safeDpi: A Language for Controlling Mobile Code
(Extended Abstract)

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

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.

1 Introduction

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

- 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, [11,8,20, 14], a version of the Picalculus, [21], in which processes may migrate between

⋆ Partially supported by EPSRC grant GR/R31485, and EU grants IST-2001-322
⋆⋆ Partially supported by EPSRC grants GR/R33465 and GR/S55538.

I. Walukiewicz (Ed.): FOSSACS 2004, LNCS 2987, pp. 241–256, 2004.

locations, which in turn can be dynamically created. 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 SAFEDPI, the language of this paper, migration is represented by $k[\![\mathsf{goto}_p\, l.F]\!] \longrightarrow l[\![p!\langle F\rangle]\!]$. A thread must designate a *port* p at l in order to migrate. It then reduces to the system $l[\![p!\langle F\rangle]\!]$, which a priori 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.

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*, [23]. 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 with 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:

$$\mathsf{T}_{\mathsf{dep}}(z : \mathsf{L})\, \mathsf{F}_{\mathsf{dep}}(y : \mathsf{w}\langle \mathsf{T} \rangle_{@z} \to \mathsf{pr}[\mathsf{info} : \mathsf{r}\langle \mathsf{T} \rangle_{@\mathsf{here}},\ y : \mathsf{w}\langle \mathsf{T} \rangle_{@z}])$$

where the script type is now parametrised by locations of some type L. This allows the server to accepts 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; the client has to send with the script the actual objects on which their type is parametrised. 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. 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.

We now turn to the second main topic, typed behavioural equivalences. In untyped languages, these are normally defined coinductively, as the largest equivalences over processes which preserve some sort of labelled actions. 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 a well-typed environment will not have the power to participate in them.

Following [9,8] we introduce *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. In Section 5, this will lead, in the standard manner, to a coinductively defined, bisimulation-based, characterisation of contextual equivalence between systems. The paper finishes, in Section 6, with conclusions and a brief survey of related work. Due to space limitations, the detailed definitions, the proofs and many examples are left to the full version [10].

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. *Patterns*, ranged over by X, Y are tuples of variables. Names is partitioned into two sets, Locs ranged over by $k, l, \ldots$ for locations, and Chans ranged over by

$M, N ::=$	*Systems*	$U, V, W ::=$	*Values*
$l[\![P]\!]$	Located Process	$(\tilde{v})$	Tuples
$M \mid N$	Composition	$v ::=$	*Value components*
$(\mathsf{new}\, e : \mathsf{E})\, M$	Name Creation	$(\lambda \tilde{x} : \tilde{\mathsf{T}}) P$	Scripts
$\mathbf{0}$	Termination	u	Identifiers

$P, Q ::=$	*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

Fig. 1. Syntax of SAFEDPI

$a, b, c, \ldots$ 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 $(\mathsf{new}\, e)$. 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, see [10].

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 $\longrightarrow$ between systems and is a mild generalisation of that given in [8,11] for DPI. The main novelty is that 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 a priori 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 $l[\![p?(x)\, Q]\!]$; using another rule for local communication this will now allow the effective entry of F. In this manner the site l can control the immigration of code.

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 build on those in [11,8,23], while the second describes the type environments required to infer that systems are well-typed.

The Types: The collection of types is an extension of those used in [8,11], to which the reader is referred for more background and motivation. They are only described informally here but the reader can consult the full version for more details [10].

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. We also include a *top* type $\top$, which can be associated with any identifier.

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

$$\mathsf{r}\langle \mathsf{T} \rangle \qquad \mathsf{w}\langle \mathsf{T} \rangle \qquad \text{and} \qquad \mathsf{rw}\langle \mathsf{T}_r, \mathsf{T}_w \rangle \quad (\text{when } \mathsf{T}_w <: \mathsf{T}_r)$$

where $\mathsf{T}, \mathsf{T}_r, \mathsf{T}_w$ are *transmission* or *value types*; that is types of values which may be transmitted along channels. If an agent has a name at the latter 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. When the transmit and receive types coincide we abbreviate this type by $\mathsf{rw}\langle \mathsf{T} \rangle$.

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.

Location types, ranged over by K, L: The standard form for these is written $\mathsf{loc}[u_1 : \mathsf{C}_1, \ldots, u_n : \mathsf{C}_n]$ 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. We abbreviate the trivial type $\mathsf{loc}[]$ as loc. We also identify location types up to reorderings.

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 and $\mathsf{C}@w$ denotes the type of a channel of type C located at w. 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 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.

Finally **Transmission** or **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.

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 $u : \mathsf{E}$ where E is one of loc, $\mathsf{C}_{@}w$, $\mathsf{rc}\langle\mathsf{C}\rangle$, $\mathsf{S}_{@}w$ and $\langle\mathsf{T}\ \mathsf{with}\ \tilde{y} : \tilde{\mathsf{E}}\rangle$. The last of these is constrained to the situation in which u is a variable and represents a *package*, which will be used to handle existential types. Intuitively this defines the association $u : \mathsf{T}$ 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; let $\{V : \mathsf{T}\}$ be a list of type assumptions defined by

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

Of course there a lots of other possibilities for V and T but only those mentioned give rise to lists of assumptions. 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 omitted in this extended abstract. We also omit the definition of subtyping judgements, of the form $\Gamma \vdash \mathsf{T} <: \mathsf{U}$. Here it is worth noting that process types are ordered differently than location types. For example we have $\Gamma \vdash \mathsf{loc}[u_1 : \mathsf{C}_1, u_2 : \mathsf{C}_2] <: \mathsf{loc}[u_1 : \mathsf{C}_1]$ but

$$\Gamma \vdash \mathsf{pr}[u_1 : \mathsf{C}_{1@}k] <: \mathsf{pr}[u_1 : \mathsf{C}_{1@}k,\ u_2 : \mathsf{C}_{2@}l]$$

assuming, of course, that the various types used are well-defined relative to Γ.

Type Inference: We now 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 straightforward adaptations from the analogous rules in [11,8]. The intention is that whenever such a judgement can be inferred it will follow that Γ is a well-formed environment.

The typing rules for the judgements on processes and values, $\Gamma \vdash_{w} P : \pi$ and $\Gamma \vdash V : \mathsf{T}$, are defined simultaneously and we give the more interesting rules for these in Figure 2.

$$\frac{\Gamma \vdash_w v_i : \mathsf{E}_i\{\!|\tilde{v}/\tilde{x}|\!\} \quad \Gamma \vdash_w v : \mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}}{\Gamma \vdash_w (\widetilde{v}, v) : \mathsf{Tdep}(\widetilde{x} : \widetilde{\mathsf{E}})\,\mathsf{T}} \text{(ty-TuDep)}$$

$$\frac{\Gamma \vdash_w v_i : \mathsf{E}_i\{\!|\tilde{v}/\tilde{x}|\!\} \quad \Gamma \vdash_w v : \mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}}{\Gamma \vdash_w \langle\widetilde{v}, v\rangle : \mathsf{Edep}(\widetilde{x} : \widetilde{\mathsf{E}})\,\mathsf{T}} \text{(ty-EDep)}$$

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

$$\frac{\Gamma \vdash_w \langle\tilde{v}, v\rangle : \mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{T}}{\Gamma \vdash_w v : \mathsf{T}\{\!|\tilde{v}/\tilde{x}|\!\}} \text{(ty-Unpack)}$$

$$\frac{\Gamma \vdash_w V : \mathsf{T} \quad \Gamma \vdash \mathsf{pr}[u : \mathsf{w}\langle\mathsf{T}\rangle@w] <: \pi \quad \Gamma \vdash \mathsf{pr}_{\mathsf{ch}}[V : (\mathsf{T})@w] <: \pi}{\Gamma \vdash_w u!\langle V\rangle : \pi} \text{(ty-out)}$$

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

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

$$\frac{\Gamma \vdash_w P : \pi \quad \Gamma \vdash \pi <: \pi'}{\Gamma \vdash_w P : \pi'} \text{(ty-subproc)}$$

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

$$\frac{\Gamma \vdash_w F : \mathsf{Fdep}(\tilde{x} : \tilde{\mathsf{T}} \to \pi) \quad \Gamma \vdash_w v_i : \mathsf{T}_i}{\Gamma \vdash_w F\ (\widetilde{v}) : \pi\{\!|\tilde{v}/\tilde{x}|\!\}\{\!|w/\mathsf{here}|\!\}} \text{(ty-beta)}$$

Fig. 2. Selected rules for typing values and processes

Let us first examine those for values. 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 it's required type. The rule (ty-EDep) allows us to construct such values and is similar to the rule for dependent tuples. Dependent tuples can be deconstructed and their components accessed in the standard manner. 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 typing processes, rules (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 premis of the former; the arguments in an abstraction are relativised to the current location w. However the real interest is in the typing of the input and output processes. For example 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.

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, (premise $\Gamma \vdash \mathsf{pr}[u : \mathsf{r}\langle\mathsf{T}\rangle_{@w}] <: \pi$) 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, (premise $\Gamma,\ \{X : (\mathsf{T})_{@w}\} \vdash_w P : \pi \sqcup \mathsf{pr}_{\mathsf{ch}}[X : \mathsf{T}_{@w}]$).

We conclude this section with the following theorem, which is proved in [10].

Theorem 1 (Subject Reduction). *Suppose $\Gamma \vdash M$. Then $M \longrightarrow N$ implies $\Gamma \vdash N$.*

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 $\mathbf{unit} \to \mathsf{pr}[....]$

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.

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

Client: $\mathrm{CL}[\![\mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,\mathrm{CL}.\,\mathsf{report}!\langle x\rangle \mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots]\!]$

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; that is suppose in Γ we have the entries: $\{\mathsf{news} : \mathsf{rw}\langle\mathbf{string}\rangle_{@s}, \mathsf{req} : \mathsf{rw}\langle\mathsf{S}\rangle_{@s}, \mathsf{report} : \mathsf{rw}\langle\mathbf{string}\rangle_{@\mathrm{CL}}, \mathsf{in} : \mathsf{rw}\langle\mathsf{R}\rangle_{@\mathrm{CL}}\}$.

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

$$\mathsf{R} : \mathsf{thunk} \qquad\qquad \mathsf{S} : \mathsf{th}[\mathsf{news} : \mathsf{r}\langle\mathbf{string}\rangle_{@s},\ \mathsf{in} : \mathsf{w}\langle\mathsf{R}\rangle_{@\mathrm{CL}}]$$

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 CL; moreover the local channel news can only be used in read mode. With these types one can show that the overall system is well-typed.

The current type R, *i.e.* 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

$$\mathsf{R} : \mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle_{@\mathrm{CL}}]$$

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 CL. One can also check that with these new restrictive guardians the system is still well-typed.

Dependent Process Types: There remains a major difficulty with the servers above. The guardian type of the server S uses the name of the client CL, and therefore it can only be used by that client. To overcome this difficulty and allow servers to be accessed by different clients we need to allow process types to depend on locations and channels by using the dependency type, $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{S}$. An example of the use of such types is in the following variation of the client server from above:

$$\begin{aligned}&\text{Server:}\quad s[\![\mathsf{req}?(\xi\ \mathsf{with}\ y : \mathsf{S}_d)\,\mathsf{run}\,\xi \mid *\,\mathsf{news}!\langle scandal\rangle]\!]\\ &\text{Client:}\quad \mathrm{CL}[\![(\mathsf{newc}\ \mathsf{report})\ (\mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,\mathrm{CL}.\,\mathsf{report}!\langle x\rangle\ \mathsf{with}\ \mathrm{CL}\\ &\qquad\qquad \mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \mid \mathsf{report}?(y)\ldots)]\!]\end{aligned} \tag{1}$$

with the types

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

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 CL can generate a new reply channel report and send to the server the thunk $\mathsf{news}?(x)\,\mathsf{goto}_{\mathsf{in}}\,\mathrm{CL}.\,\mathsf{report}!\langle x\rangle$ accompanied by a required location, in this case CL.

The example we have just considered, (1), a priori 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:

$$\begin{array}{ll} \text{Server:} & s[\![\mathsf{req}?(\xi\ \mathsf{with}\ (y,z,x) : \mathsf{S}_d)\,\mathsf{run}\,\xi \ \mid\ *\,\mathsf{news}!\langle \mathit{scandal}\rangle]\!] \qquad (2)\\ \text{Client:} & \mathrm{CL}[\![(\mathsf{newc}\,\mathsf{report}, \mathsf{in} : \mathsf{rw}\langle\mathsf{R}\rangle)\ (\mathsf{goto}_{\mathsf{req}}\, s.\mathsf{news}?(x)\\ & \mathsf{goto}_{\mathsf{in}}\,\mathrm{CL}.\,\mathsf{report}!\langle x\rangle\ \mathsf{with}\ (\mathrm{CL}, \mathsf{report}, \mathsf{in}) \mid \mathsf{in}?(\xi : \mathsf{R})\,\mathsf{run}\,\xi \ \mid \mathsf{report}?(y)\ldots) \end{array}$$

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 $\mathsf{th}[\mathsf{report} : \mathsf{w}\langle\mathbf{string}\rangle@\mathrm{CL}]$. 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 (CL, report, 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.

Existential Process Types: The use of dependent script types has certain disadvantages from the point of view of the clients. For example in (2) above the client sends to the server, in addition to the script to be executed, the triple (CL, report, in). Although these are not used by the server we have defined other than as part of the received script, servers are in principle able to use them in any way they seem fit. An alternative server could be given by

$$\text{badServer:} \qquad s[\![\mathsf{req}?(\xi\ \mathsf{with}\ (y,z,x) : \mathsf{S}_d)\,\mathsf{goto}_x\, y.z!\langle \mathit{boring}\rangle]\!] \qquad (3)$$

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, $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{E}})\,\mathsf{S}$, allow the client to hide from the server the data which accompanies the incoming scripts. Let us now reformulate (2) above using existential types:

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

Here the guardian type S_e is

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

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.

Now let us reconsider the badServer from (3) above. Using existential types this example might be written

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

But one can show that this no longer typechecks. The detailed type inferences as well as other examples are found in [10].

5 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 $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{A}$, $\mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S}$, or $\mathsf{Edep}(\tilde{x} : \tilde{\mathsf{A}})\,\mathsf{S}$. Simple scripts may be simulated via the empty dependent type $\mathsf{Tdep}()\,\mathsf{S}$, as can simple first-order values, via the type $\mathsf{Tdep}()\,\mathsf{A}$. So our results can easily be extended to the full language.

5.1 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. Such equivalences, for Picalculus and Dpi, can be characterised inductively using actions of the form $(\Gamma \rhd M) \xrightarrow{\mu} (\Gamma' \rhd M')$ where $(\Gamma \rhd M)$, $(\Gamma' \rhd M')$

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.

Consider $\Gamma = l : \mathsf{loc}, b : \mathsf{rc}\langle\mathsf{rw}\langle\mathbf{unit}\rangle\rangle, a : \mathsf{rw}\langle\mathsf{loc}[b : \mathsf{rw}\langle\mathbf{unit}\rangle]\rangle@l$ and

$$M = (\mathsf{new}\, k : \mathsf{loc}[b : \mathsf{rw}\langle\mathbf{unit}\rangle])\, l[\![a!\langle k\rangle]\!] \mid k[\![b!\langle\rangle]\!]$$
$$N = (\mathsf{new}\, k : \mathsf{loc}[b : \mathsf{rw}\langle\mathbf{unit}\rangle])\, l[\![a!\langle k\rangle]\!] \mid k[\![\mathsf{stop}]\!]$$

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 not distinguish them. 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.

Definition 2 (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.*

Definition 3 (Configurations). *A* configuration *is written as* $\mathcal{I} \rhd M$ *where*

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

Definition 4 (Knowledge-indexed relations). *A* knowledge-indexed relation *over systems is a family of binary relations between systems indexed by knowledge structures. 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,* $\mathcal{I} \rhd M$ *and* $\mathcal{I} \rhd N$ *are both configurations.*

$$\frac{k \in \mathcal{I}_\mathcal{T} \quad a : \mathsf{w}\langle \mathsf{T} \rangle_@ k \in \mathcal{I}_\Gamma \quad \mathcal{I}_\Gamma \vdash_k V : \mathsf{T}}{(\mathcal{I} \rhd k[\![a?(X : \mathsf{T})\, P]\!]) \xrightarrow{k.a.V?} (\mathcal{I} \rhd k[\![P\{\!|^V\!/_{X}|\!\}]\!])} \qquad \frac{k \in \mathcal{I}_\mathcal{T} \quad a : \mathsf{w}\langle \mathsf{T} \rangle_@ k \in \mathcal{I}_\Gamma \quad \mathcal{I}_\Gamma \vdash_k V : T}{(\mathcal{I} \rhd M) \xrightarrow{k.a.V?} (\mathcal{I} \rhd M \mid k[\![a!\langle V \rangle]\!])}$$

$$\frac{k \in \mathcal{I}_\mathcal{T} \quad \mathsf{T} \text{ a first-order type} \quad a : \mathsf{r}\langle \mathsf{T} \rangle_@ k \in \mathcal{I}_\Gamma \quad \mathcal{I}_\Gamma \sqcap \{\tilde{u} : \mathsf{T}\} \vdash \mathbf{env}}{(\mathcal{I} \rhd k[\![a!\langle \tilde{u} \rangle]\!]) \xrightarrow{k.a.\tilde{u}!} (\mathcal{I} \sqcap \{\tilde{u} : \mathsf{T}\} \rhd k[\![\mathsf{stop}]\!])} \qquad \frac{k \in \mathcal{I}_\mathcal{T} \quad \mathsf{T} \text{ of the form } \mathsf{Edep}(\tilde{x} : \tilde{T})\, S \quad a : \mathsf{r}\langle \mathsf{T} \rangle_@ k \in \mathcal{I}_\Gamma \quad \mathcal{I}_\Gamma \vdash_k G : \mathsf{T} \to \mathbf{proc}}{(\mathcal{I} \rhd k[\![a!\langle F \rangle]\!]) \xrightarrow{k.a.G!} (\mathcal{I} \rhd k[\![G\ (F)]\!])}$$

$$\frac{k \in \mathcal{I}_\mathcal{T} \quad \mathsf{T} \text{ of the form } \mathsf{Tdep}(\tilde{x} : \tilde{\mathsf{E}})\, \mathsf{S} \quad a : \mathsf{r}\langle \mathsf{T} \rangle_@ k \in \mathcal{I}_\Gamma \quad \mathcal{I}_\Gamma \sqcap \{\tilde{u} : \tilde{\mathsf{E}}\} \vdash \mathbf{env} \quad \mathcal{I}_\Gamma \vdash_k G : T \to \mathbf{proc}}{(\mathcal{I} \rhd k[\![a!\langle (\tilde{u}, F) \rangle]\!]) \xrightarrow{k.a.(\tilde{u},G)!} (\mathcal{I} \sqcap \{\tilde{u} : \tilde{\mathsf{E}}\} \rhd k[\![G\ (\tilde{u}, F)]\!])} \qquad \frac{k \notin \mathcal{I}_\mathcal{T} \quad \mathcal{I}_\Gamma \vdash_k p!\langle V \rangle : \mathbf{proc}}{(\mathcal{I} \rhd M) \xrightarrow{\mathsf{go}_p k.V} (\mathcal{I} \rhd M \mid k[\![p!\langle V \rangle]\!])}$$

Fig. 3. Labelled Transition System Axioms

We will use knowledge-indexed relations to propose a notion of behavioural equivalence appropriate to this setting. We do this in an established manner [12,6,9] by proposing that we consider the largest equivalence which respects reductions, a suitable notion of observation barb, and is closed under system contexts. We write $\approx_{cxt}$ for the largest such relation.

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.

5.2 A Bisimulation Equivalence

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

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

where it is assumed that $k, a, p \notin \tilde{n}, \tilde{m}$. Most of these actions will be familiar to those familiar with PICALCULUS labelled transitions. The action $\mathsf{go}_p k.F$ 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.

With this notation we define judgements of the form

$$(\mathcal{I} \rhd M) \xrightarrow{\alpha} (\mathcal{I}' \rhd N) \tag{4}$$

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 N, and the knowledge, from $\mathcal{I}$ to $\mathcal{I}'$. Typically this is an *increase* in knowledge of the testing environment of the system, represented as the change from the type environment, $\mathcal{I}_\Gamma$ to $\mathcal{I}'_\Gamma$. Consequently we need a notation for augmenting type environments. We extend that used in [8], by defining a partial binary operation $\Gamma \sqcap \Gamma'$ on lists of type associations. Intuitively this constructs a new list of associations by combining those in Γ and Γ'; if u happens to have a type in both, say E and E' then in the newly constructed list it will have the type $\mathsf{E} \sqcap \mathsf{E}'$; for the operation to be defined all such meets are required to exist. The details of the construction are omitted as it is only a mild generalisation of that from [8].

The axioms for the judgements (4) are given in Figure 3; these are based on the rules in Figure 10 of [8]. Further, structural rules are also required but the details are omitted here. The standard definition of bisimulation gives a coinductive bisimilarity relation over configurations: 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, we are able to compare bisimilarity directly with the touchstone behavioural equivalence $\approx_{cxt}$ which leads us to our second main result

Theorem 5 (Full abstraction of $\approx_{bis}$ for $\approx_{cxt}$). *$\mathcal{I} \models M \approx_{cxt} N$ if and only if $\mathcal{I} \models M \approx_{bis} N$.*

6 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, [22], [13], 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.

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 [11], [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.

References

1. Michele Boreale and Davide Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proc. 13th LICS Conf.* IEEE Computer Society Press, 1998.
2. Luca Cardelli, Giorgio Ghelli, and Andrew Gordon. Ambient groups and mobility types. In *Proc. IFIP TCS 2000*, volume 1872 of *LNCS*. Springer-Verlag, 2000.
3. Luca Cardelli and Andrew Gordon. Mobile ambients. In *Proc. FoSSaCS '98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
4. Giuseppe Castagna and Francesco Zappa Nardelli. The Seal calculus revisited: Contextual equivalences and bisimilarity. In *Proc. of FSTTCS*, LNCS, 2002.
5. Giuseppe Castagna, Jan Vitek, and Francesco Zappa Nardelli. The Seal calculus. 2003. Available from ftp://ftp.di.ens.fr/pub/users/castagna/seal.ps.gz.

6. Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In *Proc. CONCUR*, volume 1119 of *LNCS*. Springer-Verlag, 1996.
7. Matthew Hennessy and Massimo Merro. Bisimulation congruences in safe ambients. In *Proc. POPL '02, the* 29^{th} *ACM Symposium on Principles of Programming Languages*. ACM Press, 2002.
8. Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. Technical Report 2002:01, COGS, University of Sussex, 2002. Extended Abstract published in the Proc. of FoSSaCS 2003.
9. Matthew Hennessy and Julian Rathke. Typed behavioural equivalences for processes in the presence of subtyping. In James Harland, editor, *Proc. CATS '02, Computing: Australasian Theory Symposium*, volume 61 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2002. To appear in *Mathematical Structures in Computer Science*.
10. Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. The full version of this paper. Technical Report 2003:02, Department of Informatics, University of Sussex, 2003. Available from www.cogs.susx.ac.uk/julianr/.
11. Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
12. Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 152(2):437–486, 1995.
13. Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proc. of POPL '02, the* 29^{th} *ACM Symposium on Principles of Programming Languages*, pages 331–342, 2002.
14. Cédric 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*, LNCS. Springer-Verlag, 2003.
16. Massimo Merro and Vladimiro Sassone. Typing and subtyping mobility in boxed ambients. In *Proc. CONCUR 02*, volume 1644 of *LNCS*. 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 *LNCS*, pages 25–35. Springer-Verlag, 1998.
18. George C. Necula. Proof-carrying code. In *Proc. of POPL '97, the* 24^{th} *ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.
19. Benjamin Pierce and Davide 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 *Proc. of POPL '99, the* 26^{th} *ACM 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. David Walker. A type system for expressive security properties. In *Proc. Popl '00, the* 27^{th} *ACM Symposium on Principles of Programming Languages, Boston*, pages 254–267, 2000.
23. Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. *Information and Computation*, 172:82–120, 2002.