Resource Access Control in Systems of Mobile Agents

Matthew Hennessy[†] and James Riely[‡]

[†] COGS, University of Sussex, Brighton BN1 9QH, UK; and
[‡] CTI, DePaul University, 243 S. Wabash Ave., Chicago, IL 60604, USA.

Received ?; accepted ?

We describe a typing system for a distributed π-calculus which guarantees that distributed agents cannot access the *resources* of a system without first being granted the *capability* to do so. The language studied allows agents to move between distributed *locations* and to augment their set of capabilities via communication with other agents. The type system is based on the novel notion of a *location type*, which describes the set of resources available to an agent at a location. Resources are themselves equipped with capabilities, and thus an agent may be given permission to send data along a channel at a particular location without being granted permission to read data along the same channel. We also describe a *tagged* version of the language, where the capabilities of agents are made explicit in the syntax. Using this tagged language we define *access violations* as runtime errors and prove that well-typed systems are incapable of such errors.

1. INTRODUCTION

Mobile computation, where independent agents roam widely distributed networks in search of resources and information, is fast becoming a reality. A number of programming languages, APIs and protocols have recently emerged which seek to provide high-level support for mobile agents. These include Java [30], Odyssey [15], Aglets [19], Voyager [24] and the latest revisions of the Internet protocol [25, 2]. In addition to these commercial efforts, many prototype languages have been developed and implemented within the programming language research community — examples include Linda [8, 9], Facile [16], Obliq [7], Infospheres [11], the join calculus [13], and Nomadic Pict [33]. In this paper we address the issue of resource access control for such languages.

Central to the paradigm of mobile computation are the notions of agent, resource and location. *Agents* are effective entities that perform computation and interact with other

First published as *University of Sussex Computer Science Technical Report 98/02*, U. of Sussex, 1998. Presented at *MFPS XIV* and *The 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, vol. 16(3) of Electronic Notes in Theoretical Computer Science, Elsevier, 1998. Research funded by CONFER II and EPSRC project GR/K60701. Author contact information available at `http://www.informatics.sussex.ac.uk/users/matthewh` and `http://www.depaul.edu/~jriely`.

agents. Interaction is achieved using shared *resources* such as memory cells, M-structures, objects (with shared methods and state) or communication channels. The use of the term "mobile" implies that agents are bound to particular *locations* and that this binding may vary over time, *i.e.* agents can *move*. Resources, on the other hand, are often fixed to a single location, although proxies and mirrors may be set up in order to distribute their contents.

In *open* distributed systems, such as the internet, it is unwise to assume that all agents are benign, and thus a certain amount of effort must be spent to ensure that vital resources are protected from unauthorized access. This can be accomplished by using a system of *capabilities* and by predicating resource access on possession of the appropriate capability. It is unreasonable, however, to expect that *every* use of *every* resource in a system be thus verified dynamically; such a requirement surely would degrade system performance unacceptably. Thus it is attractive to develop static analyses, or *typing systems* that guarantee controlled access to system resources.

We present a typed language for mobile agents which allows fine control over the use of resources in a system. We also define a *tagged* version of the language in which agents explicitly carry the sets of capabilities which they have acquired. Using this tagged language, we capture resource access violations as *runtime errors* and show that well-typed terms are incapable of such errors.

The language studied in this paper, called Dπ, is a distributed variant of the π-calculus [23], and thus the *resources* of interest are *channels* which support binary communication between agents. We take agents to be located *threads*, which are simply terms of the ordinary polyadic π-calculus [22], extended with primitives for *movement* between locations and for the creation of *new* locations. The language is similar to that studied by Amadio [3, 1]. There are two major differences: we ignore location failure and we restrict communication to be local. The second of these differences is more important. In Amadio's language and in most other distributed versions of the π-calculus [29, 13], there are two forms of movement: one for agents and another for *messages*, which can be seen as very simple agents consisting only of a value that is to be communicated remotely. Here we limit mobility to a single language construct, eliminating the possibility of remote communication without explicit movement. The language is presented in Section 2. We give several examples of its use in Section 3.

The type system is based on the notion of *location types* of the form:

$$\mathsf{loc}\{a_1{:}\mathrm{A}_1, ..., a_n{:}\mathrm{A}_n\}$$

Here each a_i is channel name, and each A_i is a *channel type*. The idea is that the type of a location embodies the sets of capabilities that an agent has at that location. If an agent knows of a location ℓ at type $\mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}$, then the agent has permission to use channels a and b at ℓ, but not any other channels. Capabilities are communicated through channels, and thus channel types may have the form $\mathsf{res}\langle \mathrm{L}\rangle$, which is the type for channels which communicate locations of type L.

Agents may restrict access to a resource by controlling the type of the channel over which the name of the resource is sent. Thus if an agent sends the name ℓ over a channel of type $\mathsf{res}\langle\mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}\rangle$, then the recipient gains access to channels a and b at ℓ. Instead, when the same name is communicated over a channel of type $\mathsf{res}\langle\mathsf{loc}\{a{:}\mathrm{A}\}\rangle$, the recipient gains access only to channel a at ℓ. Of course for such communication to be sound, the sender must have, for the value it is sending, all of the capabilities that the channel requires. Otherwise a sender could “forge” arbitrary capabilities. To formalize this requirement we introduce a *subtyping* relation on types. On location types, the subtyping relation is the same as traditional record or object subtyping:

$$\mathsf{loc}\{a_1{:}\mathrm{A}_1, ..., a_k{:}\mathrm{A}_k, ..., a_n{:}\mathrm{A}_n\} <: \mathsf{loc}\{a_1{:}\mathrm{A}_1, ..., a_k{:}\mathrm{A}_k\}$$

We develop the typing system in stages. In Section 4, we present a *simple* typing system in which subtyping applies to locations, but not channels. Using this type system we set up the major results of the paper: subject reduction and type-safety. These results are repeated for subsequent typing systems as well.

In Section 5 we observe that the simple type system, while natural, is overly restrictive. An important aspect of mobile agents is the ability to acquire capabilities from multiple sources. For example, an agent located at ℓ may have a capability at k which allows it to acquire additional capabilities at k. To exercise this right, the agent may spawn a “sub-agent” to go over to k, get the new capabilities, then come back and report. The difficulty is that when the new capabilities are received back at the original agent, they are received with respect to a separate *instance* of the name k. In order to establish subject reduction, the simple type system makes it impossible to use in concert capabilities acquired on different instances of a location. Some examples which require this extra expressiveness are given in Section 3. To overcome this limitation, we weaken the simple type system by allowing capabilities to be *merged* from different instances of a location name using a match (or equality) operator “$\mathsf{if}\ z = k\ \mathsf{then}\ P$”. Crucial to the new type system and to the proof of its soundness is the fact that the subtyping relation is *bounded complete*, *i.e.* whenever two types have a common subtype they have a *greatest* common subtype.

In Section 6 we extend the improved type system to a language with *channel subtyping*, based on read and write capabilities. The extended type system is based on that of Pierce and Sangiorgi [26], who first studied channel subtyping for the π-calculus. Pierce and Sangiorgi’s definition of subtyping, however, is not bounded complete. To rectify this, we use a type language and subtyping relation which generalize those of [26]. In this section we also augment location types with explicit capabilities for channel creation and agent movement.

The paper concludes with a discussion of related work and open issues.

2. THE LANGUAGE

In this section we describe Dπ, defining many auxiliary notations that are used throughout the paper. Before describing the syntax and reduction semantics, we first present an example which gives an overview of the features of the language.

A typical Dπ system is the following:

$$\ell[\![P]\!] \mid (\nu_k\, a{:}\mathrm{A})\,(\ell[\![Q]\!] \mid k[\![R]\!])$$

There are three agents running in parallel: $\ell[\![P]\!]$ and $\ell[\![Q]\!]$ running at location ℓ and $k[\![R]\!]$ running at location k. Moreover Q and R share a private channel a, declared at location k. Suppose that $\ell[\![P]\!]$ has the from:

$$\ell[\![b?(x)\,P_1 \mid c?(Y)\,P_2]\!]$$

Figure 1 Syntax

e	$::= k$	Location	u, v, w	$::= \mathsf{bv}$	Base Value
	$\mid a$	Resource		$\mid e$	Name
X, Y	$::= x$	Variable		$\mid x$	Variable
	$\mid X@z$	Located Pattern		$\mid u@w$	Located Value
	$\mid (X_1, .., X_n)$	Tuple		$\mid (u_1, .., u_n)$	Tuple
P, Q, R	$::= \mathsf{stop}$	Termination	M, N	$::= \mathbf{0}$	Empty
	$\mid P \mid Q$	Composition		$\mid M \mid N$	Composition
	$\mid (\nu e{:}\mathrm{E})\, P$	Restriction		$\mid (\nu_k\, e{:}\mathrm{E})\, N$	Restriction
	$\mid \mathsf{go}\, u. P$	Movement		$\mid k[\![P]\!]$	Agent
	$\mid u!\langle v\rangle\, P$	Output			
	$\mid u?(X{:}\mathrm{T})\, P$	Input			
	$\mid *P$	Replication			
	$\mid \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching			

This agent contains two subthreads, which when split will run in parallel. The first subthread awaits input on channel b, whereas the second awaits input on channel c. If agent $k[\![R]\!]$ has the form $k[\![b!\langle d\rangle\, R']\!]$, one might expect that communication could occur between P and R on channel b. This is not the case, however. The two instances of b refer to resources at different locations, even though they have the same name.

To communicate with P, R must first *move* from k to ℓ and then use b at ℓ. We write such an agent as $k[\![\mathsf{go}\,\ell.\, b!\langle d\rangle\, R']\!]$. This term can reduce to $\ell[\![b!\langle d\rangle\, R']\!]$, enabling local communication between P and R. After the communication the system is:

$$\ell[\![P_1\{\!\!\{{}^{d}\!/\!{}_{x}\}\!\!\}]\!] \mid \ell[\![c?(Y)\, P_2]\!] \mid (\nu_k\, a)\,(\ell[\![Q]\!] \mid \ell[\![R']\!])$$

The asynchronous form of this idiom (where R' is stop) is used so frequently that in Section 3 we introduce the notation "$\ell.a!\langle v\rangle$" as shorthand for "$\mathsf{go}\,\ell.\, a!\langle v\rangle\,\mathsf{stop}$."

In order for $k[\![\mathsf{go}\,\ell.\, b!\langle d\rangle\, R']\!]$ to be well-typed, it must be that the name d communicated is also located at ℓ. To enable the communication of non-local names, a different syntax must be used. Suppose that $\ell[\![Q]\!]$ now wishes to send the private name a (located remotely at k) to the agent $\ell[\![c?(Y)\, P_2]\!]$. In this case we must write $\ell[\![Q]\!]$ as $\ell[\![c!\langle a@k\rangle\, Q']\!]$. We motivate this syntactic distinction in our discussion of types on page 5.

2.1. Syntax

The syntax of the language is given in Figure 1. In defining the syntax, we presuppose the existence of a set *Var* of *variables*, ranged over by x–z, and a set *Name* of *names*, ranged over by a–m. Both variables and names are typed; however, since we consider different type systems in the course of the paper, we do not report the syntax of types in Figure 1. The type system used in Sections 4 and 5 is described on page 5. For the moment, suffice it to say that names are assigned *atomic types* ranged over by E, which may be either *channel types*, A–C, or *location types*, K–M, or some predefined basic types $\mathcal{B}$. Variables may additionally be assigned one of the *compound* or *value* types, ranged over by T and S. To improve readability we usually use k–m to range over names of location type and a–c for names of channel type; we use e–g when the type of a name is unimportant. We also routinely drop type annotations when they are not of interest, and use without comment some basic types for values such as integers and booleans.

SYSTEMS, AGENTS AND THREADS. The main syntactic category is that of a *system* N. Intuitively, a system consists of a set of agents running independently in parallel. An *agent* $\ell[\![P]\!]$ is a *located thread*, where a *thread* is simply a term of the thread language, described below. Systems are combined using the *static* combinators of the π-calculus, namely parallel composition $|$ and restriction (νe). We further discuss the form of the restriction operator below, after describing types.

The syntax of threads is similar to that of the synchronous polyadic π-calculus [22], with some small extensions to deal with locations. First, locations are names and thus many of the usual operators of the π-calculus apply to them. In particular, new locations may be created using $(\nu\ell)$, locations may be compared using the conditional, and locations may be communicated using the input and output constructs. Second, we introduce a *go* operator that allows a thread to move from one location to another; for example, the thread $\mathsf{go}\, k.\, P$ must move to location k before continuing to execute P. The move operator is also studied by Amadio [1, 3], who writes "$\mathsf{go}\, k.\, P$" as "spawn(k, P)."

All of the operators but "go" are well known from the π-calculus. These include the static combinators $|$ and (νe), as well as constructs for output $u!\langle v\rangle\, P$, input $u?(X)\, Q$, (mis)matching $\mathsf{if}\ u = v\ \mathsf{then}\ Q\ \mathsf{else}\ R$ and iteration $*P$ (in the literature, iteration is often written $!P$). Communication between agents occurs along *channels*. As discussed in the example at the beginning of this section, communication is purely *local* (unlike [3, 12, 29]) in that agents can only communicate with other agents at the same location, using channels that have been allocated at that location.

In the concrete syntax, "go" has greater binding power than composition. Thus $\mathsf{go}\,\ell.\, P \mid Q$ should be read $(\mathsf{go}\,\ell.\, P) \mid Q$. We adopt several standard abbreviations. For example, we omit trailing occurrences of stop and often denote tuples and other groups using a tilde; *e.g.* we may write $\widetilde{a}$ instead of $(a_1, .., a_n)$ and $(\nu\widetilde{e}{:}\widetilde{\mathrm{E}})\, P$ instead of $(\nu e_1{:}\mathrm{E}_1)\, ..(\nu e_n{:}\mathrm{E}_n)\, P$. We also may write "$\mathsf{if}\ u = v\ \mathsf{then}\ P$" instead of "$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{stop}$" and "$\mathsf{if}\ u \neq v\ \mathsf{then}\ Q$" instead of "$\mathsf{if}\ u = v\ \mathsf{then}\ \mathsf{stop}\ \mathsf{else}\ Q$."

TYPES, VALUES AND PATTERNS. We view knowledge of channel a at ℓ as a *capability* to use a at ℓ. These capabilities are the basis of *simple location types*, which are defined as follows:

$$\mathrm{K}, \mathrm{L}, \mathrm{M} ::= \mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}\}, \quad a_i \text{ distinct}$$

A location type is simply a set of capabilities of the form $a{:}\mathrm{A}$, where no two capabilities refer to the same channel. We identify types up to reordering of capabilities and drop empty capability sets, writing "loc" instead of "$\mathsf{loc}\{\}$". We also write "$\mathrm{L}, a{:}\mathrm{A}$" for the extension of the location type L with the new channel a at type A. Thus $\mathsf{loc}\{a{:}\mathrm{A}, a{:}\mathrm{A}'\}$ is not a valid type, and if $\mathrm{L} = \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}$ then $\mathrm{L} = \mathsf{loc}\{b{:}\mathrm{B}, a{:}\mathrm{A}\}$ and $\mathrm{L}, c{:}C = \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}, c{:}\mathrm{C}\}$.

To ensure that well typing is preserved by reduction, we will require that agents receive data at the type intended. The "intended" type is negotiated by typing the channel upon which the data is communicated: sender and receiver must agree on the type of the communication channel. Thus channel types have the form

$$\mathrm{A}, \mathrm{B}, \mathrm{C} ::= \mathsf{res}\langle \mathrm{T}\rangle$$

where T is the type of data transmitted over the channel. A typical example might be $\mathsf{res}\langle \mathrm{L}, \mathrm{K}\rangle$, the type of a channel which can transmit a pair of objects, the objects having the types L and K respectively. We write $\mathsf{obj}(\mathrm{A})$ to denote the transmission type, or *object type* used in the channel type A, *i.e.* $\mathsf{obj}(\mathsf{res}\langle \mathrm{T}\rangle) = \mathrm{T}$. For simplicity, we do not allow recursive types (but see our comments in the conclusion of the paper).

It remains only to define the value types T. There are two basic forms of values: channels and locations. When a channel is communicated, it is assumed to be local; when a location is communicated, nothing is assumed about its location (*i.e.*, it could be the current location or not). One choice, then, would be to allow values that are tuples of channels and locations, *i.e.* of type $\widetilde{\mathrm{E}}$, where $\mathrm{E} ::= \mathrm{A} \mid \mathrm{K}$. In many cases, however, these types are not sufficiently expressive.

For example, consider a remote procedure call in which a thread at ℓ sends a request to a procedure at k, then waits to receive a reply. Using this type system, the example might be written:

$$\ell[\![\mathsf{go}\, k.\, a!\langle \ell\rangle \mid r?(X)\, Q]\!] \quad \mid \quad k[\![a?(z{:}\mathsf{loc}\{r{:}\mathrm{C}\})\, \mathsf{go}\, z.\, r!\langle v\rangle]\!]$$

Here the channel r is a *response channel* which the thread at k uses to reply to the request. Note that r must be globally known to be available at ℓ. This strategy breaks down, however, if there are many concurrent requests to the same remote procedure. In this case, one would like to be able to create a fresh response channel for each request. One attempt is the following:

$$\ell[\![(\nu r)\, \mathsf{go}\, k.\, a!\langle \ell, r\rangle \mid r?(X)\, Q]\!] \quad \mid \quad k[\![a?(z{:}\mathsf{loc}, x{:}\mathrm{B})\, \mathsf{go}\, z.\, x!\langle v\rangle]\!]$$

However, here we have violated the principle that channel communication is local; *i.e.* channel r, when communicated at k, is not local to k, but rather to ℓ.

The crucial link missing is the *dependence* of r on ℓ. To express this dependence, we write the communicated value as $r@\ell$ rather than (ℓ, r). The example can now be written as

$$\ell[\![(\nu r)\, \mathsf{go}\, k.\, a!\langle r@\ell\rangle \mid r?(X)\, Q]\!] \quad \mid \quad k[\![a?(x@z{:}\mathrm{B}@\mathsf{loc})\, \mathsf{go}\, z.\, x!\langle v\rangle]\!]$$

In standard terminology, we have introduced a form of *existential* types, along with constructors and destructors for those types. Thus, one can read the type "$x@z : \mathrm{B}@\mathsf{loc}$" as "$z : \exists x.\mathsf{loc}\{x{:}\mathrm{B}\}$".

In summary, value types are of the form:

$$\mathrm{T} ::= \mathrm{A} \mid \widetilde{\mathrm{A}}@\mathrm{K} \mid \widetilde{\mathrm{T}} \mid \mathcal{B}$$

where $\mathcal{B}$ represents some set of base types. The complete definition of allowed types is given in Figure 4. In value types we sometimes use K as a shorthand for the type $()@\mathrm{K}$. We use the term *existential location type* to refer to value types of the form $(\mathrm{A}_1, .., \mathrm{A}_n)@\mathrm{K}$, particularly if $n > 0$. By contrast, the term *simple location type* is reserved for types of the form K. The syntax of *values* and *patterns*, given in Figure 1, follows the structure of value types, providing a constructor and destructor for every type. Following our shorthand notation for these types, in values and patterns, we treat $u{:}\mathrm{K}$ as shorthand for $()@u{:}()@\mathrm{K}$.

NAME CREATION. There are two forms of private name creation in threads, for channels and locations respectively, but in an effort to keep the syntax compact these are represented

Figure 2 Reduction Relation

$$
\begin{array}{lrcll}
(\text{R-GO}) & \ell[\![\mathsf{go}\,k.P]\!] & \longrightarrow & k[\![P]\!] & \\
(\text{R-COMM}) & \ell[\![a!\langle v\rangle\,P]\!] \mid \ell[\![a?(X)\,Q]\!] & \longrightarrow & \ell[\![P]\!] \mid \ell[\![Q\{\!|{}^{v}\!/\!{}_{X}|\!\}]\!] & \\
(\text{R-EQ}_1) & \ell[\![\mathsf{if}\ u = u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] & \longrightarrow & \ell[\![P]\!] & \\
(\text{R-EQ}_2) & \ell[\![\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] & \longrightarrow & \ell[\![Q]\!] & \text{if } u \neq v
\end{array}
$$

$$
(\text{R-STR})\ \frac{N \longrightarrow N'}{(\nu e)\,N \longrightarrow (\nu e)\,N'} \qquad \frac{N \longrightarrow N'}{M \mid N \longrightarrow M \mid N'} \qquad \frac{N \equiv M \quad M \longrightarrow M' \quad M' \equiv N'}{N \longrightarrow N'}
$$

by the same syntax, $(\nu e{:}\mathrm{E})\,P$. (We use the terms "name creation" and "name restriction" interchangeably.) The thread $(\nu e{:}\mathrm{E})\,P$ creates a new private name e of type E, called the *declaration type* of e, and then executes the thread P. If E is a channel type then e is considered a channel co-located with the thread, whereas if it is a location type K then it is considered a new location. This name creation is handled in exactly the same way as name creation in the π-calculus; use of e is restricted to P, although during execution, P may *enlarge* the scope of the restriction by outputting e.

BINDERS AND SUBSTITUTION. We assume the standard notion of *free* and *bound* occurrences of variables and names in systems and threads. Variables are bound by the input construct, whereas names are bound by name creation. A term with no free variables is *closed*. We write $\mathrm{fn}(N)$ for the function which returns the set of free names occurring in N.

Note that channel names can appear in types and therefore they must be taken into account in the definition of free names. So, for example the free names of $c?(X{:}\mathrm{T})\,P$ include the names which appear in T.

We also assume the standard notions of *alpha-equivalence* and *substitution*, where $N\{\!|{}^{u}\!/\!{}_{x}|\!\}$ denotes the capture-avoiding substitution of u for x in N. The notation $N\{\!|{}^{v}\!/\!{}_{X}|\!\}$ generalizes this in an obvious way as a sequence of substitutions. For $\{\!|{}^{v}\!/\!{}_{X}|\!\}$ to be well-defined, it must be that the structure of the X matches the structure of v. No special provision is necessary for location values such as $\widetilde{a}@\ell$; in substitution these are treated as simple tuples, *e.g.* $N\{\!|{}^{a@\ell}\!/\!{}_{x@z}|\!\} = N\{\!|{}^{\ell}\!/\!{}_{z}|\!\}\{\!|{}^{a}\!/\!{}_{x}|\!\}$. Note that we have been careful to define location types over channel *names* rather than channel *identifiers*, so that substitution need not occur in types. This prevents one from writing, for example, $a?(x@z)\,b?(y@w)\,(\nu\ell{:}\mathsf{loc}\{x,y\})$, which clearly does not make sense.

In the sequel we identify terms up to alpha-equivalence.

2.2. Reduction Semantics

The reduction semantics, given in Figure 2, is defined as a reduction relation between systems; thus judgments are of the form

$$N \longrightarrow N'$$

where N and N' are (closed) system terms. Most of the rules are familiar from the π-calculus, with a few changes to accommodate the fact that agents are explicitly located.

Figure 3 Structural Equivalence

$$\begin{array}{rrcll}
(\textsc{s-extr}) & M \mid (\nu_k\, e{:}\mathrm{E})\, N & \equiv & (\nu_k\, e{:}\mathrm{E})\,(M \mid N) & \text{if } e \notin \mathrm{fn}(M) \\
(\textsc{s-copy}) & \ell[\![*P]\!] & \equiv & \ell[\![P]\!] \mid \ell[\![*P]\!] & \\
(\textsc{s-new}) & \ell[\![(\nu e{:}\mathrm{E})\, P]\!] & \equiv & (\nu_\ell\, e{:}\mathrm{E})\, \ell[\![P]\!] & \text{if } e \neq \ell \\
(\textsc{s-split}) & \ell[\![P \mid Q]\!] & \equiv & \ell[\![P]\!] \mid \ell[\![Q]\!] & \\
(\textsc{s-garbage}_1) & (\nu_k\, e{:}\mathrm{E})\, \mathbf{0} & \equiv & \mathbf{0} & \\
(\textsc{s-garbage}_2) & \ell[\![\mathsf{stop}]\!] & \equiv & \mathbf{0} &
\end{array}$$

The main new rule is that for code movement, $(\textsc{r-go})$, which allows an agent to move from one location to another, say from ℓ to k. In the semantics this is represented by termination of the thread at ℓ and the initiation of a new thread at k: $\ell[\![\mathsf{go}\, k.\, P]\!] \longrightarrow k[\![P]\!]$. Note, however, that P carries with it to k all of the capabilities that were acquired by the original agent (via substitution of names for variables).

The rule $(\textsc{r-comm})$ for communication allows two agents running at the same location ℓ to exchange a value v along a common channel a:

$$\ell[\![a!\langle v\rangle\, P]\!] \mid \ell[\![a?(X)\, Q]\!] \longrightarrow \ell[\![P]\!] \mid \ell[\![Q\{\!\!\{v/X\}\!\!\}]\!]$$

It is worth emphasizing that the agents must be co-located for communication to occur. As discussed at the beginning of this section, agents that wish to communicate on a remote channel must first move to the remote location using the asynchronous "go" operation. If an agent does not wish to move, it may spawn a new thread which "splits" from the agent and then performs the desired move/communication. Information thus acquired may be returned to the primary agent later via communication. We discuss this further in Example 3.4 of Section 3.

The *structural equivalence* [4, 22], defined in Figure 3, relates closed systems ($N \equiv M$). The purpose of the structural equivalence is to abstract from the static structure of terms, *i.e.* from the irrelevant details of the syntactic relation between composition ($P \mid Q$), restriction ($(\nu e)\, P$) and location ($\ell[\![P]\!]$). The structural equivalence is defined to be the least equivalence relation that is closed under composition and restriction, satisfies the monoid laws for composition,[1] and satisfies the axioms given in Figure 3.

In addition to the standard axiom for name extrusion $(\textsc{s-extr})$, the structural equivalence includes axioms that allow restriction and composition to be lifted from threads to systems. The most important of these is the rule $(\textsc{s-split})$ which allows an agent to split into two independent agents ($\ell[\![N \mid M]\!] \equiv \ell[\![N]\!] \mid \ell[\![M]\!]$). The rule $(\textsc{s-nil})$ allows for garbage collection of terminated agents, whereas $(\textsc{s-itr})$ provides a standard interpretation of iteration. Note that when a name is extracted from a thread using $(\textsc{s-new})$ ($\ell[\![(\nu e{:}\mathrm{E})\, P]\!] \equiv (\nu_\ell\, e{:}\mathrm{E})\, \ell[\![P]\!]$) it is necessary to record in the "global" restriction the location ℓ at which the name was defined. In fact this will only have effect when E is a channel type.

[1] The monoid laws are: $N \mid \mathbf{0} \equiv N$, $N \mid M \equiv M \mid N$, and $N \mid (M \mid O) \equiv (N \mid M) \mid O$.

3. EXAMPLES

In order to simplify the presentation of examples, we will assume a set of basic datatypes such as integers and adopt a few notational conventions. In particular, we will define threads using the notation

$$\mathrm{X}(h, \widetilde{y}) \Leftarrow P$$

where the first parameter h stands for the initial location, or *home*, of the thread, and $\widetilde{y}$ an aribtrary set of further parameters. This allows us to write $\underline{\mathbf{X}}(\ell, \widetilde{v})$ as shorthand for the agent $\ell[\![\mathrm{X}(\ell, \widetilde{v})]\!]$, *i.e.* the agent with code $\mathrm{X}(\ell, \widetilde{v})$ running at location ℓ. We will also write $\ell.a!\langle v\rangle\, P$ for the message-sending thread $\mathsf{go}\,\ell.\, a!\langle v\rangle\, \mathsf{stop} \mid P$; note that the "continuation" P is actually asynchronous with respect to the sending of the message.

EXAMPLE 3.1 (A COUNTER). As a first example, we present a simple counter which uses the global names up, dn and rd:

$$\begin{aligned}\mathrm{Count}(h, n) \Leftarrow (\nu s{:}\mathsf{int})\, s!\langle n\rangle &\mid *up?(y@z)\ s?(x)\ (s!\langle x+1\rangle \mid z.y!\langle\rangle)\\ &\mid *dn?(y@z)\ s?(x)\ (s!\langle x-1\rangle \mid z.y!\langle\rangle)\\ &\mid *rd?(y@z)\ s?(x)\ (s!\langle x\rangle \quad\ \ \mid z.y!\langle x\rangle)\end{aligned}$$

This describes a counter, running at location h, and initialized to the value n. The value of the counter is stored using a private channel s. There are three public channels (or methods): up for incrementing the stored value, dn for decrementing it, and rd for retrieving it. The definition of the methods follows the style typical of the asynchronous π-calculus: to invoke a method, a user creates a response channel r, sends this response channel as argument to the method call, then awaits an answer on r. For example, the rd method executes as follows. Upon receiving a rd request, the method reads the local state $s?(x)$ (thus locking the object), then re-stores the state (releasing the lock) and sending a response to the user with the value read. For the methods up and dn, no value is returned, and thus the response simply indicates that the operation is complete.

Suppose that the counter is located at location cnt. Then a typical user might be coded as follows:

$$\begin{aligned}\mathrm{U}(h, cnt) \Leftarrow (\nu r)\ & cnt.up!\langle r@h\rangle\\ & r?()\ cnt.up!\langle r@h\rangle\\ &\quad r?()\ (\nu t)\ cnt.rd!\langle t@h\rangle\\ &\qquad\qquad t?(x)\, out!\langle x\rangle\, U'\end{aligned}$$

The user increments the counter twice, then reads its value, reporting the result on the channel out located at h. We write the combined system as

$$\underline{\mathbf{U}}(k, cnt) \mid \underline{\mathbf{Count}}(cnt, 0)$$

which is shorthand for:

$$k[\![\mathrm{U}(k, cnt)]\!] \mid cnt[\![\mathrm{Count}(cnt, 0)]\!]$$

After a certain amount of reduction the user agent of this system will be able to perform the action $out!\langle 2\rangle$ at location k. □

EXAMPLE 3.2 (A COUNTER SERVER). A *counter server*, cS, for generating new counters can be defined as

$$\mathrm{cS}(h) \Leftarrow *req?(x@z)\ (\nu cnt{:}\mathrm{L_c})\ z.x!\langle cnt\rangle \mid \mathsf{go}\, cnt.\mathrm{Count}(cnt,0)$$

where $\mathrm{L_c} = \mathsf{loc}\{up{:}\mathrm{A_{up}}, dn{:}\mathrm{A_{dn}}, rd{:}\mathrm{A_{rd}}\}$ is the appropriate type for a Counter. Upon receiving a request, the server creates a new counter location cnt, spawns the counter code at that location, initialized to 0, and then sends the name of the counter location to the user. If the server is at location $serv$, then a client would take the form:

$$\mathrm{cU}_i(h) \Leftarrow (\nu r)\ serv.req!\langle r@h\rangle \mid r?(z{:}\mathrm{L_c})\ \mathrm{U}_i(h,z)$$

Now consider two clients running in parallel with the server:

$$\underline{\mathbf{cU}}_1(k_1) \mid \underline{\mathbf{cS}}(serv) \mid \underline{\mathbf{cU}}_2(k_2)$$

After a request from each of the clients to the server the system can evolve to the following state:

$$\begin{aligned} \underline{\mathbf{cS}}(serv) \mid (\nu cnt_1)\,\big(\underline{\mathbf{U}}_1(k_1, cnt_1) \mid \underline{\mathbf{Count}}(cnt_1,0)\big) \\ \mid (\nu cnt_2)\,\big(\underline{\mathbf{U}}_2(k_2, cnt_2) \mid \underline{\mathbf{Count}}(cnt_2,0)\big) \end{aligned} \qquad (*)$$

Here each of the clients cU_i has a copy of the counter running at a private location cnt_i. Note that it would be an error if a user, say U_1, were of the form:

$$\mathrm{U}_1(h,z) \Leftarrow z.compute\_prime\_factors!\langle 10203\rangle$$

Such a user should be considered to be erroneous because the method $compute\_prime\_factors$ is not declared at the counter; only calls to the declared methods up, dn, rd are allowed.

In addition, the use of types in the language allow users to manage access to their private counters, *e.g.* sending the counter to other agents with only the rd capability. Moreover these capabilities can be managed independently by the two clients. U_1, for example, may send to associates the name cnt_1 with the only method rd, while U_2 may, more trustingly, send cnt_2 with both methods rd and up. This selective distribution of knowledge is accomplished by *typed* channels, *i.e.* channels on which values of a restricted type may be transmitted.

To explain this, at least informally, suppose there are two additional agents in the system presented in (*). Call these F_1 and F_2 as they are "friends" of U_1 and U_2 respectively. In addition, suppose that each friend F_i has a channel a_i of type $\mathsf{res}\langle \mathrm{L}_i\rangle$, where:

$$\begin{aligned} \mathrm{L}_1 &= \mathsf{loc}\{rd{:}\mathrm{A_{rd}}\} \\ \mathrm{L}_2 &= \mathsf{loc}\{rd{:}\mathrm{A_{rd}}, up{:}\mathrm{A_{up}}\} \end{aligned}$$

Thus each channel a_i is constrained to transmit values of type L_i. So when a location name is transmitted on a_1, only the permission to use method rd at that location is granted, whereas a_2 also confers permission to use method up. Now consider the system

$$N \mid \underline{\mathbf{F}}_1(k_1) \mid \underline{\mathbf{F}}_2(k_2)$$

where N is the system from (*), and users and their friends have the form:

$$\begin{aligned} \mathrm{U}_i(h, cnt) &\Leftarrow a_i!\langle cnt\rangle \\ \mathrm{F}_i(h) &\Leftarrow a_i?(z{:}\mathrm{L}_i)\ (\nu r)\ z.up!\langle r@h\rangle \end{aligned}$$

Then, after receiving the counter, F_1 should be forbidden from proceeding, as it attempts to access the up method of cnt_1 without sufficient permission. F_2, instead, should be allowed to proceed, as its use of up is justified by the permissions it received for cnt_2.

The general philosophy is that *an agent should only use names, or resources, in accordance with the capabilities/permissions that it has acquired for that name*. In the next section we consider two methods for imposing such constraints. The first is a type system which will reject the agent F_1 as untypable. The second is an augmented semantics which will report a *runtime error* when F_1 attempts to access the up method of cnt_1. □

EXAMPLE 3.3 (A LOCAL COUNTER SERVER). It is also possible for the server to start the counter code within a location specified by the user, rather than creating a new location for this purpose. In this case, however, conflicts on the use of the names up, etc. arise if many counters are created at the same location. To solve this problem, the server may create fresh names for the methods of the cell, rather than using the common names up, dn and rd. The following code implements such a *local counter server* and client:

$$\begin{aligned} \mathrm{cS}'(h) &\Leftarrow *req?(x@z)\,\mathsf{go}\,z.(\nu u,d,r)\ \ \mathrm{Count}_{u,d,r}(z,0) \\ &\qquad\qquad\qquad\qquad\qquad\quad \mid x!\langle u,d,r\rangle \\ \mathrm{cU}'(h) &\Leftarrow (\nu r)\ \ serv.req!\langle r@h\rangle \\ &\qquad\quad \mid r?(u,d,r)\,\mathrm{U}'(h,u,d,r) \end{aligned}$$

In this implementation, the user receives as a response not the name of a freshly allocated location, but rather the names of three freshly allocated channels. Here we use $\mathrm{Count}_{u,d,r}$ to denote the code used in Example 3.1, with the names up, dn, rd, replaced by u, d, r respectively. Now let us consider the system consisting of the server and two co-located clients.

$$\underline{\mathbf{cU}'}(k) \mid \underline{\mathbf{cS}'}(serv) \mid \underline{\mathbf{cU}'}(k)$$

After servicing requests from both clients this can evolve to the following:

$$\begin{aligned} \underline{\mathbf{cS}'}(serv) \mid\ &(\nu u_1,d_1,r_1)\big(\underline{\mathbf{U}}'(k,u_1,d_1,r_1) \mid \underline{\mathbf{Count}}_{u_1,d_1,r_1}(k,0)\big) \\ \mid\ &(\nu u_2,d_2,r_2)\big(\underline{\mathbf{U}}'(k,u_2,d_2,r_2) \mid \underline{\mathbf{Count}}_{u_2,d_2,r_2}(k,0)\big) \end{aligned}$$

Here all of the agents, counters and users, are located at a single location k. Each user has private access to its counter, however, due to the use of the restriction operator. □

EXAMPLE 3.4 (REMOTE CHANNEL CREATION). We now examine an implementation issue. Dπ as defined allows an agent to create new channels only at its current location. It is often useful, however, to be able to create channels remotely, without having to move back and forth. First we consider the special case in which an agent creates a new location, and would like to place some channels at that location at the same time. We might introduce a new thread notation for this, $\mathsf{T}(h) \Leftarrow (\nu(a,b)@\ell)\,P$, which we take as shorthand for:

$$\mathsf{T}(h) \Leftarrow (\nu\ell)\ \mathsf{go}\,\ell.(\nu a,b)\ \mathsf{go}\,h.P$$

However this is best viewed as a specification; as an implementation method it involves the transmission of the continuation thread P, which may be huge, around the network. A more useful implementation would be:

$$\begin{aligned}\mathsf{T}(h) \Leftarrow (\nu r)\ (\nu\ell)\ \mathsf{go}\,\ell.(\nu(a,b))\ \mathsf{go}\,h.\,r!\langle(a,b)@\ell\rangle \\ \mid\ r?((x,y)@z)\,P\{\!|^{(x,y)@z}/_{(a,b)@\ell}|\!\}\end{aligned}$$

Here the new location ℓ with associated channels are created, and then sent back to the main agent at h; the sending of the potentially large continuation P across the network is avoided.

This implementation schema can also be adapted to the generation of new channels at an existing location, say ℓ, simply by removing the restriction $(\nu\ell)$:

$$\mathsf{T}(h) \Leftarrow \mathsf{go}\,\ell.(\nu a,b)\ \mathsf{go}\,h.P$$

We use the notation $(\nu_\ell(a,b))\,P$ to abbreviate this idiom. A lightweight implementation then might be:

$$\begin{aligned}\mathsf{T}(h) \Leftarrow (\nu r)\ \mathsf{go}\,\ell.(\nu a,b)\ \mathsf{go}\,h.\,r!\langle(a,b)@\ell\rangle \\ \mid\ r?((x,y)@z)\,\mathsf{if}\ z=\ell\ \mathsf{then}\ P\{\!|^{(x,y)@z}/_{(a,b)@\ell}|\!\}\end{aligned}$$

This example raises some interesting issues concerning the type system which are discussed at length in Section 5. ☐

EXAMPLE 3.5 (ROUTED FORWARDING). Finally, we develop a longer program. We write a program $\mathit{Forwarder}(\mathit{in}@h, s@d)$ which establishes a connection between the local channel in and the (possibly remote) channel s. By "connection" we mean that messages sent into in should eventually find their way to the service channel s at destination d. Such a program is trivial to write in Dπ:

$$*\mathit{in}?(x)\ \mathsf{go}\,d.\,s!\langle x\rangle$$

The unpleasant part of the problem specification is that we are not allowed to assume that there is a direct connection from the current location to d. Instead, the program must consult the local method $\mathit{route}(d)$ which returns the name of the neighboring location that is closest to d, *i.e.* somewhere between the current location and d.

To make the program readable, we assume some additional syntactic conventions, including recursive definitions, let-expressions and the notation for remote channel creation introduced in the previous example. The $\mathit{Forwarder}$ can be implemented as follows:

$$\begin{aligned}\mathit{Forwarder}(\mathit{in}@h, s@d)\ \Leftarrow\ & \mathsf{if}\ h=d\ \mathsf{then}\\ & \quad *\mathit{in}?(x)\ s!\langle x\rangle\\ & \mathsf{else}\\ & \quad \mathsf{let}\ n \leftarrow \mathit{route}(d)\\ & \quad \mathsf{in}\ \ (\nu_n c)\ \mathsf{go}\,n.\,\mathit{Forwarder}(c@n, s@d)\\ & \qquad\qquad \mid *\mathit{in}?(x)\ \mathsf{go}\,n.\,c!\langle x\rangle\\ & \mathsf{endif}\end{aligned}$$

When the $\mathit{Forwarder}$ is started, it checks to see if the destination d is the same as the current location h. If h and d are the same, then there is no need for routing, and the program can simply set up a forwarding process from in to s: "$*in?(x)\ s!\langle x\rangle$". If h and d are different, then the name of a neighbor n is retrieved, where n is between h and d on the network. Then a new copy of the code is started at n, and a forward process is set up between in and n. □

4. TYPES

In this section we define a typing relation for the language presented in Section 2 and show that it is sound. To prove soundness, one normally proves two properties: subject reduction and type safety. *Subject reduction* says simply that well-typedness is preserved by reduction; *i.e.* if N is well typed and $N \longrightarrow N'$ then N' is also well typed. Intuitively, *type safety* asserts that a well typed term "does nothing bad"; combined with subject reduction it guarantees that a term can *never* do the "bad" thing. What exactly is "bad" varies from one language to another. In the lambda calculus, the bad thing may be to reach an irreducible form that is not canonical; thus the type safety theorem states that if a term is well typed, then either it is canonical or it can reduce.

In reactive languages which lack such canonical forms, such as the polyadic π-calculus, the statement of type safety is more delicate. Milner [22] describes type safety as freedom from *arity mismatches*. For example, the system

$$\ell[\![c!\langle a, b\rangle]\!] \mid \ell[\![c?(z{:}\mathsf{loc})\,\mathsf{go}\,z.\,Q]\!]$$

gives rise to a runtime error because the first thread sends a pair of channels, whereas the second expects a singleton location. This definition of type safety is related to that for the lambda calculus: arity matching is required for substitution (and therefore the reduction rule (R-COMM)) to be defined.

Type safety for π-calculi with capabilities was first studied by Pierce and Sangiorgi [26]; we presented an alternative formulation in [27], which we now recount. The basic idea is that every instance of a name is tagged with certain capabilities and each instance may only be used as its capabilities allow; attempts to use a name without the proper capability result in runtime error. For the simple type language of Section 2.1, only locations need be tagged, and each instance of a location must be tagged with the set of channel names available to that instance. For example, the name $\ell_{\{a,b\}}$ is an instance of location ℓ at which only the channels a and b may be used. Thus, the term $\ell_{\{a,b\}}[\![c!\langle a\rangle]\!]$ would produce a runtime error since c is used at ℓ without permission.

The key to such a tagged semantics is the rule for tagged communication. The rule must capture the fact that when an instance of a name is communicated, the permissions it carries may be reduced. Thus in the reduction

$$\ell_{\{c\}}[\![c!\langle k_{\{a,b\}}\rangle]\!] \mid \ell_{\{c\}}[\![c?(z{:}\mathsf{loc}\{a{:}\mathrm{A}\})\,Q]\!] \longrightarrow \ell_{\{c\}}[\![Q\{\!|{}^{k_{\{a\}}}/z|\!\}]\!]$$

k is received with the capability to use only channel a; *i.e.* the tagged value $k_{\{a\}}$ is substituted for z in Q. The tag for channel b is removed since the receiving thread has not explicitly requested a location with capability b, but rather a location z of type $\mathsf{loc}\{a{:}\mathrm{A}\}$. Thus, if Q has the form $\mathsf{go}\,z.\,b!\langle v\rangle$, then a runtime error would occur when Q attempts to communicate on b.

This form of error is related to errors which occur in untyped languages with object or record types: if method b is requested of an object instance that does not provide method b,

Figure 4 Simple Types

Types:			Subtyping:
LType:	K	$::= \mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}\}$, a_i distinct	$\mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}, \widetilde{b}{:}\widetilde{\mathrm{B}}\} <: \mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}\}$
RType:	A	$::= \mathsf{res}\langle \mathrm{T} \rangle$	
AType:	E	$::= \mathrm{A} \mid \mathrm{K} \mid \mathcal{B}$	$\mathrm{A} <: \mathrm{A}$
VType:	T	$::= \mathrm{A} \mid \mathcal{B}$	$\mathcal{B} <: \mathcal{B}$
		$\mid (\mathrm{A}_1, .., \mathrm{A}_n)@\mathrm{K}$	$\widetilde{\mathrm{A}}@\mathrm{K} <: \widetilde{\mathrm{A}}@\mathrm{L}$ if $\mathrm{K} <: \mathrm{L}$
		$\mid (\mathrm{T}_1, .., \mathrm{T}_n)$	$\widetilde{\mathrm{S}} <: \widetilde{\mathrm{T}}$ if $\forall i{:}\ \mathrm{S}_i <: \mathrm{T}_i$

a runtime error occurs. The tagged system is even more discriminating, however, because it may be an error to request b even at objects which do, in fact, provide b: it is also important that the particular agent requesting b have received the capability (or permission) to use b.

The section proceeds as follows. First, we describe the subtyping relation. Section 4.2 then defines the typing relation and presents subject reduction. The following subsection presents a series of examples displaying the use of types. Section 4.4 describes the tagged language and proves type safety. The tagged language improves on the one sketched above: rather than tag *instances of names* with capabilities, we tag *agents*; thus as agents roam the system, their accumulated capabilities are explicitly recorded.

4.1. Subtyping

Figure 4 recalls the definition of simple types from Section 2.1 and presents the definition of subtyping for these types (it is the least relation that satisfies the five rules given). A *location capability* has the form $a{:}\mathrm{A}$, *i.e.* a channel a of type A. Then we say that L is a *supertype* of K if every capability of L is also a capability of K. We use the obvious notation $a{:}\mathrm{A} \in \mathrm{K}$ to indicate that location type K contains the capability $a{:}\mathrm{A}$. Extending this view of location types as sets of capabilities, subtyping can be expressed simply as reverse inclusion:

$$\mathrm{K} <: \mathrm{L} \text{ if } \mathrm{K} \supseteq \mathrm{L}$$

(Recall that we identify location types up to reordering of capabilities.) As an example, note that $\mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\} <: \mathsf{loc}\{a{:}\mathrm{A}\}$. For simplicity of exposition, in this section there is no non-trivial subtyping on channel types; we study these in Section 6.

It is easy to show that <: forms a preorder on types (*i.e.* <: is reflexive and transitive); in fact it is a partial order, but for the sake of generality we use preorders. This subtyping relation also has an interesting property which will play a crucial role in Section 5: it is *finite bounded complete*, or FBC .

DEFINITION 4.1. We say that a preorder $(\mathbf{S}, \sqsubseteq)$ is *finite bounded complete* (FBC) if for every finite nonempty subset $S \subseteq \mathbf{S}$, if S has a lower bound then S has a greatest lower bound. That is, there exists a *partial meet* operator $\sqcap$ that satisfies the following property. Let $R \in \mathbf{S}$ and $S \subseteq \mathbf{S}$, S finite and nonempty; if for every $s_i \in S$, $R \sqsubseteq s_i$ (*i.e.* R is a lower bound of S) then $\sqcap S$ is defined and:

- $\sqcap S \sqsubseteq s_i$, for all $s_i \in S$ and
- $R \sqsubseteq \sqcap S$. □

To show that a preorder is finite bounded complete it suffices to define a partial binary meet operator $\sqcap$ that is commutative and associative and satisfies the conditions of Definition 4.1 on pairs.[2] The binary operator can then be lifted to nonempty finite sets in the obvious way:

$$\sqcap S = s_1 \sqcap s_2 \sqcap ... \sqcap s_n \quad \text{where } s_i \in S$$

DEFINITION 4.2. We define a partial binary operator $\sqcap$ over the types of Figure 4. The definition is by induction on the structure of types.

$$\begin{array}{rl}
\mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}\} \sqcap \mathsf{loc}\{\widetilde{b}{:}\widetilde{\mathrm{B}}\} & = \mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}, \widetilde{b}{:}\widetilde{\mathrm{B}}\} \text{ if } \forall i,j : a_i = b_j \text{ implies } \mathrm{A}_i = \mathrm{B}_j \\
\mathrm{A} \sqcap \mathrm{A} & = \mathrm{A} \\
\mathcal{B} \sqcap \mathcal{B} & = \mathcal{B} \\
\widetilde{\mathrm{T}} \sqcap \widetilde{\mathrm{S}} & = (\mathrm{T}_1 \sqcap \mathrm{S}_1, ..., \mathrm{T}_n \sqcap \mathrm{S}_n) \\
\widetilde{\mathrm{A}}@\mathrm{K} \sqcap \widetilde{\mathrm{B}}@\mathrm{L} & = (\widetilde{\mathrm{A}} \sqcap \widetilde{\mathrm{B}})@(\mathrm{K} \sqcap \mathrm{L})
\end{array}$$ $\square$

The only non-trivial case in the definition is that for location types. For channel types the meet is undefined except when the types are identical; $\mathrm{A} \sqcap \mathrm{B}$ is undefined if $\mathrm{A} \neq \mathrm{B}$. At other value types it is simply a homomorphic extension, strict in undefinedness. Note that the meet is only defined for types that have the same structure and for which all constituent components have a meet. Thus $(\mathrm{A},\mathrm{B}) \sqcap (\mathrm{A})$ is undefined, as is $\mathsf{loc}\{a{:}\mathrm{A}\} \sqcap \mathsf{loc}\{a{:}\mathrm{B}\}$, if $\mathrm{A} \neq \mathrm{B}$.

PROPOSITION 4.3. *The operator $\sqcap$ defined above is a partial meet operator.*

Proof. By induction on the structure of types (and thus on the definition of $\sqcap$), it is straightforward to establish that $\sqcap$ is commutative, associative and satisfies the constraints of Definition 4.1 for all pairs of types. $\square$

4.2. A Simple Type System

TYPE ENVIRONMENTS. The primary judgments of the type system will be of the form $\Gamma \vdash N$ where Γ is a *type environment* and N is a system term; the judgment $\Gamma \vdash N$ is read "the term N is well-typed with respect to environment Γ." The purpose of the type environment is to provide a type for all of the free identifiers in N. Since the type system is static and therefore must be defined over open terms, type environments must provide types for variables in addition to names. The type environment thus provides a view of every free *identifier*, where the type (indeed the existence) of a channel name or variable depends upon its location. We allow variables to receive values other than simple names; so in addition to channel and location types, a variable may have a tuple type $\widetilde{\mathrm{T}}$, or an existential type $(\mathrm{A}_1, .., \mathrm{A}_n)@\mathrm{L}$ (where n is greater than zero). Given these considerations, we take type environments to be maps from identifiers to *open location types*, which have the form $\mathsf{loc}\{\widetilde{u}{:}\widetilde{\mathrm{T}}\}$. By contrast, the location types of Section 2.1 ($\mathsf{loc}\{\widetilde{a}{:}\widetilde{\mathrm{A}}\}$) are referred to as *closed*. As an example, the following is a type environment:

$$\Delta = \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}'\}\}$$

[2] By commutativity, we mean: If $\mathrm{T} \sqcap \mathrm{S}$ is defined, then $\mathrm{S} \sqcap \mathrm{T}$ is defined and $\mathrm{T} \sqcap \mathrm{S} \lesseqgtr \mathrm{S} \sqcap \mathrm{T}$, where $\lesseqgtr$ is the kernel of <: ($\mathrm{T} \lesseqgtr \mathrm{S}$ iff $\mathrm{T} <: \mathrm{S}$ and $\mathrm{S} <: \mathrm{T}$). Similarly, by associativity we mean: If $(\mathrm{T} \sqcap \mathrm{S}) \sqcap \mathrm{R}$ is defined then $\mathrm{T} \sqcap (\mathrm{S} \sqcap \mathrm{R})$ is defined and $(\mathrm{T} \sqcap \mathrm{S}) \sqcap \mathrm{R} \lesseqgtr \mathrm{T} \sqcap (\mathrm{S} \sqcap \mathrm{R})$.

We write $\Gamma(w)$ to refer to the type of the location w in Γ, and $\Gamma(w,u)$ to refer to the type of the channel or variable u at location w. So for Δ as defined above, $\Delta(z) = \mathsf{loc}\{a{:}\mathrm{A}'\}$ and $\Delta(\ell, x) = \mathrm{B}$, whereas $\Delta(z,x)$ is undefined.

We use the same metavariables (K-M) to range over both open and closed location types. It is important to remember, however, that open types may only appear in type environments; all types in *terms* are *closed*. Thus, substitution of values into terms has no effect on the types that may appear in those terms.

The subtyping relation is extended in the obvious manner from closed to open location types, using *open location capabilities* of the form $u{:}\mathrm{T}$. Both subtyping and the partial meet operator extend pointwise to environments. For subtyping we have:

$$\Gamma <: \Delta \ \text{ iff } \ \forall w \in \mathrm{dom}(\Delta) : \ \Gamma(w) <: \Delta(w)$$

The partial meet operator $\Delta \sqcap \Gamma$ is undefined if $\Delta(w) \sqcap \Gamma(w)$ is undefined for some $w \in \mathrm{dom}(\Delta) \cap \mathrm{dom}(\Gamma)$, otherwise:

$$\begin{aligned} \Delta \sqcap \Gamma = {} & \{w{:}\mathrm{L} \mid \Delta(w) \sqcap \Gamma(w) = \mathrm{L}\} \\ & \cup \ \{w{:}\mathrm{L} \mid \Delta(w) = \mathrm{L} \text{ and } w \notin \mathrm{dom}(\Gamma)\} \\ & \cup \ \{w{:}\mathrm{L} \mid \Gamma(w) = \mathrm{L} \text{ and } w \notin \mathrm{dom}(\Delta)\} \end{aligned}$$

For example, if $\mathrm{K} \sqcap \mathrm{K}'$ is defined then:

$$\{\ell{:}\mathrm{L}, k{:}\mathrm{K}\} \sqcap \{k{:}\mathrm{K}', m{:}\mathrm{M}\} \ = \ \{\ell{:}\mathrm{L}, k{:}\mathrm{K} \sqcap \mathrm{K}', m{:}\mathrm{M}\}$$

ENVIRONMENT EXTENSION. We use $\Gamma,_{(w)}\{w{:}\mathrm{K}\}$ to represent the environment Γ augmented by the new entry which maps the identifier w to the location type K; this is only defined if w is new to Γ, *i.e.* w is not already in the domain of Γ.

We use a similar notation for identifiers at other types: $\Gamma,_{(u)}\{_w\, u{:}\mathrm{T}\}$ augments the type of w in Γ with the new capability $u{:}\mathrm{T}$; to be defined w must already be in the domain of Γ and u must be new to $\Gamma(w)$. For example, taking Δ as defined previously, we have

$$\begin{aligned} \Delta &= \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}'\}\} \\ \Delta,_{(x)}\{_z\, x{:}\mathrm{B}'\} &= \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}', x{:}\mathrm{B}'\}\} \end{aligned}$$

whereas $\Delta,_{(x)}\{_\ell\, x{:}\mathrm{B}'\}$ is undefined. This notation extends structurally to values.[3] For example, consider Δ as above and let $\mathrm{L} = \mathsf{loc}\{a{:}\mathrm{A}''\}$; then we have:

$$\Delta,_{((x,y@w))}\{_z\,(x, y@w){:}(\mathrm{B}', \mathrm{C}@\mathrm{L})\} = \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}', x{:}\mathrm{B}'\}, w{:}\mathsf{loc}\{a{:}\mathrm{A}'', y{:}\mathrm{C}\}\}$$

---

[3] Formally, the definition is as follows.

$$\begin{aligned} \Gamma,_{(u)}\{_w\, u{:}\mathrm{T}\} &= \Gamma \sqcap \{w{:}\mathsf{loc}\{u{:}\mathrm{T}\}\} && \text{if } w \in \mathrm{dom}(\Gamma) \text{ and } u \notin \mathrm{dom}(\Gamma(w)) \\ \Gamma,_{(\widetilde{v})}\{_w\, \widetilde{v}{:}\widetilde{\mathrm{T}}\} &= \Gamma,_{((v_1)}\{_w\,(v_1{:}\mathrm{T}_1)\} \sqcap \cdots,_{((v_n)}\{_w\,(v_n{:}\mathrm{T}_n)\} \\ \Gamma,_{((\widetilde{v}@u))}\{_w\,(\widetilde{v}@u){:}(\widetilde{\mathrm{A}}@\mathrm{K})\} &= \Gamma,_{(u)}\{u{:}\mathrm{K}\},_{(\widetilde{v})}\{_u\, \widetilde{v}{:}\widetilde{\mathrm{A}}\} && \text{if } u \notin \mathrm{dom}(\Gamma) \end{aligned}$$

Note that an environment cannot be extended with a location that is already defined. Also note that for $\Gamma,_{(v)}\{_w\, v{:}\mathrm{T}\}$ to be defined, w must already be defined in Γ. Thus $\Gamma,_{((\widetilde{a}@w))}\{_w\,(\widetilde{a}@w){:}(\widetilde{\mathrm{A}}@\mathrm{K})\}$ is *undefined* for any Γ and w.

In the new environment, z is augmented with the extra capability x:B$'$, and w is introduced as a new location with capabilities a:A$''$ and y:C. By way of contrast, consider:

$$\Delta,_{((x,y))}\{_z(x,y){:}(\mathrm{B}',\mathrm{C@L})\} = \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}, x{:}\mathrm{B}\}, z{:}\mathsf{loc}\{a{:}\mathrm{A}', x{:}\mathrm{B}', y{:}\mathrm{C@L}\}\}$$

Here the existential location value y of type C@L is not fully deconstructed. When studying the type system, note that little can be done with identifiers such as y; in fact they can only be used in output values.

THE TYPING SYSTEM. The typing system is given in Figure 5. The definition uses auxiliary judgments for threads, identifiers and values. For threads, judgments have the form $\Gamma \vdash_w P$, indicating that the thread P is well-typed to run at location w, where $w \in \mathrm{dom}(\Gamma)$. Finally, channels and other values are typed using judgments of the form $\Gamma \vdash_w v{:}\mathrm{T}$, which indicates that the value v is well formed at w and has at least the capabilities specified by T. Recall that in values, we treat u:L as shorthand for $()@u{:}()@\mathrm{L}$. Location types, both simple and existential, are independent of the location w at which they are typed.

The heart of the typing system are the rules for threads, and in particular the rules for communicating terms, (T-W) and (T-R). For example, to deduce that $u!\langle v\rangle\, Q$ is well-typed to run at location w

$$\Gamma \vdash_w u!\langle v\rangle\, Q$$

it is necessary to establish

- $\Gamma \vdash_w v{:}\mathrm{T}$, *i.e.* v is a well formed value at w with capabilities specified by some type T,
- $\Gamma \vdash_w u{:}\mathsf{res}\langle\mathrm{T}\rangle$, *i.e.* u is a channel at location w which may communicate values of type T, and
- $\Gamma \vdash_w Q$, *i.e.* Q is well-typed to run at w.

The input construct is similar. To deduce $\Gamma \vdash_w u?(X{:}\mathrm{T})\, Q$ we must, as before, establish that u is a channel of type $\mathsf{res}\langle\mathrm{T}\rangle$ at location w, but in deducing that Q is well-typed we may use the augmented environment $\Gamma,_{(X)}\{_w X{:}\mathrm{T}\}$.

In the rule for code movement, (T-GO), the location of the thread changes: to type $\Gamma \vdash_w \mathsf{go}\, u.\, P$ one must ensure that P is well typed at u, not w; therefore the premise is $\Gamma \vdash_u P$. The remaining rules for threads are straightforward. The rules for (mis)matching are standard. The rules for name creation (T-NEW) simply augment the typing environment in the appropriate manner. Note that because of our notation for extending environments location types are independent of the location w at which they are typed. The other rules are purely structural.

The extension to systems is also straightforward. The only interesting rule is (N-RUN) for located threads, which has the same structure as (T-GO). The remaining rules are structural rules, similar to those for threads.

PROPERTIES OF TYPING. We now sketch some results related to the typing system of Figure 5. The following property is immediate from the definition of subtyping.

LEMMA 4.4 (TYPE SPECIALIZATION).
If $\Gamma \vdash_w v{:}\mathrm{T}$ *and* $\mathrm{T} <: \mathrm{S}$ *then* $\Gamma \vdash_w v{:}\mathrm{S}$. □

The following Proposition states that well-typing is preserved when the typing environment is augmented; for the proof see the Appendix.

Figure 5 A Type System

Values (rules for base values not shown):

$$\text{(V-UNIV)}\ \frac{\Gamma(u) <: \mathrm{K}}{\Gamma \vdash_w u{:}\mathrm{K}} \qquad \text{(V-LOCAL)}\ \frac{\Gamma(w) <: \mathsf{loc}\{u{:}\mathrm{A}\}}{\Gamma \vdash_w u{:}\mathrm{A}} \qquad \text{(V-DEP)}\ \frac{\Gamma \vdash_w u{:}\mathrm{K} \quad \Gamma \vdash_u \widetilde{v}{:}\widetilde{\mathrm{B}}}{\Gamma \vdash_w \widetilde{v}@u : \widetilde{\mathrm{B}}@\mathrm{K}} \qquad \text{(V-TUP)}\ \frac{\Gamma \vdash_w u_i{:}\mathrm{T}_i \ (\forall i)}{\Gamma \vdash_w \widetilde{u}{:}\widetilde{\mathrm{T}}}$$

Threads:

$$\text{(T-W)}\ \frac{\Gamma \vdash_w u{:}\mathsf{res}\langle\mathrm{T}\rangle \quad \Gamma \vdash_w v{:}\mathrm{T} \quad \Gamma \vdash_w P}{\Gamma \vdash_w u!\langle v\rangle\, P} \qquad \text{(T-R)}\ \frac{\Gamma \vdash_w u{:}\mathsf{res}\langle\mathrm{T}\rangle \quad \mathrm{fv}(X) \text{ disjoint } \mathrm{fv}(\Gamma) \quad \Gamma,_{(X)}\{_w X{:}\mathrm{T}\} \vdash_w Q}{\Gamma \vdash_w u?(X{:}\mathrm{T})\, Q}$$

$$\text{(T-EQ)}\ \frac{\Gamma \vdash_w u{:}\mathrm{T} \quad \Gamma \vdash_w v{:}\mathrm{T} \quad \Gamma \vdash_w P \quad \Gamma \vdash_w Q}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\text{(T-GO)}\ \frac{\Gamma \vdash_w u{:}\mathsf{loc} \quad \Gamma \vdash_u P}{\Gamma \vdash_w \mathsf{go}\, u.\, P} \qquad \text{(T-NEW)}\ \frac{e \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(e)}\{_w e{:}\mathrm{E}\} \vdash_w P}{\Gamma \vdash_w (\nu e{:}\mathrm{E})\, P} \qquad \text{(T-STR)}\ \frac{\Gamma \vdash_w P \quad \Gamma \vdash_w Q}{\Gamma \vdash_w \mathsf{stop},\, P \mid Q,\, {*}P}$$

Networks:

$$\text{(N-RUN)}\ \frac{\Gamma \vdash_k P}{\Gamma \vdash k[\![P]\!]} \qquad \text{(N-NEW)}\ \frac{e \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(e)}\{_k e{:}\mathrm{E}\} \vdash N}{\Gamma \vdash (\nu_k\, e{:}\mathrm{E})\, N} \qquad \text{(N-STR)}\ \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash \mathbf{0},\, M \mid N}$$

PROPOSITION 4.5 (WEAKENING). *If* $\Gamma \vdash N$ *and* $\Delta <: \Gamma$ *then* $\Delta \vdash N$.

In Lemma A.2 of the Appendix we show that the type environment may also be diminished by removing identifiers that do not occur free in the term being typed.

THEOREM 4.6 (SUBJECT REDUCTION).
(a) *If* $N \equiv N'$ *then* $\Gamma \vdash N$ *if and only if* $\Gamma \vdash N'$.
(b) *If* $\Gamma \vdash N$ *and* $N \longrightarrow N'$ *then* $\Gamma \vdash N'$.

As is often the case, the proof of subject reduction depends heavily on a substitution lemma:

LEMMA 4.7 (SUBSTITUTION).
If $\Gamma \vdash_w v{:}\mathrm{T}$ *and* $\Gamma,_{(X)}\{_w X{:}\mathrm{T}\} \vdash_u P$ *then* $\Gamma \vdash_{u\{v/X\}} P\{\!\!\{v/X\}\!\!\}$.

There is no result for substitution in systems since substitution always occurs at the level of threads. The proofs of these results are in the Appendix.

4.3. Examples

We now consider some simple type inferences. As a first example consider the single agent:

$$N = \ell[\![c?(z{:}\mathrm{K})\ \mathsf{go}\, z.\, a!\langle v\rangle]\!]$$

At location ℓ, N receives location z on channel c, then moves to z and calls method a. To be well-typed relative to Γ it is certainly necessary for $\Gamma(\ell)$ to have the form $\mathsf{loc}\{c{:}\mathsf{res}\langle\mathsf{loc}\{a{:}\mathrm{A}_v,...\}\rangle,...\}$, where A_v is the type $\mathsf{res}\langle\mathrm{T}_V\rangle$, and T_V is the type of v. This typing ensures that N does not cause a runtime error. For example, let:

$$M = k[\![\mathsf{go}\,\ell.\,c!\langle k\rangle \mid a?(x)\,Q]\!]$$

If $\Gamma(k)$ has the form $\mathsf{loc}\{a{:}\mathrm{A}_v,...\}$, then the agents N and M can communicate in the combined system $(N \mid M)$. The first sub-agent of M moves to ℓ and communicates with N, the second sub-agent of M waits at k for the response.

Here there is an *a priori* agreement between the two agents that any location transmitted on c will have a publicly available channel a. It is often desirable to generate new channels, as described in Example 3.3. Such an agent is the following M':

$$M' = k[\![(\nu b{:}\mathrm{A}_v)\ \mathsf{go}\,\ell.\,c!\langle b@k\rangle \mid b?(x)\,Q]\!]$$

Here a new channel b of the appropriate type is generated and the structured value $b@k$ is transmitted. The corresponding version of the first thread is

$$N' = \ell[\![c?(x@z{:}\mathrm{A}_v@\mathsf{loc})\ \mathsf{go}\,z.\,x!\langle v\rangle]\!]$$

Rather than communicating over the public channel a, the threads use the private channel b, which is bound to x in the receiver. The use of structured values (or existential types, if you prefer) is essential to get N' to be well typed; the analysis of the thread $\mathsf{go}\,z.\,x!\langle v\rangle$ is performed in an environment in which the identifier z is known to contain the capability $x{:}\mathrm{A}_v$. Note that these agents may even be well typed in an environment where k has no known capabilities (*i.e.* $\Gamma(k) = \mathsf{loc}$).

A mixture of public and private channels is possible by combining the mechanisms of the previous two examples; for example:

$$\begin{aligned} N'' &= \ell[\![c?(x@z{:}\mathrm{T}_z)\ \mathsf{go}\,z.\,(x!\langle v\rangle \mid a?(y)\,P)]\!] \\ M'' &= k[\![(\nu b{:}\mathrm{A}_v)\ \mathsf{go}\,\ell.\,c!\langle b@k\rangle \mid b?(x)\ a!\langle x\rangle\,Q]\!] \end{aligned}$$

Suppose that $\mathrm{T}_z = \mathrm{A}_v@\mathsf{loc}\{a{:}\mathrm{A}_v\}$. Again a new channel b is generated and the value v is bounced back and forth at k using b and the public channel a.

We finish with some examples of systems that cannot be typed. The simplest case is the misuse of a channel. The system

$$\ell[\![c?(x)\ \mathsf{go}\,x.\,P]\!] \mid k[\![\mathsf{go}\,\ell.\,c!\langle a,b\rangle\,Q]\!]$$

is not typable in any typing environment. The two agents make inconsistent requirements on the type of the channel c at the location ℓ. The first demands a type of the form $\mathsf{res}\langle\mathsf{loc}\rangle$ while the second requires a type of the form $\mathsf{res}\langle\mathrm{T}_a,\mathrm{T}_b\rangle$.

A more interesting example is the following:

$$\ell[\![c?(x@z)\ (\nu a)\ d!\langle x, a@z\rangle]\!]$$

This agent inputs a location z with channel x, then allocates a channel at ℓ and then attempts to send z with the channels x and a. This term is clearly unsound when a location other than ℓ is sent on c; *i.e.* channel a is not known to be available at z. A similar problem occurs in the term:

$$\ell[\![c?(x@z)\ \mathsf{if}\ x = a\ \mathsf{then}\ Q']\!]$$

This term cannot be typed because the matching rule (T-EQ) requires that the two identifiers being compared (x and a) be located at ℓ, and x is not known to be located at ℓ.

Finally, suppose Γ is a type environment in which $\Gamma(k, c) = \mathsf{res}\langle \mathsf{loc}\{a{:}\mathrm{A}\}\rangle$. Then the agent $k[\![c?(z)\ \mathsf{go}\, z.\, b!\langle v\rangle]\!]$ cannot be typed relative to Γ. To do so it would be necessary to type $b!\langle v\rangle$ at location $z{:}\mathsf{loc}\{a{:}\mathrm{A}\}$.

4.4. The Tagged Language and Type Safety

We now formalize a suitable notion of runtime error for our language and prove that well-typed systems are free of such errors. To do so, the language must be enriched with permissions; a runtime error occurs, then, when a name is used without permission. In [27] we defined such an enriched language by placing tags on every instance of a name. Here we take an alternative approach, which we feel is conceptually and notationally simpler. Rather than tag every instance of a name, we tag threads; thus as a thread evolves, its accumulated capabilities are explicitly recorded. A runtime error occurs if a thread attempts to use a name contrary to the limitations imposed by these explicit capabilities.

The syntax and semantics of the tagged language are given in Figure 6. The syntax of threads and values is unchanged from that of Figure 1; only the system level is affected, and here only the clause for agents. Each agent of the original language $\ell[\![P]\!]$ is tagged with a *closed* type environment Γ which represents the capabilities (or permissions) of the agent. For example, the agent

$$\ell[\![P]\!]_{\{\ell{:}\mathsf{loc}\{a{:}\mathrm{A},\, b{:}\mathrm{B}\},\, k{:}\mathsf{loc}\{a{:}\mathrm{A}'\}\}}$$

has knowledge of resources a and b at ℓ and of resource a at k. In addition to recording the *names* of available resources, the tag also records the permissions that the agent has acquired for the *use of that resource* (the types A, B and A′). This additional information allows fine control in the definition of runtime error.

The reduction semantics of Section 2.2 is adapted to show how tags evolve over time. To avoid confusion, we write $N \longmapsto N'$ for tagged reduction. The only non-trivial change is to the rule (R-COMM). Before discussing it, we briefly describe the changes to the other rules. First consider the structural equivalence reported in Figure 6. In the rules ($\mathrm{S_T}$-SPLIT) and ($\mathrm{S_T}$-ITR), note that when an agent splits, each of the newly created "child" agents takes a copy of the capabilities provided by the "parent"; *e.g.* $\ell[\![P \mid Q]\!]_\Gamma \equiv \ell[\![P]\!]_\Gamma \mid \ell[\![Q]\!]_\Gamma$. Note also that when a "private" name becomes "public" (($\mathrm{S_T}$-NEWC) and ($\mathrm{S_T}$-NEWL)) the agent is given the capability to use the once private name.

In the reduction rules ($\mathrm{R_T}$-GO) and ($\mathrm{R_T}$-EQ), agents preserve their capabilities as they reduce. Only in rule ($\mathrm{R_T}$-COMM) are capabilities modified. The rule uses the notation "$\{_\ell v{:}\mathrm{T}\}$" which defines a type environment in which the names in v are assigned the types

Figure 6 The Tagged Language

Reduction:

$$
\begin{array}{rl}
(\text{R}_\text{T}\text{-GO}) & \ell[\![\mathsf{go}\,k.\,P]\!]_\Gamma \longmapsto k[\![P]\!]_\Gamma \\
(\text{R}_\text{T}\text{-COMM}) & \ell[\![a!\langle v\rangle\,P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})\,Q]\!]_\Delta \longmapsto \ell[\![P]\!]_\Gamma \mid \ell[\![Q\{\!\!\{v/X\}\!\!\}]\!]_{\Delta\sqcap\{_\ell v:\mathrm{T}\}} \\
(\text{R}_\text{T}\text{-EQ}_1) & \ell[\![\mathsf{if}\ u=u\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]_\Gamma \longmapsto \ell[\![P]\!]_\Gamma \\
(\text{R}_\text{T}\text{-EQ}_2) & \ell[\![\mathsf{if}\ u=v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]_\Gamma \longmapsto \ell[\![Q]\!]_\Gamma \quad \text{if } u\neq v
\end{array}
$$

Structural equivalence (rule (S-EXTR) and (S-GARBAGE$_1$) from Figure 3):

$$
\begin{array}{rl}
(\text{S}_\text{T}\text{-GARBAGE}_2) & \ell[\![\mathsf{stop}]\!]_\Gamma \equiv \mathbf{0} \\
(\text{S}_\text{T}\text{-SPLIT}) & \ell[\![P \mid Q]\!]_\Gamma \equiv \ell[\![P]\!]_\Gamma \mid \ell[\![Q]\!]_\Gamma \\
(\text{S}_\text{T}\text{-COPY}) & \ell[\![*P]\!]_\Gamma \equiv \ell[\![P]\!]_\Gamma \mid \ell[\![*P]\!]_\Gamma \\
(\text{S}_\text{T}\text{-NEW}) & \ell[\![(\nu e{:}\mathrm{E})\,P]\!]_\Gamma \equiv (\nu_\ell e{:}\mathrm{E})\,\ell[\![P]\!]_{\Gamma,_{(e)}\{_\ell e:\mathrm{E}\}} \ \text{ if } e\notin \mathsf{fn}(\Gamma)\cup\{\ell\}
\end{array}
$$

Typing relation (all rules from Figure 5 but (N-RUN)):

$$
(\text{N}_\text{T}\text{-RUN})\ \frac{\Delta \vdash_\ell P}{\Gamma \Vdash \ell[\![P]\!]_\Delta}\ \Gamma <: \Delta
$$

in T at ℓ. For example:[4]

$$
\{_\ell (a, b@k){:}(\mathrm{A}, \mathrm{B}@\mathsf{loc}\{c{:}\mathrm{C}\})\} \;=\; \{\ell{:}\mathsf{loc}\{a{:}\mathrm{A}\}, k{:}\mathsf{loc}\{c{:}\mathrm{C}, b{:}\mathrm{B}\}\}
$$

In the rule (R$_\text{T}$-COMM), which states

$$
\ell[\![a!\langle v\rangle\,P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})\,Q]\!]_\Delta \longmapsto \ell[\![P]\!]_\Gamma \mid \ell[\![Q\{\!\!\{v/X\}\!\!\}]\!]_{\Delta\sqcap\{_\ell v:\mathrm{T}\}}
$$

there are two agents at ℓ: one willing to send the value v, and the other waiting to receive a value into X. Recall that $\mathsf{obj}(\mathrm{A})$ denotes the transmission type, or *object type* used in the channel type A, *i.e.* $\mathsf{obj}(\mathsf{res}\langle\mathrm{T}\rangle) = \mathrm{T}$. The capabilities offered by the sender are determined by $\mathsf{obj}(\Gamma(\ell, a))$, *i.e.* the type that the sender assigns to channel a. The capabilities expected by the receiver are determined by the *reception type* T, which must be a supertype of $\mathsf{obj}(\Delta(\ell, a))$. If the sender is unwilling to send sufficient capabilities to satisfy the receiver (*i.e.* $\mathsf{obj}(\Gamma(\ell, a)) \not<: \mathrm{T}$), then a runtime error will occur; we discuss this later. Otherwise the communication proceeds and, after receiving the value v, the receiver's capability set is augmented with the capabilities specified by v and T at ℓ.

As an example, let $\mathrm{T} = C@\mathsf{loc}$ in the following tagged system:

$$
\ell[\![a!\langle c@k\rangle\;P]\!]_{\{\ldots,\,k:\mathsf{loc}\{b:\mathrm{B},\,c:\mathrm{C}\}\}} \mid \ell[\![a?(x@z{:}\mathrm{T})\;Q]\!]_{\{\ldots,\,k:\mathsf{loc}\{d:\mathrm{D}\}\}}
$$

[4] The notation $\{_\ell v{:}\mathrm{T}\}$ is defined by induction on v similarly to the definition of $\Gamma,_{(v)}\{_w v{:}\mathrm{T}\}$ given on page 16.

$$
\begin{array}{rl}
\{_\ell u{:}\mathrm{T}\} & = \{\ell{:}\mathsf{loc}\{u{:}\mathrm{T}\}\} \\
\{_\ell \widetilde{u}{:}\widetilde{\mathrm{T}}\} & = \{_\ell U_1{:}\mathrm{T}_1\} \sqcap \cdots \sqcap \{_\ell U_n{:}\mathrm{T}_n\} \\
\{_\ell \widetilde{u}@w{:}\widetilde{\mathrm{A}}@\mathrm{K}\} & = \{w{:}\mathrm{K}\} \sqcap \{_w \widetilde{u}{:}\widetilde{\mathrm{A}}\}
\end{array}
$$

Some properties of this definition are stated in Lemma A.6 of the Appendix.

After the communication the system is:

$$\ell[\![P]\!]_{\{\ldots,\,k:\mathsf{loc}\{b:\mathrm{B},\,c:\mathrm{C}\}\}} \mid \ell[\![Q\{\!\!|{}^{c@k}\!/\!{}_{x@z}|\!\!\}]\!]_{\{\ldots,\,k:\mathsf{loc}\{d:\mathrm{D},\,c:\mathrm{C}\}\}}$$

The receptor has gained capabilities through this communication, as mediated through the reception type T.

The typing system is extended to the tagged language simply by changing the rule (N-RUN), as shown in Figure 6. To help distinguish tagged and untagged systems, we use the symbol $\Vdash$ when writing typing judgments for tagged systems. (On threads and values the syntax and typing systems are identical.) To infer $\Gamma \Vdash \ell[\![P]\!]_\Delta$ it must be the case that $\Gamma <: \Delta$ and $\Delta \vdash_\ell P$. The first requirement simply verifies that the tags are consistent with the global types specified in Γ. The second requirement guarantees that the agent uses the resources in the system only as it is allowed. Consider:

$$k[\![a!\langle\rangle]\!]_{\{\ldots,\,k:\mathsf{loc}\{c:\mathrm{C}\}\}}$$

Here the agent is attempting to use the channel a at k, which it is not permitted to do. This term cannot be well-typed, even under a type environment that defines channel a at k.

THEOREM 4.8 (TAGGED SUBJECT REDUCTION).
If $\Gamma \Vdash M$ *and* $M \longmapsto M'$ *then* $\Gamma \Vdash M'$.

The proof of subject reduction is similar to the proof for the untagged language; the details are in the Appendix.

Before describing runtime error, we establish that tagged and untagged reduction are closely related. To do this, we define a function "tag_Γ" which takes a (closed) system in the untagged language and returns the set of tagged terms which can safely be derived from it using Γ. Throughout the rest of this discussion we will use N to range over untagged terms and M to range over tagged terms. The function "tag_Γ" is defined on the structure of systems as follows:

$$\begin{aligned}
\mathrm{tag}_\Gamma(\mathbf{0}) &= \{\mathbf{0}\} \\
\mathrm{tag}_\Gamma(\ell[\![P]\!]) &= \{\ell[\![P]\!]_\Delta \quad \mid \Gamma <: \Delta \text{ and } \Delta \vdash_\ell P \quad \} \\
\mathrm{tag}_\Gamma(P_1 \mid P_2) &= \{\, Q_1 \mid Q_2 \quad \mid Q_i \in \mathrm{tag}_\Gamma(P_i) \quad \} \\
\mathrm{tag}_\Gamma((\nu_\ell\, e{:}\mathrm{E})\, N) &= \{(\nu_\ell\, e{:}\mathrm{E})\, M \mid M \in \mathrm{tag}_{(\Gamma,_{\langle e\rangle}\{_\ell\, e:\mathrm{E}\})}(N)\ \}
\end{aligned}$$

The definition of $\mathrm{tag}_\Gamma(N)$ is adapted directly from the rules for typing tagged systems, therefore the following Lemma can be trivially verified:

LEMMA 4.9.
(a) $\mathrm{tag}_\Gamma(N)$ *is nonempty if and only if* $\Gamma \vdash N$.
(b) *If* $M \in \mathrm{tag}_\Gamma(N)$ *then* $\Gamma \Vdash M$. □

Note that a well-typed system can be regarded as a tagged system in which the tags on threads can be intuitively, and automatically, inferred from the typing. If we know that the system is well-typed with respect to Γ, then the function "tag_Γ" can be determinized, thus providing a method for generating such a term. To do so, we need only simplify the rule

Figure 7 Runtime Errors

$$
\begin{array}{lrl}
(\textsc{e-eq}) & \ell[\![\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]_\Gamma \overset{err}{\longmapsto} & \text{if } \nexists \mathrm{T}\colon\ \Gamma \vdash_\ell u{:}\mathrm{T}, v{:}\mathrm{T} \\
(\textsc{e-snd}) & \ell[\![a!\langle v\rangle\, P]\!]_\Gamma \overset{err}{\longmapsto} & \text{if } \Gamma_\ell(v) \not<: \mathsf{obj}(\Gamma(\ell, a)) \\
(\textsc{e-rcv}) & \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_\Delta \overset{err}{\longmapsto} & \text{if } \mathsf{obj}(\Delta(\ell, a)) \not<: \mathrm{T} \\
(\textsc{e-comm}) & \ell[\![a!\langle v\rangle\, P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_\Delta \overset{err}{\longmapsto} & \text{if } \mathsf{obj}(\Gamma(\ell, a)) \not<: \mathsf{obj}(\Delta(\ell, a))
\end{array}
$$

$$
(\textsc{e-new})\ \frac{N \overset{err}{\longmapsto}}{(\nu e)\, N \overset{err}{\longmapsto}}
\qquad
(\textsc{e-par})\ \frac{N \overset{err}{\longmapsto}}{N \mid M \overset{err}{\longmapsto}}
\qquad
(\textsc{e-str})\ \frac{N \equiv M \quad M \overset{err}{\longmapsto}}{N \overset{err}{\longmapsto}}
$$

for agents to $\mathrm{tag}_\Gamma(\ell[\![P]\!]) = \{\ell[\![P]\!]_\Gamma\}$. We will write "$\underline{\mathrm{tag}}_\Gamma(N)$" to refer to this determinized version of the function.

The following Proposition shows that tagged and untagged reduction can be considered interchangeable.

PROPOSITION 4.10. *Suppose* $M \in \mathrm{tag}_\Gamma(N)$*, then*
(a) $N \longrightarrow N'$ *implies* $\exists M'\colon\ M' \in \mathrm{tag}_\Gamma(N')\colon\ M \longmapsto M'$
(b) $M \longmapsto M'$ *implies* $\exists N'\colon\ M' \in \mathrm{tag}_\Gamma(N')\colon\ N \longrightarrow N'$

Proof. In both cases, one must first establish a similar result for the structural equivalence, using the subject reduction results for the structural equivalence. The main results can then be verified. The proof is by induction on the definition of reduction, using the Subject Reduction Theorems for the tagged and untagged semantics. The proof is simple and tedious and left to the interested reader. □

RUNTIME ERRORS. Intuitively, a runtime error occurs whenever *an agent uses a name contrary to the capabilities it has acquired for that name.* This informal idea is readily formalized in the tagged language. The definition is given in Figure 7 as a unary predicate over tagged systems $N \overset{err}{\longmapsto}$. We write $N \overset{err}{\not\longmapsto}$ for $\neg(N \overset{err}{\longmapsto})$. For example, the rule (E-EQ) states that it is in an error for an agent to attempt to compare two channel names that do not belong at the current location. There is no rule for the "go" operator, because in the current context there are no errors associated with that operator (but see Section 6).

The most interesting axioms are for communication. These use the notation $\Gamma_\ell(v)$ to denote the least type, if any, which the typing environment Γ can assign to the value v at ℓ. $\Gamma_\ell(v)$ is defined inductively on the structure of v:

$$
\begin{aligned}
\Gamma_\ell(a) &= \Gamma(\ell, a) \\
\Gamma_\ell(v@k) &= \Gamma_k(v)@\Gamma(k) \\
\Gamma_\ell((v_1, .., v_n)) &= (\Gamma_\ell(v_1), ..., \Gamma_\ell(v_n))
\end{aligned}
$$

As usual, $\Gamma_\ell(v)$ is strict in undefinedness; for example, if $a \notin \mathrm{dom}(\Gamma(\ell))$, then $\Gamma_\ell((a, b))$ is undefined. One can easily check the following Lemma.

LEMMA 4.11. *For any value* v*, if* $\Gamma \vdash_\ell v{:}\mathrm{T}$ *then* $\Gamma_\ell(v)$ *is defined and* $\Gamma_\ell(v) <: \mathrm{T}$*, and therefore by Lemma 4.4* $\Gamma \vdash_\ell v{:}\Gamma_\ell(v)$. □

The simplest form of communication error is an arity mismatch, *i.e.* the sent value v cannot be typed at the reception type T. As discussed in Section 4.3, such errors in our setting include senders which communicate values without sufficient permission, or receivers that attempt to secretly “bump up” the capabilities on a received name. A simple rule for communication errors might thus be:

$$\ell[\![a!\langle v\rangle\, P]\!]_{\Gamma} \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_{\Delta} \overset{err}{\longmapsto} \text{ if } \Gamma_{\ell}(v) \not<: \mathrm{T}$$

This rule requires that the sender have all of the permissions on v that the receiver requests (via T). While this rule prevents senders from “making up” capabilities, it doesn't keep receivers from doing so. For example, let

$$\mathrm{A} = \mathsf{res}\langle \mathsf{loc}\{b{:}\mathrm{B}\}\rangle$$

and suppose $\Gamma(\ell, c) = \mathsf{res}\langle \mathrm{A}\rangle$. Then using the rule proposed above, the system

$$\ell[\![(\nu a{:}\mathrm{A})\;\; c!\langle a\rangle\;\; a!\langle k\rangle]\!]_{\Gamma} \mid \ell[\![c?(x{:}\mathrm{A})\;\; x?(z{:}\mathsf{loc}\{b{:}\mathrm{B}, d{:}\mathrm{D}\})\;\; Q]\!]_{\Delta}$$

will not produce an error, as long as $\Gamma(k)$ actually has the b and d capabilities. However, the receiving agent has clearly gained more capabilities at k than the sender intended (indeed, more capabilities the sender has itself), namely access to d. The problem here is that the intermediary role of channel a is ignored.

To avoid such problems more refined rules are required. We use those given in Figure 7. Using these rules (in particular (E-RCV)), note that the agent

$$\ell[\![c?(x{:}\mathrm{A})\;\; x?(z{:}\mathsf{loc}\{b{:}\mathrm{B}, d{:}\mathrm{D}\})\;\; Q]\!]_{\Delta}$$

will produce an error after its first input.

With this motivation, let us discuss each of the rules in turn. There are three different reasons why a runtime error might occur due to communication.

- The sender attempts to forge capabilities. Rule (E-SND) says that v may not be sent on a if a requires more capabilities than available at v. Thus an error occurs if the *sender's* view of the value to be sent does not satisfy the requirements of (the *sender's* view) of the communication channel a. Note that this includes the possibility that $\Gamma(\ell, a)$ is not defined, *i.e.* the *sender* has no a capability at ℓ.

- The receiver attempts to forge capabilities. Rule (E-RCV) says that a sender may not assign a received value more capabilities than are allowed by a. Thus an error occurs if the *receiver's* view of the value to be received exceeds the capabilities of (the *receiver's* view) of the values communicated on channel a. Again this includes the case when $\Delta(\ell, a)$ is undefined.

- The sender and receiver cannot agree on the use of a. Rule (E-COMM) precludes this possibility. Thus an error occurs if the sender's view of channel a is incompatible with the receiver's view.

The interested reader is invited to check that the examples of systems which cannot be typed, given at the end of the last section, can all formally give rise to runtime errors.

Taken together, the rules say that in order for the system

$$\ell[\![a!\langle v\rangle\, P]\!]_{\Gamma} \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_{\Delta}$$

to avoid runtime error, each of the following constraints must be satisfied:

$$\Gamma_\ell(v) <: \mathsf{obj}(\Gamma(\ell,a)) <: \mathsf{obj}(\Delta(\ell,a)) <: \mathrm{T}$$

THEOREM 4.12 (TYPE SAFETY). $\Gamma \Vdash M$ *implies* $M \not\xmapsto{err}$.

Proof. We prove the contrapositive, namely $M \xmapsto{err}$ implies that for no Γ can we prove $\Gamma \Vdash M$. The proof proceeds by induction on the definition of $M \xmapsto{err}$. For the rule involving the structural equivalence, we use the Subject Reduction Theorem, which states that if $M \equiv M'$ then $\Gamma \vdash M$ iff $\Gamma \vdash M'$. The other cases are all straightforward. We present a representative case, (E-SND). The rule states:

$$\ell[\![a!\langle v\rangle\, P]\!]_\Gamma \xmapsto{err} \quad \text{if } \Gamma_\ell(v) \not<: \mathsf{obj}(\Gamma(\ell,a))$$

By way of contradiction, assume that $\Delta \Vdash \ell[\![a!\langle v\rangle\, P]\!]_\Gamma$. We show that from this premise we may conclude $\Gamma_\ell(v) <: \mathsf{obj}(\Gamma(\ell,a))$, leading to a contradiction.

Using the premise and the rule ($\mathrm{N_T}$-RUN), we have that $\Gamma \vDash_\ell a!\langle v\rangle\, P$. This judgment can only be achieved using (T-W), and therefore we have that $\Gamma \vDash_\ell a{:}\mathsf{res}\langle \mathrm{T}\rangle, v{:}\mathrm{T}$ for some T. Using Lemma 4.11, we therefore may conclude that $\Gamma_\ell(v) <: \mathrm{T}$. Using similar reasoning, we have $\mathsf{res}\langle \mathrm{T}\rangle = \Gamma(\ell,a)$, and thus $\mathrm{T} = \mathsf{obj}(\Gamma(\ell,a))$. We therefore may conclude that $\Gamma_\ell(v) <: \mathsf{obj}(\Gamma(\ell,a))$, as desired. □

The Type Safety and Subject Reduction Theorems ensure that well-typed systems do not give rise to runtime errors. Specifically if $\Gamma \vdash N$ then we can take N to represent the tagged term $\underline{\mathrm{tag}}_\Gamma(N)$. Proposition 4.10 ensures that N and $\underline{\mathrm{tag}}_\Gamma(N)$ have essentially the same reduction sequences. Moreover, if $\underline{\mathrm{tag}}_\Gamma(N) \longmapsto^* M'$ then $M' \not\xmapsto{err}$. This later property follows from a more general corollary:

COROLLARY 4.13.
(a) *If* $\Gamma \vdash N$ *and* $N \longrightarrow^* N'$ *then* $\mathrm{tag}_\Gamma(N')$ *is nonempty.*
(b) $M' \in \mathrm{tag}_\Gamma(N')$ *implies* $M' \not\xmapsto{err}$.
(c) *If* $M \in \mathrm{tag}_\Gamma(N)$ *and* $M \longmapsto^* M'$ *then* $M' \not\xmapsto{err}$.

Proof. (a) follows from Proposition 4.10, since $\Gamma \vdash N$ implies that $\mathrm{tag}_\Gamma(N)$ is nonempty. (b) follows from Lemma 4.9b and Theorem 4.12. (c) follows from Lemma 4.9b, Theorem 4.8 (using induction on $\longmapsto^*$) and Theorem 4.12. □

5. AN IMPROVED TYPING SYSTEM

Here we argue that the typing system of the previous section is too restrictive and suggest a simple modification which enables a much larger class of systems to be typed.

Consider the following thread:

$$a?(x_1 @ z_1{:}\mathrm{T}_1)\ b?(x_2 @ z_2{:}\mathrm{T}_2)\ \mathsf{go}\, z_1 .\, d!\langle x_1, x_2\rangle$$

This thread cannot be typed, and reasonably so, as it can easily give rise to runtime errors. The thread receives location z_1 with private channel x_1 and location z_2 with private channel x_2. The variables z_1 and z_2 may, of course, be bound to different locations at runtime; nonetheless, the thread attempts to use x_2 as though it were local to z_1 providing the potential for a runtime error.

If the use of x_2 at z_1 is guarded by the condition $z_1 = z_2$, however, no such runtime error can occur:

$$R \stackrel{def}{=} a?(x_1@z_1{:}\mathrm{T}_1)\ b?(x_2@z_2{:}\mathrm{T}_2)\ \mathsf{if}\ z_1 = z_2\ \mathsf{then\ go}\, z_1 .\, d!\langle x_1, x_2\rangle \qquad (*)$$

Assuming that for some E_i, the types T_i satisfy the constraints

$$\mathrm{T}_1 <: \mathrm{E}_1@\mathsf{loc}\{d{:}\mathsf{res}\langle \mathrm{E}_1, \mathrm{E}_2\rangle\} \qquad \mathrm{T}_2 <: \mathrm{E}_2@\mathsf{loc} \qquad (**)$$

then this term (more formally, a tagged version of it) can never give rise to a runtime error. The output on d is only ever executed when it has been established that the two received channels are at the same location. Nonetheless, our type system will reject it. The reason is that the rule for matching takes no notice of a match:

$$\frac{\Gamma \vdash_w u{:}\mathrm{T}, v{:}\mathrm{T}, P, Q}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

To type "$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$" the subterms P and Q must be well-typed with respect to the original type environment Γ.

Consider the thread R given in (*). Suppose that we are attempting to prove that $\Delta \vdash_\ell R$, where the T_i which appear in R are the greatest types that satisfy (**) (*i.e.*, take the inequations in (**) to be equations). Then when typing the subterm "$d!\langle x_1, x_2\rangle$" of R we are obliged to show

$$\Delta, {}_{(z_1)}\{z_1{:}\mathsf{loc}\{d, x_1\}\}, {}_{(z_2)}\{z_2{:}\mathsf{loc}\{x_2\}\} \vdash_{z_1} d!\langle x_1, x_2\rangle$$

but this clearly is not provable since x_2 is undefined at z_1.

There are safe ways to type such terms, however. One approach would be to augment the type environment with an equivalence relation between types. The solution we adopt, instead, extends the definition of the typing relation (Figure 5) with one additional rule, given in Figure 8. We write $\Gamma \vdash' N$ to indicate that N is well-typed using this slightly weaker typing system; similarly we write $\Gamma \Vdash' M$, if M is tagged. Whereas the use of an equivalence relation in the type system is somewhat more general, our approach has the advantage of simplicity and is sufficient for all of the examples we have found.

The new typing system improves over the old by replacing (T-EQ) with the new rule (T-EQ′) for matching identifiers. In the old type system, matching is "a no-op" in the sense that the fact that two identifiers have been found to be equal provides no "advantage" to the thread that makes the match. The new rule allows the system to type threads that *do* take advantage of a match.

(T-EQ′) states that the thread $\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is well-typed to run at w, relative to Γ, Q is well-typed with respect to Γ and P is well-typed with respect to the *augmented* environment which equates the capabilities of u and v. The weaker requirement on P is reasonable because after the match $u = v$ these locations are known to be identical. Note that subtyping on channels is trivial, and therefore for channel types this rule is no more powerful than the original rule (T-EQ). (However in the next Section such subtyping will be introduced.) Also recall that location types all have a common supertype "loc", and therefore the rule (T-EQ′) can be applied to any match between location names simply by taking L and K to be loc, although in this case there is also no advantage to using (T-EQ′)

Figure 8 An Improved Type System

All rules from Figure 5 except (T-EQ).

$$\frac{\begin{array}{l}(\text{T-EQ}')\\ \Gamma \vdash_w u{:}\mathrm{S}\\ \Gamma \vdash_w v{:}\mathrm{T}\\ \Gamma \sqcap \{_w u{:}\mathrm{T}\} \sqcap \{_w v{:}\mathrm{S}\} \vdash_w P\\ \Gamma \vdash_w Q\end{array}}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

over the old rule (T-EQ). Advantage is only obtained when the types to which it is applied are non-trivial location types.

The new type system would be useless if it weren't sound. All of the results from Section 4.2 and Section 4.4 also apply to the improved type system. We state only subject reduction and type safety below. For proofs see the Appendix.

THEOREM 5.1 (SUBJECT REDUCTION, TYPE SAFETY).
(a) *If* $\Gamma \vdash' N$ *and* $N \longrightarrow N'$ *then* $\Gamma \vdash' N'$.
(b) *If* $\Gamma \Vdash' N$ *and* $N \longmapsto N'$ *then* $\Gamma \Vdash' N'$.
(c) $\Gamma \Vdash' N$ *implies* $N \not\overset{err}{\longmapsto}$.

EXAMPLES. Revisiting example (*), we see that to derive $\Delta \vdash_\ell' R$ it is sufficient to establish:

$$\Delta,\, _{(z_1)}\{z_1{:}\mathsf{loc}\{d, x_1\}\},\, _{(z_2)}\{z_2{:}\mathsf{loc}\{x_2\}\} \vdash_\ell' \mathsf{if}\ z_1 = z_2\ \mathsf{then}\ \mathsf{go}\, z_1 . d!\langle x_1, x_2\rangle$$

Now using the new rule $(\text{T-EQ}')$ this can be reduced to

$$\Delta, z_1{:}\mathsf{loc}\{d, x_1, x_2\}, z_2{:}\mathsf{loc}\{d, x_1, x_2\} \vdash_\ell' \mathsf{go}\, z_1 . d!\langle x_1, x_2\rangle$$

which is straightforward to establish, assuming the constraints of (**).

The augmented type system is also needed in order to type the "remote channel creation" code reported in Example 3.4 of Section 3. There we presented an encoding of

$$\mathrm{T}(h) \Leftarrow \mathsf{go}\, \ell.(\nu a, b)\ \mathsf{go}\, h. P \qquad (\dagger)$$

as:

$$\begin{array}{l}\mathrm{T}(h) \Leftarrow (\nu r)\ \mathsf{go}\, \ell.(\nu a, b)\ \mathsf{go}\, h. R!\langle (a, b)@\ell\rangle\\ \qquad\qquad \mid R?((x, y)@z)\,\mathsf{if}\ z = \ell\ \mathsf{then}\ P\{\!\!\{^{(x,y)@z}/_{(a,b)@\ell}\}\!\!\}\end{array} \qquad (\ddagger)$$

Using the type system of Section 4.2, the fact that (†) is well-typed does not guarantee that (‡) is well-typed; using $(\text{T-EQ}')$, however, this property can be established. The "routed forwarding" example (Example 3.5) also requires the improved type system.

In fact there are many cases in which it is useful for an agent to accumulate knowledge of the capabilities of a location as computation proceeds. This appears to be essential for coding certain types of programs in a language such as ours where access to distributed resources is controlled using explicit capabilities.

As a particularly simple example, consider a server agent that provides information about a freshly created location piecemeal:

$$k[\![(\nu\ell{:}\mathsf{loc}\{a,b,c\})\ d!\langle a@\ell\rangle\ e!\langle b@\ell\rangle\ f!\langle c@\ell\rangle]\!]$$

Here the server creates a new location ℓ with three local methods a, b and c and gradually exports knowledge of ℓ and its resources, one at a time, on the public channels d, e and f. A client of such a server, knowing to expect this trickle of information, might take the form:

$$k[\![d?(x_1@z_1)\ e?(x_2@z_2)\ \mathsf{if}\ z_1 = z_2\ \mathsf{then}\ f?(x_3@z_3)\ \mathsf{if}\ z_1 = z_3\ \mathsf{then}\ Q]\!]$$

As communication with the server agent proceeds, the client gets more and more capabilities at ℓ.

6. TYPE EXTENSIONS

In this section we show how to extend our results to a richer type system with non-trivial subtyping on channel types. Following Pierce and Sangiorgi [26], channel subtyping is defined using *read* and *write* capabilities. Our requirement that all types be FBC, however, forces us to follow a more general approach than that of [26]. Examples of the use of these extended types may be found towards the end of the section.

TYPES AND SUBTYPING. The definition of extended *pre-types* is given in Figure 9, where we explicitly introduce syntactic categories for *location capabilities* κ-λ and *channel capabilities* α-β. We define *types* below, after discussing subtyping.

In the extended language we will require explicit capabilities to perform operations on locations; thus the set of location capabilities is extended from that of Section 4. The new capabilities are:

- go, the ability to move to the location, and
- newc, the ability to create a new local channel.

In other languages, such as that considered in [27], further capabilities might be defined, such as the capability to *halt* or *migrate* a location or the ability to create *sublocations*.

Since we allow subtyping on channel types the definition of subtyping on location types must generalize that of previous sections. There, subtyping corresponded to reverse subset inclusion on capabilities. For the extended type system, we have that $\mathrm{K} <: \mathrm{L}$ if for every capability $\lambda \in \mathrm{L}$ there exists a capability $\kappa \in \mathrm{K}$ which is "at least as good", *i.e.* $\kappa <: \lambda$. Here the location capabilities κ and λ are compared inductively using the associated types, *e.g.* $a{:}\mathrm{A} <: a{:}\mathrm{B}$ if $\mathrm{A} <: \mathrm{B}$.

We also define capabilities for channels, which may be interpreted as follows:

- $\mathsf{r}\langle \mathrm{T}\rangle$ grants permission for an agent to receive values v from a channel and then to use each v with *at most* the permissions specified by T; and
- $\mathsf{w}\langle \mathrm{S}\rangle$ grants permission for an agent to send values v into a channel, as long as that agent has, on each v sent, *at least* the permissions specified by S.

Subtyping for channels is just as for locations: $\mathrm{A} <: \mathrm{B}$ if for every capability $\beta \in \mathrm{B}$ there exists a capability $\alpha \in \mathrm{A}$ such that $\alpha <: \beta$. But the subtyping relation on channel

Figure 9 Extended Pre-Types

Capabilities:	Subtyping:
$\kappa ::= \mathsf{go} \mid \mathsf{newc}$	$\kappa <: \kappa$
$\quad \mid a{:}\mathrm{A}$	$a{:}\mathrm{A} <: a{:}\mathrm{B}$ if $\mathrm{A} <: \mathrm{B}$
$\alpha ::= \mathsf{r}\langle \mathrm{T}\rangle$	$\mathsf{r}\langle \mathrm{T}\rangle <: \mathsf{r}\langle \mathrm{T}'\rangle$ if $\mathrm{T} <: \mathrm{T}'$
$\quad \mid \mathsf{w}\langle \mathrm{S}\rangle$	$\mathsf{w}\langle \mathrm{S}\rangle <: \mathsf{w}\langle \mathrm{S}'\rangle$ if $\mathrm{S}' <: \mathrm{S}$
Pre-Types:	
$\mathrm{K} ::= \mathsf{loc}\{\widetilde{\kappa}\}$	$\mathrm{K} <: \mathrm{L}$ if $\forall \lambda \in \mathrm{L}: \exists \kappa \in \mathrm{K}: \kappa <: \lambda$
$\mathrm{A} ::= \mathsf{res}\{\widetilde{\alpha}\}$	$\mathrm{A} <: \mathrm{B}$ if $\forall \beta \in \mathrm{B}: \exists \alpha \in \mathrm{A}: \alpha <: \beta$
$\mathrm{E} ::= \mathrm{A} \mid \mathrm{K} \mid \mathcal{B}$	
$\mathrm{T} ::= \mathrm{A} \mid \mathcal{B}$	$\mathcal{B} <: \mathcal{B}$
$\quad \mid \mathrm{A}_1, .., \mathrm{A}_n @\mathrm{K}$	$\widetilde{\mathrm{A}}@\mathrm{K} <: \widetilde{\mathrm{B}}@\mathrm{L}$ if $\mathrm{K} <: \mathrm{L}$ and $\widetilde{\mathrm{A}} <: \widetilde{\mathrm{B}}$
$\quad \mid (\mathrm{T}_1, .., \mathrm{T}_n)$	$\widetilde{\mathrm{S}} <: \widetilde{\mathrm{T}}$ if $\forall i: \mathrm{S}_i <: \mathrm{T}_i$

capabilities is more interesting:

$$\begin{array}{l} \mathsf{r}\langle \mathrm{T}\rangle <: \mathsf{r}\langle \mathrm{T}'\rangle \text{ if } \mathrm{T} <: \mathrm{T}' \\ \mathsf{w}\langle \mathrm{S}\rangle <: \mathsf{w}\langle \mathrm{S}'\rangle \text{ if } \mathrm{S}' <: \mathrm{S} \end{array}$$

As one should expect from the intuitive descriptions given above, the read capability is covariant, whereas the write capability is contravariant. Thus a receiver can always take *fewer* capabilities than specified by T, whereas a sender can always send *more* capabilities than specified by S.

DEFINITION 6.1 (EXTENDED TYPES).

(a) A location pre-type K is a type if $a{:}\mathrm{A} \in \mathrm{K}$ and $a{:}\mathrm{A}' \in \mathrm{K}$ imply $\mathrm{A} = \mathrm{A}'$.

(b) A channel pre-type A is a type if:

$$\begin{array}{lll} \mathsf{r}\langle \mathrm{T}\rangle \in \mathrm{A} \text{ and } \mathsf{r}\langle \mathrm{T}'\rangle \in \mathrm{A} & \text{imply } \mathrm{T} = \mathrm{T}' \\ \mathsf{w}\langle \mathrm{S}\rangle \in \mathrm{A} \text{ and } \mathsf{w}\langle \mathrm{S}'\rangle \in \mathrm{A} & \text{imply } \mathrm{S} = \mathrm{S}' \\ \mathsf{r}\langle \mathrm{T}\rangle \in \mathrm{A} \text{ and } \mathsf{w}\langle \mathrm{S}\rangle \in \mathrm{A} & \text{imply } \mathrm{S} <: \mathrm{T} \end{array}$$

and S, T are in turn types

(c) Pre-types of the form $\widetilde{\mathrm{T}}$ and $\widetilde{\mathrm{A}}@\mathrm{K}$ are types if their constituent components are types. □

As before, location types are allowed at most one capability for each channel. Channel type are also constrained to have at most one read and one write capability. The final constraint on channel types is a consistency requirement. It prevents agents from "fabricating" capabilities. For example, it prevents an agent from sending a value at type $\mathsf{loc}\{a{:}\mathrm{A}\}$ and then receiving the same value at type $\mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}$. We discuss this further after presenting the Soundness Theorem for the typing system.

Note that $\mathsf{loc}\{\}$ (often abbreviated to loc) is a supertype of every simple location type and $\mathsf{res}\{\}$ is a supertype of every channel type.

SIMPLE AND "PS" TYPES. The extended types include the "simple" types studied in Sections 4 and 5. The simple channel type $\mathsf{res}\langle \mathrm{T}\rangle$ is here identified with the type $\mathsf{res}\{\mathsf{w}\langle \mathrm{T}\rangle, \mathsf{r}\langle \mathrm{T}\rangle\}$.

On simple types, the subtyping relation of Figure 9 degenerates to that of Figure 4. To establish $\mathsf{res}\{\mathsf{w}\langle \mathrm{T}\rangle, \mathsf{r}\langle \mathrm{T}\rangle\} <: \mathsf{res}\{\mathsf{w}\langle \mathrm{S}\rangle, \mathsf{r}\langle \mathrm{S}\rangle\}$, it is required that both $\mathrm{T} <: \mathrm{S}$ and $\mathrm{S} <: \mathrm{T}$; therefore T and S must be identical.

Readers who are familiar with [26] will notice that Pierce and Sangiorgi's channel types — "PS" types — are also representable in our type system (ignoring recursion). The PS read type $[\mathrm{T}]^{\mathsf{r}}$ is identified with $\mathsf{res}\{\mathsf{r}\langle \mathrm{T}\rangle\}$, the PS write type $[\mathrm{T}]^{\mathsf{w}}$ is identified with $\mathsf{res}\{\mathsf{w}\langle \mathrm{T}\rangle\}$, and the PS read/write type $[\mathrm{T}]^{\mathsf{rw}}$ is identified with $\mathsf{res}\{\mathsf{w}\langle \mathrm{T}\rangle, \mathsf{r}\langle \mathrm{T}\rangle\}$. For these PS types, our definition of subtyping coincides with that of Pierce and Sangiorgi.

Our channel types include many types that are not definable using the system of Pierce and Sangiorgi, however. For example, the type

$$\mathrm{C} = \mathsf{res}\{\mathsf{r}\langle \mathsf{loc}\{a{:}\mathrm{A}\}\rangle, \mathsf{w}\langle \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}\rangle\}$$

is not expressible as a PS type. Nonetheless, it is easy to see how such types arise when agents are granted different permissions on the names in a system. Say that agent N has a channel c at type C. The type of the channel allows N to use channel a at locations that it reads from c. Other agents, however, may have been granted additional permissions on c. For example, agent M may be able to use both channels a and b at locations read from c. Thus, if N wishes to send a location k on c, it is *required* that N know that both a and b are defined at k; thus N *must have* permissions for both a and b at k. This is true even though N itself cannot read k from c into a variable z and then immediately use channel b at z.

FINITE BOUNDED COMPLETENESS. Before we can adapt the typing rules of Figure 8 to this new language for types or describe the tagged language, we must first define a partial *meet* operator $\sqcap$ and thus prove that the subtype relation is FBC. Because the type system has a contravariant operator, the definition of meet $\sqcap$ requires that the type language also have a *join* $\sqcup$.

The definitions of the partial meet and join operators are given in Figure 10. To make the definitions more readable, we write types simply as sets of capabilities, dropping the res and loc. We also write "$a{:}- \notin \mathrm{K}$" as shorthand for "there exists no A such that $a{:}\mathrm{A} \in \mathrm{K}$." Similarly, "$\mathsf{r}\langle - \rangle \notin \mathrm{A}$" is shorthand for "there exists no T such that $\mathsf{r}\langle \mathrm{T}\rangle \in \mathrm{A}$." Also, let γ range over the set $\{\mathsf{go}, \mathsf{newc}\}$ of *primitive capabilities*.

The definition is long, but it is not complicated, simply tedious. Intuitively, the meet of two types takes the union of their capabilities, whereas the join takes the intersection. In the case that two types have conflicting capabilities, the meet is undefined. On the other hand, the join simply ignores conflicting capabilities, leaving them out. For example, suppose that we are looking at two incompatible channel types, one of which has a capability to read pairs and the other which has a capability to read triples. The meet is undefined; it is not possible have a channel that can read both pairs and triples. The join is defined, but does not include a read capability.

On channels, in order for $\{\mathsf{r}\langle \mathrm{T}\rangle, \mathsf{w}\langle \mathrm{S}\rangle\} \sqcap \{\mathsf{r}\langle \mathrm{T}'\rangle, \mathsf{w}\langle \mathrm{S}'\rangle\}$ to be defined, the types must satisfy the following constraints. (In the figure, arrows indicate inclusion in the subtype relation, *i.e.* "$\mathrm{S} \to \mathrm{T}$" means "$\mathrm{S} <: \mathrm{T}$.")

$$\begin{tikzcd} \mathrm{T} & \mathrm{T}' \\ \mathrm{S} \arrow[u] \arrow[ur] & \mathrm{S}' \arrow[u] \arrow[ul] \end{tikzcd}$$

Figure 10 Partial Meet and Join Operators for Extended Types

For location types, $K \sqcap K'$ is *undefined* if there exists an a such that $a{:}A \in K$ and $a{:}A' \in K'$ and $A \sqcap A'$ is undefined. Otherwise:

$$\begin{aligned} K \sqcap K' = \quad & \{\gamma \mid \gamma \in K \text{ or } \gamma \in K'\} \\ \cup\ & \{a{:}A \mid a{:}A \in K \text{ and } a{:}- \notin K'\} \\ \cup\ & \{a{:}A' \mid a{:}- \notin K \text{ and } a{:}A' \in K'\} \\ \cup\ & \{a{:}A'' \mid a{:}A \in K \text{ and } a{:}A' \in K' \text{ and } A'' = A \sqcap A'\} \end{aligned}$$

For channel types, $A \sqcap A'$ is *undefined* if any of the following hold:

$$\begin{array}{l} \mathsf{r}\langle T\rangle \in A \text{ and } \mathsf{r}\langle T'\rangle \in A' \text{ and } T \sqcap T' \text{ undefined} \\ \mathsf{w}\langle S\rangle \in A \text{ and } \mathsf{w}\langle S'\rangle \in A' \text{ and } S \sqcup S' \text{ undefined} \\ \mathsf{r}\langle T\rangle \in A \text{ and } \mathsf{w}\langle S'\rangle \in A' \text{ and } S' \not<: T \\ \mathsf{w}\langle S\rangle \in A \text{ and } \mathsf{r}\langle T'\rangle \in A' \text{ and } S \not<: T' \end{array}$$

Otherwise the definition is:

$$\begin{aligned} A \sqcap A' = \ & \{\ \mathsf{r}\langle T\rangle \mid \mathsf{r}\langle T\rangle \in A \text{ and } \mathsf{r}\langle -\rangle \notin A'\} \\ \cup\ & \{\ \mathsf{r}\langle T'\rangle \mid \mathsf{r}\langle -\rangle \notin A \text{ and } \mathsf{r}\langle T'\rangle \in A'\} \\ \cup\ & \{\ \mathsf{r}\langle T''\rangle \mid \mathsf{r}\langle T\rangle \in A \text{ and } \mathsf{r}\langle T'\rangle \in A' \text{ and } T'' = T \sqcap T'\} \\ \cup\ & \{\ \mathsf{w}\langle S\rangle \mid \mathsf{w}\langle S\rangle \in A \text{ and } \mathsf{w}\langle -\rangle \notin A'\} \\ \cup\ & \{\ \mathsf{w}\langle S'\rangle \mid \mathsf{w}\langle -\rangle \notin A \text{ and } \mathsf{w}\langle S'\rangle \in A'\} \\ \cup\ & \{\ \mathsf{w}\langle S''\rangle \mid \mathsf{w}\langle S\rangle \in A \text{ and } \mathsf{w}\langle S'\rangle \in A' \text{ and } S'' = S \sqcup S'\} \end{aligned}$$

The join on location types ($K \sqcup K'$) and on channel types ($A \sqcup A'$) is always defined:

$$\begin{aligned} K \sqcup K' = \quad & \{\gamma \mid \gamma \in K \text{ and } \gamma \in K'\} \\ \cup\ & \{\ a{:}A'' \mid a{:}A \in K \text{ and } a{:}A' \in K' \text{ and } A'' = A \sqcup A'\} \\ A \sqcup A' = \ & \{\ \mathsf{r}\langle T''\rangle \mid \mathsf{r}\langle T\rangle \in A \text{ and } \mathsf{r}\langle T'\rangle \in A' \text{ and } T'' = T \sqcup T'\} \\ \cup\ & \{\ \mathsf{w}\langle S''\rangle \mid \mathsf{w}\langle S\rangle \in A \text{ and } \mathsf{w}\langle S'\rangle \in A' \text{ and } S'' = S \sqcap S'\} \end{aligned}$$

For other types, $\sqcap$ and $\sqcup$ are defined by strict homomorphic extension as in Definition 4.2. (Both meet and join are undefined on types that structurally dissimilar.)

As an example of the use of these operators, consider the following:

$$\begin{array}{ll} \mathrm{L} = \mathsf{loc}\{\mathsf{go}, a{:}\mathrm{A}, b{:}\mathrm{B}\} & \mathrm{A} \;= \mathsf{res}\{\mathsf{w}\langle \mathsf{loc}\{d{:}\mathrm{D}, e{:}\mathrm{E}\}\rangle, \mathsf{r}\langle \mathsf{loc}\{d{:}\mathrm{D}\}\rangle\} \\ \mathrm{K} = \mathsf{loc}\{\mathsf{go}, a{:}\mathrm{A}', c{:}\mathrm{C}\} & \mathrm{A}' = \mathsf{res}\{\mathsf{w}\langle \mathsf{loc}\{d{:}\mathrm{D}, f{:}\mathrm{F}\}\rangle\} \end{array}$$

Then we have:

$$\begin{array}{ll} \mathrm{L} \sqcap \mathrm{K} & = \mathsf{loc}\{\mathsf{go}, a{:}(\mathrm{A} \sqcap \mathrm{A}'), b{:}\mathrm{B}, c{:}\mathrm{C}\} \\ \mathrm{L} \sqcup \mathrm{K} & = \mathsf{loc}\{\mathsf{go}, a{:}(\mathrm{A} \sqcup \mathrm{A}')\} \\ \mathrm{A} \sqcap \mathrm{A}' & = \mathsf{res}\{\mathsf{w}\langle \mathsf{loc}\{d{:}\mathrm{D}\}\rangle, \mathsf{r}\langle \mathsf{loc}\{d{:}\mathrm{D}\}\rangle\} \\ \mathrm{A} \sqcup \mathrm{A}' & = \mathsf{res}\{\mathsf{w}\langle \mathsf{loc}\{d{:}\mathrm{D}, e{:}\mathrm{E}, f{:}\mathrm{F}\}\rangle\} \end{array}$$

PROPOSITION 6.2. *The operator $\sqcap$ defined in Figure 10 is a partial meet operator.*

Proof. By induction on the definition of $\sqcap$ one can establish that $\sqcap$ and $\sqcup$ are commutative and associative. Therefore, to establish the result we need only show that for every type T, S, R:

$$\begin{array}{llll} \text{(a)} & \mathrm{R} <: \mathrm{T} \text{ and } \mathrm{R} <: \mathrm{S} & \text{imply} & \mathrm{T} \sqcap \mathrm{S} \text{ defined and } \mathrm{R} <: \mathrm{T} \sqcap \mathrm{S} \\ \text{(b)} & \mathrm{T} <: \mathrm{R} \text{ and } \mathrm{S} <: \mathrm{R} & \text{imply} & \mathrm{T} \sqcup \mathrm{S} \text{ defined and } \mathrm{T} \sqcup \mathrm{S} <: \mathrm{R} \\ \text{(c)} & \mathrm{T} \sqcap \mathrm{S} \text{ defined} & \text{implies} & \mathrm{T} \sqcap \mathrm{S} <: \mathrm{T} \\ \text{(d)} & \mathrm{T} \sqcup \mathrm{S} \text{ defined} & \text{implies} & \mathrm{T} <: \mathrm{T} \sqcup \mathrm{S} \end{array}$$

First note that $<:$, $\sqcap$ and $\sqcup$ are only defined for structurally similar types. The properties (a)-(d) may therefore be established using structural induction. (a) and (b) must be proved together as a single induction hypothesis; likewise (c) and (d). The most interesting case is for channels. We describe this case for each of the four properties.

(a) Suppose that $\mathrm{A}'' <: \mathrm{A}$ and $\mathrm{A}'' <: \mathrm{A}'$. We must show that $\mathrm{A} \sqcap \mathrm{A}'$ is defined and that $\mathrm{A}'' <: \mathrm{A} \sqcap \mathrm{A}'$. The proof proceeds by case analysis on the capabilities in A and A′. We treat the most difficult case, in which A and A′ each contain both read and write capabilities; the other cases can immediately be derived from this one. Let:

$$\begin{array}{ll} \mathrm{A} & = \mathsf{res}\{\mathsf{r}\langle \mathrm{T}\rangle,\ \mathsf{w}\langle \mathrm{S}\rangle\} \\ \mathrm{A}' & = \mathsf{res}\{\mathsf{r}\langle \mathrm{T}'\rangle,\ \mathsf{w}\langle \mathrm{S}'\rangle\} \\ \mathrm{A}'' & = \mathsf{res}\{\mathsf{r}\langle \mathrm{T}''\rangle, \mathsf{w}\langle \mathrm{S}''\rangle\} \end{array}$$

Using the assumption ($\mathrm{A}'' \to \mathrm{A}$ and $\mathrm{A}'' \to \mathrm{A}'$), the induction hypothesis and the fact that A'' is a type, we have:

$$\begin{tikzcd} & \mathrm{S} \arrow[dr, "\text{assumption}"] & & & \mathrm{T} \\ \mathrm{S} \sqcup \mathrm{S}' \arrow[rr, "\text{induction}"] & & \mathrm{S}'' \arrow[r, "\mathrm{A}'' \text{ a type}"] & \mathrm{T}'' \arrow[ur, "\text{assumption}"] \arrow[r, "\text{induction}"] \arrow[dr, "\text{assumption}"'] & \mathrm{T} \sqcap \mathrm{T}' \\ & \mathrm{S}' \arrow[ur, "\text{assumption}"'] & & & \mathrm{T}' \end{tikzcd} \qquad (*)$$

One can easily check that the conditions for definedness of meet at channel types (Figure 10) are satisfied for $\mathrm{A} \sqcap \mathrm{A}'$, and thus:

$$\mathrm{A} \sqcap \mathrm{A}' = \mathsf{res}\{\mathsf{r}\langle \mathrm{T} \sqcap \mathrm{T}'\rangle, \mathsf{w}\langle \mathrm{S} \sqcup \mathrm{S}'\rangle\}$$

From (*) it follows that $\mathsf{r}\langle \mathrm{T}''\rangle <: \mathsf{r}\langle \mathrm{T} \sqcap \mathrm{T}'\rangle$ and $\mathsf{w}\langle \mathrm{S}\rangle <: \mathsf{w}\langle \mathrm{S} \sqcup \mathrm{S}'\rangle$. Thus $\mathrm{A}'' <: \mathrm{A} \sqcap \mathrm{A}'$, as required.

(b) Assume that $\mathrm{A} <: \mathrm{A}''$ and $\mathrm{A}' <: \mathrm{A}''$. We must show that $\mathrm{A} \sqcup \mathrm{A}' <: \mathrm{A}''$; *i.e.* if $\alpha \in \mathrm{A}''$ then there exists a $\beta \in \mathrm{A} \sqcup \mathrm{A}'$ such that $\beta <: \alpha$. Suppose that A'' contains a read capability $\mathsf{r}\langle \mathrm{T}'' \rangle$. Then by the assumption we have that for some T and T′:

$$\begin{array}{llll} \mathsf{r}\langle \mathrm{T} \rangle & \in \mathrm{A} & \qquad \mathrm{T} & <: \mathrm{T}'' \\ \mathsf{r}\langle \mathrm{T}' \rangle & \in \mathrm{A}' & \qquad \mathrm{T}' & <: \mathrm{T}'' \end{array}$$

By induction, $\mathrm{T} \sqcup \mathrm{T}'$ is defined and $\mathrm{T} \sqcup \mathrm{T}' <: \mathrm{T}''$. Therefore $\mathsf{r}\langle \mathrm{T} \sqcup \mathrm{T}' \rangle \in \mathrm{A} \sqcup \mathrm{A}'$. Using the definition of capability subtyping, we also have $\mathsf{r}\langle \mathrm{T} \sqcup \mathrm{T}' \rangle <: \mathsf{r}\langle \mathrm{T}'' \rangle$, as required.

The argument is similar when A'' contains a write capability $\mathsf{w}\langle \mathrm{S}'' \rangle$.

(c) Suppose that $\mathrm{A} \sqcap \mathrm{A}'$ is defined. We show that $\mathrm{A} \sqcap \mathrm{A}' <: \mathrm{A}$. Suppose that $\mathsf{w}\langle \mathrm{S} \rangle \in \mathrm{A}$; we must show that $\mathrm{A} \sqcap \mathrm{A}'$ has a write capability dominated by $\mathsf{w}\langle \mathrm{S} \rangle$. There are two possibilities to consider. (a) First, suppose that A' contains no write capability ($\mathsf{w}\langle - \rangle \notin \mathrm{A}'$). Then, by definition $\mathsf{w}\langle \mathrm{S} \rangle \in \mathrm{A} \sqcap \mathrm{A}'$, and the proof is done. (b) Otherwise it must be that for some S', $\mathsf{w}\langle \mathrm{S}' \rangle \in \mathrm{A}'$. Since $\mathrm{A} \sqcap \mathrm{A}'$ is defined, it must be that $\mathrm{S} \sqcup \mathrm{S}'$ is defined, and therefore $\mathsf{w}\langle \mathrm{S} \sqcup \mathrm{S}' \rangle \in \mathrm{A} \sqcap \mathrm{B}$. By induction, $\mathrm{S} <: \mathrm{S} \sqcup \mathrm{S}'$ and thus $\mathsf{w}\langle \mathrm{S} \sqcup \mathrm{S}' \rangle <: \mathsf{w}\langle \mathrm{S} \rangle$, as required.

The argument is similar when A contains a read capability $\mathsf{r}\langle \mathrm{S} \rangle$.

(d) Suppose that $\mathrm{A} \sqcup \mathrm{A}'$ is defined. We show that $\mathrm{A} <: \mathrm{A} \sqcup \mathrm{A}'$. Suppose that $\mathsf{w}\langle \mathrm{S}'' \rangle \in \mathrm{A} \sqcup \mathrm{A}'$. Therefore it must be that for some S, S″:

$$\mathrm{S}'' = \mathrm{S} \sqcup \mathrm{S}' \qquad \mathsf{w}\langle \mathrm{S} \rangle \in \mathrm{A} \qquad \mathsf{w}\langle \mathrm{S}' \rangle \in \mathrm{A}'$$

By induction, $\mathrm{S} \sqcup \mathrm{S}' <: \mathrm{S}$. Thus, $\mathsf{w}\langle \mathrm{S} \rangle <: \mathsf{w}\langle \mathrm{S} \sqcup \mathrm{S}' \rangle$, as required.

The argument is similar when $\mathrm{A} \sqcup \mathrm{A}'$ contains a read capability $\mathsf{r}\langle \mathrm{T}'' \rangle$. □

We now demonstrate that no partial meet operator exists for PS types. To make the counterexample readable, let us use the following abbreviations:

$$\mathsf{r}\langle \mathrm{T} \rangle = \mathsf{res}\{\mathsf{r}\langle \mathrm{T} \rangle\} \qquad \mathsf{w}\langle \mathrm{T} \rangle = \mathsf{res}\{\mathsf{w}\langle \mathrm{T} \rangle\} \qquad \mathsf{rw}\langle \mathrm{T} \rangle = \mathsf{res}\{\mathsf{w}\langle \mathrm{T} \rangle, \mathsf{r}\langle \mathrm{T} \rangle\}$$

There are three PS types of the form $io\langle\rangle$, where io is an "i/o tag" ($io ::= \mathsf{r} \mid \mathsf{w} \mid \mathsf{rw}$). These are ordered by subtyping as follows, where we drop the final empty brackets, writing "io" instead of "$io\langle\rangle$":

$$\begin{tikzcd} \mathsf{r} & & \mathsf{w} \\ & \mathsf{rw} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Next, consider types of the form $io\langle io'\langle\rangle\rangle$:

$$\begin{tikzcd}[column sep=small] & & & & & \mathsf{w}\langle\mathsf{rw}\rangle & \\ \mathsf{r}\langle\mathsf{w}\rangle & & \mathsf{r}\langle\mathsf{r}\rangle & & \mathsf{w}\langle\mathsf{r}\rangle \arrow[ur] & & \mathsf{w}\langle\mathsf{w}\rangle \arrow[ul] \\ & \mathsf{r}\langle\mathsf{rw}\rangle \arrow[ul] \arrow[ur] & & \mathsf{rw}\langle\mathsf{r}\rangle \arrow[ul] \arrow[ur] & & & \\ & & & \mathsf{rw}\langle\mathsf{rw}\rangle \arrow[ull] \arrow[uuurr, bend right] & & & \\ & & & \mathsf{rw}\langle\mathsf{w}\rangle \arrow[uuulll, bend left] \arrow[uuurrr, bend right] & & & \end{tikzcd}$$

Already here we can see that the type system is not FBC. For example the types $\mathsf{r}\langle\mathsf{r}\rangle$ and $\mathsf{w}\langle\mathsf{rw}\rangle$ have lower bounds, but they have no *greatest* lower bound: $\mathsf{rw}\langle\mathsf{r}\rangle$ and $\mathsf{rw}\langle\mathsf{rw}\rangle$ are incomparable.

THE TYPING SYSTEM. The typing relation $\Gamma \vdash'' N$ is defined in Figure 11. The new typing system has exactly the same rules for values (and most of the same rules for systems) as Figure 5. The new rules for input, output, movement and channel creation are stronger than the old rules: they require explicit capabilities for each of these actions. Because restriction is treated structurally, we must also strengthen the rule for channel creation. (Without the stronger rule, subject reduction fails for the structural equivalence.) But for location creation the rule remains the same as before. The rule for matching names is as in the previous section; this is now useful for channel names since they support non-trivial subtyping, unlike in the previous section.

The extended typing system has a corresponding notion of extended runtime error, presented in Figure 12. Here we use the partial functions "robj" and "wobj" which, given a channel type, return the types of the objects that may read or written on the channel, if these capabilities are defined:

$$\mathrm{robj}(\mathrm{A}) \stackrel{def}{=} \mathrm{T}, \text{ if } \mathsf{r}\langle\mathrm{T}\rangle \in \mathrm{A} \qquad \mathrm{wobj}(\mathrm{A}) \stackrel{def}{=} \mathrm{S}, \text{ if } \mathsf{w}\langle\mathrm{S}\rangle \in \mathrm{A}$$

Finally, to define *tagged reduction* we must also make a small change to the structural equivalence on tagged terms, in line with the change to the typing rules for systems. (No changes are required to the definition of reduction for *untagged* terms.) The agent $\ell[\![(\nu a{:}\mathrm{A})\, P]\!]_\Gamma$ is only allowed to create a at ℓ if it has the newc permission at ℓ; therefore, the rule (ST-NEW) is replaced with the following two rules; that for channel creation has this requirement as a side condition:

$$\begin{array}{lll} (\text{S}_\text{T}\text{-CHAN}'') & \ell[\![(\nu a{:}\mathrm{A})\, P]\!]_\Gamma \equiv (\nu_\ell\, a{:}\mathrm{A})\, \ell[\![P]\!]_{\Gamma,_{(a)}\{_\ell\, a:\mathrm{A}\}} & \text{if } a \notin \mathsf{fn}(\Gamma),\ \Gamma(\ell) <: \mathsf{loc}\{\mathsf{newc}\} \\ (\text{S}_\text{T}\text{-LOC}'') & \ell[\![(\nu k{:}\mathrm{K})\, P]\!]_\Gamma \equiv (\nu_\ell k{:}\mathrm{K})\, \ell[\![P]\!]_{\Gamma,_{(k)}\{_\ell\, k:\mathrm{K}\}} & \text{if } k \notin \mathsf{fn}(\Gamma) \cup \{\ell\} \end{array}$$

Using these definitions we have the standard subject reduction and type safety theorems, which are proved in the Appendix.

THEOREM 6.3 (SOUNDNESS).
(a) *If* $\Gamma \vdash'' N$ *and* $N \longrightarrow N'$ *then* $\Gamma \vdash'' N'$.
(b) *If* $\Gamma \Vdash'' M$ *and* $M \longmapsto M'$ *then* $\Gamma \Vdash'' M'$.
(c) $\Gamma \Vdash'' M$ *implies* $M \stackrel{err''}{\not\longmapsto}$.

The requirement that read and write capabilities on a channel must not conflict (if both are defined) is essential for the validity of the theorem. Suppose two agents share a channel c at ℓ with type $\mathrm{C} = \mathsf{res}\{\mathsf{w}\langle\mathrm{A}_\mathsf{w}\rangle, \mathsf{r}\langle\mathrm{A}_\mathsf{rw}\rangle\}$, where $\mathrm{A}_\mathsf{rw} = \mathsf{res}\{\mathsf{w}\langle\mathrm{T}\rangle, \mathsf{r}\langle\mathrm{T}\rangle\}$ and $\mathrm{A}_\mathsf{w} = \mathsf{res}\{\mathsf{w}\langle\mathrm{T}\rangle\}$ for some T. Note that this is not a valid type (and thus are not allowed by our type system) because the read and write capabilities conflict ($\mathrm{A}_\mathsf{rw} <: \mathrm{A}_\mathsf{w}$). If we did allow such types, however, then we could find Γ, y and Q such that:

$$\Gamma \vdash'' \ell[\![c!\langle a\rangle]\!]_{\{\ldots,\ell:\mathsf{loc}\{c:\mathrm{C},\, a:\mathrm{A}_\mathsf{w},\ldots\}\}} \mid \ell[\![c?(x{:}\mathrm{A}_\mathsf{rw})\, x?(y)\, Q]\!]_{\{\ldots,\ell:\mathsf{loc}\{c:\mathrm{C}\}\}}$$

But it is easy to see that this tagged term leads to a runtime error due to rule (E-COMM$''$); the type of the sent value and the type of the received value do not match. It is appropriate

Figure 11 Extended Typing System

Values as in Figure 5.

Threads: (T-STR) and (T-EQ) from Figure 5

$$\textsc{(t-w}'')\quad \frac{\Gamma \vdash''_w u{:}\mathsf{res}\{\mathsf{w}\langle \mathrm{T}\rangle\} \quad \Gamma \vdash''_w v{:}\mathrm{T} \quad \Gamma \vdash''_w P}{\Gamma \vdash''_w u!\langle v\rangle\, P}$$

$$\textsc{(t-r}'')\quad \frac{\Gamma \vdash''_w u{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{T}\rangle\} \quad \mathrm{fv}(X) \text{ disjoint } \mathrm{fv}(\Gamma) \quad \Gamma,_{(X)}\{_w X{:}\mathrm{T}\} \vdash''_w Q}{\Gamma \vdash''_w u?(X{:}\mathrm{T})\, Q}$$

$$\textsc{(t-eql}'')\quad \frac{\Gamma \vdash''_w u{:}\mathrm{S} \quad \Gamma \vdash''_w v{:}\mathrm{T} \quad \Gamma \sqcap \{_w u{:}\mathrm{S}\} \sqcap \{_w v{:}\mathrm{T}\} \vdash''_w P \quad \Gamma \vdash''_w Q}{\Gamma \vdash''_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{(t-go}'')\quad \frac{\Gamma(u){:}\mathsf{loc}\{\mathsf{go}\} \quad \Gamma \vdash''_u P}{\Gamma \vdash''_w \mathsf{go}\, u.P}$$

$$\textsc{(t-newl}'')\quad \frac{\Gamma(w){:}\mathsf{loc} \quad l \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(\ell)}\{\ell{:}\mathrm{L}\} \vdash''_w P}{\Gamma \vdash''_w (\nu \ell{:}\mathrm{L})\, P}$$

$$\textsc{(t-newc}'')\quad \frac{\Gamma(w){:}\mathsf{loc}\{\mathsf{newc}\} \quad a \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(a)}\{_w a{:}\mathrm{A}\} \vdash''_w P}{\Gamma \vdash''_w (\nu a{:}\mathrm{A})\, P}$$

Networks: rules (N-RUN) and (N-STR) from Figure 5

$$\textsc{(n-newl}'')\quad \frac{\ell \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(l)}\{l{:}\mathrm{L}\} \vdash'' N}{\Gamma \vdash'' (\nu_k \ell{:}\mathrm{L})\, N}$$

$$\textsc{(n-newc}'')\quad \frac{\Gamma(k){:}\mathsf{loc}\{\mathsf{newc}\} \quad a \notin \mathrm{fn}(\Gamma) \quad \Gamma,_{(a)}\{_k a{:}\mathrm{A}\} \vdash'' N}{\Gamma \vdash'' (\nu_k a{:}\mathrm{A})\, N}$$

Figure 12 Extended Runtime Errors

Rules (E-EQ), (E-NEW), (E-PAR) and (E-STR) from Figure 7.

$$\begin{array}{lrl}
\textsc{(e-go}'') & \ell[\![\mathsf{go}\, k.P]\!]_\Gamma \xmapsto{err''} & \text{if } \Gamma(k) \not<: \mathsf{loc}\{\mathsf{go}\} \\
\textsc{(e-subc}'') & \ell[\![(\nu a)\, P]\!]_\Gamma \xmapsto{err''} & \text{if } \Gamma(k) \not<: \mathsf{loc}\{\mathsf{newc}\} \\
\textsc{(e-snd}'') & \ell[\![a!\langle v\rangle\, Q]\!]_\Gamma \xmapsto{err''} & \text{if } \Gamma_\ell(v) \not<: \mathsf{wobj}(\Gamma(\ell,a)) \\
\textsc{(e-rcv}'') & \ell[\![a?(X{:}\mathrm{T})\, P]\!]_\Delta \xmapsto{err''} & \text{if } \mathsf{robj}(\Delta(\ell,a)) \not<: \mathrm{T} \\
\textsc{(e-comm}'') & \ell[\![a!\langle v\rangle\, P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_\Delta \xmapsto{err''} & \text{if } \mathsf{wobj}(\Gamma(\ell,a)) \not<: \mathsf{robj}(\Delta(\ell,a))
\end{array}$$

that an error should occur here. The result of the communication, $\ell[\![a?(y)\ Q]\!]_{\{\ldots,\ell:\mathsf{loc}\{c:\mathrm{C},\,a:\mathrm{A}_{\mathsf{rw}}\}\}}$, is clearly undesirable, since the read capability on a has been fabricated. Note that if $\Gamma(a) = \mathrm{A}_{\mathsf{w}}$, then subject reduction also fails as a result of this communication.

EXAMPLE. As an example of the use of these extended types, consider a server for read/write (get/put) cells similar to the counter server from Section 3.

$$\mathrm{S}(h) \Leftarrow *req?(y@z)\ (\nu cell{:}\mathrm{L}_{\mathsf{cell}})\ z.y!\langle cell\rangle \mid \mathsf{go}\, cell.\mathrm{Cell}(cell, 0)$$

Here "Cell" represents the code for the cell, for example:

$$\begin{aligned}\mathrm{Cell}(h,n) \Leftarrow (\nu s{:}\mathsf{int})\, s!\langle n\rangle &\mid *g?(y@z) \quad s?(x)\ (s!\langle x\rangle \mid z.y!\langle x\rangle)\\ &\mid *p?(y@z,v)\ s?(x)\ (s!\langle v\rangle \mid z.y!\langle\rangle)\end{aligned}$$

Let us use the abbreviations for PS types introduced above. The *allocation type* $\mathrm{L}_{\mathsf{cell}}$ of the cell location *cell* can then be written:

$$\begin{aligned}\mathrm{L}_{\mathsf{cell}} &= \mathsf{loc}\{\mathsf{go}, \mathsf{newc}, g{:}\mathsf{rw}\langle \mathrm{T}_{\mathsf{g}}\rangle, P{:}\mathsf{rw}\langle \mathrm{T}_{\mathsf{p}}\rangle\}\\ \mathrm{T}_{\mathsf{g}} &= \mathsf{w}\langle \mathsf{int}\rangle @\mathsf{loc}\{\mathsf{go}\}\\ \mathrm{T}_{\mathsf{p}} &= (\mathsf{w}\langle\rangle @\mathsf{loc}\{\mathsf{go}\}, \mathsf{int})\end{aligned}$$

Location *cell* must be given at least the type $\mathrm{L}_{\mathsf{cell}}$ in order for the cell code to typecheck (it may also be given a subtype). Note that the channels g and p must be declared with both read and write capabilities as the server reads from them and a user must be able to write to them. The cell requires only the write capability on the response channels it receives on p and g.

The user's capabilities on the cell are determined by the *transmission type* $\mathrm{T}_{\mathsf{req}}$ of channel *req* (which must have type $\mathsf{rw}\langle \mathrm{T}_{\mathsf{req}}\rangle$). If one takes

$$\begin{aligned}\mathrm{T}_{\mathsf{req}} &= \mathsf{w}\langle \mathrm{L}'_{\mathsf{cell}}\rangle @\mathsf{loc}\{\mathsf{go}\}\\ \mathrm{L}'_{\mathsf{cell}} &= \mathsf{loc}\{\mathsf{go}, g{:}\mathsf{w}\langle \mathrm{T}_{\mathsf{g}}\rangle, P{:}\mathsf{w}\langle \mathrm{T}_{\mathsf{p}}\rangle\}\end{aligned}$$

then this type ensures that a cell user cannot "redefine" the methods p or g (by intercepting messages sent on these channels), nor can it create new channels at the cell location.

To emphasize this point consider the following user:

$$\mathrm{U}(h) \Leftarrow (\nu r)\ serv.req!\langle r@h\rangle \mid r?(z)\ \mathrm{U}'(h,z)$$

U requests a cell using the response channel r. Then the system $\underline{\mathbf{S}}(serv) \mid \underline{\mathbf{U}}(k)$ can reduce to

$$\underline{\mathbf{S}}(serv) \mid (\nu cell{:}\mathrm{L}_{\mathsf{cell}})\ \underline{\mathbf{U}}'(k, cell) \mid \underline{\mathbf{Cell}}(cell)$$

If $\mathrm{T}_{\mathsf{req}}$ is as above, then one can be sure that the agent $\mathrm{U}(k, cell)$ has restricted access to *cell* in this system. For example, if U' has the form

$$\mathrm{U}'(h, cell) \Leftarrow \mathsf{go}\, cell.\, p?(x)\ldots$$

then $\mathrm{U}(k)$ will be untypable. Viewed as a tagged system, one can see this definition of U' will also cause a runtime error. After receiving the cell, the tagged user is of the form

$$k[\![\mathsf{go}\, cell.\, p?(x)\ldots]\!]_{\{\ldots,cell:\mathsf{loc}\{\mathsf{go},\, g:\mathsf{w}\langle \mathrm{T}_{\mathsf{g}}\rangle,\, p:\mathsf{w}\langle \mathrm{T}_{\mathsf{p}}\rangle\}\}}$$

Clearly this agent will produce a runtime error when it attempt to read on p.

We should point out that this typing also affords some level of protection to the user. The response channel r is sent to the server with write capability only; thus the server may not intercept other messages that the user may wish to receive on r. Perhaps more important, the user's location is sent without the privilege to create new channels there, keeping the server from performing any computation at the users location.

7. CONCLUSIONS

SUMMARY. We have presented a typing system for controlling the use of resources in languages that describe mobile agents. The typing system has been developed for a distributed version of the π-calculus in which *agents* are located terms of the ordinary π-calculus and *resources* are channels which agents use to communicate. A central assumption of the type system is that every resource is fixed to a particular location, whereas agents are free to move from one location to another. This assumption has lead us to define the notions of *location type* and *location subtyping* which we believe to be novel.

We have developed the typing system in stages. The first typing system used a language of *simple* types in which the only non-trivial types were locations. The second typing system was also defined over simple types, but was more permissive than the first, allowing agents to use simultaneously capabilities acquired from disparate sources. The third and final typing system used an *extended* type language which supported subtyping on resource types (i.e. channel types). Another development of these types, to ensure *receptiveness* of channel names, can be found in [5].

Crucial in the development of the typing systems has been the presence of a *partial meet* operator at all types. The need for such an operator forced us to abandon the notion of resource types proposed by Pierce and Sangiorgi [26] in favor of more general types.

The usefulness of the typing systems has been shown by introducing a *tagged* language in which agents are annotated with their capability sets. The tagged language and the associated reduction relation appear to be novel.

The π-calculus [23] itself is a language for resource access control, using the mechanisms of restriction and scope extrusion to regulate the availability of resources. Distributed π-calculi such as Dπ inherit the same mechanisms, so one might wonder why location types are needed at all. The ultimate goal of our work is to provide a semantics for "partially-typed" systems in which locations need only consider *local* resources when type-checking incoming agents. Obviously, to define such a system, the notion of *local resource* must be clearly understood, leading us to define location types. In this paper, we have attempted to fully explore the properties of location types using various type systems and examples. Although we have not here defined a typing system that meets this ultimate goal, we feel we have laid adequate foundations. Preliminary investigations towards useful partial typing systems, based on the ideas developed in the current paper, have been reported in [28, 18].

RELATED WORK. There are numerous languages now in the literature for describing distributed systems; Dπ is perhaps closest in spirit to [13, 3, 29, 6, 32, 33] which also take as their point of departure the π-calculus, although with each there are significant differences. For example in the join calculus [13] message routing is *automatic* as the restricted syntax ensures that all channels have a unique location at which they are serviced. In Dπ, to send a message to a remote location, an agent must first spawn a sub-agent which moves to that location; locations are more *visible* in Dπ. In Nomadic Pict [33] *threads* are named (called *agents* in [33]) and may move from site to site; however resources belong to agents rather then, as in Dπ, sites. In addition, several of these languages [13, 29, 6] adopt *location movement* as the mechanism for agent mobility.

Many channel-based typing systems for π-calculi and related languages have been proposed. For example in [26], discussed at length in Section 6, Pierce and Sangiorgi define a type system for the π-calculus with read and write capabilities on channels. Sewell [29] generalizes the type system of [26] to distinguish between *local* communication, which can be efficiently implemented, and *non-local* communication. Fournet *et al*. [14] have developed an *ML*-style typing system for the join calculus where channels are allowed a certain amount of polymorphism. Amadio [3] has presented a type system that guarantees that channel names are defined at exactly one location, whereas the type system of Kobayashi *et al*. [21] ensures that some channels are used linearly.

The work closest to ours is that of de Nicola, Ferrari and Pugliese [10]. Their goals are the same as ours, but the specifics of their solution are quite different. They work with a variant of Linda [9] with multiple "tuple spaces". Tuple spaces correspond to locations in our setting, and tuples (named data) correspond to resources. The type system of [10] controls access to tuple spaces, rather than specific tuples. In order to acheive the fine level of control provided by our system, [10] requires that each resource be allocated to a separate tuple space and that adjacency of resources be encoded in their types. Since types in [10] encode an adjacency graph, they are both more general and substantially more complex than our location types.

Static analyses for proving various security properties of programs have also been proposed by several authors; two recent references are [20, 17].

FURTHER WORK. There are many directions in which our research could be extended. Perhaps the most obvious is to allow *recursive types* in the various typing systems we have presented. To do so, one would need to replace every instance of type equality in the paper with a weaker relation such as bisimilarity [31] or equality up to unfolding [29]. We do not expect that the proofs of subject reduction and type safety would be much effected by this change; nor do we foresee any difficulty in extending the proof of finite bounded completeness.

A more difficult issue is the inclusion of more sophisticated notions of *code migration* in our language. Our notion of location, although not static, is one dimensional. In languages such as the join calculus [13], Ambients [6] and the Seal calculus [32], locations are hierarchical, in the sense that locations may contain sub-locations. In such languages, it is *locations* which move, rather than *threads*, and thus *agents* are identified with locations, rather than threads. This has the advantage that agents may be multi-threaded, and thus agents that may move "at any time" are easier to express. Providing type systems for controlling access to resources in the presence of such *location movement* remains an interesting challenge.

A. PROOFS

A.1. Proofs from Section 4.2

We first prove the Weakening Lemma. The result for systems, stated in the text, relies on similar results for threads and values.

PROPOSITION (4.5).

(a) *If* $\Gamma \vdash N$ *and* $\Delta <: \Gamma$ *then* $\Delta \vdash N$.
(b) *If* $\Gamma \vdash_w P$ *and* $\Delta <: \Gamma$ *then* $\Delta \vdash_w P$.
(c) *If* $\Gamma \vdash_w v{:}\mathrm{T}$ *and* $\Delta <: \Gamma$ *then* $\Delta \vdash_w v{:}\mathrm{T}$.

Proof. All three results are proved, in a straightforward manner, by judgment induction (*i.e.* by induction on the length of the type inference). We give one example for each result.

(a) ((N-RUN)) Suppose $\Gamma \vdash \ell[\![P]\!]$ because $\Gamma \vdash_\ell P$. Using the auxiliary results we obtain $\Delta \vdash_\ell P$. Using (N-RUN), we have $\Delta \vdash \ell[\![P]\!]$.

(b) ((T-R)) Suppose $\Gamma \vdash_w u?(X{:}\mathrm{T})\,P$ because:

$$\Gamma \vdash_w u{:}\mathsf{res}\langle\mathrm{T}\rangle \qquad \text{and} \qquad \Gamma, {}_{(X)}\{_w X{:}\mathrm{T}\} \vdash_w P$$

Since we identify terms up to alpha-equivalence, the variables in X can also be chosen to be new to Δ, in which case $\Delta, {}_{(X)}\{_w X{:}\mathrm{T}\}$ is well-defined, and it is easy to see that $\big(\Delta, {}_{(X)}\{_w X{:}\mathrm{T}\}\big) <: \big(\Gamma, {}_{(X)}\{_w X{:}\mathrm{T}\}\big)$. So we may apply induction to the above two statements to obtain:

$$\Delta \vdash_w u{:}\mathsf{res}\langle\mathrm{T}\rangle \qquad \text{and} \qquad \Delta, {}_{(X)}\{_w X{:}\mathrm{T}\} \vdash_w P$$

The rule (T-R) may now be employed to infer $\Delta \vdash_w u?(X{:}\mathrm{T})\,P$ as required.

(c) ((V-LOCAL)) Suppose $\Gamma \vdash_w u{:}\mathrm{A}$ because $\Gamma(w) <: \mathsf{loc}\{u{:}\mathrm{A}\}$. Since $\Delta <: \Gamma$ then by transitivity we have $\Delta(w) <: \mathsf{loc}\{u{:}\mathrm{A}\}$. Using (V-LOCAL), one can the infer $\Delta \vdash_w u{:}\mathrm{A}$, as required. □

As corollaries we immediately have the following:

COROLLARY A.1. (a) *If* $\Gamma \vdash N$ *then* $\Gamma, {}_{(v)}\{_w v{:}\mathrm{T}\} \vdash N$.
(b) *If* $\Gamma, {}_{(v)}\{_w v{:}\mathrm{S}\} \vdash N$ *and* $\mathrm{T} <: \mathrm{S}$ *then* $\Gamma, {}_{(v)}\{_w v{:}\mathrm{T}\} \vdash N$. □

Proposition 4.5 states that well-typing is preserved when the typing environment is augmented. It is also preserved when the typing environment is decreased by omitting all occurrences of identifiers that do not occur free in the system being typed. Let $\Gamma \setminus u$ denote the result of eliminating u from Γ, *i.e.* $(\Gamma \setminus u)(u)$ is undefined and $(\Gamma \setminus u)(w,u)$ is undefined for every w. For any syntactic element t, let "fid(t)" return the free identifiers in t.

LEMMA A.2 (RESTRICTION).
(a) *If* $\Gamma \vdash N$ *and* $u \notin \mathrm{fid}(N)$ *then* $\Gamma \setminus u \vdash N$.
(b) *If* $\Gamma \vdash_w P$ *and* $u \notin \mathrm{fid}(P) \cup \{w\}$ *then* $\Gamma \setminus u \vdash_w P$.
(c) *If* $\Gamma \vdash_w u{:}\mathrm{S}$ *and* $u \notin \mathrm{fid}(u) \cup \{w\}$ *then* $\Gamma \setminus u \vdash_w u{:}\mathrm{S}$.

Proof. In each case the result follows by a straightforward judgment induction. We leave the details to the interested reader. □

As a corollary we have that typing is preserved by scope extrusion:

COROLLARY A.3. *Suppose* e *does not appear free in* M. *Then* $\Gamma \vdash (\nu_\ell e)(M \mid N)$ *if and only if* $\Gamma \vdash M \mid (\nu_\ell e)\,N$.

Proof. We examine the case when e is a channel; the case in which e is a location is similar. Suppose $\Gamma \vdash (\nu_\ell\, a{:}\mathrm{A})\,(M \mid N)$. Then using (N-NEW) and (N-STR), we have that $\Gamma,_{(a)}\{_\ell\, a{:}\mathrm{A}\} \vdash M$ and $\Gamma,_{(a)}\{_\ell\, a{:}\mathrm{A}\} \vdash N$. Applying Lemma A.2 to the first of these we obtain $(\Gamma,_{(a)}\{_\ell\, a{:}\mathrm{A}\}) \setminus a \vdash M$, *i.e.* $\Gamma \vdash M$ since a is new to Γ. Applying (N-NEW) to the second statement we obtain $\Gamma \vdash (\nu_\ell \mathrm{A})\, N$ and therefore (N-STR) gives $\Gamma \vdash M \mid (\nu_\ell\, a{:}\mathrm{A})\, N$.

The converse uses the same arguments, in the reverse direction. □

As a step toward proving subject reduction, note that closed terms are preserved by reduction.

LEMMA A.4. *If N is closed and $N \longrightarrow N'$ then N' is closed.*

Proof. By induction on the judgment $N \longrightarrow N'$. □

The proof of subject reduction for the typing system depends, as is often the case, on a substitution lemma. However in this case before the appropriate version can be proved we need the following technical Lemma.

LEMMA A.5.

(a) *If* $\Gamma \vdash k{:}\mathrm{K}$ *and* $\Gamma,_{(z)}\{z{:}\mathrm{K}\},_{(\widetilde{x})}\{_z\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_w P$ *then* $\Gamma,_{(\widetilde{x})}\{_k\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_{w\{k/z\}} P\{\!\{k/z\}\!\}$.

(b) *If* $\Gamma \vdash k{:}\mathrm{K}$ *and* $\Gamma,_{(z)}\{z{:}\mathrm{K}\},_{(\widetilde{x})}\{_z\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_w u{:}\mathrm{S}$ *then* $\Gamma,_{(\widetilde{x})}\{_k\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_{w\{k/z\}} u\{\!\{k/z\}\!\}{:}\mathrm{S}$.

Proof. For both results the proof is similar. Informally the proof proceeds, in the case of threads, by taking a derivation of the judgment $\Gamma,_{(z)}\{z{:}\mathrm{K}\},_{(\widetilde{x})}\{_z\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_w P$, substituting k for z throughout and thereby obtaining a derivation of $\Gamma,_{(\widetilde{x})}\{_k\, \widetilde{x}{:}\widetilde{\mathrm{T}}\} \vdash_{w\{k/z\}} P\{\!\{k/z\}\!\}$. Formally it is a straightforward induction on type judgments. We omit the details. □

PROOF OF THE SUBSTITUTION LEMMA. We present the proof for the extended type system of Section 6. For this proof only, we write $\vdash$ as shorthand for $\vdash''$. The proofs for the other type systems are somewhat simpler.

LEMMA (4.7). *For any closed value v:*

(a) *If* $\Gamma \vdash_v v{:}\mathrm{T}$ *and* $\Gamma,_{(X)}\{_v\, X{:}\mathrm{T}\} \vdash_w P$ *then* $\Gamma \vdash_{w\{v/X\}} P\{\!\{v/X\}\!\}$.

(b) *If* $\Gamma \vdash_v v{:}\mathrm{T}$ *and* $\Gamma,_{(X)}\{_v\, X{:}\mathrm{T}\} \vdash_w u{:}\mathrm{S}$ *then* $\Gamma \vdash_{w\{v/X\}} u\{\!\{v/X\}\!\}{:}\mathrm{S}$.

Proof. Note that there is no corresponding substitution result for systems, because values must be typed at a specific location.

Throughout the proof we use primes to indicate terms in which the substitution has been performed; *i.e.* for t an element of any syntactic category, t' denotes $t\{\!\{v/X\}\!\}$.

We first prove the result (b) for values. The proof proceeds by induction on the structure of X. There are four cases: X may be w, X may be some identifier other than w, or X may have the the form $\widetilde{X}$ or $\widetilde{x}@z$.

First, suppose that $X = w$. Because $X = w$ it must be that $w' = v = k$ for some k. We proceed by induction on u to show that $\Gamma \vdash_k u'{:}\mathrm{S}$.

- Suppose that $u = w$ and therefore $u' = v = k$. The second premise may be written $\Gamma,_{(w)}\{_v\, w{:}\mathrm{T}\} \vdash_w w{:}\mathrm{S}$. Here we know that T <: S and therefore the result follows by applying weakening to the first premise ($\Gamma \vdash k{:}\mathrm{T}$).

• Suppose that $u = u \neq w$. The second premise may be written $\Gamma, {}_{(w)}\{_v w{:}\mathrm{T}\} \vdash_w u{:}\mathrm{S}$. There are two possibilities. If S is a location type, we must have that $\Gamma(u) <: \mathrm{S}$ and thus $\Gamma \vdash_k u{:}\mathrm{S}$. Otherwise T must be of the form $\mathsf{loc}\{u{:}\mathrm{S}', ...\}$ where $\mathrm{S}' <: \mathrm{S}$. Since $\Gamma \vdash_v k{:}\mathrm{T}$ we can therefore conclude that $\Gamma \vdash_k u{:}\mathrm{S}$.

• In the other cases, $u = \widetilde{u}{:}\widetilde{\mathrm{S}}$ and $u = b@\ell{:}\widetilde{\mathrm{B}}@\mathrm{L}$, the result follows using the innermost induction.

Suppose, instead, that $X = x \neq w$. In this case it must be that $w' = w$. Again we proceed by induction on u to show that $\Gamma \vdash_w u'{:}\mathrm{S}$.

• Suppose that $u = x$ and therefore $u' = v$. Either S is a location type and so by the first premise $\Gamma \vdash v{:}\mathrm{S}$, or S is another type and so v must be equal to w and again the first premise give the required result $\Gamma \vdash_w v{:}\mathrm{S}$.

• Suppose that $u = u \neq x$. The result is immediate by applying the Restriction Lemma (Lemma A.2) to the second premise.

• Again, the other cases follow by straightforward induction.

Suppose $X = \widetilde{X}{:}\widetilde{\mathrm{T}}$. Therefore v must have the form $\widetilde{v}$ and by assumption we have that:

$$\Gamma \vdash_v \widetilde{v}{:}\widetilde{\mathrm{T}} \quad \text{and} \quad \Gamma, {}_{(\widetilde{X})}\{_v \widetilde{X}{:}\widetilde{\mathrm{T}}\} \vdash_w u{:}\mathrm{S}$$

We can rewrite this as:

$$\Gamma \vdash_v v_1{:}\mathrm{T}_1, ..., v_n{:}\mathrm{T}_n \quad \text{and} \quad \Gamma, {}_{(X_1)}\{_v X_1{:}\mathrm{T}_1\}, ..., {}_{(X_n)}\{_v X_n{:}\mathrm{T}_n\} \vdash_w u{:}\mathrm{S}$$

Using induction we have:

$$\Gamma, {}_{(X_1)}\{_v X_1{:}\mathrm{T}_1\}, ..., {}_{((X_{n-1})}\{_v (X_{n-1}{:}\mathrm{T}_{n-1})\} \vdash_{w\{^{v_n}\!/\!_{X_n}\}} u\{\!|^{v_n}\!/\!_{X_n}|\!\}{:}\mathrm{S}$$

Repeating this process n times yields $\Gamma \vdash_{w'} u'{:}\mathrm{S}$, as desired.

Finally, suppose $X = \widetilde{x}@z{:}\widetilde{\mathrm{A}}@\mathrm{K}$. Therefore v must have the form $\widetilde{a}@k$ and by assumption we have that:

$$\Gamma \vdash_v \widetilde{a}@k{:}\widetilde{a}@\mathrm{K} \quad \text{and} \quad \Gamma, {}_{(\widetilde{a}@k)}\{_v \widetilde{a}@k{:}\widetilde{a}@\mathrm{K}\} \vdash_w u{:}\mathrm{S}$$

We can rewrite this as:

$$\Gamma \vdash k{:}\mathrm{K} \quad \text{and} \quad \Gamma \vdash_k \widetilde{a}{:}\widetilde{\mathrm{A}} \quad \text{and} \quad \Gamma, {}_{(z)}\{z{:}\mathrm{K}\}, {}_{(\widetilde{x})}\{_z \widetilde{x}{:}\widetilde{\mathrm{A}}\} \vdash_w u{:}\mathrm{S}$$

Using Lemma A.5 we have:

$$\Gamma, {}_{(\widetilde{x})}\{_k \widetilde{x}{:}\widetilde{\mathrm{A}}\} \vdash_{w\{^k\!/\!_z\}} u\{\!|^k\!/\!_z|\!\}{:}\mathrm{S}$$

Applying induction yields $\Gamma \vdash_{w'} u'{:}\mathrm{S}$, as desired.

Having established the result for values, we now prove the result (a) for threads:

$$\Gamma \vdash_v v{:}\mathrm{T} \text{ and } \Gamma, {}_{(X)}\{_v X{:}\mathrm{T}\} \vdash_w P \text{ imply } \Gamma \vdash_{w'} P'$$

Again we proceed by induction on the structure of X. The inductive cases are as before, so we only present the base case where X is an identifier x. This case is established by a secondary induction on the judgment $\Gamma, {}_{(X)}\{{}_{v}X{:}\mathrm{T}\} \vdash_{w} P$. Most of the cases in the secondary induction are straightforward, the exceptions being the cases for input and channel restriction. We show these two cases.

First consider the case for $(\text{T-R}'')$. Our proof obligation is to show:

$$\Gamma \vdash_{v} v{:}\mathrm{T} \text{ and } \Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} u?(Y{:}\mathrm{S})\,Q \text{ imply } \Gamma \vdash_{w'} u'?(Y{:}\mathrm{S})\,Q' \qquad (*)$$

There are two cases to consider, $x = w$ and $x \neq w$. First suppose that $x = w$. Here T must be a location type, say K and therefore v must be a location name, say k. The premises in (*) may therefore be written:

$$\Gamma \vdash k{:}\mathrm{K} \quad \text{and} \quad \Gamma, {}_{(w)}\{w{:}\mathrm{K}\} \vdash_{w} u?(Y{:}\mathrm{S})\,Q$$

From $(\text{T-R}'')$ we have:

$$\Gamma, {}_{(w)}\{w{:}\mathrm{K}\} \vdash_{w} u{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{S}\rangle\} \quad \text{and} \quad \Gamma, {}_{(w)}\{w{:}\mathrm{K}\}, {}_{(Y)}\{{}_{w}Y{:}\mathrm{S}\} \vdash_{w} Q$$

Using Lemma A.5 twice, we obtain:

$$\Gamma \vdash_{k} u'{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{S}\rangle\} \quad \text{and} \quad \Gamma, {}_{(Y)}\{{}_{k}Y{:}\mathrm{S}\} \vdash_{k} Q'$$

Finally, $(\text{T-R}'')$ can be applied to arrive at the desired conclusion, $\Gamma \vdash_{k} u'?(Y{:}\mathrm{S})\,Q'$.

Continuing the case for $(\text{T-R}'')$, suppose $x \neq w$. This case is a standard application of induction. The details are as follows. Using the second premise of (*) and $(\text{T-R}'')$, we can conclude that:

$$\Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} u{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{S}\rangle\} \quad \text{and} \quad \Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\}, {}_{(Y)}\{{}_{w}Y{:}\mathrm{S}\} \vdash_{w} Q$$

Note that since $w \neq x$, we may rewrite the above as:

$$\Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} u{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{S}\rangle\} \quad \text{and} \quad \Gamma, {}_{(Y)}\{{}_{w}Y{:}\mathrm{S}\}, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} Q$$

Now we may use the inner induction to conclude:

$$\Gamma \vdash_{w} u'{:}\mathsf{res}\{\mathsf{r}\langle \mathrm{S}\rangle\} \quad \text{and} \quad \Gamma, {}_{(Y)}\{{}_{w}Y{:}\mathrm{S}\} \vdash_{w} Q'$$

Therefore using $(\text{T-R}'')$ we have, as desired, $\Gamma \vdash_{w} u'?(Y{:}\mathrm{S})\,Q'$.

Now consider the case for channel restriction $(\text{T-NEWC}'')$. In this case the proof obligation is:

$$\Gamma \vdash_{v} v{:}\mathrm{T} \text{ and } \Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} (\nu a{:}\mathrm{A})\,Q \text{ imply } \Gamma \vdash_{w'} (\nu a{:}\mathrm{A})\,Q' \qquad (**)$$

Using the second premise of (**) and $(\text{T-NEWC}'')$, we can conclude that:

$$\Gamma, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\}, {}_{(a)}\{{}_{w}a{:}\mathrm{A}\} \vdash_{w} Q \qquad (***)$$

At this point we must consider two cases, either $x \neq w$ or $x = w$. First suppose that $x \neq w$. Then (***) can be rewritten as $\Gamma, {}_{(a)}\{{}_{w}a{:}\mathrm{A}\}, {}_{(x)}\{{}_{v}x{:}\mathrm{T}\} \vdash_{w} Q$ and we can apply induction to get $\Gamma,_{(a)}\{_w\, a{:}\mathrm{A}\} \vdash_{w'} Q'$ and then (T-NEWC$''$), since the second premise of (**) implies $a \notin \mathsf{fn}(\gamma)$ and $\Gamma(w) <: \mathsf{loc}\{\mathsf{newc}\}$ to get $\Gamma \vdash_{w'} (\nu a{:}\mathrm{A})\, Q'$, as required.

On the other hand if $x = w$, then we must use Lemma A.5. Since $x = w$, it must be that v is a location name and thus $v = k$ and $\mathrm{T} = \mathrm{K}$ for some k, K. We can therefore rewrite the first premise of (**) and the statement (***) as:

$$\Gamma \vdash k{:}\mathrm{K} \quad \text{and} \quad \Gamma,_{(w)}\{w{:}\mathrm{K}\},_{(a)}\{_w\, a{:}\mathrm{A}\} \vdash_w Q$$

These can be applied to Lemma A.5 to yield $\Gamma,_{(a)}\{_k\, a{:}\mathrm{A}\} \vdash_k Q'$ and thus, using (T-NEWC$''$), $\Gamma \vdash_k (\nu a{:}\mathrm{A})\, Q'$, as required. □

PROOF OF THE SUBJECT REDUCTION THEOREM.

THEOREM (4.6).
(a) *If* $N \equiv N'$ *then* $\Gamma \vdash N$ *if and only if* $\Gamma \vdash N'$.
(b) *If* $N \longrightarrow N'$ *then* $\Gamma \vdash N$ *implies* $\Gamma \vdash N'$.

Proof. The first statement is proved by induction on the proof of $N \equiv N'$. The main axiom, scope extrusion (S-EXTR), is covered by the Corollary A.3. The other axioms and rules are straightforward calculations left to the interested reader.

The second statement is proved by induction on the proof of $N \longrightarrow N'$. The rule (R-STR) follows from the first part; the remaining rules are, again, straightforward calculations. We give two examples.

- (R-GO) states $\ell[\![\mathsf{go}\, u.\, P]\!] \longrightarrow k[\![P]\!]$. By supposition $\Gamma \vdash \ell[\![\mathsf{go}\, u.\, P]\!]$. Then using (N-RUN) we have $\Gamma \vdash_\ell \mathsf{go}\, u.\, P$. Then using (T-GO), $\Gamma \vdash_k P$ and therefore by (N-RUN) $\Gamma \vdash k[\![P]\!]$.

- (R-COMM) states $\ell[\![a!\langle v\rangle\, P]\!] \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!] \longrightarrow \ell[\![P]\!] \mid \ell[\![Q\{\!|{}^{v}\!/\!{}_{X}|\!\}]\!]$. Suppose $\Gamma \vdash \ell[\![a!\langle v\rangle\, P]\!] \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]$. To satisfy the proof obligation, it is sufficient to show that $\Gamma \vdash_\ell P$ and $\Gamma \vdash_\ell Q\{\!|{}^{v}\!/\!{}_{X}|\!\}$. The first is easy to establish from the hypothesis, which entails $\Gamma \vdash_\ell a!\langle v\rangle\, P$. Using the hypothesis and the rules for typing it must also be that:

$$\Gamma \vdash_\ell v{:}\mathrm{T} \qquad \Gamma \vdash_\ell a{:}\mathsf{res}\langle \mathrm{T}\rangle \qquad \Gamma \vdash_\ell a{:}\mathsf{res}\langle \mathrm{T}\rangle \qquad \Gamma,_{(X)}\{_\ell X{:}\mathrm{T}\} \vdash_\ell Q$$

Note here that v is a closed value. We can apply the Substitution Lemma to obtain $\Gamma \vdash_\ell Q\{\!|{}^{v}\!/\!{}_{X}|\!\}$, as required. □

A.2. Proofs from Section 4.4

In this subsection we prove the Subject Reduction Theorem for the tagged language. We first present a lemma characterizing the definition $\{_\ell v{:}\mathrm{T}\}$ given on page 21.

LEMMA A.6.
(a) $\{_\ell v{:}\mathrm{T}\} \vdash_\ell v{:}\mathrm{T}$
(b) $\Xi \vdash_\ell v{:}\mathrm{T}$ *implies* $\Xi <: \{_\ell v{:}\mathrm{T}\}$.

Proof. By induction on v, using the definition of typing for values and the definition of the notation "$\{_\ell v{:}\mathrm{T}\}$" given on page 21. □

THEOREM (4.8). *For all tagged systems*
(a) *If* $N \equiv N'$ *then* $\Gamma \Vdash N$ *if and only if* $\Gamma \Vdash N'$.

(b) *If* $N \longmapsto N'$ *then* $\Gamma \Vdash N$ *implies* $\Gamma \Vdash N'$.

Proof. As in Theorem 4.6, the proof of (a) is straightforward by induction on the derivation of $N \equiv N'$. We show the argument for the rule ($\textsc{s}_\textsc{t}$-NEW), applied to channels, which states

$$\ell[\![(\nu a{:}\mathrm{A})\, P]\!]_\Delta \equiv (\nu_\ell\, a{:}\mathrm{A})\, \ell[\![P]\!]_{\Delta,_{(a)}\{_\ell\, a:\mathrm{A}\}}$$

Suppose $\Gamma \Vdash \ell[\![(\nu a{:}\mathrm{A})\, P]\!]_\Delta$, and therefore $\Gamma <: \Delta$ and $\Delta \Vdash_\ell (\nu a{:}\mathrm{A})\, P$. Then using (T-NEW) $\Delta,_{(a)}\{_\ell a{:}\mathrm{A}\} \Vdash_\ell P$. From which we obtain from (T-GO) that $\Delta,_{(a)}\{_\ell a{:}\mathrm{A}\} \Vdash \ell[\![P]\!]$. Since $\Gamma,_{(a)}\{_\ell a{:}\mathrm{A}\} <: \Delta,_{(a)}\{_\ell a{:}\mathrm{A}\}$ we can apply the new typing rule for tagged agents, ($\textsc{n}_\textsc{t}$-RUN), to obtain $\Gamma,_{(a)}\{_\ell a{:}\mathrm{A}\} \Vdash \ell[\![P]\!]_{\Delta,_{(a)}\{_\ell a:\mathrm{A}\}}$ and an application of (N-NEWC) gives the required $\Gamma \Vdash (\nu_\ell\, a{:}\mathrm{A})\, \ell[\![P]\!]_{\Delta,_{(a)}\{_\ell a:\mathrm{A}\}}$. The argument in the other direction is much the same.

The proof of (b) is by induction on why $N \longmapsto N'$. The only non-trivial case is the communication rule for tagged threads, ($\textsc{r}_\textsc{t}$-COMM). So suppose

$$\Xi \Vdash \ell[\![a!\langle v\rangle\, P]\!]_\Gamma \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]_\Delta,$$

that is, $\Gamma \Vdash_\ell a!\langle v\rangle\, P$, $\Delta \Vdash_\ell a?(X{:}\mathrm{T})\ Q$ and $\Xi <: \Gamma$, $\Xi <: \Delta$. We must show

$$\Xi \Vdash \ell[\![P]\!]_\Gamma \mid \ell[\![Q\{\!|v/X|\!\}]\!]_{\Delta'}$$

where Δ' denotes $\Delta \sqcap \{_\ell v{:}\mathrm{T}\}$. Thus our proof obligations are three: we must show that $\Gamma \Vdash_\ell P$, $\Delta' \Vdash_\ell Q\{\!|v/X|\!\}$ and $\Xi <: \Delta'$.

Most of the required work has already been carried out in Theorem 4.6. The proof of $\Gamma \vdash_\ell P$ is identical, thus satisfying the first obligation.

Since $\Xi <: \Gamma$ and $\Xi <: \Delta$ it follows that $\Gamma(\ell, a)$ and $\Delta(\ell, a)$ must coincide at the type $\mathsf{res}\langle\mathrm{T}\rangle$ (because there is no subtyping on channels). Using Lemma A.6b we have $\{_\ell v{:}\mathrm{T}\} \vdash_\ell v{:}\mathrm{T}$, and therefore $\Delta' \vdash_\ell v{:}\mathrm{T}$, by weakening (Proposition 4.5). The hypothesis also implies $\Delta,_{(X)}\{_\ell X{:}\mathrm{T}\} \vdash_\ell Q$ and thus, again by weakening we have $\Delta',_{(X)}\{_\ell X{:}\mathrm{T}\} \vdash_\ell Q$. We can now use the Substitution Lemma to obtain the second obligation, $\Delta' \vdash_\ell Q\{\!|v/X|\!\}$.

Using the premise and type rules we have $\Gamma \vdash_\ell v{:}\mathrm{T}$; thus by weakening we have $\Xi \vdash_\ell v{:}\mathrm{T}$ and therefore $\Xi <: \{_\ell v{:}\mathrm{T}\}$, by Lemma A.6a. Using this and the fact that $\Xi <: \Delta$ we obtain the third obligation, $\Xi <: \Delta'$. □

A.3. Proofs from Section 5

The proofs of the following results extend immediately to the improved type rules: type specialization (Lemma 4.4), weakening (Proposition 4.5), substitution (Lemma 4.7, and tagged/untagged reduction (Proposition 4.10).

THEOREM (5.1).
(a) *If* $\Gamma \vdash' N$ *and* $N \longrightarrow N'$ *then* $\Gamma \vdash' N'$.
(b) *If* $\Gamma \Vdash' N$ *and* $N \longmapsto N'$ *then* $\Gamma \Vdash' N'$.
(c) $\Gamma \Vdash' N$ *implies* $N \not\overset{err}{\longmapsto}$.

Proof. The new type rules do not affect terms that can be shown to be structurally equivalent and therefore the results for the structural equivalences follow from Theorem 4.6 and Theorem 4.8. The proof of (c) is also unchanged from that of Theorem 4.12.

Both (a) and (b) follow by induction on the definition of reduction. The argument is much as in Theorem 4.6 and Theorem 4.8. The only case which changes is that for successful matching location names. We treat the untagged case; the tagged case is similar.

The reduction rule (R-EQ_1) states:

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

So suppose $\Gamma \vdash' \ell[\![\mathsf{if}\ k = k\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!]$. This must be typed using $(\text{T-EQL}')$, thus we have that for some T, S, $\Gamma \vdash' u{:}\mathrm{T}$, $\Gamma \vdash' u{:}\mathrm{S}$ and $\Gamma \sqcap \{u{:}\mathrm{T}, u{:}\mathrm{S}\} \vdash_\ell' P$. This must mean that $\Gamma(k) <: \mathrm{S}$ and $\Gamma(k) <: \mathrm{T}$; thus $\Gamma(k) <: \mathrm{T} \sqcap \mathrm{S}$. It follows from weakening (Proposition 4.5) that $\Gamma \vdash' \ell[\![P]\!]$. □

A.4. Proofs from Section 6

The proofs of the following results extend immediately to the extended type system: type specialization (Lemma 4.4), weakening (Proposition 4.5), substitution (Lemma 4.7, and tagged/untagged reduction (Proposition 4.10).

THEOREM (THEOREM 6.3).
(a) *If* $\Gamma \vdash'' N$ *and* $N \longrightarrow N'$ *then* $\Gamma \vdash'' N'$.
(b) *If* $\Gamma \Vdash'' M$ *and* $M \longmapsto M'$ *then* $\Gamma \Vdash'' M'$.
(c) $\Gamma \Vdash'' M$ *implies* $M \overset{err''}{\not\longmapsto}$.

Proof. The proof of the result for the structural congruence in Theorem 4.6 extends directly to both the tagged and untagged language.

The proofs of (a) and (b) are, as usual, by induction on the definition of reduction. We discuss the untagged case. The only interesting case is (R-COMM), which states:

$$\ell[\![a!\langle v\rangle\, P]\!] \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!] \longrightarrow \ell[\![P]\!] \mid \ell[\![Q\{\!\!\{{}^{v}\!/\!{}_{X}\}\!\!\}]\!]$$

Suppose $\Gamma \vdash'' \ell[\![a!\langle v\rangle\, P]\!] \mid \ell[\![a?(X{:}\mathrm{T})\, Q]\!]$. To satisfy the proof obligation, it is sufficient to show that $\Gamma \vdash_\ell'' P$ and $\Gamma \vdash_\ell'' Q\{\!\!\{{}^{v}\!/\!{}_{X}\}\!\!\}$. The first is easy to establish from the supposition, which entails $\Gamma \vdash_\ell'' a!\langle v\rangle\, P$. Using the supposition it must also be that for some S:

$$\Gamma \vdash_\ell'' v{:}\mathsf{S} \qquad \Gamma \vdash_\ell'' a{:}\mathsf{res}\{\mathsf{w}\langle\mathrm{S}\rangle\} \qquad \Gamma \vdash_\ell'' a{:}\mathsf{res}\{\mathsf{r}\langle\mathrm{T}\rangle\} \qquad \Gamma,_{(X)}\{{}_\ell X{:}\mathrm{T}\} \vdash_\ell'' Q$$

By the rules on valid types it must be that $\mathrm{S} <: \mathrm{T}$. And therefore by type specialization, $\Gamma \vdash_\ell'' v{:}\mathrm{T}$. We can now apply the Substitution Lemma to obtain $\Gamma \vdash_\ell'' Q\{\!\!\{{}^{v}\!/\!{}_{X}\}\!\!\}$, as required.

To prove (c), we proceed as in Theorem 4.12, proving the contrapositive (that $M \overset{err''}{\longmapsto}$ implies for no Γ can we prove $\Gamma \Vdash'' M$) by induction on the definition of $M \overset{err''}{\longmapsto}$. As before each of the cases is straightforward, the new rules in Figure 7 presenting no additional difficulty. We omit the details. □

REFERENCES

1. 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.
2. R. Amadio and S. Prasad. Modelling IP mobility. Internal Report 244, Laboratoire d'Informatique de Marseille, 1997.
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. 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.

5. G. Boudol and R. Amadio and C. Lhoussaine. The receptive distributed pi-calculus. In Proceedings of the FST-TCS'99, volume 1282 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999.

6. L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS'98*, Springer-Verlag, 1998.

7. 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.

8. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

9. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science, pages 66–76. Springer-Verlag, 1995.

10. De Nicola, R., Ferrari, G. and Pugliese, R., Klaim: a Kernel Language for Agents Interaction and Mobility, IEEE Trans. on Software Engineering, Vol.24(5), 1998.

11. K. Mani Chandy *et al*. A world-wide distributed system using java and the internet. In *IEEE International Symposium on High Performance Distributed Computing*. IEEE, August 1996.

12. 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.

13. 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.

14. Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *CONCUR: Proceedings of the International Conference on Concurrency Theory*, Lecture Notes in Computer Science, Warsaw, August 1997. Springer-Verlag.

15. General Magic Inc. Agent technology. `hhttp://www.genmagic.com/html/agent_overview.html`, 1997.

16. 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.

17. Nevin Heintz and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

18. Matthew Hennessy and James Riely. *Type-Safe Execution of Mobile Agents in Anonymous Networks*. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, number 1603 in Lecture Notes in Computer Science, Springer-Verlag, 1999.

19. IBM Corp. The IBM aglets workbench. `http://www.trl.ibm.co.jp/aglets/`, 1996.

20. Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4), 1997.

21. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Paris, January 1996. ACM Press.

22. 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.

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

24. ObjectSpace Inc. Objectspace voyager. `http://www.objectspace.com/voyager`, 1997.

25. C. Perkins. IP mobility support. RFC 2002, 1996.

26. 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.

27. James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

28. James Riely and Matthew Hennessy. Trust and Partial Typing in Open Systems of Mobile Agents. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Antonio, January 1999. ACM Press.

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

30. Sun Microsystems Inc. Java home page. `http://www.javasoft.com/`, 1995.

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

32. J.Vitek and G. Castagna. A Calculus of Secure Mobile Computations. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, number 1603 in Lecture Notes in Computer Science, Springer-Verlag, 1999.

33. Pawel T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents". *IEEE Concurrency*, 8(2):42–52, 2000.