A Dependently Typed Ambient Calculus*

Cédric Lhoussaine and Vladimiro Sassone

University of Sussex, UK

Abstract. *The Ambient calculus is a successful model of distributed, mobile computation, and has been the vehicle of new ideas for resource access control. Mobility types have been used to enforce elementary access control policies, expressed indirectly via classification of ambients in groups by means of 'group types.' The paper presents a theory of dependent types for the Ambient calculus which allows greater flexibility, while keeping the complexity away from the programmer into the type system.*

1 Introduction

The work on types for *Mobile Ambients* [6] (MA), initiated in [7], has introduced some interesting innovations. In particular, it led to natural concepts of mobility types (cf., e.g. [4,3,12,8]). The common framework proposed so far in the literature for mobility control relies on *groups* and *group types*. Groups represent collections of ambients with uniform access rights, and policies like '*n CanMoveTo m*' are expressed by saying, for instance, '*n belongs to group G* and all ambients of *group G can move to m*.' These and similar simple constraints, such as '*m accepts ambients of group G*,' are then enforced statically by embodying groups in types. This approach's merit is to simplify considerably the type system. It allows to avoid the difficulties of dependent types, since it replaces (variable) ambient names with (constant) group names. The loss in expressiveness and flexibility, however, is self-evident.

This paper generalises the approaches to access control based on groups by using types explicitly dependent on ambient names, so that policies like '*n CanMoveTo m*' can be expressed directly. The task involves issues of technical complexity, and we believe that our approach contributes significantly to advance the theory of type-based resource access control in the Ambient calculus. The guiding principle behind our work is to allow for flexibility in control policies, while pushing the complexity to the type system. We elaborate further on this below.

Dependent types vs groups. At the formal level, dependent types are indeed trickier to work with than groups, as already suggested in [4]. However, they are very expressive and ultimately, at the programming level, not necessarily more difficult to use than groups. It may indeed be challenging, if not impossible, to

* Research supported by 'MyThS: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 171–187, 2004.

partition ambient names into groups in order to implement the desired security policy in complex applications. By referring directly to names, instead, a programmer avoids a level of indirection, and is less prone to policy specification errors. The type systems we propose here aim at harnessing the complexity of dependent types and relegating it to inner working of the type system, so as to keep the burden off the programmer, and consequently make programs more robust.

To illustrate the point, let us consider a naive and specialised taxi server whose intention is to provide a taxi service usable only by ambient n:

$$\mathit{TaxiServ}(n) = !(\nu\, \mathit{taxi}\colon A)\,(\mathit{taxi}[] \mid n[\mathsf{in}\ \mathit{taxi}]).$$

The process creates a new ambient called *taxi*, with some type A, together with an ambient n which can move to it. Due to the scoping rules, an external ambient $m[P]$ can move into the taxi by first moving to n and then escaping it after being transported to *taxi*. To avoid this, A must restrict the children allowed of *taxi* to, say, only ambient n. Using groups – for instance in the setting of [12], where an ambient type $a : \mathsf{amb}[G, \mathsf{mob}[\mathcal{G}]]$ assigns group G to a and specifies that $\mathcal{G}$ is the set of groups of those ambients which a is allowed to move to – to prevent any ambient other than n to occupy the taxi, n must be the *only* ambient with a type of the form $\mathsf{amb}[H, \mathsf{mob}[\{G\} \cup \mathcal{H}]]$, where G is the group of *taxi*. This condition is clearly beyond the control of the $\mathit{TaxiServ}(n)$ application, as the environment may create new names with types that violate it. Making G private is not a solution, as it forces n to be confined too, so defeating its very purpose. Indeed, in a term such as $(\nu\, G)\,(\nu\, n\colon A)\, P \mid Q$, with G in A, n can not be passed to Q (not even via subtyping [12]), as the type of any suitable receiving variable would contain G.

With dependent types we would simply and directly mention n as a possible (or perhaps the only) child for *taxi*. This leads to ambient types of the form $\mathsf{amb}[\mathsf{mob}[\mathcal{P}, \mathcal{C}]]$, where $\mathcal{P}$ is the set of names of those ambients allowed to contain *taxi* and, dually, $\mathcal{C}$ is the set of names of those ambients allowed to reside within *taxi*. Type A above can be simply $\mathsf{amb}[\mathsf{mob}[\{\mathit{top}\}, \{n\}]]$, where name *top* represents the top-level ambient. Yet, the *taxi* ambient would be guaranteed to have n and only n as a possible child, with no need for n to be private. Of course, such simplicity in expressing types comes at the cost of additional complexity at the type system level. Primarily, we need to enforce consistency between the type of a parent ambient and those of its children. To exemplify, when a taxi is created, n receives a new capability to have *taxi* as its parent; the type of n – which actually predates the creation of the fresh instance of *taxi* – must then be updated accordingly. Observe that in a conventional type system, the effect of the new name would be confined inside the scope of its definition, and would certainly not affect the type of n. In our case, however, leaving n's type unchanged leads to an unsound system (cf. Example 1).

In order for n to be aware of any additional capability it may acquire over time, we introduce the pivotal notion of *abstract names*. Such names are used only in the derivation of typing judgements, to record potential name creations, either by a process or by its environment. As it will be clear later, abstract

names intuitively play the role of group names in their ability to cross the scope of name restrictions.

Flexibility. The main contribution of this paper is the use of dynamic types to increase expressiveness, which can be fleshed out as the ability to deploy security policies at run time. More precisely, we want servers flexible enough to provide services specialised to each particular client, so leading towards secure, highly modular programming. In other words, a service specifically provided for client c_1 must be unusable by client c_2. Such objectives may be achieved in several different ways, as for instance by adding special operators to the calculus. In this paper we stick to the basic primitives of the Ambient calculus and investigate the issue of personalised services at the level of types.

The taxi server discussed above is not particularly interesting, because it is hardly a server at all: it does not interact with generic clients, but provides again and again the same 'secure' service to the same client. A brief reflection proves that if we were to design a 'true' server using groups, we would need to exchange group names, yielding soon a form of (group) dependent types. The type system we propose for dynamic types is a natural extension of the one we illustrated above, for mobility only. Namely, we simply require that communications of ambient names and variables be reflected in the types. For instance, the dynamic taxi server will have the form:

$$!(x)\,\mathit{TaxiServ}(x) = !(x)(\nu\, \mathit{taxi} : \mathsf{amb}[\mathsf{mob}[\{\mathit{top}\}, \{x\}]])\,(\mathit{taxi}[] \mid x[\mathsf{in}\ \mathit{taxi}]).$$

Communication increases the complexity of the type system, because the rules need to track information about receivers, too. It makes little sense to type communication via the usual exchange types, because receiving different names may lead to orthogonal behavioural and mobility properties. Subtyping does not help towards a unifying treatment (cf. Example 2). We approach the problem by tracking the names that may be received at a given variable. This enriches ambient types which, in addition to a mobility component, acquire a communication one. A typing assignment $n : \mathsf{amb}[\,\mathsf{mob}[\mathcal{P}, \mathcal{C}], \mathsf{com}[\mathcal{E}, \mathcal{L}]\,]$, where $\mathcal{E}$ and $\mathcal{L}$ are sets of ambient names, means that n may be communicated inside any ambient in $\mathcal{E}$, and dually, that any ambient in $\mathcal{L}$ may be communicated within n. It is important to remark that even if such components only carry finite sets, the volume of information exchanged is in general not bounded: the joint effect of name creation and the use of abstract names, allows to distribute new communication capabilities at run time, and communicate infinitely many names.

Plan. The paper proceeds as follows. Section 2 presents a simple type system with dependent types. We focus on mobility, that is the Ambient calculus with no communication at all. Then, in Section 3, we investigate dynamic types allowing communication of ambient names. For the sake of simplicity, we do not address the issue of communication of capabilities, even though this can easily be integrated in our approach.

2 A Simple Dependent Type System

In the paper we use the standard Ambient calculus with *asynchronous*, *monadic* communication. (We remark that our results do not depend on such choice.) Syntax and reduction semantics can be found in [6]. In this section we consider the pure Ambient calculus, that is the calculus restricted to the mobility features. Communication will be studied in Section 3.

2.1 Types and Definitions

The syntax of types is given in Figure 1 where N denotes an infinite set of *abstract ambient names* and the symbol $\star$ stands for the *self*-ambient, whose meaning is explained below. The type $\mathsf{cap}[\mathcal{P}]$ denotes the type of a capability which can be exercised within any ambient whose name occurs in $\mathcal{P}$. An ambient type has, for the moment, just one component: a mobility type. Ambient n with mobility type $\mathsf{mob}[\mathcal{P},\mathcal{C}]$ is allowed to be a sub-ambient of all ambients whose name occurs in $\mathcal{P}$; i.e. $\mathcal{P}$ is the set of *possible parents* of n. Moreover, an ambient is allowed to occur as sub-ambient of n if its name occurs in $\mathcal{C}$, the set of *possible children* of n. We define some useful notations on types:

$$\mathsf{amb}[M]^{\mathsf{mob}} = M \qquad \mathsf{mob}[\mathcal{P},\mathcal{C}]^{\uparrow} = \mathcal{P} \qquad \mathsf{mob}[\mathcal{P},\mathcal{C}]^{\downarrow} = \mathcal{C}$$

We use two kinds of typing contexts: *abstract contexts* (ranged over by $\Xi, \Theta, \dots$) and *concrete contexts* (ranged over by $\Gamma, \Delta, \dots$), which map respectively abstract and concrete names to ambient types. We use Π to denote either an abstract or a concrete context. We note by (Π, Π') the union of disjoint contexts of the same kind. According to the informal meaning of types, type assignments in concrete typing contexts are related to each other. Consider for instance

$$\Gamma = n : \mathsf{amb}[\,\mathsf{mob}[\mathcal{P},\mathcal{C}]\,],\ m : \mathsf{amb}[\,\mathsf{mob}[\emptyset,\{n\}]\,],$$

where the type of m allows it to have n as a child. Coherently, the type of n should allow n to have m as a parent, i.e. $m \in \mathcal{P}$. This can be expressed by the central notion of *context update* $\Gamma^{(n:A)}$, which updates Γ with respect to a fresh type assignment $n : A$:

$$\begin{aligned} (\Gamma,\Delta)^{(n:A)} &= \Gamma^{(n:A)}, \Delta^{(n:A)} \\ (m : \mathsf{amb}[M_0])^{(n:\mathsf{amb}[N])} &= m : \mathsf{amb}[M_1], \end{aligned} \tag{1}$$

where

$$M_1^{\uparrow} = \begin{cases} M_0^{\uparrow} \cup \{n\} \text{ if } m \in N^{\downarrow}, \\ M_0^{\uparrow} \qquad \text{otherwise;} \end{cases} \qquad \text{and} \qquad M_1^{\downarrow} = \begin{cases} M_0^{\downarrow} \cup \{n\} \text{ if } m \in N^{\uparrow}, \\ M_0^{\downarrow}. \end{cases}$$

Context update is pivotal in our type systems to express the typing of name creation. We define union (resp. inclusion and equality) of types as component-wise set union (resp. inclusion and equality); the notation $nm(A)$ (resp. $an(A)$) stands for the set of (abstract) names occurring in A. It is extended to all types and typing contexts.

Ambient (concrete) names	$n, m, \ldots \in \mathcal{N}$	*Abstract names*	$\mathsf{n}, \mathsf{m}, \ldots \in \mathsf{N}$
Capability types	$K ::= \mathsf{cap}[\mathcal{P}]$	*Mobility types*	$M, N ::= \mathsf{mob}[\mathcal{P}, \mathcal{C}]$
Ambient types	$A ::= \mathsf{amb}[M]$		where $\mathcal{P}, \mathcal{C} \subseteq \mathcal{N} \cup \mathsf{N} \cup \{\star\}$

Fig. 1. Types

$$\frac{A \in \mathit{Types}(\Gamma, \Theta, a)}{\Gamma \vdash^{\Theta} a : A}\ (\textsc{VAmb}) \qquad \frac{\Gamma \vdash^{\Theta} U : K, V : K}{\Gamma \vdash^{\Theta} U.V : K}\ (\textsc{VPfx})$$

$$\frac{\Gamma \vdash^{\Theta} a : A, a_i : A_i \quad A^{\mathsf{mob}\uparrow} \subseteq A_i^{\mathsf{mob}\uparrow} \quad 1 \le i \le n}{\Gamma \vdash^{\Theta} \mathsf{out}\, a : \mathsf{cap}[\{a_1, \ldots, a_n\}]}\ (\textsc{VOut}) \qquad \frac{\Gamma \vdash^{\Theta} a : A \qquad \mathcal{P} \subseteq A^{\mathsf{mob}\downarrow}}{\Gamma \vdash^{\Theta} \mathsf{in}\, a : \mathsf{cap}[\mathcal{P}]}\ (\textsc{VIn})$$

$$\frac{\Gamma \vdash^{\Theta} a : A, a_i : A_i \qquad A^{\mathsf{mob}} \subseteq A_i^{\mathsf{mob}} \qquad 1 \le i \le n}{\Gamma \vdash^{\Theta} \mathsf{open}\, a : \mathsf{cap}[\{a_1, \ldots, a_n\}]}\ (\textsc{VOpen})$$

$$\frac{\Gamma \vdash_a^{\Xi;\Theta} P \quad \Gamma \vdash^{\Xi,\Theta} a : A \quad b \in A^{\mathsf{mob}\uparrow}}{\Gamma \vdash_b^{\Xi;\Theta} a[P]}\ (\textsc{Amb}) \qquad \frac{}{\Gamma \vdash_a^{\emptyset;\Theta} \mathbf{0}}\ (\textsc{Nil}) \qquad \frac{\Gamma \vdash_a^{\Xi;\Theta,\Xi\sigma} P}{\Gamma \vdash_a^{\Xi;\Theta} !P}\ (\textsc{Rep})$$

$$\frac{\Gamma \vdash_a^{\Xi;\Theta} P \quad \Gamma \vdash^{\Theta,\Xi} V : \mathsf{cap}[a]}{\Gamma \vdash_a^{\Xi;\Theta} V.P}\ (\textsc{Pfx}) \qquad \frac{\Gamma \vdash_a^{\Xi_1;\Xi_2,\Theta} P \qquad \Gamma \vdash_a^{\Xi_2;\Xi_1,\Theta} Q}{\Gamma \vdash_a^{\Xi_1,\Xi_2;\Theta} P \mid Q}\ (\textsc{Par})$$

$$\frac{\Gamma^{(n:A)}, n : A\{n/\star\} \vdash_a^{\Xi\{n/\mathsf{n}\};\Theta} P}{\Gamma \vdash_a^{n:A,\Xi;\Theta} (\nu\, n : A)\, P}\ (\textsc{Res})$$

Fig. 2. Simple Type System

2.2 The Type System

Our type system deals with typing judgements for values and processes, that is

$$\Gamma \vdash^{\Theta} V : T \qquad \text{and} \qquad \Gamma \vdash_a^{\Xi;\Theta} P,$$

where T is either an ambient or a capability type. The first judgement reads as "V has type T with respect to the concrete context Γ and the abstract context Θ." In the second one, ambient name a stands for the ambient where P is running, which we call the *current location* of P. In other words, P is guaranteed to be well-typed if run in a. Abstract contexts Ξ and Θ are used to account for potential name creations: those arising from P are registered in the *local* abstract context Ξ, those performed by the environment appear in the *external* abstract context Θ. The role of such contexts is to "internalise" (a dynamic notion of) group names, which move from being part of the language to being a inner mechanism of the type system.

Example 1. Let

$$\mathit{TaxiServ}(n) = !(\nu\, \mathit{taxi}\colon A)\,(\mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}]), \quad \text{for} \quad A = \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}\}, \{n\}]\,].$$

Let $Q = n[\,\mathsf{in}\, n \mid m[\mathsf{out}\, n.\mathsf{out}\, n]\,]$. We study the system $P = \mathit{TaxiServ}(n) \mid Q$, which we assume running in some ambient named *top*. The server $\mathit{TaxiServ}(n)$ creates an ambient *taxi* whose type allows it to accept only n as a child. The scope (or "visibility") of *taxi* is $(\mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}])$. We assume that Q is an ambient n which may move in an ambient of the same name n and contains a child ambient m, willing to escape twice out n. Thus, m must be allowed to run at the same level of n, and a type system should ensure that any possible parent for n is also a possible one for m. This leads to an assignment like Γ below.

$$\Gamma = \left\{ \begin{array}{l} \mathit{top}\colon \mathsf{amb}[\,\mathsf{mob}[\emptyset, \{n, m\}]\,] \\ \quad n\colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n\}, \{n, m\}]\,] \\ \quad m\colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n\}, \emptyset]\,] \end{array} \right. \quad \Delta = \left\{ \begin{array}{l} \mathit{top}\colon \mathsf{amb}[\,\mathsf{mob}[\emptyset, \{n, m, \mathit{taxi}\}]\,] \\ \quad n\colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n, \mathit{taxi}\}, \{n, m\}]\,] \\ \quad m\colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n\}, \emptyset]\,], \\ \mathit{taxi}\colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}\}, \{n\}]\,] \end{array} \right.$$

In $(\mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}])$, ambient n gains access to *taxi* by means of name creation, viz. *taxi*, and the type assignment must evolve to Δ above (which actually is of the form $\Gamma^{(\mathit{taxi}:A)}, \mathit{taxi}\colon A$). However, running P may obviously lead to m becoming a child of *taxi*, thus violating the specification expressed by the type of *taxi*. So, P must be considered ill-typed. A naive approach would however try to typecheck $(\mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}])$ (against Δ) and Q (against Γ) independently, and therefore accept P. Indeed, from Q's point of view, *taxi* does not exist (it is bound in $\mathit{TaxiServ}(n)$), and Q behaves well. From the viewpoint of $\mathit{TaxiServ}(n)$, although *taxi* exists, the specification given by its type is respected: ambient n is allowed to move into *taxi*. Considering $\mathit{TaxiServ}(n)$ and Q as stand-alone processes prevents from realising that their interaction may lead to a breach of the intended access policy. □

Example 1 motivates the use of abstract contexts. They are a means to record the new capabilities which may potentially arise from name creations performed in external processes. Since such names are bound, they cannot be referred to by name in typing contexts: we use *abstract names* to represent them. The typechecking of process Q in Example 1 must then be carried out with an *external* abstract context $\Theta = \mathsf{taxi}\colon A$, representing the potential creation of a name of type A, and an empty *local* abstract context, as Q does not create names. In other words, we are led to prove $\Gamma \vdash_{top}^{\emptyset;\Theta} Q$. From Γ and Θ, we can deduce a more informative type for n, viz. its *global type* $\mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n, \mathsf{taxi}\}, \{n, m\}]\,]$, which states that n is allowed to run within *top*, n and some ambient taxi, to be created by its environment and whose actual name we do not known yet. (We stress that the name taxi is just a placeholder; the information that leads to the global type is carried by A.) Since m does not appear in Θ, its global type remains $\mathsf{amb}[\mathsf{mob}[\{\mathit{top}, n\}, \emptyset]]$, and we can conclude that Q is ill-typed. Indeed, there exists a possible parent of n which is not a possible one for m, namely taxi.

We define the *symmetric type* $\Pi[\alpha]$ of a name α with respect to a typing context Π, as $\mathsf{amb}[\mathsf{mob}[\mathcal{P},\mathcal{C}]]$, where $\mathcal{P} = \{\beta \mid \alpha \in \Pi(\beta)^{\mathsf{mob}^{\downarrow}}\}$ and $\mathcal{C} = \{\beta \mid \alpha \in \Pi(\beta)^{\mathsf{mob}^{\uparrow}}\}$.

Definition 1. The *global type* of a name α with respect to typing contexts Γ and Θ is defined as

$$gt(\Gamma;\Theta)(\alpha) = \begin{cases} \Gamma(\alpha) \cup \Theta[\alpha] & \text{if } \alpha \in \mathcal{N} \\ \Theta(\alpha)\{\alpha/_{\star}\} \cup \Theta[\alpha] & \text{if } \alpha \in \mathsf{N} \end{cases}$$

We define $Types(\Gamma,\Theta,a) = \{gt(\Gamma;\Theta)(a)\}$. (We use a singleton set here for uniformity with next section.) Turning back to the previous discussion, one can verify that indeed

$$gt(\Gamma;\Theta)(n) = \mathsf{amb}[\mathsf{mob}[\{top, n, \mathsf{taxi}\}, \{n, m\}]].$$

The notion of symmetric type leads to the definition of *symmetric typing context.*

Definition 2. A concrete typing context Γ is said *symmetric* if $\Gamma(n) = \Gamma[n]$, for all $n \in nm(\Gamma)$.

Symmetric typing context formalises the notion of "consistency" of a typing context. It ensures that a name n has a type allowing m as a child of n if and only if m itself has a type allowing n as a parent. It also guarantees that a typing context is "complete," i.e. that any name occurring in it has a type assignment. In the following we will assume all concrete typing contexts to be symmetric, and all abstract contexts to be "complete" for abstract names, that is $dom(\Theta) = an(\Theta)$.

Symmetric typing replicates parenthood information both in children and in parents types. Nevertheless, both components of mobility types are necessary to give sufficient access control in the presence of name creation. We remark that comparable mechanisms play in groups based approaches via the assignment of names to groups. Using the types of [12] to exemplify, recall that $\mathcal{G}$ in $n : \mathsf{amb}[G, \mathsf{mob}[\mathcal{G}]]$ is the set of groups of ambients allowed as parents of n. Therefore, assigning n to G to has the effect of determining the children allowed of n: namely, those ambients whose mobility component contains G.

Let us now comment on the rules of the Figure 2, starting with the judgements for values. We look at the rules from conclusions to premises. A judgement $\Gamma \vdash^{\Theta} U : T, V : T'$ stands for $\Gamma \vdash^{\Theta} U : T$ and $\Gamma \vdash^{\Theta} V : T'$. Axiom (Vamb) simply gives the global type of a name, while (VPfx) asserts that the prefix $U.V$ has the capability type K if both U and V have such type. In order to type a $\mathsf{out}\, a$ performed in some ambient a_i, (Vout) checks that the possible parents of a are allowed for a_i (for $i \in \{1,\ldots,n\}$). Rule (VIn) typechecks the capability of ambient b (a member of $\mathcal{P}$) to move to ambient a. This is allowed if b is a possible child of a. Finally, (VOpen) states that some ambient a_i can open a if both the allowed parents and children of a are allowed for a_i, $i \in \{1,\ldots,n\}$. This ensures that whatever happens to a_i because of the capabilities inherited from a via the open is compatible with its specification.

A process is always typed with respect to a current location. Process $a[P]$ is well-typed in ambient b (rule AMB) if b is allowed as a's parent and P is well-typed in a. Process $\mathbf{0}$ is always well-typed with local abstract context empty. Process $V.P$ is well-typed in a if P is well-typed in a and V is a capability that can be exercised in a. In order to type a parallel composition of processes (rule (PAR)), the local abstract context must be split in two parts: in the typing of each component one of these remains local, the other becomes external.

Rule (RES) deserves a close analysis. The typing of $(\nu\, n : A)\, P$ requires the local abstract context to assign type A to some abstract name (here n). The rule consumes such assignment, and P is then typed with its concrete typing context updated with n's typing. We apply the substitution $\{n/\star\}$ to A, where $\star$ is an "alias" for the self ambient (n here). This is necessary, since in the conclusion n cannot occur in A. Indeed, we make the usual assumption that a bound name does not occur in the typing contexts and in the current location (which here means $n \neq a$). Finally, the substitution $\{n/\mathsf{n}\}$ is applied to the local abstract context, as types in Ξ may refer to n. This would be the case of, e.g., the term

$$(\nu\, n : \mathsf{amb}[\,\mathsf{mob}[\{top\}, \emptyset]\,])\,(\nu\, m : \mathsf{amb}[\,\mathsf{mob}[\{n\}, \emptyset]\,])\, P$$

typed with the local abstract context $\mathsf{n} : \mathsf{amb}[\,\mathsf{mob}[\{top\}, \emptyset]\,],\ \mathsf{m} : \mathsf{amb}[\,\mathsf{mob}[\{\mathsf{n}\}, \emptyset]\,]$.

In Rule (REP) we assume that σ is an *abstract renaming*. By this we mean an injective substitution whose domain is the set of abstract names occurring in Ξ and whose codomain is a set of fresh abstract names. In other words, if σ is an abstract renaming, then $\Xi\sigma$ is a fresh copy of Ξ. Process $!P$ can be seen as an infinite parallel composition of copies of P. Therefore, typing $!P$ is equivalent to typing P in an environment which can provide new capabilities as P can. These are represented by the local abstract context of $!P$, which explains why we add a copy of the local abstract context to the external one. It is easy to prove that one copy is sufficient.

It is worth remarking that there are behavioural properties which are expressible with groups but not with this system of dependent types. This is because in the present system, name creation assigns types which may only refer to names already in the scope of the created name. For instance, in

$$(\nu\, n : A)\, P \mid (\nu\, m : A')\, Q$$

m (resp. n) can not occur in A (resp. A'). This means that n and m will not be able to interact. With group types one may use group names shared by m and n, i.e. whose scope contains both names. The extension of the type system to dynamic types introduced in the next section circumvents the problem by the usual mechanisms of scope extrusion in name passing calculi. Indeed, in order for m to give (and acquire) new capabilities to n, we send n to m, which then may refer in A' to the received name.

3 Dynamic Types

As illustrated in the previous section, name creation is itself a means to dynamically change ambient capabilities. In the type system, abstract names are used

$$\begin{array}{llll} \textit{Abstract variables} & \mathsf{x}, \mathsf{y}, \ldots \in \mathsf{X} & \textit{Capability types} & K ::= \mathsf{cap}[\mathcal{P}] \\ \textit{Mobility types} & M, N ::= \mathsf{mob}[\mathcal{P}, \mathcal{C}] & \textit{Communication types} & C ::= \mathsf{com}[\mathcal{E}, \mathcal{L}] \\ \textit{Ambient types} & A ::= \mathsf{amb}[M, C] & \textit{Variable types} & B ::= \mathsf{var}[\mathcal{B}] \end{array}$$

$$\text{where } \mathcal{P}, \mathcal{C} \subseteq \mathcal{N} \cup \mathsf{N} \cup \mathcal{X} \cup \mathsf{X} \cup \{\star\} \quad \text{and} \quad \mathcal{B}, \mathcal{E}, \mathcal{L} \subseteq \mathcal{N} \cup \mathsf{N} \cup \{\star\}$$

Fig. 3. New types for communication

to track the relevant changes outside the actual scope of the name. New names may refer directly in their types to known ambient names. For instance, the taxi server

$$(\nu\, \mathit{taxi} \colon \mathsf{amb}[\mathsf{mob}[\{\mathit{top}\}, \{n\}]])\,(\mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}])$$

creates a name *taxi* which may have n as a child. At the same time, the type of n is enriched with the capability to have *taxi* as a parent. This process, however, is restricted in that it provides a private taxi for one and only one ambient, namely n. By enabling communication of ambient names, types in name creation constructs may also refer to a received name, so boosting the dynamic aspects of the calculus. This allows, for instance, to write a *generic* taxi server which dynamically provides private taxis for any ambient whose name is received by a communication:

$$\mathit{DTaxiServ} = !(x)(\nu\, \mathit{taxi} \colon \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}\}, \{x\}]\,])\,(\mathit{taxi}[\,] \mid x[\mathsf{in}\, \mathit{taxi}]),$$

Such an increase in dynamic adaptation, and therefore in expressiveness, comes with a corresponding increase in the complexity of the typing system. There are two reasons:

- ▷ firstly, types may now refer to ambients indirectly, via variables;
- ▷ secondly, upon communication a variable may become instantiated by a name or by another, thus leading to different capabilities, that is different access control policies. Moreover, such policies are not necessarily comparable, which, for instance, makes subtyping useless.

Example 2. Let *DTaxiServ* be as above, and consider $P = \langle n \rangle \mid \langle m \rangle \mid \mathit{DTaxiServ}$. Process P may evolve to either

$$P_1 = \langle n \rangle \mid \mathit{taxi}[\,] \mid m[\mathsf{in}\, \mathit{taxi}] \mid \mathit{DTaxiServ} \quad \text{or} \quad P_2 = \langle m \rangle \mid \mathit{taxi}[\,] \mid n[\mathsf{in}\, \mathit{taxi}] \mid \mathit{DTaxiServ}$$

according to whether *DTaxiServ* receives m or n. As regards to the type assigned to *taxi*, in P_1 only m – that is the received name – is allowed to go inside *taxi*, whereas in P_2 only n is. Such situations are orthogonal and thus incomparable: in the first one, m acquires a new capability ('*CanMoveTo*' *taxi*) and n remains unchanged, whereas in the second it is the other way around. This means that the type system will have to deal with all possible communications. □

Example 2 suggests that an ambient name may have different possible types depending on the communications that may happen.

3.1 New Types and Definitions

The needed additional types are given in Figure 3, where X is an infinite set of *abstract variables*. Ambient types have now two components: one for mobility, which remains unchanged, and one for communication. Following our "symmetric style" for types, a communication type consists of two sets of names: an *external* one, $\mathcal{E}$, which describes the ambients where the name in object may be communicated, and a *local* one, $\mathcal{L}$, which describes the names which may be communicated within the object ambient. For instance, if n has type $\mathsf{amb}[M, \mathsf{com}[\{m\}, \{o\}]]$, then n can be exchanged inside m, and o can be communicated inside n; two terms which satisfy such specification are: $m[\langle n \rangle]$ and $n[\langle o \rangle]$.

A variable type is the set of ambients where a variable may be bound. For instance, in $n[(x)P]$ the variable x may have type $\mathsf{var}[\{n\}]$. Because we deal with an ambient calculus with the open capability, a variable may of course be bound in several places. Variable x in

$$n[\langle o_1 \rangle \mid \mathsf{open}\, m \mid m[\langle o_2 \rangle \mid (x)P]$$

may be bound in n or in m, depending on whether m is opened before x is bound or not. The type of x in the term above must therefore be $\mathsf{var}[\{m, n\}]$.

We remark that, contrary to usual type systems, but as in [1], we do not use the so-called *exchange* types, which build on the notion of "topic of conversation" permitted within a given ambient. This is a particular type (modulo subtyping) which all processes within that ambient agree to exchange. Indeed, suppose that n is an ambient whose exchange type is A, and m_1 and m_2 have type A. Then, the term

$$n[\langle m_1 \rangle \mid \langle m_2 \rangle \mid (x : A)P]$$

would be well-typed. However, if an external process gives a new capabilities to m_1 via name creation, then $\langle m_1 \rangle$ may become ill-typed. An instance of this is the following term.

$$(\nu\, o : \mathsf{amb}[\mathsf{mob}[\{m_1\}, \emptyset], C])\, Q \mid n[\langle m_1 \rangle \mid \langle m_2 \rangle \mid (x : A)P]$$

Here, $\langle m_2 \rangle$ is still well-typed, but $\langle m_1 \rangle$ is manifestly not. In fact, after scope extrusion, m_1 has not anymore type A, but A enriched with the capability to admit o as a child, which leads to a type error when trying to bind x to m_1. Our type system does not feature uniform exchange types, but it allows in principle to bind a variable to names which may have completely different types. This also leads to an increase in expressiveness.

Assuming the set of communicable ambient names to appear in types may at first appear to give a very restricted form of communication, where only finitely many names can be communicated in a given ambient. Due to the dynamic nature of types, this is actually not the case: name creation can give new communication capabilities to existing ambients. For instance, if ambient n has a communication type $\mathsf{com}[\emptyset, \{o\}]$ – that is, only o may be sent in n – the creation of name m with communication type $\mathsf{com}[\{n\}, \emptyset]$ gives n the new capability to

have m has communicable ambient. Combining name creation with replication leads to potentially infinite communicable names in a given ambient.

Finally, we remark that variable and communication types, contain only ambient names and no variables. This means that communication is forbidden inside ambients created using a received name, as in $(x)x[P]$. We embraced such restriction for the sake of simplicity: allowing variables in such types is possible, at the price of a more complex definition of the set of types (Definition 7) and of additional side conditions in the type system's rules.

We introduce notations for communication types corresponding to those given for mobility types in (1). We define $\mathsf{amb}[M, C]^{\mathsf{com}} = C$, and then $C^{\uparrow}$, $C^{\downarrow}$ and $(\Gamma, \Delta)^{(n:A)}$, where $(x : B)^{(n:A)} = x : B$, as obvious. The notion of *symmetric type* is extended to communication types straightforwardly.

No variable occurs in symmetric types, therefore the notion of symmetric typing context must be defined accordingly.

Definition 3. A concrete typing context Γ is *symmetric* if, for all $n \in nm(\Gamma)$, we have $\Gamma(n) - (\mathsf{X} \cup \mathcal{X}) = \Gamma[n]$.

Here $\Gamma(n) - (\mathsf{X} \cup \mathcal{X})$ denotes the type of n in Γ with all variables removed. Definition 1 of global types for ambient names remains unchanged; we complete it below for variables.

Definition 4. The *global type* of a variable α with respect to typing contexts Γ and Θ is defined by

$$gt(\Gamma; \Theta)(\alpha) = \begin{cases} \Gamma[\alpha] \cup \Theta[\alpha] & \text{if } \alpha \in \mathcal{X} \\ \Theta[\alpha] & \text{if } \alpha \in \mathsf{X} \end{cases}$$

The global type of a variable is the ambient type constructed from the capabilities the variable is given by typing contexts. In case of an abstract variable, only the abstract context provides the information, as no abstract variable occurs in the concrete context.

We finally define the *local type* of a name, as the usual notion of type provided by a typing context.

Definition 5. The *local type* of a name α with respect to typing contexts Γ and Θ is defined by

$$lt(\Gamma; \Theta)(\alpha) = \begin{cases} \Gamma(\alpha) & \text{if } \alpha \in dom(\Gamma) \\ \Theta(\alpha) & \text{if } \alpha \in dom(\Theta) \\ \text{undefined otherwise.} \end{cases}$$

For $B = \mathsf{var}[\mathcal{B}]$, we usually identify B and $\mathcal{B}$ writing, for instance, $\alpha \in B$ instead of $\alpha \in \mathcal{B}$.

$$\frac{\Gamma \vdash^{\Theta} a : A, a_i : A_i \qquad \begin{array}{c}\text{for any } \alpha,\ \text{if } lt(\Gamma;\Theta)(\alpha) = B \text{ then } a \in B \text{ iff } a_i \in B \\ A^{\mathsf{mob}} \subseteq A_i^{\mathsf{mob}} \qquad A^{\mathsf{com}} = A_i^{\mathsf{com}} \qquad \forall i \in \{1, \ldots n\}\end{array}}{\Gamma \vdash^{\Theta} \mathsf{open}\, a : \mathsf{cap}[\{a_1, \ldots, a_n\}]}\ (\textsc{VOpen'})$$

$$\frac{\Gamma, x : B \vdash_a^{\Xi\{x/\mathsf{x}\};\Theta} P \qquad a \in B}{\Gamma \vdash_a^{\mathsf{x}:B,\Xi;\Theta} (x)P}\ (\textsc{Input}) \qquad \frac{\Gamma \vdash^{\Theta} V : A \qquad a \in A^{\mathsf{com}\uparrow}}{\Gamma \vdash_a^{\emptyset;\Theta} \langle V \rangle}\ (\textsc{Ouput})$$

Fig. 4. Revised typing system and additional ones for communication

3.2 Revised Typing System

The type system for the calculus with communication of ambient names is obtained from Figure 2 adding rules (Input) and (Ouput) for communication in Figure 4, and replacing (VOpen) with (VOpen').

Rule (VOpen') has two new conditions. The first verifies that the communication type is the same in the opened ambient and in the enclosing one. The condition on the top line checks that variables may be bound in the opened ambient if and only if they may in the opening one. Rule (Input) is similar to (Res): we pick up an abstract variable from the local abstract context whose type contains the current location. We do not perform any context update, since such updates only concern ambient names. Finally, rule (Output) says that we can send the value V inside a if that is allowed by the type of V.

The most significant revision is in the definition of $\mathit{Types}(\Gamma, \Theta, a)$, which is not anymore just a singleton, since names may now have several types. This means that any typing derivation with an axiom (VAmb) as a premise has to be read as: "for all types A such that $\Gamma \vdash^{\Theta} a : A$." Before we give the formal definition of $\mathit{Types}(\Gamma, \Theta, a)$, let us focus on an example to gather some intuition.

Example 3. Let us consider a non-replicated version of the taxi server of Example 2, but with the new communication types:

$$\mathit{DTaxiServ}_1 = (x)(\nu\, \mathit{taxi} : \mathsf{amb}[M, C])\,(\mathit{taxi}[] \mid x[\mathsf{in}\, \mathit{taxi}]),$$

with $M = \mathsf{mob}[\{\mathit{top}\}, \{x\}]$ and $C = \mathsf{com}[\emptyset, \emptyset]$. Let $P = \langle n \rangle \mid \langle m \rangle \mid \mathit{DTaxiServ}_1$, and suppose to work with the following typing context which assigns types to the free ambient names:

$$\Gamma = \begin{cases} n : A_n, \\ m : A_m, \\ \mathit{top} : \mathsf{amb}[\,\mathsf{mob}[\emptyset, \{m, n\}], \mathsf{com}[\emptyset, \{m, n\}]\,], \end{cases}$$

where

$$A_n = \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}\}, \{m\}], \mathsf{com}[\{\mathit{top}\}, \emptyset]\,]$$
$$A_m = \mathsf{amb}[\,\mathsf{mob}[\{\mathit{top}, n\}, \emptyset], \mathsf{com}[\{\mathit{top}\}, \emptyset]\,].$$

Here, n and m can be communicated in top, and have top as a parent. Moreover, n may have m as child and, consequently, m may have n as parent. One can verify

that Γ is symmetric. In order to type $DTaxiServ_1$ we need an abstract context which assigns variable type $\mathsf{var}[\{top\}]$ to an abstract variable, say x, because x may be bound in *top*. Also, it has to assign an ambient type A to an abstract ambient name, say taxi. Type A has to match $\mathsf{amb}[M, C]$, but we cannot directly use the latter, since M contains x which is initially bound. However, we can refer to the abstract name x, meant to correspond to x. Therefore the abstract context is

$$\Xi = \mathsf{x} : \mathsf{var}[\{top\}], \mathsf{taxi} : \mathsf{amb}[N, C],$$

for $N = \mathsf{mob}[\{top\}, \{\mathsf{x}\}]$. It is easy to check that the proof of $\Gamma \vdash_{top}^{\Xi;\emptyset} DTaxiServ_1$ leads to the judgement $\Delta \vdash_{top}^{\emptyset;\emptyset} (taxi[] \mid x[\mathsf{in}\ taxi])$, by application of the rules (Input) and (Res), where

$$\Delta = \begin{cases} n : A_n, \\ m : A_m, \\ top : \mathsf{amb}[\,\mathsf{mob}[\emptyset, \{m, n, taxi\}], \mathsf{com}[\emptyset, \{m, n\}]\,], \\ x : \mathsf{var}[\{top\}], \\ taxi : \mathsf{amb}[\,\mathsf{mob}[\{top\}, \{x\}], \mathsf{com}[\emptyset, \emptyset]\,]. \end{cases}$$

In $(taxi[] \mid x[taxi])$, x is an ambient whose global *ambient* type is

$$A_x = \Delta[x] = \mathsf{amb}[\,\mathsf{mob}[\{taxi\}, \emptyset], \mathsf{com}[\emptyset, \emptyset]\,].$$

This is actually the least type assignable to x, since x is intended to be instantiated by an ambient name. Indeed, x may be bound to n or m, which means that the global types it may possibly assume include

$$A_x \cup A_n = \mathsf{amb}[\,\mathsf{mob}[\{top, taxi\}, \{m\}], \mathsf{com}[\{top\}, \emptyset]\,],$$

if n is received for x, and

$$A_x \cup A_m = \mathsf{amb}[\,\mathsf{mob}[\{top, taxi, n\}, \emptyset], \mathsf{com}[\{top\}, \emptyset]\,],$$

if m is received for x. Type $A_x \cup A_n$ is possible for n in the scope of the restriction. Indeed, if n is received, then the newly created ambient *taxi* gives to n – the received name – the capability to be its child. Outside the scope of the restriction, a possible type for n is rather

$$\Xi[\mathsf{x}] \cup A_n = \mathsf{amb}[\,\mathsf{mob}[\{top, \mathsf{taxi}\}, \{m\}], \mathsf{com}[\{top\}, \emptyset]\,].$$

Since *taxi* is not visible, we have to refer to its abstract representative taxi. Inspecting the various possibilities, we are therefore led to consider the following sets of possible ambient types.
(1) Outside the scope of *taxi*:

$$Types(\Gamma, \Xi, n) = \{\, A_n, A_n \cup \Xi[\mathsf{x}] \,\} \quad \text{and} \quad Types(\Gamma, \Xi, m) = \{A_m, A_m \cup \Xi[\mathsf{x}]\,\};$$

(2) Inside the scope of *taxi*:

$$Types(\Delta, \emptyset, n) = \{\, A_n, A_n \cup \Delta[x] \,\}, \qquad Types(\Delta, \emptyset, m) = \{\, A_m, A_m \cup \Delta[x] \,\}$$

$$\text{and} \quad Types(\Delta, \emptyset, x) = \{\, A_x \cup A_n, A_x \cup A_m \,\}. \square$$

3.3 Estimating the Set of Possible Types

In this section we give the formal estimate of the set of possible types for name a with respect to chosen typing contexts. As illustrated by Example 3, the possible types of an ambient name n are given by its global type merged with those of the variables n might instantiate. Consider the term

$$R = !\langle n\rangle \mid \langle m\rangle \mid (x)P \mid (y)Q,$$

and suppose that n and m have respectively global types A_n and A_m. Then, n may instantiate variables x and y with global types, say, A_x and A_y respectively. The possible types of n are, therefore, A_n, $A_n \cup A_x$, $A_n \cup A_y$ and $A_n \cup A_x \cup A_y$. The latter corresponds to the case in which n instantiates both x and y. The possible types of m are A_m, $A_m \cup A_x$, $A_m \cup A_y$, and $A_m \cup A_x \cup A_y$. (Observe that the latter is actually impossible, since m may instantiate at most one variable; it is an approximation error introduced by our estimation of possible types.) Symmetrically, the possible types for x are $A_x \cup A_n$, $A_x \cup A_m$, $A_x \cup A_y \cup A_n$ and $A_x \cup A_y \cup A_m$.

We first define the set of names which a given name may possibly instantiate or be instantiated by. To this aim we use binary relations over names. Intuitively, if a name α can instantiate a variable β, then the pair (α, β) is in one such a relation. We focus exclusively on so-called binding relations, which only depend on types and typing contexts. Basically, this amounts to saying that α is related to β if β is a variable which might be bound in some ambient where α may be communicated, as formalised by the definition below.

Definition 6. The *binding relation* with respect to typing contexts Γ and Θ is

$$\mathit{Bind}(\Gamma;\Theta) = \{\, (\alpha,\beta) \mid \mathit{gt}(\Gamma;\Theta)(\alpha) = A \text{ and } \mathit{gt}(\Gamma;\Theta)(\beta) = B \text{ with } A^{\mathsf{com}\uparrow}\{\alpha/\star\} \cap B \neq \emptyset \,\}.$$

Observe that $\mathit{Bind}(\Gamma,\Theta)$ is actually a subset of $(\mathsf{N} \cup \mathcal{N}) \times (\mathsf{X} \cup \mathcal{X})$. Indeed, due to our restrictions, the global (ambient) type of a variable is always empty – more precisely, it is always $\mathsf{com}[\emptyset,\emptyset]$. It follows that a variable cannot be related to another variable.

Relying on binding relations, we can estimate which variables may be instantiated by a given ambient name, and which ambient names may instantiate a given variable.

$$\mathit{Vars}(\Gamma;\Theta;\alpha) = \{\, \beta \mid (\alpha,\beta) \in \mathit{Bind}(\Gamma;\Theta) \,\};$$

$$\mathit{Ambs}(\Gamma;\Theta;\alpha) = \{\, \beta \mid (\beta,\alpha) \in \mathit{Bind}(\Gamma;\Theta) \,\}.$$

We use the notation $A \triangleright \mathcal{T}$, for $\mathcal{T}$ a set of ambient types, to denote the set $\{A \cup B \mid B \in \mathcal{T}\}$, and $A \triangleright^\star \mathcal{T}$ for $\{A\} \cup A \triangleright \mathcal{T}$.

Definition 7. The set $\mathit{Types}(\Gamma;\Theta;\alpha)$ of possible types for α with respect to typing contexts Γ and Θ is defined by

$$\mathit{Types}(\Gamma;\Theta;\alpha) = \mathit{gt}(\Gamma;\Theta)(\alpha) \triangleright^\star \left\{\, \mathit{gt}(\Gamma;\Theta)(\beta) \;\middle|\; \beta \in \mathit{Vars}(\Gamma;\Theta;\alpha) \,\right\}, \qquad \text{if } \alpha \in \mathsf{N} \cup \mathcal{N}, \text{ and}$$

$$\mathit{Types}(\Gamma;\Theta;\alpha) = \bigcup_{\beta \in \mathit{Ambs}(\Gamma;\Theta;\alpha)} \mathit{gt}(\Gamma;\Theta)(\alpha) \triangleright \mathit{Types}(\Gamma;\Theta;\beta), \qquad \text{if } \alpha \in \mathsf{X} \cup \mathcal{X}.$$

Observe that the constraints on our typing rules are expressed essentially as type inclusion constraints. Therefore, we do not really need to consider all the possible types of an ambient name. In practice, as well as in proofs, it is more convenient the use notions of maximum and minimum types which are respectively the union and the intersection of all possible types. Due to space limitation, we leave to the reader to reformulate our rules in such terms. For instance, rule (VAMB) can be removed, whilst (VIN) is rewritten by replacing A (standing for all possible types for a) by the minimum type for a.

The main result of this paper is a 'Subject Reduction' theorem for the type systems we introduced. Write $\Gamma \vdash_a P$ if there exist some abstract contexts Ξ and Θ such that $\Gamma \vdash_a^{\Xi;\Theta} P$, it can be expressed as follows.

Theorem 1 (Subject Reduction). If $\Gamma \vdash_n P$ and $P \to Q$, then $\Gamma \vdash_n Q$.

4 Conclusion, Related and Future Work

The paper proposed a type system to describe access control policies and related behavioural properties of processes in the Ambient calculus. Whilst all type systems in the literature make use of groups to describe such properties [4,12,8], ours is based on dependent types. Despite an additional complexity in building typing derivations, our system has simple, natural types which may make policy specifications easier, and, therefore, simplify the programming task.

The major technical device of our approach is the novel notion of *abstract names*, which mimic the role of groups internally to typing derivations. More precisely, they keep track of – and dynamically embody in an ambient's type – any new capability that the ambient may gain during its lifespan. Abstract names are, in a sense, a dynamic notion of group, made internal to the type system rather than part of the language.

We showed how to extend a basic system to deal with communication of ambient names. The resulting type system is, we believe, the main contribution of this paper. It allows modular, per-client programming and is, to the best of our knowledge, the first one dealing at such a level with dynamic specifications of security policies in the ambient calculus. Communication types come together with a noticeable increase of complexity of typing derivations in the system. Indeed, when typing a communication, one has to consider all ambient names that can possibly instantiate a given variable. However, only a finite number of names are necessary for the proofs, and the notions of maximum and minimum types simplify the matter considerably.

We plan to study in future work the question of decidability of type checking, as a positive answer would make our type system usable in practice. We conjecture that it should be possible to devise a type checking algorithm, for the following reasons.

- ▷ The use of abstract contexts is not a concern: for a typeable term P, it is easy to provide (algorithmically or "by hand"), an abstract context Θ, such

that $\Gamma \vdash_n^{\Theta;\emptyset} P$ for some Γ and n. Indeed, Θ is essentially the collection of bound names in P.

▷ Communication is quite hard to manage "manually" because it requires the synthesis of several global types. However, given variable types and communication types, it does not seem difficult to design an algorithm that does the job.

Because processes are typed against an environment – represented by the external abstract contexts – our type system is not compositional, in that $\Gamma \vdash_a P$ and $\Gamma \vdash_a Q$ does not imply $\Gamma \vdash_a P \mid Q$. This may be problematic in the framework of open (or partially typed) systems which have to deal with the arrival of unknown agents. Thus, the following questions arises: if $\Gamma \vdash_n^{\Xi;\emptyset} P$ is provable, does a class exist of external abstract contexts Θ such that $\Gamma \vdash_n^{\Xi;\Theta} P$? Such a class would determine the environments which make P typeable. Another question concerns the design of a type inference algorithm, which is a crucial component in open systems, where information coming from external sources cannot always be trusted. As studied in [11] for $D\pi$, dependent types call for a very accurate treatment, which would certainly be required for our type system as well. Such challenging questions are topics of our current and future research.

Related work. Beside the already mentioned work on groups [4,12,8], which provided direct inspiration for our research, related work includes [14,10]. These papers introduce dynamic and dependent types for a distributed π calculus, and study their impact on behavioural semantics. A difficulty arises in [14] with referring to bound names created in external processes, at all analogous to the one we tackled here with the introduction of *abstract names*. As a matter of fact, it is pointed out in *loc. cit.* as the main limitation of their work. The problem has been addressed in [13], completely independently of our work, using existential types. There appear to be analogies between Yoshida's existential types and our approach, although at this stage it is difficult to assess them in the details.

Since our type system works linearly on internal abstract contexts, and classically on external ones, the closest match appears to be with Yoshida's linear type discipline. However, while we conjecture we can reformulate local abstract names with (some form of) existential types, it looks as though they cannot provide a notion corresponding to our external abstract contexts. But such contexts are the keystones of our work. They allow, in particular, a correct treatment of parallel composition and replication (cf. Figure 2), where it is not always the case that if P is correct, so is $P \mid P$ or, a fortiori, $!P$. We plan to investigate this matter further.

Concerning typing systems for the ambient calculus, [1] uses dependent types for communication in order to achieve advanced type polymorphism of the sort usually encountered in lambda calculi. Types in *loc. cit.* track – like ours – all messages exchanged, and not rely on topics of conversation. They are also used to bound the nesting of ambients.

We conclude by observing that, as pointed out in [5] there are intriguing connections between groups and channels and binders of the flow analysis, as in [2,9]. Indeed, our approach has a lot to share with control flow analysis, and we believe our work can shed further light on such connections.

References

1. T. Amtoft and J. Wells. Mobile processes with dependent communication types and singleton types for names and capabilities. Technical Report 2002-3, Kansas State University, 2002.
2. C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Static analysis for the π calculus with applications to security. *Information and Computation*, 168:68–92, 2001.
3. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01*, volume 2215 of *Lecture Notes in Computer Science*, pages 38–63. Springer, 2001.
4. L. Cardelli, G. Ghelli, and A. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2000.
5. L. Cardelli, G. Ghelli, and A. Gordon. Secrecy and group creation. In *CONCUR'00*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2000.
6. L. Cardelli and A. Gordon. Mobile ambients. In *FOSSACS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.
7. L. Cardelli and A. Gordon. Types for mobile ambients. In *POPL'99*, pages 79–92. ACM Press, 1999.
8. M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. M3: Mobility types for mobile processes in mobile ambients. In *CAT'03*, volume 78 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.
9. P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proceedins of ASIAN'00*, volume 1961 of *LNCS*, pages 199–214. Springer-Verlag, 2000.
10. M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *FOSSACS'03*, Lecture Notes in Computer Science. Springer, 2003.
11. C. Lhoussaine. Type inference for a distributed π-calculus. In *ESOP'03*, volume 2618 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2003.
12. M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *CONCUR'02*, volume 2421 of *Lecture Notes in Computer Science*, pages 304–320. Springer, 2002.
13. N. Yoshida. Channel dependent types for higher-order mobile processes. In *POPL'04*, 2004. To appear.
14. N. Yoshida and M. Hennessy. Assigning types to processes. In *LICS'00*, pages 334–345, 2000.