M^3: Mobility Types for Mobile Processes in Mobile Ambients [5]

Mario Coppo [a,1] Mariangiola Dezani-Ciancaglini [a,2]
Elio Giovannetti [a,3] Ivano Salvo [b,4]

[a] *Dipartimento di Informatica*
Università di Torino
corso Svizzera 185, 10149 Torino, Italia
e-mail: {coppo,dezani,elio}@di.unito.it

[b] *Dipartimento di Scienze dell'Informazione*
Università di Roma "La Sapienza"
via Salaria 113, 00198 Roma, Italia
e-mail: salvo@dsi.uniroma1.it

Abstract

We present an ambient-like calculus in which the open capability is dropped, and a new form of "lightweight" process mobility is introduced. The calculus comes equipped with a type system that allows the kind of values exchanged in communications and the access and mobility properties of processes to be controlled. A type inference procedure determines the "minimal" requirements to accept a system or a component as well typed. This gives a kind of principal typing. As an expressiveness test, we show that some well known calculi of concurrency and mobility can be encoded in our calculus in a natural way.

1 Introduction

In the last few years a new conceptual dimension of computing has acquired a prominent role and is looking for an adequate theoretical foundation: space

[1] Partially supported by MURST Cofin'01 NAPOLI Project and by the EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222.
[2] Partially supported by MURST Cofin'01 COMETA Project and by the EU within the FET - Global Computing initiative, project DART IST-2001-33477.
[3] Partially supported by MURST Cofin'01 NAPOLI Project and by EU within the FET - Global Computing initiative, project DART IST-2001-33477.
[4] Partially supported by MURST Cofin'01 NAPOLI Project and by EU within the FET - Global Computing initiative, project MyThS IST-2001-32167.
[5] The funding bodies are not responsible for any use that might be made of the results presented here.

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and movement in space.

A huge amount of computational entities distributed worldwide, exchanging data, moving from one location to another, interacting with each other (either cooperatively or competitively), give rise to global computing activity. Computation has therefore to be abstractly described as something that develops not only in time and in memory space, either sequentially (λ-calculus) or as a dynamic set of concurrent processes (π-calculus), but also in a wide geographical and administrative space (ambient calculi).

The calculus of Mobile Ambients (MA)[5], building upon the concurrency paradigm represented by the π-calculus [11,10], introduced the notion of an ambient as "a bounded place where (generally multithreaded) computation happens", which can contain nested subambients in a tree structure, and which can move in and out of other ambients, i.e., up and down the tree (thus rearranging the structure itself). Direct communication can only occur locally within each ambient (through a common anonymous channel); communication and interaction between different ambients has to be mediated by movement and by the dissolution of ambient boundaries.

Ambients are intended to model mobile agents and processes, messages or packets exchanged over the network, mobile devices, physical and virtual locations, administrative and security domains, etc. in a uniform way.

For this reason, in ambient calculi the distinction between processes and (possibly mobile) containers of processes is intentionally blurred. In MA there are implicitly two main forms of entities, which we will respectively call *threads* (or *lightweight processes*) and *ambient-processes* (or *heavy-weight processes*). The former are unnamed lists of actions [6] $\mathsf{act}_1.\mathsf{act}_2 \ldots \mathsf{act}_m$ to be executed sequentially, generally in concurrency with other threads: they can perform communication and drive their containers through the spatial hierarchy, but cannot individually go from one ambient to another. The latter are named containers of concurrent threads $m[P_1 \mid P_2 \ldots \mid P_n]$: they can enter and exit other ambients, driven by their internal processes, but cannot directly perform communication.

Therefore, mobile processes must be represented by ambient-processes; communication between them is represented by the exchange of other ambient-processes of usually shorter life, which have their boundaries dissolved by an open action so as to expose their internal lightweight processes performing the input-output proper. Such capability of opening an ambient, however, has been perceived by many as potentially dangerous, since it could be used inadvertently or maliciously to open and thus destroy the individuality of a mobile agent.

Among the many proposed variations of MA handling this issue, the calculus of Safe Ambients [8,2] introduced the notion of coaction, by which –

[6] As a matter of fact, a sequence of actions may also end with an asynchronous output, or an ambient-process creation $m[P]$, or a forking into different parallel threads.

among other things – an ambient cannot be opened if it is not willing to.

In the calculus of Boxed Ambients [3], on the other hand, open is dropped altogether, and its absence is compensated by the possibility of direct communication between parent and children ambients.

In the present work, we explore a slightly different approach, where we intend to keep the purely local character of communication so that no hidden costs are present in the input-output primitives. At the same time we also want to represent inter-ambient communication by pure input-output between lightweight processes, avoiding the more general opening mechanism.

We do this by recovering the idea, present in Distributed π-calculus (Dπ) [7], that a lightweight process may move directly from one location to another, without the need of being enclosed in an ambient. Mobile threads also seem to more closely represent *strong software mobility*, by which a procedure (or function, or method, or script, depending on the programming model) can – through a go instruction – suspend its execution on one machine and resume it exactly from the same point on another (generally remote) machine, though for the moment we do not explore modelling this kind of application, which would probably need additional constructs.

All ambient calculi come with type systems as essential components, since – like any formal description – they are intended as foundations for reasoning about program behaviours in the new global computing reality. In our proposal too the calculus is an elementary basis for a type discipline able to control communication as well as access and mobility properties. We have tried to abstract (or extract), from the many complex features that could be envisaged, a system that is non-trivial but simple enough to be easily readable and understandable.

The system is incremental in the sense that it can type components in incomplete environments, and is supplied with a type inference algorithm that determines the "minimal" requirements for accepting a component as well typed.

In spite of its simplicity, the (typed) calculus seems to possess sufficient expressive power, admitting natural encodings of two standard calculi of concurrency, π and Dπ.

As usual, our system $\mathbf{M}^3$, of *Mobile processes and Mobile ambients with Mobility types*, consists of two main interconnected components: a calculus and a type system.

The rest of the paper is as follows: in section, 2 we present the calculus, followed by the type system in section 3; some examples of the expressiveness of the calculus and of the type system are given in section 4, by showing how to encode other mobility primitives, well-known calculi for concurrency, and some common protocols; in section 5, we describe a sound and complete type inference algorithm for our type system. Finally, some remarks and possible directions for future work conclude the paper.

2 The Calculus

The structural syntax of the terms of our calculus, shown in Figure 1, is the same as that of MA except for the absence of open and the presence of the new primitive to for lightweight process mobility. Also, synchronous output is allowed, of which the asynchronous version is a particular case. As in MA,

$M,\ N,\ L$ ::=		**messages**
	$m, n, \ldots x, y, \ldots$	ambient names, variables
	$\mathsf{in}\ M$	moves the containing ambient into ambient M
	$\mathsf{out}\ M$	moves the containing ambient out of ambient M
	$\mathsf{to}\ M$	goes out from its ambient into sibling ambient M
	$M.M'$	path
$P,\ Q,\ R$ ::=		**processes**
	$\mathsf{0}$	null
	$M\,.\,P$	prefixed
	$\langle M\rangle\,.\,P$	synchronous output
	$(x{:}\,W)\,.\,P$	typed input
	$P \mid Q$	parallel composition
	$M[P]$	ambient
	$!\,P$	replication
	$(\nu n{:}\,\mathsf{amb}(g))P$	name restriction
	$(\nu\{\overrightarrow{\mathbf{g}{:}\mathbf{G}}\}_{(k)})P$	group restriction

where: W is a message type, g is a group name, $\nu\{\overrightarrow{\mathbf{g}{:}\mathbf{G}}\}_{(k)}$ is a concise notation for $\nu\{g_1{:}\,G_1, \ldots, g_k{:}\,G_k\}$, with $g_1, \ldots, g_k$ group names and $G_1, \ldots, G_k$ group types (see Fig. 4).

Fig. 1: Syntax

we introduce types – for the sake of simplicity – already in the term syntax, namely in the input construct and in the restrictions w.r.t. ambient names and group names. The terms defined in Figure 1 are not exactly the terms of our calculus, since the type constraints are not yet taken into account. This will be done by the typing rules of Figure 5. The notions of reduction and structural equivalence do not rely upon the fact that terms are well-typed but obviously they are only meaningful for well-typed terms.

Since group types contain group names, it is crucial to restrict sets of group names. We denote by $\nu\{g_1{:}\,G_1, \ldots, g_k{:}\,G_k\}$ the simultaneous restriction of the group names $g_1, \ldots, g_k$ having respectively group types $G_1, \ldots, G_k$. Simultaneous restriction is needed since groups can have mutually dependent group types. We adopt the standard convention that action prefixing takes precedence over parallel composition: if act denotes a generic prefix, $\mathsf{act}.\alpha \mid \beta$ is read as $(\mathsf{act}.\alpha)|\beta$.

The operational semantics is given by a reduction relation along with a structural congruence, as usual.

Structural congruence (shown in Figure 2) is almost standard for the usual

ambient constructors [4], but for the rule $(\nu n\colon g)n[\mathsf{0}] \equiv \mathsf{0}$ which is added to get a form of garbage collection in absence of the open primitive and for the rules to handle simultaneous group restriction. These new rules allows to permute, split and erase group restrictions under suitable conditions. Despite their awkward look they are essentially similar to the rules for names restriction. What complicates notations is the fact that mutually dependent group types must be handled contemporarily. We omit the standard rules of α-conversion of bound names.

equivalence:

$$P \equiv P \qquad P \equiv Q \implies Q \equiv P \qquad P \equiv Q,\ Q \equiv R \implies P \equiv R$$

congruence:

$$\begin{array}{ll} P \equiv Q \implies M\,.\,P \equiv M\,.\,Q & P \equiv Q \implies M[P] \equiv M[Q] \\ P \equiv Q \implies \langle M\rangle\,.\,P \equiv \langle M\rangle\,.\,Q & P \equiv Q \implies !\,P \equiv !\,Q \\ P \equiv Q \implies (x\colon W)\,.\,P \equiv (x\colon W)\,.\,Q & P \equiv Q \implies (\nu n\colon \mathsf{amb}(g))P \equiv (\nu n\colon \mathsf{amb}(g))Q \\ P \equiv Q \implies P \mid R \equiv Q \mid R & P \equiv Q \implies (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})P \equiv (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})Q \end{array}$$

prefix associativity:

$$(M.M').P \equiv M.M'.P$$

parallel composition – associativity, commutativity, zero:

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathsf{0} \equiv P$$

replication:

$$!P \equiv P \mid !P \qquad !\,\mathsf{0} \equiv \mathsf{0}$$

restriction swapping and group restriction splitting :

$$n \neq m \implies (\nu n\colon \mathsf{amb}(g))(\nu m\colon \mathsf{amb}(g'))P \equiv (\nu m\colon \mathsf{amb}(g'))(\nu n\colon \mathsf{amb}(g))P$$

$$g_i \neq g'_j \,\&\, g_i \notin GN(G'_j) \,\&\, g'_j \notin GN(G_i)(1 \leq i \leq k)(1 \leq j \leq h)$$
$$\implies (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})(\nu\{\overrightarrow{\mathbf{g}'\colon\mathbf{G}'}\}_{(h)})P \equiv (\nu\{\overrightarrow{\mathbf{g}'\colon\mathbf{G}'}\}_{(h)})(\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})P$$

$$g \neq g_i (1 \leq i \leq k) \implies (\nu n\colon \mathsf{amb}(g))(\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})P \equiv (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})(\nu n\colon \mathsf{amb}(g))P$$

$$g_i \notin GN(G_{k+j}) \,\&\, g_{k+j} \notin GN(G_i)(1 \leq i \leq k)(1 \leq j \leq h)$$
$$\implies (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k+h)})P \equiv (\nu\{g_1\colon G_1, \ldots, g_k\colon G_k\})(\nu\{g_{k+1}\colon G_{k+1}, \ldots, g_{k+h}\colon G_{k+h}\})P$$

scope extrusion:

$$\begin{array}{l} n \notin AN(Q) \implies (\nu n\colon g)P \mid Q \equiv (\nu n\colon g)(P \mid Q) \\ n \neq m \implies (\nu n\colon \mathsf{amb}(g))m[P] \equiv m[(\nu n\colon \mathsf{amb}(g))P] \\ g_i \notin GN(Q)(1 \leq i \leq k) \implies (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})P \mid Q \equiv (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})(P \mid Q) \\ (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})m[P] \equiv m[(\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})P] \end{array}$$

equivalence to zero:

$$\begin{array}{l} (\nu n\colon \mathsf{amb}(g))\mathsf{0} \equiv \mathsf{0} \\ (\nu\{\overrightarrow{\mathbf{g}\colon\mathbf{G}}\}_{(k)})\mathsf{0} \equiv \mathsf{0} \\ (\nu n\colon g)n[\mathsf{0}] \equiv \mathsf{0} \end{array}$$

where $AN(Q)$ is the set of free ambient names in Q, $GN(Q)$ is the set of free group names in Q, and $GN(G)$ is the set of free group names in G.

Fig. 2: Structural congruence

The reduction rules, shown in Figure 3, are the same as those for MA, with the obvious difference consisting in the synchronous output and the missing

open, and with the new rule for the to action, similar to the go primitive of Dπ or to the "migrate" instructions for strong code mobility in software agents.

A lightweight process executing a $\mathsf{to}\, m$ action moves between sibling ambients: more precisely it goes from an ambient n, where it is initially located, to a (different) ambient of name m that is a sibling of n, thus crossing two boundaries in one step; the boundaries are however at the same level, so that – differently from moving upward or downward – the process does not change its nesting level.

Observe that the form of the rule, while entailing nondeterminism among different destinations of the same name, guarantees that the destination, though possibly having the same name as the source, must be a different ambient: so that a term of the form $m[\mathsf{to}\, m.P]$ cannot reduce to $m[P]$, with a jump from one to the very same location!

Basic reduction rules:

$$\begin{array}{lrcl}
\text{(R-in)} & n[\,\mathsf{in}\ m\,.\,P\,|\,Q\,]\,|\,m[R] & \rightarrow & m[\,n[\,P\,|\,Q\,]\,|\,R\,] \\
\text{(R-out)} & m[\,n[\,\mathsf{out}\ m\,.\,P\,|\,Q\,]\,|\,R\,] & \rightarrow & n[\,P\,|\,Q\,]\,|\,m[R] \\
\text{(R-to)} & n[\mathsf{to}\, m\,.\,P\,|\,Q]\,|\,m[R] & \rightarrow & n[Q]\,|\,m[P\,|\,R] \\
\text{(R-comm)} & (x\colon W)\,.\,P\,|\,\langle M\rangle\,.\,Q & \rightarrow & P\{x := M\}\,|\,Q
\end{array}$$

Structural reduction rules:

$$\begin{array}{lrcl}
\text{(R-in)} & P \rightarrow Q & \Rightarrow & P\,|\,R \rightarrow Q\,|\,R \\
\text{(R-amb)} & P \rightarrow Q & \Rightarrow & n[P] \rightarrow n[Q] \\
\text{(R-}\equiv\text{)} & P' \equiv P',\ \ P \rightarrow Q,\ \ Q \equiv Q' & \Rightarrow & P' \rightarrow Q' \\
\text{(R-}\nu\text{)} & P \rightarrow Q & \Rightarrow & (\nu n\colon g)P \rightarrow (\nu n\colon g)Q \\
\text{(R-}\nu\text{-group)} & P \rightarrow Q & \Rightarrow & (\nu\{\overrightarrow{\mathbf{g}:\mathbf{G}}\}_{(k)})P \rightarrow (\nu\{\overrightarrow{\mathbf{g}:\mathbf{G}}\}_{(k)})Q
\end{array}$$

Fig. 3: Reduction

3 The Type System

The syntax definition of Figure 1 is not really complete without giving the rules for defining well-typed terms. Types firstly ensure that meaningless terms cannot be defined or be produced by computation.

In addition, we are also interested in using types for the control of access and mobility. As already observed in [4], expressing such properties in terms of single ambients leads to *dependent types*, for example in judgments of the form $n\colon \mathit{CanEnter}(\{m_1, \ldots, m_k\})$. Following the seminal work of [4], and [9] among others, we therefore adopt an approach based on ambient *groups*, which permits us to avoid direct dependence of types on values.

As usual for ambients, there are three fundamental categories of types: ambient types, capability types, and process types, corresponding to the three

		$g, h, \ldots$	groups
		$\mathscr{S}, \mathscr{C}, \mathscr{E}, \ldots$	sets of groups; $\mathbb{G}$ is the universal set of groups
Amb	::=	$\mathsf{amb}(g)$	ambient type: ambients of group g
Pro	::=	$\mathsf{proc}(g)$	process type: processes that can stay in ambients of group g
Cap	::=	$Pro_1 \rightarrow Pro_2$	capability type: capabilities that, prefixed to a process of type Pro_1, turn it into a process of type Pro_2
W	::=		message type
		Amb	ambient type
		Cap	capability type
T	::=		communication type
		shh	no communication
		W	communication of messages of type W
G	::=	$\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)$	group type

Fig. 4: Types

main syntactic categories of terms. Since only ambient names and capabilities, but not processes, can be transmitted, message types – i.e., the ones explicitly attached to input variables – can only be ambient or capability types.

Syntactically, groups are merely names $g, h, \ldots$ occurring as basic components of other types. Formally, they may be considered atomic types, which represent sets of ambients sharing some common features.

There is a subtle difference w.r.t. [4] and [9]. In those systems, ambient types are of the schematic form $\mathsf{amb}(g, \mathcal{B})$, where $\mathcal{B}$ is the expression of some behavioural properties concerning mobility and communication. In our proposal the property $\mathcal{B}$ is instead (the content of) the type of the group. The typing judgment $m : \mathsf{amb}(g, \mathcal{B})$ becomes, in our system, $m : \mathsf{amb}(g)$, $g : \mathcal{B}$.

The first form is more general, allowing different ambient types for the same group. In our approach, on the other hand, a group represents a set of ambients guaranteed to share common mobility and access properties (and communication behaviour), as specified by the group's type.

The only component of an ambient type $\mathsf{amb}(g)$ or a process type $\mathsf{proc}(g)$ is a group name g, whose type[7] G describes – in terms of other group names (possibly including the very group g typed by G) – the properties of all the ambients and processes of that group. In a sense, groups and group types work as indirections between types and values, so as to avoid that types directly "point to" (i.e., depend on) values.

As is standard, the connection between ambients and processes is given by the fact that processes of type $\mathsf{proc}(g)$ can run safely only within ambients of

[7] Observe that a group type is the type of a type. Thus, following a rather standard terminology, it is a *kind*; moreover, since it contains group names, it might be considered a "kind dependent on types". However, this double level is used, as is clear, only in a very limited and ad hoc way, with no real stratification; it is therefore justified not to use the expression *group kind*, but simply stick to *group type*.

type $\mathsf{amb}(g)$.

The form of capability types is a relative novelty: they are (very particular) sorts of function types from processes to processes, corresponding to the fact that, from a syntactic point of view, a prefix turns a process into another process; $\mathsf{proc}(g_1) \to \mathsf{proc}(g_2)$ is the type of a capability that, prefixed to a process of type $\mathsf{proc}(g_1)$, transforms it into a process of type $\mathsf{proc}(g_2)$; or, viewed at runtime, a capability that, when exercised by a process of type $\mathsf{proc}(g_2)$, of course located in an ambient of type $\mathsf{amb}(g_2)$, leaves a continuation of type $\mathsf{proc}(g_1)$, located in an ambient of type $\mathsf{amb}(g_1)$.

This form bears some non-superficial resemblance to that of [1], where a capability type is a type context which, when filled with a process type, yields another process type.

Notational Remark: we shall simply write g both for $\mathsf{amb}(g)$ and for $\mathsf{proc}(g)$, the distinction always being clear from the context. As a consequence, capability types will be written in the concise form $g_1 \to g_2$.

Besides, we may use the abbreviations *g-ambients* and *g-processes* respectively for *the ambients of group g* and *the processes of group g*.

The communication type is completely standard: it is either the atomic type shh, denoting absence of communication, or a message type, which in turn may be an ambient type or a capability type. Note that the type shh, typical of ambient systems, is not the type of empty messages (which can be used for synchronization), but the one denoting the very absence of input-output.

Finally, *group types* (ranged over by G) consist of four components and are of the form $\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)$, where $\mathscr{S}, \mathscr{C}$ and $\mathscr{E}$ are sets of group names, and T is a communication type. If g is a group of type $\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)$, then the intuitive meanings of the type's components are the following:

- $\mathscr{S}$ is the set of ambient groups where the ambients of group g can stay (and is never empty);
- $\mathscr{C}$ is the set of ambient groups that g-ambients can cross, i.e., those that they may be driven into or out of, respectively, by in or out actions; clearly, it must be $\mathscr{C} \subseteq \mathscr{S}$ (and is empty if the ambients of group g are immobile);
- $\mathscr{E}$ is the set of ambients that (lightweight) g-processes can "enter": more precisely, those to which a g-process may send its continuation by means of a to action (it is empty if lightweight g-processes are immobile);
- T is the (fixed) communication type (or topics of conversation) within g-ambients.

The information of $\mathscr{S}$ and $\mathscr{C}$ component of a group type was considered also in the work of Merro and Sassone [9].

If $G = \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)$ is a group type, we write $\mathscr{S}(G)$, $\mathscr{C}(G)$, $\mathscr{E}(G)$, $T(G)$ respectively to denote the components $\mathscr{S}$, $\mathscr{C}$, $\mathscr{E}$, T of G.

A *type environment* Γ is defined by the following syntax:

$$\Gamma ::= \varnothing \ \mid \ \Gamma, g : G \ \mid \ \Gamma, n : g \ \mid \ \Gamma, x : T$$

The *domain* of an environment is defined by:

$$\begin{aligned} Dom(\varnothing) &= \varnothing \\ Dom(\Gamma, g : G) &= Dom(\Gamma) \cup \{g\} \\ Dom(\Gamma, n : g) &= Dom(\Gamma) \cup \{n\} \\ Dom(\Gamma, x : T) &= Dom(\Gamma) \cup \{x\} \end{aligned}$$

$GN(G)$ denotes the set of all group names occurring in a group type G, and $GN(\Gamma)$ denotes the set of all group names occurring in Γ, not only in $Dom(\Gamma)$ but also in the components of the types in Γ.

Let ξ range over group names, ambient names and variables. Type environments are seen as sets of statements and considered modulo permutations. We use the standard notation $\Gamma, \xi : A$ to denote an environment containing a statement $\xi : A$, assuming that $\xi \notin Dom(\Gamma)$.

An environment Γ is *well-formed* if for each $\xi \in Dom(\Gamma)$ there is exactly one type associated to it in Γ, i.e., there cannot exist $\xi : A$, $\xi : B \in \Gamma$ with B distinct from A. We assume that all type environments are well-formed.

Two environments Γ_1 and Γ_2 are *compatible*, written $\Gamma_1 \sharp \Gamma_2$, if $\Gamma_1 \cup \Gamma_2$ is a well-formed environment, i.e., if $\forall \xi \in Dom(\Gamma_1) \cap Dom(\Gamma_2), \Gamma_1(\xi) = \Gamma_2(\xi)$.

The formal definition of the type assignment rules is shown in Figure 5.

The system's fundamental rule (Amb) is quite standard: it requires that in a term $m[P]$ the ambient m and its content P be of the same group, while the process $m[P]$, being a completely passive object, unable both to communicate and to move other ambients, may in turn stay in any ambient of any group g' (i.e., it may be of any group g'), *provided* its "membrane" m, of type g, has permission from the specification G to stay in a g'-ambient.

Since a process executing an action $\mathsf{to}\, m$ goes from its ambient (in)to an ambient m, the rule (To) states that the action $\mathsf{to}\, m$, if performed by a process of group g_2 (in a g_2-ambient), leaves as continuation a process of group g_1, if g_1 is the group of m and moreover is one of the groups to which g_2-*processes* are allowed to go (i.e., to send their continuations) by a to.

The rules (In) and (Out) state that a process exercising an $\mathsf{in}/\mathsf{out}\, m$ capability does not change its group g_2 since it does not change its enclosing g_2-ambient, which must however have permission to cross the g_1-ambient m; in the case of (Out), moreover, the g_2-ambient – being driven out of m – becomes a sibling of m, and must therefore have permission to stay where m stays (i.e., the condition $\mathscr{S}(G_1) \subseteq \mathscr{S}(G_2)$). The analogous side condition in the rule (In), ensuring that the moving g_2-ambient has the permission to

$$\frac{\xi : T \in \Gamma}{\Gamma \vdash \xi : T}\ (\textsc{Env}) \qquad \frac{}{\Gamma \vdash \mathsf{0} : g}\ (\textsc{Null})$$

$$\frac{\Gamma \vdash g_2 : G_2 \qquad \Gamma \vdash M : g_1 \qquad g_1 \in \mathscr{C}(G_2)}{\Gamma \vdash \mathsf{in}\ M : g_2 \to g_2}\ (\textsc{In})$$

$$\frac{\Gamma \vdash g_1 : G_1 \quad \Gamma \vdash g_2 : G_2 \quad \Gamma \vdash M : g_1 \quad g_1 \in \mathscr{C}(G_2) \quad \mathscr{S}(G_1) \subseteq \mathscr{S}(G_2)}{\Gamma \vdash \mathsf{out}\ M : g_2 \to g_2}\ (\textsc{Out})$$

$$\frac{\Gamma \vdash g_2 : G_2 \qquad \Gamma \vdash M : g_1 \qquad g_1 \in \mathscr{E}(G_2)}{\Gamma \vdash \mathsf{to}\ M : g_1 \to g_2}\ (\textsc{To})$$

$$\frac{\Gamma \vdash M : g_3 \to g_2 \qquad \Gamma \vdash N : g_1 \to g_3}{\Gamma \vdash M.N : g_1 \to g_2}\ (\textsc{Path})$$

$$\frac{\Gamma \vdash M : g_1 \to g_2 \quad \Gamma \vdash P : g_1}{\Gamma \vdash M.P : g_2}\ (\textsc{Prefix})$$

$$\frac{\Gamma, x : W \vdash P : g}{\Gamma \vdash (x : W)\,.\,P : g}\ (\textsc{Input}) \qquad \frac{\Gamma \vdash P : g \ \ \Gamma \vdash M : W \quad \Gamma \vdash g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, W)}{\Gamma \vdash \langle M \rangle\,.\,P : g}\ (\textsc{Output})$$

$$\frac{\Gamma \vdash P : g \qquad \Gamma \vdash M : g \qquad \Gamma \vdash g : G \qquad g' \in \mathscr{S}(G)}{\Gamma \vdash M[P] : g'}\ (\textsc{Amb})$$

$$\frac{\Gamma_1 \vdash P : g \qquad \Gamma_2 \vdash Q : g \qquad \Gamma_1 \sharp \Gamma_2}{\Gamma_1 \cup \Gamma_2 \vdash P \,|\, Q : g}\ (\textsc{Par}) \qquad \frac{\Gamma \vdash P : g}{\Gamma \vdash !\, P : g}\ (\textsc{Repl})$$

$$\frac{\Gamma, m : g' \vdash P : g}{\Gamma \vdash (\nu m : g')P : g}\ (\textsc{Amb Res})$$

$$\frac{\Gamma, g_1 : G_1, \ldots, g_k : G_k \vdash P : g \quad g_i \notin GN(\Gamma) \ \ g_i \neq g \ \ (1 \leq i \leq k)}{\Gamma \vdash (\nu\{g_1 : G_1, \ldots, g_k : G_k\})P : g}\ (\textsc{Grp Res})$$

Fig. 5: Typing rules

stay inside the g_1-ambient m ($g_1 \in \mathscr{S}(G_2)$) is subsumed by the condition $\mathscr{C}(G) \subseteq \mathscr{S}(G)$ on group types.

The rules (Path) and (Prefix) are as expected from the informal definitions of process and capability types: kinds, respectively, of function composition and function application. The other rules are standard: in the group restriction the set of group names $g_1, \ldots, g_k$ that are abstracted from the environment (i.e., moved from the l.h.s. to the r.h.s. of the turnstile) cannot contain the group g of the restricted term.

The type assignment system is clearly syntax-directed and therefore a Gen-

eration Lemma trivially holds.

Lemma 3.1 (Generation Lemma) (i) *If* $\Gamma \vdash \mathsf{in}\ M : g_2 \to g_2$ *then* $\Gamma \vdash g_2 : G_2$, $\Gamma \vdash M : g_1$, *and* $g_1 \in \mathscr{C}(G_2)$ *for some* g_1.

(ii) *If* $\Gamma \vdash \mathsf{out}\ M : g_2 \to g_2$ *then* $\Gamma \vdash g_1 : G_1$, $\Gamma \vdash g_2 : G_2$, $\Gamma \vdash M : g_1$, $g_1 \in \mathscr{C}(G_2)$, *and* $\mathscr{S}(G_1) \subseteq \mathscr{S}(G_2)$ *for some* g_1.

(iii) *If* $\Gamma \vdash \mathsf{to}\ M : g_1 \to g_2$ *then* $\Gamma \vdash g_2 : G_2$, $\Gamma \vdash M : g_1$, *and* $g_1 \in \mathscr{E}(G_2)$.

(iv) *If* $\Gamma \vdash M.N : g_1 \to g_2$ *then* $\Gamma \vdash M : g_3 \to g_2$, *and* $\Gamma \vdash N : g_1 \to g_3$ *for some* g_3.

(v) *If* $\Gamma \vdash M.P : g_2$ *then* $\Gamma \vdash M : g_1 \to g_2$, *and* $\Gamma \vdash P : g_1$ *for some* g_1.

(vi) *If* $\Gamma \vdash (x : W)P : g$ *then* $\Gamma, x : W \vdash P : g$.

(vii) *If* $\Gamma \vdash \langle M \rangle P : g$ *then* $\Gamma \vdash P : g$, $\Gamma \vdash M : W$, *and* $\Gamma \vdash g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, W)$ *for some* $\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, W)$.

(viii) *If* $\Gamma \vdash M[P] : g$ *then* $\Gamma \vdash P : g'$, $\Gamma \vdash M : g'$, $\Gamma \vdash g' : G$, *and* $g \in \mathscr{S}(G)$ *for some* g', G.

(ix) *If* $\Gamma \vdash P \mid Q : g$ *then* $\Gamma \vdash P : g$, *and* $\Gamma \vdash Q : g$.

(x) *If* $\Gamma \vdash !\,P : g$ *then* $\Gamma \vdash P : g$.

(xi) *If* $\Gamma \vdash (\nu m : g')P : g$ *then* $\Gamma, m : g' \vdash P : g$.

(xii) *If* $\Gamma \vdash (\nu\{g_1 \colon G_1, \ldots, g_k \colon G_k\})P : g$ *then* $\Gamma, g_1 : G_1, \ldots, g_k : G_k \vdash P : g$, $g_i \notin GN(\Gamma)$ *and* $g_i \neq g$ $(1 \leq i \leq k)$.

The usual property of subject reduction holds, which guarantees the soundness of the system by ensuring that typing is preserved by computation. Notice that we do not need to expand environments as [4], since we allow environments to contain group names in types also when there is no group type associated to them (provided this is compatible with the assignment rules), i.e., we allow $\xi : g \in \Gamma$ even if $g \notin Dom(\Gamma)$.

Theorem 3.2 (Subject reduction) *Let* $\Gamma \vdash P : g$. *Then*

(i) $P \equiv Q$ *implies* $\Gamma \vdash Q : g$.

(ii) $P \to Q$ *implies* $\Gamma \vdash Q : g$

Proof. The proof is standard, by induction on the derivations of $P \equiv Q$ and $P \to Q$ using the Generation Lemma. We only consider rule (R-to):

$$n[\mathsf{to}\, m \,.\, P \mid Q] \mid m[R] \;\to\; n[Q] \mid m[P \mid R].$$

If $\Gamma \vdash n[\mathsf{to}\, m \,.\, P \mid Q] \mid m[R] : g$ then by Lemma 3.1(ix) $\Gamma \vdash n[\mathsf{to}\, m \,.\, P \mid Q] : g$ and $\Gamma \vdash m[R] : g$. By 3.1(viii) we must have $\Gamma \vdash \mathsf{to}\, m \,.\, P \mid Q : g_n$, $\Gamma \vdash n : g_n$, $\Gamma \vdash g_n : G_n$, $g \in \mathscr{S}(G_n)$, $\Gamma \vdash R : g_m$, $\Gamma \vdash m : g_m$, $\Gamma \vdash g_m : G_m$ and $g \in \mathscr{S}(G_m)$, for some g_n, G_n, g_m, G_m. From $\Gamma \vdash \mathsf{to}\, m \,.\, P \mid Q : g_n$ by Lemma 3.1(ix) we have $\Gamma \vdash Q : g_n$ and $\Gamma \vdash \mathsf{to}\, m \,.\, P : g_n$, which imply by Lemma 3.1(v) and (iii) $\Gamma \vdash P : g_m$.
Rule (Amb) applied to $\Gamma \vdash Q : g_n$, $\Gamma \vdash n : g_n$, $\Gamma \vdash g_n : G_n$ gives $\Gamma \vdash n[Q] : g$

being $g \in \mathscr{S}(G_n)$. Rule (Par) applied to $\Gamma \vdash P : g_m$, $\Gamma \vdash R : g_m$ gives $\Gamma \vdash P \,|\, R : g_m$. Since $g \in \mathscr{S}(G_m)$ we can deduce $\Gamma \vdash m[P \,|\, R] : g$ using rule (Amb). We conclude $\Gamma \vdash n[Q] \,|\, m[P \,|\, R] : g$ from $\Gamma \vdash n[Q] : g$ and $\Gamma \vdash m[P \,|\, R] : g$ by rule (Amb). □

4 Examples

In this section we test the expressiveness of our calculus, by showing first how to model some common protocols considered in the literature (such as a mail server, a firewall, and a defence against Trojan-horse attacks), and then how to encode other process mobility primitives (up and down) and other well-known calculi for modelling concurrent (π-calculus) and distributed (Dπ) systems.

4.1 Protocols

Mail Server

The basic idea, taken from [6], consists in modelling a mail server as an (immobile) ambient containing mailboxes as immediate subambients. Messages contain address headers that drive them to their destination mailboxes. Each user can read the messages addressed to it by entering its own mailbox. We define:

$$\text{MS} = ms[mbox_1[-] \,|\, \ldots \,|\, mbox_i[-] \,|\, \ldots \,|\, mbox_n[-]]$$
$$\text{MSG} = (\nu\ msg : g_{msg})\ msg[\mathsf{in}\ ms\,.\,\mathsf{in}\ mbox_i\,.\,\mathsf{to}\ u_i\,.\,\langle M\rangle]$$
$$\text{U}_i = u_i[\mathsf{in}\ ms\,.\,\mathsf{in}\ mbox_i\,.\,(x : t_M)\,.\,\mathsf{out}\ mbox_i\,.\,\mathsf{out}\ ms\,.\,P]$$

The process MS is the mail server, containing a set of mailbox-ambients $mbox_i$. A message MSG directed to the user U_i enters the mailbox $mbox_i$ and sends to the user-ambient u_i the message M (of type t_M) which is given as input to the consumer process P.

The user-and-mailbox system is given by a top-level term $\text{MS} \,|\, \text{MSG} \,|\, \text{U}_i$, which can be typed with the group g_0 from the following type assumptions on groups (we adopt the convention that an ambient ms belongs to the group g_{ms}, and so on):

$$g_{ms} : \mathsf{gr}(\{g_0\}, \varnothing, \varnothing, \mathsf{shh}),$$
$$g_{mbox_i} : \mathsf{gr}(\{g_{ms}\}, \varnothing, \varnothing, \mathsf{shh})$$
$$g_{u_i} : \mathsf{gr}(\{g_0, g_{ms}, g_{mbox_i}\}, \{g_{ms}, g_{mbox_i}\}, \varnothing, t_M)$$
$$g_{msg} : \mathsf{gr}(\{g_0, g_{ms}, \ldots, g_{mbox_i}, \ldots\}, \{g_{ms}, \ldots, g_{mbox_i}, \ldots\}, \{\ldots, g_{u_i}, \ldots\}, \mathsf{shh})$$

Observe that the type system prevents a user U_i from entering a mailbox $mbox_j$ with $j \neq i$.

Firewall

A system protected by a firewall can be viewed as an ambient *fw* that supplies the incoming *agent* with a "password" (represented by its very name) which allows the process P to enter it. We define:

$\text{AG} = agent[(x : g_{ag} \to g_{ag}).x.P].$

$\text{FW} = (\nu\, fw : g_{fw}).fw[\mathsf{to}\ agent.\langle \mathsf{in}\ fw\rangle \mid Q].$

The system agent-plus-firewall is represented by the top-level process $\text{AG} \mid \text{FW}$. It can be typed with the group g_0 by assuming $agent : g_{ag}$ and $fw : g_{fw}$, where the groups are typed as follows:

$g_{ag} : \mathsf{gr}(\{g_0\}, \{g_{fw}\}, \varnothing, g_{ag} \to g_{ag})$

$g_{fw} : \mathsf{gr}(\{g_0\}, \varnothing, \{g_{ag}\}, \mathsf{shh})$

The Trojan Horse Attack

In this example, we show how our type system can detect a Trojan horse attack. Ulysses is naturally encoded as a mobile ambient-process that enters an ambient *horse* containing an in *troy* action, and then goes out of it into Troy to destroy Priam's palace. The initial situation is represented by the term:

$$ulysses[\mathsf{in}\ horse.\mathsf{out}\ horse.\mathsf{to}\ palace.\text{DESTROY}] \mid horse[\mathsf{in}\ troy] \mid troy[palace[\text{P}]]$$

If ambients $ulysses, horse, \ldots$ belong respectively to groups $g_{ulysses}, g_{horse}, \ldots$, the whole *myth*ic process can be typed by a group g_{myth} w.r.t. an environment which contains the following assumptions:

$g_{ulysses} : \mathsf{gr}(\{g_{myth}, g_{troy}, g_{horse}\}, \{g_{horse}\}, \{g_{palace}\}, \mathsf{shh})$

$g_{horse} : \mathsf{gr}(\{g_{myth}, g_{troy}\}, \{g_{troy}\}, \varnothing, \mathsf{shh})$

$g_{troy} : \mathsf{gr}(\{g_{myth}\}, \varnothing, \varnothing, \mathsf{shh})$

$g_{palace} : \mathsf{gr}(\{g_{troy}\}, \varnothing, \varnothing, \mathsf{shh})$

from which it is clear that ambients of group $g_{ulysses}$ must have permission to stay within ambients of group g_{troy} and to send processes to ambients of group g_{palace} in order to make the myth well-typed.

4.2 Encoding Process Calculi

Encoding other process mobility actions

The main choice in designing a calculus with mobile (lightweight) processes is the one of the mobility primitives for them. We have chosen to introduce, for the moment, only one primitive, to, [8] since already present, though in a context of immobile locations, in a well established concurrent calculus such as

[8] the pun was initially unintended.

Dπ [7]. Also, this primitive might be argued to naturally model the elementary instruction by which an agent moves from one location to another at the same level.

A natural alternative, or a natural extension, would be a thread mobility analogous to that for ambients, i.e., capabilities to go one step up or down the tree hierarchy, by exiting or entering an ambient.

Consider for example the two primitives up and down, with the following reduction rules:

$$\text{(R-down)} \qquad \mathsf{down}\ m\,.\,P \mid m[R] \ \rightarrow\ m[\,P \mid R\,]$$

$$\text{(R-up)} \qquad m[\,p[\,\mathsf{up}\ m\,.\,P \mid Q\,] \mid R\,] \ \rightarrow\ m[\,P \mid p[Q\,] \mid R]$$

where also the up takes as argument the destination ambient, instead of the source (as the analogous out does). It is interesting that both can be encoded in the to-language, though only as actions in process prefixes and not as capabilities transmissible in a message.

Such encodings are carried out by means of auxiliary ambients, with an interplay of ambient and process mobility. The action down can be defined as:

$$[\![\mathsf{down}\ m\,.\,P]\!] = (\nu g_n : G_n)(\nu n : g_n)n[\mathsf{to}\ m\,.\,P]$$

where $G_n = \mathsf{gr}(\{g\}, \varnothing, \{g_m\}, \mathsf{shh})$, g is the type of the whole process and g_m is the group of m. This encoding is more permissive, from the typing point of view, than the natural typing rule associated to down, which is:

$$\frac{\Gamma \vdash g : G \quad \Gamma \vdash m : g_m \quad \Gamma \vdash P : g_m \quad g_m \in \mathscr{E}(G)}{\Gamma \vdash \mathsf{down}\ m.P : g} \qquad (\mathsf{down}\ m)$$

In fact, the term $n[\mathsf{to}\,m\,.\,P]$ may be given a type g by a derived rule with the same premisses as the rule $\mathsf{down}\ m.P$, but without the condition that $g_m \in \mathscr{E}(G)$. Of course, such assumption on the group type of g may always be made.

In the case of up, we can only define the encoding of an action up^p where p is the name of the ambient wherefrom the process comes, needed to simulate the action with the basic to primitive:

$$[\![\mathsf{up}\ ^p m\,.\,P]\!] = (\nu g_n : G_n)(\nu n : g_n)n[\mathsf{out}\ p\,.\,\mathsf{out}\ m\,.\,\mathsf{to}\ m\,.\,P]$$

where $g_p \colon G_p$ and $g_m \colon G_m$ are respectively the groups of p and m, obviously $g_m \in \mathscr{S}(G_p)$, and $G_n = \mathsf{gr}(\{g_p\} \cup \mathscr{S}(G_p) \cup \mathscr{S}(G_m), \{g_p, g_m\}, \{g_m\}, \mathsf{shh})$. This definition correctly simulates the reduction rule of the up action, as the fol-

lowing reduction sequence shows:

$$\begin{aligned} & m[p[(\nu g_n : G_n)(\nu n : g_n)n[\mathsf{out}\ p\,.\,\mathsf{out}\ m\,.\,\mathsf{to}\ m\,.\,P] \mid Q] \mid R] \\ \to\ & (\nu g_n : G_n)(\nu n : g_n)(n[\mathsf{to}\ m\,.\,P] \mid m[p[Q] \mid R]) \\ \to\ & (\nu g_n : G_n)(\nu n : g_n)n[\,] \mid m[P \mid p[Q] \mid R] \equiv m[P \mid p[Q] \mid R] \end{aligned}$$

Observe that if there is another ambient named m in parallel with the one which is the subject of the definition, the $\mathsf{to}\, m$ action may take the process into the wrong m, i.e., the following reduction is possible:

$$\begin{aligned} & m[p[(\nu g_n : G_n)(\nu n : g_n)n[\mathsf{out}\ p\,.\,\mathsf{out}\ m\,.\,\mathsf{to}\ m\,.\,P] \mid Q] \mid R] \mid m[R'] \\ \to^*\ & m[p[Q] \mid R] \mid m[P \mid R']. \end{aligned}$$

Thus the encoding may nondeterministically allow, besides the effect of the original up action, also a completely different evolution (one might say, following the terminology of [8], that it suffers from a grave interference).

The encoding is consistent w.r.t. typing, in the same sense as the down action considered above.

Encoding the π-calculus

A standard expressiveness test for the Ambient Calculus and its variants is the encoding of communication on named channels via local anonymous communication within ambients. We consider a core fragment of the typed monadic synchronous π-calculus, given by the following grammar (we use letters a–d for channel names and x–z for variables):

$$P \ ::= \ c(x : T).P \ \mid \ c\langle a\rangle.P \ \mid \ (\nu c : T)P \ \mid \ P \mid Q \ \mid \ !P \ \mid \ 0$$

T ranges over a type linear hierarchy:

$$T \ ::= \ Ch() \ \mid \ Ch(T)$$

The reduction relation, which will be denoted by $\to_\pi$, is defined by one basic rule:

$$c(x : T).P \mid c\langle a\rangle.Q \to_\pi P\{x := a\} \mid Q$$

and by structural reduction rules similar to those of Figure 3 for our calculus, except (R-amb) and (R-ν-group).

The type system, defined by the typing rules of Figure 6, derives judgements of the form $\Gamma \vdash P$, where Γ is a set of judgements of the form $c : Ch(T)$. The informal meaning of $\Gamma \vdash P$ is that P is a well-typed process w.r.t. the environment Γ, i.e., communication is well-typed in P w.r.t. assumptions Γ.

The basic idea of the encoding consists, as usual, in representing each channel as an ambient: processes prefixed with a communication action on

$$\frac{\Gamma \vdash c : Ch(T) \quad \Gamma, x : T \vdash P}{\Gamma \vdash c(x : T).P} \qquad \frac{\Gamma \vdash c : Ch(T) \quad \Gamma \vdash a : T \quad \Gamma \vdash P}{\Gamma \vdash c\langle a \rangle.P}$$

$$\frac{\Gamma, c : T \vdash P}{\Gamma \vdash (\nu c : T)P} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \,|\, Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \frac{}{\Gamma \vdash \mathsf{0}}$$

Fig. 6: Type system for π-calculus

a channel c are encoded as mobile processes that first go (in)to the ambient $c[\,]$ where they communicate, and then go back to where they belong. Since, however, a to action can only move a process between sibling ambients, the introduction is needed, in parallel with the channel-ambients, of an ambient p containing the encoding proper $[\![P]\!]$ of the top-level π-calculus term P.

The infinite sequence of π-calculus types $Ch(), Ch(Ch()), \ldots, Ch^n(), \ldots$ is encoded as an infinite sequence of group names $g_0, g_1, \ldots, g_{n-1}, \ldots$ along with the sequence of their respective group types $G_0, G_1, \ldots, G_{n-1}, \ldots$:

$$[\![Ch()]\!] = g_0 \qquad \text{with} \qquad g_0 : G_0 = \mathsf{gr}(\{g, g_p\}, \{g_p\}, \{g_p\}, \mathsf{shh})$$

$$[\![Ch(T)]\!] = g_{j+1} \text{ if } [\![T]\!] = g_j \quad \text{with} \quad g_{j+1} : G_{j+1} = \mathsf{gr}(\{g, g_p\}, \{g_p\}, \{g_p\}, g_j)$$

where g_p is the group of the above-mentioned ambient p, while g is the group of the top-level $\mathbf{M}^3$ ambient-processes, i.e., both the process $p[[\![P]\!]]$ and the channel-processes $a[\ldots]$.

Let P be a π-process, the set $\{c_1, \ldots, c_h\}$ and the integer k be such that:

- $\{c_1, \ldots, c_h\}$ contains the set $\mathsf{fn}(P)$ of the free channel names occurring in P;
- k is greater or equal to the maximum nesting of $Ch()$ in types occurring in P.

The global encoding of P, denoted by $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k)$, is then the process of group g given by:

$$\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k) = p[[\![P]\!]] \,|\, c_1[\,] \,|\, \ldots \,|\, c_h[\,]$$

where $[\![P]\!]$ is defined in Figure 7. Of course, this requires that the (finite) set of free channel names occurring in the term be known in advance. We can, however, always assume to be working on closed terms.

If one defines the translation of a type environment $[\![\Gamma]\!]$ as the set of assumptions $\{c : [\![T]\!] \;\mid\; c : T \in \Gamma\}$, the Theorem 4.1 below states that the translation respects types. Also, the translation is correct in the sense expressed by Theorem 4.2.

$$
\begin{aligned}
[\![c(x:T).P]\!] &= \mathsf{to}\ c.(x:[\![T]\!]).\mathsf{to}\ p.[\![P]\!] \\
[\![c\langle a\rangle.P]\!] &= \mathsf{to}\ c.\langle a\rangle.\mathsf{to}\ p.[\![P]\!] \\
[\![(\nu c:T)P]\!] &= (\nu c:[\![T]\!])(c[\mathsf{out}\ p]\,|\,[\![P]\!]) \\
[\![P\,|\,Q]\!] &= [\![P]\!]\,|\,[\![Q]\!] \\
[\![!P]\!] &= ![\![P]\!] \\
[\![0]\!] &= 0
\end{aligned}
$$

Fig. 7: Encoding of π-calculus

In the following, Π denotes the environment

$$\Pi = \{g_0 : G_0, \ldots, g_k : G_k, g_p : G_p, p : g_p\}$$

where $G_p = \mathsf{gr}(\{g\}, \varnothing, \{g_i \,|\, 0 \leq i \leq k\}, \mathsf{shh})$.

Theorem 4.1 *Let $\Gamma \vdash P$, $\{c_1, \ldots, c_h\} \supseteq \mathsf{fn}(P)$, and k is greater or equal to the maximum nesting of $Ch()$ in types occurring in P. Then, for any group type G: $\Pi, [\![\Gamma]\!], g : G \vdash \mathcal{C}([\![P]\!], c_1, \ldots, c_h, k) : g$.*

Proof. By induction on the structure of the process one can show that, for every well-typed π-process P, the judgement $\Delta \vdash [\![P]\!] : g_p$ is always derivable by taking as Δ the following environment:

$$\Delta = \Pi,\ [\![\Gamma]\!], c_1 : [\![T_1]\!], \ldots, c_h : [\![T_h]\!], g : G$$

As an example, let us consider $P \equiv c(x:T).P'$. We have the following typing derivation for $[\![P]\!]$ (let $c : g_c$ be the type assumption for the name c in Δ):

$$
\dfrac{\dfrac{\dfrac{\Delta, x : [\![T]\!] \vdash \mathsf{to}\ p : g_p \multimap g_c \quad \Delta, x : [\![T]\!] \vdash [\![P']\!] : g_p}{\Delta, x : [\![T]\!] \vdash \mathsf{to}\ p.[\![P']\!] : g_c}}{\Delta \vdash (x : [\![T]\!]).\mathsf{to}\ p.[\![P']\!] : g_c} \qquad \Delta \vdash \mathsf{to}\ c : g_c \multimap g_p}{\Delta \vdash \mathsf{to}\ c.(x : [\![T]\!]).\mathsf{to}\ p.[\![P']\!] : g_p}
$$

The statement of the theorem easily follows, considering the type judgements $\Delta \vdash c_i[\,] : g$, $\Delta \vdash p[[\![P]\!]] : g$ and then using type derivation rules (Par), (Amb Res), and (Grp Res).

□

Theorem 4.2 *Let P be a term of the π-calculus such that $\{c_1, \ldots, c_h\} \supseteq \mathsf{fn}(P)$, and k is greater or equal to the maximum nesting of $Ch()$ in types occurring in P.*
(i) If $P \rightarrow_\pi Q$ then $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k) \rightarrow^* \mathcal{C}([\![Q]\!], c_1, \ldots, c_h, k)$.
(ii) If $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k) \rightarrow^* \mathcal{C}(Q, c_1, \ldots, c_h, k)$, and $Q = [\![R]\!]$ for some π-calculus process R, then $P \rightarrow_\pi^* R$.

Proof. (i) Let us consider, as interesting case, the one where P is of the form $c(x:T).P' \mid c\langle a\rangle.P'' \mid R$, which can reduce to $Q \equiv P'\{x := a\} \mid P'' \mid R$. By definition of $[\![\cdot]\!]$, we have that $[\![P]\!]$ is $\mathsf{to}\, c.(x : [\![T]\!]).\mathsf{to}\, p.[\![P']\!] \mid \mathsf{to}\, c.\langle a\rangle.\mathsf{to}\, p.[\![P'']\!] \mid [\![R]\!]$. Of course, c is a free variable of P, and in the term $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k)$ there is by hypothesis an ambient $c[\,]$ which runs in parallel with the process $[\![P]\!]$. We have therefore the following reduction (we only show the relevant subprocesses):

$$\begin{aligned} & p[\mathsf{to}\, c.(x : [\![T]\!]).\mathsf{to}\, p.[\![P']\!] \mid \mathsf{to}\, c.\langle a\rangle.\mathsf{to}\, p.[\![P'']\!]] \mid c[\,] \\ \rightarrow^* \; & p[\,] \mid c[(x : [\![T]\!]).\mathsf{to}\, p.[\![P']\!] \mid \langle a\rangle.\mathsf{to}\, p.[\![P'']\!]] \\ \rightarrow \; & p[\,] \mid c[\mathsf{to}\, p.[\![P']\!]\{x := a\} \mid \mathsf{to}\, p.[\![P'']\!]] \\ \rightarrow^* \; & p[[\![P']\!]\{x := a\} \mid [\![P'']\!]] \mid c[\,] \end{aligned}$$

whereas $[\![Q]\!]$ is $[\![P'\{x := a\}]\!] \mid [\![P'']\!] \mid [\![R]\!]$.

An easy induction on the definition of the translation concludes the proof by showing that $[\![P\{x := a\}]\!] = [\![P]\!]\{x := a\}$.

(ii) By the definition of the encoding, $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k)$ consists of a parallel composition of processes running inside the ambient $p[\,]$. In this case the only possible move the system can perform is a to action. Consider a subprocess of the form $\mathsf{to}\, c.(x : [\![T]\!]).\mathsf{to}\, p.[\![Q']\!]$: this is the translation of the π-calculus process $Q \equiv (x : T).Q'$, hence $[\![P]\!] \equiv [\![Q]\!] \mid [\![U]\!]$ for some process U. The $\mathsf{to}\, c$ action can always be performed, if $c = c_i$, for some i, or if we are in the scope of a restriction of the c name. Once the $\mathsf{to}\, c$ action is performed, the process will reduce to a process in the form $\mathcal{C}([\![R]\!], c_1, \ldots, c_h, k)$ only if the channel c is cleared. To obtain this the other two actions in the prefix must be fired. This happens only if the input action is consumed inside the ambient c, hence if there is a process $[\![U]\!] \equiv \mathsf{to}\, c.\langle a\rangle.\mathsf{to}\, p.[\![Q'']\!] \mid [\![U']\!]$ that translates a π-calculus process $c\langle a\rangle.Q'' \mid U'$. To conclude the proof, it suffices to observe that after clearing channel c the process $\mathcal{C}([\![P]\!], c_1, \ldots, c_h, k)$ reduces to $\mathcal{C}([\![Q']\!]\{x := a\} \mid [\![Q'']\!] \mid [\![U']\!], c_1, \ldots, c_h, k)$. Lastly observe that $P = (x : T).Q' \mid \langle a\rangle.Q'' \mid U'. \rightarrow_\pi Q'\{x := a\} \mid Q" \mid U'$ and $[\![Q'\{x := a\}]\!] = [\![Q']\!]\{x := a\}$. □

Encoding the Dπ-calculus

We refer to the version of Dπ presented in [7], but assuming a much simpler type system. The basic syntactic categories are shown in Figure 8, where the set of values V consists of channel and location names.

We assume in this presentation a very basic type system similar to that of π, aimed only at preventing communication errors. The main limitation is that channels with the same name in different locations must transmit values of the same type. We assume the following syntax for types:

$$\mathsf{loc} \;\mid\; \mathsf{ch}(T)$$

Thread		*System*	
$p, q, r =$	stop	$P, Q, R =$	$\mathsf{0}$
	$u?(X:T).p$		$P \mid Q$
	$u!\langle V\rangle.p$		$l[p]$
	go $l.p$		$(\nu_l u : \mathsf{ch}(T))P$
	$p \mid q$		
	$*P$		
	$(\nu u : \mathsf{ch}(T))p$		

Fig. 8: Syntax of Dπ-calculus

The reduction semantics of the Dπ-calculus, that we will denote with $\rightarrow_D$, has the following reduction rules:

$$l[u?(X:T).p] \mid l[u!\langle V\rangle.q] \rightarrow_D l[p\{X := V\}] \mid l[q]$$
$$l[\text{go } l'.p] \rightarrow_D l'[p]$$

Operational semantics, as usual, is given together with rules that define structural equivalence. Besides standard ones, it is worth mentioning the following rules, peculiar to Dπ, which state that two locations with the same name are the same location (differently from our calculus), and that we can enlarge the scope of a channel restriction outside a location, provided that we keep information about the location:

$$l[p \mid q] \equiv l[p] \mid l[q]$$
$$l[(\nu u)p] \equiv (\nu_l u) l[p]$$

The typing rules we consider are similar to those of π-calculus (see Figure 9).

Like in the π-calculus encoding, we assume that we know in advance the (finite) set of locations (and of free channel names for each location, written $l^{u_1,\ldots,u_n}$) required to run the process. The set of free channel names used within a location is statically determinable, as shown by Dπ type systems. We also remark that in our version of Dπ channel names are absolute and not relative to locations. This implies that channels with the same name must communicate values of the same type also if they are in different locations.

The basic idea of the translation is the following. Each location l is represented by an ambient $l[\,]$ that contains as subambients the encodings of channels used in communications local to l. Moreover, each location l contains an internal ambient *self* that contains a process $!\langle l\rangle$ offering as output the location name. This is used to allow processes to perform communications on channels

$$\frac{\Gamma \vdash u : \mathsf{ch}(T) \quad \Gamma, x : T \vdash P}{\Gamma \vdash u?(x:T).p} \qquad \frac{\Gamma \vdash u : \mathsf{ch}(T) \quad \Gamma \vdash V : T \quad \Gamma \vdash p}{\Gamma \vdash u!\langle V \rangle.p}$$

$$\frac{\Gamma, u : T \vdash p}{\Gamma \vdash (\nu u : T)p} \qquad \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \,|\, q} \qquad \frac{\Gamma \vdash p}{\Gamma \vdash *p} \qquad \frac{}{\Gamma \vdash \mathrm{stop}}$$

$$\frac{\Gamma, u : T \vdash P}{\Gamma \vdash (\nu_l u : T)P} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \,|\, Q} \qquad \frac{\Gamma \vdash p}{\Gamma \vdash l[p]} \qquad \frac{}{\Gamma \vdash 0}$$

Fig. 9: Type system for Dπ-calculus

located elsewhere with respect to their original site. In particular, a system P using at most locations l_1 (with at most local free channels $u_1^1, \ldots, u_1^{k_1}, k_1 \geq 0$), $\ldots$, l_n (with at most local free channels $u_1^n, \ldots, u_{k_n}^n, k_n \geq 0$) will be represented by:

$$[\![P]\!] \mid l_1[self[!\langle l_1 \rangle] \,|\, u_1^1[\,] \,|\, \ldots u_{k_1}^1[\,]], \mid \ldots \mid, l_n[self[!\langle l_n \rangle] \,|\, u_1^n[\,] \,|\, \ldots u_{k_n}^n[\,]]$$

We will use for this term the notation $\mathcal{D}([\![M]\!], l_1^{u_1^1,\ldots,u_{k_1}^1}, \ldots, l_n^{u_1^n,\ldots,u_{k_n}^n}, k)$, where k is an integer greater or equal to the maximum nesting of $\mathsf{ch}()$ in types occurring in P.

Let now be g_{loc} the group of locations and g_D the group of the whole system. We define the following group and group types:

$$g_0 = g_{loc} : G_{loc} = \mathsf{gr}(\{g_D\}, \varnothing, \{g_{loc}\}, \mathsf{shh})$$
$$g_{i+1} : G_{i+1} = \mathsf{gr}(\{g_{loc}, g_D\}, \{g_{loc}\}, \{g_{aux}\}, \{g_i\})$$

Our types are encoded in the following way:

$$[\![\mathsf{loc}]\!] = g_{loc}$$
$$[\![\mathsf{ch}(T)]\!] = g_{i+1} \quad \text{where } [\![T]\!] = g_i$$

Assume the following groups and group types:

$$g_{aux} : G_{aux} = \mathsf{gr}(\{g_{loc}, g_D\}, \{g_{loc}\}, \{g_{self}, g_{loc}\}, \mathsf{shh})$$
$$g_{self} : G_{self} = \mathsf{gr}(\{g_{loc}\}, \varnothing, \{g_0, \ldots, g_k\}, g_{loc})$$

Moreover assume $l_i : g_{loc}$ (for all location names that occur in the current process), $self : g_{self}$, $u : g_{i+1}$ where $g_i = [\![T]\!]$ and $\mathsf{ch}(T)$ is the type of com-

munications on channel u (for all channel names that occur in the current process).
The encoding of Dπ-calculus terms is given in Figure 10.

THREADS :

$$
\begin{array}{ll}
[\![\text{stop}]\!] & = \mathsf{0} \\
[\![p \mid q]\!] & = [\![p]\!] \mid [\![q]\!] \\
[\![\text{go } l.\, p]\!] & = \mathsf{to}\ l.[\![p]\!] \\
[\![u?(X{:}T).p]\!] & = (\nu n{:}\, g_{aux})n[\mathsf{to}\ \mathit{self}.(x{:}\, g_{loc}).\mathsf{to}\ u.(X{:}\, [\![T]\!]).\mathsf{to}\ n.\mathsf{out}\ x.\mathsf{to}\ x.[\![p]\!]] \\
[\![u!\langle V\rangle.p]\!] & = (\nu\, n{:}\, g_{aux})n[\mathsf{to}\ \mathit{self}.(x{:}\, g_{loc}).\mathsf{to}\ u.\langle [\![V]\!]\rangle.\mathsf{to}\ n.\mathsf{out}\ x.\mathsf{to}\ x.[\![p]\!]] \\
[\![(\nu e{:}\,\mathsf{ch}(T))p]\!] & = (\nu e{:}\, g_{i+1})(e[\,] \mid [\![p]\!]) \quad \text{where } [\![T]\!] = g_i \\
[\![(*\,p)]\!] & = !\,[\![p]\!]
\end{array}
$$

SYSTEM :

$$
\begin{array}{ll}
[\![\mathsf{0}]\!] & = \mathsf{0} \\
[\![P \mid Q]\!] & = [\![P]\!] \mid [\![Q]\!] \\
[\![l[p]]\!] & = (\nu\, n{:}\, g_{aux})n[\mathsf{to}\ l.[\![p]\!]] \\
[\![(\nu_l e{:}\,\mathsf{ch}(T))P]\!] & = (\nu e{:}\, g_{i+1})(e[\mathsf{in}\ l] \mid [\![P]\!]) \quad \text{where } [\![T]\!] = g_i
\end{array}
$$

Fig. 10: Encoding of Dπ-calculus

The correctness of the translation, in the same sense as for the π-calculus, is ensured by analogous theorems, where $\rightarrow_D$ denotes reduction in the Dπ-calculus, and the translation of a type environment $[\![\Gamma]\!]$ is the set of assumptions $\{u : [\![T]\!] \mid u : T \in \Gamma\}$. Moreover let Π_D be the environment:

$$g_{loc} : G_{loc}, g_1 : G_1, \ldots, g_k : G_k, g_{aux} : G_{aux}$$
$$g_{self} : G_{self}, l_1 : g_{loc}, \ldots, l_h : g_{loc}, \mathit{self} : g_{self}$$

Theorem 4.3 *Let P be a term of the Dπ-calculus, such that the set of locations is a subset of $\{l_1, \ldots, l_h\}$, and for each location l_i the set of free channel names located at l_i is a subset of $\{u_i^1, \ldots, u_i^{k_i}\}$ ($\overline{u}_i$ for short). Moreover let j be an integer greater or equal to the maximum nesting of $\mathsf{ch}()$ in types occurring in P. If $\Gamma \vdash P$ then, for any group type G we have: $\Pi_D, [\![\Gamma]\!], g_D : G \vdash \mathcal{D}([\![P]\!], l_1^{\overline{u}_1}, \ldots, l_h^{\overline{u}_h}, j) : g_D$.*

Theorem 4.4 *Let P be a term of the $D\pi$-calculus, such that the set of locations is a subset of $\{l_1, \ldots, l_h\}$, and for each location l_i the set of free channel names located at l_i is a subset of $\{u_i^1, \ldots, u_i^{k_i}\}$ ($\overline{u}_i$ for short). Moreover let j be an integer greater or equal to the maximum nesting of $\mathsf{ch}()$ in types occurring in P. We have:*
(i) If $P \rightarrow_D Q$ then $\mathcal{D}([\![P]\!], l_1^{\overline{u}_1}, \ldots, l_h^{\overline{u}_h}, j) \rightarrow^* \mathcal{D}([\![Q]\!], l_1^{\overline{u}_1}, \ldots, l_h^{\overline{u}_h}, j)$;
(ii) If $\mathcal{D}([\![P]\!], l_1^{\overline{u}_1}, \ldots, l_h^{\overline{u}_h}, j) \rightarrow^* \mathcal{D}(Q, l_1^{\overline{u}_1}, \ldots, l_h^{\overline{u}_h}, j)$, and $Q = [\![R]\!]$, for some $D\pi$ process R, then $P \rightarrow_D^* Q$.

5 Type Inference

In this section, we present a type inference algorithm for our type assignment system. A type inference algorithm is a desirable feature in distributed systems, because it allows a type analysis to be performed even when only incomplete type assumptions about a process are available.

Given a raw process P, i.e., a well-formed process in which all type annotations have been erased, our type inference algorithm introduces the needed type annotations and computes the environment satisfying the minimal requirements on the typings of the (free and bound) names occurring in P and in the related groups, thus producing a process P' which is well typed w.r.t. such environment. The typing given to P' is principal in the sense of [13], since all other possible typings that can be given to processes obtained from P by introducing type annotations can be derived through a set of suitable operations from the inferred typing of P'. The inference algorithm is then proved to be sound and complete with respect to the rules of section 3.

Type Variables, Substitutions and Environment Operations

We first introduce some technical tools that will be useful in defining the inference procedure and its properties. The algorithm deals with type variables, substitutions, unifiers, and merging of type environments; moreover, a partial order relation on group types and type environments is needed for the derivation of a principal typing property.

Let a *raw* process R be a well-formed process in which all type annotations have been erased. In a raw process, in particular, group restrictions are missing, name restrictions are the form $\nu n.R$, and input is of the form $(x).P$. If P is a process, its raw form is the raw process $|P|$ obtained from P by erasing all group restrictions and all type annotations.

In order to describe the inference algorithm, it is necessary to generalize the previously introduced syntactic categories in the following four respects:

- we extend the syntax of exchange types by a set $\mathcal{V}$ of type variables (denoted by t);
- we extend the syntax of processes by allowing exchange types to be variables in $\mathcal{V}$;

- we extend the syntax of group types by allowing the sets $\mathscr{S}$ to contain starred group names (denoted by g^*);
- we allow non-well-formed environments.

The introduction of type variables and of non-well-formed environments is standard in type inference algorithms. The use of starred group names is a tool for taking into account the fact that if an ambient m goes out of another ambient n (consuming an $\mathsf{out}\ n$ capability), then the set of ambients where m can stay must include the set of ambients where n is allowed to stay.

As usual, in computing types and type environments the algorithm is driven by the syntax of the process; it has therefore to put together distinct environments whenever the process has more than one subprocess. A fresh group name is assigned to each name, but when different environments are put together groups must be equated. This is achieved by means of substitutions. A substitution maps group names to group names and type variables to communication types (extended with variables). Let $\mathbb{T}$ be the set of communication types.

Definition 5.1 A *substitution* is a finite mapping in $(\mathbb{G} \to \mathbb{G}) \cup (\mathcal{V} \to (\mathbb{T} \cup \mathcal{V}))$. Let φ_i range over $\mathbb{G} \cup \mathcal{V}$ and T_i over $\mathbb{T} \cup \mathcal{V}$. A substitution σ can be represented as an expression $[\varphi_1 := T_1, \ldots, \varphi_n := T_n]$, where $i \neq j$ implies $\varphi_i \neq \varphi_j$. As usual, we assume

$$\sigma(\varphi) = \begin{cases} T_j & \text{if } \varphi = \varphi_j \text{ for some } 1 \leq j \leq n \\ \varphi & \text{otherwise.} \end{cases}$$

Hence, for a type variable t, $\sigma(t)$ can be either an element of $\mathbb{G} \cup \mathcal{V}$, or shh, or a capability type of the form $\varphi_1 \to \varphi_2$. The application of a substitution to an environment (denoted by $\sigma(\Gamma)$) is defined in the standard way. A substitution can also be applied to a process P. If σ restricted to group names is $[g_1 := g'_1, \ldots, g_n := g'_n]$, we assume that $g'_1, \ldots, g'_n$ are all distinct from the names of the groups occurring in P under the operators of group restriction. This condition can always be satisfied by an α-renaming of the restricted group names in P.

Type inference is not, in general, closed under substitution since, taken an arbitrary substitution σ, $\sigma(\Gamma)$ could be non-well-formed. For instance if $g_1 : G_1, g_2 : G_2 \in \Gamma$ we could have $\sigma(g_1) = \sigma(g_2)$ but $\sigma(G_1) \neq \sigma(G_2)$. However it is easy to see that if $\Gamma \vdash P : g$ and $\sigma(\Gamma)$ is well-formed then $\sigma(\Gamma) \vdash \sigma(P) : \sigma(g)$.

When two different environments are put together, it may be necessary to *unify* different message types. This is impossible if they are an ambient type and a capability type: in such case we get a *failure*. Therefore, in the definitions below, operations on types and environments may yield an *undefined* result denoting failure. A failure is raised whenever none of the cases considered in definitions can be applied. Failure propagates, and an operation produces a *undefined* result whenever some step in its evaluation produces *undefined*. To

simplify notation, we assume failure propagation as understood and we avoid indicating it explicitly in the definitions.

Definition 5.2 We denote by $\phi(\{(T_i, T_i')|1 \leq i \leq n\})$ the *two-sorted most general unifier* of the set of equations $\{T_i = T_i' \mid 1 \leq i \leq n\}$ such that it is a substitution according to definition 5.1. We will simply call $\phi(\{(T_i, T_i')|1 \leq i \leq n\})$ the *unifier* of $\{(T_i, T_i')|1 \leq i \leq n\}$.

As already observed, the application of a substitution σ to an environment Γ gives an environment that is not, in general, well-formed, because σ can map two distinct group names of $Dom(\Gamma)$ to the same group name. In the sequel, we show how to recover a well-formed environment after the application of a substitution. This task is performed in two steps:

(i) by unifying the communication types in different type assumptions for the same group name (*completion*, Definition 5.3);

(ii) by merging (by componentwise set union) group types having the same communication type (*compression*, Definition 5.5).

Completion and compression are also useful to recover a well-formed environment when the algorithm needs to merge two environments, for example in typing a parallel composition. They will also be used in the Definition 5.6 of the *plus-union* operation.

Definition 5.3 (i) Let χ range over (action or name) variables and ambient names. An environment Γ is *consistent* if for all $\chi : T_1,\ \chi : T_2 \in \Gamma$ we have that $T_1 = T_2$ and for all $g : G_1,\ g : G_2 \in \Gamma$ we have that $T(G_1) = T(G_2)$. This last condition does not imply $G_1 = G_2$, so consistent environments are not in general well-formed.

(ii) The *completion* $\Sigma(\Gamma)$ of an environment is the substitution obtained by the following steps:

1. if Γ is consistent, return the empty substitution $[\,]$;
2. if $\chi : T_1,\ \chi : T_2 \in \Gamma$ with $T_1 \neq T_2$ let $\sigma = \phi(T_1, T_2)$. Then return $\Sigma(\sigma(\Gamma)) \circ \sigma$.
3. if $g : G_1,\ g : G_2 \in \Gamma$ with $T(G_1) \neq T(G_2)$ let $\sigma = \phi(T(G_1), T(G_2))$. Then return $\Sigma(\sigma(\Gamma)) \circ \sigma$.

The completion procedure always terminates, either with a failure or by returning a substitution σ. Such termination property is due to the fact that the number of inconsistent pairs, with respect to the computed substitution, decreases at each step. Moreover, $\sigma(\Gamma)$ is a consistent environment whenever it is defined (but it is not in general well-formed). The basic properties of completion are formalized by the following.

Lemma 5.4 *Let Γ be a type environment and $\sigma = \Sigma(\Gamma)$. If $\Sigma(\Gamma)$ is defined, then:*

(i) *$\sigma(\Gamma)$ is a consistent environment;*

(ii) *for all* σ' *such that* $\sigma'(\Gamma)$ *is consistent, there is a substitution* σ'' *such that* $\sigma' = \sigma'' \circ \sigma$.

Otherwise there is no substitution σ *such that* $\sigma(\Gamma)$ *is consistent.*

Definition 5.5 The *compression* $\alpha(\Gamma)$ of a consistent environment Γ is the well-formed environment Γ' such that:

- $\chi : T \in \Gamma'$ if $\chi : T \in \Gamma$ and χ is a (action or name) variable or an ambient name
- $g : \mathsf{gr}(\bigcup_{i\in I} \mathscr{S}_i,\ \bigcup_{i\in I} \mathscr{C}_i,\ \bigcup_{i\in I} \mathscr{E}_i,\ T) \in \Gamma'$, where $I = \{i \mid g : \mathsf{gr}(\mathscr{S}_i, \mathscr{C}_i, \mathscr{E}_i, T) \in \Gamma\}$.

Using completion and compression, we define the *plus-union* operation, which merges (if it is possible) two environments Γ_1 and Γ_2 having disjoint group types into a single well-formed one. Note that Γ_1 and Γ_2 are not assumed to be well-formed or consistent.

Definition 5.6 Let Γ_1 and Γ_2 be two environments, and let χ range over (action or name) variables and ambient names. Define:

$$\begin{array}{lcll} \overline{\Phi}(\Gamma_1, \Gamma_2) = & & \Sigma(\sigma(\Gamma_1 \cup \Gamma_2)) \circ \sigma & \\ & & \text{where } \sigma = \phi(\{(T_1, T_2) \mid \chi : T_1 \in \Gamma_1 \ \&\ \chi : T_2 \in \Gamma_2\}) & \\ \Gamma_1 \uplus \Gamma_2 & = & \alpha(\sigma'(\Gamma_1 \cup \Gamma_2)) & \text{where } \sigma' = \overline{\Phi}(\Gamma_1, \Gamma_2). \end{array}$$

$\overline{\Phi}(\Gamma_1, \Gamma_2)$ is called the *unifier* of Γ_1 and Γ_2 and is denoted by $\overline{\Phi}(\Gamma_1, \Gamma_2)$.

Note that $\sigma'(\Gamma_1 \cup \Gamma_2)$ is consistent but not well-formed. In order to discuss properties of the $\uplus$ operator (and of the algorithm), we introduce a partial order on types and environments.

Definition 5.7 The partial order on group types is defined by letting:

$$\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T) \leq \mathsf{gr}(\mathscr{S}', \mathscr{C}', \mathscr{E}', T')$$

if $\mathscr{S} \subseteq \mathscr{S}'$ and $\mathscr{C} \subseteq \mathscr{C}'$ and $\mathscr{E} \subseteq \mathscr{E}'$ and ($T = \mathsf{shh}$ or $T = T'$).
This order is extended monotonically to environments by adding set inclusion:

$$\Gamma \leq \Gamma' \iff \forall \xi : A \in \Gamma\ \exists \xi : A' \in \Gamma' \text{ such that } A \leq A'.$$

In the soundness proof, it is also useful to consider two variants of $\leq$.

Definition 5.8 . We define:

- $\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T) \leq_{\mathscr{S}} \mathsf{gr}(\mathscr{S}', \mathscr{C}, \mathscr{E}, T)$ if $\mathscr{S} \subseteq \mathscr{S}'$;
- $\mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T) \leq_{-\mathscr{S}} \mathsf{gr}(\mathscr{S}', \mathscr{C}, \mathscr{E}, T)$ if $\mathscr{C} \subseteq \mathscr{C}'$, $\mathscr{E} \subseteq \mathscr{E}'$ and ($T = \mathsf{shh}$ or $T = T'$).

Note that $G \leq G'$ iff $G \leq_{\mathscr{J}} G'$ and $G \leq_{-\mathscr{J}} G'$. Type assignment is preserved by $\leq$ in the following sense.

Lemma 5.9 *If* $\Gamma \vdash P : g$ *and* $\Gamma \leq \Gamma'$ *then* $\Gamma' \vdash P : g$.

The basic properties of the $\uplus$ operator are formalized by the following lemma.

Lemma 5.10 *Let* Γ_1 *and* Γ_2 *be two environments. Then* $\Gamma_1 \uplus \Gamma_2$ *is a well-formed environment.*
Moreover, for all substitutions σ *and well-formed environments* Γ' *such that* $\sigma(\Gamma_1 \cup \Gamma_2) \leq \Gamma'$, *there is a substitution* σ'' *such that* $\sigma''(\Gamma_1 \uplus \Gamma_2) \leq \Gamma'$.

$$\vdash_{\mathtt{I}} \chi : \langle t; \{\chi : t\}\rangle \quad \text{(I-Name)} \qquad \text{where } \chi \text{ is a variable or an ambient name}$$

$$\vdash_{\mathtt{I}} \mathsf{to}\ M : \langle g_1 \to g_2; \{M : g_1, g_1 : \mathsf{gr}(\varnothing,\varnothing,\varnothing,t_1), g_2 : \mathsf{gr}(\varnothing,\varnothing,\{g_1\},t_2)\}\rangle \quad \text{(I-to)}$$

$$\vdash_{\mathtt{I}} \mathsf{in}\ M : \langle g_2 \to g_2; \{M : g_1, g_1 : \mathsf{gr}(\varnothing,\varnothing,\varnothing,t_1), g_2 : \mathsf{gr}(\{g_1\},\{g_1\},\varnothing,t_2)\}\rangle \quad \text{(I-in)}$$

$$\vdash_{\mathtt{I}} \mathsf{out}\ M : \langle g_2 \to g_2; \{M : g_1, g_1 : \mathsf{gr}(\varnothing,\varnothing,\varnothing,t_1), g_2 : \mathsf{gr}(\{g_1^*\},\{g_1\},\varnothing,t_2)\}\rangle \quad \text{(I-out)}$$

$$\frac{\vdash_{\mathtt{I}} M : \langle W; \Gamma\rangle \qquad \vdash_{\mathtt{I}} N : \langle W'; \Gamma'\rangle}{\vdash_{\mathtt{I}} M.N : \langle \overline{\Phi}(\sigma(\Gamma),\sigma(\Gamma'))(g_3 \to g_2); \sigma(\Gamma) \uplus \sigma(\Gamma'), g_i : \mathsf{gr}(\varnothing,\varnothing,\varnothing,t_i)\rangle} \quad \text{(I-Path)}$$

where $\sigma = \phi(\{(W, g_1 \to g_2), (W', g_3 \to g_1)\})$ and $i = 1,2,3$

Fig. 11: Type reconstruction for messages

Type Inference Algorithm

The inference procedure is defined in natural semantics style. Type inference for messages is represented by a judgment

$$\vdash_{\mathtt{I}} M : \langle W; \Gamma\rangle$$

where W is the message type inferred for M from the environment Γ. The inference and type reconstruction procedure for raw processes is represented by a judgement

$$\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g; \Gamma\rangle$$

where R is a raw process and P is a still incomplete typed version of R (there are still type variables and starred group names). The meaning of g, Γ is as before. The final inference judgment has the form

$$\vdash_{\mathtt{R}} R \Rightarrow P : \langle g; \Gamma\rangle$$

which gives the (most general) typing for the raw process R. Indeed, $\vdash_{\mathtt{R}}$ is a *type reconstruction* procedure which recovers a well typed term from a raw one which, as such, does not belong to the language.

$$\vdash^{-}_{\mathtt{R}} 0 \Rightarrow 0 : \langle g; \{g : \mathsf{gr}(\varnothing, \varnothing, \varnothing, t)\}\rangle \quad \text{(I-Null)}$$

$$\frac{\vdash_{\mathtt{I}} M : \langle W; \Gamma\rangle \qquad \vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g_1; \Gamma'\rangle}{\vdash^{-}_{\mathtt{R}} M.R \Rightarrow M.\sigma'(P) : \langle \sigma'(g_2); \sigma(\Gamma) \uplus \sigma(\Gamma')\rangle} \text{(I-Prefix)}$$
$$\text{where } \sigma = \phi(W, g_1 \to g_2) \text{ and } \sigma' = \overline{\Phi}(\sigma(\Gamma), \sigma(\Gamma'))$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g;\ \Gamma, x : W\rangle \quad g : G \in \Gamma}{\vdash^{-}_{\mathtt{R}} (x).R \Rightarrow (x : \sigma(W)).\sigma(P) : \langle \sigma(g); \alpha(\sigma(\Gamma))\rangle} \text{(I-Input)}$$
$$\text{where } \sigma = \Sigma(\phi(W, T(G))(\Gamma))$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g;\ \Gamma\rangle \quad g : G \in \Gamma \quad x \notin \mathit{Dom}(\Gamma)}{\vdash^{-}_{\mathtt{R}} (x).R \Rightarrow (x : T(G)).P : \langle g; \Gamma\rangle} \text{(I-Input-Fresh)}$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g;\ \Gamma\rangle \qquad g : G \in \Gamma \qquad \vdash_{\mathtt{I}} M : \langle W; \Gamma'\rangle}{\vdash^{-}_{\mathtt{R}} \langle M\rangle.R \Rightarrow \langle M\rangle.\sigma'(P) : \langle \sigma'(g); (\sigma(\Gamma) \uplus \sigma(\Gamma'))\rangle} \text{(I-Output)}$$
$$\text{where } \sigma = \phi(W, T(G)) \text{ and } \sigma' = \overline{\Phi}(\sigma(\Gamma), \sigma(\Gamma'))$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g_1; \Gamma\rangle \quad \vdash^{-}_{\mathtt{R}} S \Rightarrow Q : \langle g_2; \Gamma'\rangle}{\vdash^{-}_{\mathtt{R}} R \,|\, S \Rightarrow \sigma(P \,|\, Q) : \langle \sigma(g_2); \Gamma'' \uplus \Gamma'\rangle} \text{(I-Par)}$$
$$\text{where } \Gamma'' = [g_1 := g_2](\Gamma) \text{ and } \sigma = \overline{\Phi}(\Gamma'', \Gamma')$$

$$\frac{\vdash_{\mathtt{I}} \chi : \langle t; \chi : t\rangle \qquad \vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g;\ \Gamma, g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)\rangle}{\vdash^{-}_{\mathtt{R}} \chi[R] \Rightarrow \chi[\sigma(P)] : \langle g';\ \Gamma', \sigma(g) : \mathsf{gr}(\mathscr{S}' \cup \{g'\}, \mathscr{C}', \mathscr{E}', T'),\ g' : \mathsf{gr}(\varnothing, \varnothing, \varnothing, t')\rangle} \text{(I-Amb)}$$
$$\text{where } \sigma = \overline{\Phi}(\chi : g\,,\ (\Gamma, g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)))$$
$$\text{and } \chi : g \uplus (\Gamma, g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)) = \Gamma'\,,\ \sigma(g) : \mathsf{gr}(\mathscr{S}', \mathscr{C}', \mathscr{E}', T')$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g; \Gamma, n : g'\rangle}{\vdash^{-}_{\mathtt{R}} \nu n.R \Rightarrow (\nu n : g')P : \langle g; \Gamma'\rangle} \text{(I-Res)}$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g; \Gamma, n : t\rangle}{\vdash^{-}_{\mathtt{R}} \nu n.R \Rightarrow (\nu n : g')[t := g'](P) : \langle g; [t := g'](\Gamma')\rangle} \text{(I-Var-Res)}$$

$$\frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g; \Gamma\rangle}{\vdash^{-}_{\mathtt{R}}! R \Rightarrow! P : \langle g; \Gamma\rangle} \text{(I-Repl)} \qquad \frac{\vdash^{-}_{\mathtt{R}} R \Rightarrow P : \langle g;\ \Gamma\rangle}{\vdash_{\mathtt{R}} R \Rightarrow P : \langle g; \mathcal{F}(\Gamma,\ g)\rangle} \text{(I-final)}$$

Fig. 12: Type reconstruction for raw processes

Figure 11 gives the inference rules for messages ($\vdash_{\mathtt{I}}$) and Figure 12 those for raw processes ($\vdash^{-}_{\mathtt{R}}$ and $\vdash_{\mathtt{R}}$). In all the type inference rules, group names and type variables that appear in the conclusion and do not appear in the premises are fresh.

In all the rules with two premises (i.e., rules (I-Path), (I-Prefix), (I-Output) and (I-Par)), the algorithm merges the two environments of the premises by

means of the $\uplus$ operator. In rules (I-Path), (I-Prefix) and (I-Par) two groups are identified. More precisely: in the rule (I-Path) the output group of N is identified with the input group of M; in rule (I-Prefix) the input group of M is identified with the group of P; in rule (I-Par) the group name subset of P and Q.

In the rule (I-Output), the algorithm identifies the communication types of P and M using the unifier defined in Definition 5.2. The same kind of unification is performed by the rule (I-Input).

In the (I-Amb) rule the ambient name χ can only be obtained by using rule (I-Name). The type of the resulting process is a fresh group name g' with group type $\mathsf{gr}(\varnothing, \varnothing, \varnothing, t)$, where t is fresh. The group g' is added to the set of the ambient groups where g ambients can stay.

In order to complete the type inference algorithm (rule (I-final) in Figure 12), we have to get rid of the type variables which occur only in the fourth components of group types and of the marker $*$ on group names. Notice that the marker $*$ is only introduced in rule (I-Output): it has no effect on the above operations and does not propagate in substitutions (it is attached to some occurrences of group names, not to the group names themselves). We employ it for obtaining the correct sets of the groups of ambients where an ambient can stay. A distinguished group name $g_\top$ is assumed, which has no associated group type and can be interpreted as the group type of a virtual process representing the top level ambient [9]. It must belong to the $\mathscr{S}$ component of the group type of all ambients that can run at top level. We have to avoid empty $\mathscr{S}()$ components in the final basis: we do this by using fresh group names. Moreover, all type variable which occur only in the fourth components of group types are replaced by shh. This is done by means of the following operations.

Definition 5.11 (i) The *closure* of an environment Γ (written $\mathcal{C}(\Gamma)$), where Γ may contain occurrences of $*$, is the environment computed as follows:

repeat
if *for some* $g^* \in \mathscr{S}(G)$, $g : G' \in \Gamma$, and $\mathscr{S}(G') \not\subseteq \mathscr{S}(G)$ **then** replace $\mathscr{S}(G)$ by $\mathscr{S}(G) \cup \mathscr{S}(G')$
until *there are no more* g^* *satisfying the above condition*
Then erase all *.

(ii) The *finishing* of an environment $\Gamma = \Gamma', g : \mathsf{gr}(\mathscr{S}, \mathscr{C}, \mathscr{E}, T)$ w.r.t. the group g is the environment obtained by replacing in $\mathcal{C}(\Gamma',\ g : \mathsf{gr}(\mathscr{S} \cup \{g_\top\}, \mathscr{C}, \mathscr{E}, T))$;
- $\mathscr{S}(G)$ by $\{g'\}$ if $\mathscr{S}(G) = \varnothing$ where all g' are fresh
- all type variables (elements of $\mathcal{V}$) which occur only in the components of group types by shh.

[9] i.e. if $\top$ is the name of an ambient including the top level process P, $g_\top$ is the group of the process $\top[P]$.

Let $\mathcal{F}(\Gamma, g)$ denote the finishing of Γ w.r.t. g.

Note that $\Gamma \leq_{\mathscr{S}} \mathcal{C}(\Gamma)$.

Soundness and Completeness

In this subsection, we sketch the soundness and completeness proofs of the inference algorithm. To this aim, we introduce restricted versions of our type assignment system. Let $\vdash_{\text{-}\nu\text{g}}$ denote type assignment system for a variant of our language in which there are no group restrictions (i.e. rule (Grp Res)). Moreover, let ${\vdash_{\text{-}\nu\text{g}}}^{+\mathcal{V}}$ denote type assignment for a variant of $\vdash_{\text{-}\nu\text{g}}$ in which also type variables are permitted to occur in communication types. Note that the inference rules in Figure 5 are not affected by the presence of variables. Finally, let $\vdash_{\text{-}\nu\text{g}\_\mathscr{S}}^{+\mathcal{V}}$ denote the type inference system obtained from ${\vdash_{\text{-}\nu\text{g}}}^{+\mathcal{V}}$ by ignoring the $\mathscr{S}$ fields of group types. This implies the elimination of the premises involving $\mathscr{S}$ in rules (Out) and (Amb).

In the following, Π denotes a process or an action and τ an ambient type or a message type. Deductions in ${\vdash_{\text{-}\nu\text{g}}}^{+\mathcal{V}}$ and $\vdash_{\text{-}\nu\text{g}\_\mathscr{S}}^{+\mathcal{V}}$ are easily connected.

Lemma 5.12 *(i) Let* D *be a deduction of* $\Gamma \vdash_{\text{-}\nu g}^{+\mathcal{V}} \Pi : \tau$. *Then*

(1) For any occurrence of a statement

$$\Gamma, m : g_1, g_1 : G_1, g_2 : G_2 \vdash_{\text{-}\nu g}^{+\mathcal{V}} \mathsf{out}\ m : g_2 \to g_2$$

in D, $\mathscr{S}(G_1) \subseteq \mathscr{S}(G_2)$ *holds.*

(2) For any occurrence of a statement $\Gamma, m : g, g : G \vdash_{\text{-}\nu g}^{+\mathcal{V}} m[P] : g'$ *in* D *we have that* $g' \in \mathscr{S}(G)$.

(ii) Take a derivation of $\Gamma \vdash_{\text{-}\nu g\_\mathscr{S}}^{+\mathcal{V}} \Pi : \tau$ *which satisfies points (1) and (2) of lemma 5.12. Then* $\Gamma \vdash_{\text{-}\nu g}^{+\mathcal{V}} \Pi : \tau$.

It is now easy to see, by induction on deductions, that if $\Gamma \vdash_{\text{-}\nu g\_\mathscr{S}}^{+\mathcal{V}} \Pi : \tau$ the environment $\mathcal{C}(\Gamma)$ is the minimal well-formed environment (with respect to $\leq$) which satisfies the conditions (1) and (2) of lemma 5.12.

Lemma 5.13 *Let* $\Gamma \vdash_{\text{-}\nu g\_\mathscr{S}}^{+\mathcal{V}} \Pi : \tau$. *Then* $\mathcal{C}(\Gamma) \vdash_{\text{-}\nu g}^{+\mathcal{V}} \Pi : \tau$. *Moreover if* $\Gamma' \vdash_{\text{-}\nu g}^{+\mathcal{V}} \Pi : \tau$, *where* $\Gamma \leq_{\mathscr{S}} \Gamma'$ *then* $\mathcal{C}(\Gamma) \leq_{\mathscr{S}} \Gamma'$.

The main lemma for the soundness proof is the following.

Lemma 5.14 *(i) If* $\vdash_{\mathtt{I}} M : \langle T; \Gamma \rangle$ *then* $\Gamma \vdash_{\text{-}\nu g\_\mathscr{S}}^{+\mathcal{V}} M : T$.
(ii) Let $\Gamma \vdash_{\mathtt{R}}^{-} R \Rightarrow P : \langle g; \Gamma \rangle$ *then* $\Gamma \vdash_{\text{-}\nu g\_\mathscr{S}}^{+\mathcal{V}} P : g$.

Statements (i) and (ii) are proved simultaneously by induction on the typing deduction (systems $\vdash_{\mathtt{I}}$ and $\vdash_{\mathtt{R}}^{-}$) using lemma 5.9.

From lemmas 5.14 and 5.13, we have that $\Gamma \vdash_{\mathtt{R}}^{-} R \Rightarrow P : \langle g; \Gamma \rangle$ implies $\Gamma \vdash_{\text{-}\nu g}^{+\mathcal{V}} P : g$ (similar for actions). Finally, the finishing function replaces with shh all the type variables that occur only in the fourth components of

$$
\begin{array}{ll}
Gout^+(\mathsf{0},\Delta) & = \langle \mathsf{0},\Delta\rangle \\
Gout^+(M\,.\,P,\Delta) & = \langle M\,.\,P',\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+(\langle M\rangle\,.\,P,\Delta) & = \langle\langle M\rangle\,.\,P',\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+((x{:}W)\,.\,P,\Delta) & = \langle (x{:}W)\,.\,P',\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+(P\mid Q,\Delta) & = \langle P'\mid Q',\Delta'\cup\Delta''\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
 & \qquad \text{and } \ Gout^+(Q,\Delta)=\langle Q',\Delta''\rangle \\
Gout^+(M[P],\Delta) & = \langle M[P'],\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+(!\,P,\Delta) & = \langle !\,P',\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+((\nu n{:}\,g)P,\Delta) & = \langle (\nu n{:}\,g)P',\Delta'\rangle \text{ if } Gout^+(P,\Delta)=\langle P',\Delta'\rangle \\
Gout^+((\nu\{g_1{:}\,G_1,\ldots,g_k{:}\,G_k\})P,\Delta) & = \langle P',\Delta'\cup\{g_1:G_1,\ldots,g_k:G_k\}\rangle \\
 & \qquad \text{if } \ Gout^+(P,\Delta)=\langle P',\Delta'\rangle
\end{array}
$$

Fig. 13: Definition of $Gout^+(\cdot,\cdot)$

group types. By an inspection of the inference rules, it is easy to see that such variables can occur only in one group type and never as components of a capability type. So there is no harm in replacing them with shh. In the resulting term, there can still be left some type variables in input variables (of the form $(x:t)$) which can be replaced by arbitrary communication types.

The soundness result is stated as follows.

Theorem 5.15 (Soundness) (i) *If* $\vdash_{\mathtt{I}} M : \langle g_1 \to g_2;\Gamma\rangle$ *then* $\Gamma \vdash M : g_1 \to g_2$.

(ii) *Let* $\vdash_{\mathtt{R}} R \Rightarrow P : \langle g;\Gamma\rangle$. *Then* $R = |P|$ *and* $\sigma_0(\Gamma) \vdash \sigma_0(P) : g$ *where* σ_0 *is any substitution that replaces the remaining type variables in* P, Γ *by arbitrary communication types.*

Completeness holds for all typings in which group restrictions are not introduced. This is not restrictive since group restrictions can always be added and moved inside, while respecting the side conditions of rule (Grp Res). To the aim of taking away group restrictions from a typing deduction, we associate to each process the pair consisting of the process obtained by removing all group restrictions and the environment containing all removed group restrictions (mapping $Gout(\cdot)$).

Definition 5.16 The mapping $Gout(\cdot)$ associates to each process a pair ⟨process, basis⟩ as follows:

$$Gout(P) = Gout^+(P,\varnothing)$$

where $Gout^+(\cdot,\cdot)$ is defined in Figure 13.

Assuming that in a typing deduction all bound group names are different from each other and from the free ones (this can always be achieved by α-

conversion), we have immediately the following property.

Lemma 5.17 *Let* $\Gamma \vdash P : g$ *and* $Gout(P) = \langle P', \Delta \rangle$. *Then* $\Gamma \cup \Delta$ *is well-formed and* $\Gamma \cup \Delta \vdash_{\nu g} P' : g$.

Finally, we remark that the type inference algorithm adds a typing assumption for each group name which appears in a type. This condition does not hold in general for the type assignment system and therefore we need to add these assumptions in the comparison between inference and assignment.

Using lemmas 5.4 and 5.10, we can prove by induction on the typing proof of a term the following property.

Lemma 5.18 (i) *Let* $\Gamma \vdash M : g_1 \to g_2$. *Then* $\vdash_{\mathtt{I}} M' : \langle g_1' \to g_2'; \Gamma' \rangle$ *is defined. Moreover there exists a substitution* σ *such that* $\sigma(g_i') = g_i$ *(*$i = 1, 2$*) and* $\sigma(\Gamma') \leq_{-\mathscr{S}} \Gamma$.

(ii) *Let* $\Gamma \vdash_{\nu g} P : g$. *Then* $\vdash_{\mathtt{R}}^{-} |P| \Rightarrow P' : \langle g'; \Gamma' \rangle$ *is defined. Moreover there exists a substitution* σ *such that* $\sigma(g') = g$ *and* $\sigma(\Gamma') \leq_{-\mathscr{S}} \Gamma$.

Completeness of the type inference procedure follows using lemmas 5.13, 5.17, and 5.18.

Theorem 5.19 *If* $\Gamma \vdash P : g$ *and* $Gout(P) = \langle Q, \Delta \rangle$ *then there are* P', Γ', g' *and* σ *such that:*

- $\vdash_{\mathtt{R}} |P| \Rightarrow P' : \langle g'; \Gamma' \rangle$
- $\sigma(g') = g$
- $\sigma(P') \equiv Q$
- $\sigma(\Gamma') \leq \Gamma \cup \Delta \cup \{g_1 : \mathsf{gr}(\{g_1'\}, \varnothing, \varnothing, \mathsf{shh}), \ldots, g_h : \mathsf{gr}(\{g_h'\}, \varnothing, \varnothing, \mathsf{shh})\}$

where $\{g_1, \ldots, g_h\} = \{g' \in Dom(\Gamma) \mid g' \notin Dom(\Gamma')\}$ *and* $g_1', \ldots, g_h'$ *are fresh.*

6 Conclusion

We have presented a simple calculus that combines ambient mobility with general process mobility, putting together the standard in and out Mobile Ambients actions with the to primitive of the Distributed π-calculus [7]. As observed in section 4, other choices were possible for process mobility, namely moving up to the parent ambient, or down to a child ambient.

With the down primitive alone, even in the presence of the in and out ambient primitives, it is impossible to express the up and to moves.

The $\mathsf{up}\, m \,.\, P$ construct (m being the destination ambient) is the most powerful of the three, since it allows the other two kinds of process movements to be expressed by means of it, with the help of in and out; in particular, the $\mathsf{down}\, m \,.\, P$ construct may be obtained in a context-free way as $(\nu p)p[\mathsf{in}\, m \,.\, \mathsf{up}\, m \,.\, P]$, while a $\mathsf{to}\, m.P$ process may be contextually encoded, within an ambient n, by the term $(\nu p)p[\mathsf{out}\, n \,.\, \mathsf{in}\, m \,.\, \mathsf{up}\, m \,.\, P]$.

Since the up primitive is deterministic (the parent ambient is only one), the encodings of the other two constructs do not contain any additional nondeterminism and are therefore correct in a strong sense, in contrast with the opposite encoding of up by means of to, where the latter's inherent nondeterminism w.r.t. homonymous ambients does not allow an expression of the former that is correct in all contexts, as remarked in section 4.

Nevertheless, we have chosen to adopt the to primitive for its greater meaningfulness in the context of mobile computing, and its grounding in a well-known foundational system such as D-π (whose untyped calculus can thus be very easily encoded in our system).

We also have chosen, for the moment, to privilege simplicity. The type system is indeed defined in such a way that subject reduction almost holds by definition. The seeming complexity of the four components of the group types is not real. From the point of view of the type inference, the $\mathscr{C}$ component merely records in and out actions, while the $\mathscr{E}$ component records to moves. From the type-checking perspective, $\mathscr{C}$ simply lists the ambient groups whereinto or wherefrom a given ambient is permitted to move (driven by in or out, respectively), while $\mathscr{E}$ lists the groups whereto a given ambient may send processes.

The $\mathscr{C}$ component, already present in other type systems (for example, in [9]), is essential for controlling ambient mobility, as shown in all the examples of section 4: for instance, the $\mathscr{C}$ components of the various types state that the mail server and the mailboxes are immobile, and so is the firewall, and Troy and the king's palace; at the same time they permit the user to enter its own mailbox only, they permit an agent to cross the firewall, etc.

The novel $\mathscr{E}$ component is needed to control the potentially most dangerous to moves, as is also apparent from the examples: the agent ambient is not authorized to send processes into the firewall-protected site, while the firewall is authorized to send the in capability to the agent. Observe that, as usual, types are only static preconditions that do not prevent more restrictive properties from being checked at runtime; for example, though the agent is granted permission to cross the firewall by its group type, it cannot actually do it if in addition it is not provided at runtime with the appropriate capability.

$\mathscr{C}$ and $\mathscr{E}$ serve therefore different purposes, and are not interrelated: as is apparent, the former is mainly intended to control mobility, the latter is relevant for security.

The $\mathscr{S}$ component, on the other hand, is a superset of $\mathscr{C}$; it is not directly connected with security, and the reasons for its presence are not compelling (as a matter of fact, in the first version of the system it was absent, and was added later). However, without $\mathscr{S}$ the control of ambient mobility is rather lop-sided, owing to the fact that in the standard out m construct the argument m is the ambient one comes out of, instead of that whereinto one enters (like in the to m construct). Thus the $\mathscr{C}$ component cannot control which ambients are allowed to enter a given ambient from downwards, and

another more general set of permissions is necessary.

For instance, without $\mathscr{S}$ the Trojans have no way of knowing that Ulysses may come into Troy, since he comes in hidden within the horse. The case is well-known, and in previous ambient systems this knowledge may be directly relevant for security. In our system, on the contrary, the danger represented by Ulysses is already clear from his permission to send general processes into the palace. Actually, with the to mechanism, the Greeks do not need the horse to set Troy on fire: they might merely send incendiary processes to Troy from the outside (the $\mathscr{E}$ component serves to protect Trojans exactly from these missiles, either coming from the outside or from the inside).

In the absence of open, the simple Ulysses' presence in Troy is not dangerous: if he is not allowed to send out processes, he can only take a harmless stroll in the city. However, the knowledge is desirable of which ambients may move where, and completely forbidding Ulysses the access to Troy is, after all, a good policy.

As a final remark, we observe that the very simplicity of our type system, which grants it an easy readability and usability, does not allow the control of finer properties, expressible through much more sophisticated types such as the ones by Pierce-Sangiorgi [12] for the π-calculus. Even the basic type system for D-π [7] is only incompletely rendered, since our system cannot encode the assignment of different types to homonymous channels belonging to different locations; this is made possible in D-π by the presence of local typing judgments, which cannot be simulated by our only global judgments.

References

[1] T. Amtoft, A. J. Kfoury, and S. M. Pericas-Geertsen. What are polymorphically-typed ambients? In D. Sands, editor, *ESOP '01*, volume 2028 of *LNCS*, pages 206–220, Berlin, 2001. Springer-Verlag.

[2] M. Bugliesi and G. Castagna. Secure safe ambients. In *POPL'01*, pages 222–235, New York, 2001. ACM Press.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In B. Pierce, editor, *TACS'01*, volume 2215 of *LNCS*, pages 38–63. Springer-Verlag, 2001.

[4] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility types for mobile ambients. In J. Wiederman, P. van Emde Boas, and M. Nielsen, editors, *ICALP'99*, volume 1644 of *LNCS*, pages 230–239, Berlin, 1999. Springer-Verlag.

[5] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155, Berlin, 1998. Springer-Verlag.

[6] M. Dezani-Ciancaglini and I. Salvo. Security types for safe mobile ambients. In H. Jifeng and M. Sato, editors, *ASIAN'00*, volume 1961 of *LNCS*, pages 215–236, Berlin, 2000. Springer-Verlag.

[7] M. Hennessy and J. Riely. Resource access control in systems of mobile agents (extended abstract). In U. Nestmann and B. Pierce, editors, *HLCL'98*, volume 16(3) of *ENTCS*, Amsterdam, 1998. Elsevier Science B. V.

[8] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *POPL'00*, pages 352–364, New York, 2000. ACM Press.

[9] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In L. Brim, P. Jančar, M. Křetínský, and A. Kučera, editors, *CONCUR'02*, volume 2421 of *LNCS*, pages 304–320, Berlin, 2002. Springer-Verlag.

[10] R. Milner. The polyadic π-calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series F: Computer and Systems Sciences*, pages 203–246, Berlin, 1993. Springer-Verlag.

[11] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[12] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[13] J. Wells. The essence of principal typings. In P. Widmayer, F. Triguero, R. Morales, M. Hennessy, S. Eidenbez, and R. Conejo, editors, *ICALP'02*, volume 2380 of *LNCS*, pages 913–925, Berlin, 2002. Springer-Verlag.