Assigning Types to Processes

NOBUKO YOSHIDA and MATTHEW HENNESSY

ABSTRACT. In wide area distributed systems it is now common for *higher-order code* to be transferred from one domain to another; the receiving host may initialise parameters and then execute the code in its local environment. In this paper we propose a fine-grained typing system for a higher-order π-calculus which can be used to control the effect of such migrating code on local environments. Processes may be assigned different types depending on their intended use. This is 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.

A process type is essentially 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 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 the distributed systems.

1 Introduction

BACKGROUND 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 some parameters and then execute the code in the local environment [11, 20, 21]. Of course this is recognised as very dangerous and various schemes have been put forward to ensure the integrity of systems in the presence of such operations. In this paper we propose a new *subtyping system* which can be used to control the effect of migrating code on local environments. Our investigation is in terms of a higher-order π-calculus in which values, including process terms, can be exchanged along communication channels [26, 32]. We believe that our typing system based on fine-grained process types can be readily adapted to related location based distributed calculi such as those presented in [13, 8, 9, 27].

HIGHER-ORDER PROCESSES The language we consider, $\lambda\pi_v$, is essentially a call-by-value λ-calculus [24] augmented with the π-calculus primitives [19]. *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

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

Research funded by EPSRC grant GR/L93056, and ESPRTI Working Group Confer2.
E-mail:{nobuko,matthewh}@cogs.susx.ac.uk.

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 [9, 22, 32], we use `proc` to denote the type of processes.

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

In papers such as [22, 32, 23, 9] typing systems have been suggested which ensure that programs written in $\lambda\pi_v$-related languages are well-behaved. The main judgements normally take the form

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

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

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

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

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

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

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

TYPING PROCESSES In this paper we extend the typing systems of [32, 22, 13] 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. For example let Δ_{ab}, Δ_{ba} denote the environments

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

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

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

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

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

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

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

is well-typed while

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

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

This ability to constrain the effect of imported code means that hosts processes can, for example, maintain the consistency of local resources. As a simple example consider the process

$$\mathrm{c}?\ (x : \langle\Delta_a\rangle)\ (run\, x \mid Q)$$

where $\langle\Delta_a\rangle$ denotes $\langle a : (\mathtt{int})^{\mathtt{O}}\rangle$. This process knows that no matter what code is downloaded the only 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

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

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

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

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

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

which continually supplies the abstraction Fw to requesting clients. (Note that here we are omitting type annotations.) A specific client, such as R defined by

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

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

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

where a is used for input and b for output. 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

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

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

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

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

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

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

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

Allowing channel variables to appear in process types complicates the typing system considerably. The formal definition of what constitutes a valid term and a valid type are interdependent and, as we will see, 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 [6]; there type variables appear in program terms whereas here channel variables appear in terms and types. More precisely, here channels play two interdependent roles; first they are used as *interaction ports* in terms as in the standard process calculi, and at the same time they are used to represent *types of processes*.

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 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 illustrates 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 the kinding technique analogous to the subtyping of system F [6]. Section 4 demonstrates its expressiveness by typing the examples 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. Section 6 shows one interesting extension of our typing system, extending it to the distributed version of $\lambda\pi_{\mathrm{v}}$, discussed previously in [32]. Finally Section 7 concludes the paper with a discussion of 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 $\lambda\pi_{\mathrm{v}}$.

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

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

(Abbreviation)

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

(Environment)

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

FIGURE 1. Types

TYPES The collection of types we use is a straightforward extension of that from [32]; a process type, ranged over by π, can either be the constant $\mathtt{proc}$, as in [32], or take the form $[\Delta]$, where Δ is an *environment*. The formal definition is given in Figure 1; this assumes a set of base types such as $\mathtt{unit}$ and $\mathtt{nat}$, an infinite set of channel or resource *names* $\mathbf{N}$, ranged over by $a, b, \ldots$, and an infinite set of *variables* $\mathbf{V}$, ranged over by $x, y, \ldots$ For the sake of clarity we will sometimes use $X, Y, \ldots$ as variables, whenever we intend them to be substituted specifically by higher order values rather than channels.

Channel types are as in [32], in turn an elaboration of the IO-types of [13, 22]; they take the form $\langle S_{\mathtt{I}}, S_{\mathtt{O}} \rangle$, a pair consisting of an *input sort* $S_{\mathtt{I}}$ and an *output sort* $S_{\mathtt{O}}$; these input/output sorts are in turn either a general value type or $\top$, denoting the highest capability, or $\bot$, denoting the lowest; as explained in [32] the representation of IO-types as a tuple makes the integration with the arrow types of the λ-calculus more natural. Moreover the IO-types of [22] can also be represented as a special case of our IO-types, using the abbreviations given in Figure 1.

There are three kinds of value types: base types, channel types as already explained or HO-value types, ranged over by σ_H. These can be formed using either of the functional type constructors, $\sigma_H \to \rho$ or $(x\!:\!\sigma) \to \rho$, where ρ in turn is either a HO-value type or a process type. As already explained process types can either be the constant $\mathtt{proc}$, (also denoted *ok* in [22]) or a type environment $[\Delta]$ where Δ is a mapping from $\mathbf{N} \cup \mathbf{V}$ to channel types; the formation rules for environments are also given in Figure 1.

EXAMPLE 2.1. (Types)

(Term)

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

(Identifier)

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

(Value)

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

(**Literal**)

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

(**Abbreviations**)

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

FIGURE 2. Syntax

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

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

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

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

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

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 $\lambda\pi_{\mathrm{v}}$ is given in Figure 2. It is essentially the same as that used in [32] except that we use the more expressive types, from Figure 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

(Reduction)

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

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

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

(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 $a \notin \mathsf{fn}(\sigma')$ and $b \notin \mathsf{fn}(\sigma)$

FIGURE 3. Reduction

example ignoring trailing occurrences of the empty process $\mathbf{0}$ and omitting type annotations unless they are relevant.

Nevertheless the use of more expressive types changes considerably the nature of the language as both resource names and resource variables may appear in types. This complicates the notions of free names 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 Figure 13, which is relegated to the appendix.

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

$$P \longrightarrow Q$$

between programs and follows the standard approach from [19, 22, 26]; the formal definition is given in Figure 3 and should be understandable to those familiar with either the π-calculus or the λ-calculus. It uses the standard structural equivalence $\equiv$ of the π-calculus; the axioms for $\equiv$ is also given in Figure 3 and it is also assumed to be preserved by the parallel composition " $\mid$ " and restriction operator "ν". It has an associated reduction rule (str). We also use $\twoheadrightarrow$ to denote the multi-step reduction. There are also contains the standard congruence rules for reduction, (app), (par), and (res). However the main rules are β-reduction,

(β), and communication (com). Both these require a definition of substitution of values for variable $P\{V/x\}$, which we have yet to define. Complications arise when the value to be substituted V is a channel name and is best explained with an example:

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

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

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

which is not even a program; it contains an occurrence of the free variable x.

The proper definition of reduction requires that b is also substituted into the types occurring in the body of the function, to give the program

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

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

The formal definition of value substitution into terms, $P\{V/x\}$, is defined inductively on the structure of terms. We only show one instance, the input process, in Figure 4; the remainder can be inferred by the reader. However this instance uses the substitution of values into types, $\rho\{V/x\}$. If V is anything other than a channel name or variable this is the identity. So we need only define the substitution $\rho\{v/x\}$, where v is a channel or channel variable; this is also defined in Figure 4. We will see later that when these substitutions are made to well-defined terms and types, in well-defined environments, the results will also be well-defined. Note also that substitution into types is essentially different from that employed in the standard polymorphic λ-calculus [6] and the polymorphic π-calculus [23]; there type variables are instantiated by *types* (say `int`) whereas here channel variables occurring in types are instantiated by *channels*, not by types.

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

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

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

(Terms)

$$(u?(x_1:\tau_1,\ldots x_n:\tau_n).P)\{V/x\} \;=\; u\{V/x\}?(x_1:\tau_1\{V/x\},\ldots x_n:\tau_n\{V/x\}).P\{V/x\} \text{ with } x\neq x_i$$

$$\vdots$$

(Types) $\top\{v/x\}=\top,\ \bot\{v/x\}=\bot,\ \sigma_G\{v/x\}=\sigma_G,\ \mathtt{proc}\{v/x\}=\mathtt{proc}$

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

FIGURE 4. Name Substitution into Terms and Types

should be $\langle a:(\mathtt{int})^{\mathtt{IO}}\rangle$. This is reflected in the final clause in Figure 4:

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

Here $\sqcup$ is an operator on types which intuitively acts like a (partial) least upper bound with respect to, yet to be defined, a subtyping order on types. The formal definition of $\sqcup$ and the associated greatest lower bound operator $\sqcap$ is given in Figure 14 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 rights of two processes.

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

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

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

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 apriori 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. [32]) 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

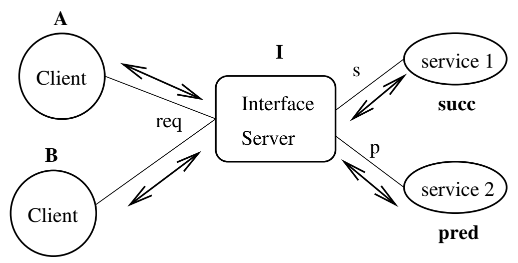

FIGURE 5. Interface Server and Distributed Services

use integers as data and assume that we have a literal, such as $\mathtt{succ}$, representing the successor function, for the operation. Then for a given name a, let $\mathbf{Succ}(a)$ represent the process $*a?(y,z)\ z!\langle\mathtt{succ}(y)\rangle$, which we write as

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

This represents a service (for $\mathtt{succ}$) 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 $\lambda(x)\ \mathbf{Succ}(x)$ 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:

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

Now $\mathbf{sServ}(\mathrm{req}) \,|\, \mathbf{Client}(\mathrm{req})$ is reduced as:

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

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 former 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 5. 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 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, $\mathtt{succ}$ as before and a similar one for the predecessor function, $\mathtt{pred}$:

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

The server may then be defined by:

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

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

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

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

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

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

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

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

3 The Fine-Grained Typing System

3.1 Well-formed Types and Environments

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

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)

$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \top{:}\mathtt{tp},\ \bot{:}\mathtt{tp},\ \sigma_G{:}\mathtt{tp},\ \mathtt{proc}{:}\mathtt{tp},\ [\,]{:}\mathtt{tp}} \qquad \text{(t-sort)}\ \frac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash (\tau_1,\ldots,\tau_n){:}\mathtt{tp}}$$

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

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

Subtyping

(s-id) (s-sort) (s-base)

$$\frac{\Gamma \vdash \rho : \mathtt{tp}}{\Gamma \vdash \rho \le \rho}\ \text{etc.} \qquad \frac{\Gamma \vdash \tau_i : \mathtt{tp}}{\Gamma \vdash \bot \le (\tau_1,\ldots,\tau_n) \le \top} \qquad \frac{\Gamma \vdash [\Delta]{:}\mathtt{tp}}{\Gamma \vdash [\Delta] \le \mathtt{proc}}$$

(s-abs$_H$) (s-abs$_N$)

$$\frac{\Gamma \vdash \sigma'_H \le \sigma_H, \quad \rho \le \rho'}{\Gamma \vdash \sigma_H \to \rho \le \sigma'_H \to \rho'} \qquad \frac{\Gamma \vdash \sigma_2 \le \sigma_1 \quad \Gamma, x{:}\sigma_1 \vdash \rho_1 \le \rho_2}{\Gamma \vdash (x{:}\sigma_1) \to \rho_1 \le (x{:}\sigma_2) \to \rho_2}$$

(s-chan) (s-proc)

$$\frac{\begin{array}{c}\Gamma \vdash S_{\mathtt{I}1} \le S_{\mathtt{I}2}, \ S_{\mathtt{O}2} \le S_{\mathtt{O}1} \\ \Gamma \vdash \langle S_{\mathtt{I}i}, S_{\mathtt{O}i} \rangle : \mathtt{tp} \quad (i = 1,2)\end{array}}{\Gamma \vdash \langle S_{\mathtt{I}1}, S_{\mathtt{O}1} \rangle \le \langle S_{\mathtt{I}2}, S_{\mathtt{O}2} \rangle} \qquad \frac{\begin{array}{c}\Gamma \vdash [\Delta_1] : \mathtt{tp} \\ \forall u \in \mathsf{dom}(\Delta_2).\ \Gamma \vdash \Delta_1(u) \le \Delta_2(u)\end{array}}{\Gamma \vdash [\Delta_2] \le [\Delta_1]}$$

FIGURE 6. Well-formed Types and Subtyping

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

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

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

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

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

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

will not necessarily imply

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

When constructing well-formed environments only types which are currently well-formed may be used. This is the purpose of the second form of judgement. So for example we can not deduce

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

To do so we would need to be able to deduce

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

This in turn is not possible, basically because y is not in the domain of Γ.
The rules for valid type formation are very straightforward; one is only constrained to use identifiers which are already declared in the current environment. In (t-chan), the condition $S_{\mathtt{I}} \ge S_{\mathtt{O}}$ is necessary to ensure that readers of a channel always receive at most the capabilities given by a sender. More details may be found in [13, 32]. There are only two novelties. In the formation rule for dependent types, (t-abs$_N$) the bound variable x is allowed to be used in the construction of the result type ρ. Secondly the rule (t-proc) ensures a process always has a type Δ which does not exceed the current environment Γ.

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

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

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

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

The rules for subtyping are a straightforward extension of those given in [32, 22, 13], apart from the necessity to only use identifiers declared in the current environment. Both function types are contravariant in their first arguments and covariant in their second. Similarly in (s-chan) channel types are covariant in the input capability and contravariant in the output. Again the only real novelty is the sub-typing rule for process types, (s-proc); this means the the ordering of process types is *contravariant* w.r.t. the ordering of [32, 22].

There is almost an endless series of consistency lemmas which one may prove about this system of judgements. Here we give a brief representative list and we leave the proofs to the reader. They are invariably deduced by induction on the derivations in the standard manner. Remember informally $X, Y, Z, \ldots$ denote variables with higher-order types, as opposed to channel types.

LEMMA 3.1.

(1) (Renaming) *Suppose* $u \notin \mathsf{fv}(\Gamma, v, \tau, \Gamma')$. Then we have $\Gamma,\ u{:}\tau,\ \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma,\ v{:}\tau,\ \Gamma'\{v/u\} \vdash \mathtt{J}\{v/u\}$.

(2) (Implied judgement) $\Gamma,\ \Gamma' \vdash \mathtt{J}$ *implies* $\Gamma \vdash \mathtt{Env}$ *and* $\Gamma,\ u{:}\tau,\ \Gamma' \vdash \mathtt{Env}$ *implies* $\Gamma \vdash \tau : \mathtt{tp}$.

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

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

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

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

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

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

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

Note that in general we can not replace $X : \sigma_H$ with $u{:}\sigma$ in the statements (3) and (8) above since the channel u may appear freely in Γ and $\mathtt{J}$. This underlines the major technical difference between our system of types and those in the system $F_{<:}$, [6]. 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 $F_{<:}$ [6], 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 3.2. (FBC, cf. [13, 32]) We say that a partial order $(\mathbf{S}, \sqsubseteq)$ is *finite bounded complete* (FBC) for every finite nonempty subset $S \leq \mathbf{S}$, if S has a lower bound then S has a greatest lower bound. □

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

PROOF. Reflexivity ($\Gamma \vdash \alpha \leq \alpha$), anti-symmetry ($\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \beta \leq \alpha$ implies $\alpha = \beta$), and transitivity, ($\Gamma \vdash \alpha \leq \beta$ and $\Gamma \vdash \beta \leq \gamma$ implies $\Gamma \vdash \alpha \leq \gamma$) are straightforward inductions on type inference. As an example we consider one case of transitivity, when α is a process type. If this is $\mathtt{proc}$ then one can show, again by induction on type inference that β, and therefore γ, is also $\mathtt{proc}$. So without loss of generality we can assume that these are of the form $[\Delta_1], [\Delta_2], [\Delta_3]$ respectively. Then by (Implied judgement), Lemma 3.1, $\Gamma \vdash [\Delta_{1,2,3}] : \mathtt{tp}$, then by (t-proc) and (s-proc), for all $u \in \mathsf{dom}(\Delta_1)$, we have $\Gamma \vdash \Delta(u) \leq \Delta_3(u) \leq \Delta_2(u) \leq \Delta_1(u)$, hence by induction, we have $\Gamma \vdash [\Delta_1] \leq [\Delta_3]$.

Now, as discussed in Definition 4.1 in [13], we have to show that there exist a partial binary meet operator $\sqcap$ and a partial join operator $\sqcup$ which 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	$\Gamma \vdash \beta \leq \beta \sqcup \gamma$		

Parts (A) and (B) are proved simultaneously, by induction on the combined length of the proof derivations. We consider here only one of 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 $\Gamma \vdash \Delta_i(u) \leq \langle \top, \bot \rangle$ 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 $\mathsf{dom}(\Delta_1) - \mathsf{dom}(\Delta_2)$ or $\mathsf{dom}(\Delta_2) - \mathsf{dom}(\Delta_1)$. 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 $\sqcup$. The detailed proof is left to the reader. □

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

COROLLARY 3.4. (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 3.5. (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 $\Gamma, x\!:\!\sigma, \Gamma' \vdash \mathtt{J}$. For the most part this proceeds as a standard Substitution Lemma for a typing system (see [32]). But because names may now appear in types there some novel cases. Here we examine one, when the judgement has the form $\Gamma, x:\sigma, \Gamma' \vdash [\Delta] : \mathtt{tp}$. This is only interesting 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 $x \notin \mathsf{fn}(\sigma, \Gamma(u))$ by the formation rule of well-formed environment. Hence we have $\sigma\{u/x\} = \sigma$ and $\Gamma(u)\{u/x\} = \Gamma(u)$. Then by applying the inductive hypothesis to the above sequents, respectively, 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

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

Now by transitivity, $(\star\star)$ and $(*)$ 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 $v_i \in \mathsf{dom}(\Delta) - \{u, x\}$, by the inductive hypothesis, we also have:

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

Hence

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

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

3.2 Type Inference

The typing system is given in Figure 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 (SUB$_N$).

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

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

(Common)

$$(\textsc{Val})\ \frac{\vdash \Gamma, u\!:\!\tau, \Gamma' : \mathtt{Env}}{\Gamma, u\!:\!\tau, \Gamma' \vdash u : \tau} \qquad (\textsc{Con})\ \frac{\vdash \Gamma : \mathtt{Env}}{\Gamma \vdash 1 : \mathtt{nat}} \quad \text{etc.}$$

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

(Function)

$$(\textsc{Abs}_H)\ \frac{\Gamma, X\!:\!\sigma_H \vdash P : \rho}{\Gamma \vdash \lambda(X\!:\!\sigma_H)P : \sigma_H \to \rho} \qquad (\textsc{App}_H)\ \frac{\Gamma \vdash P : \sigma_H \to \rho \quad \Gamma \vdash Q : \sigma_H}{\Gamma \vdash PQ : \rho}$$

$$(\textsc{Abs}_N)\ \frac{\Gamma, x\!:\!\sigma \vdash P : \rho}{\Gamma \vdash \lambda(x\!:\!\sigma)P : (x\!:\!\sigma) \to \rho} \qquad (\textsc{App}_N)\ \frac{\Gamma \vdash P : (x\!:\!\sigma) \to \rho \quad \Gamma \vdash u : \sigma}{\Gamma \vdash Pu : \rho\{u/x\}}$$

(Process)

$$(\textsc{Nil})\ \frac{\vdash \Gamma : \mathtt{Env}}{\Gamma \vdash \mathbf{0} : [\,]} \qquad (\textsc{Par})\ \frac{\Gamma \vdash P_{1,2} : \pi}{\Gamma \vdash P_1 \,|\, P_2 : \pi} \qquad (\textsc{Rep})\ \frac{\Gamma \vdash P : \pi}{\Gamma \vdash *P : \pi} \qquad (\textsc{Res})\ \frac{\Gamma, a\!:\!\sigma \vdash P : \pi}{\Gamma \vdash (\nu a\!:\!\sigma)P : \pi / a}$$

$$(\textsc{Out})\ \frac{\begin{array}{l}\pi \vdash_\Gamma u : (\tau_1, \ldots, \tau_n)^{\mathtt{O}} \qquad \Gamma \vdash P : \pi \\ \Gamma \vdash V_i : \tau_i \quad \tau_i = \sigma_i \Rightarrow \pi \vdash_\Gamma V_i : \sigma_i\end{array}}{\Gamma \vdash u!\langle V_1, \ldots, V_n \rangle P : \pi} \qquad (\textsc{In})\ \frac{\begin{array}{l}\pi \vdash_\Gamma u : (\tau_1, \ldots, \tau_n)^{\mathtt{I}} \\ \Gamma, x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n \vdash P : \pi, x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n\end{array}}{\Gamma \vdash u?(x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n)P : \pi}$$

FIGURE 7. Typing System for $\lambda\pi_v$

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

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

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

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

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 Figure 4. 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 [22, 13, 32]. However many of the rules are sufficient novel to warrant detailed explanation.

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

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

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

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

A meta-result about our typing system ensures that, from the hypotheses we can conclude $\Gamma \vdash \pi_i : \mathtt{tp}$. Thus from Corollary 3.4 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 ($\textsc{Par}$) justifies this derived rule.

THE OUTPUT RULE, ($\textsc{Out}$): Under what circumstances can we conclude $\Gamma \vdash a!\langle V \rangle P : \pi$? We require at least the following:

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

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

$$\Gamma \vdash [b\!:\!\sigma] \leq \pi$$

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

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

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

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

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

and therefore

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

THE INPUT RULE, (IN): The rule for prefixing is a straightforward generalisation of that in [32]:

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

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

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

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

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

Note first in the above definition "$\pi, x : \tau$" denotes "π" if τ is not a channel type. In (IN), by the first 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 $[\Delta], x{:}\sigma \overset{\text{def}}{=} [\Delta, x{:}\sigma]$ in the second assumption.

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

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

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, $(\texttt{int})^{\texttt{O}}$. Then we have

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

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

FIGURE 8. Name Erasing from Types

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

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

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

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

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

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

LEMMA 3.6. (Channel narrowing) $\Gamma, a{:}\sigma, \Gamma' \vdash J$ *implies* $\Gamma, \Gamma'/a \vdash J/a$. $\square$

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

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. § 2. CHANNEL ABSTRACTION) As a first example let us consider the simple forwarder, already discussed in the Introduction, this time with type annotations:

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

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:

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

Let us now see how we can use this typing to assign a type to the process R, 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

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

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

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

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 slight modified environment, where the capabilities of a and b are interchanged. □

EXAMPLE 4.2. (Typed compute server) We now revisit Example 2.2. In the definition of **Succ**(a), a pair of values for input on a, an integer to be treated and

a channel, on which the resulting integer is returned. So we annotate the code with types as follows:

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

Note the 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:

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

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, $(x{:}\sigma_s^{\mathtt{I}}) \to [x{:}\sigma_s^{\mathtt{I}}]$.

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 $(\ X a\ |\, a!\langle 1, c_1\rangle \,|\, \cdots)$, 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

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

However we require a more general environment. Let Δ denote Δ_c, req$:((\tau_\lambda)^{\mathtt{O}})^{\mathtt{IO}}$. Then we can also derive

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

An application of (RES) now gives

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

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

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

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

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

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

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

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

Hence applying $\textsc{Sub}_N$ to **sServ**(req) and **Client**(req), we have:

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

Observe that the type $\tau_\lambda \stackrel{\mathrm{def}}{=} (x:\sigma_s^{\mathtt{I}}) \to [x:\sigma_s^{\mathtt{I}}]$ which annotates X 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 [32, 22, 9], then the client could input any function $\lambda(x:\sigma)Q$, where Q an arbitrary process; such incoming code may harm the client's resources. □

EXAMPLE 4.3. (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 returns 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:

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

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.

Using subsumption we can form the judgement

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

where P_A denotes the body of the script sent by $\mathbf{Client}_A$, namely

$$\lambda(s,p)\ ((\nu c)s!\langle k,c\rangle c?(z)\ s!\langle z,c\rangle \mathtt{Fw}(cr_A))$$

By channel abstraction we therefore have

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

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

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

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

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 send capabilities on the return channels. □

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 have 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* $\Gamma \vdash \alpha:\mathtt{tp}$.

(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* $\Gamma, \Gamma'/a \vdash P:\alpha/a$.

(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 B. □

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; again it may be viewed as a generalisation of Lemma 3.5:

LEMMA 5.2. (Substitution Lemma) *Assume* $\Gamma \vdash V:\tau$. *Then* $\Gamma, x:\tau, \Gamma' \vdash P:\alpha$ *implies* $\Gamma, \Gamma'\{V/x\} \vdash P\{V/x\} : \alpha\{V/x\}$.

PROOF. Note that if x is a higher-order variable (that is a variable which does not have a channel type) then $\Gamma'\{V/x\}$ and $\alpha\{V/x\}$ are simply Γ' and α respectively; in this case the proof is straightforward as it is similar to the corresponding proofs in [30, 22, 13]. However when it is a channel variable this substitution, of a channel name for a channel variable, may change the structure of α and the proof is more delicate. See Appendix B for details. □

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 5.3. $\Gamma \vdash P:\pi$ *and* $P \equiv P'$ *imply* $\Gamma \vdash P':\pi$.

PROOF. By induction on the derivation of $P \equiv Q$. As an example we examine 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 $\pi'/a = \pi$. 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 $\Gamma \vdash \pi'/a \leq \pi'$ (for example if we let π' denote $[b{:}\langle a{:}\sigma\rangle^{\mathtt{IO}}]$ it is easy to find an instance of Γ, Q such that $\Gamma, a{:}\sigma \vdash \pi'$ but $\Gamma \not\vdash Q:\pi'$).

However, by Corollary 3.4 we know that $\pi'/a \sqcup \pi'$ is always defined, and we have $\pi' \leq (\pi'/a \sqcup \pi')$ and $\pi'/a \leq (\pi'/a \sqcup \pi')$. So we may apply subsumption and the rule PAR to obtain $\Gamma, a{:}\sigma \vdash P \,|\, Q : (\pi'/a \sqcup \pi')$. A simple calculation gives $(\pi'/a \sqcup \pi')/a = \pi'/a = \pi$ 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. □

THEOREM 5.4. **(Subject Reduction)**

If $\Gamma \vdash P:\rho$ *and* $P \longrightarrow P'$, *then* $\Gamma \vdash P':\rho$.

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

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

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

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 (ABS$_N$). Then we have that σ and σ' coincide and $\Gamma, x:\sigma \vdash P:\rho$. 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 (SUB$_H$). In this case we know $\Gamma \vdash \lambda(x{:}\sigma)P : (x{:}\sigma'') \rightarrow \rho'$ and $\Gamma \vdash (x{:}\sigma'') \rightarrow \rho' \leq (x{:}\sigma') \rightarrow \rho$. From the latter we can calculate $\Gamma \vdash \sigma' \leq \sigma''$ and therefore by subsumption we have $\Gamma \vdash v:\sigma''$. 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 3.5 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, $\Gamma \vdash (\lambda(x{:}\sigma)P)\,v : \rho$. 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 ABS$_N$. Here we know $\Gamma \vdash \lambda(x{:}\sigma)P : (x{:}\sigma') \rightarrow \rho'$, $\Gamma \vdash v:\sigma'$ 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. Moreover we assume the value sent is a channel; for other values the calculations are simpler. So we are examining the case

$$a?(x{:}\sigma)\; P \,|\, a!\langle v\rangle Q \longrightarrow P\{v/x\} \,|\, Q$$

and the hypothesis is $\Gamma \vdash a?(x{:}\sigma)\; P \,|\, a!\langle v\rangle Q : \pi$. 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 `proc` the prove follows standard lines. So we assume it has the form $[\Delta]$.

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

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

FIGURE 9. Run-time errors

Analysing the hypothesis we obtain

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

Noting $x \notin \mathsf{fv}(\sigma)$, we can apply (Channel narrowing), Lemma 3.6, to obtain $\Gamma \vdash [u:(\sigma)^{\mathtt{I}}] \leq [\Delta_1]$. Then we have: $\Gamma \vdash \Gamma(u) \leq \Delta(u) \leq \Delta_1(u) \leq (\sigma)^{\mathtt{I}}$ and $\Gamma \vdash \Gamma(u) \leq \Delta(u) \leq \Delta_2(u) \leq (\sigma')^{\mathtt{O}}$, 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 $\Gamma \vdash P\{v/x\} : [\Delta]$. □

5.2 *Type Safety*

Out typing system is an extension of that for the λ-calculus from [10] and that for the π-calculus from [22]; 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 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 Figure 9, using a unary predicate $P \xrightarrow{\Gamma,\pi}_{err}$. The first two clauses are the most significant. The first says that, relative to Γ, P violates the interface π if it can input on the channel a but the interface π does not assign any input capability to a; the second is similar, but for output. Combining these rules, we can also derive the following communication runtime error between input and output processes:

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

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

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

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

THEOREM 5.5. **(Type Safety)** *If* $\Gamma \vdash P:\pi$ *then* $P \not\xrightarrow{\Gamma,\pi}_{err}$

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

6 Extensions

In this section we briefly indicate some further uses of our fine grained process types.

6.1 *Distributed Higher-Order π-calculus*

In this subsection, we show that our typing system can be used to ensure various kinds of *host security*; that is, protecting hosts from untrusted imported code. To discuss this issue more explicitly, we add simple distributed primitives to the core calculus, as in [32]. The term $N \parallel M$ represents two systems N, M running at two physically distinct locations, while the process $\mathtt{Spawn}(P)$ creates a new location at which the process P is launched. The details are in Figure 10.

The reduction semantics is extended straightforwardly; again this is outlined in Figure 10. The first two rules are the most important, namely spawning of a process at a new location (spawn) and communication between physically distinct locations (com$_s$). The additional structural equivalence of systems is defined by changing "$|$" to "$\parallel$" and P, Q, R to M, N, N' in Figure 3.

EXAMPLE 6.1. Consider the following system:

$$a?(X)\, X c \;\parallel\; (b?(z)\, Q \,|\, a!\langle \lambda y. y!\langle b?(z)\, Q\rangle\rangle \,|\, c?(X)\; run\, X)$$

It consists of two processes, residing at two distinct sites; the first services the resource a while the second services both b and c. Note that the second location has three distinct threads. One of these can send a process abstraction to the other physical location, the site servicing a. This instantiates the abstraction, which results in a thunked process being immediately returned to the original location, where it is executed. □

Syntax: others from Figure 2.

System:	$M, N, \ldots ::=$	$P \mid N \parallel M \mid (\nu a{:}\sigma)N \mid \mathbf{0}$	
Term:	$P, Q, \ldots ::=$	$\mathtt{Spawn}(P) \mid \cdots$	as in Figure 2

Distributed Reduction Rules:

$$(\text{spawn}) \quad (\cdots Q \mid \mathtt{Spawn}(P)) \longrightarrow (\cdots Q) \parallel P$$

$$(\text{com}_s) \quad (u?(x_1{:}\tau_1, \ldots, x_n{:}\tau_n)P \mid \cdots) \parallel (u!\langle V_1, \ldots, V_n\rangle Q \mid \cdots) \longrightarrow (P\{V_1, \ldots, V_n/x_1, \ldots, x_n\} \mid \cdots) \parallel (Q \mid \cdots)$$

$$(\text{par}_s)\ \frac{M \longrightarrow M'}{M \parallel N \longrightarrow M' \parallel N} \qquad (\text{res}_s)\ \frac{N \longrightarrow N'}{(\nu a{:}\sigma)N \longrightarrow (\nu a{:}\sigma)N'}$$

$$(\text{str}_s)\ \frac{N \equiv N' \longrightarrow M' \equiv M}{N \longrightarrow M}$$

Distributed Typing Rules:

$$(\text{SPA})\ \frac{\Gamma \vdash P : \pi}{\Gamma \vdash \mathtt{Spawn}(P) : \pi} \qquad (\text{PAR}_s)\ \frac{\Gamma_{1,2} \vdash N_{1,2} : \pi_{1,2} \qquad \Gamma_1 \asymp \Gamma_2}{\Gamma_1 \sqcap \Gamma_2 \vdash N_1 \parallel N_2 : \pi_1 \sqcup \pi_2}$$

$$(\text{RES}_s)\ \frac{\Gamma, a{:}\sigma \vdash M : \pi}{\Gamma \vdash (\nu a{:}\sigma)M : \pi/a}$$

FIGURE 10. Syntax and Distributed Reduction, and Distributed Typing Rules

EXAMPLE 6.2. The following process can be considered to be a simple form of *compute server*:

$$*\text{req}?(X{:}\langle a{:}(\mathtt{int})^{\mathtt{O}}, b{:}(\mathtt{bool})^{\mathtt{IO}}\rangle)\ \mathtt{Spawn}(\mathit{run}\ X)$$

It repeatedly receives thunked processes and sets them running at independent locations. However the input type ensures that the spawned process will have limited capabilities; namely it can at most output integers to channel a and send/receive booleans on channel b. □

Our type inference can be extended to these distributed systems using the approach of local type-checking from [32]. The judgements take the form $\Gamma \vdash N : \pi$ and they use the previous judgements, from Section 3, for processes. The main rule is (PAR$_s$), where we use $\Gamma_1 \asymp \Gamma_2$ to denote that $\Gamma_1 \sqcap \Gamma_2$ is defined. Note that, according to this rule subsystems can be typechecked independently. Only when they are composed do we need to check the compatibility of their use of resources.

We leave the reader to check:

PROPOSITION 6.3. (Subject reduction) *If* $\Gamma \vdash N : \pi$ *and* $N \longrightarrow\!\!\!\!\rightarrow M$ *then* $\Gamma \vdash M : \pi$. □

Input Predicate:

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

$$\frac{N \downarrow a^{\mathtt{I}} \text{ or } M \downarrow a^{\mathtt{I}}}{(N \parallel M) \downarrow a^{\mathtt{I}}} \qquad \frac{N \downarrow a^{\mathtt{I}} \quad a \neq c}{(\nu c)N \downarrow a^{\mathtt{I}}}$$

Locality Error:

$$\frac{N \downarrow a^{\mathtt{I}} \quad M \downarrow a^{\mathtt{I}}}{(N \parallel M) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(N \parallel M) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(M \parallel N) \xrightarrow{lerr}} \qquad \frac{N \xrightarrow{lerr}}{(\nu c)N \xrightarrow{lerr}}$$

FIGURE 11. Locality Error

Note that here the judgements for systems could be rendered simply as $\Gamma \vdash N : \mathtt{sys}$, indicating that the system N is well-typed with respect to Γ; the more discriminating types $[\Delta]$ are not used for systems. However in the next subsection they will play a role.

6.2 Locality in Distributed Higher Order Processes

Here we show that our typing system can easily be modified to ensure that systems preserve an important global invariance property, namely *channel locality*. In [32] this required a complicated type system involving *sendable types*; here the presence of fine-grained process types makes the modifications much more straightforward.

The locality of channels means that every input channel is associated with a *unique* location, which is a desirable constraint when we regard a receptor as a resource or an object existing in the unique name space (cf. [3, 8, 15]). This constraint is violated in, for example,

$$a?(y)\ P \ \parallel\ (a?(z)\ Q \mid b?(x_1)\ R_1 \mid b?(x_2)\ R_2)$$

because the name a can receive input at two distinct locations. Note however that the name b is located uniquely, although at that location a call can be serviced in two different ways. Also in the system discussed in Example 6.1 all channels are uniquely located.

A formal definition of this concept, or rather its negation *locality error*, is given in Figure 11, using a predicate on systems, $N \xrightarrow{lerr}$. Intuitively this means that in the system N there is a runtime error, namely there is some channel a which is ready to receive input at two distinct locations. The definition uses an input predicate $P \downarrow a^{\mathtt{I}}$ which means P can immediately perform input on name a.

(PAR$_l$)

$$\frac{\Gamma_i \vdash_{\mathtt{l}} N_i : [\Delta_i] \quad (i=1,2) \qquad \Gamma_1 \asymp \Gamma_2 \quad \Delta_1 \asymp_{\mathtt{l}} \Delta_2}{\Gamma_1 \sqcap \Gamma_2 \vdash_{\mathtt{l}} N_1 \parallel N_2 : [\Delta_1] \sqcup [\Delta_2]}$$

(SPA$_l$)

$$\frac{\Gamma \vdash_{\mathtt{l}} P : [u_1 : S_1^{\mathtt{O}}, \ldots, u_n : S_n^{\mathtt{O}}]}{\Gamma \vdash_{\mathtt{l}} \mathtt{Spawn}(P) : [u_1 : S_1^{\mathtt{O}}, \ldots, u_n : S_n^{\mathtt{O}}]}$$

(OUT$_d$)

$$\frac{\pi \vdash_\Gamma u : (\tau_1, \ldots, \tau_n)^{\mathtt{O}} \quad \Gamma \vdash_{\mathtt{l}} P : \pi \quad \Gamma \vdash_{\mathtt{l}} V_i : \tau_i \quad \tau_i = \sigma_i \;\Rightarrow\; \sigma_i = S_i^{\mathtt{O}} \wedge \pi \vdash_\Gamma V_i : S_i^{\mathtt{O}}}{\Gamma \vdash_{\mathtt{l}} u!\langle V_1, \ldots, V_n \rangle P : \pi}$$

(OUT$_l$)

$$\frac{\pi \vdash_\Gamma u : \langle (\tau_1', \ldots, \tau_n'), (\tau_1, \ldots, \tau_n) \rangle \quad \Gamma \vdash_{\mathtt{l}} P : \pi \quad \Gamma \vdash_{\mathtt{l}} V_i : \tau_i \quad \tau_i = \sigma_i \;\Rightarrow\; \pi \vdash_\Gamma V_i : \tau_i}{\Gamma \vdash_{\mathtt{l}} u!\langle V_1, \ldots, V_n \rangle P : \pi}$$

Other rules are from Figures 7 and 10, replacing $\vdash$ with $\vdash_{\mathtt{l}}$.

FIGURE 12. Locality Typing System

A typing system which ensures the absence of such locality errors is given in Figure 12. For processes we have to change the channel output rules , to (OUT$_d$) and (OUT$_l$). The former ensures that channels are only distributed to different locations with output capabilities while the latter allows their transmission to the current location with more capabilities. Similarly spawning a process is only allowed if it contains no input capabilities. However the main rule is for the parallel composition of systems, (PAR$_s$) , when they are checked for composability. In this rule $\Delta_1 \asymp_{\mathtt{l}} \Delta_2$ means

$$\text{if } a : \langle S_{\mathtt{I}i}, S_{\mathtt{O}i} \rangle \in \Delta_i \text{ with } i = 1,2, \text{ then either } S_{\mathtt{I}i} = \top \text{ or } S_{\mathtt{I}i} = \top$$

If $N_1 \parallel N_2$ is well-typed and N_1 uses an input capability at a, then this capability will occur in the interface of N_1, namely $[\Delta_i]$. So the constraint $\Delta_1 \asymp_{\mathtt{l}} \Delta_2$ ensures that N_2 is not allowed to use a as an active input point.

This typing system is considerably simpler than that in [32]; it is not necessary to introduce additional typing constructs such as *sendability*. However it is also more powerful. The process

$$a?(X)\; run X \mid b?(x)\, R \;\parallel\; a!\langle b?(x)\, Q \rangle$$

can be typed with the rules from Figure 12 but are not typable in [32]; similarly for the system discussed in Example 6.1. These examples emphasise that run-time errors, and mistypings, only occur the same input name is *actually used* in two different locations.

THEOREM 6.4.

(1) **(Subject Reduction for Locality)**
If $\Gamma \vdash_{\mathtt{l}} N : [\Delta]$ and $N \longrightarrow\!\!\!\!\to M$, then $\Gamma \vdash_{\mathtt{l}} M : [\Delta]$.

(2) **(Locality Safe)** $\Gamma \vdash_{\mathtt{l}} N : [\Delta]$ and $N \longrightarrow\!\!\!\!\to M$ imply $M \overset{lerr}{\not\longrightarrow}$.

PROOF. For (1), we only have to consider the transmission of channels to different sites. Th proof is essentially similar to that of Theorem 5.4.

The proof of (2) relies on the fact that $P \downarrow a^{\mathtt{I}}$ and $\Gamma \vdash P : [\Delta]$ imply that a has an input capability in Δ, that is $a : \langle (\tau_1, \ldots, \tau_n), S_{\mathtt{O}} \rangle \in \Delta$ for some τ_i by Theorem 5.5. □

We also remark that the extension to the more general channel constraints studied by [27] is easily achieved by a simple modification of the definition of composability of process types and distributed channel constraints (cf. [32]).

7 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. [14, 30, 22, 23, 32]) and λ-calculus (e.g. [6, 10]).

In the literature on typed process calculi [14, 22, 23, 30], including extensions to mobile processes [9, 8, 25, 13, 27, 32], 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 [20].

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, 30, 22], record types [10, 29], polymorphic types [6, 23], and dynamic types [2, 25]. A particularly useful extension would be *linear/affine types*, as in the π-calculus and the linear λ-calculus [18, 5, 14, 31]. 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, 12] 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 [18, 22, 23, 31]. 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 [26]; we leave this for future work.

TYPE LIMITATIONS One limitation of our typing system is that, while name variables in types can be abstracted by channel dependency types $(x:\sigma) \to \rho$ of the channel λ-abstraction $\lambda(x:\sigma)\,P$, a similar abstraction is not allowed when we bound name variables by input prefix $a?(x:\sigma)\ P$. The result is that there is a loss of information in many of the types we can assign to processes. A typical example is the process $a?(x)\ b!\langle x!\langle v\rangle\rangle$. 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 $\langle \mathtt{proc}\rangle$.

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

$$a?(x)\ b!\langle x!\langle v\rangle\rangle \mid a!\langle c\rangle \mid a!\langle d\rangle$$

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 $[b:\langle c:\sigma\rangle^{\mathtt{o}}]$ or $[b:\langle d:\sigma\rangle^{\mathtt{o}}]$.

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 $[b:\langle\ \rangle^{\mathtt{o}}]$ 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

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

Here essentially the only the possibility for τ is the type $\langle \mathtt{proc}\rangle$.

Here we should be able to say that b can output a (thunked) process which contains some *unknown* channel name of type $(\mathtt{nat})^{\mathtt{o}}$, and the input type associated with b should be able to accommodate such constraints. Some form of existential quantification over types may be appropriate but integrating such a construct into the 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 [10, 6], where type variables play an important role; as with our channel names they may appear, and be bound, both in terms and types. However there is an essential technical difference between our use of channels and the use of type variables in the polymorphic λ-calculus. Name instantiation can result in dynamic changes to the types annotating a term. Names are exchanged as values between processes but they also appear as interaction points in the types of processes. This feature necessitated the development of new concepts of well-formed type, subtyping, well-formed substitution, etc., independent of those developed for the polymorphic λ-calculus.

Pierce and Sangiorgi [23] recently proposed a polymorphic π-calculus with an existential quantifier over types and used a refined typed behavioural equivalence to reason about encodings of the concurrent abstract data types. Since their polymorphic types are based on those of the polymorphic λ-calculus (that is they abstract over *type variables* via the operator ∃), they are quite different from ours. In particular they do not address the issue of assigning fine-grained types to processes.

For sequential computations, Tofte and Talpin have developed the effect typing system [28] and this was recently applied to Facile by Kirh [16]; he showed its usefulness in solving the marshalling problem in [17]. Again his typing system is different from ours since he adds the original effect system to the arrow types; hence all processes have the unique constant type *Unit* and a channel cannot carry nested effects. More precisely, the region types in [28] are used to represent the region allocation of values during β-reduction, while our process types are used to represent interaction effects between concurrent processes. Hence an integration of the effect typing system of the λ-calculus and the IO-subtyping system of the π-calculus would have difficulty in expressing the kind of constraints guaranteed by our typing system.

De Nicola, Ferrari and Pugliese studied a subtyping system for a language based on Linda [7], and showed that it can be used to control the mobility of mobile agents. In their language, each located process is equipped with different capabilities (read, input, our, eval and newloc) rather than the unique process type (i.e. $\mathtt{proc}$), 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.

A Auxiliary Definitions

B Proofs for Section 5

In this appendix we examine some of the results announced, but not proven, in Section 5.

(Free Names)

Terms:

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

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

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

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

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

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

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

Types:

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

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

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

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

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

$\mathsf{fn}([\Delta]) = \{\mathsf{fn}(u) \cup \mathsf{fn}(\sigma) \mid u : \sigma \in \Delta\}$

(Free Variables) $\mathsf{fv}(\alpha)$ and $\mathsf{fv}(P)$ are defined by $\mathsf{fv}(x) = \{x\}$, $\mathsf{fv}(a) = \emptyset$, and other rules replacing $\mathsf{fn}(\ )$ by $\mathsf{fv}(\ )$.

FIGURE 13. Free Names and Variables

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

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 $\lambda(x : \tau)Q$ and $\Gamma \vdash \lambda(x : \tau)Q : \sigma_H \to \rho$ 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 Figure 6, gives $\Gamma \vdash \sigma_H \to \rho$, 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 $\Gamma \vdash Q : (x : \sigma) \to \rho$ and $\Gamma \vdash u : \sigma$. Applying induction to the first inference we obtain the judgement $\Gamma \vdash (x : \sigma) \to \rho : \mathtt{tp}$. In turn by induction on the inference of this judgement one can show that for some σ', ρ',

$$\Gamma, x : \sigma' \vdash \rho' \leq \rho \quad \text{and} \quad \Gamma \vdash \sigma \leq \sigma'$$

An application of subsumption gives $\Gamma \vdash u : \sigma'$ and since this implies $\Gamma \vdash \Gamma(u) \leq$

(o-base) $\quad \alpha \sqcap \alpha \overset{\text{def}}{=} \alpha \sqcup \alpha \overset{\text{def}}{=} \alpha$

(o-$\top, \bot$) $\quad \bot \sqcap S \overset{\text{def}}{=} S \sqcap \bot \overset{\text{def}}{=} \bot, \top \sqcup S \overset{\text{def}}{=} S \sqcup \top \overset{\text{def}}{=} \top,$
$\quad \bot \sqcup S \overset{\text{def}}{=} S \sqcup \bot \overset{\text{def}}{=} \top \sqcap S \overset{\text{def}}{=} S \sqcap \top \overset{\text{def}}{=} S.$

(o-vec) $\quad (\tau_1, .., \tau_n) \sqcap (\tau'_1, .., \tau'_n) \overset{\text{def}}{=} (\tau_1 \sqcap \tau'_1, .., \tau_n \sqcap \tau'_n)$
$\quad (\tau_1, .., \tau_n) \sqcup (\tau'_1, .., \tau'_n) \overset{\text{def}}{=} (\tau_1 \sqcup \tau'_1, .., \tau_n \sqcup \tau'_n)$

(o-abs$_H$) $\quad (\sigma_H \to \rho) \sqcap (\sigma'_H \to \rho') \overset{\text{def}}{=} \sigma_H \sqcup \sigma'_H \to \rho \sqcap \rho'$
$\quad (\sigma_H \to \rho) \sqcup (\sigma'_H \to \rho') \overset{\text{def}}{=} \sigma_H \sqcap \sigma'_H \to \rho \sqcup \rho'$

(o-abs$_N$) $\quad ((x : \sigma) \to \rho) \sqcap ((x : \sigma') \to \rho') \overset{\text{def}}{=} (x : \sigma \sqcup \sigma') \to \rho \sqcap \rho'$
$\quad ((x : \sigma) \to \rho) \sqcup ((x : \sigma') \to \rho') \overset{\text{def}}{=} (x : \sigma \sqcap \sigma') \to \rho \sqcup \rho'$

(o-chan) $\quad \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \sqcup \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle \overset{\text{def}}{=} \langle S_{\mathtt{I}} \sqcup S_{\mathtt{I}}, S_{\mathtt{O}} \sqcap S'_{\mathtt{O}}\rangle$
$\quad \langle S_{\mathtt{I}}, S_{\mathtt{O}}\rangle \sqcap \langle S'_{\mathtt{I}}, S'_{\mathtt{O}}\rangle \overset{\text{def}}{=} \langle S_{\mathtt{I}} \sqcap S'_{\mathtt{I}}, S_{\mathtt{O}} \sqcup S'_{\mathtt{O}}\rangle$
$\qquad$ if $S_{\mathtt{I}} \geq S'_{\mathtt{O}}$ and $S'_{\mathtt{I}} \geq S_{\mathtt{O}}$ else undefined.

(o-cenv) $\quad \Gamma_1 \sqcap \Gamma_2 \overset{\text{def}}{=} \Gamma_1/\mathsf{dom}(\Gamma_2) \cup \Gamma_2/\mathsf{dom}(\Gamma_1)$
$\qquad \cup \{u : (\Gamma_1(u) \sqcap \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$
$\quad \Gamma_1 \sqcup \Gamma_2 \overset{\text{def}}{=} \{u : (\Gamma_1(u) \sqcup \Gamma_2(u)) \mid u \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2)\}$

(o-proc) $\quad [\Delta_1] \sqcap [\Delta_2] \overset{\text{def}}{=} [\Delta_1 \sqcup \Delta_2] \quad [\Delta_1] \sqcup [\Delta_2] \overset{\text{def}}{=} [\Delta_1 \sqcap \Delta_2]$
$\quad \mathtt{proc} \sqcup \pi = \mathtt{proc} \quad \mathtt{proc} \sqcap \pi = \pi$

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 $S \sqcup S' = \top$).

FIGURE 14. Partial Meet and Join Operators

σ' we can apply Name Substitution, Lemma 3.5, to obtain

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

Now by Lemma 3.1 (7) again, we have: $\Gamma \vdash \rho\{u/x\}$, as desired.

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

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 $(x :$

$\sigma) \to \rho$ and we know $\Gamma, a{:}\sigma', \Gamma', x{:}\sigma \vdash Q : \rho$. Then by the inductive hypothesis, we have $\Gamma, \Gamma'/a, x{:}\sigma/a \vdash Q : \rho/a$. However since we are assuming $a \notin \mathsf{fn}(P)$ we know $a \notin \mathsf{fn}(\sigma)$, and hence $\sigma/a = \sigma$. Therefore we have $\Gamma, \Gamma'/a, x{:}\sigma \vdash Q : \rho/a$. We may now apply ($\textsc{Abs}_N$) to obtain

$$\Gamma, \Gamma'/a \vdash \lambda(x{:}\sigma).Q : (x{:}\sigma) \to \rho/a$$

as desired.

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

$$\Gamma, \Gamma'/a \vdash u?(x_1{:}\tau_1,...,x_n{:}\tau_n)Q : \pi/a \qquad \square$$

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

Lemma B.1. (Substitution into channel) *$\Gamma \vdash v : \sigma$ 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: $\Gamma, x{:}\sigma, \Gamma' \vdash \sigma_u \leq \sigma'$ and therefore, by (Name substitution), Lemma 3.5, $\Gamma, \Gamma'\{v/x\} \vdash \sigma_u\{v/x\} \leq \sigma'\{v/x\}$. 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 3.5 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 B.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 why $\Gamma, x{:}\tau, \Gamma' \vdash P : \rho$ and as usual an examination of the last inference rule used. Most cases are straightforward and we outline three examples.

Suppose P has the form $Q\,w$ and the last inference rule used is ($\textsc{App}_N$). Then we have:

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

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

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

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 if $x = w$, then the above sequent is $\Gamma, \Gamma' \vdash V : \sigma$ it follows by inductive hypothesis. Noting $\rho'\{w/y\}\{V/x\} = (\rho'\{V/x\})\{w\{V/x\}/y\}$ because $x \neq y$, and applying ($\textsc{App}_N$) again, we have:

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

as required.

Suppose P is $u!\langle V_1,...,V_n\rangle Q$ and the last rule used is (Out). Then we have: $\Gamma, x{:}\sigma, \Gamma' \vdash Q : \pi$, $\quad \Gamma, x{:}\sigma, \Gamma' \vdash V_i : \tau_i$, $\quad \Gamma, x{:}\sigma, \Gamma' \vdash [u{:}(\tau_1,...,\tau_n)^{\mathtt{O}}] \leq \pi$ 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:

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

Applying (Name substitution), Lemma 3.5, again, we obtain:

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

An application of Out now gives the required result:

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

The case (Res) is also a straightforward use of the inductive hypothesis, noting that $(\pi/a)\{V/x\} = (\pi\{V/x\})/a$ whenever x and a are different. $\square$

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., Boudol, G. and Lhoussain, C., The receptive distributed π-calculus, To appear in Proc. FSTTCS'99, LNCS, Springer-Verlag, 1999.

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

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

[6] Cardelli, L., Martini, S., Mitchell, J. and Scedrov, A., An extension of system F with subtyping, *Proc. TACS'91*, LNCS 526, pp.750–770, Springer-Verlag, 1991.

[7] 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.

[8] 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.

[9] 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.

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

[11] Sun Microsystems Inc., Java home page. http://www.javasoft.com/, 1995.

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

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

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

[15] Honda, K. and Tokoro, M., A Small Calculus for Concurrent Objects, in *OOPS Messenger*, 2(2):50-54, Association for Computing Machinery, 1991.

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

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

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

[19] Milner, R., Parrow, J.G. and Walker, D.J., A Calculus of Mobile Processes. *Information and Computation*, 100(1), pp.1–77, 1992.

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

[21] ObjectSpace Inc. ObjectSpace Voyager. `http://www.objectspace.com/voyager`, 1997.

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

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

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

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

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

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

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

[29] Vasconcelos, V., Typed Concurrent Objects, Proc. ECOOP'94, LNCS, Springer-Verlag, 1994.

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

[31] 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, ECS-LFCS-96-350, 1996.

[32] 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.