SAFEDPI: a language for controlling mobile code

MATTHEW HENNESSY, JULIAN RATHKE and NOBUKO

threads P , , 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

$$l[\![(\mathsf{n}\ \ \mathsf{wloc}\, k : \mathsf{K})\, \mathsf{with}\, C\ \ \mathsf{in}\ \ \mathsf{xpt}_1!\langle k\rangle \mid \mathsf{xpt}_2!\langle k\rangle]\!]$$

a new location k is created at type K, the code C 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 fing(r) 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

$$\mathsf{xpt}_1 : \mathsf{w}\langle \mathsf{loc}[\mathsf{ping} : \mathsf{w}\langle \mathsf{P}\rangle]\ \ \mathsf{P}\ \ [\mathrm{i}$$

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

the type

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

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

participate in them. As a simple example consider the system

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

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

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

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.

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

2 The language SAFEDPI

YNTAX: 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 $(\mathsf{n}\ \ \mathsf{w}\, e : \mathsf{E})\, -$.

M, N	::=	*Systems*
$l[\![P]\!]$		Located Process
M		

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 $(\mathsf{n\ wc}\, c : \mathsf{C})\ P$ a new local channel named c is declared, while $(\mathsf{n\ wr\ g}\, n : \mathsf{N})\ P$ represents the generation of a new globally registered name n for channels; see [8] for motivation. When a new location is declared, in $(\mathsf{n\ wloc}\, k : \mathsf{K})\,\mathsf{with}\quad \mathsf{in}\ P$, its declaration type K can only involve channel names which have been registered. This construct generates the new location k, sets the code 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 be gained via a port; so here 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, gives 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 our 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 EMANTICS: 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 $O \mid M \,\mathcal{R}\, O \mid N$

(R-COMM)

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

(R-SPLIT)

$$k[\![P \mid \quad]\!] \longrightarrow k[\![P]\!] \mid k[\![\quad]\!]$$

(R-N.CREATE)

(R-MOVE)

$$k[\![(\mathsf{n}\ \ \mathsf{wr}\ \ \mathsf{g}\, n:\mathsf{N})\ \ P]\!] \longrightarrow (\mathsf{n}\ \ \mathsf{w}\, n:\mathsf{N})\, k[\![P]\!]$$

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

(R-L.CREATE)

$$k[\![(\mathsf{n}\ \ \mathsf{wloc}\, l:\mathsf{L})\,\mathsf{with}\, C\ \ \mathsf{in}\ \ P]\!] \longrightarrow (\mathsf{n}\ \ \mathsf{w}\, l:\mathsf{L})(k[\![P]\!] \mid l[\![C]\!])$$

(R-C.CREATE)

(R-UNWIND)

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

$$k[\![(\mathsf{n}\ \ \mathsf{wc}\, c:\mathsf{C})\ \ P]\!] \longrightarrow (\mathsf{n}\ \ \mathsf{w}\, c:\mathsf{C}@k)\, k[\![P]\!]$$

(R-EQ)

(R-BETA)

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

$$k[\![\mathsf{if}\ u = u\ \mathsf{th}\ \ \mathsf{n}\ P\ \ \mathsf{ls}\quad]\!] \longrightarrow k[\![P]\!]$$

(R-NEQ)

$$k[\![\mathsf{if}\ u = \mathsf{th}\ \ \mathsf{n}\ P\ \ \mathsf{ls}\quad]\!] \longrightarrow k[\![$$

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.

CRIPT 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

- $x : \langle \mathsf{T} \text{ with } \tilde{y} : \tilde{\mathsf{E}} \rangle$. 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

- $\{v : \mathsf{C}_@ w\} = v : \mathsf{C}_@ w$
- $\{x : \mathsf{S}\} = x : \mathsf{S}$
- $\{v : \mathsf{loc}[u_1 : \mathsf{C}_1, \ldots u_n : \mathsf{C}_n]\} = v : \mathsf{loc}, u_1 : \mathsf{C}_{1@}v, \ldots u_n : \mathsf{C}_{n@}v$
- $\{(\tilde{y}, x) : \mathsf{Tdep}(\tilde{y} : \tilde{\mathsf{E}})\, \mathsf{T}\} = \{y_1 : \mathsf{E}_1\} \ldots, \{y_n : \mathsf{E}_n\},\ \{x : \mathsf{T}\}$
- $\{x : \mathsf{Edep}(\tilde{y} : \tilde{\mathsf{E}})\, \mathsf{T}\} = x : \langle \mathsf{T} \text{ with } \{y_1 : \mathsf{E}_1\} \ldots, \{y_n : \mathsf{E}_n\} \rangle$ ■

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]. 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 less then 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.

It is worth pointing out that with our type expressions set $\mathbf{Types}_\Gamma$ turns out to be not only a preorder but also a partial order. However this would no longer be the case if we allowed recursive types; nevertheless with this extension our results would still apply. Note also that because of the existence of the top type $\top$, useful in Section 6, joins of types are always guaranteed to exist.

3.3 Type inference

We are now ready to describe the type inference system for ensuring that systems are well-typed.

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

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 it's 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 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 rules (ty-new72.83)Td(

safeDpi:

tantly the type at

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 ite 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.

Server: $s[\![\mathsf{r}\ \mathsf{q}?(\xi : \mathsf{S})\,\mathsf{run}\,\xi \mid *\,\mathsf{n}\ \mathsf{ws}!\langle_{scandal}\rangle]\!]$

Client: $c[\![\mathsf{goto}_{\mathsf{req}}\ s.\mathsf{n}\ \mathsf{ws}?(x)\,\mathsf{goto}_{\mathsf{in}}\ c.\,\mathsf{r}\ \mathsf{port}!\langle x\rangle$

$\mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{r}\ \mathsf{port}?(y\mathrm{goto}$

server, S. By *dethunking* we get the requirement

$$\Gamma \vdash_s \mathsf{n}\ \ \mathsf{ws}?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{r}\ \ \mathsf{port}!\langle x\rangle\ \ : \mathsf{pr}[\mathsf{n}\ \ \mathsf{ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

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

$$\Gamma, x : \mathbf{string} \vdash_s \mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{r}\ \ \mathsf{port}!\langle x\rangle\ \ : \mathsf{pr}[\mathsf{n}\ \ \mathsf{ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c]$$

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

$$\Gamma, x : \mathbf{string} \vdash_c \mathsf{in}!\langle\ \mathsf{r}\ \ \mathsf{port}!\langle x\rangle\rangle\ \ : \mathsf{pr}[\mathsf{n}\ \ \mathsf{ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c] \qquad (9)$$

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 $\mathsf{pr}_{\mathsf{ch}}[\lambda\ ().\ \mathsf{r}\ \ \mathsf{port}!\langle x\rangle : \mathbf{proc}]$ 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

$$\begin{aligned} \mathsf{R}: &\quad \mathsf{th}[\mathsf{r}\ \ \mathsf{port} : \mathsf{w}\langle\mathbf{string}\rangle@c] \\ \mathsf{S}: &\quad \mathsf{th}[\mathsf{n}\ \ \mathsf{ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle@c] \end{aligned}$$

Here the guardian for the client only allows in agents which write to the local port r port; 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

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

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

It might be tempting to define the guardians by

$$\begin{aligned} \mathsf{R}: &\quad \mathsf{th}[\mathsf{r}\ \ \mathsf{port} : \mathsf{w}\langle\mathbf{string}\rangle@c] \\ \mathsf{S}: &\quad \mathsf{th}[\mathsf{n}\ \ \mathsf{ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ \mathsf{in} : \mathsf{w}\langle\mathsf{thunk}\rangle@c] \end{aligned}$$

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

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

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 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

client server from (8) above:

$$
\begin{array}{rl}
\text{Server:} & s[\![\,\mathsf{r}\ \mathsf{q}?(\xi\ \mathsf{with}\ y : \mathsf{S}_d)\,\mathsf{run}\,\xi\ \mid\ *\,\mathsf{n}\ \mathsf{ws}!\langle \mathit{scandal}\rangle\,]\!] \\
\text{Client:} & c[\![(\mathsf{n}\ \mathsf{wc}\,\mathsf{r}\ \mathsf{port}) \\
& \quad \mathsf{goto}_{\mathsf{req}}\, s.\mathsf{n}\ \mathsf{ws}?(x)\,\mathsf{goto}_{\mathsf{in}}\, c.\,\mathsf{r}\ \mathsf{port}!\langle x\rangle\ \mathsf{with}\ c\ \mid \\
& \quad \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi\ \mid \mathsf{r}\ \mathsf{port}?(y)\ldots]\!]
\end{array}
\tag{12}
$$

with the types

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

Here the

which in turn requires the premise

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

the usual process to the server but now accompanies it with the triple $(c, \mathsf{r\ port}, \mathsf{in})$

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

$$\mathsf{S}_d : \quad \mathsf{Tdep}(y : \mathsf{loc},\ z : \mathsf{w}\langle\mathbf{string}\rangle@y,\ x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y)$$
$$\mathsf{th}[\mathsf{n\ \ ws} : \mathsf{r}\langle\mathbf{string}\rangle@s,\ x : \mathsf{w}\langle\mathsf{th}[z : \mathsf{w}\langle\mathbf{string}\rangle@y]\rangle@y]$$

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

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:

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

Here the guardian type S_e is

$$\mathsf{Edep}(y : \mathsf{loc}, z : \mathsf{w}\langle \mathbf{string}\rangle @ y, x : \mathsf{w}\langle \mathsf{th}[z : \mathsf{w}\langle \mathbf{string}\rangle @ y]\rangle @ y)$$

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-SUBTYPING) from Figure 6 we obtain the required judgement (18) above.

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

$$\Gamma, \mathsf{r}\ \mathsf{port} : \mathsf{rw}\langle\mathbf{string}\rangle @ c \vdash_s \mathsf{r}\ \mathsf{q}!\langle \mathsf{n}\ \mathsf{ws}?(x)\, \mathsf{goto}_{\mathsf{in}}\, c.\, \mathsf{r}\ \mathsf{port}!\langle x\rangle\rangle \quad : \mathbf{proc}$$

Here the relevant rule is (judgement:

$$\Gamma,$$

$(F\ \mathsf{foo})$, 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

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

To see this let Γ be as described on 24. Then, using a simple variation on the inference described there, we can infer

$\Gamma, y : \mathsf{r}\langle\mathbf{string}\rangle@s \vdash_s y?(x)\,\mathsf{goto}_{\mathsf{in}}\,c.\,\mathsf{r}\ \ \mathsf{port}!\langle x\rangle\ :\ \mathsf{pr}[y : \mathsf{r}\langle\mathbf{string}\rangle@\mathsf{h}\ \ \mathsf{r}\ , \mathsf{in} :$

which is in turn an elaboration of the example we have just considered:

$$\begin{array}{ll}
\text{Server:} & s[\![\mathsf{r}\ \mathsf{q}?(\xi:\mathsf{S}_{se})\,(\xi\ \mathsf{n}\ \mathsf{ws}) \mid *\mathsf{n}\ \mathsf{ws}!\langle_{scandal}\rangle]\!] \\
\text{Client:} & c[\![(\mathsf{n}\ \mathsf{wc}\,\mathsf{r}\ \mathsf{port}) \\
& \qquad (\mathsf{n}\ \mathsf{wc}\,\mathsf{in}:\mathsf{rw}\langle\mathsf{R}\rangle) \\
& \qquad\qquad \mathsf{goto}_{\mathsf{req}}\ s.F \mid \\
& \qquad\qquad \mathsf{in}?(\xi:\mathsf{R})\ \ \mathbf{string}\rangle.\ y?(x \quad \text{new}
\end{array}$$

the server, and at the same time the server is

F to

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 (UBSTITUTION). *uppose* $\Gamma \vdash_{w_1} v : \mathsf{T}$ *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 three 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

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

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

- 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 $\Gamma \vdash_{w_2}$*

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

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

So applying the previous lemma we obtain

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

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 $\Gamma \equiv \Gamma, u_1 : \mathsf{C}_1@v, \ldots u_n : \mathsf{C}_n@v$ from which the required

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

follows. ■

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* $\Gamma, y : \langle \mathsf{T} \text{ with } \tilde{x} : \tilde{\mathsf{E}} \rangle, \Gamma' \vdash_w J : \mathsf{T}$. *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 $y : \langle \mathsf{T} \text{ with } \tilde{x} : \tilde{\mathsf{E}} \rangle$ 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 ■

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

LEMMA 5.7 (SUBSTITUTION). *Suppose* $\Gamma \vdash_{w_1} \langle \tilde{v}, v \rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}$. *Then* $\Gamma, y : \langle (\mathsf{T})@w_1 \text{ with } \tilde{x} : \tilde{\mathsf{E}} \rangle, \Delta \vdash_{w_2} J : \mathsf{T},\ w_2 : \mathsf{loc}$ *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 depends on the corresponding result for well-typed environments and subtyping judgements, Proposition A.7 in the Appendix. ■

THEOREM 5.8 (SUBJECT REDUCTION).

Suppose $\Gamma \vdash M$. *Then* $M \longrightarrow N$ *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

- $(\tilde{u}, F)$ 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,

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

write␣⟨⟩

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

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 $\mathcal{I} \models N \approx_{cxt} N$)
- 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 practice, [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:

$$\alpha ::= \tau \mid (\tilde{n} : \tilde{E})\mathsf{go}_p k.F$$

(m-receive)
$$\frac{k \in \mathcal{I}_{\mathcal{T}} \quad \mathsf{T} = \bigsqcap \mathcal{I}^w_\Gamma(a,k) \qquad \mathcal{I}^w_\Gamma(a,k) \neq \emptyset \quad \mathcal{I}_\Gamma \vdash_k V : \mathsf{T}}{(\mathcal{I} \rhd k[\![a?(X : \mathsf{U})\ P]\!]) \xrightarrow{k.a.V?} (\mathcal{I} \rhd k[\![P\{^V/_X\}]\!])}$$

(m-deliver)
$$\frac{k \in \mathcal{I}_{\mathcal{T}} \quad \mathsf{T} = \bigsqcap \mathcal{I}^w_\Gamma(a,k) \qquad \mathcal{I}^w_\Gamma(a,k) \neq \emptyset \quad \mathcal{I}_\Gamma \vdash_k V : T}{(\mathcal{I} \rhd M) \xrightarrow{k.a.V?} (\mathcal{I} \rhd M \mid k[\![a!\langle V\rangle]\!])}$$

(m-send.val)
$$\frac{k \in \mathcal{I}_{\mathcal{T}} \qquad \mathsf{T} \text{a first-order type} \quad \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \quad \mathcal{I}_\Gamma, \{\tilde{u} : (\mathsf{T})_@k\} \vdash \mathbf{env}}{(\mathcal{I} \rhd k[\![a!\langle \widetilde{u}\rangle]\!]) \xrightarrow{k.a.\tilde{u}!} (\mathcal{I}, \{\widetilde{u} : (\mathsf{T})_@k\} \rhd k[\![\mathsf{stop}]\!])}$$

(m-send.script)
$$\frac{k \in \mathcal{I}_{\mathcal{T}} \qquad \mathsf{T} \text{ of the form } \mathsf{Edep}(\tilde{x} : \tilde{T})\ S \quad \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \quad \mathcal{I}_\Gamma \vdash_k G : \mathsf{T} \to \mathbf{proc}}{(\mathcal{I} \rhd k[\![a!\langle F\rangle]\!]) \xrightarrow{k.a.G!} (\mathcal{I} \rhd k[\![G\ (F)]\!])}$$

(m-send.dep.script)
$$\frac{k \in \mathcal{I}_{\mathcal{T}} \qquad \mathsf{T} \text{ of the form } \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\ \mathsf{S} \quad \mathsf{T} = \bigsqcap \mathcal{I}^r_\Gamma(a,k) \qquad \mathcal{I}^r_\Gamma(a,k) \neq \emptyset \quad \mathcal{I}_\Gamma, \{\tilde{u} : (\tilde{\mathsf{E}})_@k\} \vdash \mathbf{env} \quad \mathcal{I}_\Gamma \vdash_k G : T \to \mathbf{proc}}{(\mathcal{I} \rhd k[\![a!\langle(\widetilde{u}, F)\rangle]\!]) \xrightarrow{k.a.(\widetilde{u},G)!} (\mathcal{I}, \{\widetilde{u} : (\widetilde{\mathsf{E}})_@k\} \rhd k[\![G\ (\widetilde{u}, F)]\!])}$$

(m-goto)
$$\frac{k \notin \mathcal{I}_{\mathcal{T}} \quad \mathcal{I}_\Gamma \vdash_k p!\langle V\rangle : \mathbf{proc}}{(\mathcal{I} \rhd M) \xrightarrow{\mathsf{go}_p k.V} (\mathcal{I} \rhd M \mid k[\![p!\langle V\rangle]\!])}$$

Figure

Figure

Figure

(M-

The inference rules for the action judgements (23) are given in Figures 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 $(\tilde{n} : \tilde{E})(m, \tilde{m})\alpha'$ whenever α is $(\tilde{n} : \tilde{E})(\tilde{m})\alpha'$. 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.

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 $\mathcal{I} \rhd M$ is a configuration from which $(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd N)$, 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 $(\tilde{m}) \subset (\tilde{u})$*

ASYNCHRONOUS-GOTO: *$(\tilde{n} : \tilde{\mathsf{E}})go_p k.F$, where $(\tilde{n}) \subseteq \mathsf{fn}(F)$.*

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

PROPOSITION 6.8 (WELL-DEFINEDNESS). *Suppose $\mathcal{I} \rhd M$ is a configuration. Then $(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd N)$ implies $\mathcal{I}' \rhd N$ is also a configuration.*

Proof: By induction on the inference of $(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd N)$, 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 $\mathcal{I}$,

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

where $\stackrel{\hat{\alpha}}{\Longrightarrow}$ 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 $(\mathcal{I} \rhd M)\ \mathcal{R}\ (\mathcal{I} \rhd N)$. ■

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 relaton $\approx_{bis}$ 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

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

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

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

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 $(\mathcal{I} \rhd M \mid k[\![P]\!]) \xrightarrow{\alpha} (\mathcal{I}' \rhd M')$ and now must show that there is a corresponding matching move from

with $l, k \in \mathcal{I}''_{\mathcal{T}}$ also. Therefore, by definition of $\mathcal{R}$ again, we see that

$$\mathcal{I} \models (\mathsf{n\ \ w}\, l : \mathsf{L})(M \mid l[\![C]\!] \mid k[\![\ \ ]\!])\ \mathcal{R}\ (\mathsf{n\ \ w}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![\ \ ]\!]) \quad (25)$$

We know that $(\mathcal{I} \triangleright N \mid k[\![P]\!]) \xrightarrow{\tau} (\mathcal{I} \triangleright (\mathsf{n\ \ w}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![\ \ ]\!]))$ and that $M' \equiv M \mid M'' \equiv (\mathsf{n\ \ w}\, l : \mathsf{L})(M \mid l[\![C]\!] \mid k[\![\ \ ]\!])$, so by (25), we have

$$\mathcal{I} \models M'\ \mathcal{R}\ (\mathsf{n\ \ w}\, l : \mathsf{L})(N \mid l[\![C]\!] \mid k[\![\ \ ]\!])$$

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

$$\mathcal{I} \models M \mid l[\![p!\langle F\rangle]\!]\ \mathcal{R}\ N \mid his$$

where $\mathsf{U}' <: \mathsf{U}$. Call the target knowledge structure $\mathcal{I}'''$. This tells us, by (24) that there exists a matching transition

$$(\mathcal{I}'_\Gamma, \mathcal{I}_\mathcal{T}) \rhd N \xrightarrow{(\tilde{m})k.a.(\tilde{m}', G)!} (\mathcal{I}$$

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 $(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd M')$, where $\mathcal{I} \models M \mathcal{R} N$. We must find a matching move $(\mathcal{I} \rhd N) \stackrel{\alpha}{\Longrightarrow} (\mathcal{I}' \rhd N')$, 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 $\mathcal{C}_\alpha^{\mathcal{I}}$, typeable from $\mathcal{I}_\Gamma, \{k_0 : K_0\}$, which satisfies the conditions of contextuality for

The case in which $(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd M')$ 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 [24], where unicity of the location of channel names was addressed, rather than that of [25], 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 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

safeDpi:

ously, 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 [25]. The judgements also check that the identifiers used in T, U are actually declared appropriately in Γ.

PROPOSITION A.1 (ANITY CHECKS).

- $\Gamma \vdash \mathsf{T} <: \mathsf{U}$ *implies* $\Gamma \vdash$ **env**
- $\Gamma \vdash \mathsf{T} <: \mathsf{U}$ *implies* $\Gamma \vdash \mathsf{T} :$ **ty** *and* $\Gamma \vdash \mathsf{U} :$ **ty**
- $\Gamma \vdash \mathsf{T} <: \mathsf{U}, \Gamma \vdash \mathsf{U} <: \mathsf{R}$ *implies* $\Gamma \vdash \mathsf{T} <: \mathsf{R}$
- $\Gamma, u : \mathsf{T} \vdash$ **env** *implies* $\Gamma \vdash$ **env** *and* $\Gamma \vdash \mathsf{T} :$ **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
  $\mathsf{pr}[u : \mathsf{C}@w] \sqcap \mathsf{pr}[u_1 : \mathsf{C}_1@w, \ldots, u_n : \mathsf{C}_n@w_n] = \mathsf{pr}[]$,
  $\mathsf{pr}[u : \mathsf{C}@w] \sqcup \mathsf{pr}[u_1 : \mathsf{C}_1@w_1, \ldots, u_n : \mathsf{C}_n@w_n] =$
  $\mathsf{pr}[u : \mathsf{C}@w, u_1 : \mathsf{C}_1@w_1 \ldots, u_n : \mathsf{C}_n@w]$
- $\mathsf{pr}[u_1 : \mathsf{GC}_1, \ldots, u_n : \mathsf{GC}_n] \sqcap \pi =$
  $(\mathsf{pr}[u_1 : \mathsf{GC}_1] \sqcap \pi) \sqcup \ldots \sqcup (\mathsf{pr}[u_n : \mathsf{GC}_n] \sqcap \pi)$,
  $\mathsf{pr}[u_1 : \mathsf{GC}_1, \ldots, u_n : \mathsf{GC}_n] \sqcup \pi = \mathsf{pr}[u_1 : \mathsf{GC}_1] \sqcup (\ldots (u_n : \mathsf{GC}_n \sqcup \pi) \ldots)$
- $\mathbf{proc} \sqcap \pi = \pi, \quad \mathbf{proc} \sqcup \pi = \mathbf{proc}$

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:

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

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. ■

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 $\top$.

UBSTITUTIONS: 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:

- $\mathsf{loc}[u' : \mathsf{C}]\{\!|^{v}/_{u}|\!\} = \mathsf{loc}[u'\{\!|^{v}/_{u}|\!\} \ : \ \mathsf{C}\{\!|^{v}/_{u}|\!\}]$
- $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots u_n : \mathsf{C}_n]\{\!|^{v}/_{u}|\!\} =$
  $(\mathsf{loc}[u_1 : \mathsf{C}_1]\{\!|^{v}/_{u}|\!\}) \sqcap \ldots \sqcap (\mathsf{loc}[u_n : \mathsf{C}_n]\{\!|^{v}/_{u}|\!\})$
- $\mathsf{pr}[u \ \mathsf{Tpr}$:

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: Conte tual 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.ces

[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 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.

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