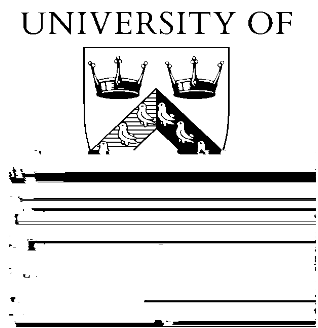
NI ERbI Y OF b bbEX

COMP ER bCIENCE



sour A ss Contro n st s o ob A nts

Matthew Hennessy and Ja es Rie y

Report February

Resource Access Control in Systems of Mobile Agents

MATTHEW HENNESSY AND JAMES RIELY

ABSTRACT

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,[1] is a distributed variant of the π-calculus [21], and thus the *resources* of interest are *channels*

$\mathsf{chan}\langle \mathrm{L} \rangle$, 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 $\mathsf{chan}\langle \mathsf{loc}\{a{:}\mathrm{A}, b{:}\mathrm{B}\}\rangle$, then the recipient gains access to channels a and b at ℓ. Instead, when the same name is communicated over a channel of type $\mathsf{chan}\langle \mathsf{loc}\{a{:}\mathrm{A}\}\rangle$, 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:

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

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 [24], who first studied channel subtyping for π-calculi. 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 [24]. 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:

$$\ell[\![p]\!] \mid (\nu a{:}\{k{:}\mathrm{A}\})(\ell[\![q]\!] \mid k[\![r]\!])$$

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:

$$\ell[\![b?(x)p_1 \mid c?(Y)p_2]\!]$$

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 $k[\![\ell :: b!\langle d\rangle\, r']\!]$. This term can reduce to $\ell[\![b!\langle d\rangle\, r']\!]$, enabling local communication between p and r. After the communication the system is:

$$\ell[\![p_1\{\!\!\{d/x\}\!\!\}]\!] \mid \ell[\![c?(Y)p_2]\!] \mid (\nu a{:}\{k\})(\ell[\![q]\!] \mid \ell[\![r']\!])$$

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

In order for $k[\![\ell :: b!\langle d\rangle\, r']\!]$ 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 $\ell[\![c!\langle k[a]\rangle\, q']\!]$. We motivate this syntactic distinction in our discussion of types on page 6.

2.1 Syntax

The syntax of the language is given in Table 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 Table 1. The type system used in Sections 4 and 5 is described on page 6. For the moment, suffice it to say that names are assigned *atomic types*, E-G, which may be either *channel types*, A-C, and *location types*, K-M. Variables may additionally be assigned one of the *compound*

TABLE 1 Syntax

Systems:	Threads:	Ids, Patterns, Values:
$P\text{-}R \ ::= \ \mathsf{nil} \mid P \mid Q$	$p\text{-}r \ ::= \ \mathsf{nil} \mid p \mid q$	$u\text{-}w \ ::= e \mid x$
$\mid \ (\nu a{:}\Lambda)\, P$	$\mid \ (\nu a{:}\mathrm{A})\, p$	$X\text{-}Z \ ::= x \mid z[\widetilde{x}] \mid \widetilde{X}$
$\mid \ (\nu m{:}\mathrm{M})\, P$	$\mid \ (\nu m{:}\mathrm{M})\, p$	$U\text{-}W \ ::= u \mid w[\widetilde{u}] \mid \widetilde{U}$
$\mid \ \ell[\![p]\!]$	$\mid \ u :: p$	
	$\mid \ u!\langle V\rangle\, p \mid u?(X{:}\zeta)\, q$	
	$\mid \ {*p} \mid \mathsf{if}\ u = v\ \mathsf{then}\ p\ \mathsf{else}\ q$	

or *value* types, ranged over by ζ and ξ. 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.

Systems, Agents and Threads.

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 "if $u = v$ then p" instead of "if $u = v$ then p else nil" and "if $u \neq v$ then q" instead of "if $u = v$ then nil 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:

K-M ::

TABLE 2 Reduction Relation

$$
\begin{array}{rrcl}
(\textsf{r-move}) & \ell[\![k :: p]\!] & \longrightarrow & k[\![p]\!] \\
(\textsf{r-comm}) & \ell[\![a!\langle V\rangle\, p]\!] \mid \ell[\![a?(X)\, q]\!] & \longrightarrow & \ell[\![p]\!] \mid \ell[\![q\{\!|^{V}\!/_{X}|\!\}]\!] \\
(\textsf{r-eq}_1) & \ell[\![\textsf{if } e = e \textsf{ then } p \textsf{ else } q]\!] & \longrightarrow & \ell[\![p]\!]
\end{array}
$$

(r

TABLE 3 Structural Equivalence

(s-nil)	$\ell[\![\mathsf{nil}]\!] \equiv \mathsf{nil}$	
(s-split)	$\ell[\![p \mid q]\!] \equiv \ell[\![p]\!] \mid \ell[\![q]\!]$	
(s-itr)	$\ell[\![*p]\!] \equiv \ell[\![p]\!] \mid \ell[\![*p]\!]$	
(s-newc)	$\ell[\![(\nu a{:}\mathrm{A})\,p]\!] \equiv (\nu a{:}\{\ell{:}\mathrm{A}\})\,\ell[\![p]\!]$	
(s-newl)	$\ell[\![(\nu k{:}\mathrm{K})\,p]\!] \equiv (\nu k{:}\mathrm{K})\,\ell[\![p]\!]$	if $k \neq \ell$
(s-extr)	$Q \mid (\nu e)\,P \equiv (\nu e)(Q \mid P)$	if $e \notin \mathrm{fn}(Q)$

variables).

The rule r-comm

allows an agent to split into two independent agents ($\ell[\![P \mid Q]\!] \equiv \ell[\![P]\!] \mid \ell[\![Q]\!]$). The rule s-nil allows for garbage collection of terminated agents, whereas s-itr provides a standard interpretation of iteration. Note that when a channel name is extracted from a thread using s-newc ($\ell[\![(\nu a{:}\mathrm{A})\,p]\!] \equiv (\nu a{:}\{\ell{:}\mathrm{A}\})\,\ell[\![p]\!]$) it is necessary to record in the "global" restriction the location at which the name was defined.

3 Examples

In order to simplify the presentation of examples, we will assume a set of basic

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

U

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 $\mathsf{chan}\langle \mathrm{L}_i \rangle$, where:

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

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

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

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

$$\begin{aligned}\mathrm{U}_i(h, cnt) &\Leftarrow a_i!\langle cnt\rangle\\ \mathrm{F}_i(h\end{aligned}$$

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 P is well typed and $P \longrightarrow P'$ then P' 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 [20] describes type safety as freedom from *arity mismatches*. For example, the system

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

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 [24]; we presented an alternative formulation in [25], 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

TABLE 4 Simple Types

Types:

LType: $\mathsf{K} ::= \mathsf{loc}\{\widetilde{a}$:

Subtyping:

has a greatest lower bound. That is, there exists a *partial meet*

$\Gamma \vdash P$ is read "the term P 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 P. 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 $\widetilde{\zeta}$, or an existential type $\mathrm{L}[(\mathrm{A}_1 \ldots \mathrm{A}_n)]$ (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 $\mathsf{loc}\{\widetilde{u}{:}\widetilde{\zeta}\}$. 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:

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

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-.8Tf16 [illegible]

value V is well formed at w and has at least the capabilities specified by ζ. Recall that in values, we treat u:L as shorthand for $u[\,]$: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, $\mathsf{t\text{-}w_t}$ and $\mathsf{t\text{-}r_t}$. 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

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

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

In the rule for code movement, $\mathsf{t\text{-}move_t}$, the location of the thread changes: to type $\Gamma \vdash_w u :: p$ 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 $\mathsf{t\text{-}newl_t}$, $\mathsf{t\text{-}newc_t}$ simply augment the typing environment in the appropriate manner. The other rules are purely structural.

The extension to systems is also straightforward. The only interesting rule is $\mathsf{t\text{-}run_s}$ for located threads, which has the same structure as $\mathsf{t\text{-}move_t}$. 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 Table 5. The following property is immediate from the definition of subtyping.

LEMMA

TABLE 5 A Type System

Threads:

$$(\mathsf{t\text{-}r_t})\ \frac{\Gamma \vdash_w u{:}\mathsf{chan}\langle\zeta\rangle \quad \Gamma, {}_wX{:}\zeta \vdash_w q}{\Gamma \vdash_w u?(X{:}\zeta)\, q} \qquad (\mathsf{t\text{-}w_t})\ \frac{\Gamma \vdash_w u{:}\mathsf{chan}\langle\zeta\rangle,\ V{:}\zeta,\ p}{\Gamma \vdash_w u!\langle V\rangle\, p}$$

$$(\mathsf{t\text{-}eql_t})\ \frac{\Gamma \vdash_w u{:}\mathrm{L}, v{:}\mathrm{L}, p, q}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ p\ \mathsf{else}\ q} \qquad (\mathsf{t\text{-}eqc_t})\ \frac{\Gamma \vdash_w u{:}\mathrm{A}, v{:}\mathrm{A}, p, q}{\Gamma \vdash_w}$$

4.3 Examples

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

$$P = \ell[\![c?(z{:}\mathrm{K})\ z :: a!\langle V\rangle]\!]$$

At location ℓ, P receives location z on channel c

TABLE 6

In the rule $\mathsf{r_t}$-comm, which states

$$\ell[\![a!\langle V\rangle\, p]\!]_\Gamma \mid \ell[\![a?(X{:}\zeta)\, q]\!]_\Delta \longmapsto \ell[\![p]\!]_\Gamma \mid \ell[\![q\{\!|{}^{V}\!/\!_{X}|\!\}]\!]_{\Delta\sqcap\{_\ell V:\zeta\}}$$

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 $\mathrm{obj}(\mathrm{A})$ denotes the transmission type, or *object type* used in the channel type A, *i.e.* obj(chan

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 P to range over untagged terms and Q to range over tagged terms. The function "tag_Γ" is defined on the structure of systems as follows:

$$\begin{aligned} \text{tag}_\Gamma(\mathsf{nil}) &= \{\mathsf{nil}\} \\ \text{tag}_\Gamma(\ell[\![p]\!]) &= \{\ell[\![p]\!]_\Delta \quad \mid \Gamma \leq \Delta \text{ and } \Delta \vdash_\ell p \;\} \\ \text{tag}_\Gamma(P_1 \mid P_2) &= \{Q_1 \mid Q_2 \quad \mid Q_i \in \text{tag}_\Gamma(P \end{aligned}$$

TABLE 7 Runtime Errors

(e-eql)	$\ell[\![\mathsf{if}\ k = m\ \mathsf{then}\ p\ \mathsf{else}\ q]\!]_\Gamma \overset{err}{\longmapsto}$	if $k \notin \Gamma$ or $m \notin \Gamma$
(e-eqc)	$\ell[\![\mathsf{if}\ a = b\ \mathsf{then}\ p\ \mathsf{else}\ q]\!]_\Gamma \overset{err}{\longmapsto}$	if $a \notin \Gamma(\ell)$ or $b \notin \Gamma(\ell)$
(e-snd)	$\ell[\![a!\langle V\rangle\, p]\!]_\Gamma \overset{err}{\longmapsto}$	if $\Gamma_\ell(V) \not\leq \mathsf{obj}(\Gamma(\ell, a))$
(e-rcv)	$\ell[\![a?(X$	

bilities, it doesn't keep receivers from doing so. For example, let

$$\mathrm{A} = \mathsf{chan}\langle \mathsf{loc}\{b{:}\mathrm{B}\}\rangle \qquad \mathrm{C} = \mathsf{chan}\langle \mathrm{A}\rangle$$

and suppose $\Gamma(\ell, c) = \mathsf{chan}\langle \mathrm{A}\rangle$. 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 $\Delta(k)$ 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. Thus we are led to the refined rules given in Table 7. Using these rules (in particular e-rcv), 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 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

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

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

$$\ell[\![a!\langle V\rangle p]\!]_\Gamma \xmapsto{err} \quad \text{if } \Gamma_\ell(V) \not\leq \mathsf{obj}(\Gamma(\ell, a))$$

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

Using the premise and the rule $\mathsf{t_t\text{-}run_s}$, we have that $\Gamma \vdash_\ell a!\langle V\rangle p$. This judgment can only be achieved using $\mathsf{t\text{-}w_t}$, and therefore we have that $\Gamma \vdash_\ell a{:}\mathsf{chan}\langle\zeta\rangle, V{:}\zeta$ for some ζ. Using Lemma 4.11, we therefore may conclude that $\Gamma_\ell(V) \leq \zeta$. Using similar reasoning, we have $\mathsf{chan}\langle\zeta\rangle = \Gamma(\ell, a)$, and thus $\zeta = \mathsf{obj}(\Gamma(\ell, a))$. We therefore may conclude that $\Gamma_\ell(V) \leq \mathsf{obj}(\Gamma(\ell, a))$, as desired. □

GThe Ty77]00-113eft and Subject Reduction The47(or)-6.9987(e46(m)-2.00195(s)-709.995(e)4((

Γ

Prposition 4.0d e(s)9u(r)-6.9987(e)3.99577(s-4510.006(t)-2.00195(ha)4(t)-2.00195]TJ/R2412Tf-2

TABLE 8 An Improved Type System

All rules from Table 5 except $\mathsf{t\text{-}eql_t}$.

$$(\mathsf{t\text{-}eql'_t}) \quad \Gamma \vdash' u{:}\mathrm{K}, v{:}\mathrm{L} \quad \Gamma \vdash'_w q \qquad \Gamma \sqcap \{u{:}\mathrm{L}, v{:}$$

The augmented type system is also needed in order to type the “remote channel creation” code reported in Example 4 of Section 3. There we presented an encoding of

$$\mathrm{T}(h) \Leftarrow \ell :: (\nu a, b)\ h :: p \tag{†}$$

as:

$$\begin{aligned} \mathrm{T}(h) \Leftarrow (\nu r)\ \ell :: (\nu a, b)\ h :: r!\langle \ell[a,b] \rangle \\ \mid r?(z[x,y])\,\mathsf{if}\ z = \ell\ \mathsf{then}\ p\{\!|^{z[x,y]}/_{\ell[a,b]}|\!\} \end{aligned} \tag{‡}$$

Using the type system of Section 4.2, the fact that (†) is well-typed does not guarantee that (‡) is well-typed; using $\mathsf{t\text{-}eql}'_{\mathsf{t}}$, however, this property can be established. The “routed forwarding” example (Example 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:

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

Here the server creates a new location ℓ with three local methods a, b and c and gradually exports knowledge of ℓ and its r60Td[(,)10.0016]TJ/R2412-612Tfh99246Tf4.1or

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 [24], 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 [24]. 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 Table 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:

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

)9(uc)11(xam).2)htTLiT*[t6(r.98(a)40377.0(8(s)9(ub)e2602(c)187(a)(2)-]327.07(r)-342.007a]TJ/R19(he)-J/R21121(s)8.998

TABLE 9 Extended Pre-Types

Capabilities:	Subtyping:	
κ ::= **move** | **newc**	$\kappa \leq \kappa$	
| a:A	a:A $\leq$ a:B	if A $\leq$ B
α ::= $\mathsf{r}\langle\zeta\rangle$	$\mathsf{r}\langle\zeta\rangle \leq \mathsf{r}\langle\zeta'\rangle$	if $\zeta \leq \zeta'$
| $\mathsf{w}\langle\xi\rangle$	$\mathsf{w}\langle\xi\rangle \leq \mathsf{w}\langle\xi'\rangle$	if $\xi' \leq \xi$
Pre-Types:		
K ::= $\mathsf{loc}\{\widetilde{\kappa}\}$	K $\leq$ L	if $\forall\lambda \in$ L: $\exists\kappa \in$ K: $\kappa \leq \lambda$
A ::= $\mathsf{chan}\{\widetilde{\alpha}\}$	A $\leq$ B	if $\forall\beta \in$ B: $\exists\alpha \in$ A: $\alpha \leq \beta$
ζ ::= A | $\widetilde{\zeta}$	$\widetilde{\zeta} \leq \widetilde{\xi}$	if $\forall i$: $\zeta_i \leq \xi_i$
| K$[\widetilde{\mathrm{A}}]$	K$[\widetilde{\mathrm{A}}]$ $\leq$ L$[\widetilde{\mathrm{B}}]$	if K $\leq$ L and $\widetilde{\mathrm{A}} \leq \widetilde{\mathrm{B}}$

DEFINITION 6.1 (EXTENDED TYPES).

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

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

$$\begin{array}{l} \mathsf{r}\langle\zeta\rangle \in \mathrm{A} \text{ and } \mathsf{r}\langle\zeta'\rangle \in \mathrm{A} \text{ imply } \zeta = \zeta' \\ \mathsf{w}\langle\xi\rangle \in \mathrm{A} \text{ and } \mathsf{w}\langle\xi'\rangle \in \mathrm{A} \text{ imply } \xi = \xi' \\ \mathsf{r}\langle\zeta\rangle \in \mathrm{A} \text{ and } \mathsf{w}\langle\xi\rangle \in \mathrm{A} \text{ imply } \xi \leq \zeta \end{array}$$

(c) Pre-types of the form $\widetilde{\zeta}$ and K$[\widetilde{\mathrm{A}}]$ 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 ag(L)11]TJ/R21129.987(e)4(nt)-2(s)-191(a49.99.987(nt)-187(e)9(D7112/R940.24Tf002

TABLE 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:

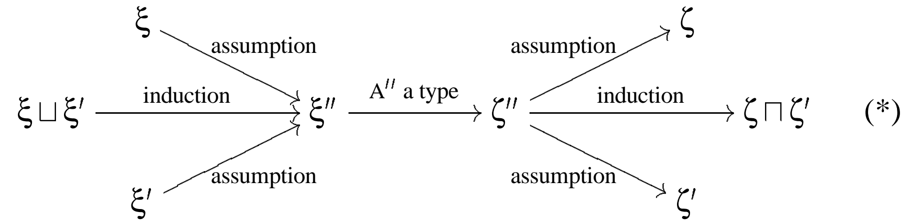
$$\begin{aligned} \mathsf{r}\langle\zeta\rangle \in A \text{ and } \mathsf{r}\langle\zeta'\rangle \in A' \text{ and } \zeta \sqcap \zeta' \text{ undefined} \\ \mathsf{w}\langle\xi\rangle \in A \text{ and } \mathsf{w}\langle\xi'\rangle \in A' \text{ and } \xi \sqcup \xi' \text{ undefined} \\ \mathsf{r}\langle\zeta\rangle \in A \text{ and } \mathsf{w}\langle\xi'\rangle \in A' \text{ and } \xi' \not\leq \zeta \\ \mathsf{w}\langle\xi\rangle \in A \text{ and } \mathsf{r}\langle\zeta'\rangle \in A' \text{ and } \xi \not\leq \zeta' \end{aligned}$$

Otherwise the definition is:

$$A \sqcap A' =$$

On channels, in order for $\{\mathsf{r}\langle\zeta\rangle, \mathsf{w}\langle\xi\rangle\} \sqcap \{\mathsf{r}\langle\zeta'\rangle, \mathsf{w}\langle\xi'\rangle\}$ to be defined, the types must satisfy the following constraints. (In the figure, arrows indicat

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



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

$$\mathsf{r}\langle\zeta\rangle = \mathsf{chan}\{\mathsf{r}\langle\zeta\rangle\} \qquad \mathsf{w}\langle\zeta\rangle = \mathsf{chan}\{\mathsf{w}\langle\zeta\rangle\} \qquad \mathsf{rw}\langle\zeta\rangle = \mathsf{chan}\{\mathsf{r}\langle\zeta\rangle, \mathsf{w}\langle\zeta\rangle\}$$

There are three PS types of the form $io\langle\rangle$, where io is an "i/o tag" ($io ::= \mathsf{r}$

T

TABLE 12 Extended Runtime Errors

Rules e-eql, e-eqc and e-str from Table 7.

Rule	Term	Condition
(e-move″)	$\ell[\![k :: p]\!]_\Gamma \xrightarrow{err''}$	if $\Gamma(k) \not\leq \mathsf{loc}\{\mathbf{move}\}$
(e-subc″)	$\ell[\![(\nu a)p]\!]_\Gamma \xrightarrow{err''}$	if $\Gamma(k) \not\leq \mathsf{loc}\{\mathbf{newc}\}$
(e-snd″)	$\ell[\![a!\langle V\rangle\, q]\!]_\Gamma \xrightarrow{err''}$	if $\Gamma_\ell(V) \not\leq \mathsf{wobj}(\Gamma(\ell,a))$
(e-rcv″)	$\ell[\![a?(X{:}\zeta)p]\!]_\Delta \xrightarrow{err''}$	if $\mathsf{robj}(\Delta(\ell,a)) \not\leq \zeta$
(e-comm″)	$\ell[\![a!\langle V\rangle p]\!]_\Gamma \mid \ell[\![a?(X{:}\zeta)\, q]\!]_\Delta \xrightarrow{err''}$	if $\mathsf{wobj}(\Gamma(\ell,a)) \not\leq \mathsf{robj}(\Delta(\ell,a))$

a valid type (and thus are not allowed by our type system) because the read and write capabilities conflict ($\mathrm{A}_{\mathsf{rw}} \leq \mathrm{A}_{\mathsf{w}}$). If we did allow such types, however, then we could find Γ, Y and q such that:

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

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 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.

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

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

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

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:

$$\begin{aligned}\mathrm{L}_{\mathsf{cell}} &= \mathsf{loc}\{\mathbf{move},\mathbf{newc},g{:}\mathsf{rw}\langle\zeta_{\mathsf{g}}\rangle,p{:}\mathsf{rw}\langle\zeta_{\mathsf{p}}\rangle\}\\ \zeta_{\mathsf{g}} &= \mathsf{loc}\{\mathbf{move}\}[\mathsf{w}\langle\mathsf{int}\rangle]\\ \zeta_{\mathsf{p}} &= (\mathsf{loc}\{\mathbf{move}\}[\mathsf{w}\langle\rangle], \mathsf{int})\end{aligned}$$

Location *cell* fiust be g.00277880195(i)]TJ324.480Td[(on)-240.007(3.)10.0016]TJ-280.32-220.2013

site. In many cases, it may be possible to establish this requirement statically, thus making the dynamic matching redundant; however, our type system is not powerful enough to do so. Suppose that we extend the language with a type for "channels at w": $@_w\mathsf{chan}\{\zeta\}$. Consider the threads:

$$p = a?(z{:}\mathsf{loc})\ b?(x{:}@_z\mathsf{chan})$$
$$q = (\nu\ell[c])\ a!\langle\ell\rangle\ b!\langle c\rangle$$

The thread q creates a location ℓ with channel c, sends ℓ and then sends c. The thread p, instead, waits to receive a location z and then to receive a channel x at z. This code can be statically checked to guarantee that when x is received, x is indeed located at z (*i.e.* ℓ). However, if we have two copies of p running in parallel with q and another thread, r,

$$r = (\nu k[d])\ a!\langle k\rangle\ b!\langle d\rangle$$

then it is no longer guaranteed that each copy of p will receive a matching location and channel. To eliminate such problems, one might adopt the notion of *linear* channels [19] and require that channels such as a and b have at most one sender.

Type Extrusion. One limitation of our language is that names can be extruded,

Bth

is not allowed by our syntax; we require all types to be closed. We might hope for a cleverer solution, such as the following, where the original server sends a

of work in concurrency theory; it is also a "special case" of extreme practical importance.

The language of this paper can be considered "minimal" in the sense that there is only one form of movement: code movement. We are also interested in type systems for languages in which the only form of movement is location movement. However, location movement, in a simple language such as Dπ, is not powerful enough to express interaction between agents. This is because all interaction occurs *within* a location, and therefore interaction *between* locations is not possible without some extension to the language. In variants of the distributed join calculus [12, 26, 25], in addition to location movement, code movement is

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* $\Gamma \vdash P$ *and* $\Delta \leq \Gamma$ *then* $\Delta \vdash P$.
(b) *If* $\Gamma \vdash_{w} p$ *and* $\Delta \leq \Gamma$ *then* $\Delta \vdash_{w} p$.
(c) *If* $\Gamma \vdash_{w} V{:}\zeta$ *and* $\Delta \leq \Gamma$ *then* $\Delta \vdash_{w} V{:}\zeta$.

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) ($\mathsf{t\text{-}run_s}$) Suppose $\Gamma \vdash \ell[\![p]\!]$ because $\Gamma \vdash \ell{:}\mathsf{loc}$ and $\Gamma \vdash_{\ell} p$. Using the auxiliary results we obtain $\Delta \vdash \ell{:}\mathsf{loc}$ and $\Delta \vdash_{\ell} p$. Using $\mathsf{t\text{-}run_s}$, we have $\Delta \vdash \ell[\![p]\!]$.

(b) ($\mathsf{t\text{-}r_t}$) Suppose $\Gamma \vdash_{w} u?(X{:}\zeta$

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

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 Q. Then* $\Gamma \vdash (\nu e)(Q \mid P)$ *if and only if* $\Gamma \vdash Q \mid (\nu e) P$

Proof. We examine the case when e is a channel; the case in which e is a location is similar. Suppose $\Gamma \vdash (\nu a{:}\Lambda)(Q \mid P)$. Then using $\mathtt{t\text{-}newc_s}$ and $\mathtt{t\text{-}str_s}$, we have that $\Gamma, \Lambda_a \vdash Q$ and $\Gamma, \Lambda_a \vdash P$. Applying Lemma A.2 to the first of these we obtain $(\Gamma, \Lambda_a) \setminus a \vdash Q$, *i.e.* $\Gamma \vdash Q$ since a is new to Γ. Applying $\mathtt{t\text{-}newc_s}$ to the second statement we obtain $\Gamma \vdash (\nu a{:}\Lambda)$ and therefore $\mathtt{t\text{-}str_s}$ gives $\Gamma \vdash Q \mid (\nu a{:}\Lambda) P$.

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 P is closed and* $P \longrightarrow P'$ *then* P' *is closed.*

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

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* $\Gamma \vdash k{:}K$ *and* $\Gamma, z{:}K, {}_zX{:}\zeta \vdash_w p$ *then* $\Gamma, {}_kX{:}\zeta \vdash_{w\{\!|k/z|\!\}} p\{\!|k/z|\!\}$.
(b) *If* $\Gamma \vdash k{:}K$ *and* $\Gamma, z{:}K, {}_zX{:}\zeta \vdash_w U{:}\xi$ *then* $\Gamma, {}_kX{:}\zeta \vdash_{w\{\!|k/z|\!\}} U\{\!|k/z|\!\}{:}\xi$.

Proof. For both results the proof is similar. Informally the proof proceeds, in the case of threads, by taking a derivation of the judgment $\Gamma, z{:}\mathrm{K}, {}_zX{:}\zeta \vdash_w p$, substituting k for z throughout and thereby obtaining a derivation of $\Gamma, {}_kX{:}\zeta \vdash_{w\{\!|k/z|\!\}} p\{\!|k/z|\!\}$. Formally it is a straightforward induction on type judgments. We omit the

We can rewrite this as:

$$\Gamma \vdash_{\bar{v}} V_1{:}\zeta_1, \ldots, V_n{:}\zeta_n \quad \text{and} \quad \Gamma, {}_{v}X_1{:}\zeta_1, \ldots, {}_{v}X_n{:}\zeta_n \vdash_{w} U{:}\xi$$

Using induction we have:

$$\Gamma, {}_{v}X_1{:}\zeta_1, \ldots, {}_{v}(X_{n-1}{:}\zeta_{n-1}) \vdash_{w\{V_n/X_n\}} U\{\!|^{V_n}$$

Finally, $\text{t-r}''_{\text{t}}$

- r-move states $\ell[\![u :: p]\!] \longrightarrow k[\![p]\!]$. By supposition $\Gamma \vdash \ell[\![u :: p]\!]$. Then using t-run$_\mathsf{s}$ we have $\Gamma \vdash_{\ell} u :: p$. Then using t-move$_\mathsf{t}$, $\Gamma \vdash_{k} p$ and therefore by t-run$_\mathsf{s}$ $\Gamma \vdash k[\![p]\!]$.

- r-comm states $\ell[\![a!\langle V \rangle p]\!] \ \mid\ \ell[\![a?(X{:}\zeta$

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

References

Conference Record of the ACM Symposium on Principles of Programming Languages, Paris, January 1996. ACM Press.

[20] 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.

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

[22] ObjectSpace Inc. Objectspace voyager. `http://www.objectspace.com/voyager`, 1997.

[23] C. Perkins. IP mobility support. RFC 2002, 1996.

[24] 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.

[25] 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.

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

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

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