Global/Local Subtyping and Capability Inference for a Distributed π-calculus

Peter Sewell

University of Cambridge

Peter.Sewell@cl.cam.ac.uk

Abstract

This paper considers how locality restrictions on the use of capabilities can be enforced by a static type system. A distributed π-calculus with a simple reduction semantics is introduced, integrating location and migration primitives from the Distributed Join Calculus with asynchronous π communication. It is given a type system in which the input and output capabilities of channels may be either global, local or absent. This allows compile-time optimization where possible but retains the expressiveness of channel communication. Subtyping allows all communications to be invoked uniformly. We show that the most local possible capabilities for internal channels can be inferred automatically.

1 Introduction

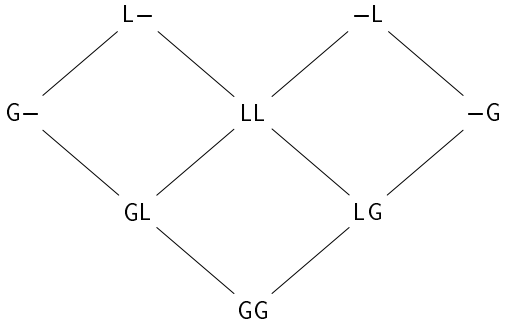
A central theme in programming language and system design is that of restricting access to resources to be in some sense *local*. This can support clean design, allow efficient implementation, and provide robustness against accidental errors and malicious attacks. The development of ubiquitous networking, particularly of systems in which executing agents or executable code are communicable, brings new kinds of resource and locality to the fore. One has resources such as the capability to input or output on a communication channel, to read or write a distributed reference cell, and to decrypt or encrypt with a cryptographic key pair. It may be desirable to restrict the use of these capabilities to, for example, a single agent, a group of trusted agents, a region of the network or a machine address space.

In this paper we consider how such restrictions, particularly the first, can be enforced by a static type system. We introduce a *distributed π-calculus*, with primitives for location, migration and π-calculus style channel communication, as an idealisation of a mobile agent programming language. It is given a type system in which the input and output capabilities for a communication channel can be either *global*, and therefore usable within any location, restricted to be *local*, and therefore usable only within the location where the channel is declared, or *absent*. The type system allows local communication to be implemented efficiently, while subtyping and subsumption ensure that, from the programmer's point of view, it is not unduly restrictive. The constructs for input and output along a channel are independent of whether its capabilities are global or local, thus facilitating programming. At the same time the programmer can distinguish between local and (potentially expensive) global communications via the typing of channel declarations. For the type system to be pragmatically usable it is essential that capability annotations can often be inferred automatically – one would like internal channels to be given the most local capabilities possible, to allow optimisation. We show that this can be done, by showing that typing is preserved by least upper bounds in a modified subtype order.

The distributed π-calculus used is introduced in Section 2. It is designed to allow the global/local type system to be presented clearly; it does not address other issues, such as name services, failure and administrative domains, that arise in mobile agent programming. It builds on the *π-calculus* of Milner, Parrow and Walker [MPW92]. The π-calculus is often described as a calculus of mobile processes. Strictly, however, this refers to the mobility of the scopes of channel declarations – channels are statically scoped, but their scopes may change over time as channel names are sent outside their current scopes. There is no other notion of locality or of identity of processes, so to directly model distributed phenomena, such as migration of agents, failure of machines or knowledge of agents, one must add primitives for grouping π-calculus

processes, into units of migration, failure or shared knowledge respectively. This was done by Amadio and Prasad [AP94] in order to model an abstraction of the failure semantics of *Facile* [TLK96], an extension of ML with distribution primitives. More recently, Fournet et al have proposed the *Distributed Join Calculus* and a related programming language [FG96, FGL$^+$96]. To model migrating agents one needs at least a two level hierarchy, with agents, each containing some processes, located at named virtual machines. The Distributed Join Calculus takes a generalisation of this, with primitive tree-structured *locations*. Immediate sublocations of the root model virtual machines; descendants of these represent hierarchically structured mobile agents. Locations are named; new locations can be created and their names can be scope-extruded by communication just as π-calculus channel names can be. Locations that are not immediate sublocations of the root may migrate, changing their parent location. The sublocations of a location thereby migrate with it. We adopt similar location and migration primitives (this choice gives a reasonably clean calculus, but it is not critical for the type system). For communication between agents a wide variety of primitives may be useful in practice. In this paper we adopt those of an asynchronous π-calculus [Bou92, HT92]. They are rather high level, operating independently of the physical locations of agents, and expressive, allowing any number of readers and writers on a channel. As in [AP94] channels have locations, giving the agents in which their queues of blocked writers or readers are stored.

The type system is introduced in Section 3. Generalising the Input/Output type system of Pierce and Sangiorgi [PS96], it has channel types annotated with capabilities. Here they are of the form $\updownarrow_{io} T$, with input and output capabilities i and o each taken from $\{-, \mathsf{L}, \mathsf{G}\}$. The intuition is that a G capability may be used at any location, an L capability may be used only at the location of the channel concerned and a $-$ capability may not be used at all. The types $\updownarrow_{--} T$ are replaced by a single top type $\top$, leaving the capabilities below.



For example, consider a channel x of type $\updownarrow_{io} T$, which is located at a location named k. If $io = \mathsf{GG}$ then x is usable at any location, both for input and for output. If $io = \mathsf{LL}$ then x is usable only within location k, but still for both input and output. If $io = \mathsf{LG}$ then x can be used for output anywhere, but for input only within location k. Such a channel might be used for sending requests to a server located at k. Conversely, if $io = \mathsf{GL}$ then x can be used for input anywhere, but for output only at location k. Such a channel might be used for receiving results from servers, or for 'pushed' data from an information source. The tags $\pm, -, +$ of [PS96] correspond to the capabilities $\mathsf{GG}, \mathsf{G}-, -\mathsf{G}$, for global communication, and to $\mathsf{LL}, \mathsf{L}-, -\mathsf{L}$, for local communication.

A subtyping order is lifted from the tag ordering above (with $i-$ and $-o$ covariant and contravariant respectively), allowing subcapabilities to be communicated. For example if $x : \updownarrow_{\mathsf{LG}} T$ then x may be transmitted globally, along channels of type $\updownarrow_{\mathsf{GG}} \updownarrow_{-\mathsf{G}} T$, to readers that are guaranteed to use it only at type $\updownarrow_{-\mathsf{G}} T$.

The expressiveness of the system should aid programmers by detecting errors at compile time, including communications that inadvertently potentially involve network communication. In an implementation, channels with tags LL, $\mathsf{L}-$ and $-\mathsf{L}$ can be implemented with data structures that are local to a single agent, and so always on the same (albeit possibly changing) machine. Their names need not be globally unique, but only unique within their location (note that this implies that equality testing of channel names should not be available) and need not

be registered with global name services. Channels with tags GL (resp. LG) are subject to fewer optimizations, but still allow the references to the channel data structures by writers (resp. readers) to be local pointers.

The typing rules involve two novel features — the formation of certain types must be forbidden by kinding rules and the capabilities of channels must be compared with capabilities at which they can be used by readers. The main soundness result is subject reduction (Theorem 1); in addition one can see by examination of the typing rules that no well-typed process can immediately use a capability that it does not have.

In Section 4 we show that the most local possible capabilities for channels can be inferred (Theorem 2). Some related work and possible generalisations are discussed in Section 5. Proofs are omitted for lack of space. Further discussion of alternative calculi, and the proof of soundness for an extension of the type system with type variables and recursive types, may be found in [Sew97a].

2 A distributed π-calculus

In this section the syntax and operational semantics of our distributed π-calculus (dpi for short) are given. The operational semantics is a rather mild extension of that for the asynchronous π-calculus. It is a reduction semantics, defining reductions over process terms (no additional notion of configuration is required) using a structural congruence. It differs from an asynchronous π semantics in only two respects — there is a reduction rule for migration and the standard structural congruence and reduction rules are adapted to terms containing location information. Location and channel names are both subject to scope extrusion, just as π-calculus channel names are. The semantics and type system can therefore be simplified by treating new location and channel declarations similarly, taking a single binder $(\mathbf{new}\ x : @_l T)\_$ which declares x to be a sublocation of l (respectively a channel located at l) if the type T is the type loc of location names (respectively a channel type).

Types To the types of channels and locations introduced above we add *base types*, ranged over by B, for example a unit type 1 and Int, pairs, as a first step towards more interesting datatypes, and a type to be the top of the subtype order. The *pre-types*, ranged over by S, T, U, V, are given by

$$T \quad ::= \quad B \;\Big|\; T \times T \;\Big|\; \updownarrow_{io} T \;\Big|\; \mathrm{loc} \;\Big|\; \top$$

Only some pre-types will be considered well-formed. The syntax of processes involves types, and hence the reduction semantics does also. Its definition does not depend on them in any interesting way, however, so we defer the type formation rules to Section 3.

Processes We take an infinite set $\mathcal{X}$ of *names*, ranged over by a, j, k, l, x, y, z and containing a distinguished name top. We let m, n, p, q range over $\mathbb{N}$. *Values*, ranged over by u, v, w, are

$$v \quad ::= \quad b \;\Big|\; x \;\Big|\; \langle v, v \rangle$$

where b ranges over elements of the base types. There are two extremal possibilities for adding location information to terms. In one a locator applies to the largest possible unit, with all co-located subterms gathered into a single subterm. This is adopted, for example, in the *Ambient Calculus* of Cardelli and Gordon [CG98]. For dpi, however, communication is possible *across* the location tree structure, so to give a reduction semantics (in which writers and readers at different locations must be brought syntactically adjacent by a structural congruence) the other extreme is adopted, with each elementary subterm explicitly located. Accordingly, *processes*,

ranged over by P, Q, R, are:

P	::=	$@_u\overline{v}w$	at location u, output value w on channel v
		$@_u v(y).P$	at u, input a value from channel v and bind it to y in P
		$@_u !\, v(y).P$	replicated input
		$@_u$**migrate_to** v **then** P	migrate location u to become a sublocation of v
		$@_u$**let** $\langle y:T, y':T'\rangle = w$ **in** P	at u, bind the halves of the pair w to y and y' in P
		(**new** $y:@_u T)P$	declare a new channel or location named y, of type T, located at u and binding in P
		0	the null process
		$P \mid P$	parallel composition

The names y and y' above, which must be distinct in the **let** case, bind in the respective subterms P (in particular, in (**new** $y:@_u T)P$ the scope of y does not include u); we work up to alpha conversion of bound names. The free names of a value v and process P will be denoted by $\mathrm{fn}(v)$ and $\mathrm{fn}(P)$ respectively. The substitution of a value v for a name x in P will be written $\{v/x\}P$. Output values and input binders of type 1 will often be elided.

The syntax of processes includes some nonsensical terms, which the reduction semantics gives nonsensical reductions to. They will be formally excluded by the typing rules but two points are worth mentioning now. Firstly, in well-typed processes the u and v appearing in the grammar will always be names. They are allowed to be arbitrary values in the syntax so that substitution of values for names is always defined. Secondly, the syntax includes terms which can teleport after a prefix, e.g.

$$@_k x(y).@_l x(z).0 \qquad @_k x(y).@_y x(z).0$$

For conceptual simplicity we would like migration to be the *only* way in which processes may move, and so want locators $@_l\_$ to describe the locations of processes rather than cause them to move. Teleporting terms are excluded by considering the set of free inhabited locations $\mathrm{fl}(P)$ of a process P. This is the set of the free location names that are inhabited in P by outputs, inputs, migrates, pair splits, channels or locations. It is the set of names which occur free in P within the argument u of a subterm $@_u\_$. For example, $\mathrm{fl}((\mathbf{new}\ j:@_l\mathrm{loc})(@_j\overline{x} \mid @_l\overline{x} \mid @_k\overline{x})) = \{k,l\}$. The type system will require, in every prefix located at l with continuation P, that $\mathrm{fl}(P) \subseteq \{l\}$.

Reduction semantics Structural congruence $\equiv$ is the least congruence relation over processes satisfying the following.

$$P \mid 0 \equiv P \tag{1}$$
$$P \mid Q \equiv Q \mid P \tag{2}$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \tag{3}$$
$$(\mathbf{new}\ x:@_u S)(\mathbf{new}\ y:@_v T)P \equiv (\mathbf{new}\ y:@_v T)(\mathbf{new}\ x:@_u S)P \qquad x \notin \mathrm{fn}(v), y \wedge y \notin \mathrm{fn}(u) \tag{4}$$
$$P \mid (\mathbf{new}\ x:@_v T)Q \equiv (\mathbf{new}\ x:@_v T)(P \mid Q) \qquad x \notin \mathrm{fn}(P) \tag{5}$$

The first three equations are standard, allowing parallel compositions to be treated as multisets. Equation 5 allows scope extrusion, both of channel names and of location names. Equation 4 allows new-binders to be permuted; the side condition ensures that the location tree structure, and the locations of channels, are preserved.

The reduction relation $\longrightarrow$ over processes is the least relation satisfying the following.

$$@_k\overline{x}v \mid @_l x(y).P \longrightarrow \{v/y\}P$$
$$@_k\overline{x}v \mid @_l !\, x(y).P \longrightarrow \{v/y\}P \mid @_l !\, x(y).P$$
$$@_l\mathbf{let}\ \langle y_1:T_1, y_2:T_2\rangle = \langle v_1, v_2\rangle\ \mathbf{in}\ P \longrightarrow \{v_1/y_1\}\{v_2/y_2\}P$$
$$(\mathbf{new}\ l:@_j T)(\mathbf{new}\ \Delta)(Q \mid @_l\mathbf{migrate\_to}\ k\ \mathbf{then}\ P) \longrightarrow (\mathbf{new}\ l:@_k T)(\mathbf{new}\ \Delta)(Q \mid P)$$
$$\text{if } \{k,l\} \cap \mathrm{dom}(\Delta) = \{\} \wedge k \neq l$$

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \qquad \frac{P \longrightarrow Q}{(\mathbf{new}\ x:@_l T)P \longrightarrow (\mathbf{new}\ x:@_l T)Q} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q}$$

where we define $(\mathbf{new}\ \Delta)P$, for lists of the grammar $\Delta ::= \bullet \mid \Delta, x\colon @_l T$, by $(\mathbf{new}\ \bullet)P = P$ and $(\mathbf{new}\ \Delta, x\colon @_l T)P = (\mathbf{new}\ \Delta)(\mathbf{new}\ x\colon @_l T)P$. The first two reduction rules are the standard communication rules for an asynchronous π-calculus; note that the communications can take place irrespective of the locations of the writer, reader and channel. The third is an unproblematic pair splitting reduction. The fourth is the only substantially new reduction rule. It allows location l to migrate from being a sublocation of j to become a sublocation of k. After the migration the continuation P is released. The additional context $(\mathbf{new}\ \Delta)(Q \mid \_)$, which is preserved by the reduction, is required as the scope of l may contain other location and channel declarations, and processes, that mention l. In particular, note that Q may contain other subterms $@_l \ldots$ that remain located at l as it migrates. Note also that the side condition means that the rule is not applicable if k is a sublocation of l. Such migrations, which would introduce a cycle into the location tree, are blocked, although later migrations may unblock them. The last three rules are standard.

The sublocation tree of a migrating location is unchanged, and so migrates with it. The unit of migration is thus a subtree of locations with all their processes and channels. The largest unit that is guaranteed to stay together (and so always be on the same machine), however, is not a subtree but just the processes and channels at a single location — its sublocations may migrate away. The tree structure is therefore essentially orthogonal to global/local typing.

Examples We give some simple example processes that will be well-typed in the empty context. First, a server that returns the result of some computation (in this trivial example it simply pairs the argument with itself):

$$
\begin{array}{l}
(\mathbf{new}\ \mathrm{pairServer}\colon @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{client}\colon @_{\mathrm{top}}\mathrm{loc}) \\
\quad (\mathbf{new}\ \mathrm{pair}\colon @_{\mathrm{pairServer}} \updownarrow_{\mathsf{LG}} (\mathrm{Int} \times \updownarrow_{-\mathsf{G}} (\mathrm{Int} \times \mathrm{Int}))) \\
\qquad @_{\mathrm{pairServer}}\,!\,\mathrm{pair}(y).@_{\mathrm{pairServer}}\mathbf{let}\ \langle n\colon \mathrm{Int}, c\colon \updownarrow_{-\mathsf{G}} (\mathrm{Int} \times \mathrm{Int})\rangle = y\ \mathbf{in}\ @_{\mathrm{pairServer}}\overline{c}\langle n, n\rangle \\
\qquad \mid \\
\qquad (\mathbf{new}\ c\colon @_{\mathrm{client}} \updownarrow_{\mathsf{LG}} (\mathrm{Int} \times \mathrm{Int}))@_{\mathrm{client}}\overline{\mathrm{pair}}\langle 7, c\rangle \mid @_{\mathrm{client}}c(x).\cdots
\end{array}
$$

Note the use of subsumption for typing the output $@_{\mathrm{client}}\overline{\mathrm{pair}}\langle 7, c\rangle$. Secondly, a rudimentary tracker, that receives location names on a channel move (perhaps provided by an active badge system controller) and migrates to them:

$$
\begin{array}{l}
(\mathbf{new}\ \mathrm{l}_1\colon @_{\mathrm{top}}\mathrm{loc}) \cdots (\mathbf{new}\ \mathrm{l}_3\colon @_{\mathrm{top}}\mathrm{loc}) \\
(\mathbf{new}\ \mathrm{controller}\colon @_{\mathrm{top}}\mathrm{loc}) \\
\quad (\mathbf{new}\ \mathrm{move}\colon @_{\mathrm{controller}} \updownarrow_{\mathsf{GL}} \mathrm{loc}) \\
\\
\qquad @_{\mathrm{controller}}\overline{\mathrm{move}}\,\mathrm{l}_1 \mid \cdots \mid @_{\mathrm{controller}}\overline{\mathrm{move}}\,\mathrm{l}_3 \\
\qquad \mid \\
\qquad (\mathbf{new}\ \mathrm{follower}\colon @_{\mathrm{controller}}\mathrm{loc}) \\
\qquad\quad @_{\mathrm{follower}}\,!\,\mathrm{move}(l).@_{\mathrm{follower}}\mathbf{migrate\_to}\ l\ \mathbf{then}\ \cdots \\
\qquad\quad \mid @_{\mathrm{follower}} \cdots
\end{array}
$$

A more realistic tracker would have additional communications so that the moves could be sequentialised. As these examples show, the syntax of processes contains redundant location information. The design of a less verbose representation, allowing co-located processes to be gathered together at compile time, is discussed in [Sew97a]. Also discussed there is a calculus that is better suited to use by programmers, allowing locators to occur more freely. Both of these require a more complex operational semantics.

Action calculus semantics There is a rather large space of possible calculi with reduction semantics. One way of understanding it, particularly for comparing different calculi, is to put them into a common framework, such as the *Action Calculi* of Milner [Mil96]. This provides a well-understood structural congruence, with a graphical intuition, that has clarified the design of dpi. As an illustration, we give an action calculus presentation of the fragment of dpi

without pairs or base type values. We take the arity monoid $(\mathbb{N}, +, 0)$, the names of arity 1 to be $\mathcal{X}$ and controls:

$$\mathbf{new}_T : 1 \to 1 \quad \mathbf{out} : 3 \to 0 \qquad \frac{a : 1 \to 0}{\begin{array}{l}\mathbf{in}(a) : 2 \to 0 \\ \mathbf{rep}(a) : 2 \to 0\end{array}} \qquad \frac{a : 0 \to 0}{\mathbf{mig}(a) : 2 \to 0}$$

Comparing with the action calculus for the π-calculus $\mathsf{AC}(\nu, \mathbf{out}, \mathbf{box}, \mathbf{rep})$, from [Mil96, §5.4], the arities of **new**, **out**, **in** and **rep** are obtained by adding one to the source of their corresponding arities; the name binding the new port on a control giving the location of that control. There is an obvious mapping taking processes in the fragment of dpi considered to actions of arity $0 \to 0$. Taking the reaction rules of the action calculus to be the translation of the first, second and fourth dpi rules, this is a bijection, up to structural congruence, that preserves one-step reaction.

3 Global/local subtyping

This section gives the global/local type system. It defines a judgement $\Gamma \vdash P : \text{process}$ which should be read as 'under assumptions Γ the process P is well-formed'. As usual these contexts Γ contain assumptions on the types of names that may occur free in P. They must also contain assumptions on the locations of such names. *Pre-contexts* are therefore lists:

Γ	$::=$	$\bullet$	the empty context
		$\Gamma, x : @_l T$	Γ extended with name x, located at l, of type T

We now illustrate the three main phenomena that the type system must address. Firstly, a channel name must only be used (for input or output) if it has the appropriate capability, i.e. L or G for usages at its location; G for usages at other locations. For example, with respect to the context

$$\Gamma \stackrel{def}{=} k : @_{\mathrm{top}}\mathrm{loc},\ l : @_{\mathrm{top}}\mathrm{loc},\ w : @_l \updownarrow_{-\mathsf{G}} 1,\ z : @_l \updownarrow_{-\mathsf{L}} 1$$

we should have

$$\Gamma \vdash @_l \overline{w} : \text{process} \quad \Gamma \vdash @_l \overline{z} : \text{process}$$
$$\Gamma \vdash @_k \overline{w} : \text{process} \quad \Gamma \not\vdash @_k \overline{z} : \text{process}$$

Secondly, local capabilities must not be sent outside their locations. Consider the context

Γ	$\stackrel{def}{=}$	$k : @_{\mathrm{top}}\mathrm{loc},$	top level location
		$l : @_{\mathrm{top}}\mathrm{loc},$	top level location
		$z : @_l \updownarrow_{\mathsf{LL}} 1,$	local channel carrying 1, at l
		$x : @_l \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1$	global channel carrying names of local channels carrying 1, at l

and the process $P \stackrel{def}{=} @_l \overline{x} z \mid @_k x(y).@_k \overline{y}$. At first sight one might expect $\Gamma \vdash P : \text{process}$, but the reduction

$$@_l \overline{x} z \mid @_k x(y).@_k \overline{y} \longrightarrow @_k \overline{z}$$

can send both L capabilities of z out of l — it is clear that $\Gamma \vdash @_k \overline{z} : \text{process}$ should not hold, and hence that $\Gamma \vdash P : \text{process}$ should not. It is prevented by restricting type formation, ruling out channel types, such as $\updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1$, that can be used to communicate local capabilities globally.

Thirdly, there must be a restriction on the mention of names outside their locations. This is a little delicate, as one cannot simply forbid all such mentions of the names of channels that are declared with some local capability. Suppose $x : \updownarrow_{\mathsf{LL}} \updownarrow_{-o} 1$ is located at l and $z : \updownarrow_{-o'} 1$, and consider when $@_l \overline{x} z$ should be well typed. If $o' = \mathsf{G}$ then z may be located anywhere, as its output capability is global. If $o = \mathsf{G}$ and $o' = \mathsf{L}$ then z is not a subtype of the expected value, so $@_l \overline{x} z$ should never be well typed. On the other hand, if $o = o' = \mathsf{L}$ then the output capability of z is local and may be used by a reader on x, so z must be located at l also. The essential point is whether the capabilities of z and the capabilities at which it can be used

by readers (as determined by the type of x) share a local capability (either for input or for output). This will be captured by a relation of *colocality* over types. Note that if $x : \updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{L}} 1$ and $z : \updownarrow_{\mathsf{LG}}$ was not located at l the output should still be well typed, despite the fact that both types have a local capability.

Kinds, Contexts, Types and Values We first define four mutually recursive judgements, $\vdash T : K$, read as 'type T is well-formed and has kind K', $\vdash \Gamma\ ok$, read as 'context Γ is well-formed', $\Gamma \vdash v : T$, read as 'value v has type T', and $\Gamma \vdash x@l$, read as 'name x is located at l'.

The *kinds*, ranged over by K, are $\mathrm{Type}_{\gamma\varepsilon}$ where γ and ε range over the 2-point lattices $\mathsf{G} \leqslant -$ and $\mathsf{E} \leqslant -$ respectively. They are ordered by the product order. The intuition is that types that have a kind $\mathrm{Type}_{\mathsf{G}\varepsilon}$ are *global*, with values of such types being freely communicable between locations. Types that have a kind $\mathrm{Type}_{\gamma\mathsf{E}}$ are *extensible*; new names at these types may be created by new-binders. We write $\sqcup$ for the least upper bounds in these lattices. The kinding rules for *types* are:

$$\frac{}{\vdash B : \mathrm{Type}_{\mathsf{G}-}} \qquad \frac{\vdash T : \mathrm{Type}_{\gamma\varepsilon} \quad \vdash T' : \mathrm{Type}_{\gamma'\varepsilon'}}{\vdash T \times T' : \mathrm{Type}_{(\gamma\sqcup\gamma')-}} \qquad \frac{}{\vdash \mathrm{loc} : \mathrm{Type}_{\mathsf{GE}}} \qquad \frac{}{\vdash \top : \mathrm{Type}_{\mathsf{GE}}}$$

$$\frac{\vdash T : \mathrm{Type}_{\mathsf{G}\varepsilon} \quad io \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}\}}{\vdash \updownarrow_{io} T : \mathrm{Type}_{\mathsf{GE}}} \qquad \frac{\vdash T : \mathrm{Type}_{\mathsf{G}\varepsilon} \quad io \in \{\mathsf{GL}, \mathsf{LG}\}}{\vdash \updownarrow_{io} T : \mathrm{Type}_{-\mathsf{E}}} \qquad \frac{\vdash T : \mathrm{Type}_{\gamma\varepsilon} \quad io \in \{\mathsf{LL}, -\mathsf{L}, \mathsf{L}-\}}{\vdash \updownarrow_{io} T : \mathrm{Type}_{-\mathsf{E}}} \qquad \frac{\vdash T : K \quad K \leqslant K'}{\vdash T : K'}$$

The rules for channel types prevent the formation of types that could be used to carry local capabilities between locations. For example, we have:

$$\vdash \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1 : \mathrm{Type}_{-\mathsf{E}} \qquad \not\vdash \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1 : \mathrm{Type}_{--}$$

$$\vdash \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{GG}} 1 : \mathrm{Type}_{-\mathsf{E}} \qquad \vdash \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{GG}} 1 : \mathrm{Type}_{\mathsf{GE}}$$

and $\vdash \updownarrow_{io} \updownarrow_{i'o'} 1 : \mathrm{Type}_{-\mathsf{E}}$ iff $io \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}, \mathsf{GL}, \mathsf{LG}\} \Rightarrow i'o' \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}\}$, i.e. if io is at all global then $i'o'$ must be not at all local. Products are global only if both their components are global. Base types and $\top$ are global, as is loc, so location names may be communicated freely. For illustration, the types (in boxes) and global types (in double boxes) of the form $\updownarrow_{io} \updownarrow_{i'o'} 1$ are shown in Figure 1. The only extensible types are channel types, loc, and $\top$. The formation rules for *contexts* are:

$$\frac{}{\vdash \bullet\ ok} \qquad \frac{\vdash T : K \quad (\Gamma \vdash l : \mathrm{loc}) \vee (l = \mathrm{top}) \quad x \notin \mathrm{dom}(\Gamma) \cup \{\mathrm{top}\}}{\vdash \Gamma, x : @_l T\ ok}$$

Contexts thus contain location and type assumptions on free names. The rules ensure that locations are tree structured, with root top. The typing rules for values, and the rule for the location of names, are straightforward.

$$\frac{\vdash \Gamma, x : @_l T, \Delta\ ok}{\Gamma, x : @_l T, \Delta \vdash x : T} \qquad \frac{\vdash \Gamma\ ok \quad b \in |B|}{\Gamma \vdash b : B} \qquad \frac{\Gamma \vdash v : T \quad \Gamma \vdash v' : T'}{\Gamma \vdash \langle v, v' \rangle : T \times T'} \qquad \frac{\vdash \Gamma, x : @_l T, \Delta\ ok}{\Gamma, x : @_l T, \Delta \vdash x@l}$$

Subtyping The ordering on tags induces a subtype order on types — if $io \leqslant i'o'$ then a channel of type $\updownarrow_{io} T$ may be used as if it were a channel of type $\updownarrow_{i'o'} T$, which has weaker capabilities. As in [PS96], a tag io is covariant iff $o = -$, contravariant if $i = -$ and non-variant otherwise. The subtype order $\leqslant$ is the least binary relation over the pre-types such that

$$\frac{}{B \leqslant B} \qquad \frac{S_1 \leqslant T_1 \quad S_2 \leqslant T_2}{S_1 \times S_2 \leqslant T_1 \times T_2} \qquad \frac{io \leqslant i'o' \quad i' \neq - \Rightarrow S \leqslant T \quad o' \neq - \Rightarrow T \leqslant S}{\updownarrow_{io} S \leqslant \updownarrow_{i'o'} T} \qquad \frac{}{\mathrm{loc} \leqslant \mathrm{loc}} \qquad \frac{}{S \leqslant \top}$$

Figure 1: The subtype order over well-formed types $\updownarrow_{io} \updownarrow_{i''o''} 1$. They are ordered by $\updownarrow_{io} \updownarrow_{i''o''} 1 \leqslant \updownarrow_{i'o'} \updownarrow_{i'''o'''} 1$ iff $io \leqslant i'o'$ and $o' = - \Rightarrow i'o' \leqslant i'''o'''$, $i' = - \Rightarrow i'''o''' \leqslant i'o'$ and $o' \neq - \wedge i' \neq - \Rightarrow i'o' = i'''o'''$.

The replacement of the tag $--$ by a single top type ensures that subtyping is a partial order – otherwise names of types $\updownarrow_{--} T$ would be communicable but not usable, so we would have $\updownarrow_{--} S \leqslant \updownarrow_{--} T$ for all S and T. The subtype order over well-formed types $\updownarrow_{io} \updownarrow_{i'o'} 1$ is

illustrated in Figure 1. Note that the well-formed types are not up, down or convex-closed under the subtype order on pre-types.

Colocality We say that a tag is *local* if it contains an L capability and that two tags are *colocal* if they share a common L capability, i.e. $\mathrm{local}(io) \stackrel{def}{\Leftrightarrow} i = \mathsf{L} \vee o = \mathsf{L}$ and $\mathrm{colocal}(io, i'o') \stackrel{def}{\Leftrightarrow} (i = \mathsf{L} \wedge i' = \mathsf{L}) \vee (o = \mathsf{L} \wedge o' = \mathsf{L})$. The key properties of these definitions are that $\mathrm{colocal}(io, io) \iff \mathrm{local}(io)$ and that, if $io \leqslant i'o' \leqslant i''o''$ and $\mathrm{colocal}(io, i''o'')$, then $\mathrm{colocal}(io, i'o')$ and $\mathrm{colocal}(i'o', i''o'')$. Note that the local tags are neither up, down or convex closed in the tag ordering. Further, colocal is a symmetric relation but is not reflexive or transitive, or closed under relational composition with the tag ordering. It does satisfy $\mathrm{colocal}(io, i'o') \Rightarrow (io \leqslant i'o' \vee i'o' \leqslant io)$. Colocality is lifted from tags to a relation on well-formed types that are in the subtype relation as follows.

$$\frac{\begin{array}{l}(i = i' = \mathsf{L}) \vee (o = o' = \mathsf{L}) \\ \vdash \updownarrow_{io} S : \mathrm{Type}_{--} \\ \vdash \updownarrow_{i'o'} T : \mathrm{Type}_{--} \\ \updownarrow_{io} S \leqslant \updownarrow_{i'o'} T\end{array}}{\mathrm{colocal}(\updownarrow_{io} S, \updownarrow_{i'o'} T)} \qquad \frac{\begin{array}{l}\mathrm{colocal}(S_i, T_i) \\ \vdash S_{1-i} : \mathrm{Type}_{--} \\ \vdash T_{1-i} : \mathrm{Type}_{--} \\ S_{1-i} \leqslant T_{1-i}\end{array}}{\mathrm{colocal}(S_0 \times S_1, T_0 \times T_1)} i \in \{0, 1\}$$

We define the colocal names of a value with respect to two types that are in the subtype relation:

$$\frac{\begin{array}{l}\Gamma \vdash x : S \\ \mathrm{colocal}(S, T)\end{array}}{\Gamma \vdash x \in \mathrm{colocaln}(x, S, T)} \qquad \frac{\begin{array}{l}\Gamma \vdash x \in \mathrm{colocaln}(v_i, S_i, T_i) \\ \Gamma \vdash v_{1-i} : S_{1-i} \\ \vdash T_{1-i} : \mathrm{Type}_{--} \\ S_{1-i} \leqslant T_{1-i}\end{array}}{\Gamma \vdash x \in \mathrm{colocaln}(\langle v_0, v_1 \rangle, S_0 \times S_1, T_0 \times T_1)} i \in \{0, 1\}$$

The key properties lift as follows. If a value has any colocal names with respect to two types then those types are colocal, the types of the colocal names of a value are themselves local and the set of colocal names of a value varies contravariantly with the upper type.

Lemma 1 *If* $\Gamma \vdash x \in \mathrm{colocaln}(v, V, T)$ *then* $\mathrm{colocal}(V, T)$ *and there exists* U *such that* $\Gamma \vdash x : U$ *and* $\mathrm{colocal}(U, U)$. *If in addition* $\vdash S : \mathrm{Type}_{--}$ *and* $V \leqslant S \leqslant T$ *then* $\Gamma \vdash x \in \mathrm{colocaln}(v, V, S)$.

Processes Finally the typing rules for processes can be given.

$$\textsc{Out}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash x : \updownarrow_{io} T \\ \Gamma \vdash v : T' \\ T' \leqslant T \\ o \leqslant \mathsf{L} \\ o = \mathsf{L} \Rightarrow \Gamma \vdash x@l \\ \forall a\ .\ \Gamma \vdash a \in \mathrm{colocaln}(v, T', T) \Rightarrow \Gamma \vdash a@l\end{array}}{\Gamma \vdash @_l \overline{x} v : \mathrm{process}} \qquad (\textsc{Rep-})\textsc{In}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash x : \updownarrow_{io} T \\ \Gamma, y : @_l T \vdash P : \mathrm{process} \\ i \leqslant \mathsf{L} \\ i = \mathsf{L} \Rightarrow \Gamma \vdash x@l \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_l x(y).P : \mathrm{process} \\ \Gamma \vdash @_l\, !\, x(y).P : \mathrm{process}\end{array}}$$

$$\textsc{Let}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash v : T' \\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process} \\ T' \leqslant T_1 \times T_2 \\ \forall a\ .\ \Gamma \vdash a \in \mathrm{colocaln}(v, T', T_1 \times T_2) \Rightarrow \Gamma \vdash a@l \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \mathbf{let}\ \langle y_1 : T_1, y_2 : T_2 \rangle = v\ \mathbf{in}\ P : \mathrm{process}} \qquad \textsc{Mig}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash v : \mathrm{loc} \\ \Gamma \vdash P : \mathrm{process} \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \mathbf{migrate\_to}\ v\ \mathbf{then}\ P : \mathrm{process}}$$

$$\textsc{New}\ \frac{\begin{array}{l}\vdash T : \mathrm{Type}_{-\mathsf{E}} \\ \Gamma, x : @_l T \vdash P : \mathrm{process}\end{array}}{\Gamma \vdash (\mathbf{new}\ x : @_l T) P : \mathrm{process}} \qquad \textsc{Nil}\ \frac{\vdash \Gamma\ ok}{\Gamma \vdash 0 : \mathrm{process}} \qquad \textsc{Par}\ \frac{\begin{array}{l}\Gamma \vdash P : \mathrm{process} \\ \Gamma \vdash Q : \mathrm{process}\end{array}}{\Gamma \vdash P \mid Q : \mathrm{process}}$$

Most of the premises of these rules are routine; we discuss the others briefly.

Out The first premise ensures that l is a location. The second through fifth premises are analogous to those of the OUT rule of [PS96]. Name x must be a channel, value v must be of a subtype of the type carried by the channel, and the channel must have an output capability (either G or L). The fourth and fifth premises could be replaced by $\updownarrow_{io} T \leqslant \updownarrow_{-\mathsf{L}} T'$. The penultimate premise addresses the first phenomenon discussed at the beginning of this section, ensuring that if x has only a local output capability then it can only be used at its own location. The last premise addresses the third such phenomenon, ensuring that any transmitted channel names that have a local capability which can be used by receivers on x are located at l.

(Rep-)In This is very similar to OUT except for the premise $\mathrm{fl}(P) \subseteq \{l\}$, which prevents teleportation after the input. Note that for typing P it is assumed that y is located at l.

New This allows new-binding of names at channel types, loc, and $\top$.

Let This is similar to a combination of OUT and (REP-)IN (as, indeed, the reduction rule for LET is).

A few remarks: (1) The rules allow locations and channels, but not processes, to be located at top. This is consistent with the intuition that immediate sublocations of top model virtual machines. For other applications of the calculus different treatments of top are appropriate and should be straightforward. (2) Local channels can be sent outside their location (with reduced capabilities) and then back inside. Their local capabilities cannot then be used, however. (3) A name may be assumed to have a local type in a process P and still, if P is placed in a process context, engage in cross-location communication. (4) The let construct includes an explicit type for its pattern, which may be a supertype of the type of its value. Without this the set of typable processes would be unduly restricted. In the input construct the type of the pattern can be left implicit, as it is bounded by the type of the channel. (5) To add recursive types contexts must contain kind assumptions on type variables, type formation rules must be relativised to contexts, and enforce guardedness, subtyping must be defined coinductively, and type unfolding must be allowed in the value typing rules and definitions of subtyping and colocality. The details can be found in [Sew97a].

Soundness The main soundness result is that typing is preserved by reduction.

Theorem 1 (Subject reduction) *If* $\Gamma \vdash P : \mathrm{process}$ *and* $P \longrightarrow Q$ *then* $\Gamma \vdash Q : \mathrm{process}$.

To prove this it is necessary to show that typing is preserved by legitimate *context permutations*, by *relocation* (changes of location assumptions for names of non-local types), by *narrowing* (taking type assumptions of lower types, while keeping location assumptions constant) and by *substitution*. For reasons of space we state only the substitution lemma for processes.

Lemma 2 (Substitution — Processes) *The rule below is admissible.*

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash P : \mathrm{process} \\ \Gamma \vdash u : U \\ U \leqslant V \\ \forall a \,.\, \Gamma \vdash a \in \mathrm{colocaln}(u, U, V) \Rightarrow \Gamma \vdash a @ j \\ z \notin \mathrm{fl}(P)\end{array}}{\Gamma \vdash \{u/z\}P : \mathrm{process}}$$

The first three premises are standard. The fourth ensures that any names of the substituted value u are located at the same place as the substituted variable z was assumed to be at, if their actual and assumed types are colocal. The last premise ensures that no locators in P can be affected by the substitution.

In addition, it is easy to see from the typing rules that no well-typed process can immediately use a local capability outside its location. This can be made precise by immediate-soundness results such as the following.

Proposition 1 *If* $\Gamma \vdash (\mathbf{new}\ \Delta)(@_l \overline{x}v \,|\, Q) : \mathrm{process}$, $\Gamma, \Delta \vdash x : \updownarrow_{i\mathsf{L}} T$ *and* $\Gamma, \Delta \vdash x@k$ *then* $k = l$.

4 Capability Inference

One would like to be able to automatically infer the most local possible type for new-bound channels, to allow compile-time optimisation. Unfortunately, this is not possible in any straightforward sense based on the subtype order. Consider for example $k : @_{\mathrm{top}}\mathrm{loc}, z : @_k \updownarrow_{\mathsf{LL}} 1 \vdash (\mathbf{new}\ x : @_k T)(@_k \overline{x}z \,|\, @_k x(y).@_k \overline{y})$. This holds iff T is either $\updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{L}} 1$ or $\updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1$; these types are not related by subtyping. We can, however, infer the most local possible top-level capabilities for T. Take the modified 'subtype' order $\sqsubseteq$ (with all channel type constructors covariant) to be the least relation over the pre-types such that

$$\frac{}{B \sqsubseteq B} \qquad \frac{S_1 \sqsubseteq T_1 \quad S_2 \sqsubseteq T_2}{S_1 \times S_2 \sqsubseteq T_1 \times T_2} \qquad \frac{T \sqsubseteq T' \quad io \leqslant i'o'}{\updownarrow_{io} T \sqsubseteq \updownarrow_{i'o'} T'} \qquad \frac{}{\mathrm{loc} \sqsubseteq \mathrm{loc}} \qquad \frac{}{\top \sqsubseteq \top} \qquad \frac{}{\updownarrow_{io} S \sqsubseteq \top}$$

and define $\simeq$ to be the least relation over the pre-types such that

$$\frac{}{B \simeq B} \qquad \frac{S_1 \simeq T_1 \quad S_2 \simeq T_2}{S_1 \times S_2 \simeq T_1 \times T_2} \qquad \frac{T \simeq T'}{\updownarrow_{io} T \simeq \updownarrow_{i'o'} T'} \qquad \frac{}{\mathrm{loc} \simeq \mathrm{loc}} \qquad \frac{}{\top \simeq \top} \qquad \frac{}{\updownarrow_{io} S \simeq \top \quad \top \simeq \updownarrow_{io} S}$$

relating any two types that have essentially the same shape, neglecting capabilities. Say a set of types $\tilde{T} = \{ T_n \mid n \in N \}$ is *compatible* if it is non-empty and $\forall m, n \in N \,.\, T_m \simeq T_n$. One can show that any compatible $\tilde{T}$ has a least upper bound, written $\sqcup\tilde{T}$, with respect to $\sqsubseteq$. Lifting $\sqsubseteq$, $\simeq$, compatibility and $\sqcup$ pointwise to pre-contexts and processes, one can show that the typing judgements are preserved by taking least upper bounds with respect to $\sqsubseteq$.

Theorem 2 (Capability Inference)

1. *If* $\tilde{T}$ *is compatible and* $\forall n \in N \,.\vdash T_n : K_n$ *then* $\vdash \sqcup\tilde{T} : \sqcup\tilde{K}$.
2. *If* $\tilde{S}$ *is compatible,* $\tilde{T}$ *is compatible and* $\forall n \,.\, S_n \leqslant T_n$ *then* $\sqcup\tilde{S} \leqslant \sqcup\tilde{T}$.
3. *If* $\tilde{\Gamma}$ *is compatible,* $\tilde{S}$ *is compatible,* $\tilde{T}$ *is compatible,* $\forall n \,.\, \Gamma_n \vdash v : S_n$, $\forall n \,.\vdash T_n : \mathrm{Type}_{--}$, $\forall n \,.\, S_n \leqslant T_n$ *and* $\sqcup\tilde{\Gamma} \vdash x \in \mathrm{colocaln}(v, \sqcup\tilde{S}, \sqcup\tilde{T})$ *then* $\exists n \,.\, \Gamma_n \vdash x \in \mathrm{colocaln}(v, S_n, T_n)$
4. *If* $\tilde{\Gamma}$ *is compatible,* $\tilde{P}$ *is compatible and* $\forall n \in N \,.\, \Gamma_n \vdash P_n : \mathrm{process}$ *then* $\sqcup\tilde{\Gamma} \vdash \sqcup\tilde{P} : \mathrm{process}$.

For any pre-type S the set $\{ T \mid S \sqsubseteq T \}$ is finite. Given some $\Gamma \vdash P : \mathrm{process}$ (perhaps with types containing only GG capabilities, inferred by an algorithm along the lines of [Gay93, VH93, Tur96]) one can therefore compute the least upper bound of $\{ P' \mid P \sqsubseteq P' \wedge \Gamma \vdash P' : \mathrm{process} \}$. For the example above this gives $T = (\updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{L}} 1) \sqcup (\updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1) = \updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{L}} 1$. A more efficient algorithm will clearly be required in practice.

5 Conclusion

We conclude by briefly mentioning some related type systems and some possible future work. Capability-based type systems for process calculi have been given by De Nicola, Ferrari and Pugliese [DFP97], for a variant of Linda with localities, and by Riely and Hennessy [RH98], for a distributed π-calculus with site failure. Several authors have given type systems that enforce information flow properties, e.g. [HR98, SV98]. A type system that enforces secrecy and freshness for the Spi Calculus [AG97] has been proposed by Abadi in [Aba97]. In [Ste96] Steckler has given a static analysis technique for distributed Poly/ML with similar motivation to ours — to detect when channels are guaranteed to be local to a single processor. It incorporates also some reachability analysis, but does not separate input and output capabilities. Finally, Nielson and Nielson have studied static analysis techniques for CML, focussing on the number of usages of capabilities, in [NN95].

Special cases Three special cases of the type system may be of interest. In the Join Calculus the names introduced by a definition **def** D **in** P can only be used in P for output (to a first approximation D declares a single replicated reader on these names). For typing P, therefore, they are analogous to channels with capability $-\mathsf{G}$. One could allow the output capability to be local, taking the suborder of tags $-\mathsf{G} \leqslant -\mathsf{L}$. In some circumstances it may not be necessary to allow the input and output capabilities of channels to vary separately, cutting down to the suborder of tags $\mathsf{GG} \leqslant \mathsf{LL}$. This greatly reduces the complexity (although also the expressiveness) of the type system as all channel type constructors become nonvariant. It can be used to prevent the extrusion of local references from agents. A milder simplification is to omit the tags GL and LG, i.e. to take the product of the tags $\pm, -, +$ of [PS96] with the two-point lattice $\mathsf{G} \leqslant \mathsf{L}$. For such tags, if $io \leqslant i'o'$ then $\mathrm{colocal}(io, i'o') \iff \mathrm{local}(io) \wedge \mathrm{local}(i'o')$.

Linearity and Location Types In a distributed application one would expect many channels to be in some sense *linear*; in particular many servers will have a single replicated receiver (this observation motivates the introduction of join patterns in [FG96]). The integration of global/local typing with some form of linearity or receptiveness [Ama97, KPT96, San97] would allow more precise typing, and hence further optimizations, while retaining the expressiveness of general channel communication. One might also refine the system to allow location names to be local, with types $\mathrm{loc}_{\mathsf{G}}$ and $\mathrm{loc}_{\mathsf{L}}$, enabling migration to locations to be restricted, and allow locations to be *immobile* or *mobile*, restricting the migration *of* locations. Linearity would again be useful — a common case is that of *one-hop* locations (c.f. Java Applets).

Behavioural equivalences In order to reason about dpi processes a labelled transition system and behavioural congruence are required, perhaps building on the bisimulation congruence results of Riely and Hennessy [RH97, RH98], together with an understanding of the appropriate extensional equivalence for a mobile agent programming language, building on [Sew97b].

Typing for secrecy properties The focus of this paper has been on locality information that can be used for implementation optimization. Very similar type systems should be applicable to the enforcement of secrecy properties for cryptographic keys or nonces. For this it would be desirable to take capabilities not just from $\{\mathsf{G}, \mathsf{L}, -\}$ but from the lattice of arbitrary sets of location names, lifted above a bottom element G. These (dependent) types would allow new names (modelling keys, for example, as in the Spi Calculus) to be created that are restricted to a dynamically calculated set of individuals. One would want a rather strong soundness result — the analogue of Theorem 1 would only show that secrecy is preserved by well-typed processes, whereas an attacker may perform some ill-typed computation.

Acknowledgements The author would like to thank Cédric Fournet, Robin Milner, Benjamin Pierce, Paweł Wojciechowski, and the Thursday group, for interesting discussions about this work, and to acknowledge support from EPSRC grant GR/K 38403 and Esprit Working group 21836 (CONFER-2).

References

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *TACS '97 (open lecture), LNCS 1281*, pages 611–638, September 1997.

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich*, pages 36–47. ACM Press, April 1997.

[Ama97] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. COORDINATION 97, LNCS 1282*, 1997.

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of* 14^{th} *FST and TCS Conference, FST-TCS'94. LNCS 880*, pages 205–216. Springer-Verlag, 1994.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98*, March 1998.

[DFP97] Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proc. COORDINATION '97, LNCS 1282*, 1997.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd POPL*, pages 372–385. ACM press, January 1996.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. In *Proceedings of the 20th POPL*. ACM Press, 1993.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th POPL*, January 1998.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing. LNCS 612*, pages 21–51, 1992.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of the 23rd POPL*, pages 358–371. ACM press, January 1996.

[Mil96] Robin Milner. Calculi for interaction. *Acta Informatica*, 33:707–737, 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[NN95] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *Proceedings of TAPSOFT 95 (FASE). LNCS 915*, 1995.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of ICALP '97. LNCS 1256*, pages 471–481. Springer-Verlag, July 1997.

[RH98] James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th POPL*, January 1998.

[San97] Davide Sangiorgi. The name discipline of uniform receptiveness. In *Proceedings of ICALP '97. LNCS 1256*, pages 303–313, 1997.

[Sew97a] Peter Sewell. Global/local subtyping for a distributed π-calculus. Technical Report 435, University of Cambridge, August 1997. Available from `http://www.cl.cam.ac.uk/users/pes20/`.

[Sew97b] Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer-Verlag, 1997.

[Ste96] Paul Steckler. Detecting local channels in distributed Poly/ML. Technical Report ECS-LFCS-96-340, University of Edinburgh, January 1996.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th POPL*, January 1998.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *Proceedings of CONCUR '96. LNCS 1119*, pages 278–298. Springer-Verlag, August 1996.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation.* PhD thesis, University of Edinburgh, 1996.

[VH93] Vasco Thudichum Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic π-calculus. In *Proceedings of CONCUR '93. LNCS 715*, pages 524–538, 1993.