A Typed Language for Distributed Mobile Processes

(Extended abstract)

James Riely and Matthew Hennessy*

Abstract

We describe a foundational language for specifying dynamically evolving networks of distributed processes, Dπ. The language is a distributed extension of the π-calculus which incorporates the notions of remote execution, migration, and site failure. Novel features of Dπ include

1. Communication channels are *explicitly located*: the use of a channel requires knowledge of both the channel and its location.
2. Names are endowed with *permissions*: the holder of a name may only use that name in the manner allowed by these permissions.

A type system is proposed in which the types control the allocation of permissions; in well-typed processes all names are used in accordance with the permissions allowed by the types. We prove Subject Reduction and Type Safety Theorems for the type system. In the final section we define a semantic theory based on *barbed bisimulations* and discuss its characterization in terms of a bisimulation relation over a relativized labelled transition system.

1 Introduction

Due to the rapid advances in networking technology there has been a recent proliferation of commercial programming languages for distributed processes, such as TeleScript, Java and ActiveX. This has been accompanied by the development of prototype languages — such as Obliq [9], Pict [17], Oz [23], Facile [13] and the join calculus (DJoin) [11, 12] — within the programming language research community, and the development of more abstract calculi, such as the π-calculus [15] and its variations [2, 3, 18, 20, 21, 22], that directly address semantic and verification issues. This work should be considered a contribution to this last research effort.

*Research funded by HCM EXPRESS and EPSRC project GR/K60701. Authors' address: School of Cognitive and Computing Sciences, Univ. of Sussex, Falmer, Brighton, BN1 9QH, UK, {jamesri,matthewh}@cogs.susx.ac.uk

To appear in *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

Our goal has been to develop a robust and useful semantic theory for a process language in which computation is distributed over different *locations*, in which processes may *migrate* from one site to another, and in which sites may *fail*. We present a foundational language, called Dπ, for describing distributed systems. The starting point is the π-calculus, a language in which processes are described in terms of their ability to send and receive values along communication channels. Although the values allowed in the π-calculus are very simple — only names may be communicated — the calculus is still very powerful, due to the ability to generate *new* names which can be communicated and shared privately between processes.

As in [2, 12] we use a subset of names to represent *locations*, or *sites*, which can also be freshly generated and exchanged between processes. Distribution is achieved by requiring that each *basic* process (or *thread*) be *located*. The thread P running at site ℓ is denoted $\ell[P]$; collections of such terms are called *located* processes (or simply *processes*). Thus, in the process

$$(\nu_\ell a)\big(\ell[P] \mid k[Q]\big) \mid \ell[R]$$

location ℓ is running threads P and R, whereas k is running thread Q; in addition, P and Q share a private channel a, located at ℓ. Newly created locations are also located and therefore the collection of extant locations forms a *tree* based on a *sublocation* relation. As in TeleScript, communication is purely local; in order to send a value on a channel a a thread must first move to the location at which a is declared. Thus the communication construct of DJoin is here syntactically split in two, with the syntax more closely matching the reduction semantics. In addition certain forms of site failure can be modeled using a "halt" primitive. The syntax of Dπ and its reduction semantics are given in Section 2.

In Section 3, we introduce a type system for Dπ which allows a programmer to control the *capabilities*, or *permissions*, associated with each instance of a channel or location. For example, one may wish to export the capability to communicate with a particular location without also exporting the capability to halt all processes running at that location (*i.e.* to "kill" the location). Channels are constrained by the values which they may transmit and both locations and channels are constrained by the following *permissions*, which restrict their use:

snd to *send* data along a channel,
rcv to *receive* data along a channel,

run to *run* a thread at a location,
newc to create *new channels* at a location,
subl to place *sublocations* at a location,
mig to *move* a location (with its threads and sublocations), and
halt to *kill* a location, stopping it from running any threads.

We prove a Type Safety Theorem using a tagged version of the language, where each occurrence of a name is explicitly tagged with a set of permissions indicating the manner in which that instance of the name may be used. When a name is communicated, certain permissions are communicated with it, as negotiated at the time of communication. Well-typed terms are guaranteed to use received names only as allowed by the received permissions.

The typing system is based on that of Pierce and Sangiorgi [16]; however, the related theorems and proof techniques — in particular our formulation of the tagged language — appear to be novel.

In Section 4 we outline a semantic theory for Dπ. We define a variation on *barbed congruence* [20] relativized to a typing constraint on the environment. Suppose that P and Q are processes and Δ is a type environment intended to constrain the activity of processes interacting with P and Q. We then say, roughly, that P and Q are *barbed congruent at* Δ if for every well-typed context $\mathbb{C}$ which satisfies the constraint Δ, $\mathbb{C}[P]$ and $\mathbb{C}[Q]$ are *barbed bisimilar* [20]. We discuss an alternative characterization of this congruence as a *labelled* bisimulation relation over a labelled transition system for well-typed processes, relativized to constraints. The labels in the transition system identify the actions of input and output, failure and migration.

We discuss related work in the conclusion.

Due to space limitations, proofs are presented in outline form and the details of the labelled bisimulation relation (Section 4) have been omitted entirely.

2 Language

2.1 Syntax

The language we define, Dπ, may be seen as a distributed version of core Pict, [17], with facilities for local synchronous communication, code movement, and failure.

The syntax is defined using a set *Loc* of *locations* or *sites*, *k-m*, a set *Chan* of *channels* or *ports*, *a-c* and a set *Var* of *variables*, *x-z*. We let r and s range over the set of *names*, $Name = Loc \cup Chan$. The set *Id* of *identifiers*, *u-w*, includes names and variables, as described in Section 1. For the moment we will ignore types, ζ, and tags, γ, in our description of the language.

The syntax of *basic processes* or *threads*, given in Section 1, looks very much like that of the π-calculus, with some extensions. Input and output operations are placed at channels using the prefix $u?F$ or $u!C$, where u is an

$$
\begin{array}{rrcl}
\textit{Pred}: & \varphi,\psi & ::= & \uparrow u \mid \rightarrow u \mid \twoheadrightarrow u \mid u = v \\
\textit{BProc}: & P\text{-}R & ::= & \mathsf{nil} \mid P \mid Q \mid (\nu a{:}\kappa)P \mid (\nu m{:}\lambda)P \mid *P \\
 & & \mid & \textstyle\sum_i u_i?(X_i{:}\zeta_i)P_i \mid u!\langle V\rangle P \\
 & & \mid & u::P \mid \mathsf{mig}\,u.P \mid \mathsf{halt} \\
 & & \mid & \mathsf{if}\ \varphi\ \mathsf{then}\ P\ \mathsf{else}\ Q \\
\textit{LProc}: & P\text{-}R & ::= & \mathsf{nil} \mid P \mid Q \mid (\nu_\ell a{:}\kappa)P \mid (\nu^\delta_\ell m{:}\lambda)P \quad \{P,Q \in \textit{LProc}\} \\
 & & \mid & \ell_\gamma[P] \quad \{P \in \textit{BProc}\} \\
\textit{Type}: & \kappa & ::= & \mathsf{chan}_\gamma\zeta \quad \gamma \subset \{\mathbf{snd},\mathbf{rcv}\} \\
 & \lambda & ::= & \mathsf{loc}_\gamma \quad \gamma \subseteq \{\mathbf{run},\mathbf{halt},\mathbf{mig},\mathbf{newc},\mathbf{subl}\} \\
 & \zeta,\xi & ::= & \kappa \mid \lambda \mid \lambda::\widetilde{\kappa} \mid \widetilde{\zeta} \\
\textit{Id}: & u\text{-}w & ::= & x \mid a_\gamma \mid \ell_\gamma \\
\textit{Val}: & U\text{-}V & ::= & u \mid u::V \mid (V^1 \ldots V^n),\ n \geq 0 \\
\textit{Pat}: & X\text{-}Z & ::= & x \mid x::X \mid (X^1 \ldots X^n),\ n \geq 0
\end{array}
$$

Table 1: Syntax

identifier, F is an *abstraction* $(X{:}\zeta)P$, and C is a *concretion* $\langle V\rangle Q$. More generally we allow finite choice of input guards $\sum u_i?(X_i{:}\zeta_i)P_i$, where i ranges over an implicit finite index set I.

In addition to communication, termination (nil), parallel composition $(P \mid Q)$, iteration $(*P)$, and restriction $((\nu r)P)$ — all of which appear in some form in the π-calculus — the thread language includes the constructs:

- $u::P$, pronounced "go to u", which moves the thread P to location u,
- $\mathsf{mig}\,u.P$, pronounced "migrate to u", which moves the *current location* of the thread to be a sublocation of location u,
- halt, which halts the current location, and
- if φ then P else Q, which allows the thread to test the *position* of the current location (relative to the sublocation relation), to test the running/halted *status* of any location and to compare names (*cf.* (mis)matching in the π-calculus).

The thread P running at site ℓ is denoted $\ell[P]$; collections of such terms are called *located* processes (or simply *processes*). Thus, as explained in the Introduction, in the process

$$(\nu_\ell a)\big(\ell[P] \mid k[Q]\big) \mid \ell[R]$$

location ℓ is running threads P and R, whereas k is running thread Q; in addition, P and Q share a private channel a, located at ℓ. We say that ℓ is the *current location* of P and Q and k is the current location of Q. (It is worth noting that

while we distinguish basic and located processes, we use the metavariables *P*-*R* for both; the intended meaning should be clear from context.)

Restriction. For located processes, there are two forms of restriction, for channels and locations, respectively. When a thread running at ℓ creates a new name r, the name r is considered to be located at ℓ. At the level of located processes, we note this by writing $(\nu_\ell r)$, indicating that ℓ is the parent of r. The parent location of a channel is static, whereas the parent of a location may change as a result of a migration. In addition, a location, once created, may have one of two states: running ($\uparrow$) and halted ($\downarrow$). Within a thread this information need not be recorded as we assume that every location is live when it is first created. Within a process, however, the state of a location may change as a result of a halt operation and therefore we additionally record the state of a bound location. Thus $(\nu_\ell^\uparrow m{:}\zeta)P$ indicates that the located process P has a private location m which is a *live* child of ℓ, while $(\nu_\ell^\downarrow m{:}\zeta)P$ indicates that m is a private *dead* child of ℓ. Admittedly the latter is of little use but since locations can be killed, and their status tested, this information must be retained for the operational semantics. The general form of restriction for locations is $(\nu_\ell^\delta m{:}\zeta)$ where $\delta \in \{\uparrow, \downarrow\}$.

Names, variables and values. In the π-calculus the only *values* that can be transmitted between processes are names. Here we allow a more general set of values, and therefore in the input construct $a?(X{:}\zeta)\,P$, X is a *pattern* against which the more general values may be matched. The allowed class of values, and associated patterns, are defined in Section 1. (For $(X{:}\zeta)\,P$ to be well formed, the pattern X must be *linear*, *i.e.* each variable may appear at most once, and the structure of X must match the structure of ζ.) Note that the set of values includes the set of patterns. We say that a value is *closed* if it contains no variables.

Closed values of channel type have the form a, whereas those of location type may have the form ℓ or the form $\ell{::}\widetilde{a}$. Intuitively, when a location is communicated, a process may also communicate a subset of the channels defined at that location. For example, a process that receives the value $\ell{::}(a,b)$ is granted knowledge of the location ℓ and the channels a and b located at ℓ.

We assume the standard notion of *free* and *bound* occurrences of variables and names. Variables are bound by the input construct, whereas names are bound by restriction. A term with no free variables is *closed*. Except where noted, we assume all terms are closed. We also assume the standard notions of *alpha-conversion* and *substitution*, where $P\{\!\{^u/_x\}\!\}$ denotes the capture-avoiding substitution of u for x in P. The notation, $P\{\!\{^U/_X\}\!\}$ generalizes this in an obvious way; for $\{\!\{^U/_X\}\!\}$ to be well-defined, it must be that the structure of the U exactly matches the structure of X.

Types. Our main interest in introducing types is to control the *capabilities*, or *permissions*, associated with each instance of a name. For example when exporting a new location name it is natural to wish to limit the use of that location by the recipient. We use the set of permissions given in the Introduction. These are indicative examples chosen from a large range of possibilities. One can easily think of other capabilities that would be interesting to control, such as the ability to test for equality between names, to *communicate* them, or even to communicate them with restricted permissions.

The syntax for types, ζ, is built up using sets of the capabilities, γ, as given in Section 1. Channels and locations are of types κ and λ respectively, whereas tuples of values have type $\widetilde{\zeta}$. The channel types generalize those of Pierce and Sangiorgi [16] and location types are an extension of the same approach to locations. Note that in the channel type, $\mathsf{chan}_\gamma\zeta$, γ dictates the use of the channel (send, receive, neither or both) while the type ζ constrains values that can be *communicated* on the channel. The meaning of the type $\lambda{::}\widetilde{\kappa}$ is less obvious. Values of this type are used to communicate channels at fresh locations. For example, the process $\ell[a!\,\langle k{::}b\rangle]$ may send the value $k{::}b$ along channel a. This value informs the receiver (who must also be located at ℓ) of location k and of the channel b located at k. In order to use b in any way (*e.g.* to distinguish it from other names at k), the receiver must receive the **run** capability on k.

Note that, as in [16] in the syntax for processes all bound occurrences of identifiers must have associated with them an explicit type. Thus in the thread $a?(X{:}\zeta)\,P$, the type ζ indicates the type of value which can be received, whereas in the process $(\nu_\ell r{:}\xi)P$ the type ξ determines the capabilities of the newly created name r; here we call ζ a *reception type* and ξ an *allocation type*.

As stated above we are mainly interested in the type system as a way of controlling permissions, and thus we have not endowed it with features such as recursive types, polymorphism, linearity, etc., many of which are entirely separate concerns; for example, the generalization to recursive types is straightforward [16, 24].

In Section 3 we introduce a type system for Dπ and prove that well-typed programs are free of runtime type errors; to prove that the type system is safe, we also give a definition of *runtime error* in Section 3. The definition formalizes the intuition that an error occurs when there is an arity mismatch in a communication (as in the polymorphic π-calculus), or, for example, when a process attempts to use a name without obeying the proper restrictions on its permissions. Consider a process P which sends a fresh location ℓ to Q, explicitly denying the **halt** capability; if Q subsequently attempts to kill ℓ then a runtime error occurs. Note that different instances of a name may have different permissions: P may have the **halt** capability on ℓ, even though it does not communicate this capability to Q.

Tags. In order to define runtime errors, all instances of names in Dπ are *tagged* with a capability set (excluding the restriction operators, whose meaning is independent of tags). The name r tagged with capabilities γ is written r_γ. However

the type system, in Section 3, does not refer to these tags, and after proving the Type Safety and Subject Reduction Theorem these tags can be ignored so long as one considers only well-typed processes. For this reason in most of the informal examples discussed in the paper tags will not be used.

Notation. We end this introduction to the syntax with a description of some convenient notation.

- We write $\mathrm{fn}(P)$ for the function which returns the set of free names occurring in P. Similarly, $\mathrm{locs}(P)$ returns the set of free locations occurring in P, and $\mathrm{n}(P)$ returns the set of *all* names occurring in P. These functions are also defined, in the obvious way, for other syntactic categories.
- We routinely drop annotations from terms when they are uninteresting or clear from context; thus we may write $(\nu_\ell^\delta m{:}\zeta)$ as $(\nu_\ell^\delta m)$, $(\nu_\ell m)$, $(\nu m{:}\zeta)$ or simply (νm).
- We often denote groups of similar things using a tilde; *e.g.* we write $(\nu\widetilde{r})P$ instead of $(\nu r^1)\ldots(\nu r^n)P$ and $\widetilde{a}$ instead of $(a^1\ldots a^n)$. We also adopt other standard abbreviations from the π-calculus.
- We use underscores (*e.g.* $\underline{r}$ and $\underline{\underline{r}}$) to indicate that a name is tagged, and define the projection functions "name" and "perm" in the obvious way. We adopt the meta-syntactic convention that $\mathrm{name}(\underline{r}) = \mathrm{name}(\underline{\underline{r}}) = r$, although $\mathrm{perm}(\underline{r})$ and $\mathrm{perm}(\underline{\underline{r}})$ may differ. We also use the function "perm" on simple types: for example, $\mathrm{perm}(\mathsf{chan}_\gamma\kappa) = \gamma$. □

2.2 Reduction semantics

We give the operational semantics to Dπ in terms of a reduction relation between *process configurations*. The judgments are of the form

$$\mathcal{L} \triangleright P \longrightarrow \mathcal{L}' \triangleright P'$$

where P and P' are (closed) located processes, and $\mathcal{L}$ and $\mathcal{L}'$ are runtime environments for the system, recording the position and status of each location. We sometimes refer to $\mathcal{L}' \triangleright P'$ as the *continuation* or the *residual* of $\mathcal{L} \triangleright P$.

To see that the position and status of locations must be recorded dynamically, observe that the position may change due tomigration and the status may change due to a halt. Note that even without the conditional construct, the position and status of a location *do* affect the meaning of processes. For example, the term $\ell[P]$ is unable to reduce if ℓ or any of its ancestors is halted. In addition to direct execution of the halt operation, a location will halt if it migrates to a parent location that is halted.

Location trees. To represent the runtime environment we take $\mathcal{L}$ to be a *location tree*, *i.e.* a tree with nodes drawn from *Loc* (each location name may appear at most once). In addition to the position of locations in the tree, L records also the status δ of each node $\ell \in \mathrm{locs}(\mathcal{L})$, where as before $\delta \in \{\uparrow, \downarrow\}$. We suppose that the root node of the tree is always live, as otherwise reduction is impossible. Below the root, the "top-level" locations are meant, intuitively, to correspond to physical machines or network addresses, while subsequent descendants might correspond to "processes" or "subprocesses".

We do not give an implementation of location trees, but rather describe them abstractly. Location trees support the following predicates:

- $\mathcal{L} \vdash_\ell \uparrow k$ if k and all of its ancestors are alive;
- $\mathcal{L} \vdash_\ell \to k$ if k is the parent of ℓ; and
- $\mathcal{L} \vdash_\ell \twoheadrightarrow k$ if k is an ancestor of ℓ (other than its parent).

These predicates, together with the (mis)matching construct $u = v$, make up the formulae φ of Section 1. We write $\mathcal{L} \vdash \varphi$ if the location ℓ is unimportant for establishing the property φ; note that this is the case for the matching predicate. In the semantics we also use conjunction and negation, with the obvious meanings.

Location trees also support the following functions, which we write postfix:

- $\mathcal{L}, {}^{\delta}_{k}\ell$ adds location ℓ as a child k with status δ. To be defined, k must appear in $\mathcal{L}$ and ℓ must not; therefore in $\mathcal{L}, {}^{\delta}_{k}\ell$ the node ℓ is a leaf.
- $\mathcal{L}\{\!|\ell{\to}k|\!\}$ changes the tree ordering so that k is the parent of ℓ. To be defined the operation must preserve the tree structure.
- $\mathcal{L}\{\!|{\downarrow}\ell|\!\}$ marks ℓ as dead.

The definition of the judgments $\mathcal{L} \triangleright P \longrightarrow \mathcal{L}' \triangleright P'$ is given in Section 2 where it is assumed that all the process configurations are *well formed*, *i.e.* all of the free locations in P are found in $\mathcal{L}$ (*i.e.* $\mathrm{locs}(P) \subseteq \mathrm{locs}(\mathcal{L})$).

Structural equivalence. Following [5, 14], we define reduction using an auxiliary structural equivalence, which we now explain. The structural equivalence $\equiv$ includes many standard rules and axioms. As usual, we presuppose that $\equiv$ is, in fact, an equivalence (reflexive, symmetric and transitive) and that it relates all terms that differ only in the names of bound identifiers. Also as usual, we suppose that $\equiv$ is preserved by composition and restriction (*i.e.* $P \equiv Q$ implies $P \mid R \equiv Q \mid R$ and $(\nu r)P \equiv (\nu r)Q$), and obeys the monoid axioms for composition: $P \equiv P \mid \mathsf{nil}$, $P \mid Q \equiv Q \mid P$ and $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.

The axioms specific to Dπ are given below:

(s-rep)	$\underline{\ell}[*P]$	$\equiv \underline{\ell}[P] \mid \underline{\ell}[*P]$	
(s-nil)	$\underline{\ell}[\mathsf{nil}]$	$\equiv \mathsf{nil}$	
(s-split)	$\underline{\ell}[P \mid Q]$	$\equiv \underline{\ell}[P] \mid \underline{\ell}[Q]$	
(s-chan)	$\underline{\ell}[(\nu a)P]$	$\equiv (\nu_\ell a)\underline{\ell}[P]$	if $\mathbf{newc} \in \mathrm{perm}(\underline{\ell})$
(s-loc)	$\underline{\ell}[(\nu m)P]$	$\equiv (\nu_\ell^{\uparrow} m)\underline{\ell}[P]$	if $\mathbf{subl} \in \mathrm{perm}(\underline{\ell})$ and $m \neq \ell$
(s-extr)	$Q \mid (\nu_\ell s)P$	$\equiv (\nu_\ell s)(Q \mid P)$	if $s \notin \mathrm{fn}(Q)$
(s-swap)	$(\nu_k r)(\nu_\ell s)P$	$\equiv (\nu_\ell s)(\nu_k r)P$	if $s \notin \{k, r\}$ and $r \notin \{\ell, s\}$

$$\text{(r-comm)}\quad \mathcal{L} \rhd \underline{\ell}\big[\textstyle\sum_i \underline{a_i}?\,(X_i{:}\zeta_i)\,P_i\big] \mid \underline{\ell}\big[\underline{b}!\,\langle V\rangle\,Q\big] \longrightarrow \mathcal{L} \rhd \underline{\ell}\big[P_i\{^{V'}/_{X_i}\}\big] \mid \underline{\ell}\big[Q\big] \quad \text{if } \mathcal{L} \vdash \uparrow\ell \text{ and } a_i = b \text{ and } \mathrm{refine}(V, \zeta_i) = V'$$

$$\begin{array}{rlll}
\text{(r-cond}_1\text{)} & \mathcal{L} \rhd \underline{\ell}\big[\text{if } \varphi \text{ then } P \text{ else } Q\big] & \longrightarrow \mathcal{L} \rhd \underline{\ell}\big[P\big] & \text{if } \mathcal{L} \vdash_{\ell} \uparrow\ell \wedge \varphi \\
\text{(r-cond}_2\text{)} & \mathcal{L} \rhd \underline{\ell}\big[\text{if } \varphi \text{ then } P \text{ else } Q\big] & \longrightarrow \mathcal{L} \rhd \underline{\ell}\big[Q\big] & \text{if } \mathcal{L} \vdash_{\ell} \uparrow\ell \wedge \neg\varphi \\
\text{(r-goto)} & \mathcal{L} \rhd \underline{\ell}\big[\underline{k} :: P\big] & \longrightarrow \mathcal{L} \rhd \underline{k}\big[P\big] & \text{if } \mathcal{L} \vdash \uparrow\ell \wedge \uparrow k \\
\text{(r-mig)} & \mathcal{L} \rhd \underline{\ell}\big[\mathsf{mig}\,\underline{k}.P\big] & \longrightarrow \mathcal{L}\{\!|\ell{\to}k|\!\} \rhd \underline{\ell}\big[P\big] & \text{if } \mathcal{L} \vdash \uparrow\ell \\
\text{(r-halt)} & \mathcal{L} \rhd \underline{\ell}\big[\mathsf{halt}\big] & \longrightarrow \mathcal{L}\{\!|{\downarrow}\ell|\!\} \rhd \mathsf{nil} & \text{if } \mathcal{L} \vdash \uparrow\ell
\end{array}$$

$$\text{(r-rstr)}\ \frac{\mathcal{L}, {}^{\delta}_{\ell} m \rhd P \longrightarrow \mathcal{L}, {}^{\delta'}_{\ell'} m \rhd P'}{\mathcal{L} \rhd (\nu^{\delta}_{\ell} m)P \longrightarrow \mathcal{L} \rhd (\nu^{\delta'}_{\ell'} m)P'} \qquad \frac{\mathcal{L} \rhd P \longrightarrow \mathcal{L}' \rhd P'}{\mathcal{L} \rhd (\nu_{\ell} a)P \longrightarrow \mathcal{L}' \rhd (\nu_{\ell} a)P'}$$

$$\text{(r-str)}\ \frac{\mathcal{L} \rhd P \longrightarrow \mathcal{L}' \rhd P'}{\mathcal{L} \rhd R \mid P \longrightarrow \mathcal{L}' \rhd R \mid P'} \qquad \frac{P \equiv Q \quad \mathcal{L} \rhd Q \longrightarrow \mathcal{L}' \rhd Q' \quad Q' \equiv P'}{\mathcal{L} \rhd P \longrightarrow \mathcal{L}' \rhd P'}$$

Table 2: Reduction relation

Of these, s-rep, s-extr and s-swap are standard axioms, merely adapted to our syntax. The rule s-nil allows for the garbage collection of threads, whereas s-split allows the thread $P \mid Q$ to split into two independent threads P and Q. The rule s-split provides a clear contrast between the treatment of locations in Dπ and the treatment of *ambients* in the ambient calculus [8]; in the ambient calculus, s-split does not hold. The rule s-chan states that $\underline{\ell}[(\nu a)P]$ is equivalent to $(\nu_{\ell} a)\underline{\ell}[P]$ as long as $\underline{\ell}$ contains the permission to create new channels. Note that when a channel declaration is "pulled out" of a thread its location is recorded. Rule s-loc states that the same is true for location declarations, although here we also record the fact that the new location is presumed to be alive (at least until a reduction occurs).

Reduction. We now briefly describe some of the rules in the reduction semantics. The goto rule

$$\mathcal{L} \rhd \underline{\ell}[\underline{k} :: P] \longrightarrow \mathcal{L} \rhd \underline{k}[P] \ \text{ if } \ \mathcal{L} \vdash \uparrow\ell$$

states that the thread $k :: P$, running at the live location ℓ, can move the thread P to location k. In contrast, the migration rule

$$\mathcal{L} \rhd \underline{\ell}[\mathsf{mig}\,\underline{k}.P] \longrightarrow \mathcal{L}\{\!|\ell{\to}k|\!\} \rhd \underline{\ell}[P] \ \text{ if } \ \mathcal{L} \vdash \uparrow\ell$$

does not change the location of P, but rather the *parent location* of the current location, ℓ. In this reduction, ℓ is moved to become a child of k (assuming that ℓ is alive), and subsequently the thread P and *any other threads running at* ℓ are executed under the new set of ancestors. Note that, unlike goto, the effect of a migration is *non-local*. The same is true of halt. It is for this reason that the effect of these operators is recorded in the runtime environment $\mathcal{L}$, rather than in the local process term.

In Cardelli and Gordon's terminology [8], migration is *subjective* — a local thread initiates ℓ's move to the new parent location — whereas goto is *objective* — a foreign thread "invades" location k. In addition, migration moves running code, whereas goto moves inactive code; and migration maintains location boundaries, whereas goto does not — when using goto, the exported thread merges into the threads of the new location. Movement operators which combine these attributes in other ways are also possible; some of these are discussed in [8].

$$\begin{array}{lll}
\mathrm{refine}(a_{\gamma}, \mathsf{chan}_{\gamma'}\zeta) = a_{\gamma'} & \text{if } \gamma \supseteq \gamma' \\
\mathrm{refine}(k_{\gamma}, \mathsf{loc}_{\gamma}) = k_{\gamma'} & \text{if } \gamma \supseteq \gamma' \\
\mathrm{refine}(u{::}U, \lambda{::}\widetilde{\kappa}) = v{::}V & \text{if } \mathrm{refine}(u,\ \lambda) = v \\
 & \quad \text{and } \mathrm{refine}(U,\ \widetilde{\kappa}) = V \\
\mathrm{refine}(\widetilde{U}, \widetilde{\zeta}) = \widetilde{V} & \text{if } \forall i\colon\ \mathrm{refine}(U^i, \zeta^i) = V^i
\end{array}$$

Table 3: The partial function "refine"

The rule r-comm is the most complicated of the rules in Section 2. Here, an abstraction $(X{:}\zeta)\,Q$ and a concretion $\langle V\rangle\,P$ are ready to synchronize at channel b of location ℓ. The permissions required by the abstraction are advertised by the reception type ζ, whereas the permissions offered by the concretion are manifest in the tagged value V. Obviously for a communication to occur, the arities of V and ζ must match, as in the polyadic π-calculus. In addition, the permissions offered by V must satisfy the requirements of ζ. For example, if the abstraction expects to receive a location with only the **halt** permission, but the concretion offers a value without this permission, then communication cannot occur (in fact there is a runtime error, as discussed in Section 3). Further, we wish to guarantee that the tagged value V' made available to the body of the abstraction includes only those permissions requested by ζ and does not include any extra permissions that happen to be available in V but aren't subject to negotiation. Continuing the above example, even if the concretion offers a instance of location m which happens to have the **mig** permission in addition to **halt**, the abstraction should only be able to exploit the **halt** permission, since no mention is made of **mig** in ζ.

We formalize these activities — extended arity checking and permission refinement — using the partial function "refine". If the permissions offered by V satisfy the requirements of ζ, then refine(V,ζ) will be defined. In addition, the value V' returned by refine(V,ζ) is *refined* in the sense that the permissions on the tags in V are reduced to match those declared in ζ. In r-comm it is the refined value V' that is substituted into the body of the abstraction. The definition of "refine" is given in Section 3. For example:

$$\begin{array}{lll} \text{refine}((k_\varnothing, m_\varnothing), & \mathsf{loc}_\varnothing) & \text{is undefined} \\ \text{refine}(k_{\{\mathsf{run}\}}, & \mathsf{loc}_{\{\mathsf{run},\mathsf{halt}\}}) & \text{is undefined} \\ \text{refine}(k_{\{\mathsf{run},\mathsf{halt},\mathsf{mig}\}}, & \mathsf{loc}_{\{\mathsf{run},\mathsf{halt}\}}) & = k_{\{\mathsf{run},\mathsf{halt}\}} \end{array}$$

Note that if refine$(\zeta, V) = V'$, then V and V' are the same when tags are ignored.

The use of tags and tag refinement is the major difference between our reduction semantics and that of [16]. Here when values are communicated the associated permissions are explicitly communicated as well, appropriately refined by the type of the channel used for the communication, while in [16] only the names but not the permissions are communicated.

The rules for the other constructs of the language are straightforward although it is worth noting how the rules for restriction are used. For example, using the structural equivalence and the migration rule, r-rstr can be used to derive:

$$\mathcal{L} \triangleright (\nu_\ell m)\ m[(\mathsf{mig}\, k.\mathsf{nil}) \mid Q] \longrightarrow \mathcal{L} \triangleright (\nu_k m)\ m[Q]$$

Here the private sublocation m of ℓ has been moved from ℓ to k; one effect of this is to move the thread Q from ℓ to k as well.

But for the use of "refine", our reduction semantics takes no account of the type of identifiers or the tags on names. In fact, these rules allow reductions which should be forbidden by the explicit permissions on names. We address this issue with a typing system and a notion of runtime error in Section 3.

This ends our description of the reduction semantics. We should point out that, independently of the explicit use of locations, the presence of permissions in types makes the language considerably more complicated than the π-calculus. Consider the process located at k

$$k[(\nu\ell)\ (\ell :: (\nu a)(\nu b)P)]$$

which can reduce to:

$$(\nu_k^{\uparrow}\ell)(\nu_\ell a)(\nu_\ell b)\ \ell[P]$$

This process has a private location ℓ with two private channels a and b at ℓ, with their capabilities specified by the allocation types ζ_ℓ, ζ_a and ζ_b respectively. As in the π-calculus the thread P may make these private resources known to other threads via communication. Here, however, P need not communicate all of ℓ's resources at once. For example, if P contains a component such as

$$k :: \left(u!\,\langle \ell :: a\rangle\, w!\,\langle \ell :: b\rangle\, \mathsf{nil}\right)$$

then the receptor on channel u will gain knowledge only of a, whereas the receptor on w will gain knowledge only of b. More importantly, the permissions received with these names are determined by the channels u and w, and in general (in a well-typed system) the reception types will be more restrictive than the allocation types. Thus as the system evolves individual components will gain different views of the capabilities associated with channels and locations.

2.3 Examples

We now present a series of examples based on a simple read/write cell. The examples use recursive definitions of the form $A \Longleftarrow P$, where P is a basic process. It is well-known that such recursive definitions can be implemented using the replication operator $*P$ (see for example [14]). First consider the following definition of a "cell" $C(v)$.

$$C(v) \Longleftarrow \mathsf{p}?\,(x)\,C(x)\ +\ \mathsf{g}?\,(y :: z)\,\left(C(v) \mid y :: z!\,\langle v\rangle\,\mathsf{nil}\right)$$

$C(v)$ contains two channels: p for "putting" data into the cell and g for "getting" data out. To read the value of the cell, a user must send the name of a continuation channel on g, along with the location of that channel. For example, a "user" at k can be defined:

$$U_1 \Longleftarrow (\nu i)\ \ i?\,(y)\,U_1'(y)\ \mid\ \ell :: \mathsf{g}!\,\langle k :: i\rangle\,\mathsf{nil}$$

Then the system $\ell[C(v)] \mid k[U_1]$ can reduce, via code movement, to

$$\ell[C(v)]\ \mid\ \ell[\mathsf{g}!\,\langle k :: i\rangle\,\mathsf{nil}]\ \mid\ k[i?\,(y)\,U_1'(y)]$$

then, via local communication at ℓ, to

$$\ell[C(v)]\ \mid\ \ell[k :: i!\,\langle v\rangle\,\mathsf{nil}]\ \mid\ k[i?\,(y)\,U_1'(y)]$$

then, via code movement to

$$\ell[C(v)]\ \mid\ k[i!\,\langle v\rangle\,\mathsf{nil}]\ \mid\ k[i?\,(y)\,U_1'(y)]$$

and finally, via local communication at k, to:

$$\ell[C(v)]\ \mid\ k[U_1'(v)]$$

In this final configuration, the cell has returned to its initial state and the user has obtained the cell's current contents v. In this example, the asynchronous output of the π and join calculi is mimicked by a particularly simple form of code movement followed by local communication; indeed in further examples we will use the abbreviation "$k.i!\,\langle v\rangle$" for the thread "$k :: i!\,\langle v\rangle\,\mathsf{nil}$."

As a variation, we now define a "cell server" which generates "new cells" on request from a client. In the following

example, the server creates the cell at a new location and then informs the client of its whereabouts.

$$CS_2 \Longleftarrow \mathsf{req?}\,(x{::}y)\,\big(NC_2 \,|\, CS_2\big)$$
$$NC_2 \Longleftarrow (\nu m)\, m :: (\nu \mathsf{p},\mathsf{g})\ \ (C_{\mathsf{p},\mathsf{g}} \mid x.y!\,\langle m{::}(\mathsf{p},\mathsf{g})\rangle)$$

where $C_{\mathsf{p},\mathsf{g}}$ is the code for a cell, with some appropriate initial value.

A user which requests a new cell from the cell server located at ℓ might take the form:

$$U_2 \Longleftarrow \ell.\mathsf{req}!\,\langle k{::}i\rangle \mid i?\,(x{::}(y,z))\,U_2'(x,y,z)$$

Then the system $\ell[CS_2] \,|\, k[U_2]$ can evolve to:

$$\ell[CS_2] \ \mid\ (\nu_\ell m)(\nu_m \mathsf{p},\mathsf{g})\big(m[C_{\mathsf{p},\mathsf{g}}] \mid k[U_2'(m,\mathsf{p},\mathsf{g})]\big)$$

In this configuration, there is a new cell running at location m, and the user has knowledge of this location together with its methods p and g. Note that here the new cell is running *at the server*; *i.e.* the cell location m is a sublocation of the server location ℓ.

An alternative is to define the server so that it generates the new cell at a sublocation of the client, or more generally at a location determined by the client.

$$CS_3 \Longleftarrow \mathsf{req?}\,(x{::}i)\,\big(NC_3 \,|\, CS_3\big)$$
$$NC_3 \Longleftarrow x :: (\nu \mathsf{p},\mathsf{g})\big(C_{\mathsf{p},\mathsf{g}} \,|\, i!\,\langle \mathsf{p},\mathsf{g}\rangle\big)$$

Here the server receives a location x together with a channel i at that location. It first goes to the received location x where it starts a new cell and, in parallel, sends the names of the cell's methods on the local channel i.

To use such a server, a user might generate a new sublocation with an associated channel i for receiving information; then send these names to the server; and finally start a process at the new location to receive the names of the cell methods. Such a user is defined as follows:

$$U_3 \Longleftarrow (\nu m) m :: \big((\nu i)\ \ell.\mathsf{req}!\,\langle m(i)\rangle \mid i?\,(x,y)\,U_3'(x,y)\big)$$

Now the system $\ell[CS_3] \mid k[U_3]$ evolves to a configuration in which a new cell is generated at the client site k:

$$\ell[CS_3] \ \mid\ (\nu_k m)(\nu_m \mathsf{p},\mathsf{g})\big(m[C_{\mathsf{p},\mathsf{g}}] \mid m[U_3'(\mathsf{p},\mathsf{g})]\big),$$

We should point out a similar cell generator could also be defined using the migrate primitive:

$$CS_4 \Longleftarrow \mathsf{req?}\,(x{::}i)\,(NC_4 \,|\, CS_4)$$
$$NC_4 \Longleftarrow (\nu_\ell m)(\nu_m \mathsf{p},\mathsf{g}) m :: \big(C_{\mathsf{p},\mathsf{g}} \,|\, \mathsf{mig}\,x.x.i!\,\langle m{::}(\mathsf{p},\mathsf{g})\rangle\big)$$

On receiving a request first the generator creates a new local sublocation m and starts a new cell there. This sublocation then migrates to the client's location and informs the client (*e.g.* U_2) of the new method names.

These cell generators are open to various forms of intentional and non-intentional misuse by clients. For example the sublocations of the new cells may be killed by arbitrary clients, or the methods might be interfered with: a client may start new threads at the sublocation m that intercepts data received on these channels. This undesirable activity can be constrained by restricting the capabilities passed to users. For example if i is defined to communicate values of type

$$\mathsf{loc}_{\{\mathbf{run}\}} :: (\mathsf{chan}_{\{\mathbf{snd}\}}\zeta_p, \mathsf{chan}_{\{\mathbf{snd}\}}\zeta_g)$$

then any user of the cell will only be able to use the channels p and g to *send* values. Additional constraints can be specified in the types ζ_p and ζ_g.

3 The Typing System

Judgments of the type system for basic processes have the form

$$\Delta \vdash_w P{:}\mathsf{proc}$$

which may be read: "in the type environment Δ the process P is properly typed to run at location w." For located processes, the script w is dropped.

Type environments. In the example judgment above, the *type environment* Δ records the type and location of the free identifiers in P. We represent type environments (Δ, Γ) as partial maps in $Id \rightharpoonup Type \times Id$ and adopt several related notations. Data stored in a type environment is retrieved using the projection functions "type_Δ" and "loc_Δ" which return the type and location of an identifier, respectively. After allocation, the location of an identifier is only significant for channel types. For channel identifiers u, $\mathrm{loc}_\Delta(u)$ returns the location at which u was allocated.

The *update* function is written postfix as $\Delta, {}_w u{:}\zeta$, which denotes the environment obtained by adding the identifier u to Δ with type ζ at location w. To be well defined, w must be a location already defined in Δ and u must be fresh. In fact, we make a stronger requirement: for locations, "$\Delta, {}_w u{:}\lambda$" is defined only if

$$\mathbf{subl} \in \mathrm{perm}(\mathrm{type}_\Delta(w))$$

and, similarly for channels, "$\Delta, {}_w u{:}\kappa$" is defined only if

$$\mathbf{newc} \in \mathrm{perm}(\mathrm{type}_\Delta(w))$$

For example, assuming that w has the **subl** and **newc** permissions in Δ, we have the following:

$$\mathrm{type}_{\Delta,{}_w u:\kappa}(u) = \kappa \qquad \mathrm{loc}_{\Delta,{}_w u:\kappa}(u) = w$$
$$\mathrm{type}_{\Delta,{}_w u:\lambda}(u) = \lambda \qquad \mathrm{loc}_{\Delta,{}_w u:\lambda}(u) = u$$

The update function is generalized, structurally, to values. Thus, we write $\Delta, {}_w U{:}\zeta$ for the extension of Δ with the identifiers in U at type ζ and location w. For example:

$$\Delta, {}_w(m,a){:}(\lambda,\kappa) = \Delta, {}_w m{:}\lambda, {}_w a{:}\kappa$$
$$\Delta, {}_w(m{::}a){:}(\lambda{::}\kappa) = \Delta, {}_w m{:}\lambda, {}_m a{:}\kappa$$

$$(V1)\ \frac{\mathrm{type}_\Delta(x) \le \kappa}{\Delta \vdash_w x{:}\kappa}\ \mathrm{loc}_\Delta(x) = w \qquad (L1)\ \frac{\Delta \vdash P, Q}{\Delta \vdash \mathsf{nil} \quad \Delta \vdash P \,|\, Q} \qquad (B1)\ \frac{\Delta \vdash_w P, Q}{\Delta \vdash_w \mathsf{nil} \quad \Delta \vdash_w *P \quad \Delta \vdash_w P \,|\, Q}$$

$$(V2)\ \frac{\mathrm{type}_\Delta(a) \le \kappa}{\Delta \vdash_w \underline{a}{:}\kappa}\ \mathrm{loc}_\Delta(a) = w \qquad (L2)\ \frac{\Delta, {}_\ell a{:}\kappa \vdash P}{\Delta \vdash (\nu_\ell a{:}\kappa)P} \qquad (B2)\ \frac{\Delta, {}_w a{:}\kappa \vdash_w P}{\Delta \vdash_w (\nu a{:}\kappa)P}$$

$$(V3)\ \frac{\mathrm{type}_\Delta(x) \le \lambda}{\Delta \vdash_w x{:}\lambda \quad \Delta \vdash x{:}\lambda} \qquad (L3)\ \frac{\Delta, {}_\ell m{:}\lambda \vdash P}{\Delta \vdash (\nu_\ell m{:}\lambda)P} \qquad (B3)\ \frac{\Delta, {}_w m{:}\lambda \vdash_w P}{\Delta \vdash_w (\nu m{:}\lambda)P}$$

$$(V4)\ \frac{\mathrm{type}_\Delta(\ell) \le \lambda}{\Delta \vdash_w \underline{\ell}{:}\lambda \quad \Delta \vdash \underline{\ell}{:}\lambda} \qquad (L4)\ \frac{\Delta \vdash \underline{\ell}{:}\mathsf{loc}_{\{\mathbf{run}\}} \qquad \Delta \vdash_\ell P}{\Delta \vdash \underline{\ell}[P]} \qquad (B4)\ \frac{\Delta \vdash_w u{:}\mathsf{loc}_{\{\mathbf{run}\}} \quad \Delta \vdash_u P}{\Delta \vdash_w u :: P}$$

$$(V5)\ \frac{\forall i{:}\ \ \Delta \vdash_w V^i{:}\zeta^i}{\Delta \vdash_w \widetilde{V}{:}\widetilde{\zeta}} \qquad (B5)\ \frac{\Delta \vdash_w w{:}\mathsf{loc}_{\{\mathbf{mig}\}}, u{:}\mathsf{loc}_{\{\mathbf{subl}\}}, P}{\Delta \vdash_w \mathsf{mig}\, u.P}$$

$$(V6)\ \frac{\Delta \vdash_w u{:}\lambda \quad \Delta \vdash_u V{:}\widetilde{\kappa}}{\Delta \vdash_w (u{::}V){:}(\lambda{::}\widetilde{\kappa})} \qquad (P1)\ \frac{\Delta \vdash_w u{:}\zeta, v{:}\zeta}{\Delta \vdash_w u = v{:}\mathsf{bool}} \qquad (B6)\ \frac{\Delta \vdash_w w{:}\mathsf{loc}_{\{\mathbf{halt}\}}}{\Delta \vdash_w \mathsf{halt}}$$

$$(P2)\ \frac{\Delta \vdash_w u{:}\lambda}{\Delta \vdash_w \uparrow u{:}\mathsf{bool} \quad \Delta \vdash_w \to u{:}\mathsf{bool} \quad \Delta \vdash_w \twoheadrightarrow u{:}\mathsf{bool}} \qquad (B7)\ \frac{\Delta \vdash_w \varphi{:}\mathsf{bool}, P, Q}{\Delta \vdash_w \mathsf{if}\ \varphi\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$(B8)\ \frac{\forall i{:}\ \ \Delta \vdash_w u_i{:}\mathsf{chan}_{\{\mathbf{rcv}\}}\zeta_i \quad \Delta, {}_w X_i{:}\zeta_i \vdash_w P_i}{\Delta \vdash_w \sum_i u_i?\,(X_i{:}\zeta_i)\,P_i} \qquad (B9)\ \frac{\Delta \vdash_w u{:}\mathsf{chan}_{\{\mathbf{snd}\}}\zeta, V{:}\zeta, Q}{\Delta \vdash_w u!\,\langle V\rangle\, Q}$$

Table 5: Typing relation for values (V), located processes (L), predicates (P), and basic processes (B)

$$\begin{array}{rll}
\mathsf{chan}_{\gamma'}\zeta' \le \mathsf{chan}_\gamma\zeta & \text{if } \gamma' \supseteq \gamma \text{ and if } \mathbf{rcv} \in \gamma \text{ then } \zeta' \le \zeta & \\
& \text{if } \mathbf{snd} \in \gamma \text{ then } \zeta \le \zeta' & \\
\mathsf{loc}_{\gamma'} \le \mathsf{loc}_\gamma & \text{if } \gamma' \supseteq \gamma & \\
\lambda'{::}\zeta' \le \lambda{::}\zeta & \text{if } \lambda' \le \lambda \text{ and } \zeta' \le \zeta & \\
(\xi^1 \ldots \xi^n) \le (\zeta^1 \ldots \zeta^n) & \text{if } \xi^i \le \zeta^i, 1 \le i \le n &
\end{array}$$

Table 4: The subtyping relation

Subtyping. The typing system is built up using the *subtype relation* defined in Section 4, which adapts the subtyping relation of [16] to our type system. Intuitively $\xi \le \zeta$ indicates that ξ is *less restrictive* than ζ, in the sense that whenever a context is well formed under the assumption that r has type ζ, then it is also well formed assuming that r has the more general type ξ. Said another way, every value of type ξ is also a value of type ζ.

For example, if a context is well formed assuming that ℓ has the permission **run**, then clearly it is also well formed under the assumption that ℓ has the permissions **run** and **mig**. Therefore $\mathsf{loc}_{\{\mathbf{run},\mathbf{mig}\}}$ is considered *more permissive* than $\mathsf{loc}_{\{\mathbf{run}\}}$; that is:

$$\mathsf{loc}_{\{\mathbf{run},\mathbf{mig}\}} \le \mathsf{loc}_{\{\mathbf{run}\}}$$

Note that more permissive types are *lower* in the ordering.

In the case of channel types $\mathsf{chan}_\gamma\zeta$, the parameter ζ constitutes a contract between sender and receiver. A receiver may use the received value with *at most* the capabilities specified by ζ. More permissive types allow the receiver to assume that the data has more capabilities. For example:

$$\mathsf{chan}_{\{\mathbf{rcv}\}}(\mathsf{loc}_{\{\mathbf{run},\mathbf{mig}\}}) \le \mathsf{chan}_{\{\mathbf{rcv}\}}(\mathsf{loc}_{\{\mathbf{run}\}})$$

On the other hand, a sender is obliged to send values that have *at least* the capabilities specified by ζ. More permissive types allow the sender to send data with fewer capabilities. Thus:

$$\mathsf{chan}_{\{\mathbf{snd}\}}(\mathsf{loc}_{\{\mathbf{run}\}}) \le \mathsf{chan}_{\{\mathbf{snd}\}}(\mathsf{loc}_{\{\mathbf{run},\mathbf{mig}\}})$$

In short, input (**rcv**) is covariant and output (**snd**) is contravariant. For further discussion, see [16].

Typing. The judgments of the typing system are given in Section 5, where we abbreviate the statement "$\Delta \vdash_w P{:}\mathsf{proc}$" to "$\Delta \vdash_w P$" and "$\Delta \vdash P{:}\mathsf{proc}$" to "$\Delta \vdash P$". In the table two auxiliary typing judgments are also given: for values and predicates. The typing system is defined on explicitly *tagged* terms, although it ignores tags entirely. Tags are included so that runtime error and type safety can be defined below.

Many of the rules are adapted from those of [16], although the style of presentation is somewhat different. We make heavy use of judgments concerning identifiers. The

judgment $\Delta \vdash_w u{:}\kappa$ should be read "in Δ, u is a channel identifier at location w with *at least* the permissions declared in κ." Similarly, both $\Delta \vdash_w u{:}\lambda$ and $\Delta \vdash u{:}\lambda$ should be read "in Δ, u is a location identifier with at least the permissions declared in λ."

As an example of the use of the type rules, consider the judgment $\Delta \vdash_w u?(X{:}\zeta)P$. To infer that $u?(X{:}\zeta)P$ is well-typed to run at w it must be that:

- the identifier u is a channel at w which has at least the permission **rcv** and a transmission type at least as permissive as ζ; and
- the continuation P is typable given the additional assumption that the values to be input have the type ζ.

So in order to infer that $a?(x{:}\kappa)P$ is well-typed at w, it must be that P is typable using the extra assumption that x is a channel (of type κ) located at w. Similarly, to infer that $a?((z{::}x){:}(\lambda{::}\kappa))P$ is well-typed, it must be that P is typable under the extra assumptions that z is a location (of type λ) and x a channel (of type κ) located at z.

It is worth pointing out a difference between rules (B4) and (B5) for thread movement and migration, respectively. To type $\mathsf{mig}\,u.P$ at w it must be that P is well-typed at the *same* location, w. On the other hand, to type $u :: P$ at w it must be that P is well-typed to at the *new* location u.

There is also a subtlety concerning the matching of channel names. In well-typed terms channels can only be compared at their home location. Let us write $[a=b]P$ as an abbreviation for the matching term if $a=b$ then P else nil. This term can only be typed in an environment in which $\mathrm{loc}_\Delta(a) = \mathrm{loc}_\Delta(b)$. This means that

$$a?(x{::}y)\ b?(x'{::}y')\ [y=y']P$$

cannot be typed. However the following term is typable, assuming that P can also be typed in the appropriate environment:

$$a?(x{::}(y,y'))\ x :: [y=y']P$$

The typing system satisfies a number of standard properties which are collected below. First we lift the subtyping relation to type environments. We say that Γ is *refined by* Δ ($\Gamma \le \Delta$) if for every u in $\mathrm{dom}(\Delta)$:

$$\mathrm{type}_\Gamma(u) \le \mathrm{type}_\Delta(u) \ \text{ and } \ \mathrm{loc}_\Gamma(u) = \mathrm{loc}_\Delta(u)$$

Lemma 3.1 (Weakening, narrowing).
(a) *If* $\Delta \vdash P$ *then* $\Delta, {}_w u{:}\zeta \vdash P$.
(b) *If* $\Delta, {}_w u{:}\zeta \vdash P$ *and* $\xi \le \zeta$ *then* $\Delta, {}_w u{:}\xi \vdash P$.
(c) *If* $\Delta \vdash P$ *and* $\Gamma \le \Delta$ *then* $\Gamma \vdash P$.

Proof. (c) is immediate from (a) and (b). (a) and (b) follow by induction on the type relation, relying on similar results for basic processes. □

$$
\begin{array}{rl}
\text{(e-run)} & \underline{\ell}[P] \xrightarrow{err} \text{ if } \mathbf{run} \notin \mathrm{perm}(\underline{\ell}) \\
\text{(e-rcv)} & (\nu_k a_i)\, \underline{\ell}[\sum_i \underline{a}_i?(X_i)P_i] \xrightarrow{err} \\
 & \qquad \text{if } k \neq \ell \text{ or } \mathbf{rcv} \notin \mathrm{perm}(\underline{a}) \\
\text{(e-snd)} & (\nu_k a)\, \underline{\ell}[\underline{a}!\langle V\rangle Q] \xrightarrow{err} \text{ if } k \neq \ell \text{ or } \mathbf{snd} \notin \mathrm{perm}(\underline{a}) \\
\text{(e-comm)} & \underline{\ell}[\sum_i \underline{a}_i?(X_i{:}\zeta_i)P_i] \mid \underline{\underline{\ell}}[\underline{b}!\langle V\rangle Q] \xrightarrow{err} \\
 & \qquad \text{if } \exists i:\ a_i = b:\ \mathrm{refine}(V,\zeta_i) \text{ undefined} \\
\text{(e-cond}_1\text{)} & (\nu_k a)\, \underline{\ell}[\text{if } \underline{a}=\underline{b} \text{ then } P \text{ else } Q] \xrightarrow{err} \text{ if } k \neq \ell \\
\text{(e-cond}_2\text{)} & (\nu_k b)\, \underline{\ell}[\text{if } \underline{a}=\underline{b} \text{ then } P \text{ else } Q] \xrightarrow{err} \text{ if } k \neq \ell \\
\text{(e-goto)} & \underline{\ell}[\underline{k} :: P] \xrightarrow{err} \text{ if } \mathbf{run} \notin \mathrm{perm}(\underline{k}) \\
\text{(e-mig)} & \underline{\ell}[\mathsf{mig}\,\underline{k}.P] \xrightarrow{err} \text{ if } \mathbf{mig} \notin \mathrm{perm}(\underline{\ell}) \\
 & \qquad \text{or } \mathbf{subl} \notin \mathrm{perm}(\underline{k}) \\
\text{(e-halt)} & \underline{\ell}[\mathsf{halt}] \xrightarrow{err} \text{ if } \mathbf{halt} \notin \mathrm{perm}(\underline{\ell}) \\
\text{(e-loc)} & \underline{\ell}[(\nu m)P] \xrightarrow{err} \text{ if } \mathbf{subl} \notin \mathrm{perm}(\underline{\ell}) \\
\text{(e-chan)} & \underline{\ell}[(\nu a)P] \xrightarrow{err} \text{ if } \mathbf{newc} \notin \mathrm{perm}(\underline{\ell}) \\
\text{(e-str)} & \dfrac{P \xrightarrow{err}}{(\nu r)P \xrightarrow{err} \quad P \mid R \xrightarrow{err}} \qquad \dfrac{P \equiv Q \quad Q \xrightarrow{err}}{P \xrightarrow{err}}
\end{array}
$$

Table 6: The error predicate

Theorem 3.2 (Subject Reduction).
(a) *If* $P \equiv P'$ *then* $\Delta \vdash P$ *if and only if* $\Delta \vdash P'$.
(b) *If* $\mathcal{L} \triangleright P \longrightarrow \mathcal{L}' \triangleright P'$ *then* $\Delta \vdash P$ *implies* $\Delta \vdash P'$.

Proof. By induction on the definitions of structural equivalence and reduction, respectively. The only non-trivial case is the reduction rule for communication. □

Runtime errors. We now turn our attention to explaining in what way our typing system excludes the possibility of runtime error. Informally, a process produces a runtime error if:

- it attempts to send a value on a channel that violates the channel's type, or
- it attempts to perform an action without having the necessary permissions.

In Section 6, we formalize this intuition by defining an error predicate on terms (which we write postfix as $P \xrightarrow{err}$): if $P \xrightarrow{err}$ then P may immediately produce a runtime error. The definition is long but straightforward. For example the process $\underline{\ell}[\underline{k} :: p]$ produces a runtime error if the term lacks permission either to run at ℓ (**run** $\notin \mathrm{perm}(\underline{\ell})$) or to run at k (**run** $\notin \mathrm{perm}(\underline{k})$).

The most complicated case is for a potential communication:

$$R = \underline{\ell}[\underline{a}?(X{:}\zeta)P] \mid \underline{\underline{\ell}}[\underline{a}!\langle V\rangle Q]$$

Note that here, according to our conventions, $\mathrm{name}(\underline{r}) = \mathrm{name}(\underline{\underline{r}}) = r$, so both the abstraction and the concretion must be at the same location and channel in order for communication (or error) to occur. This term R produces a runtime error under any of the following conditions:

- $\underline{a}$ lacks **rcv** permission,
- $\underline{a}$ lacks the **snd** permission,
- either $\underline{\ell}$ or $\underline{\underline{\ell}}$ lacks the **run** permission, or
- the value V is incompatible with the received type ζ, *i.e.* refine(V,ζ) is undefined.

R will also produce an error if it is placed within a context that allocates a at a location k different from ℓ.

Type safety. In general it is not reasonable to expect that well-typed terms are free of runtime errors for the simple reason that, by design, the typing system *ignores tags*, which, instead, are the basis for the definition of runtime error. For example, if **halt** is not in the permissions of $\underline{\ell}$, then $\underline{\ell}[\mathsf{halt}]$ will generate a runtime error, even though the term can be typed by any Δ that provides $\underline{\ell}$ with **run** and **halt** permissions. The problem is that the permissions decorating $\underline{\ell}$ need not be *consistent* with the type environment. The problem is resolved by adding side conditions to the rules for names in Section 5. These rules become:

$$(V2)\ \frac{\mathrm{type}_\Delta(a) \le \kappa \quad \mathrm{loc}_\Delta(a) = w}{\Delta \vdash_w \underline{a}{:}\kappa \quad \mathrm{perm}(\underline{a}) \supseteq \mathrm{perm}(\kappa)}$$

$$(V4)\ \frac{\mathrm{type}_\Delta(\ell) \le \lambda}{\Delta \vdash_w \underline{\ell}{:}\lambda \quad \Delta \vdash \underline{\ell}{:}\lambda}\ \mathrm{perm}(\underline{\ell}) \supseteq \mathrm{perm}(\lambda)$$

We write $\Delta \Vdash P$ to indicate that $\Delta \vdash P$ can be derived in this slightly more exacting typing system, where the tags on identifiers are examined to ensure that they are consistent with their intended use, as indicated by their derived types. We have the following:

Theorem 3.3 (Type Safety).
(a) *If* $\mathcal{L} \triangleright P \longrightarrow \mathcal{L}' \triangleright P'$ *then* $\Delta \Vdash P$ *implies* $\Delta \Vdash P'$.
(b) $\Delta \Vdash P$ *implies* $\neg(P \xrightarrow{err})$.

Proof. The proof of (a) is readily adapted from the proof of Theorem 3.2. (b) is proved contrapositively, by induction on the definition of errors, relying on the fact that "untypability" is preserved by $\equiv$, composition and restriction. □

In light of the Type Safety Theorem, we are justified in dropping tags from well-typed terms; in particular the reduction relation, given in Section 2, can safely be interpreted on untagged processes. If $\Delta \vdash P$ is a closed term, we can generate an error-free tagged term simply by decorating every occurrence of a name r in P with the permissions found in $\mathrm{type}_\Delta(r)$ (of course, Δ must be augmented when passing through a restriction). Note, however, that this translation is not preserved by reduction: the permissions associated with an instance of a name are *refined* when the name is communicated.

4 The Semantic Theory

In this section we show how a semantic theory can be developed for Dπ, using the ideas of *barbed congruence* from

$$\begin{aligned}
\mathcal{L} \triangleright \ell[\textstyle\sum_i a_i?(X_i).Q] \downarrow_{a_i?} &\quad \text{if } \mathcal{L} \vdash \uparrow\ell \\
\mathcal{L} \triangleright \ell[a!\langle V\rangle.P] \downarrow_{a!} &\quad \text{if } \mathcal{L} \vdash \uparrow\ell \\
\mathcal{L} \triangleright (\nu_\ell a)P \downarrow_\beta &\quad \text{if } \mathcal{L} \triangleright P \downarrow_\beta \text{ and } a \notin \mathrm{n}(\beta) \\
\mathcal{L} \triangleright (\nu_\ell m)P \downarrow_\beta &\quad \text{if } \mathcal{L} \triangleright P \downarrow_\beta \\
\mathcal{L} \triangleright P \mid Q \downarrow_\beta &\quad \text{if } \mathcal{L} \triangleright P \downarrow_\beta \\
\mathcal{L} \triangleright P \downarrow_\beta &\quad \text{if } P \equiv Q \text{ and } \mathcal{L} \triangleright Q \downarrow_\beta
\end{aligned}$$

Table 7: The commitment predicates

[20]. The basic approach is to say that two processes P and Q are semantically equivalent if in every *appropriate* context $\mathbb{C}$, $\mathbb{C}[P] \dot{\approx} \mathbb{C}[Q]$, where $\dot{\approx}$ is a simple behavioral equivalence based on some notion of *observation*. For this simple equivalence $\dot{\approx}$ we adapt the definition of *barbed bisimulation*; we are then left with the question of what are appropriate contexts in this typed language.

We first adapt the definition of barbed bisimulation [20] to Dπ. At this point we ignore entirely the tags on names, instead working with closed, well-typed terms. Throughout this section, let Δ and Γ range over *closed* type environments (*i.e.* environments whose domain contains no variables). For convenience, we extend our type system to process configurations using the rule

$$\frac{\Delta \vdash P \quad \mathrm{locs}(\mathcal{L}) \subseteq \mathrm{locs}(\Delta)}{\Delta \vdash \mathcal{L} \triangleright P}$$

and define:

$$\begin{aligned}
\mathit{PConfig} &\stackrel{def}{=} \{M \mid \exists \Delta{:}\ \Delta \vdash M\} \\
\mathit{PConfig}(\Delta) &\stackrel{def}{=} \{M \mid \Delta \vdash M\}
\end{aligned}$$

The (strong) commitment predicates, defined over configurations in Section 7, determine the ability of a located process to immediately communicate on a specific channel. We use β to range over the set of *commitments*, $\{a!, a? \mid a \in \mathit{Chan}\}$. The commitment $a?$ indicates that a process is willing to accept data on channel a, whereas the commitment $a!$ indicates that it is willing to offer data on channel a. The strong commitment predicates are generalized to *weak* predicates in the standard manner: let $\Longrightarrow$ denote the reflexive transitive closure of $\longrightarrow$, and let $M \Downarrow_\beta$ if $M \Longrightarrow M'$ and $M' \downarrow_\beta$.

Definition 4.1 (Barbed bisimilarity). For each Δ, let $\dot{\approx}_\Delta$ be the largest symmetric relation over $\mathit{PConfig}(\Delta) \times \mathit{PConfig}(\Delta)$ such that whenever $M \dot{\approx}_\Delta N$:

(a) $\forall \beta{:}\quad M \Downarrow_\beta$ implies $N \Downarrow_\beta$, and
(b) $\forall M'{:}\ M \longrightarrow M'$ implies $\exists N'{:}\ N \Longrightarrow N'$ and $M' \dot{\approx}_\Delta N'$.

We say that configurations M and N are *barbed bisimilar at* Δ if $M \dot{\approx}_\Delta N$. □

We now define the related contextual congruence. Intuitively we wish to say that M is equivalent to N at Δ if M

and N are in $PConfig(\Delta)$ and for every *appropriate* context $\mathbb{C}$, $\mathbb{C}[M] \dot{\approx}_\Delta \mathbb{C}[N]$. These contexts are intended to provide *testing* scenarios for the terms M and N [10]; therefore it is sufficient to restrict our attention to contexts in which a located process R (the experimenter, or observer) is run in parallel with M and N. Thus, *barbed equivalence* at Δ, ($\approx_\Delta$) is derived from barbed bisimilarity by quantifying over a restricted set of contexts:

Definition 4.2 (Barbed equivalence). $(\mathcal{L} \triangleright P) \approx_\Delta (\mathcal{K} \triangleright Q)$ if $\forall R\colon\ \Delta \vdash R\colon\ (\mathcal{L} \triangleright P \,|\, R) \dot{\approx}_\Delta (\mathcal{K} \triangleright Q \,|\, R)$ □

Note that while the *free names* in R are constrained by Δ, this property is not preserved by reduction: R may export an arbitrary number of private names into P and Q, effectively making these names free in the continuations of R.

Barbed equivalence provides a primitive proof technique for reasoning about processes. Indeed, substantial theorems can be established this way [16, 1]. Proofs using the definition of barbed congruence directly, however, are hard work due to the quantification over all possible observers. It is useful, therefore, to find alternative characterizations of the equivalence which do not involve universal quantification over observers. Such alternative characterizations, in the form of (labelled) bisimulation relations, have been given, for example, for the synchronous and asynchronous π-calculi [20, 4] and for distributed CCS [18]. In the full paper, we present such an alternative characterization of barbed equivalence for Dπ, for image-finite processes. Space does not permit us to present the full definition here, rather we discuss some of the issues involved in developing the *labelled transition system* (LTS) which is the basis of the alternative characterization.

In constructing a labelled transition relation for the ordinary π-calculus [15, 20, 4], one must be careful to distinguish the communication of a *free name* (which a testing context may already know about) from the communication of a *bound name* (which is guaranteed to be fresh for any testing context). In the π-calculus only the *possession* of a name is important: either a tester has a name or it doesn't (*i.e.* either the name is free or it's not).

In Dπ the story is more complex. A testing context may have a channel, for example, without having the permission to communicate on it; or it may have a location, without having permission to kill it. To see the effect that this will have on the labelled transition relation, consider the process

$$P = \ell[(\nu a)(\nu m)\, c!\langle a, m\rangle\, (Q \,|\, a?(z)\,\mathsf{nil})]$$

where c has the type $\kappa_c = \mathsf{chan}_{\{\mathbf{rcv},\mathbf{snd}\}}(\kappa_a, \lambda_m)$ and $\kappa_a = \mathsf{chan}_{\{\mathbf{rcv},\mathbf{snd}\}}(\lambda_a)$. Using the ordinary sorting rules of the π-calculus, a process that receives the value (a,m) is immediately able to send the value m on channel a. Thus in the LTS, one expects the transitions:

$$P \xrightarrow{(\nu_\ell a)(\nu_\ell m) c!(a,m)} \ell[Q] \mid \ell[a?(z)\,\mathsf{nil}] \xrightarrow{a?(m)} \ell[Q]$$

In Dπ, the permission capabilities λ_m and λ_a (given in the types κ_c and κ_a, above) are crucial in determining whether there is any context that can observe this series of actions. λ_a specifies the capabilities required for values that are sent on a, and λ_m specifies the capabilities that the context may assume to be present in the received location m. Thus, the edge labelled "$a?(m)$" is possible only if the received capabilities satisfy the requirements on a, *i.e.* $\lambda_m \leq \lambda_a$. Therefore the transitions of the LTS must be parameterized by a type constraint, expressing the knowledge of the context or environment.

Continuing to discuss this example, note that while the context's ability to use the received value a is constrained by the type of the channel on which the value was received, this is not true for the thread Q. The use of the names a and m in Q is constrained only by the allocation type of these names (via the restriction operator). The allocation type is in general more permissive than the received type. In particular, Q may be able to send m on a, even though no valid context is able to do this.

A second complication arises due to the fact that the name of a location may be communicated while the name of the location's parent remains hidden, or private. For example, we will have (for appropriate Δ):

$$\mathcal{L} \triangleright (\nu_\ell k)\,(\nu_k m)\,\ell[a!\langle m\rangle\, a!\langle k\rangle\, P] \xrightarrow{\Delta} a!m\ (\nu k)\,(\mathcal{L},\, {}_\ell k,\, {}_k m \triangleright \ell[a!\langle k\rangle\, P])$$

Note that in the residual, the restriction on m is lifted, while that on its parent k is maintained. In addition $\mathcal{L}$ must be updated to record the ancestry of m, which of course includes k, and thus the restriction on k is forced to sit outside the configuration, encompassing $\mathcal{L}$.

In such a restricted configuration, the restriction operator limits the power of an outside observer to establish the structure of the location tree. Suppose in the above example that m is alive and is communicated with the **run** capability. In this case an observer can establish that m is an descendant of ℓ (using the predicate $\twoheadrightarrow\ell$ at m) but cannot establish m's parentage; *i.e.* for no k will the predicate $\rightarrow k$ be true at m. From the standpoint of the receiver, the location m is an *orphan*. In the example, the subsequent communication of k helps to clarify the ancestry of m:

$$(\nu k)\,(\mathcal{L},\, {}_\ell k,\, {}_k m \triangleright \ell[a!\langle k\rangle\, P]) \xrightarrow{\Delta} a!k\mathcal{L},\, {}_\ell k,\, {}_k m \triangleright \ell[P]$$

The reverse situation occurs when a process receives a location from the environment without knowledge of its parentage. Rather than a *restricted configuration*, the result is a configuration with a *partial location tree*, *i.e.* a local tree in which some nodes are "missing". The LTS is defined over these restricted, partial configurations.

The transitions of the LTS are labelled with actions μ, defined as follows:

$$\begin{aligned}\mu \ ::=\ & \tau \mid a!V{:}\zeta \mid (\nu\widetilde{r})a?V \\ & \mid \downarrow\ell \mid \downarrow L \mid \ell{\rightarrow}k \mid \ell{\rightsquigarrow}k \mid L{\rightsquigarrow}k\end{aligned}$$

Here, the label τ represents an *autonomous* action, *i.e.* an action that does not require the cooperation of the surrounding context. These include internal communication, goto, migration, halt and conditional testing. Note that some of these actions are not "internal" in the traditional sense. In the full paper, we prove that $\xrightarrow{\tau}$ and the reduction relation coincide. The other actions all require participation by the surrounding context. Four of these forms of actions are straightforward, being simple generalizations of those used in [18]:

$a!V{:}\zeta$ the context receives V with permissions ζ.
$(\nu\widetilde{r})a?V$ the context sends V, revealing private names $\widetilde{r}$.
$\downarrow\ell$ the context kills location ℓ.
$\ell{\rightarrow}k$ the context moves ℓ to k.

The other three forms of actions involve the manipulation of private locations maintained by the context but hidden from the process. In the first of these, the context moves location ℓ to be a child of a private location.

$\ell{\rightsquigarrow}k$ the context moves ℓ to a private descendant of k.

In the last two forms of action, the context manipulates locations which are already sublocated at a private location. In the following explanations, suppose that L is a set of orphans and that m is a hidden ancestor of (all locations in) L, and that m is a descendant of all known ancestors of (all locations in) L.

$\downarrow L$ the context kills m, a private ancestor of L.
$L{\rightsquigarrow}k$ the context moves m, a private ancestor of L, to k or to a private descendant of k.

Having defined the LTS, we face one remaining complication before arriving at a suitable notion of bisimulation. In the ordinary π-calculus, when a context receives a name repeatedly it is only the first reception that "matters"; after the first reception the name is known, and it remains known henceforth. In Dπ, again, the situation is more subtle. Consider the process:

$$P = (\nu_m a)(\nu_m b)\ \ell[c!\langle m{::}a\rangle\, c!\langle m{::}b\rangle]$$

After a context receives both communications on c, it "knows" of both channels a and b at m. We might expect that it could, therefore, send the pair (a,b); however, no well typed context is capable of sending this value. The two copies of m are received into separate variables with separate typings.

It is worthwhile pointing out that communication of *local* names is somewhat more powerful than the communication of *remote* names. For example, consider:

$$Q = (\nu_m a)(\nu_m b)\ \ell[m :: c!\langle a\rangle\, c!\langle b\rangle]$$

After receiving both a and b from Q, a context can indeed send the pair (a,b), as evidenced by the process:

$$k[m :: c?(x)\, c?(y)\, d!\langle x,y\rangle]$$

A similar problem arises with permissions: receiving two copies of a name, one with **snd** permission and another with **rcv** permission, does not grant the same capabilities as receiving a single copy of the name with both **snd** and **rcv** permissions. This characteristic is shared by both remote and local communication, as exemplified by P and Q, above.

Our solution to the problem is to define bisimulation with respect to a more general notion of type environment which allows us to distinguish permissions associated with each instance of a name. A similar approach has been developed independently by Boreale and Sangiorgi in order to define bisimulations for the π-calculus without matching [6].

5 Conclusions

We have presented a novel foundational language, Dπ, for the study of typed distributed systems. The language includes constructs for process migration and failure. In the operational semantics, explicit tags are used to indicate the permissions associated with each instance of a name; when passing values, processes communicate tags as well as names, possibly reducing the permissions available on a name before sending it. We then defined a type system for Dπ which ensures that for well-typed terms, tags can be ignored without the risk of names being used in ways that violated their permissions. Finally we defined barbed congruence under constraint Δ and outlined the design a labeled transition system which captures this relation.

Related work. There are two strains of related work; the first concerns the language itself, the second, the type system. Our model of location hierarchy, migration and failure is similar to model used in the distributed join calculus (DJoin) of Fournet, Gonthier and their co-workers [12]. Dπ is a larger language, however; in addition to permissions, Dπ includes synchronous communication, the goto operator, for code movement, and position testing; all of these require nontrivial encodings in DJoin. In addition, message routing is not "automatic" as it is in DJoin. To send a message to a remote location in Dπ, a process must first spawn a thread which goes to that location. These features make locations more "visible" in Dπ than they are in DJoin.

The goto operation is based on the "spawn" operation found in Facile [13] and related calculi [2, 3]; this operator is objective [8] and operates only on *inactive* code, making it very inexpensive to implement. By contrast, the subjective "migration" operator of DJoin operates on running code, making it more flexible and costlier to implement. By including both types of code movement in Dπ, we bring the semantics of these operations up to the "top level" of the calculus, rather than relying on complex encodings whose semantic implications are difficult to ascertain.

Process movement is also the central concern of Cardelli and Gordon's ambient calculus [8], although in their work locations (or *ambients*) are used to model a hierarchy of administrative domains, rather than, as in Dπ, a hierarchy of

physical distribution as determined by failure dependencies.

Dπ arose from an attempt to understand the use of permissions in distributed systems and in this sense, it is related to work on the spi-calculus [1] of Abadi and Gordon. There, however, the permissions are used to control the ability to interpret data that has been received.

The type system most closely related to ours is that of Pierce and Sangiorgi [16]. Besides the fact that we treat a distributed language, with an extended collection of types, we have made two main contributions, building on [16]. First we presented our language in such a way that the communication of permissions is explicit; we believe that this gives our Type Safety Theorem more operational intuition than that of [16]. Second, we have outlined an alternative characterization of barbed congruence, relativised to a typing constraint, as a bisimulation relation. We have been careful to construct the language so that a context can determine the structure of entire location tree and can test every name for equality. Without these properties our alternative characterization would fail.

Other type systems for controlling the use of names in distributed systems have been presented by Amadio [3] and Sewell [22]. Amadio's type system seeks to guarantee that names are defined at only one location; his type system also guarantees that at every moment there is exactly one abstraction placed at each channel. Sewell studies a language similar to Dπ, but closer in spirit to the join calculus. He generalizes the type system of Pierce and Sangiorgi by distinguishing local from non-local communication, with the goal of allowing compiler optimizations.

Recently, Boreale and Sangiorgi [6] have presented an alternative characterization of the equivalence studied in [16] for a calculus without matching. Using their technique, one should be able to extend our results to a distributed language with the ability to match names explicitly predicated upon a permission.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, January 1997.

[2] R. Amadio and S. Prasad. Localities and failures. In *Proc. 14th Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[3] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[4] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. In U. Montanari and V. Sassone, editors, *CONCUR: Proceedings of the International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 147–162, Pisa, August 1996. Springer-Verlag.

[5] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 81–94, San Francisco, January 1990. ACM Press.

[6] Michele Boreale and Davide Sangiorgi. Typed bisimulation for the pi-calculus. Talk at EXPRESS97, September 1997.

[7] G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, Sophia-Antipolis, 1992.

[8] L. Cardelli and A. D. Gordon. Mobile ambients, 1997. Draft, Available from `http://www.cl.cam.ac.uk/users/adg/`.

[9] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. A preliminary version appeared in Proceedings of the 22nd ACM Symposium on Principles of Programming.

[10] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[11] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Paris, January 1996. ACM Press.

[12] C. Fournet, G. Gonthier, J.J. Levy, L. Marganget, and D. Remy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *CONCUR: Proceedings of the International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, August 1996. Springer-Verlag.

[13] A. Giacalone, P. Mishra, and S. Prasad. A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[14] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.

[16] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. Extended abstract in LICS '93.

[17] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[18] James Riely and Matthew Hennessy. Distributed processes and location failures. Computer Science Technical Report 2/97, University of Sussex, Department of Computer Science, 1997. Available from `http://www.cogs.susx.ac.uk/`.

[19] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. Computer Science Technical Report 4/97, University of Sussex, Department of Computer Science, 1997. Available from `http://www.cogs.susx.ac.uk/`.

[20] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[21] Davide Sangiorgi. Localities and true-concurrency in calculi for mobile processes. *Theoretical Computer Science*, 155, 1996.

[22] Peter Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, Computer Laboratory, University of Cambridge, August 1997.

[23] Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[24] David Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Edinburgh University, 1995.