Under consideration for publication in Formal Aspects of Computing

SAFEDPI: a language for controlling mobile code

Matthew Hennessy[1], Julian Rathke[1] and Nobuko Yoshida[2]
[1]Department of Informatics
University of Sussex,
Brighton UK
[2]Department of Computing Imperial College,
London, UK[1]

Abstract. SAFEDPI is a distributed version of the PICALCULUS, in which processes are located at dynamically created *sites*. Parametrised code may be sent between sites using so-called *ports*, which are essentially higher-order versions of PICALCULUS communication channels. A host location may protect itself by only accepting code which conforms to a given type associated to the incoming port.

We define a sophisticated static type system for these ports, which restrict the capabilities and access rights of any processes launched by incoming code. Dependent and existential types are used to add flexibility, allowing the behaviour of these launched processes, encoded as *process types*, to depend on the host's instantiation of the incoming code.

We also show that a natural contextually defined behavioural equivalence can be characterised coinductively, using bisimulations based on *typed actions*. The characterisation is based on the idea of knowledge acquisition by a testing environment and makes explicit some of the subtleties of determining equivalence in this language of highly constrained distributed code.

Keywords: distributed calculus, security, static types, behavioural equivalence, bisimulations

1. Introduction

In this paper we elaborate a theory of distributed systems which incorporates resource policies. Our main results are:

[1] The first two authors would like to acknowledge the financial support of the two EU Global computing projects, **Mikado** and **Myths**
Correspondence and offprint requests to: Matthew Hennessy, Department of informatics, University of Sussex, Brighton BN1 9QH, UK. email: matthewh@sussex.ac.uk

- a language for distributed systems in which access to hosts by mobile code is controlled using capability-based types
- a fine-grained type system using novel forms of dependent and existential types which gives hosts considerable flexibility in determining the allowed behaviour of incoming code
- a coinductive characterisation of a natural contextual equivalence, based on the notion of typed actions.

This is developed in terms of an extension of the language DPI, [10, 8, 20, 14], a version of the PICALCULUS, [21], in which processes may migrate between locations, which in turn can be dynamically created. In DPI a typical system takes the form

$$l[\![P]\!] \mid (\mathsf{new}\, e : \mathsf{E})(k[\![Q]\!] \mid l[\![R]\!])$$

where there are two threads P and R running at l and one, Q, running at k. The threads Q and R share the private name e at type E. The threads P , Q, R are similar to processes in the PICALCULUS in that they can receive and send values on local channels; the types of these channels indicate the kind of values which may be transmitted. Locations may be dynamically created. For example in

$$l[\![(\mathsf{newloc}\, k : \mathsf{K})\, \mathsf{with}\, Q \;\; \mathsf{in} \;\; \mathsf{xpt}_1!\langle k\rangle \mid \mathsf{xpt}_2!\langle k\rangle]\!]$$

a new location k is created at type K, the code Q is installed at k and the name of the new location is exported via the channels xpt_i. Location types are similar to record types, their form being

$$\mathsf{loc}[c_1 : \mathsf{C}_1, \ldots c_n : \mathsf{C}_n]$$

This indicates that the channels, or resources, c_i at types C_i are available at the location. So for example K above could be

$$\mathsf{loc}[\mathsf{ping} : \mathsf{rw}\langle \mathsf{P}\rangle, \mathsf{fing} : \mathsf{rw}\langle \mathsf{F}\rangle]$$

indicating that the services ping and $\mathsf{fing(er)}$ are supported at k; r indicates the permission to *read from* a channel, while w indicates the permission to *write to* the channel. However the types at which k becomes known depends on the types of the exporting channels. Suppose for example these had the types

$$\mathsf{xpt}_1 : \mathsf{w}\langle \mathsf{loc}[\mathsf{ping} : \mathsf{w}\langle \mathsf{P}\rangle]\rangle$$
$$\mathsf{xpt}_2 : \mathsf{w}\langle \mathsf{loc}[\mathsf{fing} : \mathsf{w}\langle \mathsf{F}\rangle]\rangle$$

Then processes receiving the name k from the source xpt_1 would only be able to write to the ping service at k, i.e. send messages to that service, while the source xpt_2 only allows similar restricted access to the finger service. It is in this way, by selectively distributing names at particular subtypes, that resource access policies are implemented in DPI.

In this paper we make two extensions to DPI. The first allows more control to locations over code which wishes to access their computation space. In DPI the migration rule is given by

$$k[\![\mathsf{goto}\, l.P]\!] \longrightarrow l[\![P]\!];$$

any thread is allowed to migrate to the site l. In SAFEDPI, the language of this paper, migration is represented by

$$k[\![\mathsf{goto}_p\, l.P]\!] \longrightarrow l[\![p!\langle P\rangle]\!]$$

A thread must designate a *port* p at l in order to migrate. It then reduces to the system $l[\![p!\langle P\rangle]\!]$, which apriori represents a thread running at location l. However this thread will have no effect until the site l makes available a corresponding thread of the form $l[\![p?(x)\, Q]\!]$; using standard communication this will now allow the effective entry of F. In this manner, by programming the presence or absence of ports, the site l can control the immigration of code.

Effectively we have replaced unconstrained spawning of processes at arbitrary sites by *higher-order communication*. Moreover these ports, higher-order channels, have types associated with them. The types on ports are the second major extension to the language. In general we allow *scripts*, parameterised code, to be sent via ports. These take the form

$$\lambda(\tilde{x} : \tilde{\mathsf{T}})P$$

where each x_i can be matched by arbitrary *transmittable values*; it is the types T_i which determine the nature of the abstraction. But when such a script is transmitted it may be instantiated at the receiving site by values of the appropriate type. This gives added security to sites by controlling the type at which scripts will be accepted. This of course depends on the granularity of the type structure for scripts.

The most straightforward form of type for scripts is

$$(\tilde{x} : \tilde{\mathsf{T}}) \to \mathbf{proc}$$

stating that, whenever a script of this type is instantiated with appropriate parameters, the result is guaranteed to be a well-typed process. But a priori there is no constraint on the resources it can use. To limit the access of incoming code to resources we introduce fine-grained *process types*, [26]. These dictate the capabilities, on both local and third-party channels, which the code is allowed to access, and take the form of a record:

$$\mathsf{pr}[c_1 : \mathsf{C}_1 @ k_1, \ldots, c_n : \mathsf{C}_n @ k_n]$$

A process of this type can use *at most* the set of channels c_i, located respectively at the locations k_i, with the capabilities C_i; in these process types the use of a local channel c is indicated by an entry of the form $c : \mathsf{C}@\mathsf{here}$.

When these process types are incorporated into script types a host location can have much more effective control over the behaviour of incoming code, particularly when we use a form of *dependent* function type. For example suppose a port only accepts scripts at the type

$$\mathsf{Fdep}(x : \mathsf{r}\langle \mathsf{T} \rangle \to \mathsf{pr}[x : \mathsf{r}\langle \mathsf{T} \rangle @ \mathsf{here},\ \mathsf{reply} : \mathsf{w}\langle \mathsf{T} \rangle @ k])$$

Then an incoming script can only be instantiated by a local channel, with read capability at type T. Moreover the resulting running code is now only allowed to read from this local channel and write to the third-party channel called reply located at the specific location k. With a port at the type

$$\mathsf{Fdep}(y : \mathsf{w}\langle \mathsf{T} \rangle @ k \to \mathsf{pr}[\mathsf{info} : \mathsf{r}\langle \mathsf{T} \rangle @ \mathsf{here},\ y : \mathsf{w}\langle \mathsf{T} \rangle @ k])$$

the host can instantiate the incoming script with some channel located at the site k, on which it has write permission, and the running code is restricted to writing there, and reading from a local channel called info.

Note that in both these examples the location k is built into the script types. Thus a server with an access port at this type would only allow entry to scripts which guarantee to write only at k. However *dependent types* can be used to allow this target site to be parameterised. Consider the simple example

$$\mathsf{Tdep}(z : \mathsf{L})\ \mathsf{Fdep}(y : \mathsf{w}\langle \mathsf{T} \rangle @ z \to \mathsf{pr}[\mathsf{info} : \mathsf{r}\langle \mathsf{T} \rangle @ \mathsf{here},\ y : \mathsf{w}\langle \mathsf{T} \rangle @ z])$$

where the script type is now parameterised by locations of some type L. This allows the server to accept scripts which can write the information at sites determined by the client.

Although these dependent types add considerable flexibility to the interaction between clients and servers, they have potential drawbacks; as we will see the client has to send with the script the actual objects on which their type is parameterised. In principle this opens up the possibility of (rogue) servers abusing this extra information. However *existential types* provide extra protection to clients, because, as we will see, this extra information is not required as part of the communication.

The language safeDpi is formally defined in Section 2, together with a reduction semantics. In Section 3 we define the set of types and the type inference system; the formal development relies heavily on the type systems already given in [8, 19]. In Section 4 we develop a series of example systems. These are designed on the one hand, to explain the intricacies of the the type inference rules, and on the other to demonstrate the power and flexibility of the types. This is followed by a section devoted to establishing the expected properties of the type system, in particular Subject Reduction.

We now turn to the second topic of the paper, typed behavioural equivalences. In untyped languages, these are normally defined coinductively, as the largest equivalences over processes which preserve, in some sense, actions of the form

$$M \xrightarrow{\mu} M' \tag{1}$$

Typically these actions describe the possible forms of interactions between a process and its environment. In a typed setting many of these actions will not be possible, because the environment will not have the power to participate in them. As a simple example consider the system

$$l[\![(\mathsf{newc}\, c : \mathsf{C})\ (\mathsf{xpt}!\langle c \rangle \ \mid c?(x)\, Q)]\!]$$

M, N ::=	*Systems*
$l[\![P]\!]$	Located Process
$M \mid N$	Composition
$(\mathsf{new}\, e : \mathsf{E})\, M$	Name Creation
$\mathbf{0}$	Termination

P, Q ::=	*Processes*
$u!\langle V\rangle$	Output
$u?(X : \mathsf{T})\, P$	Input
$\mathsf{goto}_u\, v.P$	Migration
$\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching
$(\mathsf{newc}\, c : \mathsf{C})\ P$	Channel creation
$(\mathsf{newreg}\, n : \mathsf{N})\ P$	Global name creation
$(\mathsf{newloc}\, k : \mathsf{K})\, \mathsf{with}\, Q\ \mathsf{in}\ P$	Location creation
$P \mid Q$	Composition
$F\ (\tilde{v})$	Application
$* P$	Iteration
stop	Termination
U, V, W ::=	*Values*
$(\tilde{v})$	Tuples
v ::=	*Value components*
$\lambda\, (\tilde{x} : \tilde{\mathsf{T}}).\, P$	Scripts
u	Identifiers

Fig. 1. Syntax of SAFEDPI

in an environment in which the export channel xpt can only send channels with the read capability. The environment will receive c along xpt but will not be able to transmit on c. Consequently the potential input actions on c by the process above will not be possible.

Following [9, 8] we replace the untyped actions in (1) with typed actions of the form

$$\mathcal{I} \triangleright M \xrightarrow{\mu} \mathcal{I}' \triangleright M'$$

where M is the system being observed while $\mathcal{I}$ is a constraint on the observing environment representing its knowledge of the system M. Actions change both the processes and the environment in which they are being observed. This will lead, in the standard manner, to a coinductively defined, bisimulation-based, relation between systems, which we denote by

$$\mathcal{I} \models\ M \approx_{bis} N$$

In our second main result of the paper, we prove that this coinductive relation coincides with a naturally defined contextual equivalence. One of the features of our approach is the explicit representation of the information which the environment can obtain from systems through testing with contexts. In such a highly constrained setting as this, this becomes a genuine aid in understanding the equivalence. This is the topic of Section 6.

The paper ends, in Section 7, with some conclusions and a brief survey of related work.

2. The language SAFEDPI

Syntax: The syntax, given in Figure 1, is a slight extension of that of DPI from [8]. It is explicitly typed, but for expository purposes we defer the description of types until Section 3. The syntax also presupposes a general set of channel names NAMES, ranged over by n, m, and a set of variables VARS, ranged over by x, y. *Identifiers*, ranged over by u, w, may come from either of these sets. NAMES is partitioned into two sets, LOCS ranged over by $k, l, \ldots$ for locations, and CHANS ranged over by $a, b, c, \ldots$ for channels.

There is also a distinguished subset of channels called *ports*, and ranged over by $p, q, \ldots$, which are used to handle higher-order values. Similarly we will sometimes use ξ, ξ' for variables which will be instantiated by higher-order values.

The syntax for systems, ranged over by M, N, O, is the same as in DPI, allowing the parallel composition of located processes $l[\![P]\!]$, which may share defined names, using the construct $(\mathsf{new}\, e : \mathsf{E})\, -$.

The syntax for processes, ranged over by P, Q is an extension of the PICALCULUS, [21], with primitives for migration between locations. Parallelism is allowed, we have the terminated process stop, and we also allow matching and mismatching, with the construct $\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$, and a form of iteration $*\,P$.

In the input construct $u?(X : \mathsf{T})\, P$ we take X to be a pattern which is used to deconstruct incoming values; this is a value which only contains distinct occurrences of variables. In our somewhat restricted format for values this means that X has the form $(\tilde{x})$, with each x_i being distinct. The output construct is asynchronous, $u!\langle V\rangle$. Here V is a tuple consisting of either identifiers or higher-order values. The latter can take the form of scripts, $\lambda\,(\tilde{x} : \tilde{\mathsf{T}}).\, P$, where P is an arbitrary process term; we will often use F to indicate an arbitrary script, whereas v will be reserved for the individual components in a tuple V; thus it will represent either an identifier or a script. Of particular interest to us will be tuples of the form $(\tilde{v}, F)$ which will be interpreted as *dependent values*; intuitively the script F depends on the values $\tilde{v}$.

At the risk of being verbose, the syntax has explicit notations for the various forms of names which can be declared. In $(\mathsf{newc}\, c : \mathsf{C})\ P$ a new local channel named c is declared, while $(\mathsf{newreg}\, n : \mathsf{N})\ P$ represents the generation of a new globally registered name n for channels. This allows channel names to be shared among a set of different locations in a coherent manner; see [8] for motivation. When a new location is declared, in $(\mathsf{newloc}\, k : \mathsf{K})\, \mathsf{with}\, Q\ \mathsf{in}\ P$, its declaration type K can only involve channel names which have been registered. This construct generates the new location k, sets the code Q running there, and in parallel continues with the execution of P. This specific construct for new locations is required since code may only be executed at a location once entry has been gained via a port; so here Q represents the code with which the location is initialised.

The main novelty in SAFEDPI, over DPI, is the construct

$$\mathsf{goto}_p\, k.F$$

Intuitively this means: migrate to location k via the port p with the code F. Our type system will ensure that F is in fact a script with a type appropriate to the port p; moreover entry will only be gained if at the location k the port p is currently active.

The various binding structures, for names and variables, give rise to the standard notions of free and bound occurrences of identifiers, α-conversion, and (capture-avoiding) substitution of values for identifiers in terms, $P\{\!|{}^{v}\!/\!_{u}|\!\}$; this is extended to patterns, $P\{\!|{}^{V}\!/\!_{X}|\!\}$ in the standard manner. We omit the details but three points are worth emphasising. The first is that many such substitutions may give rise to badly formed process terms but our typing system will ensure that this will never occur in well-typed terms. The second is that identifiers may occur in certain types and therefore we require a notion of substitution into types; this will be explained in Section 3. Finally terms will be identified up to α-equivalence, and bound identifiers will always be chosen to be distinct, and different from any free identifiers.

In the sequel we use *system* to refer to a *closed* system term, that is a system term which contain no free occurrences of variables; similarly a *process* means a closed process term.

Reduction Semantics: This is given in terms of a binary relation between systems

$$M \longrightarrow N$$

and is a mild generalisation of that given in [8, 10] for DPI.

Definition 2.1 (Contextual relations). A relation $\mathcal{R}$ over systems is said to be *contextual* if it preserves all the system constructors of the language; that is $M\ \mathcal{R}\ N$ implies

- $M \mid O\ \mathcal{R}\ N \mid O$ and $O \mid M\ \mathcal{R}\ O \mid N$
- $(\mathsf{new}\, e : \mathsf{E})\, M\ \mathcal{R}\ (\mathsf{new}\, e : \mathsf{E})\, N$. ■

The reduction relation is defined to be the least contextual relation which satisfies the axioms and rules in Figure 2. The rule (R-STR) merely says that we are working up to a structural equivalence, $\equiv$, which abstracts from inessential details in the terms representing systems. Formally structural equivalence is defined to be

(R-COMM)

$$\frac{}{k[\![c!\langle V\rangle]\!] \mid k[\![c?(X:\mathsf{T})\,P]\!] \longrightarrow k[\![P\{\!\!\{{}^{V}\!/x\}\!\!\}]\!]}$$

(R-SPLIT)

$$\frac{}{k[\![P \mid Q]\!] \longrightarrow k[\![P]\!] \mid k[\![Q]\!]}$$

(R-N.CREATE)

$$\frac{}{k[\![(\mathsf{newreg}\,n:\mathsf{N})\;P]\!] \longrightarrow (\mathsf{new}\,n:\mathsf{N})\,k[\![P]\!]}$$

(R-MOVE)

$$\frac{}{k[\![\mathsf{goto}_p\,l.F]\!] \longrightarrow l[\![p!\langle F\rangle]\!]}$$

(R-L.CREATE)

$$\frac{}{k[\![(\mathsf{newloc}\,l:\mathsf{L})\;\mathsf{with}\,C\;\;\mathsf{in}\;\;P]\!] \longrightarrow (\mathsf{new}\,l:\mathsf{L})(k[\![P]\!] \mid l[\![C]\!])}$$

(R-C.CREATE)

$$\frac{}{k[\![(\mathsf{newc}\,c:\mathsf{C})\;P]\!] \longrightarrow (\mathsf{new}\,c:\mathsf{C}@k)\,k[\![P]\!]}$$

(R-UNWIND)

$$\frac{k[\![P]\!] \mid M \longrightarrow M'}{k[\![*\,P]\!] \mid M \longrightarrow k[\![*\,P]\!] \mid M'}$$

(R-EQ)

$$\frac{}{k[\![\mathsf{if}\;u=u\;\mathsf{then}\;P\;\mathsf{else}\;Q]\!] \longrightarrow k[\![P]\!]}$$

(R-BETA)

$$\frac{}{k[\![(\lambda\,(\widetilde{x}:\widetilde{\mathsf{T}}).\;P)(\widetilde{v})]\!] \longrightarrow k[\![P\{\!\!\{{}^{\widetilde{v}}\!/\widetilde{x}\}\!\!\}]\!]}$$

(R-NEQ)

$$\frac{}{k[\![\mathsf{if}\;u=v\;\mathsf{then}\;P\;\mathsf{else}\;Q]\!] \longrightarrow k[\![Q]\!]}\quad u \neq v$$

(R-STR)

$$\frac{M \equiv N,\; M \longrightarrow M',\; M' \equiv N'}{N \longrightarrow N'}$$

Fig. 2. Reduction semantics for SAFEDPI

$$\begin{array}{lrcl}
(\text{S-EXTR}) & (\mathsf{new}\,n:\mathsf{E})(M \mid N) & = & M \;\mid\; (\mathsf{new}\,n:\mathsf{E})\,N \\
 & & & \quad\text{if } n \notin \mathsf{fn}(M) \\
(\text{S-COM}) & M \mid N & = & N \mid M \\
(\text{S-ASSOC}) & (M \mid N) \mid O & = & M \mid (N \mid O) \\
(\text{S-ZERO}) & M \mid \mathbf{0} & = & M \\
(\text{S-STOP}) & k[\![\mathsf{stop}]\!] & = & \mathbf{0} \\
(\text{S-FLIP}) & (\mathsf{new}\,n:\mathsf{E})\,(\mathsf{new}\,n':\mathsf{E}')\,M & = & (\mathsf{new}\,n':\mathsf{E}')\,(\mathsf{new}\,n:\mathsf{E})\,M \\
 & & & \quad\text{if } n' \notin \mathsf{E}, n \notin \mathsf{E}'
\end{array}$$

Fig. 3. Structural equivalence for SAFEDPI

the least contextual relation between (*closed*) systems which satisfies the axioms which are given in Figure 3; these are the natural adaptations of the usual axioms for structural equivalence in the PICALCULUS.

The main reduction involves *local communication*, governed by the rule (R-COMM), taken directly from DPI. However here the value V may be a script; in other words this rule encompasses higher-order communication. Higher-order output commands are generated by (R-MOVE), which has already been explained in the introduction.

Migration to a site l must designate a *port* p at which the migrating code is to be received. The rule

$$k[\![\mathsf{goto}_p\,l.F]\!] \longrightarrow l[\![p!\langle F\rangle]\!]$$

then translates the migration command into the system $l[\![p!\langle F\rangle]\!]$, which apriori represents a thread running at the target location l. However this will have no effect until the site l makes available a corresponding thread of the form $l[\![p?(\xi)\,Q]\!]$; using the rule (R-COMM) this will now allow the effective entry of F. In this manner the site l can control the immigration of code.

The rule (R-C.CREATE) exports the new channel name c generated by a process at k to the system level, where it is tagged with the declaration type $\mathsf{C}@k$; this records the location of the new channel. There is a corresponding rule for registered names, (R-N.CREATE); but such names are global and therefore there is no need to record where they were declared. The generation of new locations is governed by (R-L.CREATE):

$$k[\![(\mathsf{newloc}\,l:\mathsf{L})\;\mathsf{with}\,C\;\;\mathsf{in}\;\;P]\!] \longrightarrow (\mathsf{new}\,l:\mathsf{L})(k[\![P]\!] \mid l[\![C]\!])$$

The code C is set to run at the new location l, and note that this name is known to the continuation thread P running at the initiating location k.

Basic types:	$\mathbf{base} ::= \mathbf{int} \mid \mathbf{string} \mid \mathbf{unit} \mid \top \mid \mathbf{proc} \mid \ldots$
Local Channels:	$\mathsf{C}, \mathsf{D} ::= \mathsf{r}\langle \mathsf{T} \rangle \mid \mathsf{w}\langle \mathsf{T} \rangle \mid \mathsf{rw}\langle \mathsf{T}, \mathsf{U} \rangle$
Locations:	$\mathsf{L}, \mathsf{K} ::= \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n],\ n \geq 0$ provided $u_i = u_j$ implies $i = j$
Global resources:	$\mathsf{N} ::= \mathsf{rc}\langle \mathsf{C} \rangle$
First-order:	$\mathsf{A} := \mathbf{base} \mid \mathsf{C} \mid \mathsf{L} \mid \mathsf{N} \mid \mathsf{C}@w$
Processes:	$\pi ::= \mathbf{proc} \mid \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_n : \mathsf{C}_n@w_n]$ provided $u_i = u_j, w_i = w_j$ implies $i = j$
Scripts:	$\mathsf{S} ::= \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi)$
Values:	$\mathsf{T}, \mathsf{U} ::= \mathsf{A} \mid \mathsf{S} \mid \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T} \mid \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T}$

Fig. 4. Type expressions - informal

The remaining axioms are self-explanatory; there is testing of simple identifiers in (R-EQ), (R-NEQ), β-reduction in the rule (R-BETA) for instantiating scripts and a standard rule for iterated processes.

For examples of reductions see Section 4.

3. Typing

In this section we discuss the types and type inference for SAFEDPI. There are three subsections. The first discusses informally the types used, which builds on those in [24, 10, 8, 26], while the second describes the type environments required to infer that systems are well-typed. Because the details are heavily syntactic, on first reading it may be better to skip directly to the final subsection which deals with the type inference rules, referring to the first two sections only on a call-by-need basis.

3.1. The Types

The collection of types is an extension of those used in [8, 10], to which the reader is referred for more background and motivation. They are described informally in Figure 4. Note however that the syntax rules given there will be constrained by well-formedness conditions to be elaborated later, which will eliminate many evidently ill-formed types. For the same reason, when giving the syntax of systems in Figure 1, we have preferred to use the neutral metavariable E for types; the precise format allowed for such E in well-typed systems will also be determined by well-formedness constraints. Intuitively the types may be classified as follows:

Base types, ranged over by $\mathbf{base}$: We include some predefined collection of types such as $\mathbf{int}, \mathbf{unit}, \mathbf{bool}$, etc. for various constants in the language. The association of a particular type with a particular constant will be *global*, that is not dependent on a particular location. We also include $\mathbf{proc}$, to indicate that a process is well-typed, and a *top* type $\top$.

Local channel types, ranged over by C, D: These take the form

$$\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle$$

where T_r, T_w are *transmission* or *value types*; that is types of values which may be transmitted along channels. If an agent has a name at this type then it can transmit values of *at most* type T_w along it and receive from it values which have *at least* type T_r. In the formal description of types there will be a subtyping constraint, that T_w must be a subtype of T_r, explained in detail in [19]. When the transmit and receive types coincide we abbreviate this type by $\mathsf{rw}\langle \mathsf{T} \rangle$. We also allow the types $\mathsf{w}\langle \mathsf{T}_w \rangle$ and $\mathsf{r}\langle \mathsf{T}_r \rangle$, which only allow the transmission, reception respectively, of values.

Global resource name types, ranged over by N: These take the form $\mathsf{rc}\langle \mathsf{C} \rangle$, where C is a channel type. Intuitively these are the types of names which are available to be used in the declaration of new locations.

They allow an individual resource name, such as print, to be used in multiple locations, resulting in a form of *dynamic typing.*

Location types, ranged over by K, L: The standard form for these is

$$\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n]$$

where C_i are channel types, and the identifiers u_i are distinct. An agent possessing a location name k with this type may use the channels/resources u_i located there at the types C_i; from the point of view of the agent, k is a site which offers the services $u_1, \ldots u_n$ at the corresponding types. In the formal definition we will require each u_i to be already declared as a global resource name. If n is zero then the agent knows of the existence of k but has no right to use resources there. We abbreviate this trivial type from $\mathsf{loc}[]$ to loc. We also identify location types up to re-orderings.

Process types, ranged over by π. The simplest process type is $\mathbf{proc}$, which can be assigned to any well-typed process. More fine-grained process types take the form

$$\mathsf{pr}[u_1 : \mathsf{C}_1 @ w_1, \ldots, u_n : \mathsf{C}_n @ w_n]$$

where the pairs (u_i, w_i) are assumed to be distinct. A process of this type can use *at most* the resource names u_i at the location w_i with their specified types C_i; these types determine the locations at which the channels u_i may be used.

Script types, ranged over by S: The general form here is

$$\mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi)$$

Scripts of this type require parameters $(\tilde{v})$ of type $(\tilde{\mathsf{T}})$; when these are supplied the resulting process will be of type $\pi\{\!|\tilde{v}/\tilde{x}|\!\}$. In other words the type of the resulting process may in general depend on the parameters. In these types we allow π to contain occurrences of a special location constant here to denote the current location.

These types will be abbreviated to $(\tilde{\mathsf{T}} \to \pi)$ whenever the variables $(\tilde{x})$ do not appear in the process type π, that is when the type of result is in fact independent of the parameters.

Script types, a generalisation of those used in [26], are one major innovation of the current paper; they allow parameterised processes, or scripts to be transmitted. Examples of such types include

$\mathsf{w}\langle\mathsf{T}\rangle \to \mathbf{proc}$**:** the type of a script which is parameterised on a local channel name, on which write permission at type T is needed.

$(\mathsf{r}\langle\mathsf{R}\rangle, \mathsf{w}\langle\mathsf{W}\rangle @ k) \to \mathbf{proc}$**:** a value of this type will be applied to a pair, the first element will be a local channel with read capability at type R and the second a channel located at k with write capability at type W.

More importantly by using fine-grained process types, access to resources by incoming code can be restricted. Here are two examples:

$$\mathsf{Fdep}(x : \mathsf{r}\langle\mathsf{T}\rangle \to \mathsf{pr}[x : \mathsf{r}\langle\mathsf{T}\rangle @ \mathsf{here},\ \mathsf{reply} : \mathsf{w}\langle\mathsf{T}\rangle @ k])$$

Incoming code received at this type, can be instantiated by any local channel, say c, from which values can be read at type T. The resulting process is then only allowed access to two channels, namely the local channel c, from which it can read, and a channel named reply at the location k, to which it can write. This process will have the type $\mathsf{pr}[c : \mathsf{r}\langle\mathsf{T}\rangle @ \mathsf{here},\ \mathsf{reply} : \mathsf{w}\langle\mathsf{T}\rangle @ k]$. Code at the type

$$\mathsf{Fdep}((x, y, z) : (\mathsf{loc}, \mathsf{r}\langle\mathsf{T}\rangle @ x, \mathsf{w}\langle\mathsf{T}\rangle) \to \mathsf{pr}[y : \mathsf{r}\langle\mathsf{T}\rangle @ x,\ z : \mathsf{w}\langle\mathsf{T}\rangle @ \mathsf{here}])$$

needs to be instantiated by a location, a channel at that location, and a local channel. For example the location could be called source, the channel located there info, from which values can be read at the type T, and the local channel record, at which values can be written at type T. The resulting process will then have type

$$\mathsf{pr}[\mathsf{info} : \mathsf{r}\langle\mathsf{T}\rangle @ \mathsf{source},\ \mathsf{record} : \mathsf{w}\langle\mathsf{T}\rangle @ \mathsf{here}]$$

It can download information from the third-party source site source via the channel info there.

Finally Transmission or V value types dictate the kind of values which can be transmitted over channels. These may be first order values, or scripts. We also allow dependent and existential types to be used. For example inputting a value of the dependent type $\mathsf{Tdep}(x : \mathsf{K})\,\mathsf{S}$ will result in the reception of a pair (k, F), where F is guaranteed to be of type $\mathsf{S}\{\!|^{k}\!/\!_{x}|\!\}$; k is the witness that the script F has the required type, and is received with the script. On the other hand inputting at the corresponding existential type $\mathsf{Edep}(x : \mathsf{K})\,\mathsf{S}$ will only result in the reception of the value F, although, as we will see, when the overall system is typechecked the witness v must be produced, to verify that F is indeed well-typed.

Notation 3.1. [Globalising types] It is worth noting that there is a crucial distinction between local channel types C and, for example, location types. The former only make sense relative to a specific location, whereas the latter are location independent, or *global types*. We can convert the local channel type C to a global type by appending a location, $\mathsf{C}@w$; this is the type of a channel of type C located at w. In various contexts it will be convenient to apply this globalisation operation to an arbitrary type, $(\mathsf{T})@w$; this will only have an effect on any components of T which are local channel or script types. The operation is defined by induction on T:

$$(\mathsf{C})@w = \mathsf{C}@w, \quad (\mathsf{S})@w = \mathsf{S}, \quad (\mathbf{base})@w = \mathbf{base}$$
$$(\mathsf{K})@w = \mathsf{K}, \quad (\mathsf{C}@w')@w = \mathsf{C}@w'$$
$$(\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T})@w = \mathsf{Tdep}(\tilde{x} : (\tilde{\mathsf{T}})@w\ )\,(\mathsf{T})@w$$
$$(\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T})@w = \mathsf{Edep}(\tilde{x} : (\tilde{\mathsf{T}})@w\ )\,(\mathsf{T})@w$$

Note that in the last two clauses we have used the obvious notation $(\tilde{\mathsf{T}})@w$, for the list $\mathsf{T}_1@w, \ldots, \mathsf{T}_n@w$. ■

There are numerous constraints on the formation rules for types, well-documented in [10, 8]. The description given in Figure 4 should be viewed as defining *pre-types*; those which satisfy the formation constraints will then be considered to be types. It is best to describe these constraints relative to a *type environment*.

3.2. Type environments

A type judgement will take the form

$$\Gamma \vdash M$$

where Γ is a *type environment*, a list of assumptions about the types to be associated with the identifiers in the system M. These can take the form

- $u : \mathbf{base}$, meaning that u is of type $\mathbf{base}$
- $u : \mathsf{loc}$, meaning u is a location
- $u : \mathsf{C}@w$, meaning the channel u located at w has type C; for technical reasons C must be a *closed* type, that is containing no free occurrences of variables.
- $u : \mathsf{rc}\langle\mathsf{C}\rangle$, meaning u is a global resource name, which may be installed at any new location.
- $x : \mathsf{S}$, meaning x can be instantiated by any script which can be inferred to have type S
- $x : \langle\mathsf{T}\ \mathsf{with}\ \tilde{y} : \tilde{\mathsf{E}}\rangle$. This represents a *package*, which will be used to handle existential types. Intuitively this defines the association $x : \mathsf{T}$ but the type T may depend on the auxiliary associations $\tilde{y} : \tilde{\mathsf{E}}$.

Lists of assumptions are created dynamically during typechecking, typically by augmenting a current environment with new assumptions on bound variables. It is convenient to introduce a particular notation for this operation:

Definition 3.2 (Forming environments). Let $\{V : \mathsf{T}\}$ be a list of type assumptions defined by

- $\{x : \mathbf{base}\} = x : \mathbf{base}$
- $\{v : \mathsf{C}@w\} = v : \mathsf{C}@w$
- $\{x : \mathsf{S}\} = x : \mathsf{S}$
- $\{v : \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots u_n : \mathsf{C}_n]\} = v : \mathsf{loc}, u_1 : \mathsf{C}_1@v, \ldots u_n : \mathsf{C}_n@v$
- $\{(\tilde{y}, x) : \mathsf{Tdep}(\tilde{y} : \tilde{\mathsf{E}})\,\mathsf{T}\} = \{y_1 : \mathsf{E}_1\} \ldots, \{y_n : \mathsf{E}_n\},\ \{x : \mathsf{T}\}$
- $\{x : \mathsf{Edep}(\tilde{y} : \tilde{\mathsf{E}})\,\mathsf{T}\} = x : \langle\mathsf{T}\ \mathsf{with}\ \{y_1 : \mathsf{E}_1\} \ldots, \{y_n : \mathsf{E}_n\}\rangle$ ■

Of course there a lots of other possibilities for V and T but only those mentioned give rise to lists of assumptions. Moreover even those given may give rise to lists which are not consistent. For example we should not be able to introduce an assumption $u : \mathsf{loc}$ if u is already designated a channel, or introduce $u : \mathsf{C}@w$ unless w is known to be a location. Since type expressions also use identifiers, before introducing this assumption we would need to ensure that C is a properly formed type; for example it should only use identifiers which are already known. In order to describe the set of valid environments we introduce judgements of the form

$$\Gamma \vdash \mathbf{env}$$

The inference rules are straightforward and consequently are relegated to the appendix, in Figure 10. We also relegate to there the definition of subtyping judgements, of the form

$$\Gamma \vdash \mathsf{T} <: \mathsf{U},$$

given in Figure 11. Again the rules are straightforward, and mostly inherited from [8]. Informally the rules for subtyping channels are essentially determined by

$$\frac{\Gamma \vdash \mathsf{T}_r <: \mathsf{U}_r, \;\; \mathsf{U}_w <: \mathsf{T}_w,}{\begin{array}{l}\Gamma \vdash \mathsf{w}\langle \mathsf{T}_w \rangle <: \mathsf{w}\langle \mathsf{U}_w \rangle, \\ \Gamma \vdash \mathsf{r}\langle \mathsf{T}_r \rangle <: \mathsf{r}\langle \mathsf{U}_r \rangle \\ \Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{rw}\langle \mathsf{U}_r, \mathsf{U}_w \rangle\end{array}} \qquad \frac{}{\begin{array}{l}\Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{w}\langle \mathsf{T}_w \rangle \\ \Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{r}\langle \mathsf{T}_r \rangle\end{array}}$$

while those for locations depend on

$$\frac{\Gamma \vdash \mathsf{C}_i <: \mathsf{C}_i',}{\Gamma \vdash \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_m : \mathsf{C}_m] <: \mathsf{loc}[u_1 : \mathsf{C}_1', \ldots, u_n : \mathsf{C}_n']} \;\; 0 \leq n \leq m$$

However it is worth noting that process types are ordered differently than location types. For example we have

$$\Gamma \vdash \mathsf{pr}[u_1 : \mathsf{C}_1@k] <: \mathsf{pr}[u_1 : \mathsf{C}_1@k, \; u_2 : \mathsf{C}_2@l]$$

but

$$\Gamma \vdash \mathsf{loc}[u_1 : \mathsf{C}_1, u_2 : \mathsf{C}_2] <: \mathsf{loc}[u_1 : \mathsf{C}_1]$$

assuming, of course, that the various types used, $\mathsf{C}_i, \mathsf{C}_j$ are well-defined relative to Γ.

These rules have been formulated so that they can also be used to say what is a valid type relative to a type expression.

Definition 3.3 (Valid types). We say the type expression T is a valid type relative to Γ, written $\Gamma \vdash \mathsf{T} : \mathbf{ty}$, whenever we can derive the judgement $\Gamma \vdash \mathsf{T} <: \mathsf{T}$. ∎

Types can be viewed intuitively as *sets of capabilities* and *unioning* these sets corresponds to performing a *meet* operation with respect to subtyping. This we now explain. Let $(D, \leq)$ be a preorder. We say a subset $E \subseteq D$ is lower-bounded by $d \in D$ if $d \leq e$ for every e in E. Upper bounds are defined in a similar manner.

Definition 3.4 (partial meets and joins). We say that the preorder $(D, \leq)$ has *partial meets* if every pair of elements in D which has a lower bound also has a greatest lower bound. This means that for every pair of elements d_1, d_2 in D which has some lower bound, that is there is some element in $d \in D$ such that $d \leq d_1, \; d \leq d_2$, there is a particular lower bound, denoted $d_1 \sqcap d_2$ which is greater than or equal to every lower bound. The upper bound of pairs of elements, $d_1 \sqcup d_2$ is defined in an analogous manner. ∎

Let $\mathbf{Types}_\Gamma$ denote the set of all type expressions T such that $\Gamma \vdash \mathsf{T} : \mathbf{ty}$.

Theorem 3.5. For every Γ, the set $\mathbf{Types}_\Gamma$, ordered by $<:$, has partial meets and partial joins.

Proof. See Proposition A.2 in Appendix A □

Intuitively the existence of $\mathsf{T} \sqcap \mathsf{U}$ means that T and U are *compatible*, in that they allow compatible capabilities on values at these types. Moreover the type $\mathsf{T} \sqcap \mathsf{U}$ may be viewed as a *unioning* of the capabilities allowed by the individual types.

$$\frac{\Gamma, n : \mathsf{rc}\langle \mathsf{C} \rangle \vdash M}{\Gamma \vdash (\mathsf{new}\, n : \mathsf{rc}\langle \mathsf{C} \rangle)\, M}\ (\text{TY-GNEW}) \qquad \frac{\Gamma, c : \mathsf{C}@k \vdash M}{\Gamma \vdash (\mathsf{new}\, c : \mathsf{C}@k)\, M}\ (\text{TY-CNEW})$$

$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash \mathbf{0}}\ (\text{TY-NIL}) \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M \mid N}\ (\text{TY-PAR})$$

$$\frac{\Gamma \vdash_k P : \mathbf{proc}}{\Gamma \vdash k[\![P]\!]}\ (\text{TY-PROC}) \qquad \frac{\Gamma, \{k : \mathsf{K}\} \vdash M \quad \Gamma \vdash \mathsf{K} : \mathbf{ty}}{\Gamma \vdash (\mathsf{new}\, k : \mathsf{K})\, M}\ (\text{TY-LNEW})$$

Fig. 5. Typing Systems

3.3. Type Inference

We are now ready to describe the type inference system for ensuring that systems are well-typed. There are three forms of judgements, for systems, processes and values. The type inference rules for the first,

$\Gamma \vdash M$,

meaning that M is a well-typed system relative to Γ, are given in Figure 5. The intention is that whenever such a judgement can be inferred it will follow that Γ is a well-formed environment.

The main inference rule is (TY-PROC). In order to ensure that $k[\![P]\!]$ is a well-typed system we must show that the process P is well-typed to run at k; at the system level it is sufficient to be able to associate *any* process type with P. The typing of processes must be relative to a location because it may use *local* channels which are required to exist at k; it also turns out that typing of scripts will depend on their location. There is also a subtlety in the typing of name creation. First note that in these, and all subsequent rules, we assume that all bound names in a judgement must be different than any free names used as part of the judgement. Thus in (TY-CNEW) we know that c is actually fresh to Γ. However we are still not guaranteed that $\Gamma, c : \mathsf{C}@k$ is a well-defined environment even when Γ is. From the type environment rules it will only be so when C is a well-defined type expression relative to Γ, and k is known as a location. There is a further complication in (TY-LNEW), the rule for new location creation. Deriving $\Gamma, \{k : \mathsf{K}\} \vdash M$ will ensure that $\Gamma, \{k : \mathsf{K}\}$ is a well-defined environment, but we must also ensure that all of the channels used in the location type K have already been declared, in Γ, as global resource names. This is enforced by the second requirement, $\Gamma \vdash \mathsf{K} : \mathbf{ty}$.

The typing rules for the judgements on processes

$\Gamma \vdash_w P : \pi$

are given in Figure 7, and are defined simultaneously with the judgements for values, in Figure 6,

$\Gamma \vdash V : \mathsf{T}$

Let us first examine those for values. The rule (TY-LOOKUP) simply looks up the type of the identifier v relative to w in Γ, whereas (TY-BASE) allows base values to be typed for free. Note that the rule (TY-LOC) ensures that the judgement $\Gamma \vdash_w v : \mathsf{K}$, can only be made when K is a valid location type; this means that each channel used in K is already known to Γ, at a suitable type, as a global resource name. The rule (TY-MEET) is required because in certain circumstances we allow multiple associations with identifiers in valid environments; of course it can only be applied for types $\mathsf{T}_1, \mathsf{T}_2$ for which $\mathsf{T}_1 \sqcap \mathsf{T}_2$ exists. Dependent tuple values are typed with (TY-TUDEP). The value $(\tilde{v}, v)$ can be assigned the type $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T}$ provided each v_i can be assigned the type $\mathsf{E}_i\{\!|\tilde{v}/\tilde{x}|\!\}$ and v the type $\mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}$. For existential types we need to invent a new kind of value $\langle \tilde{v}, v \rangle$; these do not occur in the language SAFEDPI, and are only used by the type inference system; intuitively $\langle \tilde{v}, v \rangle$ is a *package* consisting of the value v together with the witnesses $\tilde{v}$, which provide evidence (for the type inference system) that v has its required type. The rule (TY-EDEP), which might also be called (TY-PACK), allows us to construct such values. It is similar to the rule for dependent tuples. The package $\langle \tilde{v}, v \rangle$ can be assigned the type $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T}$ provided we can establish that v_i can be assigned the type $v_i : \mathsf{E}_i\{\!|\tilde{v}/\tilde{x}|\!\}$ and v the type $\mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}$. Dependent tuples can be deconstructed and their components accessed

(TY-LOOKUP)
$$\frac{\Gamma, v : (\mathsf{E})_{@}w, \Gamma' \vdash \mathbf{env}}{\Gamma, v : (\mathsf{E})_{@}w, \Gamma' \vdash_w v : \mathsf{E}}$$

(TY-BASE)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash_w b : \mathbf{base}} \quad b \in \mathbf{base}$$

(TY-SUBVAL)
$$\frac{\Gamma \vdash_w V : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} <: \mathsf{T}'}{\Gamma \vdash_w V : \mathsf{T}'}$$

(TY-MEET)
$$\frac{\Gamma \vdash_w u : \mathsf{T}_1 \qquad \Gamma \vdash_w u : \mathsf{T}_2}{\Gamma \vdash_w u : \mathsf{T}_1 \sqcap \mathsf{T}_2}$$

(TY-LOC)
$$\frac{\Gamma \vdash_w u_i : \mathsf{C}_i \qquad \Gamma \vdash \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] : \mathbf{ty}}{\Gamma \vdash_w v : \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n]}$$

(TY-TUDEP)
$$\frac{\Gamma \vdash_w v_i : \mathsf{E}_i\{\!|{}^{\tilde{v}}/_{\tilde{x}}|\!\} \qquad \Gamma \vdash_w v : \mathsf{T}\{\!|{}^{\tilde{v}}/_{\tilde{x}}|\!\}}{\Gamma \vdash_w (\widetilde{v}, v) : \mathsf{Tdep}(\widetilde{x} : \widetilde{\mathsf{E}})\,\mathsf{T}}$$

(TY-EDEP)
$$\frac{\Gamma \vdash_w v_i : \mathsf{E}_i\{\!|{}^{\tilde{v}}/_{\tilde{x}}|\!\} \qquad \Gamma \vdash_w v : \mathsf{T}\{\!|{}^{\tilde{v}}/_{\tilde{x}}|\!\}}{\Gamma \vdash_w \langle \widetilde{v}, v\rangle : \mathsf{Edep}(\widetilde{x} : \widetilde{\mathsf{E}})\,\mathsf{T}}$$

(TY-ELOOKUP)
$$\frac{\Gamma, y : \langle(\mathsf{T})_{@}w \text{ with } \tilde{x} : \tilde{\mathsf{E}}\rangle, \Gamma' \vdash \mathbf{env}}{\Gamma, y : \langle \mathsf{T}_{@}w \text{ with } \widetilde{x} : \widetilde{\mathsf{E}}\rangle, \Gamma' \vdash_w y : \mathsf{T}}$$

(TY-UNPACK)
$$\frac{\Gamma \vdash_w \langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}}{\Gamma \vdash_w v : \mathsf{T}\{\!|{}^{\tilde{v}}/_{\tilde{x}}|\!\}}$$

Fig. 6. Typing Values

in the standard manner; see the fourth clause of Definition 3.2. However the corresponding deconstruction for existential types only allows access to the final component, and not the witnesses; (TY-UNPACK) allows the value, rather than the witnesses, to be extracted at the appropriate type from the package. Similarly (TY-ELOOKUP) only allows knowledge of the value, and not the witnesses, to be deduced from an existential assumption.

In Figure 7 the rules for name generation, (TY-NEWCHAN),(TY-NEWLOC) and (TY-NEWREG), are simple adaptations of the corresponding rules at the system level; note that in (TY-NEWLOC) we are guaranteed that the new name k does not occur in the type π, because of our convention on bound names; similarly for c in (TY-NEWCHAN) and n in (TY-NEWREG). (TY-STOP), (TY-ITER) and (TY-PAR) need no commentary. The rule (TY-EQ) is adapted from the analogous rule (TY-MATCH) in [10, 8]; recall that A_i range over first-order types and therefore this rule only allows the testing of first-order identifiers. (TY-ABS) and (TY-BETA) are standard rules for abstraction and application, adapted to dependent function types. But note the use of $\{\tilde{x} : (\tilde{\mathsf{T}})_{@}w\}$ in the premise of the former; the arguments in an abstraction are relativised to the current location w. The rule for migration, (TY-GO), is justified by the reduction semantics, although we could easily have phrased it in terms of the premises of the output rule.

However the real interest is in the typing of the input and output processes. To ensure $u!\langle V\rangle$ has a process type π relative to Γ, (TY-OUT), we have to ensure that u has the output capability at some type appropriate to V. Thus we need to find some type T such that $\Gamma \vdash_w V : \mathsf{T}$ and u has the output capability on T. But we must *also* check that this capability is allowed by π. Both of these requirements are encapsulated in the second premise of the rule

$$\Gamma \vdash \mathsf{pr}[u : \mathsf{w}\langle\mathsf{T}\rangle_{@}w] <: \pi$$

But there is a further complication. If the value being sent, V, contains channels, or more precisely capabilities on channels, then these must also be allowed by π. This is the intent of the third premise

$$\Gamma \vdash \mathsf{pr}_{\mathsf{ch}}[V : \mathsf{T}_{@}w] <: \pi$$

which uses a (partial) function which constructs a process type from a value V and its type; it essentially extracts out any channels which may be in V. To define this we use $\sqcup$ which is a *join* operator on types, relative to $<:$ the subtyping order; when applied to process types it effectively takes the union of the capabilities of the individual types. It is worth noting that $\mathsf{pr}_{\mathsf{ch}}[v : \mathsf{T}]$ is the trivial process type $\mathsf{pr}[]$ when T is a script type.

(ty-out)
$$\frac{\begin{array}{l}\Gamma \vdash_w V : \mathsf{T} \\ \Gamma \vdash \mathsf{pr}[u : \mathsf{w}\langle \mathsf{T}\rangle @ w] <: \pi \\ \Gamma \vdash \mathsf{pr_{ch}}[V : (\mathsf{T}) @ w] <: \pi\end{array}}{\Gamma \vdash_w u!\langle V\rangle : \pi}$$

(ty-in)
$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{pr}[u : \mathsf{r}\langle \mathsf{T}\rangle @ w] <: \pi \\ \Gamma,\ \{X : (\mathsf{T}) @ w\} \vdash_w P : \pi \sqcup \mathsf{pr_{ch}}[X : (\mathsf{T}) @ w]\end{array}}{\Gamma \vdash_w u?(X : \mathsf{T})\, P : \pi}$$

(ty-go)
$$\frac{\Gamma \vdash_u v!\langle F\rangle : \pi}{\Gamma \vdash_w \mathsf{goto}_v\, u.F : \pi}$$

(ty-newloc)
$$\frac{\begin{array}{l}\Gamma, \{k : \mathsf{K}\} \vdash_k C : \pi \\ \Gamma, \{k : \mathsf{K}\} \vdash_w P : \pi \\ \Gamma \vdash \mathsf{K} : \mathbf{ty}\end{array}}{\Gamma \vdash_w (\mathsf{newloc}\ k : \mathsf{K})\ \mathsf{with}\ C\ \ \mathsf{in}\ \ P : \pi}$$

(ty-eq)
$$\frac{\begin{array}{l}\Gamma \vdash_w u_1 : \mathsf{A}_1, u_2 : \mathsf{A}_2 \\ \Gamma \vdash_w Q : \pi \\ \Gamma, \{u_1 : \mathsf{A}_2\}, \{u_2 : \mathsf{A}_1\} \vdash_w P : \pi \\ \text{(provided } \Gamma, \{u_1{:}\mathsf{A}_2\}, \{u_2{:}\mathsf{A}_1\} \vdash \mathbf{env}\text{)}\end{array}}{\Gamma \vdash_w \mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q : \pi}$$

(ty-abs)
$$\frac{\Gamma, \{\tilde{x} : (\tilde{\mathsf{T}}) @ w\} \vdash_w P : \pi\{\!\!\{ {}^{w}\!/\!_{\mathsf{here}} \}\!\!\}}{\Gamma \vdash_w \lambda\, (\widetilde{x} : \widetilde{\mathsf{T}}).\ P : \mathsf{Fdep}(\widetilde{x} : \widetilde{\mathsf{T}} \to \pi)}$$

(ty-iter)
$$\frac{\Gamma \vdash_w P : \pi}{\Gamma \vdash_w *P : \pi}$$

(ty-outE)
$$\frac{\begin{array}{l}\Gamma \vdash_w \langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T} \\ \Gamma \vdash \mathsf{pr}[u : \mathsf{w}\langle \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T}\rangle @ w] <: \pi \\ \Gamma \vdash \mathsf{pr_{ch}}[\tilde{v} : (\tilde{\mathsf{E}}) @ w] <: \pi\end{array}}{\Gamma \vdash_w u!\langle v\rangle : \pi}$$

(ty-subproc)
$$\frac{\begin{array}{l}\Gamma \vdash_w P : \pi \\ \Gamma \vdash \pi <: \pi'\end{array}}{\Gamma \vdash_w P : \pi'}$$

(ty-stop)
$$\frac{\Gamma \vdash \pi : \mathbf{ty}}{\Gamma \vdash_w \mathsf{stop} : \pi}$$

(ty-newchan)
$$\frac{\Gamma, c : \mathsf{C} @ w \vdash_w P : \pi \sqcup \mathsf{pr}[c @ w : \mathsf{C}]}{\Gamma \vdash_w (\mathsf{newc}\, c : \mathsf{C})\ \ P : \pi}$$

(ty-newreg)
$$\frac{\Gamma, n : \mathsf{N} \vdash_w P : \pi}{\Gamma \vdash_w (\mathsf{newreg}\, n : \mathsf{N})\ \ P : \pi}$$

(ty-beta)
$$\frac{\begin{array}{l}\Gamma \vdash_w F : \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi) \\ \Gamma \vdash_w v_i : \mathsf{T}_i\end{array}}{\Gamma \vdash_w F\ (\widetilde{v}) : \pi\{\!\!\{ {}^{\widetilde{v}}\!/\!_{\widetilde{x}} \}\!\!\}\{\!\!\{ {}^{w}\!/\!_{\mathsf{here}} \}\!\!\}}$$

(ty-par)
$$\frac{\begin{array}{l}\Gamma \vdash_w P : \pi \\ \Gamma \vdash_w Q : \pi\end{array}}{\Gamma \vdash_w P \mid Q : \pi}$$

Fig. 7. Typing Processes

$$\begin{aligned}
\mathsf{pr_{ch}}[v : \mathsf{C} @ w] &= \mathsf{pr}[v : \mathsf{C} @ w] \\
\mathsf{pr_{ch}}[v : \mathsf{K}] &= \mathsf{pr}[c_1 : \mathsf{C}_1 @ v, \ldots, c_n : \mathsf{C}_n @ v] \\
&\qquad \text{where } \mathsf{K} = \mathsf{loc}[c_1 : \mathsf{C}_1, \ldots, c_n : \mathsf{C}_n] \\
\mathsf{pr_{ch}}[\tilde{v} : \tilde{\mathsf{T}}] &= \mathsf{pr_{ch}}[v_1 : \mathsf{T}_1] \sqcup \ldots \sqcup \mathsf{pr_{ch}}[v_1 : \mathsf{T}_1] \\
\mathsf{pr_{ch}}[(\tilde{v}, v) : \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T}] &= \mathsf{pr_{ch}}[\tilde{v} : \tilde{\mathsf{E}}] \sqcup \mathsf{pr_{ch}}[v : \mathsf{T}] \\
\mathsf{pr_{ch}}[\langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{T}] &= \mathsf{pr_{ch}}[\tilde{v} : \tilde{\mathsf{E}}] \sqcup \mathsf{pr_{ch}}[v : \mathsf{T}] \\
\mathsf{pr_{ch}}[v : \mathsf{T}] &= \mathsf{pr}[] \qquad \text{otherwise}
\end{aligned}$$

The rule for transmitting existential values, (ty-outE) is a slight variation. We must establish a *package* $\langle \tilde{v}, v\rangle$ of the correct outgoing type, but only the (unpacked) value v is actually transmitted. Finally to ensure $u?(X : \mathsf{T})\, P$ has the type π, we need to check that u has the appropriate read capability, which also is allowed by π,

$$\Gamma \vdash \mathsf{pr}[u : \mathsf{r}\langle \mathsf{T}\rangle @ w] <: \pi$$

and that the capabilities exercised by the residual P are either allowed by π or inherited by values which are input and bound to X:

$\Gamma,\ \{X : (\mathsf{T})_{@}w\} \vdash_w P : \pi \sqcup \mathsf{pr_{ch}}[X : \mathsf{T}_{@}w]$

It is worth noting that the typing rules for input and output degenerate to the more standard form, for example as in [10], when we wish to establish that the processes are simply well-typed, that is have the type **proc**. For example we have the derived instances:

$$\text{(TY-OUT)}\quad \frac{\Gamma \vdash_w V : \mathsf{T} \qquad \Gamma \vdash_w u : \mathsf{w}\langle \mathsf{T}\rangle}{\Gamma \vdash_w u!\langle V\rangle : \mathbf{proc}} \qquad\qquad \text{(TY-IN)}\quad \frac{\Gamma \vdash_w u : \mathsf{r}\langle \mathsf{T}\rangle \qquad \Gamma,\ \{X : (\mathsf{T})_{@}w\} \vdash_w P : \mathbf{proc}}{\Gamma \vdash_w u?(X : \mathsf{T})\, P : \mathbf{proc}}$$

4. Examples

In this section we demonstrate the usefulness of the type system by a series of examples of increasing sophistication.

To make the examples more readable let us introduce some convenient notation. First we will abbreviate the transmission type $\mathbf{unit} \to \mathbf{proc}$, for thunked processes, simply to thunk. Then we use run as an abbreviation for the term $\lambda\xi\ \xi()$, where $()$ is the only value of type unit. So the type of run is $\mathsf{thunk} \to \mathbf{proc}$; it takes a thunked process and runs it. Thunked processes, which we often refer to as *thunks*, take the form $\lambda\,().\,P$ but in the context of $\mathsf{goto}\,p.\ldots$ and port outputs $p!\langle\ldots\rangle$ we will omit the λ abstraction; thus $\mathsf{goto}_{in}\,l.\lambda\,().\,P$ is abbreviated to $\mathsf{goto}_{in}\,l.P$. Finally we mimic the notation of process types for thunks, by letting $\mathsf{th}[....]$ denote the type $\mathbf{unit} \to \mathsf{pr}[....]$.

4.1. Installing and broadcasting services

Suppose there are two globally defined channel names ping and fing and a port name in; that is we are working in a type environment Γ with the property that

$$\Gamma \vdash \mathsf{ping} : \mathsf{rc}\langle \mathsf{D}_p\rangle,\ \mathsf{fing} : \mathsf{rc}\langle \mathsf{D}_f\rangle, \mathsf{in} : \mathsf{rc}\langle \mathsf{D}_i\rangle \tag{2}$$

for some types $\mathsf{D}_p, \mathsf{D}_f$ and D_i. Let L be a location type such that

$$\mathsf{L} <: \mathsf{loc}[\mathsf{in} : \mathsf{C}_i, \mathsf{ping} : \mathsf{C}_p, \mathsf{fing} : \mathsf{C}_f]. \tag{3}$$

Then in the system

$r[\![(\mathsf{newloc}\,l : \mathsf{L})\,\mathsf{with}\,Q\ \ \mathsf{in}\ \ P]\!]$

the site r generates a new location l with declaration type L; it evolves to the new system

$(\mathsf{new}\,l : \mathsf{L})(r[\![P]\!] \mid l[\![Q]\!])$

To be well-typed with respect to Γ we need that

- L is a proper declaration type for locations, that is $\Gamma, \{l : \mathsf{L}\} \vdash l : \mathsf{L}$. This means that all the resource names in L should be globally defined in Γ with a type which supports their use in L. For example this would require $\mathsf{D}_p <: \mathsf{C}_p$, $\mathsf{D}_f <: \mathsf{C}_f$ and $\mathsf{D}_i <: \mathsf{C}_i$ with respect to Γ.
- the residual P is well-typed to run at r, that is
$$\Gamma, \{l : \mathsf{L}\} \vdash_r P : \mathbf{proc}$$
- the installed code is well-typed to run at the new location l, that is
$$\Gamma, \{l : \mathsf{L}\} \vdash_l Q : \mathbf{proc}.$$

The residual P running at r now knows the location l and its type, and may make it known to other agents. Suppose P has the form

$*\mathsf{dist}_1!\langle l\rangle \mid *\mathsf{dist}_2!\langle l\rangle \mid R$

where dist_i are distribution channels at r for broadcasting information. Agents with access to these channels can find out about l. More importantly the type at which they receive l depends on the types of dist_i at the site r. For example suppose Γ contains

$$\begin{aligned}\mathsf{dist}_1 &: \mathsf{rw}\langle \mathsf{loc}[\mathsf{in} : \mathsf{w}\langle \mathsf{I}\rangle, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle]\rangle @ r,\\ \mathsf{dist}_2 &: \mathsf{rw}\langle \mathsf{loc}[\mathsf{in} : \mathsf{w}\langle \mathsf{I}\rangle, \mathsf{fing} : \mathsf{w}\langle \mathsf{V}_f\rangle]\rangle @ r\end{aligned} \tag{4}$$

for some types $\mathsf{I}, \mathsf{V}_p, \mathsf{V}_f$. Then agents finding out about l from the source dist_1 only know about the resource ping there (in addition to the port in), while if the source of information is dist_2 only fing may be used. Of course an agent may have access to both sources. If that is the case then they can eventually come to know l at the type $\mathsf{loc}[\mathsf{in} : \mathsf{w}\langle \mathsf{I}\rangle, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle, \mathsf{fing} : \mathsf{w}\langle \mathsf{V}_f\rangle]$, thereby obtaining knowledge of both resources. But more generally access to l will be governed by ports such as in and their programming via the installed code Q.

4.2. Servicing resources

The installed code Q determines, at least initially, who has access to the newly created site l. A typical example of the installed code C may take the form

$$*\mathsf{in}?(\xi : \mathsf{thunk}) \;\; (\mathsf{run}\, \xi) \;|\; S$$

Entry will be allowed to any well-typed thread at the port in, and the thread can subsequently interact with the servicing code S. This will only be well-typed if the original declaration type for the global name in allows values of type thunk to be received at that port. For example it will be well-typed if $\Gamma \vdash \mathsf{in} : \mathsf{rc}\langle \mathsf{rw}\langle \mathsf{thunk}\rangle\rangle$, that is setting the declaration type D_i in (2) above to be thunk, and the type I in the typing for the sources at r, in (4), to be thunk also.

Note that there is some choice in the type at which in is declared at l, in (3) above. If C_i is set to $\mathsf{rw}\langle \mathsf{thunk}\rangle$ then the servicing code S at l can both read and write at in, but the type $\mathsf{r}\langle \mathsf{thunk}\rangle$ is sufficient for well-typing, if S never writes to that port.

Consider a thread running at r such as

$$r[\![\mathsf{dist}_1?(x : \mathsf{L}_p)\, \mathsf{goto}_{\mathsf{in}}\, x.\mathsf{ping}!\langle v\rangle]\!] \tag{5}$$

which gains knowledge of the newly created location l via the source dist_1. Here we use L_p to be an abbreviation for an instance of the type used in (4) above, $\mathsf{loc}[\mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle]$. This thread is well-typed,

$$\Gamma \vdash r[\![\mathsf{dist}_1?(x : \mathsf{L}_p)\, \mathsf{goto}_{\mathsf{in}}\, x.\mathsf{ping}!\langle v\rangle]\!]$$

provided the value v can be assigned the proper type for ping, namely V_p. This follows from the fact that for such a Γ we can establish

$$\Gamma \vdash_r \mathsf{dist}_1?(x : \mathsf{L}_p)\, \mathsf{goto}_{\mathsf{in}}\, x.\mathsf{ping}!\langle v\rangle \;\; : \mathbf{proc}$$

which in turn follows from

$$\Gamma, \{x : \mathsf{L}_p\} \vdash_r \mathsf{goto}_{\mathsf{in}}\, x.\mathsf{ping}!\langle v\rangle \;\; : \mathbf{proc}$$

This is a consequence of applying the typing rule (TY-GO) to the judgement

$$\Gamma, \{x : \mathsf{L}_p\} \vdash_x \mathsf{in}!\langle \mathsf{ping}!\langle v\rangle\rangle \;\; : \mathbf{proc} \tag{6}$$

The type environment $\Gamma, \{x : \mathsf{L}_p\}$ takes the form

$$\Gamma, x : \mathsf{loc}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @ x, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle @ x$$

Therefore (6) follows from an application of the simple form of the output rule (TY-OUT), provided we can establish

$$\Gamma, x : \mathsf{loc}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @ x, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle @ x \vdash_x \lambda\, ().\; \mathsf{ping}!\langle v\rangle : \mathsf{thunk},$$

that is

$$\Gamma, x : \mathsf{loc}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @ x, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle @ x \vdash_x \mathsf{ping}!\langle v\rangle : \mathbf{proc}$$

Finally this requires the judgement

$$\Gamma, x : \mathsf{loc}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @x, \mathsf{ping} : \mathsf{w}\langle \mathsf{V}_p\rangle @x \vdash_x v : \mathsf{V}_p \tag{7}$$

Note that this checking of v is carried out relative to the variable location x; so the type V_p needs to be some *global type*, whose meaning is independent of the current location. This could be a base type such as **int**, although we will see more interesting examples, such as *return channels*, later.

4.3. Site protection

A simple infrastructure for a typical site could take the form

$$h[\![* \mathsf{in}?(\xi : \mathsf{I})\ \mathsf{run}\,\xi \mid S]\!]$$

The on-site code S could provide various services for incoming agents, repeatedly accepted at the input port in. In a relaxed computing environment the type I could simply be thunk indicating that any well-typed code will be allowed to immigrate. In the sequel we will always assume that when the type of the port in is not discussed it has this liberal type.

However constraints can be imposed on incoming code by only publicising ports which have associated with them more restrictive *guardian* types. In such cases it is important that read capabilities on the these ports be retained by the host. This point will be ignored in the ensuring discussion, which instead concentrates on the forms the guardian types can take.

Consider a system consisting of a server and client, defined below, running in parallel.

$$\begin{array}{ll} \text{Server:} & s[\![* \mathsf{req}?(\xi : \mathsf{S})\,\mathsf{run}\,\xi \mid \mathsf{news}!\langle \mathit{scandal}\rangle]\!] \\ \text{Client:} & c[\![\mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\, c.\,\mathsf{report}!\langle x\rangle \\ & \quad \mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots]\!] \end{array} \tag{8}$$

The server is straightforward; it accepts incoming code at the port req and runs it. The only service it provides is some information on a channel called news. The client, who knows of the req port at the server sends code there to collect the news and report it back to it's own channel report; the type at which it inputs from news, which obviously must be **string**, is elided. This code migrates twice, once via the port req from the client to the server, and once via the port in, from the server to the client.

The server protects its site using the guardian type S while the client protects its site using R. What should these be? Let us assume that both sites have the required channels at appropriate types; suppose in Γ we have the entries

$$\begin{array}{l} \mathsf{news} : \mathsf{rw}\langle \mathbf{string}\rangle @s,\quad \mathsf{req} : \mathsf{rw}\langle \mathsf{S}\rangle @s, \\ \mathsf{report} : \mathsf{rw}\langle \mathbf{string}\rangle @c,\quad \mathsf{in} : \mathsf{rw}\langle \mathsf{R}\rangle @c \end{array}$$

The first possibility is for the client to be relaxed but the server vigilant:

$$\begin{array}{ll} \mathsf{R} : & \mathsf{thunk} \\ \mathsf{S} : & \mathsf{th}[\mathsf{news} : \mathsf{r}\langle \mathbf{string}\rangle @s,\ \mathsf{in} : \mathsf{w}\langle \mathsf{R}\rangle @c] \end{array}$$

Here the client allows in any well-typed process, whereas the server will only accept at the port req processes which use at most the local channel news and the port in at the site c; moreover the local channel news can only be used in read mode. Note in passing that this limits severely the usefulness of the server, as results can only be returned to the client c; this limitation is addressed in Section 4.5.

With these types one can show that the overall system is well-typed. Typing the server is straightforward but to type the client we need to establish, among other requirements,

$$\Gamma \vdash_c \mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\, c.\,\mathsf{report}!\langle x\rangle\ : \mathbf{proc}$$

As usual this follows by an application of (TY-GO) from

$$\Gamma \vdash_s \mathsf{req}!\langle\ \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\, c.\,\mathsf{report}!\langle x\rangle\ \rangle\ : \mathbf{proc}$$

which in turn requires establishing

$$\Gamma \vdash_s \lambda\,().\ \mathsf{news}?(x)\,\mathsf{goto_{in}}\,c.\,\mathsf{report}!\langle x\rangle \ : \mathsf{S}$$

In other words the incoming code should match the guardian type of the server, S. By *dethunking* we get the requirement

$$\Gamma \vdash_s \mathsf{news}?(x)\,\mathsf{goto_{in}}\,c.\,\mathsf{report}!\langle x\rangle \ : \mathsf{pr}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

This is established via an application of the rule (TY-IN). The first premise is immediate since we assume $\Gamma \vdash_s \mathsf{news} : \mathsf{rw}\langle\mathbf{string}\rangle$. Moreover the second amounts to

$$\Gamma, x : \mathbf{string} \vdash_s \mathsf{goto_{in}}\,c.\,\mathsf{report}!\langle x\rangle \ : \mathsf{pr}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

because the value being received is a string; that is $\mathsf{pr_{ch}}[x : \mathbf{string}@s]$ is the trivial process type $\mathsf{pr}[]$.

The significant step in establishing this second premise is to check that the code returning to the client satisfies its guardian type R:

$$\Gamma, x : \mathbf{string} \vdash_c \mathsf{in}!\langle\ \mathsf{report}!\langle x\rangle\rangle \ : \mathsf{pr}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c] \tag{9}$$

However this is straightforward since R is the liberal guardian thunk. It follows by an application of the output rule (TY-OUT) in Figure 7. But it is important to note that in the application the third premise is vacuous, as $\mathsf{pr_{ch}}[\lambda\,().\ \mathsf{report}!\langle x\rangle : \mathbf{proc}]$ is the trivial type $\mathsf{pr}[]$.

The current type $\mathsf{R} = \mathsf{thunk}$ leaves the client site open to abuse but it is easy to check that the above reasoning is still valid if the guardians are changed to

$$\mathsf{R}: \quad \mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c]$$
$$\mathsf{S}: \quad \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

Here the guardian for the client only allows in agents which write to the local port report; note that this change requires that the guardian at the server site also uses this more restrictive type in its annotation for the port in at c.

One can check that with these new restrictive guardians the system is still well-typed. The only extra work required is in providing a proof for the judgement (9) above, ensuring that the code returning to the client satisfies the more demanding guardian. By an application of (TY-GO) and (TY-OUT) this reduces to the judgement

$$\Gamma, x : \mathbf{string} \vdash_c \lambda\,().\ \mathsf{report}!\langle x\rangle \ : \mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c]$$

which is a straightforward consequence of (TY-OUT).

It might be tempting to define the guardians by

$$\mathsf{R}: \quad \mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c]$$
$$\mathsf{S}: \quad \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@c]$$

Here both server and client protect their own resources but the server is uninterested in what happens at the client site, by allowing code with arbitrary capabilities on the client port in. However there is an intuitive inconsistency here. The client has read capability at its port, at the restrictive type R, while somehow the server has obtained a more liberal write capability there, namely thunk.

In fact the system can not be typed with these revised guardians. In particular

$$\Gamma \not\vdash s[\![\mathsf{req}?(\xi : \mathsf{S})\ \mathsf{run}\,\xi]\!]$$

Any derivation of this judgement would require the judgement

$$\Gamma, \xi : \mathsf{S} \vdash_s \mathsf{run}\,\xi$$

which in turn would require

$$\Gamma \vdash \mathsf{S} : \mathbf{ty}$$

or more formally

$$\Gamma \vdash \mathsf{S} <: \mathsf{S}$$

But as we will see this can not be inferred; that is S is not a valid type, relative to Γ.

To see why let us suppose, for simplicity, that the port in has been declared at the site c with a type of the form $\mathsf{rw}\langle \mathsf{R}, \mathsf{W}\rangle$ for some type W. One constraint in the type formation rules, (see (TY-CHAN) in Figure 11) is that the write capabilities on a channel are always a subtype of the read capabilities; in our setting this means that $\Gamma \vdash \mathsf{W} <: \mathsf{R}$. Our rules also ensure that $\Gamma \vdash_c \mathsf{in} : \mathsf{w}\langle \mathsf{T}_w \rangle$ implies $\Gamma \vdash \mathsf{T}_w <: \mathsf{W}$ and consequently $\Gamma \vdash \mathsf{T}_w <: \mathsf{R}$.

However the structure of R ensures that $\Gamma' \vdash \mathsf{thunk} <: \mathsf{R}$ for no Γ', from which we can conclude that $\Gamma \not\vdash_c \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @ c$. But this is one of the requirements, in the formation rules in Figure 11, to establish $\Gamma \vdash \mathsf{S} : \mathbf{ty}$.

4.4. Anonymous channels

Consider the following variation on the server/client system:

$$\begin{array}{ll} \text{Server:} & s[\![\mathsf{req}?(\xi : \mathsf{S})\,\mathsf{run}\,\xi \mid \mathsf{where}?((y,z) : \mathsf{T})\,\mathsf{goto}_{\mathsf{in}}\, y.z!\langle \mathit{scandal}\rangle]\!] \\ \text{Client:} & c[\![\mathsf{goto}_{\mathsf{req}}\, s.\mathsf{where}!\langle c, \mathsf{report}\rangle \mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots]\!] \end{array} \tag{10}$$

Here the protocol is somewhat different; the client simply supplies to the server, via the channel where, the address of a channel on which to supply the news; this consists of the pair of a location and a channel on which to report. The server then launches a thread which migrates to the relevant location, which is assumed to have an in port, to deliver the news.

Defining guardians is straightforward. For example these could be

$$\begin{array}{ll} \mathsf{R}: & \mathsf{thunk} \\ \mathsf{S}: & \mathsf{th}[\mathsf{where} : \mathsf{w}\langle \mathsf{T}\rangle @ s,\ \mathsf{in} : \mathsf{w}\langle \mathsf{thunk}\rangle @ c] \end{array}$$

However the difficulty is in ascertaining the required type T for the pair of values. One possibility is to set

$$\mathsf{T} = (\mathsf{I},\ \mathsf{w}\langle \mathbf{string}\rangle)$$

where I is the location type $\mathsf{loc}[\mathsf{in} : \mathsf{w}\langle \mathsf{R}\rangle]$, allowing the first component to be a location with an in port at the appropriate type and the second to be a channel for sending strings.

Unfortunately the server can not be typed with such a T. The problem arises when we try to establish

$$\Gamma, \{(y,z) : (\mathsf{T}) @ s\} \vdash_s \mathsf{goto}_{\mathsf{in}}\, y.z!\langle \mathit{scandal}\rangle : \mathbf{proc} \tag{11}$$

Unravelling the extended environment this means establishing

$$\Gamma, y : \mathsf{loc}, \mathsf{in} : \mathsf{w}\langle \mathsf{R}\rangle @ y, z : \mathsf{w}\langle \mathbf{string}\rangle @ s \vdash_y z!\langle \mathit{scandal}\rangle : \mathbf{proc}$$

which is not possible; the output rule (TY-OUT) demands that z be a channel at the location y.

So to be able to statically type this example we need to be able to use the first component in the pair (y,z) as part of the type of the second component; we need a dependent type.

Let

$$\mathsf{T} = \mathsf{Tdep}(x : \mathsf{I})\, \mathsf{w}\langle \mathbf{string}\rangle @ x$$

Note that (SUB-TUDEP) from Figure 11 ensures that this is a well-defined type:

$$\Gamma \vdash \mathsf{T} : \mathbf{ty}$$

because

$$\Gamma, \{x : \mathsf{I}\} \vdash \mathsf{w}\langle \mathbf{string}\rangle @ x : \mathbf{ty}$$

So this type can be safely used as part of a process. Moreover it is now easy to establish (11) above as the extended environment $\Gamma, \{(y,z) : \mathsf{T}\}$ unravels to $\Gamma, y : \mathsf{loc},\ \mathsf{in} : \mathsf{w}\langle \mathsf{R}\rangle @ y,\ z : \mathsf{w}\langle \mathbf{string}\rangle @ y$.

These location dependent types were introduced in [10], where they are shown to be very useful for typing migrating code, as they allow the transmission of *anonymous* channels between sites. In our example the server does not need to know, apriori, the name of the report channel at the client site. In the sequel we will borrow the notation used in [10] for these dependent types; we use $(u @ w)$ to denote any pair of identifiers

(u, w) which is expected to have a dependent type of the form $\mathsf{Tdep}(x : \mathsf{I})\,\mathsf{C}$. In a similar vein we abbreviate this type to $\mathsf{C}@\mathsf{I}$. Thus we can reformulate the example (10) above as:

$$\begin{array}{ll}
\text{Server} & s[\![\mathsf{req}?(\xi : \mathsf{S})\,\mathsf{run}\,\xi \mid \mathsf{where}?((z@y) : \mathsf{w}\langle\mathbf{string}\rangle@\mathsf{I})\,\mathsf{goto}_{\mathsf{in}}\,y.z!\langle_{scandal}\rangle]\!] \\
\text{Client} & c[\![(\mathsf{newc}\,\mathsf{report}) \\
& \qquad\qquad \mathsf{goto}_{\mathsf{req}}\,s.\,\mathsf{where}!\langle\mathsf{report}\,@c\rangle \mid \\
& \qquad\qquad \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots]\!]
\end{array}$$

Here, as a form of self-protection, the client generates a new return channel, also called report and whose obvious type is elided, which it sends to the server. The client's self-protection consists of reading this channel exactly once, which it knows will be a response to its request to the server.

Note that these location dependent types are exactly what is required to type the example (5) above. In the type judgement (7) we need to find an appropriate type V_p for values transmitted on the channel ping. We can now let V_p be the dependent type $\mathsf{w}\langle\mathbf{string}\rangle@\mathsf{loc}$, consisting of a return address; that is a location, and a write capability at some channel at that location.

4.5. Dependent process types

There remains a major difficulty with the server in (10) and (8) above. The guardian type of the server S uses the name of the client c, and therefore it can only be used by that client. To overcome this difficulty we need to allow process types to depend on locations and channels. Here the general form will be

$$\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{S}$$

where S is a script type which may depend on the variables $\tilde{x}$. A value of this type takes the form

$$(\tilde{v}, v)$$

where v is some script. But again to emphasise the occurrence of these types we will use the more descriptive syntax

$$v\ \mathsf{with}\ \tilde{v}$$

An example of the use of such types is in the following variation of the client server from (8) above:

$$\begin{array}{ll}
\text{Server:} & s[\![\mathsf{req}?(\xi\ \mathsf{with}\ y : \mathsf{S}_d)\,\mathsf{run}\,\xi \mid *\,\mathsf{news}!\langle_{scandal}\rangle]\!] \\
\text{Client:} & c[\![(\mathsf{newc}\,\mathsf{report}) \\
& \qquad \mathsf{goto}_{\mathsf{req}}\,s.(\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{report}!\langle x\rangle)\ \mathsf{with}\ c \mid \\
& \qquad \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots]\!]
\end{array} \tag{12}$$

with the types

$$\begin{array}{rl}
\mathsf{R}: & \mathsf{thunk} \\
\mathsf{S}_d: & \mathsf{Tdep}(y : \mathsf{I})\,\mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@y] \\
\mathsf{I}: & \mathsf{loc}[\mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle]
\end{array}$$

Here the important point to notice is the server's guardian type at the port req, S_d, no longer mentions any clients name; it can be used by any client which satisfies the types requirements. The server accepts a thunk, of type $\mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@y]$ which must be accompanied by a location of type I to be used in place of the variable y in S_d. A typical client c can generate a new reply channel report and send to the server

- the thunk $\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{report}!\langle x\rangle$
- accompanied by a required location, in this case c.

Let us now see how the overall system typechecks, assuming as usual an environment in which the channel news and ports req, in, have the appropriate types, and that the declaration type of report is $\mathsf{rw}\langle\mathbf{string}\rangle$. At the server let us concentrate on establishing

$$\Gamma \vdash_s \mathsf{req}?(\xi\ \mathsf{with}\ y : \mathsf{S}_d)\,\mathsf{run}\,\xi : \mathbf{proc}$$

This follows by an application of the simple form of (TY-IN) to

$$\Gamma, \{(y, \xi) : (\mathsf{S}_d)@s\} \vdash_s \mathsf{run}\,\xi : \mathbf{proc}$$

Noting that $(\mathsf{S}_d)@s$ is the same as S_d, unravelling the extended environment gives the requirement

$$\Gamma, y : \mathsf{loc},\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@y,\ \xi : \mathsf{th}_y \vdash_s \mathsf{run}\,\xi$$

where th_y is an abbreviation for the type $\mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@y]$. Apriori typing the process $\mathsf{run}\,\xi$ should be straightforward with respect to this environment. But there is a subtlety; at some point in establishing this judgement we need to apply (TY-BASE) from Figure 6 to conclude

$$\Gamma, y : \mathsf{loc},\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@y,\ \xi : \mathsf{th}_y \vdash_s () : \mathbf{unit}$$

and this requires the premise

$$\Gamma, y : \mathsf{loc},\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@y,\ \xi : \mathsf{th}_y \vdash \mathbf{env}$$

which in turn requires the premise

$$\Gamma, y : \mathsf{loc},\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@y \vdash \mathsf{th}_y : \mathbf{ty} \tag{13}$$

In other words we have to check that th_y is a well-defined type, relative to the extended environment. However this is now straightforward using the rule (SUB-PROC) from Figure 11, in the presence of the new associations involving y in the extended environment.

Let us now turn our attention to typechecking the client in (12) above, where we concentrate on ensuring that the process sent to the port req satisfies the type S_d. We have to ensure

$$\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c \vdash_s (\lambda\,().\ \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{report}!\langle x\rangle)\ \mathsf{with}\ c\ \ : \mathsf{S}_d$$

The rule (TY-TUDEP) in Figure 6 reduces this to two premises:

$$\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c \vdash_s c : \mathsf{l}$$
$$\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c \vdash_s \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{report}!\langle x\rangle\ : \\ \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

The first is immediate from our assumptions about Γ and the second is essentially the same as a derivation we have already seen on page 17.

Thus using dependent process types we can define general purpose servers which can be used by multiple clients. The example we have just considered, (12), apriori leaves the clients insecure because of the use of the liberal type thunk for the clients guardian type R. But it can be generalised so that this guardian is strengthened, allowing in only threads which are going to write to the locally declared reporting channel. Here is one possible formulation:

$$\begin{array}{ll} \text{Server:} & s[\![\mathsf{req}?(\xi\ \mathsf{with}\ (y,z,x) : \mathsf{S}_d)\,\mathsf{run}\,\xi\ \mid\ *\,\mathsf{news}!\langle_{scandal}\rangle]\!] \\ \text{Client:} & c[\![(\mathsf{newc}\,\mathsf{report}) \\ & \quad (\mathsf{newc}\,\mathsf{in} : \mathsf{rw}\langle\mathsf{R}\rangle) \\ & \qquad \mathsf{goto}_{\mathsf{req}}\,s.(\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{report}!\langle x\rangle)\ \mathsf{with}\ (c, \mathsf{report}, \mathsf{in})\ \mid \\ & \qquad \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi\ \mid \mathsf{report}?(y)\ldots]\!] \end{array} \tag{14}$$

Here a client generates a local channel report, whose type $\mathsf{rw}\langle\mathbf{string}\rangle$ we have elided, and a local port in whose declaration type is $\mathsf{rw}\langle\mathsf{R}\rangle$, where R is the more restrictive guardian type $\mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c]$. In other words in has been specially created to restrict entry to processes which will only write on the newly created channel report. The client then sends the usual process to the server but now accompanies it with the triple $(c, \mathsf{report}, \mathsf{in})$

The code for the server is the same except that accompanying the incoming thread it expects three values. Its guardian type S_d however is changed to

$$\begin{array}{ll} \mathsf{S}_d : & \mathsf{Tdep}(y : \mathsf{loc},\ z : \mathsf{w}\langle\mathbf{string}\rangle@y,\ x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y) \\ & \qquad \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y] \end{array}$$

Here, once more, this guardian type does not mention any client names, but it allows clients to employ much more restrictive guardian types at their own sites. We leave the reader to check that this revised system can still be typechecked.

4.6. Existential process types

The use of dependent script types, as in the previous subsection, has certain disadvantages from the point of view of the clients. For example in (14) above the client sends to the server, in addition to the script to be executed, the triple $(c, \mathsf{report}, \mathsf{in})$. These values are used by the server we have defined only as part of the received script. But servers are in principle able to use them in any way they deem fit. An alternative server could be given by

$$\text{badServer:} \qquad s[\![\mathsf{req}?(\xi \text{ with } (y,z,x) : \mathsf{S}_d)\, \mathsf{goto}_x\, y.z!\langle \mathit{boring}\rangle]\!] \tag{15}$$

This rogue server does not run the incoming script to obtain the latest news; instead it uses the incoming accompanying values and sends directly to the client some boring data.

Existential types allow the client to hide from the server the data which accompanies the incoming scripts. These type take the form

$$\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{S}$$

where, as with dependent types, the type of the script S may depend on the parameters $\tilde{x}$. Intuitively a value of this type is once more a form of tuple $(\tilde{v}, v)$, although access to the accompanying parameters is restricted. That is reading a value of this type from a port only results in the script being obtained, although that script itself may use these parameters. This new form of tuple, often called a *package*, is denoted by

$$\langle \tilde{v}, v \rangle$$

The important point about such a package is that it only gives access to the script v and not the internal parameters $\tilde{v}$. In our formulation to send such a value on a channel the sender must have the package $\langle \tilde{v}, v\rangle$, although only the script v is emitted. For this reason we need a special output rule for existential types; see (TY-OUTE) in Figure 7, which has already been explained in Section 3.3.

Let us now reformulate (14) above using existential types:

$$\begin{array}{ll} \text{Server:} & s[\![\mathsf{req}?(\xi : \mathsf{S}_e)\, \mathsf{run}\, \xi \mid *\mathsf{news}!\langle \mathit{scandal}\rangle]\!] \\ \text{Client:} & c[\![(\mathsf{newc}\, \mathsf{report}) \\ & \quad (\mathsf{newc}\, \mathsf{in} : \mathsf{rw}\langle \mathsf{R}\rangle) \\ & \quad\quad \mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\, \mathsf{goto}_{\mathsf{in}}\, c.\, \mathsf{report}!\langle x\rangle \mid \\ & \quad\quad \mathsf{in}?(\xi : \mathsf{R})\, \mathsf{run}\, \xi \mid \mathsf{report}?(y) \ldots]\!] \end{array} \tag{16}$$

Here the guardian type S_e is

$$\begin{array}{l} \mathsf{Edep}(y : \mathsf{loc}, z : \mathsf{w}\langle \mathbf{string}\rangle @y, x : \mathsf{w}\langle \mathsf{th}[z : \mathsf{w}\langle \mathbf{string}\rangle @y]\rangle @y) \\ \qquad \mathsf{th}[\mathsf{news} : \mathsf{r}\langle \mathbf{string}\rangle @s,\ x : \mathsf{w}\langle \mathsf{th}[z : \mathsf{w}\langle \mathbf{string}\rangle @y]\rangle @y] \end{array}$$

The server is much the same as before except that it does not receive any parameters with the incoming script. Similarly the client only sends the script.

Let us now see that this example typechecks. Establishing that the server is well-typed is a little more complicated than with dependent type S_d. The interest centres on establishing

$$\Gamma, \{\xi : (\mathsf{S}_e)@s\} \vdash_s \mathsf{run}\, \xi : \mathbf{proc}$$

and there are two essential steps. Note that, as with S_d, $(\mathsf{S}_e)@s$ is the same as S_e, and so in the sequel we will omit the $(-)@s$. The first step is deriving

$$\Gamma, \{\xi : \mathsf{S}_e\} \vdash_s () : \mathbf{unit}$$

and proceeds as with the use of S_d on page 20; unravelling the environment this amounts to establishing

$$\Gamma, \xi : \langle \mathsf{th}_y \text{ with } y : \mathsf{loc},\ z : \mathsf{w}\langle \mathbf{string}\rangle @y, x : \mathsf{w}\langle \mathsf{th}[z : \mathsf{w}\langle \mathbf{string}\rangle]\rangle @y\rangle \vdash \mathbf{env} \tag{17}$$

where now th_y represents $\mathsf{th}[\mathsf{info} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle]\rangle@y]$. Here the relevant type formation rule is (E-EDEP) from Figure 10, which requires the premise

$$\Gamma, y : \mathsf{loc},\ z : \mathsf{w}\langle\mathbf{string}\rangle@y, x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle]\rangle@y \vdash \mathsf{th}_y : \mathbf{ty}$$

However this is easily established from (SUB-SCRIPT) of the same Figure.

The second essential step in typechecking the server is

$$\Gamma, \{\xi : \mathsf{S}_e\} \vdash_s \xi : \mathbf{proc} \tag{18}$$

This is necessary in order to ensure that run can be applied to ξ. Here we use an application of (TY-ELOOKUP) from Figure 6 to obtain

$$\Gamma, \{\xi : \mathsf{S}_e\} \vdash_s \xi : \mathsf{th}_y$$

One can also establish, using the subtyping rules,

$$\Gamma, \{\xi : \mathsf{S}_e\} \vdash \mathsf{th}_y <: \mathbf{proc}$$

and therefore by (TY-SUBVAL) from Figure 6 we obtain the required judgement (18) above.

Now let us examine the client. The central point is to ensure that the $\mathsf{goto}_{\mathsf{req}}\ s.\ldots$ command is well-typed, which amounts to establishing the judgement:

$$\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c \vdash_s \mathsf{req}!\langle \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\ c.\,\mathsf{report}!\langle x\rangle\rangle \quad : \mathbf{proc}$$

Here the relevant rule is (TY-OUTE) from Figure 7. The second premise follows from our assumption about the type of req at s while the third is vacuous as π is instantiated to $\mathbf{proc}$. However the first premise requires us to find some $\tilde{v}$ such that

$$\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c \vdash_s \langle \tilde{v}, \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\ c.\,\mathsf{report}!\langle x\rangle\rangle \quad : \mathsf{S}_e \tag{19}$$

In fact the required $\tilde{v}$ is obviously going to be $(c, \mathsf{report}, \mathsf{in})$.

With these values the judgement (19) can be established using the rule (TY-EDEP) from Figure 6. This requires the following four premises, where for convenience we use Γ_e as an abbreviation for the extended environment $\Gamma, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle@c, \mathsf{in} : \mathsf{rw}\langle\mathsf{R}\rangle$. Recall that R is the type $\mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle]$.

$$\Gamma_e \vdash_s c : \mathsf{loc}$$
$$\Gamma_e \vdash_s \mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c$$
$$\Gamma_e \vdash_s \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c$$
$$\Gamma_e \vdash_s \mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\ c.\,\mathsf{report}!\langle x\rangle$$
$$: \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

The first three are simple value judgements and we have already seen a derivation of the last.

This ends our consideration of the client/server in (16) above. But let us reconsider the badServer from (15) above. Using existential types this example might be written

$$\text{badServer:} \qquad s[\![\mathsf{req}?(\xi : \mathsf{S}_e)\,\mathsf{goto}_x\ y.z!\langle \mathit{boring}\rangle]\!]$$

But one can show that this no longer typechecks. The problem arises when trying to establish

$$\Gamma, \{\xi : \mathsf{S}_e\} \vdash_y x!\langle z!\langle \mathit{boring}\rangle\rangle \tag{20}$$

We have already seen the expanded environment in (17) above, which is

$$\Gamma, \xi : \langle \mathsf{th}_y \text{ with } y : \mathsf{loc},\ z : \mathsf{w}\langle\mathbf{string}\rangle@y, x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle]\rangle@y\rangle$$

However the only way to get information from the package

$$\xi : \langle \mathsf{th}_y \text{ with } y : \ldots, z : \ldots, x : \ldots\rangle$$

in this environment is to use the rule (TY-UNPACK) from Figure 6. This will only give information on the variable ξ whereas the judgement (20) requires information on the other components of the package y, z, x which are inaccessible.

4.7. Script types

In all of the examples so far servers react to data furnished directly from clients. The general form of script types,

$\mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi)$,

allow servers to accept *parameterised* scripts, which can be instantiated by data owned, or trusted, by the server itself. Consider the following variation on the client used in (8):

$$\begin{array}{ll} \text{Client:} & c[\![\mathsf{goto}_{\mathsf{req}}\, s.F \;\mid \mathsf{in}?(\xi : \mathsf{R})\, \mathsf{run}\, \xi \;\mid \mathsf{report}?(y) \ldots]\!] \\ F = & \lambda\, y : \mathsf{w}\langle \mathbf{string} \rangle.\; y?(x)\, \mathsf{goto}_{\mathsf{in}}\, c.\, \mathsf{report}!\langle x \rangle \end{array}$$

It does not know the source of the news at the server; so it sends the script F there, a script which uses the pre-existing port and channel $\mathsf{in}, \mathsf{report}$, but is parameterised on an information channel local to the server. The server inputs the script and is now free to apply it to whatever information source it deems fit. A simple server, with the same functionality as that in (8), is given by

$$\text{Server:} \quad s[\![\mathsf{req}?(\xi : \mathsf{S}_s)\, \xi(\mathsf{news}) \;\mid\; *\, \mathsf{news}!\langle {\scriptstyle scandal} \rangle]\!]$$

It simply applies the incoming script to the local channel news. However it could also dynamically generate the local news channel, along the lines of

$$\text{ServerDy:} \quad s[\![\mathsf{req}?(\xi : \mathsf{S}_s)\, \mathsf{latest}?(z)\, (\xi\; z)]\!]$$

Note that when F is received by the server and instantiated, the type of the resulting process is dependent on that of the channel to which F is applied. Under the assumptions in place during the discussion of (8), and assuming that foo is a local channel, one would expect the process (F foo), running at s, to behave in accordance with the type

$$\mathsf{pr}[\mathsf{foo} : \mathsf{r}\langle \mathbf{string} \rangle @ s, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c]$$

This is indeed the case as F can be assigned the parameterised type

$$\mathsf{Fdep}(y : \mathsf{r}\langle \mathbf{string} \rangle \to \mathsf{pr}[y : \mathsf{r}\langle \mathbf{string} \rangle @ \mathsf{here}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c]) \tag{21}$$

To see this let Γ be as described on page 16. Then, using a simple variation on the inference described there, we can infer

$$\Gamma, y : \mathsf{r}\langle \mathbf{string} \rangle @ s \vdash_s y?(x)\, \mathsf{goto}_{\mathsf{in}}\, c.\, \mathsf{report}!\langle x \rangle \;:\; \mathsf{pr}[y : \mathsf{r}\langle \mathbf{string} \rangle @ \mathsf{here}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c]$$

An application of (TY-ABS) from Figure 7 gives the required

$$\Gamma \vdash_s F \;:\; \mathsf{Fdep}(y : \mathsf{r}\langle \mathbf{string} \rangle \to \mathsf{pr}[y : \mathsf{r}\langle \mathbf{string} \rangle @ \mathsf{here}, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c])$$

Under the further assumption that $\Gamma \vdash_s \mathsf{foo} : \mathsf{r}\langle \mathbf{string} \rangle$ an application of (TY-BETA) gives

$$\Gamma \vdash_s (F\; \mathsf{foo}) \;:\; \mathsf{pr}[\mathsf{foo} : \mathsf{r}\langle \mathbf{string} \rangle @ s, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c]$$

Following this discussion it should be apparent that to ensure that the overall system is well-typed it is sufficient to use the dependent type (21) above for the guardian type S_s. Then it is easy to check

$$\Gamma \vdash \text{Client} \mid \text{Server}$$

For example typing the server involves establishing

$$\Gamma, \xi : \mathsf{S}_s \vdash_s (\xi\; \mathsf{news}) : \mathbf{proc} \tag{22}$$

Assuming that $\Gamma \vdash_s \mathsf{news} : \mathsf{r}\langle \mathbf{string} \rangle$, we have already seen that an application of (TY-BETA) gives

$$\Gamma, \xi : \mathsf{S}_s \vdash_s (\xi\; \mathsf{news}) : \mathsf{pr}[\mathsf{news} : \mathsf{r}\langle \mathbf{string} \rangle @ s, \mathsf{in} : \mathsf{w}\langle \mathsf{thunk} \rangle @ c]$$

and the required (22) follows by subtyping.

These parameterised functional types can be used in conjunction with the other constructions we have considered, dependent and existential types, to give a very sophisticated language for guardian types which on the one hand allows non-trivial interaction between types, and on the other enables sites to protect their local resources by implementing powerful dynamic access policies. As a final example, to indicate the potential of these types, consider the the following variation on the client in (16), which is in turn an elaboration of the example we have just considered:

$$\begin{array}{ll}
\text{Server:} & s[\![\mathsf{req}?(\xi : \mathsf{S}_{se})\,(\xi\ \mathsf{news}) \mid *\,\mathsf{news}!\langle_{scandal}\rangle]\!] \\
\text{Client:} & c[\![(\mathsf{newc}\,\mathsf{report}) \\
& \quad (\mathsf{newc}\,\mathsf{in} : \mathsf{rw}\langle\mathsf{R}\rangle) \\
& \qquad \mathsf{goto}_{\mathsf{req}}\, s.F \mid \\
& \qquad \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots \\
F = & \lambda\, y : \mathsf{w}\langle\mathbf{string}\rangle.\ y?(x)\,\mathsf{goto}_{\mathsf{in}}\, c.\,\mathsf{report}!\langle x\rangle]\!]
\end{array}$$

Here the client does not know the source of the news at the server, and at the same time the server is not aware of the reply mechanisms in place at the client; indeed these are generated dynamically by the client and used to construct the script F to be sent to the server. One can show that this system is well-typed if we let the guardian type for the client and server to be

$$\begin{array}{ll}
\mathsf{R}: & \mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@c] \\
\mathsf{S}_{se} & \mathsf{Fdep}(w : \mathsf{r}\langle\mathbf{string}\rangle \to \mathsf{S}_e^w)
\end{array}$$

respectively, where S_e^w is the existential type

$$\begin{array}{l}
\mathsf{Edep}(y : \mathsf{loc}, z : \mathsf{w}\langle\mathbf{string}\rangle@y, x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y) \\
\qquad \mathsf{th}[w : \mathsf{r}\langle\mathbf{string}\rangle@s,\ x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y]
\end{array}$$

5. Subject Reduction

Many of the expected properties can be derived for our type inference system. To state these succinctly it will useful to use

$$\Gamma \vdash_w J : \mathsf{T}$$

to denote either a value judgement $\Gamma \vdash_w v : \mathsf{T}$ or a process judgement $\Gamma \vdash_w P : \mathsf{T}$. We will confine our attention to judgements in which Γ contains no occurrences of the special symbol here; thus they will only occur as part of dependent types $\mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi)$ and note that in applications of the rule (TY-ABS) from Figure 7 they are eliminated.

Proposition 5.1 (Sanity Checks).

- $\Gamma \vdash_w J : \mathsf{T}$ implies $\Gamma \vdash \mathbf{env}$.
- $\Gamma \vdash_w P : \pi$ implies $\Gamma \vdash \pi : \mathbf{ty}$

Proof. The first is proved by induction on the inference of $\Gamma \vdash_w J : \mathsf{T}$ while the second is on that of the inference of $\Gamma \vdash_w P : \pi$. It is required by the base case (TY-STOP) while in the cases (TY-OUT), (TY-OUTE), (TY-IN) and (TY-SUB) it follows from the corresponding result for subtyping, Proposition A.1. All other cases follow by induction except (TY-BETA). There we have $\Gamma \vdash F\ (\tilde{v}) : \pi\{\!|^{\tilde{v}}/_{\tilde{x}}|\!\}\{\!|^{w}/_{\mathsf{here}}|\!\}$ because

(i) $\Gamma \vdash_w v_i : \mathsf{T}_i$

(ii) $\Gamma \vdash_w F : \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi)$

The latter can only be inferred by (TY-ABS) from which we know that $\Gamma, \{\tilde{x} : (\tilde{\mathsf{T}})@w\} \vdash_w P : \pi\{\!|^{w}/_{\mathsf{here}}|\!\}$. By the induction hypothesis we have that $\Gamma, \{\tilde{x} : (\tilde{\mathsf{T}})@w\} \vdash \pi\{\!|^{w}/_{\mathsf{here}}|\!\} : \mathbf{ty}$. It follows by the substitution result, Proposition A.5, applied to (i), that $\Gamma \vdash \pi\{\!|^{w}/_{\mathsf{here}}|\!\}\{\!|^{\tilde{v}}/_{\tilde{x}}|\!\} : \mathbf{ty}$. However since we know that w is different than each x_i this type is $\pi\{\!|^{\tilde{v}}/_{\tilde{x}}|\!\}\{\!|^{w}/_{\mathsf{here}}|\!\}$. $\square$

In a similar vein we can show that well-typed processes can only use well-defined types. For example if $\Gamma \vdash_w u?(X : \mathsf{T})\,P : \mathbf{proc}$ then $\Gamma \vdash \mathsf{T} : \mathbf{ty}$.

Environments can be ordered by their ability to assign types to identifiers: $\Gamma_1 <: \Gamma_2$ if for every identifier u, $\Gamma_2 \vdash_w u : \mathsf{T}$ implies $\Gamma_1 \vdash_w u : \mathsf{T}$. We will write $\Gamma_1 \equiv \Gamma_2$ whenever $\Gamma_1 <: \Gamma_2$ and $\Gamma_2 <: \Gamma_1$.

Proposition 5.2.

- (Weakening) Suppose $\Gamma_2 \vdash_w J : \mathsf{T}$ and $\Gamma_1 <: \Gamma_2$ for some Γ_1 such that $\Gamma_1 \vdash \mathbf{env}$. Then $\Gamma_1 \vdash_w J : \mathsf{T}$.
- (Strengthening) Suppose $\Gamma, u : \mathsf{T} \vdash_w J : \mathbf{proc}$, where u does not occur free in J. Then $\Gamma \vdash_w J : \mathbf{proc}$.
- (Subtyping) Suppose $\Gamma \vdash_w J : \mathsf{T}$. Then $\Gamma \vdash \mathsf{T} <: \mathsf{T}'$ implies $\Gamma \vdash_w J : \mathsf{T}'$

Proof. The first two statements are proved by induction on the inferences. The third follows immediately from (TY-SUBPROC) in Figure 7 and (TY-SUBVAL) in Figure 6. □

Multiple occurrences of an identifier is governed by the following result:

Proposition 5.3. $\Gamma \vdash_w u : \mathsf{C}_1@w_1$ and $\Gamma \vdash_w u : \mathsf{C}_2@w_2$ implies $\Gamma \vdash_w u : \mathsf{rc}\langle\mathsf{D}\rangle$ for some D such that $\Gamma \vdash \mathsf{D} <: \mathsf{C}_1,\ \mathsf{D} <: \mathsf{C}_2$.

Proof. This property is essentially enforced by the formation rules for well-defined environments. These ensure that if $\Gamma_1, u : \mathsf{C}_1@w_1, \ldots, u : \mathsf{C}_2@w_2, \ldots$ is a well-defined environment then Γ_1 must contain an entry $u : \mathsf{rc}\langle\mathsf{D}\rangle$, where $\Gamma_1 \vdash \mathsf{D} <: \mathsf{C}_1$ and $\Gamma_1, u : \mathsf{C}_1@w_1, \ldots \vdash \mathsf{D} <: \mathsf{C}_2$.

The formal proof is by induction on the inferences of $\Gamma \vdash_w u : \mathsf{C}_1@w_1$ and $\Gamma \vdash_w u : \mathsf{C}_2@w_2$. The base case, when both are inferred from (TY-LOOKUP), depends on this property of well-defined environments. □

An interesting consequence of this result is that whenever the conditions of the proposition hold $\mathsf{C}_1 \sqcap \mathsf{C}_2$ is guaranteed to exist. This is spelled out in detail in Proposition A.2 in the Appendix.

As usual the proof of Subject Reduction relies on the fact that, in a suitable sense, type inference is preserved under substitutions. We require two such results, one for standard values, and one for the existential values used in type inference.

Lemma 5.4 (Substitution). Suppose $\Gamma \vdash_{w_1} v : \mathsf{T}$ with x not in Γ. Then $\Gamma, x : (\mathsf{T})@w_1, \Delta \vdash_{w_2} J : \mathsf{T}$ implies $\Gamma, \Delta\{\!|^v/_x|\!\} \vdash_{w_2\{\!|^v/_x|\!\}} J\{\!|^v/_x|\!\} : \mathsf{T}\{\!|^v/_x|\!\}$

Proof. First note that the entry $x : (\mathsf{T})@w_1$ can only take one of the four forms, a channel registration, $x : \mathsf{rc}\langle\mathsf{D}\rangle$, a location declaration $x : \mathsf{loc}$, a channel declaration, $x : \mathsf{C}@w'$ or a script declaration $x : \mathsf{S}$. The proof is by induction on the inference of $\Gamma, x : (\mathsf{T})@w_1, \Delta \vdash_{w_2} J : \mathsf{T}$, which can use the rules from Figure 6 or Figure 7. For convenience we use α' to denote $\alpha\{\!|^v/_x|\!\}$ for the various syntactic categories α. Also we use Γ_e as an abbreviation for the environment $\Gamma, x : (\mathsf{T})@w_1, \Delta$. First let us look at some cases from Figure 6.

- Suppose (TY-LOOKUP) is used. So $\Gamma_e \vdash_{w_2} u : \mathsf{E}$ because

  (i) $\Gamma_e \vdash \mathbf{env}$

  (ii) Γ_e has the form $\Gamma_1, u : (\mathsf{E})@w_2, \ldots$.

  The substitution result for well-defined environments, Proposition A.5 in the appendix, ensures that

  (i') $\Gamma, \Delta' \vdash \mathbf{env}$

  To obtain the corresponding

  (ii') Γ, Δ' has the form $\Delta_1, u' : (\mathsf{E}')@w_2', \ldots$

  we perform a case analysis on where $u : (\mathsf{E})@w_2$ occurs in Γ_e; with (i') and (ii') an application of the rule (TY-LOOKUP) gives the required $\Gamma \vdash_{w_2'} u' : \mathsf{E}'$.

  If it occurs in Γ then (ii') is immediate since the substitutions have no effect. If it occurs in Δ then $u' : (\mathsf{E}')@w_2'$ occurs in Δ' and so (ii') holds. Finally $u : (\mathsf{E})@w_2$ could coincide with $x : (\mathsf{T})@w_1$. There are now a number of cases, depending on the form of $(\mathsf{T})@w_1$. As an example suppose it is $\mathsf{C}@w_1$. Then w_1 and w_2 coincide and x can not appear in C, w_1. Therefore the hypothesis $\Gamma \vdash_{w_1} v : \mathsf{C}$ gives the required result, $\Gamma, \Delta' \vdash_{w_2} v : \mathsf{C}$, by Weakening.
- The case (TY-ELOOKUP) is very similar, although there are only two rather than three possibilities for the occurrence of the association in Γ_e.
- Suppose (TY-LOC) is used. So $\Gamma_e \vdash_{w_2} w : \mathsf{K}$, where K is the type $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n]$ because

  (i) $\Gamma_e \vdash_w u_i : \mathsf{C}_i$

  (ii) $\Gamma_e \vdash_{w_2} u_i : \mathsf{rc}\langle\mathsf{D}_i\rangle$

  (iii) $\Gamma_e \vdash \mathsf{D}_i <: \mathsf{C}_i$

Induction, and the substitution result for subtyping, Proposition A.5 in the Appendix, can be applied to these to obtain

(i') $\Gamma, \Delta' \vdash_{w'} u'_i : \mathsf{C}'_i$
(ii') $\Gamma, \Delta' \vdash_{w'_2} u'_i : \mathsf{rc}\langle \mathsf{D}'_i \rangle$
(iii') $\Gamma, \Delta' \vdash \mathsf{D}'_i <: \mathsf{C}'_i$

The interesting case is when both v and x occur in $u_1, \ldots u_n$; without loss of generality suppose these are u_1, u_2 respectively, in which case $u'_1 = u'_2 = v$. Then we know, by Proposition 5.3, that $\mathsf{C}'_1 \sqcap \mathsf{C}'_2$ exists and K' is $\mathsf{loc}[u'_2 : (\mathsf{C}'_1 \sqcap \mathsf{C}'_2), \ldots]$. Applying the rule (TY-MEET) to (i') above gives $\Gamma, \Delta' \vdash_{w'} u'_2 : (\mathsf{C}'_1 \sqcap \mathsf{C}'_2)$ and therefore we can apply (TY-LOC) to this, together with (i'), (ii') and (iii') to obtain the required $\Gamma, \Delta' \vdash_{w'_2} w' : \mathsf{K}'$.

The other cases from Figure 6 are similar, mostly following by induction. Now let us look at some cases from Figure 7.

- Suppose (TY-NEWLOC) is used so $\Gamma_e \vdash_{w_2} (\mathsf{newloc}\, k : \mathsf{K})\, \mathsf{with}\, C \;\mathsf{in}\; P : \pi$ because

  (i) $\Gamma_e, \{k : \mathsf{K}\} \vdash_k C : \pi$
  (ii) $\Gamma_e, \{k : \mathsf{K}\} \vdash_{w_2} P : \pi$
  (iii) $\Gamma_e, \{k : \mathsf{K}\} \vdash_{w_2} k : \mathsf{K}$

  Induction can be applied to each of these, to obtain

  (i') $\Gamma, \Delta', (\{k : \mathsf{K}\})' \vdash_k C' : \pi'$
  (ii') $\Gamma, \Delta', (\{k : \mathsf{K}\})' \vdash_{w'_2} P' : \pi'$
  (iii') $\Gamma, \Delta', (\{k : \mathsf{K}\})' \vdash_{w'_2} k : \mathsf{K}'$

  Unfortunately it is not true in general that $\Gamma, \Delta', (\{k : \mathsf{K}\})'$ is the same as $\Gamma, \Delta', (\{k : \mathsf{K}'\})$. For example if K is $\mathsf{loc}[x : \mathsf{C}'_1, v : \mathsf{C}'_2, \ldots]$ then the former contains the entries $\ldots k : \mathsf{loc}, v : \mathsf{C}'_1 @ k, v : \mathsf{C}'_2 @ k, \ldots$ whereas the latter contains $\ldots k : \mathsf{loc}, v : (\mathsf{C}'_1 \sqcap \mathsf{C}'_2) @ k, \ldots$. Nevertheless it will always be the case that

  $$\Gamma, \Delta', (\{k : \mathsf{K}\})' \equiv \Gamma, \Delta', (\{k : \mathsf{K}'\})$$

  and therefore by Weakening (i'),(ii') and (iii') apply also to the latter. So (TY-LOC) can be applied to these to obtain the required

  $$\Gamma, \Delta' \vdash_{w'_2} (\mathsf{newloc}\, k : \mathsf{K}')\, \mathsf{with}\, C' \;\mathsf{in}\; P' : \pi'$$

- Suppose (TY-IN) is used. So $\Gamma \vdash_{w_2} u?(X : \mathsf{U})\, P : \pi$ because

  (i) $\Gamma_e \vdash \mathsf{pr}[u : \mathsf{r}\langle \mathsf{U} \rangle @ w_2] <: \pi$
  (ii) $\Gamma_e, \{X : (\mathsf{U}) @ w_2\} \vdash_{w_2} P : (\pi \sqcup \mathsf{pr}_{\mathsf{ch}}[X : (\mathsf{U}) @ w_2])$

  Applying the substitution result for subtyping, Proposition A.5, we get

  (i') $\Gamma, \Delta' \vdash \mathsf{pr}[u' : \mathsf{r}\langle \mathsf{U}' \rangle @ w'_2] <: \pi'$

  since $(\mathsf{pr}[u : \mathsf{r}\langle \mathsf{U} \rangle @ w_2])'$ is $\mathsf{pr}[u' : \mathsf{r}\langle \mathsf{U}' \rangle @ w'_2]$. Applying induction to (ii) gives

  (ii') $\Gamma, \Delta', (\{X : (\mathsf{U}) @ w_2\})' \vdash_{w'_2} P' : (\pi \sqcup \mathsf{pr}_{\mathsf{ch}}[X : (\mathsf{U}) @ w_2])'$

  Now substitutions distribute over $\sqcup$ (see Proposition A.3 in the Appendix), and also over the channel extraction function (See Proposition A.4). So this may be rewritten

  (ii') $\Gamma, \Delta', (\{X : (\mathsf{U}) @ w_2\})' \vdash_{w'_2} P' : (\pi' \sqcup \mathsf{pr}_{\mathsf{ch}}[X : (\mathsf{U}') @ w'_2])$

  as x is guaranteed not to be in the pattern X. As in the previous case, we can show that

  $$\Gamma, \Delta', (\{X : (\mathsf{U}) @ w_2\})' \equiv \Gamma, \Delta', \{X : (\mathsf{U}') @ w'_2\}$$

  although because of location types they may not be identical. Nevertheless this is sufficient to be able to apply (TY-IN) to (i'),(ii') to obtain the required $\Gamma, \Delta' \vdash_{w'_2} u?(X : \mathsf{U}')\, P' : \pi$ ■

  □

This substitution result can be generalised to arbitrary patterns, but we only require it in a special case:

Corollary 5.5. Let X be a pattern and suppose $\Gamma \vdash_{w_1} V : \mathsf{T}$ where T is not an existential type. Then $\Gamma, \{X : (\mathsf{T})@w_1\} \vdash_{w_2} J : \mathsf{T}$ implies $\Gamma \vdash_{w_2\{V/X\}} J\{\!|V/X|\!\} : \mathsf{T}\{\!|V/X|\!\}$

Proof. By induction on the structure of T. The base cases are covered by the previous lemma. There are two other cases, when T is a location type and when it is a dependent type. As an example we consider the former, when it has the form $\mathsf{K} = \mathsf{loc}[u_1 : \mathsf{C}_1, \dots u_n : \mathsf{C}_n]$; in this case X must be a variable x and V and identifier, say v.

So $\Gamma, \{X : (\mathsf{K})@w\}$ is $\Gamma, x : \mathsf{loc}, u_1 : \mathsf{C}_1@x, \dots, u_n : \mathsf{C}_n@x$ which can be written as

$$\Gamma,\ x : \mathsf{loc},\ (u_1 : \mathsf{C}_1@x, \dots, u_n : \mathsf{C}_n@x)$$

So applying the previous lemma we obtain

$$\Gamma, u_1 : \mathsf{C}_1@v, \dots u_n : \mathsf{C}_n@v \vdash_{w_2\{v/x\}} J\{\!|v/x|\!\} : \mathsf{T}\{\!|v/x|\!\}$$

But $\Gamma \vdash_{w_2} v : \mathsf{K}$ means that $\Gamma \vdash_v u_i : \mathsf{C}_i$ for each i. So we see that $\Gamma \equiv \Gamma, u_1 : \mathsf{C}_1@v, \dots u_n : \mathsf{C}_n@v$ from which the required

$$\Gamma \vdash_{w_2\{v/x\}} J\{\!|v/x|\!\} : \mathsf{T}\{\!|v/x|\!\}$$

follows. $\square$

The corresponding result for existential types uses different substitutions into processes and types. The crucial property of existential values is that the use of their *witnesses* is very limited:

Proposition 5.6. Suppose $\Gamma, y : \langle \mathsf{T} \ \mathsf{with}\ \tilde{x} : \tilde{\mathsf{E}}\rangle, \Gamma' \vdash_w J : \mathsf{T}$. Then $x_i \notin \mathsf{fv}(J)$ and x_i does not occur in Γ', w.

Proof. By induction on the inference. Intuitively the result follows from the fact that the only information available, via (TY-ELOOKUP), from the entry $y : \langle \mathsf{T}\ \mathsf{with}\ \tilde{x} : \tilde{\mathsf{E}}\rangle$ is that y has the type T; no information on x_i is available. The proof relies on the corresponding result for well-defined environments and subtyping, Proposition A.6 $\square$

This result provides the central property underlying the substitution result for existential values.

Lemma 5.7 (ESubstitution). Suppose $\Gamma \vdash_{w_1} \langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}$. Then $\Gamma, y : \langle(\mathsf{T})@w_1\ \mathsf{with}\ \tilde{x} : \tilde{\mathsf{E}}\rangle, \Delta \vdash_{w_2} J : \mathsf{T},\ w_2 : \mathsf{loc}$ implies $\Gamma, \Delta\{\!|v/y|\!\} \vdash_{w_2\{v/y\}} J\{\!|v/y|\!\} : \mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}$

Proof. The proof follows the lines of that of Lemma 5.4, with frequent applications of the previous proposition, Proposition 5.6, to ensure that only the substitution of v for x is applied to process terms and names. As usual certain cases depend on the corresponding result for well-typed environments and subtyping judgements, Proposition A.7 in the Appendix. $\square$

Theorem 5.8 (Subject Reduction).

$$\text{Suppose } \Gamma \vdash M. \text{ Then } M \longrightarrow N \text{ implies } \Gamma \vdash N.$$

Proof. It is a question of examining each of the rules in Figure 2 in turn. Note that (R-STR) requires that typing is preserved by the structural equivalence; we leave the proof of this fact to the reader, as it follows the standard approach.

Consider the rule (R-COMM):

$$k[\![c!\langle V\rangle]\!] \mid k[\![c?(X : \mathsf{T})\,P]\!] \longrightarrow k[\![P\{\!|V/X|\!\}]\!]$$

and suppose $\Gamma \vdash k[\![c!\langle V\rangle]\!] \mid k[\![c?(X : \mathsf{T})\,P]\!]$. Because $\mathbf{proc}$ is a top type for processes this means that

(i) $\Gamma \vdash_k c!\langle V\rangle : \mathbf{proc}$
(ii) $\Gamma \vdash_k k[\![c?(X : \mathsf{T})\,P]\!] : \mathbf{proc}$

We need to show $\Gamma \vdash k[\![P\{\!|V/X|\!\}]\!]$ which follows easily if we can establish $\Gamma \vdash_k P\{\!|V/X|\!\} : \mathbf{proc}$.

From (i),(ii), we can show that $\Gamma \vdash_k c : \mathsf{rw}\langle \mathsf{T}, \mathsf{T}\rangle$ and $\Gamma \vdash_k V : \mathsf{T}$. There are now two cases, depending on the structure of T. First suppose it is an existential type $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{U}$, in which case the pattern X is a single variable, say y. Here (i) above can only be inferred by using (TY-OUTE), which means that V is a singleton, say v and there must be some vector $\tilde{v}$ of witnesses such that $\Gamma \vdash_{k} \langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{U}$. Deconstructing (ii) we know that $\Gamma, y : \langle \mathsf{U}\ \mathsf{with}\ \tilde{x} : \tilde{\mathsf{E}}\rangle \vdash_{k} P : \mathbf{proc}$. We may now apply Lemma 5.7 to obtain the required $\Gamma \vdash_{k} P\{\!|^{v}\!/\!_{y}|\!\}$.

When T is not an existential type the proof is similar but uses an application of Corollary 5.5 in place of Lemma 5.7.

We leave the proof for the other rules to the reader. $\square$

6. The behaviour of SAFEDPI systems

In this section we investigate what might be an appropriate notion of semantic equivalence between SAFEDPI systems. We first propose what we believe to be a natural notion of contextual equivalence. Then, in the following sections, we give a coinductive characterisation using actions between configurations, consisting of SAFEDPI systems together with the environment's current knowledge of the system.

For notational convenience we limit ourselves to the case when the only transmission types allowed are of the form

$$\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{A} \qquad \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S} \qquad \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S}$$

Effectively this means that the values transmitted must either be of the form

- $(\tilde{u})$, a tuple of first-order values, of type $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{A}$
- $(\tilde{u}, F)$ a tuple in which the last value F, a script, may depend on the first-order values $(\tilde{u})$. These have a type of the form $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S}$.
- F a script, the final component of an existential value $\langle \tilde{u}, F\rangle$ with a type of the form $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S}$.

Simple scripts may be simulated via the empty dependent type $\mathsf{Tdep}()\,\mathsf{S}$, as can simple first-order values, via the type $\mathsf{Tdep}()\,\mathsf{A}$. Our results extend to the full language, although the proofs require the development of more complicated notations.

6.1. A contextual equivalence

We intend to use a context based equivalence in which systems are asked to be deemed equivalent in all *reasonable* SAFEDPI contexts. What is perhaps not so clear here is the notion of reasonable context. In previous work on mobile calculi, [9, 8, 1], the equivalence took the form

$$\Gamma \models M \approx_{cxt} N$$

meaning, intuitively, that M and N are indistinguishable in any context typeable by the typing environment Γ. Although one is primarily interested in such judgements in which Γ has sufficient knowledge to type M and N, one is led to consider more general judgements where Γ only contains a subset of that knowledge. Such equivalences, for PICALCULUS and DPI, can be characterised inductively using actions of the form

$$(\Gamma \triangleright M) \xrightarrow{\mu} (\Gamma' \triangleright M')$$

where $(\Gamma \triangleright M)$, $(\Gamma \triangleright M')$ are configurations, consisting of systems M, M' and type environments Γ, Γ', representing the current knowledge of the testing context. In general such actions change not only the systems, M to M' but also the current knowledge, from Γ to Γ', typically by adding new information.

However, there are further subtleties which need to be considered in the current setting. We discuss this with a motivating example.

Example 6.1. Consider

$$\begin{array}{l}
M = (\mathsf{new}\, k : \mathsf{loc}[b : \mathsf{rw}\langle \mathbf{unit}\rangle])\, l[\![a!\langle k\rangle]\!] \mid k[\![b!\langle\rangle]\!] \\
N = (\mathsf{new}\, k : \mathsf{loc}[b : \mathsf{rw}\langle \mathbf{unit}\rangle])\, l[\![a!\langle k\rangle]\!] \mid k[\![\mathsf{stop}]\!] \\
\text{and} \\
\Gamma = l : \mathsf{loc}, b : \mathsf{rc}\langle \mathsf{rw}\langle \mathbf{unit}\rangle\rangle, a : \mathsf{rw}\langle \mathsf{loc}[b : \mathsf{rw}\langle \mathbf{unit}\rangle]\rangle@l
\end{array}$$

These two systems are well-typed with respect to Γ and should be considered equivalent under most reasonable notions of behavioural equivalence; it is impossible for a testing process to interact with M on b at k, even after the interaction on a at l. Indeed, consider what form a test which could achieve this must take:

$$- \mid l[\![a?(x)\,\mathsf{goto}_?\,x.b?()]\!]$$

It is clear that there is no port for the testing process to enter the location k on. Moreover, tests cannot be placed directly at k as k is only discovered through interaction.

To sum up we would expect

$$\Gamma \models M \approx_{cxt} N$$

to hold, for an appropriate formulation of contextual equivalence for SAFEDPI. But a naive labelled transition system of the form discussed above would distinguish them. For example a naive system might yield actions such as

$$(\Gamma \triangleright M) \xrightarrow{\text{outputs } k \text{ on } a \text{ at } l} (\Gamma' \triangleright l[\![\mathsf{stop}]\!] \mid k[\![b!\langle\rangle]\!])$$

where Γ' is the environment Γ updated with the knowledge about the new location $k : \mathsf{loc}[b : \mathsf{rw}\langle\mathbf{unit}\rangle]$. However, in such a system, a subsequent interaction at this newly discovered k would be possible. This interaction would suffice to distinguish M and N.

In other words we need to consider more sophisticated notions of actions in order to capture contextual equivalences for SAFEDPI.

It should be clear from this discussion then that in modelling behavioural equivalence in this setting, we must be aware of those locations at which we can, and can not, perform tests. And this is not simply a question of which locations the environment has immigration rights for, via some port.

Example 6.2. Consider the following scenario:

$$M = k[\![(\mathsf{newc}\, b : \mathsf{rw}\langle\mathbf{unit}\rangle)\;\; a!\langle b\rangle \;\mid\; b!\langle\rangle]\!]$$
$$N = k[\![(\mathsf{newc}\, b : \mathsf{rw}\langle\mathbf{unit}\rangle)\;\; a!\langle b\rangle \;\mid\; \mathsf{stop}]\!]$$
and
$$\Gamma = k : \mathsf{loc}, a : \mathsf{rw}\langle\mathsf{rw}\langle\mathbf{unit}\rangle\rangle_{@}k$$

Here the testing environment already knows about k but does not have any immigration rights there. Nevertheless M and N can be distinguished by a reasonable test, one which is typeable by Γ:

$$- \mid k[\![a?(x)\, x?\langle\rangle\, \mathsf{eureka}!\langle\rangle]\!]$$

■

Thus, in representing the environment's knowledge of the system we must also represent the information about which locations are available for direct testing. This motivates the following definition.

Definition 6.3 (Knowledge structures). A *knowledge structure* is a pair $(\Gamma, \mathcal{T})$, where

- Γ is a type environment such that $\Gamma \vdash \mathbf{env}$
- $\mathcal{T}$ is a subset of LOCS such that if $k \in \mathcal{T}$ then $k : \mathsf{loc} \in \Gamma$

We use $\mathcal{I}$ to range over knowledge structures and write $\mathcal{I}_\Gamma$ and $\mathcal{I}_\mathcal{T}$ to refer to the respective components of the structure. We sometimes refer to the locations in $\mathcal{I}_\mathcal{T}$ as those to which the information structure allows *access rights*. We often abuse notation by writing $\mathcal{I}, \Gamma$ to mean the knowledge structure $((\mathcal{I}_\Gamma, \Gamma), \mathcal{I}_\mathcal{T})$. ■

Definition 6.4 (Configurations). We write $\mathcal{I} \triangleright M$ for a *configuration* where

- $\mathcal{I}$ is a knowledge structure
- there exists some Δ such that $\Delta \vdash M$, $\Delta <: \mathcal{I}_\Gamma$, and $\mathsf{dom}(\Delta) = \mathsf{dom}(\mathcal{I}_\Gamma)$. ■

As explained at length in [9], in the configuration $\mathcal{I} \triangleright M$, $\mathcal{I}$ is meant to denote the knowledge which the user has of the capabilities of M, expressed as types. In general this will not be sufficient to fully type M, but M can be typed by an extension of it, namely the implicit Δ in the definition.

Definition 6.5 (Knowledge-indexed relations). We call a family of binary relations between systems indexed by knowledge structures a *knowledge-indexed relation* over systems. We write $\mathcal{I} \models M \, \mathcal{R} \, N$ to mean that systems M and N are related by $\mathcal{R}$ at index $\mathcal{I}$ and moreover, $\mathcal{I} \triangleright M$ and $\mathcal{I} \triangleright N$ are both configurations. ■

We will use knowledge-indexed relations to propose a notion of behavioural equivalence appropriate to this setting. We do this in an established manner [11, 6, 9] by proposing that we consider the largest equivalence closed under certain natural properties listed below.

Reduction closure: We say that a knowledge-indexed relation is *reduction closed* if whenever $\mathcal{I} \models M \, \mathcal{R} \, N$ and $M \longrightarrow M'$ there exists some N' such that $N \longrightarrow^* N'$ and $\mathcal{I} \models M' \, \mathcal{R} \, N'$.

Context closure: We say that a knowledge-indexed relation is *contextual* if

(1) $\mathcal{I} \models M \, \mathcal{R} \, N$ and $\mathcal{I}_\Gamma, k : \mathsf{loc} \vdash_{\mathbf{env}}$ implies $\mathcal{I}' \models M \, \mathcal{R} \, N$ where $\mathcal{I}'$ is $((\mathcal{I}_\Gamma, k : \mathsf{loc}), \mathcal{I}_\mathcal{T} + k)$

(2) $\mathcal{I} \models M \, \mathcal{R} \, N$ and $\mathcal{I}_\Gamma, \Gamma' \vdash_{\mathbf{env}}$ implies $\mathcal{I}, \Gamma' \models M \, \mathcal{R} \, N$

(3) $\mathcal{I} \models M \, \mathcal{R} \, N$ and $\mathcal{I}_\Gamma \vdash k[\![P]\!]$ with $k \in \mathcal{I}_\mathcal{T}$ implies
$\mathcal{I} \models (M \mid k[\![P]\!]) \, \mathcal{R} \, (N \mid k[\![P]\!])$

(4) $\mathcal{I}, \{n : E\} \models M \, \mathcal{R} \, N$ implies $\mathcal{I} \models (\mathsf{new}\, n : \mathsf{E}) \, M \, \mathcal{R} \, (\mathsf{new}\, n : \mathsf{E}) \, N$ ■

In the first condition we are assured that k is a fresh location; therefore this form of weakening allows the environment to create for itself fresh locations at which it may deploy code. The second form of weakening, in (2), allows it to invent new names with which to program processes. Condition (3) allows it to place well-typed code at sites to which it has access rights, while (4) is the standard mechanism for handling names which are private to the systems being investigated.

Barb Preservation: For any given location k and any given channel a such that $k \in \mathcal{I}_\mathcal{T}$ and $\mathcal{I}_\Gamma \vdash_k a : \mathsf{rw}\langle \mathbf{unit} \rangle$ we write $\mathcal{I} \vdash M \Downarrow^{\mathsf{barb}} a@k$ if there exists some M' such that $M \longrightarrow^* M' \mid k[\![a!\langle\rangle]\!]$. We say that a knowledge-indexed relation is *barb preserving* if $\mathcal{I} \models M \, \mathcal{R} \, N$ and $\mathcal{I} \vdash M \Downarrow^{\mathsf{barb}} a@k$ implies $\mathcal{I} \vdash N \Downarrow^{\mathsf{barb}} a@k$.

Definition 6.6 (Reduction barbed congruence). We let $\approx_{cxt}$ be the largest knowledge-indexed relation over systems which is

- pointwise symmetric (that is $\mathcal{I} \models M \approx_{cxt} N$ implies $\mathcal{I} \models N \approx_{cxt} M$)
- reduction closed
- contextual
- barb preserving ■

We take reduction barbed congruence to be our touchstone equivalence for SAFEDPI as it is based on simple observable behaviour respected in all contexts. The definition above is stated relative to choice of the knowledge structure $\mathcal{I}$. We should point out however that, for any given systems M, N and type environment Γ such that $\Gamma \vdash M$ and $\Gamma \vdash N$ then there is a canonical choice of knowledge structure $\mathcal{I}$, namely, $(\Gamma, \mathcal{T}_\Gamma)$ where we let $\mathcal{T}_\Gamma = \{ k \mid k : \mathsf{loc} \in \Gamma \}$. This choice of knowledge structure gives rise to what we feel to be a natural and intuitive notion of equivalence for well-typed SAFEDPI systems.

Of course, the quantification over all contexts makes reasoning about the equivalence virtually intractable. However it is common practise, [19, 21, 1, 9, 8], to provide some sort of model or alternative characterisation in terms of labelled transition systems, which makes the behaviour of systems much more accessible. In particular if the actions in the labelled transition system are sufficiently simple this can lead to automatic, or semi-automatic verification methods.

In the next section we show that this contextual equivalence for SAFEDPI can be characterised in a similar manner, as a bisimulation equivalence over a suitably defined labelled transition system.

6.2. A bisimulation equivalence

We first discuss the labels, or *actions*, to be used in the labelled transition system. They are given by the following grammar:

$$\alpha ::= \tau \mid (\tilde{n}:\tilde{\mathsf{E}})\mathsf{go}_p k.F \mid (\tilde{n}:\tilde{\mathsf{E}})(\tilde{m})k.a.\beta$$
$$\beta ::= V? \mid V!$$

where it is assumed that $k, a, p \notin \tilde{n}, \tilde{m}$. These are intended to be read as follows:

- τ represents *internal* communication in which no interaction with the environment takes place
- $\mathsf{go}_p k.F$ represents an attempt by the environment to enter location k on port p. The code to be deployed, if this attempt succeeds, is given by the script F.
- $k.a.V!$ represents a communication between the system and the environment in which the system exports on channel a at k. The value V in this action depends on the type of the channel. First order values can be recognised by the environment and so they are recorded in the action label. Scripts, on the other hand, can not necessarily be identified. So instead the environment provides a suitable receiving context for a script. For example suppose the system exports some script F on a channel a of script type S. To test F the environment can supply any abstraction G of type $G : S \to \mathbf{proc}$, with which F can be investigated; see rule (M-SEND − SCRIPT) in Figure 8.
- $k.a.V?$ represents a communication between system and environment in which the system imports on channel a at k. The value V is always provided by the environment.
- $(n)\alpha$ represents an action α in which the new name n has been exported from the system; it is new in the sense that it has not previously been encountered by the testing environment. The type of n is not recorded since it can be inferred from the type of the channel on which it is exported.
- $(n : E)\alpha$ represents an action α in which the fresh name n is being provided by the environment.

The following notation is useful in defining the labelled transition system. Firstly, the *subject labels*, $\mathsf{subj}(\alpha)$ of an action are given by:

$$\begin{aligned}&\mathsf{subj}(\tau) = \emptyset\\&\mathsf{subj}((\tilde{n}:\tilde{\mathsf{E}})(\tilde{m})k.a.\beta) = \{k, a\}\\&\mathsf{subj}((\tilde{n}:\tilde{\mathsf{E}})\mathsf{go}_p k.V) = \{k, p\}\end{aligned}$$

Next, we define the *object labels* of an action. These are divided into both input and output object labels using the two functions $\mathsf{obj}_?(\alpha)$ and $\mathsf{obj}_!(\alpha)$ in order to identify whether the names returned are being provided by the environment or exported from the system. We use input object labels to identify the former and output object labels the latter.

$$\begin{array}{ll}\mathsf{obj}_?(\tau) = \emptyset & \mathsf{obj}_!(\tau) = \emptyset\\ \mathsf{obj}_?(\tilde{u}!) = \emptyset & \mathsf{obj}_!(\tilde{u}!) = \mathsf{fn}(\tilde{u})\\ \mathsf{obj}_?(\tilde{V?}) = \mathsf{fn}(V) & \mathsf{obj}_!(\tilde{V?}) = \emptyset\\ \mathsf{obj}_?((\tilde{u},G)!) = \mathsf{fn}(G) & \mathsf{obj}_!((\tilde{u},G)!) = \mathsf{fn}(\tilde{u})\\ & \\ \mathsf{obj}_?((\tilde{n}:\tilde{\mathsf{E}})\mathsf{go}_p k.V) = \mathsf{fn}(V)\setminus\tilde{n} & \mathsf{obj}_!((\tilde{n}:\tilde{\mathsf{E}})\mathsf{go}_p k.V) = \emptyset\\ & \\ \mathsf{obj}_?((\tilde{n}:\tilde{\mathsf{E}})(\tilde{m})k.a.\beta) = \mathsf{obj}_?(\beta)\setminus\tilde{n} & \mathsf{obj}_!((\tilde{n}:\tilde{\mathsf{E}})(\tilde{m})k.a.\beta) = \mathsf{obj}_!(\beta)\setminus\tilde{m}\end{array}$$

The interesting case here is $\beta = (\tilde{u}, G)!$, which represents the export from the system to the environment a higher-order script, dependent on the first-order values $(\tilde{u})$. This exported script is not represented in the label; instead G, which is supplied by the environment, is applied to it. So $\mathsf{obj}_?(\beta)$ is all the free names in G, since these are supplied by the environment, while $\mathsf{obj}_!(\beta)$ are all the identifiers in $\tilde{u}$, since these are supplied by the system.

With this notation we define judgements of the form

$$(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M') \tag{23}$$

representing the effect of the system M performing the action labelled α, in an environment whose knowledge is $\mathcal{I}$. This action changes the system, from M to M', and the knowledge, from $\mathcal{I}$ to $\mathcal{I}'$. Typically this is an

[t]

(M-RECEIVE)
$$\frac{\begin{array}{l} k \in \mathcal{I}_{\mathcal{T}} \\ \mathsf{T} = \bigsqcap \mathcal{I}^w_\Gamma(a,k) \qquad \mathcal{I}^w_\Gamma(a,k) \neq \emptyset \\ \mathcal{I}_\Gamma \vdash_k V : \mathsf{T} \end{array}}{(\mathcal{I} \triangleright k\llbracket a?(X:\mathsf{U})\,P\rrbracket) \xrightarrow{k.a.V?} (\mathcal{I} \triangleright k\llbracket P\{\!|V/X|\!\}\rrbracket)}$$

(M-DELIVER)
$$\frac{\begin{array}{l} k \in \mathcal{I}_{\mathcal{T}} \\ \mathsf{T} = \bigsqcap \mathcal{I}^w_\Gamma(a,k) \qquad \mathcal{I}^w_\Gamma(a,k) \neq \emptyset \\ \mathcal{I}_\Gamma \vdash_k V : \mathsf{T} \end{array}}{(\mathcal{I} \triangleright M) \xrightarrow{k.a.V?} (\mathcal{I} \triangleright M \mid k\llbracket a!\langle V\rangle\rrbracket)}$$

(M-SEND.VAL)
$$\frac{\begin{array}{l} k \in \mathcal{I}_{\mathcal{T}} \qquad\qquad \mathsf{T}\text{a first-order type} \\ \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \\ \mathcal{I}_\Gamma, \{\tilde{u} : (\mathsf{T})@k\} \vdash \mathbf{env} \end{array}}{(\mathcal{I} \triangleright k\llbracket a!\langle\tilde{u}\rangle\rrbracket) \xrightarrow{k.a.\tilde{u}!} (\mathcal{I}, \{\tilde{u} : (\mathsf{T})@k\} \triangleright k\llbracket\mathsf{stop}\rrbracket)}$$

(M-SEND.SCRIPT)
$$\frac{\begin{array}{l} k \in \mathcal{I}_{\mathcal{T}} \qquad\qquad \mathsf{T} \text{ of the form } \mathsf{Edep}(\tilde{x}:\tilde{T})\,S \\ \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \\ \mathcal{I}_\Gamma \vdash_k G : \mathsf{T} \to \mathbf{proc} \end{array}}{(\mathcal{I} \triangleright k\llbracket a!\langle F\rangle\rrbracket) \xrightarrow{k.a.G!} (\mathcal{I} \triangleright k\llbracket G\ (F)\rrbracket)}$$

(M-SEND.DEP.SCRIPT)
$$\frac{\begin{array}{l} k \in \mathcal{I}_{\mathcal{T}} \qquad\qquad \mathsf{T} \text{ of the form } \mathsf{Tdep}(\tilde{x}:\tilde{\mathsf{E}})\,\mathsf{S} \\ \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \\ \mathcal{I}_\Gamma, \{\tilde{u} : (\tilde{\mathsf{E}})@k\} \vdash \mathbf{env} \\ \mathcal{I}_\Gamma \vdash_k G : T \to \mathbf{proc} \end{array}}{(\mathcal{I} \triangleright k\llbracket a!\langle(\tilde{u},F)\rangle\rrbracket) \xrightarrow{k.a.(\tilde{u},G)!} (\mathcal{I}, \{\tilde{u} : (\widetilde{\mathsf{E}})@k\} \triangleright k\llbracket G\ (\tilde{u},F)\rrbracket)}$$

(M-GOTO)
$$\frac{\begin{array}{l} k \notin \mathcal{I}_{\mathcal{T}} \\ \mathcal{I}_\Gamma \vdash_k p!\langle V\rangle : \mathbf{proc} \end{array}}{(\mathcal{I} \triangleright M) \xrightarrow{\mathsf{go}_p k.V} (\mathcal{I} \triangleright M \mid k\llbracket p!\langle V\rangle\rrbracket)}$$

Fig. 8. Labelled Transition System Axioms

increase in knowledge of the testing environment of the system, represented as the change from the type environment, $\mathcal{I}_\Gamma$ to $\mathcal{I}'_\Gamma$.

The axioms for the judgements (23) are given in Figure 8; these are based on the rules in Figure 10 of [8]. We make use of the following notation in the presentation of the rules: for a type environment $\mathcal{I}_\Gamma$ we write

$$\begin{array}{l} \mathcal{I}^r_\Gamma(a,k) = \{\mathsf{T} \mid a : \mathsf{r}\langle\mathsf{T}\rangle@k \in \mathcal{I}_\Gamma \text{ or } a : \mathsf{rw}\langle\mathsf{T},\mathsf{U}\rangle@k \in \mathcal{I}_\Gamma\} \\ \mathcal{I}^w_\Gamma(a,k) = \{\mathsf{U} \mid a : \mathsf{w}\langle\mathsf{U}\rangle@k \in \mathcal{I}_\Gamma \text{ or } a : \mathsf{rw}\langle\mathsf{T},\mathsf{U}\rangle@k \in \mathcal{I}_\Gamma\} \end{array}$$

The input rule (M-RECEIVE) is a mild generalisation of the corresponding rule in [8], given there as (LTS-IN). Note that the action is only possible if the environment has access rights to its location k, that is if k is in $\mathcal{I}_{\mathcal{T}}$. Because SAFEDPI is asynchronous we effectively require a form of asynchronous bisimulation to capture reduction barbed congruence. We in fact prefer to use the standard form of bisimulation, simulating the asynchrony by having a special output rule called (M-DELIVER). This represents the delivery of a value to a channel, although it may not necessarily be consumed; note again that access rights are required to the channels' location.

There are three versions of the second form of output rule, in which the value is consumed by the channel; the variation depends on the type of the channel, but all require access rights. The first, (M-SEND.VAL), for first-order values, is an extension of the corresponding rule, (LTS-OUT), from [8]; note that here the

[t]

(M-RED)
$$\frac{M \longrightarrow M'}{(\mathcal{I} \triangleright M) \xrightarrow{\tau} (\mathcal{I} \triangleright M')}$$

(M-PAR)
$$\frac{(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')}{\begin{array}{l}(\mathcal{I} \triangleright M \mid N) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M' \mid N)\\ (\mathcal{I} \triangleright N \mid M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright N \mid M')\end{array}}$$

(M-NEW)
$$\frac{(\mathcal{I}, n : \top \triangleright M) \xrightarrow{\alpha} (\mathcal{I}', n : \top \triangleright M')}{(\mathcal{I} \triangleright (\mathsf{new}\, n : \mathsf{E})\, M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright (\mathsf{new}\, n : \mathsf{E})\, M')} \quad n \notin \mathsf{n}(\alpha)$$

(M-OPEN)
$$\frac{(\mathcal{I}, m : \top \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')}{(\mathcal{I} \triangleright (\mathsf{new}\, m : \mathsf{E})\, M) \xrightarrow{(m)\alpha} (\mathcal{I}' \triangleright M')} \quad m \notin \mathsf{subj}(\alpha), m \in \mathsf{obj}_!(\alpha)$$

(M-WEAK)
$$\frac{(\mathcal{I}, \{n : \mathsf{E}\} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')}{(\mathcal{I} \triangleright M) \xrightarrow{(n:\mathsf{E})\alpha} (\mathcal{I}' \triangleright M')} \quad n \notin \mathsf{subj}(\alpha), n \in \mathsf{obj}_?(\alpha)$$

(M-τWEAK)
$$\frac{((\mathcal{I}_\Gamma, \{k : \mathsf{K}\}), \mathcal{I}_\mathcal{T} + k) \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')}{(\mathcal{I} \triangleright M) \xrightarrow{(k:\mathsf{K})\alpha} (\mathcal{I}' \triangleright M')} \quad k \notin \mathsf{subj}(\alpha), k \in \mathsf{obj}_?(\alpha)$$

Fig. 9. Labelled Transition System Rules

environment's knowledge is increased, by adding the information contained in $\{\tilde{u} : (\mathsf{T})_@k\}$. Output of scripts is handled by (M-SEND.SCRIPT), where the environment supplies an appropriate G for further investigation of the script F. Dependent scripts, $(\tilde{u}, F)$ are handled by (M-SEND.DEP.SCRIPT); here the values $(\tilde{u})$ are exported from the system to the environment, while G, used for further investigation of F is imported to the system from the environment.

The final rule in Figure 8, (M-GOTO), is novel. It allows the environment to place arbitrary (well-typed) code at a site k, even if it does not have access rights there, provided it knows of a port p at k. Of course, in accordance with our operational semantics, k is free to ignore this code, by not proffering an input at the port p. This rule plays a crucial role in the proof of Proposition 6.10, on page 35.

The inference rules for the action judgements (23) are given in Figure 9, and again they are informed by the corresponding rules in Figure 10 of [8]. Here we abuse notation a little by writing $(m)\alpha$ to mean $(\tilde{n} : \tilde{\mathsf{E}})(m, \tilde{m})\alpha'$ whenever α is $(\tilde{n} : \tilde{\mathsf{E}})(\tilde{m})\alpha'$. Note that, unlike in [8], we have two weakening rules; the new one, (M-τWEAK), allows the environment to invent a new location k at which it has access rights. This is useful in the definability part of the proof of Theorem 6.13.

As a sanity check on these judgements we give a precise description of the possible forms the actions can take; to aid readability we will use G to represent a script furnished by the environment and F to represent one furnished by the system:

Proposition 6.7. Suppose that $\mathcal{I} \triangleright M$ is a configuration from which $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright N)$, where α is not τ. Then α takes one of the following forms:

First-order: input $(\tilde{n} : \tilde{\mathsf{E}})k.a.(\tilde{u})?$, where $(\tilde{n}) \subseteq (\tilde{u})$, or output $(\tilde{m})k.a.(\tilde{u})!$, where $(\tilde{m}) \subseteq (\tilde{u})$

Script: input $(\tilde{n} : \tilde{\mathsf{E}})k.a.F?$, where $(\tilde{n}) \subset \mathsf{fn}(F)$, or output $(\tilde{n} : \tilde{\mathsf{E}})k.a.G!$ where $(\tilde{n}) \subset \mathsf{fn}(G)$

Dependent script: input $(\tilde{n} : \tilde{\mathsf{E}})k.a.(\tilde{u}, F)?$, where $(\tilde{n}) \subseteq (\tilde{u}) \cup \mathsf{fn}(F)$, or output $(\tilde{n} : \tilde{\mathsf{E}})(\tilde{m})k.a.(\tilde{u}, G)!$, where $(\tilde{n}) \subset \mathsf{fn}(G)$ and $(\tilde{m}) \subset (\tilde{u})$

Aynchronous-goto: $(\tilde{n} : \tilde{\mathsf{E}})\mathsf{go}_p k.F$, where $(\tilde{n}) \subseteq \mathsf{fn}(F)$.

Proof. By induction on the inference of $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright N)$. $\square$

Proposition 6.8 (Well-definedness). Suppose $\mathcal{I} \triangleright M$ is a configuration. Then $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright N)$ implies $\mathcal{I}' \triangleright N$ is also a configuration.

Proof. By induction on the inference of $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright N)$, and an analysis of the last rule used; the details are similar to the corresponding result, Proposition 4.4 of [8]; the access rights component of $\mathcal{I}$, $\mathcal{I}_{\mathcal{T}}$ only plays a role in one rule, (M-τWEAK), and even there it is a minor role.

The axiom (M-RECEIVE) requires an application of the substitution results, Corollary 5.5 or Lemma 5.7 depending on the transmission type involved. The remaining axioms are straightforward, as their premises contain sufficient typing information to guarantee that the residual is indeed a configuration.

The proof for the rule (M-RED) depends on Subject Reduction, Theorem 5.8, while that for (M-NEW) relies on Weakening; the remaining rules follow immediately by induction. □

With this result we now have a labelled transition system for SAFEDPI, the nodes being configurations and the actions all judgements (23) which can be inferred from Figure 8 and Figure 9. The standard definition of bisimulation therefore gives a co-inductive relation over configurations:

Definition 6.9 (Bisimulations). We say the binary relation between configurations $\mathcal{R}$ is a *typed bisimulation* if $\mathcal{C}\,\mathcal{R}\,\mathcal{D}$ implies

- $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ implies $\mathcal{D} \xRightarrow{\hat{\alpha}} \mathcal{D}'$ for $\mathcal{D}'$ such that $\mathcal{C}'\,\mathcal{R}\,\mathcal{D}'$
- $\mathcal{D} \xrightarrow{\alpha} \mathcal{D}'$ implies $\mathcal{C} \xRightarrow{\hat{\alpha}} \mathcal{C}'$ for $\mathcal{C}'$ such that $\mathcal{C}'\,\mathcal{R}\,\mathcal{D}'$

where $\xRightarrow{\hat{\alpha}}$ is the standard notation, meaning $\xrightarrow{\tau}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}^{*}$ for α not equal to τ and $\xrightarrow{\tau}^{*}$ otherwise.

We write $\mathcal{I} \models M \approx_{bis} N$ whenever there exists some bisimulation $\mathcal{R}$ such that $(\mathcal{I} \triangleright M)\,\mathcal{R}\,(\mathcal{I} \triangleright N)$. ■

With this notation, that is by viewing the knowledge-structure $\mathcal{I}$ as a parameter, we construe $\approx_{bis}$ to be a knowledge-indexed relation over systems. This enables us to compare it directly with the touchstone behavioural equivalence $\approx_{cxt}$. The main technical property we require of $\approx_{bis}$ is given in the following result:

Proposition 6.10. The knowledge-indexed relation $\approx_{bis}$ is contextual.

Proof. This follows similar lines to the equivalent statement in [8]. For this reason we only show that $\approx_{bis}$ is preserved by parallel composition here. Let $\mathcal{R}$ be defined by

$$(\mathcal{I} \triangleright (\mathsf{new}\,\tilde{n} : \tilde{\mathsf{T}}_1)\, M \mid \prod_{i \in I} k_i[\![P_i]\!])\;\mathcal{R}\;(\mathcal{I} \triangleright (\mathsf{new}\,\tilde{n} : \tilde{\mathsf{T}}_2)\, N \mid \prod_{i \in I} k_i[\![P_i]\!])$$

if and only if there exists some $\mathcal{I}'_\Gamma$, $(\tilde{\mathsf{T}})$ and $\mathcal{T}'$ such that

$$\begin{array}{l}
\mathcal{I}'_\Gamma <: \mathcal{I}_\Gamma \\
(\tilde{\mathsf{T}}_1) <: (\tilde{\mathsf{T}}) \qquad \text{and} \qquad (\tilde{\mathsf{T}}_2) <: (\tilde{\mathsf{T}}) \\
\mathcal{T}' \subseteq \tilde{n} \\
\mathcal{I}'_\Gamma \vdash k_i[\![P_i]\!] \text{ and } k_i \in \mathcal{I}_{\mathcal{T}} + \mathcal{T}' \text{ for each } i \in I \\
(\mathcal{I}'_\Gamma, \mathcal{I}_{\mathcal{T}} + \mathcal{T}'), \{\tilde{n} : \mathsf{T}\} \models M \approx_{bis} N
\end{array}$$

We aim to show that $\mathcal{R}$ is a bisimulation from which the result follows immediately. For the purposes of this exposition we will assume that $\tilde{n}$ is empty and that the indexing set I is a singleton. We take any

$$(\mathcal{I} \triangleright M \mid k[\![P]\!])\;\mathcal{R}\;(\mathcal{I} \triangleright N \mid k[\![P]\!])$$

so we have some $\mathcal{I}'_\Gamma$ such that

$$(\mathcal{I}'_\Gamma, \mathcal{I}_{\mathcal{T}}) \models M \approx_{bis} N \tag{24}$$

with $\mathcal{I}'_\Gamma \vdash k[\![P]\!]$ and $k \in \mathcal{I}_{\mathcal{T}}$. We suppose that $(\mathcal{I} \triangleright M \mid k[\![P]\!]) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')$ and now must show that there is a corresponding matching move from $(\mathcal{I} \triangleright N \mid k[\![P]\!])$. In cases in which α is not τ this is easily done by appealing to (24). For $\alpha = \tau$ we know that $\mathcal{I}' = \mathcal{I}$ also. By an analogue of the Decomposition Lemma of [8] we can obtain five possibilities:

1. $(\mathcal{I} \triangleright M) \xrightarrow{\tau} (\mathcal{I} \triangleright M'')$ such that $M' \equiv M'' \mid k[\![P]\!]$
2. $k[\![P]\!] \longrightarrow M''$ such that $M' \equiv M \mid M''$
3. for first order T, $(\mathcal{I} \triangleright M) \xrightarrow{(\tilde{m})k.a.\tilde{v}!} (\mathcal{I}'' \triangleright M'')$ with
   - $k[\![P]\!] \equiv k[\![a?(\tilde{x} : \mathsf{T})\, Q]\!]$
   - $M' \equiv (\mathsf{new}\,\tilde{m} : \mathsf{U})\, M'' \mid k[\![Q[\{\!|\tilde{v}/\tilde{x}|\!\}]]\!]$

4. for other T, $(\mathcal{I} \triangleright M) \xrightarrow{(\tilde{m})k.a.V!} (\mathcal{I}'' \triangleright M'')$ with
   - $k[\![P]\!] \equiv k[\![a?(\tilde{x} : \mathsf{T})\, Q]\!]$
   - $V = (\tilde{v}, \lambda\, \tilde{x} : \mathsf{T}.\, Q)$
   - $(\mathsf{new}\,\tilde{m} : \mathsf{U})\, M'' \longrightarrow M'$ derived from (R-BETA)
5. $(\mathcal{I} \triangleright M) \xrightarrow{(\tilde{n}:\mathsf{T})k.a.V?} (\mathcal{I}' \triangleright M'')$ with
   - $k[\![P]\!] \equiv k[\![a!\langle V\rangle]\!]$
   - $M' \equiv M'' \mid k[\![\mathsf{stop}]\!]$

For each case we show that these conditions lead to the desired matching transition. We deal with each of them in turn.

- For (1) we appeal directly to (24).
- More interesting is case (2), particularly when the reduction is generated by use of the rule (R-L.CREATE) or (R-MOVE). We examine each of these: suppose $k[\![P]\!] \longrightarrow M''$ is derived from a use of (R-L.CREATE) so that
$$P = (\mathsf{newloc}\, l : \mathsf{L})\ \mathsf{with}\, C\ \ \mathsf{in}\ \ Q$$
$$M'' \equiv (\mathsf{new}\, l : \mathsf{L})\, l[\![C]\!] \mid k[\![Q]\!]$$
We know by (24) that $(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \models M \approx_{bis} N$ and hence, by weakening to introduce a new testable location, we have
$$(\mathcal{I}'_\Gamma, l : \mathsf{loc}, \mathcal{I}_\mathcal{T} + l) \models M \approx_{bis} N$$
and by further weakening we obtain,
$$(\mathcal{I}'_\Gamma, \{l : \mathsf{L}\}, \mathcal{I}_\mathcal{T} + l) \models M \approx_{bis} N$$
Call the knowledge structure above, $\mathcal{I}''$. We know, by construction of $\mathcal{R}$, that $\mathcal{I}'_\Gamma \vdash k[\![P]\!]$ with $k \in \mathcal{I}_\mathcal{T}$, and therefore, according to the type rules (TY-NEWLOC), (TY-SUBPROC) and (TY-PROC), we must have
$$\mathcal{I}'' \vdash l[\![C]\!] \qquad \text{and} \qquad \mathcal{I}'' \vdash k[\![Q]\!]$$
with $l, k \in \mathcal{I}''_\mathcal{T}$ also. Therefore, by definition of $\mathcal{R}$ again, we see that
$$\mathcal{I} \models (\mathsf{new}\, l : \mathsf{L})(M \mid l[\![C]\!] \mid k[\![Q]\!])\ \mathcal{R}\ (\mathsf{new}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![Q]\!]) \tag{25}$$
We know that $(\mathcal{I} \triangleright N \mid k[\![P]\!]) \xrightarrow{\tau} (\mathcal{I} \triangleright (\mathsf{new}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![Q]\!]))$ and that $M' \equiv M \mid M'' \equiv (\mathsf{new}\, l : \mathsf{L})(M \mid l[\![C]\!] \mid k[\![Q]\!])$, so by (25), we have
$$\mathcal{I} \models M'\ \mathcal{R}\ (\mathsf{new}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![Q]\!])$$
and our matching transition as required.
Alternatively, suppose that $k[\![P]\!] \longrightarrow M''$ is derived from an instance of (R-MOVE). We then have
$$P = \mathsf{goto}\, p.lF \qquad \text{and} \qquad M'' \equiv l[\![p!\langle F\rangle]\!]$$
for some p, l, F. It is important to note here that the location l may not be contained in $\mathcal{I}_\mathcal{T}$ and this prevents us from immediately using the definition of relation $\mathcal{R}$ to claim that
$$\mathcal{I} \models M \mid l[\![p!\langle F\rangle]\!]\ \mathcal{R}\ N \mid l[\![p!\langle F\rangle]\!]$$
However, we do know that $\mathcal{I}'_\Gamma \vdash k[\![P]\!]$ so
$$(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright M \xrightarrow{\mathsf{go}_p l.F} (\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright M \mid l[\![p!\langle F\rangle]\!]$$
is a valid transition. The hypothesis (24) tells us that there is a matching transition
$$(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright N \xRightarrow{\mathsf{go}_p l.F} (\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright N''$$
such that $(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \models M \mid l[\![p!\langle F\rangle]\!] \approx_{bis} N''$. This tells us that there is some N' such that
$$N \longrightarrow^* N' \qquad \text{and} \qquad N' \mid l[\![p!\langle F\rangle]\!] \longrightarrow^* N''$$
Therefore, it is clear that $(\mathcal{I} \triangleright N \mid k[\![P]\!]) \Longrightarrow (\mathcal{I} \triangleright N'')$ with $\mathcal{I} \models M \mid l[\![p!F]\!] \approx_{bis} N''$ as required.

- Cases (3) and (4) are similar in nature so we only show the reasoning for the latter. We have, in this instance, that
$$(\mathcal{I} \triangleright M) \xrightarrow{(\tilde{m})k.a.(\tilde{m}',G)!} (\mathcal{I}'' \triangleright M'')$$
where
$$\begin{array}{l} G = \lambda\, \tilde{x} : \mathsf{T}.\, Q \\ P = a?(\tilde{x} : \mathsf{T})\, Q \\ M'' \longrightarrow M''' \text{ (from (R-BETA) such that } M' \equiv (\mathsf{new}\, \tilde{m} : \mathsf{U}')\, M''' \\ \tilde{m} \subseteq \tilde{m}' \end{array}$$
It is easy to check (*cf.* Lemma 4.8 of [8]) that
$$(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright M \xrightarrow{(\tilde{m})k.a.(\tilde{m}',G)!} (\mathcal{I}'_\Gamma, \{\tilde{m}' : \mathsf{U}\} \triangleright M''), \mathcal{I}_\mathcal{T}) \triangleright M''$$
where $\mathsf{U}' <: \mathsf{U}$. Call the target knowledge structure $\mathcal{I}'''$. This tells us, by (24) that there exists a matching transition
$$(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \triangleright N \xrightarrow{(\tilde{m})k.a.(\tilde{m}',G)!} (\mathcal{I}''' \triangleright N'')$$
with $\mathcal{I}''' \models M'' \approx_{bis} N''$. Note that $M'' \longrightarrow M'''$ (derived from (R-BETA)) guarantees, by confluence properties of beta-reduction, that $\mathcal{I}''' \models M''' \approx_{bis} N''$ and we can also assume, without loss of generality that N'' is stable with respect to $\beta-$reductions. By analysing the above transition we see that there exists some N''', $\tilde{n} : \mathsf{T}'$ and V such that
$$N \longrightarrow^* (\mathsf{new}\, \tilde{m} : \mathsf{U}'')\, (\mathsf{new}\, \tilde{n} : \mathsf{T}')(N''' \mid k[\![a!\langle V\rangle]\!])$$
with
$$(\mathsf{new}\, \tilde{n} : \mathsf{T}')(N''' \mid k[\![\lambda\, \tilde{x} : \mathsf{T}.\, Q(V)]\!]) \longrightarrow^* N'' \qquad \text{and} \qquad \mathsf{U}'' <: \mathsf{U}$$
Therefore we have
$$\begin{array}{rll} N \mid k[\![P]\!] & \longrightarrow^* & (\mathsf{new}\, \tilde{m} : \mathsf{U}'')\, (\mathsf{new}\, \tilde{n} : \mathsf{T}')(N''' \mid k[\![a!\langle V\rangle]\!] \mid k[\![a?(\tilde{x} : \mathsf{T})\, Q]\!]) \\ & \longrightarrow^* & (\mathsf{new}\, \tilde{m} : \mathsf{U}'')\, (\mathsf{new}\, \tilde{n} : \mathsf{T}')(N''' \mid k[\![Q\{\!|^V/_{\tilde{x}}|\!\}]\!]) \\ & \longrightarrow^* & (\mathsf{new}\, \tilde{m} : \mathsf{U}'')\, N'' \\ & \equiv & N' \end{array}$$
Given that $M' \equiv (\mathsf{new}\, \tilde{m} : \mathsf{U}')\, M'''$, we have enough to conclude that $\mathcal{I} \models M' \mathcal{R} N'$ as required.
- Finally, in case (5) we follow a similar argument to that in [8] with only a slight modification to account for the asynchronous nature of SAFEDPI. ■

□

6.3. Relating bisimulation and contextual barbed congruence

This section is devoted to showing that these equivalences, viewed as knowledge-indexed relations coincide.

Proposition 6.11 (Soundness of $\approx_{bis}$ for $\approx_{cxt}$).
$$\mathcal{I} \models M \approx_{bis} N \text{ implies } \mathcal{I} \models M \approx_{cxt} N.$$

Proof. It is evident that $\approx_{bis}$ forms a symmetric, reduction closed and barb preserving knowledge-indexed relation. Therefore, because of Proposition 6.10 $\approx_{bis}$ satisfies all the defining properties of $\approx_{cxt}$. Since $\approx_{cxt}$ is the largest such relation the result follows. □

The force of this proposition is that any distinctions made between systems by the contextual congruence can also be made by the labelled transition system. This means that we have provided enough labels of sufficient distinguishing power. We must also check that we have not provided too much distinguishing power in the labelled transition system. This is done by relating each action defined in the labelled transition system to an actual well-typed SAFEDPI context.

Proposition 6.12 (Definability (*cf. Prop 4.4 of [9]*)). For each label α and each knowledge structure $\mathcal{I}$ there exists a system $\mathcal{C}_\alpha^\mathcal{I}$ which uses the fresh barb name δ, port name δ_{in} and location k_0 and tests for α, in the sense that

- if $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')$ then $\mathcal{I}, \{k_0 : K_0\} \vdash \mathcal{C}^{\mathcal{I}}_{\alpha}$ and moreover, $\mathcal{C}^{\mathcal{I}}_{\alpha} \mid M \longrightarrow^* (\mathsf{new}\, \tilde{m} : \tilde{\mathsf{E}})(k_0[\![\delta_{\mathsf{in}}!\langle\delta!\langle\rangle\rangle]\!] \mid M'')$ with $M'' \equiv M'$
- if $\mathcal{C}^{\mathcal{I}}_{\alpha} \mid M \longrightarrow^* (\mathsf{new}\, \tilde{m} : \tilde{\mathsf{E}})(k_0[\![\delta_{\mathsf{in}}!\langle\delta!\langle\rangle\rangle]\!] \mid M'')$ and $\mathcal{I}, \{k_0 : K_0\} \vdash \mathcal{C}^{\mathcal{I}}_{\alpha}$ where $\tilde{m} = \mathsf{obj}_!(\alpha)$ then $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')$ with $M'' \equiv M'$.

where

$$K_0 = \mathsf{loc}[\delta_{\mathsf{in}} : \mathsf{rw}\langle\mathsf{thunk}\rangle, \delta : \mathsf{rw}\langle\mathbf{unit}\rangle, \delta_{\mathsf{fail}} : \mathsf{rw}\langle\mathbf{unit}\rangle, \delta_{\mathsf{succ}} : \mathsf{rw}\langle\mathbf{unit}\rangle]$$

(the barbs δ_{fail} and δ_{succ} are to be used later).

Proof. These systems are, for the most part, straightforward, and readers familiar with the work in [8, 9] will have little trouble reconstructing them.

As an example we show the systems for $k.a.(\tilde{v}, G)!$ and $\mathsf{go}_p l.V$ actions: we define

$$\mathcal{C}^{\mathcal{I}}_{k.a.(\tilde{v},G)!} \stackrel{def}{=} k[\![a?(\tilde{x}, y)\, \mathsf{if}\ \tilde{x} = \tilde{v}\ \mathsf{then}\ G(\tilde{x}, y) \mid \mathsf{goto}_{\delta_{\mathsf{in}}} k_0.\delta!\langle\rangle\ \mathsf{else}\ \mathsf{stop}]\!]$$

and

$$\mathcal{C}^{\mathcal{I}}_{\mathsf{go}_p l.V} \stackrel{def}{=} k_0[\![\delta_{\mathsf{in}}!\langle\delta!\langle\rangle\rangle \mid \mathsf{goto}_p\, l.V]\!]$$

The interested reader is invited to check that, for any configuration such that $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I} \triangleright M')$ for one of these actions then it is the case that $\mathcal{I}_\Gamma, \{k_0 : K_0\} \vdash \mathcal{C}_\alpha$ and moreover $\mathcal{C}_\alpha \mid M \longrightarrow^* k_0[\![\delta_{\mathsf{in}}!\langle\delta!\langle\rangle\rangle]\!] \mid M''$ where M'' is structurally equivalent to M' up to collection of terminated garbage threads $l[\![\mathsf{stop}]\!]$. $\square$

By providing such testing systems for each action in the lts provided above we are able to establish our second main result

Theorem 6.13 (Full abstraction of $\approx_{bis}$ for $\approx_{cxt}$).

$$\mathcal{I} \models M \approx_{cxt} N \text{ if and only if } \mathcal{I} \models M \approx_{bis} N.$$

Proof. (Sketch) One direction is given by Proposition 6.11. The converse is shown by building a bisimulation from all pairs of configurations such that $\mathcal{I} \models M \approx_{cxt} N$. Specifically, let $\mathcal{R}$ be a relation over configurations defined by

$$(\mathcal{I} \models M)\ \mathcal{R}\ (\mathcal{I} \models N)$$

if $\mathcal{I} \models M \approx_{cxt} N$. We outline the proof that $\mathcal{R}$ defines a bisimulation, from which the result follows.

To this end suppose $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')$, where $\mathcal{I} \models M\ \mathcal{R}\ N$. We must find a matching move $(\mathcal{I} \triangleright N) \stackrel{\alpha}{\Longrightarrow} (\mathcal{I}' \triangleright N')$, such that $\mathcal{I}' \models M'\ \mathcal{R}\ N'$. For the purposes of this sketch we assume for simplicity that $\mathcal{I} = \mathcal{I}'$. By Definability, Proposition 6.12. We know that there exists a system $\mathcal{C}^{\mathcal{I}}_{\alpha}$, typeable from $\mathcal{I}_\Gamma, \{k_0 : K_0\}$, which satisfies the conditions of contextuality for knowledge-indexed relations and moreover, induces an interaction when plugged with M. In other words,

$$\mathcal{C}^{\mathcal{I}}_{\alpha} \mid M \longrightarrow^* k_0[\![\delta_{\mathsf{in}}!\langle\delta!\langle\rangle\rangle]\!] \mid M'' \tag{26}$$

for some δ, δ_{in} at k_0 and M'' equivalent to M' up to structure and garbage collection. We make use of this property of $\mathcal{C}^{\mathcal{I}}_{\alpha}$ as follows: first for the barb names, δ_{fail} and δ_{succ} in K_0 let

$$\mathsf{Flip} \stackrel{def}{=} k_0[\![\delta_{\mathsf{fail}}!\langle\rangle \mid \delta?().\delta_{\mathsf{fail}}?().\delta_{\mathsf{succ}}!\langle\rangle]\!]$$

and let

$$\mathcal{D}^{\mathcal{I}}_{\alpha} \stackrel{def}{=} (k_0[\![\delta_{\mathsf{in}}?(X : \mathsf{thunk})X()]\!] \mid \mathsf{Flip} \mid \mathcal{C}^{\mathcal{I}}_{\alpha} \mid -)$$

It is easy to check that $\mathcal{I}_\Gamma, \{k_0 : K_0\} \vdash \mathcal{D}^{\mathcal{I}}_{\alpha}$ whenever $\mathcal{I}_\Gamma, \{k_0 : K_0\} \vdash \mathcal{C}^{\mathcal{I}}_{\alpha}$. We should note that the reductions (26) above extend so that (up to structure and garbage collection)

$$\mathcal{D}^{\mathcal{I}}_{\alpha}[M] \longrightarrow^* k_0[\![\delta_{\mathsf{succ}}!\langle\rangle]\!] \mid M''$$

The hypothesis $\mathcal{I} \models M \approx_{cxt} N$, the fact that $\mathcal{I}_\Gamma, \{k_0 : K_0\} \vdash \mathcal{C}^{\mathcal{I}}_{\alpha}$ and weakening, contextuality and barb preserving properties of $\approx_{cxt}$ together allow us to use $(\mathcal{I}_\Gamma, \{k_0 : K_0\}, \mathcal{I}_\mathcal{T} + k_0) \models \mathcal{D}^{\mathcal{I}}_{\alpha}[M] \approx_{cxt} \mathcal{D}^{\mathcal{I}}_{\alpha}[N]$ to find a matching transition

$$\mathcal{D}^{\mathcal{I}}_{\alpha}[N] \longrightarrow^* k_0[\![\delta_{\mathsf{succ}}!()]\!] \mid N''$$

with

$$(\mathcal{I}_\Gamma, \{k_0 : K_0\}, \mathcal{I}_\mathcal{T} + k_0) \models k[\![\delta!\langle\rangle]\!] \mid M'' \approx_{cxt} k[\![\delta!\langle\rangle]\!] \mid N''.$$

Note that we can guarantee this form by the absence of the δ_{fail} barb in $k_0[\![\delta_{\mathsf{succ}}!\langle\rangle]\!] \mid M''$ and the fact that, by symmetry, absence of barbs must also be preserved. The systems $\mathcal{C}^{\mathcal{I}}_\alpha$ are also built in such a way as to guarantee that whenever $\mathcal{D}^{\mathcal{I}}_\alpha[N] \longrightarrow^* k_0[\![\delta_{\mathsf{succ}}!\langle\rangle]\!] \mid N''$ then we must also have $\mathcal{I} \triangleright N \stackrel{\alpha}{\Longrightarrow} \mathcal{I} \triangleright N'$ where, again, N'' is equivalent to N' up to structural equivalence and garbage collection. It is easy to show directly that

$$(\mathcal{I}_\Gamma, \{k_0 : K_0\}, \mathcal{I}_\mathcal{T} + k_0) \models k_0[\![\delta_{\mathsf{succ}}!\langle\rangle]\!] \mid M'' \approx_{cxt} k_0[\![\delta_{\mathsf{succ}}!\langle\rangle]\!] \mid N''$$

implies $\mathcal{I} \models M' \approx_{cxt} N'$ which is enough to conclude with $\mathcal{I} \models M' \mathcal{R} N'$. A symmetric argument establishes that $\mathcal{R}$ is a bisimulation.

The case in which $(\mathcal{I} \triangleright M) \xrightarrow{\alpha} (\mathcal{I}' \triangleright M')$ for $\mathcal{I}'$ not equal to $\mathcal{I}$ is slightly more complicated and is dealt with using an *Extrusion Lemma* similar to that found in [6, 9, 8]. □

This provides an alternative characterisation of reduction barbed congruence which models the nature of knowledge acquisition possible by testing with highly constrained mobile code in an explicit way.

7. Conclusion

We have developed a sophisticated type system for controlling the behaviour of mobile code in distributed systems, and demonstrated that, at least in principle, coinductive proof principles can still be applied to investigate their behaviour.

The use of types in this manner could be considered as a particular case of the general approach of *proof-carrying code*, [18] and *typed assembly language (TAL)* [17]. Here hosts would publish their safety policies in terms of a type or logical proposition and code wishing to enter would have to arrive with a proof, which a typechecker or proofchecker can use to verify that it satisfies the published policy. Indeed we intend to use the types of the current paper in this manner, by extending the work in [20]. The work of [18] and [17] has inspired much further research into the use of type systems in higher-level languages for resource access and usage monitoring, [23], [12], for example. However the emphasis in these papers is on dynamics and counting of resource usage rather than using sophisticated types to specify fine-grained access control.

There has been much work on modelling mobility and locations using particular process calculi. Perhaps the calculus closest to SAFEDPI is the Seal Calculus, [5]. Seals are hierarchically organised computational sites in which inter-seal communication, which is channel-based, is only allowed among siblings or between parents and siblings. Seals may also be communicated, rather like the communication of higher-order processes along ports in SAFEDPI; indeed in some sense it is more general as the seal being transmitted may be computationally active. However the communication of seals is more complicated, as it involves agreement between three participants, the sender, the receiver, and the seal being transmitted. Seals are also typed using *interfaces*, similar to our fine-grained process types, π. But these only record the *input* capabilities a seal offers to its parents, and in order to preserve interfaces under reduction the transmission of input channel capabilities is forbidden in the language. This is a severe restriction, at least in general distributed computing, if not in the more focused application area of seals. For example the generation of new servers requires the the transmission of input capabilities. We believe that our dependent and existential types can also be applied to the Seal Calculus, to obtain a more general notion of interface, which will still be preserved by reduction.

The M-calculus, [22], a higher-order extension of the distributed join calculus, is also closely related, at least conceptually, to SAFEDPI. Here, not only are locations hierarchically organised, but are *programmable*, in the sense that entry and exit policies for each location can be explicitly programmed. In addition it has an interesting operator, called *passivation*, which can freeze the contents of a site into a value. However their type system is not related to one we have developed for SAFEDPI; the latter addresses access control issues for migrating code whereas the former is concerned with unicity of locations; in a higher-order language with a passivation operator it is important to ensure that each locality has a unique name. Thus the type system for the M-Calculus draws on that presented in [25], where unicity of the location of channel names was addressed, rather than that of [26], which developed fine-grained access control types for processes.

Type systems have also been used to explicitly control mobility in distributed calculi, most notably in variants of the Ambient calculus of Cardelli and Gordon [3]. In particular, [2], [16] use subtyping to control movement of mobile processes in a hierarchically distributed system by introducing explicit types to express

(E-EMPTY)
$$\frac{}{\vdash \mathbf{env}}$$

(E-GRES)
$$\frac{\Gamma \vdash \mathsf{C} : \mathbf{ty}}{\Gamma,\ u : \mathsf{rc}\langle \mathsf{C} \rangle \vdash \mathbf{env}} \quad \begin{array}{l} u \notin \Gamma \\ \mathsf{C} \text{ closed} \end{array}$$

(TY-LOOKUP)
$$\frac{\Gamma, u : \mathsf{T}, \Gamma' \vdash \mathbf{env}}{\Gamma, u : \mathsf{T}, \Gamma' \vdash_{lookup} u : \mathsf{T}}$$

(E-NEWLCHAN)
$$\frac{\Gamma \vdash_{lookup} w : \mathsf{loc} \qquad \Gamma \vdash \mathsf{C} : \mathbf{ty}}{\Gamma,\ u : \mathsf{C}@w \vdash \mathbf{env}} \quad \begin{array}{l} u \notin \Gamma \\ \mathsf{C} \text{ closed} \end{array}$$

(E-EDEP)
$$\frac{\Gamma, \{x_1 : \mathsf{E}_1\}, \ldots, \{x_n : \mathsf{E}_n\} \vdash \mathsf{T} : \mathbf{ty}}{\Gamma, y : \langle \mathsf{T} \text{ with } \widetilde{x} : \widetilde{\mathsf{E}} \rangle \vdash \mathbf{env}} \quad \begin{array}{l} x_i, y \notin \Gamma \\ y \neq x_i \end{array}$$

(E-SCRIPT)
$$\frac{\Gamma \vdash \mathsf{S} : \mathbf{ty}}{\Gamma, x : \mathsf{S} \vdash \mathbf{env}} \quad x \notin \Gamma$$

(E-LOC)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma,\ u : \mathsf{loc} \vdash \mathbf{env}} \quad u \notin \Gamma$$

(E-BASE)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma,\ u : \mathbf{base} \vdash \mathbf{env}} \quad u \notin \Gamma$$

(TY-ELOOKUP)
$$\frac{\Gamma, \langle \tilde{x} : \tilde{\mathsf{E}}, y : \mathsf{T} \rangle, \Gamma' \vdash \mathbf{env}}{\Gamma, \langle \widetilde{x} : \widetilde{\mathsf{E}}, y : \mathsf{T} \rangle, \Gamma' \vdash_{lookup} y : \mathsf{T}}$$

(E-LCHAN)
$$\frac{\Gamma \vdash_{lookup} w : \mathsf{loc} \qquad \Gamma \vdash_{lookup} u : \mathsf{rc}\langle \mathsf{D} \rangle \qquad \Gamma \vdash \mathsf{D} <: \mathsf{C}}{\Gamma,\ u : \mathsf{C}@w \vdash \mathbf{env}} \quad \mathsf{C} \text{ closed}$$

Fig. 10. Well-defined Environments

permission to migrate. A similar technique was used for DPI in [10], [8]. In contrast, here we control mobility only indirectly through types. Code is always permitted to migrate provided it has access to a suitable port at the target location. But by restricting the use of channels in the types this consequently restricts migration. Indeed, we decouple permission to migrate from the location name itself, affording more flexibility in the control of migration.

The coinductive characterisation presented here makes use of *higher-order actions* in the sense that, to interact with a system willing to send a script V, the environment must supply a receiving script G to which V will be applied. A similar approach is used in the characterisation theorems for various forms of ambients in [7] and [15]. Higher-order actions are also used in the bisimulation equivalence presented in [4] for the Seal calculus. However, there the three way nature of higher-order communication leads to a proliferation of such actions, some of which can not be simulated by seal contexts; see Section 4.4 of [5] for examples. As a result the bisimulation equivalence is more discriminating than the natural contextual equivalence for seals.

Such higher-order bisimulations do not directly result in automatic verification methods for distributed systems. But they do serve to focus on the essential features of systems which determine their behaviour; for example our results for SAFEDPI have demonstrated the importance of the *goto* moves $\mathsf{go}_p k.V$. Moreover they serve as a starting point for more in-depth analyses of the behaviour of SAFEDPI systems, and more particularly of interesting sub-languages. For example is it possible to use the technique of [13] to find a fully-abstract bisimulation equivalence which only uses first-order labels? There the receiving contexts for higher-order values are replaced by symbolic representatives. Although not directly applicable due to the extra complication of distribution and mobility control, it would be of great interest to pursue those ideas in the current setting.

A. Auxiliary Definitions and Results

Types and Type Environments: The judgements for well-defined environments, $\Gamma \vdash \mathbf{env}$, and subtyping, $\Gamma \vdash \mathsf{T} <: \mathsf{U}$, are defined simultaneously, using the rules in Figure 10 and Figure 11. The former are a mild extension of the corresponding rules in Figure 6 of [8] to accommodate script and dependent types and rely on a predicate $\Gamma \vdash_{lookup} u : \mathsf{T}$, which simply looks up the type associated with u in Γ. The latter is an extension of the well-known subtyping rules of types in the PICALCULUS, [21], and DPI, [10, 8]; the rules for process types are similar to those used in [26]. The judgements also check that the identifiers used in T, U are actually declared appropriately in Γ.

Proposition A.1 (Sanity Checks).

(SUB-BASE)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash \mathbf{base} <: \mathbf{base}}$$

(SUB-TOP)
$$\frac{\Gamma \vdash \mathsf{T} <: \mathsf{T}}{\Gamma \vdash \mathsf{T} <: \top}$$

(SUB-PROCTOP)
$$\frac{\Gamma \vdash \pi <: \pi}{\Gamma \vdash \pi <: \mathbf{proc}}$$

(SUB-CHAN)
$$\frac{\Gamma \vdash \mathsf{T}_r <: \mathsf{U}_r, \mathsf{U}_w <: \mathsf{T}_w, \quad \mathsf{T}_w <: \mathsf{T}_r}{\Gamma \vdash \mathsf{w}\langle \mathsf{T}_w \rangle <: \mathsf{w}\langle \mathsf{U}_w \rangle, \quad \Gamma \vdash \mathsf{r}\langle \mathsf{T}_r \rangle <: \mathsf{r}\langle \mathsf{U}_r \rangle \quad \Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{rw}\langle \mathsf{U}_r, \mathsf{U}_w \rangle}$$

$$\frac{\Gamma \vdash \mathsf{T}_r <: \mathsf{U}_r, \mathsf{U}_w <: \mathsf{T}_w \quad \mathsf{T}_w <: \mathsf{T}_r}{\Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{w}\langle \mathsf{U}_w \rangle \quad \Gamma \vdash \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle <: \mathsf{r}\langle \mathsf{U}_r \rangle}$$

(SUB-LOC)
$$\frac{\Gamma \vdash_{lookup} u_i : \mathsf{rc}\langle \mathsf{D}_i \rangle \quad \Gamma \vdash \mathsf{D}_i <: \mathsf{C}_i,\ \mathsf{D}_j <: \mathsf{C}'_j, \quad \Gamma \vdash \mathsf{C}_i <: \mathsf{C}'_i,}{\Gamma \vdash \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_m : \mathsf{C}_m] <: \mathsf{loc}[u_1 : \mathsf{C}'_1, \ldots, u_n : \mathsf{C}'_n]} \quad 0 \leq n \leq m$$

(SUB-HOM)
$$\frac{\Gamma \vdash \mathsf{C} <: \mathsf{C}' \quad \Gamma \vdash_{lookup} w : \mathsf{loc}}{\Gamma \vdash \mathsf{C}@w <: \mathsf{C}'@w \quad \Gamma \vdash \mathsf{rc}\langle \mathsf{C} \rangle <: \mathsf{rc}\langle \mathsf{C}' \rangle}$$

(SUB-SCRIPT)
$$\frac{\Gamma, \{x_1 : (\mathsf{T}_1)@\mathsf{here}\}, \ldots, \{x_n : (\mathsf{T}_n)@\mathsf{here}\} \vdash \pi <: \pi'}{\Gamma \vdash \mathsf{Fdep}(\widetilde{x} : \widetilde{\mathsf{T}} \to \pi) <: \mathsf{Fdep}(\widetilde{x} : \widetilde{\mathsf{T}} \to \pi')}$$

(SUB-PROC)
$$\frac{\Gamma \vdash u_i : \mathsf{C}_i@w_i,\ u_j : \mathsf{C}'_j@w_j \quad \Gamma \vdash \mathsf{C}'_i@w'_i <: \mathsf{C}_i@w_i}{\Gamma \vdash \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_m : \mathsf{C}_m@w_m] <: \mathsf{pr}[u_1 : \mathsf{C}'_1@w_1, \ldots, u_n : \mathsf{C}'_n@w_n]} \quad 0 \leq m \leq n$$

(SUB-TUDEP)
$$\frac{\Gamma, \{x_1 : \mathsf{E}_1\}, \ldots, \{x_n : \mathsf{E}_n\} \vdash \mathsf{T} <: \mathsf{T}'}{\Gamma \vdash \mathsf{Tdep}(\widetilde{x} : \widetilde{\mathsf{E}})\, \mathsf{T} <: \mathsf{Tdep}(\widetilde{x} : \widetilde{\mathsf{E}})\, \mathsf{T}'}$$

(SUB-EDEP)
$$\frac{\Gamma, \{x_1 : \mathsf{E}_1\}, \ldots, \{x_n : \mathsf{E}_n\} \vdash \mathsf{T} <: \mathsf{T}'}{\Gamma \vdash \mathsf{Edep}(\widetilde{x} : \widetilde{\mathsf{E}})\, \mathsf{T} <: \mathsf{Edep}(\widetilde{x} : \widetilde{\mathsf{E}})\, \mathsf{T}'}$$

Fig. 11. Subtyping

- $\Gamma \vdash \mathsf{T} <: \mathsf{U}$ implies $\Gamma \vdash \mathbf{env}$
- $\Gamma \vdash \mathsf{T} <: \mathsf{U}$ implies $\Gamma \vdash \mathsf{T} : \mathbf{ty}$ and $\Gamma \vdash \mathsf{U} : \mathbf{ty}$
- $\Gamma \vdash \mathsf{T} <: \mathsf{U}$, $\Gamma \vdash \mathsf{U} <: \mathsf{R}$ implies $\Gamma \vdash \mathsf{T} <: \mathsf{R}$
- $\Gamma, u : \mathsf{T} \vdash \mathbf{env}$ implies $\Gamma \vdash \mathbf{env}$ and $\Gamma \vdash \mathsf{T} : \mathbf{ty}$

Proof. By rule induction. □

Meets and Joins: The partial operators $\sqcap$, $\sqcup$ on type expressions are defined by extending the definitions used in [10, 8] for channel and location types. We take them to be the least reflexive and symmetric operators which satisfy a series of rules for combining together various kinds of type expressions. Those governing channel expressions are, as in [10]:

- $\mathsf{r}\langle \mathsf{T}_1 \rangle \sqcap \mathsf{r}\langle \mathsf{T}_2 \rangle = \mathsf{r}\langle \mathsf{T}_1 \sqcap \mathsf{T}_2 \rangle, \quad \mathsf{r}\langle \mathsf{T}_1 \rangle \sqcup \mathsf{r}\langle \mathsf{T}_2 \rangle = \mathsf{r}\langle \mathsf{T}_1 \sqcup \mathsf{T}_2 \rangle$
- $\mathsf{w}\langle \mathsf{T}_1 \rangle \sqcap \mathsf{w}\langle \mathsf{T}_2 \rangle = \mathsf{w}\langle \mathsf{T}_1 \sqcup \mathsf{T}_2 \rangle, \quad \mathsf{w}\langle \mathsf{T}_1 \rangle \sqcup \mathsf{w}\langle \mathsf{T}_2 \rangle = \mathsf{w}\langle \mathsf{T}_1 \sqcap \mathsf{T}_2 \rangle$
- $\mathsf{r}\langle \mathsf{T}_r \rangle \sqcap \mathsf{w}\langle \mathsf{T}_w \rangle = \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle$
- $\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle \sqcap \mathsf{r}\langle \mathsf{T}'_r \rangle = \mathsf{rw}\langle \mathsf{T}_r \sqcap \mathsf{T}'_r, \mathsf{T}_w \rangle,$
  $\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle \sqcup \mathsf{r}\langle \mathsf{T}'_r \rangle = \mathsf{rw}\langle \mathsf{T}_r \sqcup \mathsf{T}'_r, \mathsf{T}_w \rangle,$

- $\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w\rangle \sqcap \mathsf{w}\langle \mathsf{T}'_w\rangle = \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \sqcup \mathsf{T}'_w\rangle$,
  $\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w\rangle \sqcup \mathsf{r}\langle \mathsf{T}'_w\rangle = \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \sqcap \mathsf{T}'_w\rangle$,

To express the rules for location types we take advantage of the fact that the ordering of their components is immaterial:

- $\mathsf{loc}[u_1 : \mathsf{C}'_1] \sqcap \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] = \mathsf{loc}[u_1 : (\mathsf{C}'_1 \sqcap \mathsf{C}_1), \ldots, u_n : \mathsf{C}_n]$, $\mathsf{loc}[u_1 : \mathsf{C}'_1] \sqcup \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] = \mathsf{loc}[u_1 : (\mathsf{C}'_1 \sqcup \mathsf{C}_1)]$
- if u does not occur in $\{u_1, \ldots, u_n\}$ then
  $\mathsf{loc}[u : \mathsf{C}] \sqcap \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] = \mathsf{loc}[u : \mathsf{C}, u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n]$, $\mathsf{loc}[u : \mathsf{C}] \sqcup \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] = \mathsf{loc}[]$
- $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] \sqcap \mathsf{K} = \mathsf{loc}[u_1 : \mathsf{C}_1] \sqcap (\ldots (\mathsf{loc}[u_n : \mathsf{C}_n] \sqcap \mathsf{K}) \ldots)$, $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n] \sqcup \mathsf{K} = (\mathsf{loc}[u_1 : \mathsf{C}_1] \sqcup \mathsf{K}) \sqcap \ldots \sqcap (\mathsf{loc}[u_n : \mathsf{C}_n] \sqcup \mathsf{K})$

We use a similar approach to defining the operations on process types, where we use GC as an arbitrary type of the form $\mathsf{C}@w$. However the process type constructor is contravariant, whereas the location constructor is covariant.

- $\mathsf{pr}[u_1 : \mathsf{C}'_1@w_1] \sqcap \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_n : \mathsf{GC}_n] = \mathsf{pr}[u_1 : (\mathsf{C}'_1 \sqcup \mathsf{C}_1)@w_1]$,
  $\mathsf{pr}[u_1 : \mathsf{C}'_1@w_1] \sqcup \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_n : \mathsf{GC}_n] =$
  $\mathsf{pr}[u_1 : (\mathsf{C}'_1 \sqcap \mathsf{C}_1)@w_1, \ldots, u_n : \mathsf{GC}_n]$
- if $u@w$ does not occur in $\{u_1@w_1, \ldots, u_n@w_n\}$ then
  $\mathsf{pr}[u : \mathsf{C}@w] \sqcap \mathsf{pr}[u_1 : \mathsf{C}_1@w, \ldots, u_n : \mathsf{C}_n@w_n] = \mathsf{pr}[]$,
  $\mathsf{pr}[u : \mathsf{C}@w] \sqcup \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_n : \mathsf{C}_n@w_n] =$
  $\mathsf{pr}[u : \mathsf{C}@w, u_1 : \mathsf{C}_1@w_1 \ldots, u_n : \mathsf{C}_n@w]$
- $\mathsf{pr}[u_1 : \mathsf{GC}_1, \ldots, u_n : \mathsf{GC}_n] \sqcap \pi =$
  $(\mathsf{pr}[u_1 : \mathsf{GC}_1] \sqcap \pi) \sqcup \ldots \sqcup (\mathsf{pr}[u_n : \mathsf{GC}_n] \sqcap \pi)$,
  $\mathsf{pr}[u_1 : \mathsf{GC}_1, \ldots, u_n : \mathsf{GC}_n] \sqcup \pi = \mathsf{pr}[u_1 : \mathsf{GC}_1] \sqcup (\ldots (u_n : \mathsf{GC}_n \sqcup \pi) \ldots)$
- $\mathbf{proc} \sqcap \pi = \pi$, $\mathbf{proc} \sqcup \pi = \mathbf{proc}$

For the various forms of dependent types, the rules are straightforward:

- $\mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi) \sqcap \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi') = \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to (\pi \sqcap \pi'))$,
  $\mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi) \sqcup \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi') = \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,(\pi \sqcup \pi')$
- $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T} \sqcap \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T}' = \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,(\mathsf{T} \sqcap \mathsf{T}')$,
  $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T} \sqcup \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T}' = \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{T}})\,(\mathsf{T} \sqcup \mathsf{T}')$
- $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T} \sqcap \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T}' = \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,(\mathsf{T} \sqcap \mathsf{T}')$,
  $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T} \sqcup \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,\mathsf{T}' = \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{T}})\,(\mathsf{T} \sqcup \mathsf{T}')$

For the remaining kinds of type expressions we merely extend the definitions homomorphically:

- $\mathsf{rc}\langle \mathsf{C}\rangle \sqcap \mathsf{rc}\langle \mathsf{C}'\rangle = \mathsf{rc}\langle \mathsf{C} \sqcap \mathsf{C}'\rangle$, $\mathsf{rc}\langle \mathsf{C}\rangle \sqcup \mathsf{rc}\langle \mathsf{C}'\rangle = \mathsf{rc}\langle \mathsf{C} \sqcup \mathsf{C}'\rangle$
- $\mathsf{T}@w \sqcap \mathsf{T}'@w = (\mathsf{T} \sqcap \mathsf{T}')@w$

Proposition A.2.

- If there exists some type expression T such that $\Gamma \vdash \mathsf{T} <: \mathsf{T}_1$ and $\Gamma \vdash \mathsf{T} <: \mathsf{T}_2$ then $\mathsf{T}_1 \sqcap \mathsf{T}_2$ is well-defined
- When $\mathsf{T}_1 \sqcap \mathsf{T}_2$ is well-defined, $\Gamma \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 <: \mathsf{T}_i$ and $\Gamma \vdash \mathsf{T} <: \mathsf{T}_1 \sqcap \mathsf{T}_2$, for any type expression T such that $\Gamma \vdash \mathsf{T} <: \mathsf{T}_1$ and $\Gamma \vdash \mathsf{T} <: \mathsf{T}_2$.
- If there exists some type expression T such that $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}$ and $\Gamma \vdash \mathsf{T}_2 <: \mathsf{T}$ then $\mathsf{T}_1 \sqcup \mathsf{T}_2$ is well-defined
- When $\mathsf{T}_1 \sqcup \mathsf{T}_2$ is well-defined, $\Gamma \vdash \mathsf{T}_i <: \mathsf{T}_1 \sqcup \mathsf{T}_2$, and $\Gamma \vdash \mathsf{T}_1 \sqcup \mathsf{T}_2 <: \mathsf{T}$, for any type expression T such that $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}$ and $\Gamma \vdash \mathsf{T}_2 <: \mathsf{T}$.

Proof. The first and third statements are proved by induction on the derivations of $\Gamma \vdash \mathsf{T}_i <: \mathsf{T}$ and $\Gamma \vdash \mathsf{T} <: \mathsf{T}_i$ respectively. The second and fourth are by induction on the construction of $\mathsf{T}_1 \sqcap \mathsf{T}_2$, $\mathsf{T}_1 \sqcup \mathsf{T}_2$ respectively. $\square$

Note that because of the top type $\top$ the premise of the third statement is always true; so $\mathsf{T}_1 \sqcup \mathsf{T}_2$ always exists, although in many cases it will be the uninformative type $\top$.

Substitutions: Free identifiers may occur in type expressions and therefore we need to define $\mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}$ for an arbitrary type expression T; this is then used as part of the definition of substitution into process terms. The definition of $\mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}$ is by induction on the structure of T. The only interesting cases are location and process types, where the definition needs to ensure that the entries remain unique:

- $\mathsf{loc}[u' : \mathsf{C}]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = \mathsf{loc}[u'\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} \;:\; \mathsf{C}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}]$
- $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots u_n : \mathsf{C}_n]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} =$ $(\mathsf{loc}[u_1 : \mathsf{C}_1]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}) \sqcap \ldots \sqcap (\mathsf{loc}[u_n : \mathsf{C}_n]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\})$
- $\mathsf{pr}[u' : \mathsf{C}]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = \mathsf{pr}[u'\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} \;:\; \mathsf{C}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}]$
- $\mathsf{pr}[u_1 : \mathsf{C}_1, \ldots u_n : \mathsf{C}_n]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = (\mathsf{pr}[u_1 : \mathsf{C}_1]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}) \sqcup \ldots \sqcup (\mathsf{pr}[u_n : \mathsf{C}_n]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\})$
- All other cases are defined homomorphically. For example
  - $\mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w\rangle\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = \mathsf{rw}\langle \mathsf{T}_r\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}, \mathsf{T}_w\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}\rangle$
  - $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = \mathsf{Tdep}(\tilde{x} : (\tilde{\mathsf{E}}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}))\,(\mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\})$, where we assume v is different from each x_i

Proposition A.3.
- Suppose $\mathsf{T} \sqcap \mathsf{U}$ is defined. Then so is $\mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} \sqcap \mathsf{U}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}$ and (up to α-equivalence) is the same as $(\mathsf{T} \sqcap \mathsf{U})\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}$
- Similarly for $\mathsf{T} \sqcup \mathsf{U}$.

Proof. By simultaneous induction on the definitions of $\mathsf{T} \sqcap \mathsf{U}$ and $\mathsf{T} \sqcup \mathsf{U}$. □

Substitution of identifiers also commutes with the channel extraction function.

Proposition A.4. For all identifiers u, v,

$$\mathsf{pr}_{\mathsf{ch}}[V : \mathsf{T}]\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} = \mathsf{pr}_{\mathsf{ch}}[V\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\} : \mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{u}\}\!\!\}]$$

Proof. By induction on the definition of $\mathsf{pr}_{\mathsf{ch}}[V : \mathsf{T}]$. The only non-trivial case is when V is an identifier w and T a location type, when the proof depends on the peculiaries of the application of substitutions to location and process types. □

Proposition A.5 (Substitution). Suppose $\Gamma \vdash_w v : \mathsf{T}$ and $x \notin \Gamma$. Then

- $\Gamma, x : (\mathsf{T})@w, \Delta \vdash \mathbf{env}$ implies $\Gamma, \Delta\{\!\!\{{}^{v}\!/\!{}_{x}\}\!\!\} \vdash \mathbf{env}$
- $\Gamma, , x : (\mathsf{T})@w, \Delta \vdash \mathsf{T} <: \mathsf{U}$ implies $\Gamma, \Delta\{\!\!\{{}^{v}\!/\!{}_{x}\}\!\!\} \vdash \mathsf{T}\{\!\!\{{}^{v}\!/\!{}_{x}\}\!\!\} <: \mathsf{U}\{\!\!\{{}^{v}\!/\!{}_{x}\}\!\!\}$

Proof. By simulataneous induction on the derivations. Note that there are only four possibilities for the entry $x : (\mathsf{T})@w$, namely $x : \mathsf{loc}$, $x : \mathsf{rc}\langle \mathsf{D}\rangle$, $x : \mathsf{C}@w$ or $x : \mathsf{S}$. □

The corresponding substitution result for existential values depends on the following property of existential witnesses.

Proposition A.6. Let Γ_e denote $\Gamma, y : \langle \mathsf{T} \text{ with } \tilde{x} : \tilde{\mathsf{T}}\rangle, \Gamma'$. Then

- $\Gamma_e \vdash \mathbf{env}$ implies x_i does not occur in Γ'.
- $\Gamma_e \vdash \mathsf{T} <: \mathsf{U}$ implies x_i does not ocur free in T, U.

Proposition A.7. Suppose $\Gamma \vdash_w \langle \tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}$. Let Γ_e denote $\Gamma, y : \langle (\mathsf{T})@w \text{ with } \tilde{x} : (\tilde{\mathsf{E}})@w\rangle, \Delta$. Then

- $\Gamma_e \vdash \mathbf{env}$ implies $\Gamma, \Delta\{\!\!\{{}^{v}\!/\!{}_{y}\}\!\!\} \vdash \mathbf{env}$
- $\Gamma \vdash \mathsf{U}_1 <: \mathsf{U}_2$ implies $\mathsf{U}_1\{\!\!\{{}^{\tilde{v},v}\!/\!{}_{\tilde{x},y}\}\!\!\} <: \mathsf{U}_2\{\!\!\{{}^{\tilde{v},v}\!/\!{}_{\tilde{x},y}\}\!\!\}$

Proof. By simultaneous induction on the inferences. □

Adding knowledge to environments: Here we extend the meet operator $\sqcap$ to lists of type associations. This is used in Figure 9, in the rules (M-SEND.VAL) and (M-SEND.DEP.SCRIPT), for increasing the knowledge in a type environment. We first define the (partial) operation $\Gamma \sqcap u : \mathsf{E}$ between an arbitrary association list Γ and a singleton:

- If E is a located channel $\mathsf{A}@w$ then $\Gamma \sqcap u : \mathsf{E}$ is $\Gamma, u : \mathsf{E}$.
- Otherwise if u has no association in Γ then $\Gamma \sqcap u : \mathsf{E}$ is also $\Gamma, u : \mathsf{E}$.

- Otherwise $\Gamma \sqcap u : \mathsf{E}$ is obtained by replacing the association of u in Γ, say $u : \mathsf{E}'$, by the new association $u : (\mathsf{E} \sqcap \mathsf{E}')$; in this case the operation is only defined if $(\mathsf{E} \sqcap \mathsf{E}')$ exists.

The general definition of $\Gamma_1 \sqcap \Gamma_2$ then follows by induction on the size of Γ_2:

- $\Gamma_1 \sqcap \epsilon = \Gamma_1$
- $\Gamma_1 \sqcap (\Gamma_2', u : \mathsf{E}) = (\Gamma_1 \sqcap \Gamma_2') \sqcap u : \mathsf{E}$

References

[1] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proc. 13th LICS Conf.* IEEE Computer Society Press, 1998.

[2] L. Cardelli, G. Ghelli, and A. Gordon. Ambient groups and mobility types. In *Proc. IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[3] L. Cardelli and A. Gordon. Mobile ambients. In *Proc. FoSSaCS '98*, LNCS. Springer-Verlag, 1998.

[4] G. Castagna and F. Zappa Nardelli. The Seal calculus revisited: Contextual equivalences and bisimilarity. In *Proceedings of FSTTCS*, Lecture Notes in Computer Science, 2002.

[5] Giuseppe Castagna, Jan Vitek, and Francesco Zappa. The Seal calculus. 2003. Available from ftp://ftp.di.ens.fr/pub/users/castagna/seal.ps.gz.

[6] C. Fournet, G. Gonthier, J-J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proc. CONCUR*, volume 1119 of *Lecture notes in computer science*. Springer-Verlag, 1996.

[7] M. Hennessy and M. Merro. Bisimulation congruences in safe ambients. In *Proc. POPL 02*. ACM Press, 2002.

[8] Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 322:615–669, 2003.

[9] Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. *Mathematical Structures in Computer Science*, 14:651–684, 2004.

[10] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[11] K. Honda and N. Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.

[12] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 331–342, 2002.

[13] Alan Jeffrey and Julian Rathke. Contextual equivalence for higher-order π-calculus revisited. In *Proceedings MFPS XIX, Montreal*, 2003.

[14] Cedric Lhoussaine. Type inference for a distributed pi-calculus. In *ESOP'02*, volume 2618 of *LNCS*, pages 253–269. Springer-Verlag, 2002.

[15] Massimo Merro and Francesco Zappa Nardelli. Bisimulation proof techniques for mobile ambients. In *Proc. 30^{th} International Colloquium on Automata, Languages, and Programming (ICALP 2003), Eindhoven*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[16] Massimo Merro and Vladimiro Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings CONCUR 02*, volume 1644 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[17] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, volume 1473 of *Lecture notes in Computer Science*, pages 25–35. Springer-Verlag, 1998.

[18] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[19] B. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, 2000.

[20] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents (extended abstract). In *Conference Record of POPL '99 The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 93–104, 1999. To appear in the Journal of Automated Reasoning.

[21] Davide Sangiorgi and David Walker. *The π-calculus.* Cambridge University Press, 2001.

[22] A. Schmitt. and J.-B. Stefani. The M-calculus: A higher-order distributed process calculus. In *POPL2003*, January 2003.

[23] David Walker. A type system for expressive security properties. In *the twenty seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston*, pages 254–267, 2000.

[24] Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *Conference Record of POPL '04: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice,Italy, jan 2004.

[25] Nobuko Yoshida and Matthew Hennessy. Subtyping and locality in distributed higher order processes. In *Proc. CONCUR*, volume 1664 of *Lecture notes in computer science*. Springer-Verlag, 1999.

[26] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Information and Computation*, 172:82–120, 2002.