma_B$ implies α is σ_B and $\Gamma \vdash \sigma_B \leq \alpha$ implies α is σ_B.

We start by proving the relation is a partial order. Reflexivity follows immediately from the rule (s-id).

Transitivity: Here we prove, under the assumption that $\Gamma \vdash \alpha : \mathtt{tp}$, $\beta : \mathtt{tp}$, $\gamma : \mathtt{tp}$, that ($\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \beta \leq \gamma$ implies $\Gamma \vdash \alpha \leq \gamma$). The proof is by induction on the combined size of the judgements, and then a case analysis on the structure of α. The base cases, when α is either σ_B, $\top$, $\bot$ or $\mathtt{proc}$ are trivial, and the inductive cases, where α takes one of the forms $(\tau_1, ..., \tau_n)$, $\sigma_H \to \rho$ or $\alpha = \Pi(x{:}\sigma)\rho$, is straightforward by induction. Let us consider the final case when α is a process type in the form of $[\Delta_1]$. If either β or γ is $\mathtt{proc}$, then the result is straightforward. So we assume that these are of the form $[\Delta_1], [\Delta_2], [\Delta_3]$ respectively. Then by assumption, we have

$$\Gamma \vdash [\Delta_1] \leq [\Delta_2] \tag{A.1}$$

and

$$\Gamma \vdash [\Delta_2] \leq [\Delta_3] \tag{A.2}$$

By definition of (t-proc), from (A.1), we have $\Gamma \vdash \Delta_2(u) \leq \Delta_1(u)$ for all $u \in \mathsf{dom}(\Delta_1)$. Similarly from (A.2), we have $\Gamma \vdash \Delta_3(u) \leq \Delta_2(u)$ for all $u \in \mathsf{dom}(\Delta_2)$ (hence for all $u \in \mathsf{dom}(\Delta_1)$). Note that by Lemma A.1, the combined size of the judgements of $\Gamma \vdash \Delta_2(u) \leq \Delta_1(u)$ and $\Gamma \vdash \Delta_3(u) \leq \Delta_2(u)$ is smaller than the combined size of judgements of (A.1) and (A.2). Hence by applying the inductive hypothesis, we have

$$\Gamma \vdash \Delta_3(u) \leq \Delta_1(u) \quad \text{for all } u \in \mathsf{dom}(\Delta_1). \tag{A.3}$$

We also have $\Gamma \vdash [\Delta_3] : \mathtt{tp}$ by definition of (s-proc) from (A.2). Now by applying (s-proc) to this and (A.3), we are done.

Anti-symmetry: Here, under the assumption that $\Gamma \vdash \alpha : \mathtt{tp}$, $\beta : \mathtt{tp}$ we have to prove that ($\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \beta \leq \alpha$ implies $\alpha = \beta$). The structure of the proof is similar to that of transitivity and is omitted.

To show a partial order is finite bounded complete, it suffices to define a partial binary meet operator $\sqcap$ that is commutative and associative[1] and satisfies the condition of Definition 3.1 on pairs. Note because of channel types, we also have to define a partial binary operator $\sqcup$. Their definitions may be found in Table 1.

We have to show these $\sqcap$ and $\sqcup$ satisfy commutativity and associativity, and the following conditions:

(A)	$\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \alpha \leq \gamma$	imply	$\beta \sqcap \gamma$ defined	and	$\Gamma \vdash \alpha \leq \beta \sqcap \gamma$
(B)	$\Gamma \vdash \beta \leq \alpha$ and $\Gamma \vdash \gamma \leq \alpha$	imply	$\beta \sqcup \gamma$ defined	and	$\Gamma \vdash \beta \sqcup \gamma \leq \alpha$
(C)	$\beta \sqcap \gamma$ defined	implies	$\Gamma \vdash \beta \sqcap \gamma \leq \beta$		
(D)	$\beta \sqcup \gamma$ defined	implies	$\Gamma \vdash \beta \leq \beta \sqcup \gamma$		

By induction on the definition of $\sqcap$ and $\sqcup$, one can establish associativity and commutativity. Hence we only have to show the above (A–D). Parts (A-D) are proved simultaneously, by induction on size of the judgements and then a case analysis on the structure of α. It is trivial if α is either $\bot, \top$ or $\mathtt{proc}$.

Suppose $\alpha = \sigma_B$. Then we only have $\beta = \sigma_B$ and $\gamma = \sigma_B$. Hence the proof is trivial.

Suppose $\alpha = \sigma_H \to \rho$. We only prove the case (A). By definition of (s-abs$_N$), β and γ take forms of $\sigma_{H1} \to \rho_1$ and $\sigma_{H2} \to \rho_2$, respectively. Then by assumption, we have $\Gamma \vdash \sigma_{H2} \leq \sigma_{H1}$ and $\Gamma \vdash \rho_1 \leq \rho_2$. Applying parts (B) and (A) of the inductive hypothesis, respectively, $\sigma_{H1} \sqcup \sigma_{H2}$ and $\rho_1 \sqcap \rho_2$ are defined, and we have $\Gamma \vdash \sigma_{H1} \sqcup \sigma_{H2} \leq \sigma_H$ and $\Gamma \vdash \rho \leq \rho_1 \sqcap \rho_2$. Thus by (s-abs$_H$), $\Gamma \vdash \alpha \leq (\sigma_{H1} \sqcup \sigma_{H2}) \to (\rho_1 \sqcap \rho_2) = \beta \sqcap \gamma$.

Suppose $\alpha = \Pi(x{:}\sigma)\rho$. We only prove the case (A). Other cases are similar. Suppose $\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \alpha \leq \gamma$. Then by formulation of (s-abs$_N$), β and γ take forms of $\Pi(x{:}\sigma)\rho_1$ and $\Pi(x{:}\sigma)\rho_2$, respectively. Then by assumption of (s-abs$_N$), we have $\Gamma, x{:}\sigma \vdash \rho \leq \rho_1$ and $\Gamma, x{:}\sigma \vdash \rho \leq \rho_2$. Then by part (A) of the inductive hypothesis, $\Gamma, x{:}\sigma \vdash \rho_1 \sqcap \rho_2 : \mathtt{tp}$ is defined and by applying (abs$_N$-t), we have $\Gamma \vdash \Pi(x{:}\sigma)(\rho_1 \sqcap \rho_2) : \mathtt{tp}$. By part (A) of the inductive hypothesis, we also have $\Gamma, x{:}\sigma \vdash \rho \leq (\rho_1 \sqcap \rho_2)$. Thus by applying (s-abs$_N$), we have $\Gamma \vdash \alpha \leq \Pi(x{:}\sigma)(\rho_1 \sqcap \rho_2)$.

Next we consider the case when α, β and γ are channel types. To keep things simple we will omit references to the environment Γ.

(a) Suppose $\alpha = \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ and $\alpha \leq \beta$ and $\alpha \leq \gamma$. Then we can set $\beta = \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle$ and $\gamma = \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle$. We have to show $\beta \sqcap \gamma$ is defined and $\alpha \leq \beta \sqcap \gamma$. By the formulation of (s-chan), we have

$$S_{\mathtt{I}} \geq S_{\mathtt{O}}, \quad S_{\mathtt{I}} \leq S_{\mathtt{I}i}, \quad S_{\mathtt{I}i} \geq S_{\mathtt{O}i}, \quad \text{and} \quad S_{\mathtt{O}} \geq S_{\mathtt{O}i} \quad \text{with } i = 1, 2.$$

Then by applying parts (A) and (B) of inductive hypothesis to $S_{\mathtt{I}} \leq S_{\mathtt{I}i}$ and $S_{\mathtt{O}} \geq S_{\mathtt{O}i}$, respectively, $S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2}$ and $S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2}$ are defined, and we have $S_{\mathtt{I}} \leq S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2}$ and $S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2} \leq$

[1] By commutativity, we mean: If $\alpha \sqcap \beta$ is defined, then $\beta \sqcap \alpha$ is defined and $\alpha \sqcap \beta \gtreqless \beta \sqcap \alpha$ where $\gtreqless$ is the kernel of $\leq$, i.e. $\alpha \gtreqless \beta$ iff $\alpha \leq \beta$ and $\beta \leq \alpha$. Similarly, by associativity, we mean: If $\alpha \sqcap (\beta \sqcap \gamma)$ is defined, then $(\alpha \sqcap \beta) \sqcap \gamma$ is defined and $(\alpha \sqcap \beta) \sqcap \gamma \gtreqless \alpha \sqcap (\beta \sqcap \gamma)$.

$S_{\mathtt{O}}$. Then by transitivity, we have $S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2} \leq S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2}$. Thus $\beta \sqcap \gamma = \langle S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2}, S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2} \rangle$ is defined and $\alpha \leq \beta \sqcap \gamma$.

(b) Suppose $\alpha = \langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$ and $\beta \leq \alpha$ and $\gamma \leq \alpha$. Then we can write $\beta = \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle$ and $\gamma = \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle$. By the formulation of (s-chan), this time we have

$$S_{\mathtt{I}} \geq S_{\mathtt{O}}, \quad S_{\mathtt{I}} \geq S_{\mathtt{I}i}, \quad S_{\mathtt{I}i} \geq S_{\mathtt{O}i} \quad \text{and} \quad S_{\mathtt{O}} \leq S_{\mathtt{O}i} \quad \text{with } i = 1,2.$$

Then by applying parts (B) and (A) of inductive hypothesis to $S_{\mathtt{I}} \geq S_{\mathtt{I}i}$ and $S_{\mathtt{O}} \leq S_{\mathtt{O}i}$, respectively, $S_{\mathtt{I}1} \sqcup S_{\mathtt{I}2}$ and $S_{\mathtt{O}1} \sqcap S_{\mathtt{O}2}$ are defined, and we have:

$$S_{\mathtt{I}1} \sqcup S_{\mathtt{I}2} \leq S_{\mathtt{I}} \quad \text{and} \quad S_{\mathtt{O}} \leq S_{\mathtt{O}1} \sqcap S_{\mathtt{O}2}.$$

Now we have to show $S_{\mathtt{O}1} \sqcap S_{\mathtt{O}2} \leq S_{\mathtt{I}1} \sqcup S_{\mathtt{I}2}$. Then by parts (C) and (D) of the inductive hypothesis, we have

$$S_{\mathtt{O}1} \sqcap S_{\mathtt{O}2} \leq S_{\mathtt{O}1} \quad \text{and} \quad S_{\mathtt{I}1} \leq S_{\mathtt{I}1} \sqcup S_{\mathtt{I}2},$$

respectively. Then by transitivity, we have $S_{\mathtt{O}1} \sqcap S_{\mathtt{O}2} \leq S_{\mathtt{I}1} \sqcup S_{\mathtt{I}2}$. Hence $\beta \sqcup \gamma$ is defined and $\beta \sqcup \gamma \leq \alpha$.

(c) Suppose $\beta = \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle$ and $\gamma = \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle$. By assumption, we have $\beta \sqcap \gamma = \langle S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2}, S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2} \rangle$. By the part (C) of inductive hypothesis, we have $S_{\mathtt{I}1} \sqcap S_{\mathtt{I}2} \leq S_{\mathtt{I}1}$. By the part (D) of inductive hypothesis, we also have $S_{\mathtt{O}1} \leq S_{\mathtt{O}1} \sqcup S_{\mathtt{O}2}$. Hence $\beta \sqcap \gamma \leq \beta$, as required. (D) is similarly proved.

Finally we consider the cases, when α, β and γ are all non-trivial process types, say $[\Delta]$, $[\Delta_1]$, $[\Delta_2]$ respectively.

(a) Here we have the hypothesis $\Gamma \vdash [\Delta] \leq [\Delta_1], [\Delta_2]$ and we have to show both that $[\Delta_1] \sqcap [\Delta_2]$ exists and that we can derive the judgement $\Gamma \vdash [\Delta] \leq [\Delta_1] \sqcap [\Delta_2]$.

To show that it exists, by definition it suffices to show that $\Delta_1(u) \sqcup \Delta_2(u)$ exists for every u in $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. From the hypothesis we have $\Gamma \vdash [\Delta_i] : \mathtt{tp}$ and so from (t-proc) we have $\Gamma \vdash \Delta_i(u) : \mathtt{tp}$ for any such u. So we may derive the judgements $\Gamma \vdash \Delta_i(u) \leq \langle \top, \bot \rangle$ (this is obtained by smaller size of judgements by Lemma A.1), and now we may use induction, applied to part (B) to conclude that $\Delta_1(u) \sqcup \Delta_2(u)$ exists.

To derive the required judgement $\Gamma \vdash \Delta_1(w) \sqcup \Delta_2(w) \leq \Delta(w)$ for every w in $\mathsf{dom}(\Delta)$. However for every such w, from the hypothesis we know that $\Gamma \vdash \Delta_1(w), \Delta_2(w) \leq \Delta(w)$ and once more the result follows by induction applied to part (B).

(b) Here the hypothesis is $\Gamma \vdash [\Delta_1], [\Delta_2] \leq [\Delta]$ and we must show both that $[\Delta_1] \sqcup [\Delta_2]$ exists and that we can derive the judgement $\Gamma \vdash [\Delta_1] \sqcup [\Delta_2] \leq [\Delta]$.

By definition $[\Delta_1] \sqcup [\Delta_2]$ is $[\Delta_1 \sqcap \Delta_1]$ and therefore to show it exists it is sufficient to prove $\Delta_1(u) \sqcap \Delta_2(u)$ exists for every u in $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$. However from the hypothesis we have, for any such u, that $\Gamma \vdash \Delta(u) \leq \Delta_1(u), \Delta_1(u)$, from which the requirement follows by induction, this time on part (A).

To derive the judgement it is sufficient to prove that for any w in $\mathsf{dom}(\Delta_1 \sqcap \Delta_2)$, $\Gamma \vdash \Delta(w) \leq (\Delta_1 \sqcap \Delta_2)(w)$. There are three possibilities for w; it is either in $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)$, in $\mathsf{dom}(\Delta_1) \setminus \mathsf{dom}(\Delta_2)$ or $\mathsf{dom}(\Delta_2) \setminus \mathsf{dom}(\Delta_1)$. In the first case we have, from the hypothesis, that $\Gamma \vdash \Delta(w) \leq \Delta_i(w)$ and we may apply induction (on part (A)) to obtain $\Gamma \vdash \Delta(w) \leq \Delta_1(w) \sqcap \Delta_2(w)$ and the result follows, because in this case $(\Delta_1 \sqcap \Delta_2)(w) = \Delta_1(w) \sqcap \Delta_2(w)$.

The other two possibilities for w are similar but simpler; the inductive step is not required.

Parts (C) and (D) are also proved simultaneously, this time by simultaneous induction on the definition of the operators $\sqcap$ and $\sqcup$. $\square$

A.3. PROOFS FOR SECTION 5

First consider Lemma 5.1. All of these are proven by induction on the proof of the judgements concerned. We give two example cases.

Implied Judgement: $\Gamma \vdash P : \alpha$ *implies* $\Gamma \vdash \alpha : \mathtt{tp}$.

The proof is by induction on the inference of $\Gamma \vdash P : \alpha$ and examination of the last inference rule used. Most cases are completely straightforward. For example if the last rule used is subsumption, (SUB_N), the the result follows by induction followed by an application of (Implied judgement), Lemma 3.1. We examine two cases in detail.

Suppose the last case is (ABS_H); that is P has the form $\lambda(x : \sigma_H)Q$ and $\Gamma \vdash \lambda(x : \sigma_H)Q : \sigma_H \to \rho$ because $\Gamma, x : \sigma_H \vdash Q : \rho$. By induction we have $\Gamma, x : \sigma_H \vdash \rho : \mathtt{tp}$. Applying Lemma 3.1 (2), we have: $\Gamma \vdash \sigma_H : \mathtt{tp}$. Then an application of (t-abs$_H$) in Table 6, gives $\Gamma \vdash \sigma_H \to \rho : \mathtt{tp}$, as required.

Suppose the last rule used is (APP_N); that is P has the form $Q\ u$ and $\Gamma \vdash Q\ u : \rho\{u/x\}$ because $\Gamma \vdash Q : \Pi(x : \sigma)\rho$ and $\Gamma \vdash u : \sigma$. Applying induction to the first inference we obtain the judgement $\Gamma \vdash \Pi(x : \sigma)\rho : \mathtt{tp}$, and therefore we have $\Gamma, x : \sigma \vdash \rho' : \mathtt{tp}$ for some ρ' such that $\Gamma, x : \sigma \vdash \rho' \leq \rho : \mathtt{tp}$, since the rule (t-abs$_N$) and (s-abs$_N$) must be have been employed. Since the assumption $\Gamma \vdash u : \sigma$ implies $\Gamma \vdash \Gamma(u) \leq \sigma$ by (Sub$_N$), we can apply Name Substitution, Lemma 3.2, to obtain

$$\Gamma \vdash \rho'\{u/x\} \leq \rho\{u/x\}$$

Now by (Implied judgement), Lemma 3.1 again, we have: $\Gamma \vdash \rho\{u/x\} : \mathtt{tp}$, as desired.

Channel Narrowing: *Assume* $a \notin \mathsf{fn}(P)$. *Then* $\Gamma, a : \sigma, \Gamma' \vdash P : \alpha$ *implies* $\Gamma, \Gamma' \setminus a \vdash P : \alpha \setminus a$.

Here the proof is by induction on the inference of the judgement $\Gamma, a : \sigma, \Gamma' \vdash P : \alpha$, together with an examination of the last inference rule used. We consider two cases.

Suppose the last rule used is (ABS_N); that is P has the form $\lambda(x : \sigma).Q$, α is $\Pi(x : \sigma)\rho$ and we know $\Gamma, a : \sigma', \Gamma', x : \sigma \vdash Q : \rho$. Then by the inductive hypothesis, we have $\Gamma, \Gamma' \setminus a, x : \sigma \setminus a \vdash Q : \rho \setminus a$. However since we are assuming $a \notin \mathsf{fn}(P)$ we know $a \notin \mathsf{fn}(\sigma)$, and hence $\sigma \setminus a = \sigma$. Therefore we have $\Gamma, \Gamma' \setminus a, x : \sigma \vdash Q : \rho \setminus a$. We may now apply ($\mathrm{ABS}_N$) to obtain

$$\Gamma, \Gamma' \setminus a \vdash \lambda(x : \sigma).Q : \Pi(x : \sigma)\rho \setminus a$$

as desired.

Suppose the last rule used is (IN); that is P has the form $u?(x_1 : \tau_1, \ldots, x_n : \tau_n)Q$ and $\Gamma, a : \sigma, \Gamma' \vdash u?(x_1 : \tau_1, \ldots, x_n : \tau_n)Q : \pi$. Then we have:
$\Gamma, a : \sigma, \Gamma', x_1 : \tau_1, \ldots, x_n : \tau_n \vdash Q : \pi$ and $\Gamma, a : \sigma, \Gamma' \vdash [u : (\tau_1, \ldots, \tau_n)^{\mathtt{I}}] \leq \pi$. First by reasoning similar to the above case, we have $a \notin \mathsf{fn}(\tau_i)$ since $a \notin \mathsf{fn}(P)$. Then by the inductive hypothesis, we have: $\Gamma, \Gamma' \setminus a, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash Q : \pi \setminus a$ and $\Gamma, \Gamma' \setminus a \vdash [u : (\tau_1, \ldots, \tau_n)^{\mathtt{I}}] \leq \pi \setminus a$. We can now apply (IN) to obtain the required

$$\Gamma, \Gamma' \setminus a \vdash u?(x_1 : \tau_1, \ldots, x_n : \tau_n)Q : \pi \setminus a \qquad \square$$

To prove the Substitution Lemma it is convenient to first prove the following simple instance:

LEMMA A.1. (Channel Substitution) $\Gamma \vdash v : \sigma$ *and* $\Gamma, x{:}\sigma, \Gamma' \vdash u : \sigma'$ *imply* $\Gamma, \Gamma'\{v/x\} \vdash u\{v/x\} : \sigma'\{v/x\}$.

PROOF. For convenience let σ_u denote $(\Gamma, x{:}\sigma, \Gamma')(u)$. Then we know that $\Gamma, x : \sigma, \Gamma' \vdash \sigma_u \leq \sigma'$ and therefore, by (Name substitution), Lemma 3.2, $\Gamma, \Gamma'\{v/x\} \vdash \sigma_u\{v/x\} \leq \sigma'\{v/x\}$. There are now two cases.

First suppose $x = u$. Here we know σ_u and σ coincide and since x can not occur in σ, $\sigma_u\{v/x\}$ is simply σ. It is easy to show $\Gamma \vdash v : \sigma$ implies $\Gamma, \Gamma'\{v/x\} \vdash v : \sigma$ by Multiple weakening, Lemma 5.1 and so an application of subsumption rule gives the required $\Gamma, \Gamma'\{v/x\} \vdash v : \sigma'\{v/x\}$.

Now assume $x \neq u$. Here u is also in the domain of $\Gamma, \Gamma'\{v/x\}$ and $(\Gamma, \Gamma'\{v/x\})(u) = \sigma_u\{v/x\}$. So we have $\Gamma, \Gamma'\{v/x\} \vdash u : \sigma_u\{v/x\}$ and once more by subsumption we have $\Gamma, \Gamma'\{v/x\} \vdash u : \sigma'\{v/x\}$ $\square$

LEMMA A.2. (Substitution Lemma) *Assume* $\Gamma \vdash V : \tau$. *Then* $\Gamma, x{:}\tau, \Gamma' \vdash P : \rho$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash P\{V/x\} : \rho\{V/x\}$.

PROOF. By induction on the judgement of $\Gamma, x{:}\tau, \Gamma' \vdash P : \rho$ and as usual an examination of the last inference rule used.

Suppose P has the form u and the last inference rule used is (VAL). Then we have three cases.

- Assume $\Gamma_1, u{:}\tau', \Gamma_2, x{:}\tau, \Gamma_3 \vdash \mathtt{Env}$ with $x \neq u$. Then by Lemma 5.2 (1), for all V such that $\Gamma_1, u{:}\tau', \Gamma_2 \vdash V : \tau$, we have $\Gamma_1, u{:}\tau', \Gamma_2, \Gamma_3\{V/x\} \vdash \mathtt{Env}$. Hence by applying (VAL), we have $\Gamma_1, u{:}\tau', \Gamma_2, \Gamma_3\{V/x\} \vdash u : \tau'$.
- The case $\Gamma_1, x{:}\tau, \Gamma_2, u{:}\tau', \Gamma_3 \vdash \mathtt{Env}$ with $x \neq u$ is similar.
- Now suppose $\Gamma_1, x{:}\tau, \Gamma_2 \vdash \mathtt{Env}$ with $x = u$. Then by Lemma 5.2 (1) again, we know $\Gamma_1, \Gamma_2\{V/x\} \vdash \mathtt{Env}$. By assumption $\Gamma_1 \vdash V : \tau$. Hence by applying Multiple weakening) we have $\Gamma_1, \Gamma_2\{V/x\} \vdash V : \tau$. Thus $\Gamma_1, \Gamma_2\{V/x\} \vdash u\{V/u\} : \tau$, as desired.

The case (Con) is trivial, while both (Sub_H) or (Sub_N) follow immediately by induction and Lemma 5.2 (1).

We examine one of the abstraction rules, (ABS_H). Here P has the form $\lambda(y : \sigma_H)P$ and we have:

$$\Gamma, x{:}\sigma, \Gamma', y{:}\sigma_H \vdash P : \rho$$

for some ρ. Then by induction, we have

$$\Gamma, \Gamma'\{V/x\}, y{:}\sigma_H\{V/x\} \vdash P\{V/x\} : \rho\{V/x\}$$

Since $(\lambda(y : \sigma_H)P)\{V/x\} = \lambda(y : \sigma_H\{V/x\})P\{V/x\}$, an application of ($\mathrm{ABS}_H$) gives

$$\Gamma, \Gamma'\{V/x\} \vdash (\lambda(y : \sigma_H)P)\{V/x\} : \rho\{V/x\}$$

as required.

We also examine one of the application rules. Suppose P has the form Qw and the last inference rule used is (APP_N). Then we have:

$$\Gamma, x{:}\sigma, \Gamma' \vdash Q : \Pi(y{:}\sigma')\rho' \quad \text{and} \quad \Gamma, x{:}\sigma, \Gamma' \vdash w : \sigma'$$

for some σ', and ρ' such that $\rho = \rho'\{w/y\}$; moreover we may assume that and $y \neq x$. By induction we have:

$$\Gamma, \Gamma'\{V/x\} \vdash Q\{V/x\} : \Pi(y{:}\sigma'\{V/x\})\rho'\{V/x\}$$

Moreover we also have

$$\Gamma, \Gamma'\{V/x\} \vdash w\{V/x\} : \sigma'\{V/x\};$$

If τ is a channel type, and therefore V a channel identifier, this follows from the previous lemma; otherwise noting $\rho'\{w/y\}\{V/x\} = (\rho'\{V/x\})\{w\{V/x\}/y\}$ because $x \neq y$, and applying (APP_N) again, we have:

$$\Gamma, \Gamma'\{V/x\} \vdash Q\{V/x\}w\{V/x\} : \rho\{V/x\}$$

as required.

The case (RES) is also a straightforward use of the inductive hypothesis, noting that $(\pi \setminus a)\{V/x\} = (\pi\{V/x\}) \setminus a$ whenever x and a are different. As a final example we consider the rule (OUT). Here P is $u!\langle V_1,...,V_n\rangle Q$ and we have: $\Gamma, x:\sigma, \Gamma' \vdash Q : \pi$, $\Gamma, x{:}\sigma, \Gamma' \vdash V_i : \tau_i$, $\Gamma, x{:}\sigma, \Gamma' \vdash [u{:}(\tau_1,...,\tau_n)^{\mathsf{o}}] \leq \pi$ and $\Gamma, x{:}\sigma, \Gamma' \vdash [V_i{:}\sigma_i] \leq \pi$ whenever τ_i is a channel type σ_i.

Then applying the inductive hypothesis, or the previous lemma, we have:

$$\Gamma, \Gamma'\{V/x\} \vdash Q\{V/x\} : \pi\{V/x\} \quad \text{and} \quad \Gamma, \Gamma'\{V/x\} \vdash V_i\{V/x\} : \tau_i\{V/x\}$$

Applying Lemma 5.2 (1), again, we obtain:

$$\Gamma, \Gamma'\{V/x\} \vdash [u\{V/x\}{:}(\tau_1\{V/x\},...,\tau_n\{V/x\})^{\mathsf{o}}] \leq \pi\{V/x\} \quad \text{and}$$
$$\Gamma, \Gamma'\{V/x\} \vdash [V_i\{V/x\}{:}\sigma_i\{V/x\}] \leq \pi\{V/x\}$$

An application of (OUT) now gives the required result:

$$\Gamma, \Gamma'\{V/x\} \vdash u\{V/x\}!\langle V_1\{V/x\},...,V_n\{V/x\}\rangle Q\{V/x\} : \pi\{V/x\} \qquad \square$$

ACKNOWLEDGMENT

We thank anonymous referees for their detailed and valuable comments on an early version. We also thank Kohei Honda, Dilsun Kirli, Didier Rémy and Vasco Vasconcelos for their comments and discussion. The example in Remark 4 originally stated in [50] was revised by discussion with Atsushi Igarashi, Naoki Kobayashi and James Riely. Example 4.1 comes from a discussion with Joachim Parrow and Jose Luis Vivas. The first author is supported by EPSEC grant no GR/R33465/01, and the second author was supported financially by EPSRC grant no GR/M71169.

REFERENCES

1. Abadi, M., Secrecy by Typing in Security Protocols, Proc. 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. ACM TPLAS, Vol 13, no 2, pp. 237-268, 1991.
3. Amadio, R., Translating Core Facile, ECRC Research Report 944-3, 1994.
4. Amadio, R. and Cardelli, L., Subtyping Recursive Types, TOPLAS, 15(4), pp.575–631, 1993.
5. Aspinall, D. and Compagnoni, A., Subtyping dependent types. LICS'96, IEEE, 1996. The full version is available as LFCS report, ECS-LFCS-97-370, University of Edinburgh, 1997.

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, 1994.
8. Compagnoni, A., Decidable Higher Order Subtyping. LFCS report ECS-LFCS-97-365, University of Edinburgh, 1997. To appear in Info. & Comp.
9. Compagnoni, A., Subject Reduction and Minimal Types for Higher Order Subtyping. University of Edinburgh, LFCS report ECS-LFCS-97-363. 1997. To appear in Info. & Comp.
10. Girard, J.-Y., Lafont Y. and Taylor, P., *Proofs and Types*, vol. 7 of Cambridge Tracts in Theoretical Computer Science, CUP, 1989.
11. 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.
12. 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.
13. Giacalone, A., Mistra, P. and Prasad, S., Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming, *Proc. ICALP'90*, LNCS 443, pp.765–780, Springer-Verlag, 1990.
14. Gunter, C., *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992.
15. Sun Microsystems Inc., Java home page. http://www.javasoft.com/, 1995.
16. Heintze, N. and Riecke, J., The SLam Calculus: Programming with Secrecy and Integrity, *Proc. POPL'98*, pp.365-377. ACM Press, 1998.
17. Hennessy, M. and Riely, J., Resource Access Control in Systems of Mobile Agents, *Info.&.Comp.*, Vol 173(1), pp.82–120, February 25, 2002.
18. Hennessy, M. and Riely, J., Information Flow vs. Resource Access in the Information Asynchronous Pi-Calculus, *ICALP'00*, LNCS 1854, pp.415–427, Springer, 2000.
19. Honda, K., Composing Processes, *POPL'96*, pp.344-357, ACM Press, 1996.
20. Honda, K., A Theory of Types for the pi-calculus, pp.112, Typescript. November 7, 1998. Available from `http://www.dcs.qmw.ac.uk/~kohei`.
21. Honda, K. and Tokoro, M., A Small Calculus for Concurrent Objects, in *OOPS Messenger*, 2(2):50-54, Association for Computing Machinery, 1991.
22. Honda, K., Vasconcelos, V. and Yoshida, N, Secure Information Flow as Typed Process Behaviour, *ESOP'00*, LNCS 1782, pp.180–199, Springer, 2000.
23. Honda, K. and Yoshida, N., Combinatory Representation of Mobile Processes. *POPL'94*, pp.348–360, ACM Press, 1994.
24. Honda, K. and Yoshida, N., On Reduction-Based Process Semantics. *FSTTCS'13*, LNCS 761, pp. 373–387, Springer-Verlag, December 1993. Full version appeared in *TCS*, pp.437–486, No.151, North-Holland, 1995.
25. Igarashi, A. and Kobayashi, N., Type reconstruction for linear pi-calculus with I/O subtyping, *Info. & Comp.*, 161, pp.1-44, 2000.
26. Jeffrey, A. and Rathke, J., A Theory of Bisimulation for a Fragment of Concurrent ML with Local Names, *LICS'00*, IEEE, 2000.
27. Jim, T., What are principal typings and what are they good for? *POPL'96*, ACM, 1996.
28. Kirli, D., A static type system for detecting potentially transmissible functions, ECOOP Workshop MOS'99, http://cuiwww.unige.ch/ ecoopws, 1999.
29. Knabe, F.C., Language Support for Mobile Agents, PhD Thesis, Carnegie Mellon University, Dec. 1995.
30. Kobayashi, N., Pierce, B. and Turner, D., Linearity and the pi-calculus, *POPL'96*, ACM Press, 1996.
31. Kobayashi, K., Nakade, M. and Yonezawa, A., Static analysis of communication for asynchronous concurrent programming languages, *SAS'95*, LNCS 983, Springer, 1995.
32. Milner, R., Polyadic π-Calculus, *Logic and Algebra of Specification*, Springer, 1992.
33. Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Information and Computation*, 100(1), pp.1–77, 1992.
34. Necula, G., Proof-carrying code. *POPL'97*, pp.106–119, ACM, 1997.
35. ObjectSpace Inc. ObjectSpace Voyager. `http://www.objectspace.com/voyager`, 1997.

36. O'Hearn, P., Power, J., Takeyama, M., and Tennent, D., Syntactic control of interference revised, *MFPS'97*, *ENCS*, Elsevier, 1997.

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

38. Pierce, B.C. and Sangiorgi. D, Behavioral Equivalence in the Polymorphic Pi-calculus. *POPL'97*, ACM Press, 1997. Full version to appear in *Journal of the Association for Computing Machinery (JACM)*.

39. Plotkin, G., Call-by-name, call-by-value and the lambda-calculus, *TCS*, 1:125–159, 1975.

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

41. Sangiorgi, D., *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. Ph.D. Thesis, University of Edinburgh, 1992.

42. Sewell, P., Global/Local Subtyping and Capability Inference for a Distributed π-calculus, *ICALP'98*, LNCS 1443, pp.695–706, Springer-Verlag, 1998.

43. Tang, Y.-M., and Jouvelot, P., Effect systems with subtyping, *PEPM'95*, ACM Press, 1995.

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

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

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

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

48. Volpano, D. and Smith, G., Secure information flow in a multi-threaded imperative language, pp.355–364, *POPL'98*, ACM, 1998.

49. Yoshida, N., Graph Types for Monadic Mobile Processes, *FST/TCS'16*, LNCS 1180, pp. 371–386, Springer-Verlag, 1996. Full version as LFCS Technical Report, LECS-LFCS-96-350, 1996.

50. Yoshida, N. and Hennessy, M., Subtyping and Locality in Distributed Higher Order Processes. *Proc. 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, 1999.

51. Yoshida, N. and Hennessy, M., Assigning Types to Processes, CS Technical Report 02/99, University of Sussex, Available at: http://www.cogs.susx.ac.uk, 1999.

52. Yoshida, N. and Hennessy, M., Assigning Types to Processes, *LICS '00*, the 15th Annual IEEE Symposium on Logic and Computer Science, IEEE, 2000.