Subtyping and Typing Algorithms for Mobile Ambients

Pascal Zimmer

Ecole Normale Supérieure de Lyon
Pascal.Zimmer@ens-lyon.fr

Abstract. The ambient calculus was designed to model mobile processes and study their properties. A first type system was proposed by Cardelli-Gordon-Ghelli to prevent run-time faults. We extend it by introducing subtyping and present a type-checking algorithm which returns a minimal type relatively to this system. By the way, we also add two new constructs to the language. Finally, we remove the type annotations from the syntax and give a type-inference algorithm for the original type system.

1 Introduction

With the growing development of the World-Wide-Web, it becomes interesting and fruitful to investigate the problems and properties of mobile code. The *ambient calculus* was designed to model within a single framework both *mobile computing*, that is to say computation in mobile devices like a laptop, and *mobile computation*, that is to say mobile code moving between different devices, like applets or agents. It also shows how the notions of administrative domains, their crossing, firewalls, authorizations... can be formalized in a calculus. In this sense, it is more appropriate than the π-calculus ([Mil91]), even if the bases are the same (for more discussion about the problems raised by mobility and computation over wide-area networks, see [Car99a, Car99b]).

Informally, an ambient is a bounded place, with an inside and an outside, where computation happens. Many ambients can be nested so that they form a hierarchy. Each of them has a name (not necessarily distinct from other ambient names), which will be used to control access. An ambient can be moved as a whole with all the computations and subambients it contains: it can enter another ambient or exit it. It can also be opened so that its contents get visible at the current level. For more intuitions motivating the ambient calculus or its graphical vision (the *folder calculus*), we recommend reading [CG97, CG98, Car99a].

In order to prove some specific properties concerning mobility, locking, communication... in the ambient calculus, Cardelli and Gordon proposed a simple type system, nondeterministic and without subtyping (see [CG99, CGG99]): some simple valid processes like $\langle 1 \rangle \mid (x : Real).P$ were not typable. The aim of our work was to introduce a subtyping relation, deduce a typing algorithm and, in doing this, to make the type system more suitable for a treatment of mobile ambients as a "programming language for mobility" (departing thus from the "ambients as a specific language for reasoning on mobility" approach).

The rest of the paper is organized as follows. In Section 2, we will review briefly the ambient calculus and its semantics, by the way adding two new constructs to the original syntax. Then, in Section 3, we present our extension of the type system, introduce a subtyping ordering, and define a new typing system. In Section 4, we show that the set of derivable types for a term has got a minimum, and we give an algorithm to compute it efficiently. It is now possible to type-check a term of the ambient calculus automatically. The next step in Section 5 is to remove all type annotations from the term and try to find a type-inference algorithm. We give a complete solution for the original type system without subtyping.

This paper is a shortened version of an internship report [Zim99], which contains more explanations, further developments and all the proofs of the theorems enunciated in this paper.

2 Mobile Ambients: Syntax and Semantics

In this Section, we are going to briefly review the polyadic ambient calculus we will use throughout this paper. We will try to explain its main constructs and rules, but we recommend reading [CG98] (or any other paper presenting the calculus extensively) to have a more complete presentation. By the way, we are also going to extend the original calculus and introduce two new constructs.

The polyadic ambient calculus is mainly composed of processes. As in many other process calculi, we have an inactive process $\mathbf{0}$ which does nothing, we can compose processes in parallel ($P \mid Q$), we have a construct to replicate a process as many times as necessary ($!P$) and we have a restriction operator ($(\nu n)P$) which introduces a new name n and restricts its scope to the inside of P. In the π-calculus, those names represented channels; here they represent ambient names. In our calculus, we also declare the type of this ambient name ($(\nu n : Amb^{Y}[T_n, T'_n])P$), but we will not care about that until the next Section.

An ambient is composed of a name n and a process P which is running inside the ambient. We write it $n[P]$. Here we extend the syntax of the polyadic ambient calculus with a new construction. Up to now, the locking-unlocking of an ambient was defined only in the declaration of its name. So ambients with the same name had all the same locking annotation. We kept this possibility (extending it with ordering), but we changed the syntax of ambient constructions: $n[P]$ is an unlocked ambient and $n[\![P]\!]$ is a locked one. So we can now have an explicit construct which guarantees that an ambient will be locked. This can seem redundant with the locking annotation in the type declaration of n, but from a programming point of view, it just appears to be more flexible.

The process $M.P$ executes the action induced by the capability M and then continues with the process P. There are three kinds of capabilities: one to enter an other ambient (*in* n), one to exit (*out* n) and one to open up an ambient (*open* n). To build such a capability, a process must know the name n. It can also have received the capability via communication (see below). Implicitly, it is impossible to reconstruct the ambient name n out of one or all of these capabilities (this is important to prove the security of a firewall for example). (In the original calculus, we could also compose capabilities into paths $(M.M')$ where ε was the empty path; we do not use these constructs in this paper, but they are not difficult to handle; see [Zim99] for details.)

The use of these three capabilities is given by the following reduction rules:

$$
\begin{array}{ll}
n[\mathit{in}\ m.P \mid Q] \mid m[R] \ \rightarrow\ m[n[P \mid Q] \mid R] & \text{(Red In)}\\
m[n[\mathit{out}\ m.P \mid Q] \mid R] \ \rightarrow\ n[P \mid Q] \mid m[R] & \text{(Red Out)}\\
\mathit{open}\ n.P \mid n[Q] \ \rightarrow\ P \mid Q & \text{(Red Open)}
\end{array}
$$

with the convention that in (Red In) and (Red Out), each occurrence of $[.]$ can be replaced by $[\![.]\!]$ (the ambients can be locked or unlocked), whereas in (Red Open), the ambient n must be unlocked.

Here we add our second extension: the *imm* capability. A process containing it must be immobile. We added it, because also for immobility we want a language construct which obliges a process to be immobile, instead of delegating it to the types (an other reason is that, without it, no construct would introduce the mobility annotation $\veebar$ in our type system). The corresponding reduction rule

$$\mathit{imm}.P \ \rightarrow\ P \qquad \text{(Red Imm)}$$

is very simple and actually "ignores" the *imm* capability. In fact, *imm* is only useful when typing: if $\mathit{imm}.P$ is typable, then P cannot contain moving capabilities (even by receiving them). At run-time, only this guarantee is important and we can throw *imm* away.

Finally, we have two communication primitives: $(n_1 : W_1, \ldots, n_k : W_k).P$ and $\langle M_1 \times \cdots \times M_k\rangle$. The first waits for a tuple of values of respective types $W_1, \ldots, W_k$, and binds them to the variables $n_1, \ldots, n_k$ in the continuing process P. The second outputs a tuple of values. Note that the output is asynchronous (no continuing process). The corresponding reduction rule is:

$$
\begin{array}{ll}
(n_1 : W_1, \ldots, n_k : W_k).P \mid \langle M_1, \ldots, M_k\rangle & \\
\quad \rightarrow\ P\{n_1 \leftarrow M_1, \ldots, n_k \leftarrow M_k\} & \text{(Red Comm)}
\end{array}
$$

The five last rules just say that reduction (i.e. computation) can also occur beyond scope restrictions, inside ambients, or by using the structural congruence rewriting (which we will not detail here):

$$
\begin{array}{ll}
P \rightarrow Q \Rightarrow (\nu n : W)P \rightarrow (\nu n : W)Q & \text{(Red Res)}\\
P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q] & \text{(Red Amb}_\circ)\\
P \rightarrow Q \Rightarrow n[\![P]\!] \rightarrow n[\![Q]\!] & \text{(Red Amb}_\bullet)\\
P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R & \text{(Red Par)}\\
P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q' & \text{(Red} \equiv)
\end{array}
$$

Here is the complete syntax of our calculus. Note that it allows strange expressions like $(\mathit{in}\ n)[P]$ or $\mathit{out}\ (\mathit{out}\ n).P$. Rejecting those nonsense terms will be an automatic property of the type system.

$P, Q ::=$ processes		$M ::=$ expressions	
$(\nu n : Amb^Y[T, T'])P$	restriction	n	name
$\mathbf{0}$	inactivity	$in\ M$	can enter into M
$P \mid Q$	composition	$out\ M$	can exit out of M
$!P$	replication	$open\ M$	can open M
$M[P]$	unlocked ambient	imm	immobility
$M[\![P]\!]$	locked ambient		
$M.P$	action		
$(n_1 : W_1, \ldots, n_k : W_k).P$	input		
$\langle M_1, \ldots, M_k \rangle$	async output		

Terms are also identified up to the consistent renaming of bound variables, in the restriction and input constructs. Thus, we can always suppose that all the ambient names and input variables are distinct.

3 A Type System with Subtyping

In order to verify some properties of processes in the ambient calculus, Cardelli and Gordon proposed a first type system in [CG99] and extended it with Ghelli in [CGG99]. It assured that a well-typed process could not cause certain kinds of run-time fault: exchanging values of the wrong type, moving or opening an ambient if it was not allowed to... We will always refer to this type system as "CGG".

In this Section, we will describe a new type system, extending (in some way) CGG. We will first introduce some new types and define an ordering relation on them (subtyping is essential to be able to write a typing algorithm). Then, we will give new typing rules and show some properties.

3.1 Type Definitions

We start by giving all the definitions of our types:

$Z ::=$	mobility annotations	$Y ::=$	locking annotations
$\perp_{\rightarrow}$	mobility unknown	$\perp_{\circ}$	locking bottom
$\veebar$	immobile	$\bullet$	locked
$\curvearrowright$	mobile	$\circ$	unlocked
$\top^{\rightarrow}$	mobile and immobile	$\top^{\circ}$	locking top

$O, I ::=$	input/output types	$T ::=$	process type
$\perp$	bottom value	${}^Z O \rightsquigarrow I$	
$W_1 \times \cdots \times W_k \quad (k \geq 0)$	tuple		
$\top$	top value		

$W ::=$ message types	
$Amb^Y[T, T']$ with $T \leq T'$ (see below)	ambient name
$Cap[T]$	capability

3.2 Intuitive Meanings and Ordering

The main intuition in defining the order is to always respect the subsumption rule: if P has got type T and $T \leq T'$, then P has also type T'. There are a few changes in the syntax of types compared to those of CGG. They were motivated by the introduction of subtyping: what seems "intuitive" is not always correct (this problem appeared clearly in the referee reports for a draft of this paper: whereas one referee found "may be mobile < immobile, may be opened < locked" as the "implicit" relation in CGG, an other one stated that "immobile < mobile and locked < unlocked" was the "obvious subtyping of CGG").

Mobility Annotations What is the "obvious" subtyping of CGG ? It depends on the point of view. If we consider that an immobile process can generate movements, we should define $\veebar < \curvearrowright$. For example, everybody would say that the process $\mathbf{0}$ is immobile, but in order to type $in\ n.\mathbf{0}$, we should also be able to say that $\mathbf{0}$ is mobile. On the other hand, with this definition, if we restrict a process to stay immobile, it can always remove this restriction with the subsumption rule, which speaks more in favour of $\curvearrowright < \veebar$.

The subtyping relation we need depends on the property we privilege: generation of movement or restriction of immobile processes. If we want to keep both results, we have to introduce a new symbol $\perp_{\!\rightarrow}$, and keep $\curvearrowright$ and $\veebar$ incomparable. In order to have a complete lattice, we introduce also $\top^{\!\rightarrow}$ (which will also be useful in the typing algorithm and for the ambient types), and we define $\perp_{\!\rightarrow} < \veebar, \curvearrowright < \top^{\!\rightarrow}$. Since this structure is a complete lattice (4 points with a lozenge-like ordering), there is no problem to define meet and join operations on it.

Process Type In the type system of CGG, there was only one single term representing the type of values exchanged in the ambient (Shh or a tuple). In presence of subtyping, we should now accept that outputs and inputs have different types. For example, the output of the integer 1 should be accepted by an input variable of type $Real$. So we decided to track the types of output and input values exchanged in the ambient. If a process is valid, it must then have type ${}^{Z}O \rightsquigarrow I$ with $O \leq I$ to ensure that any output can be read by any input instruction (note that this is not specified in the syntax of T, but will be a property of the type system; see also below). Later, it appeared that Yoshida and Hennessy used the same approach for a higher-order π-calculus with subtyping ([YH99]). Moreover, a valid process should not contain both mobile and immobile instructions; consequently the mobility annotation $\top^{\!\rightarrow}$ is forbidden for processes.

Definition 1 (Validity). *A process type* ${}^{Z}O \rightsquigarrow I$ *is said to be* valid *if* $Z \neq \top^{\!\rightarrow}$ *and* $O \leq I$.

For the process type, we define :

$$ {}^{Z}O \rightsquigarrow I \leq {}^{Z'}O' \rightsquigarrow I' \iff \begin{cases} Z \leq Z' \\ O \leq O' \\ I \geq I' \end{cases} $$

$$
\begin{aligned}
{}^{Z}O \rightsquigarrow I \wedge {}^{Z'}O' \rightsquigarrow I' \;&\triangleq\; {}^{Z \wedge Z'}O \wedge O' \rightsquigarrow I \vee I' \\
{}^{Z}O \rightsquigarrow I \vee {}^{Z'}O' \rightsquigarrow I' \;&\triangleq\; {}^{Z \vee Z'}O \vee O' \rightsquigarrow I \wedge I'
\end{aligned}
$$

The set of all process types has a complete lattice structure, with ${}^{\perp\!\rightarrow}\perp \rightsquigarrow \top$ as the minimal element and ${}^{\top\!\rightarrow}\top \rightsquigarrow \perp$ as the maximal one. But, if we consider only valid processes, there are many maximal types: all ${}^{\curvearrowright}O \rightsquigarrow O$ and ${}^{\veebar}O \rightsquigarrow O$ for every input/output type O.

Input/Output Types Then, we have to define what are output and input types. As before, it can be a tuple. But we had to replace Shh by two different values, one for outputs ($Shh_{out} = \perp$) and one for inputs ($Shh_{in} = \top$). Then it appeared useful to consider that the meaning of these values was different for the input and output terms:

Value	Output term (O)	Input term (I)
$\perp$	No output Dumb process (Shh_{out})	There can be an input of any type
$\top$	There can be an output of any type	No input Deaf process (Shh_{in})

For example, if there are two outputs of different arities in parallel, the resulting process has type $\top$ for O. With the condition that $O \leq I$, the process is valid if and only if $I = \top$, i.e. if there are no input instruction (you can say anything only if nobody is listening). A similar argument holds by exchanging inputs and outputs. This is a different vision from [YH99] where $\top$ was forbidden as an output type and $\perp$ as an input type.

For the input/output types, we define the partial order:

$$
\begin{array}{c}
\perp < W_1 \times \cdots \times W_k < \top \quad \forall k \; \forall W_i \\
W_1 \times \cdots \times W_k \leq W'_1 \times \cdots \times W'_k \iff W_i \leq W'_i \quad \forall\; 1 \leq i \leq k
\end{array}
$$

This definition induces a complete lattice structure on input/output types, so that the meet and join operations are always defined (the obvious cases with $\perp$ and $\top$ are omitted for simplicity):

$$
W_1 \times \cdots \times W_k \wedge W'_1 \times \cdots \times W'_{k'} \;\triangleq\; \begin{cases} W_1 \wedge W'_1 \times \cdots \times W_1 \wedge W'_k & \text{if } k = k' \\ \quad \text{and } W_i \wedge W'_i \text{ is defined } \forall\; 1 \leq i \leq k & \\ & \\ \perp \quad \text{otherwise} & \end{cases}
$$

$$
W_1 \times \cdots \times W_k \vee W'_1 \times \cdots \times W'_{k'} \;\triangleq\; \begin{cases} W_1 \vee W'_1 \times \cdots \times W_1 \vee W'_k & \text{if } k = k' \\ \quad \text{and } W_i \vee W'_i \text{ is defined } \forall\; 1 \leq i \leq k & \\ & \\ \top \quad \text{otherwise} & \end{cases}
$$

Message Types Finally, we have a type for messages that can be exchanged in an ambient. It can be either an ambient name or a capability. It is safe to add here other common types like Int, $Bool$...

A capability $Cap[T]$ contains simply the effects T which will be released if the capability is executed. And the ordering is natural:

$$Cap[T] \leq Cap[T'] \iff T \leq T'$$

$$Cap[T] \wedge Cap[T'] \triangleq Cap[T \wedge T']$$
$$Cap[T] \vee Cap[T'] \triangleq Cap[T \vee T']$$

An ambient name is composed of a locking annotation and two types representing the processes running inside the ambient. Concerning the locking annotations, we can repeat the same discussion as for mobility annotations. Thus we add two new symbols $\bot_\circ$ and $\top^\circ$, and define $\bot_\circ < \bullet, \circ < \top^\circ$ with $\bullet$ and $\circ$ incomparable.

With $\bot_\circ$ all three constructions $n[P]$, $n[\![P]\!]$ and *open* n are allowed. With $\circ$ only $n[P]$ and *open* n are allowed, and with $\bullet$ only $n[\![P]\!]$ is allowed. With $\top^\circ$ none of them is allowed. However, note that all other constructions (like *in* n or $\langle n \rangle$) are valid with any locking annotation for n.

Concerning the process running inside an ambient, it seems that only one process type would be enough (it was in CGG). With subtyping, we would like to say that a process is allowed to run inside an ambient of type $Amb^Y[T]$ if and only if it has a type $T' \leq T$ (so that $\mathbf{0}$ is always accepted). T represents the maximal effects allowed in the ambient.

Now, what is the natural ordering for ambient names ? Suppose $Amb^Y[T] \leq Amb^Y[T']$ when $T \leq T'$. Then, if n has type $Amb^Y[T]$, it has type $Amb^Y[T']$ by the subsumption rule for any $T' \geq T$. Thus, $n[P]$ is typable for any process P of arbitrary type T', which is contrary to our requirements.

On the other hand, suppose $Amb^Y[T] \leq Amb^Y[T']$ when $T' \leq T$. Then, if n has type $Amb^Y[T]$, it has type $Amb^Y[T_{min}]$ where T_{min} is the minimal process type. *open* n has type $Cap[T_{min}]$, which is contrary to our intuition: n can contain processes with "stronger" effects.

This explains why we need two process types in the type of an ambient name, with two different orderings. In $Amb^Y[T, T']$, T represents the maximal type allowed for processes inside the ambient (i.e. all valid processes also have type T) (cf. rules (Proc Amb$_\circ$) and (Proc Amb$_\bullet$) in Section 3.3), whereas T' represents the maximal effect a valid process can produce (cf. rule (Exp Open)), thus the condition $T \leq T'$ to be coherent. We define:

$$Amb^Y[T_1, T_2] \leq Amb^{Y'}[T_1', T_2'] \iff \begin{cases} Y \leq Y' \\ T_1 \geq T_1' \\ T_2 \leq T_2' \end{cases}$$

At first sight, it could seem strange to declare a new ambient name with $T < T'$: if we specify the maximal allowed type T, why would we say that worse effects T' can appear when opening an ambient of that name ? In fact, we need them to be consistent with the rest of the calculus. Suppose we want to write the program $\langle n \rangle \mid \langle m \rangle \mid (x :?).P$ where n and m accept processes of type T_n and T_m respectively. What input type should we declare for x ? We can use the type of the parallel output $\langle n \rangle \mid \langle m \rangle$, which is with our ordering $Amb^Y[T_n \wedge T_m, T_n \vee T_m]$.

In general, there is no reason to have $T_n \wedge T_m = T_n \vee T_m$. This explains why we cannot replace T and T' by one single process type in an ambient name.

Note that the conditions $Z \neq \vec{\top}$ and $O \leq I$ are not required for an ambient name. For example, $Amb^Y[{}^{\vec{\top}}\top \leadsto \bot, {}^{\vec{\top}}\top \leadsto \bot]$ is the type of an ambient name allowing all processes, whereas the ambient name $Amb^Y[{}^{\vec{\bot}}\bot \leadsto \top, {}^{\vec{\bot}}\bot \leadsto \top]$ is the most restrictive one, allowing only processes which do not have inputs or outputs, i.e. only processes behaving like $\mathbf{0}$.

We can only define partial meet and join operations, since there are some incomparable types (the other cases are undefined):

$$Amb^Y[T_1, T_2] \wedge Amb^{Y'}[T'_1, T'_2] \;\triangleq\; Amb^{Y \wedge Y'}[T_1 \vee T'_1, T_2 \wedge T'_2]$$
$$\text{if } T_1 \vee T'_1 \leq T_2 \wedge T'_2 \text{ (or equivalently if } T_1 \leq T'_2 \text{ and } T'_1 \leq T_2)$$

$$Amb^Y[T_1, T_2] \vee Amb^{Y'}[T'_1, T'_2] \;\triangleq\; Amb^{Y \vee Y'}[T_1 \wedge T'_1, T_2 \vee T'_2]$$

There is no comparability between ambient names and capabilities. It is safe to add other useful types here with their usual ordering (for example, $Int \leq Real$), with no comparability with ambient names and capabilities.

The set of capability types has a structure similar to the process types (i.e. a complete lattice). The set of ambient names has a maximal element ($Amb^{\top^\circ}[{}^{\vec{\bot}}\bot \leadsto \top, {}^{\vec{\top}}\top \leadsto \bot]$), but infinitely many minimal elements: all the types $Amb^{\bot^\circ}[T, T]$ for every process type T.

3.3 Typing Rules

Having defined the types and explained their ordering, we can now give the typing rules of this new type system. A feature of the present approach w.r.t. CGG is that we avoid to introduce arbitrary types in the conclusions of typing rules. However note that this derivation system is also not deterministic because of the subsumption rules and the shape of some rules (for example, in (Proc Par), the same type T appears in two premises).

Good Environment ($E \vdash \diamond$)

$$\text{(Env } \emptyset) \; \frac{}{\emptyset \vdash \diamond} \qquad\qquad \text{(Env } n) \; \frac{E \vdash \diamond \quad n \notin dom(E)}{E, n : W \vdash \diamond}$$

These two rules are exactly the same as in CGG.

Good Expression of Type W ($E \vdash M : W$)

$$\text{(Exp In)} \; \frac{E \vdash M : Amb^Y[T, T']}{E \vdash in\ M : Cap[{}^{\frown}\bot \leadsto \top]} \qquad \text{(Exp Out)} \; \frac{E \vdash M : Amb^Y[T, T']}{E \vdash out\ M : Cap[{}^{\frown}\bot \leadsto \top]}$$

$$\text{(Exp Open)} \; \frac{E \vdash M : Amb^\circ[T, T']}{E \vdash open\ M : Cap[T']} \qquad \text{(Exp Imm)} \; \frac{E \vdash \diamond}{E \vdash imm : Cap[{}^{\veebar}\bot \leadsto \top]}$$

$$\text{(Exp } n) \; \frac{E', n : W, E'' \vdash \diamond}{E', n : W, E'' \vdash n : W} \qquad \text{(Exp Sub)} \; \frac{E \vdash M : W \quad W \leq W'}{E \vdash M : W'}$$

In (Exp In), (Exp Out) and (Exp Imm), the type in the conclusion of the rule is the minimal effect (or constraint) that the corresponding instruction produces. In (Exp Open), we use the maximal effect contained in the ambient name and we check that M is an unlocked ambient. In (Exp Sub), we allow to upgrade the type of an expression. The other rules are identical to those of CGG.

Good Process of Type $\boldsymbol{T}$ ($\boldsymbol{E \vdash P : T}$)

$$\text{(Proc Par)}\ \frac{E \vdash P : T \quad E \vdash Q : T}{E \vdash P \mid Q : T} \qquad \text{(Proc Action)}\ \frac{E \vdash M : Cap[T] \quad E \vdash P : T}{E \vdash M.P : T}$$

$$\text{(Proc Zero)}\ \frac{E \vdash \diamond}{E \vdash \mathbf{0} : {}^{\text{↳}}\bot \rightsquigarrow \top} \qquad \text{(Proc Repl)}\ \frac{E \vdash P : T}{E \vdash !P : T}$$

$$\text{(Proc Res)}\ \frac{E, n : Amb^{Y}[T_n, T'_n] \vdash P : T}{E \vdash (\nu n : Amb^{Y}[T_n, T'_n])P : T}$$

$$\text{(Proc Amb}_\circ\text{)}\ \frac{E \vdash M : Amb^{\circ}[T, T'] \quad E \vdash P : T}{E \vdash M[P] : {}^{\text{↳}}\bot \rightsquigarrow \top}$$

$$\text{(Proc Amb}_\bullet\text{)}\ \frac{E \vdash M : Amb^{\bullet}[T, T'] \quad E \vdash P : T}{E \vdash M[\![P]\!] : {}^{\text{↳}}\bot \rightsquigarrow \top}$$

$$\text{(Proc Input)}\ \frac{E, n_1 : W_1, \ldots, n_k : W_k \vdash P : {}^{Z}O \rightsquigarrow I \quad I \leq W_1 \times \cdots \times W_k}{E \vdash (n_1 : W_1, \ldots, n_k : W_k).P : {}^{Z}O \rightsquigarrow I}$$

$$\text{(Proc Output)}\ \frac{E \vdash M_1 : W_1 \quad \cdots \quad E \vdash M_k : W_k \quad (E \vdash \diamond \text{ if } k = 0)}{E \vdash \langle M_1, \ldots, M_k \rangle : {}^{\text{↳}}W_1 \times \cdots \times W_k \rightsquigarrow \top}$$

$$\text{(Proc Sub)}\ \frac{E \vdash P : T \quad T \leq T' \quad T' \ valid}{E \vdash P : T'}$$

(Proc Par), (Proc Action), (Proc Repl) and (Proc Res) are the same rules as those of CGG (with a syntax modification for (Proc Res)). In (Proc Zero), we use the minimal process type.

In (Proc Amb$_\circ$), we check that M is an unlocked ambient name and that P has the type of an allowed process inside M (with the subsumption rule, we can always upgrade it or decrease the type in the ambient name so that they match). Like for $\mathbf{0}$, we use the minimal process type in the conclusion of the rule. (Proc Amb$_\bullet$) is similar.

In (Proc Input), we just need to check that the input type of the process P is below the type generated by the input (i.e. is more specific since the ordering for input types is contravariant). This is valid: every input type is accepted in the conclusion provided that it covers the input $W_1 \times \cdots \times W_k$. If P has a bigger input type ($\top$ for $\mathbf{0}$ for example), it must first be upgraded with the subsumption rule before applying (Proc Input).

In (Proc Output), we just give the minimal effect: the output of a type $W_1 \times \cdots \times W_k$. Since the output is asynchronous, there is no condition to check like in (Proc Input).

(Proc Sub) is the classical subsumption rule, with the additional condition that the new process type must be valid (we are explicitly typing a process here and not a capability or an ambient name).

3.4 Results

Theorem 2 (Subject reduction). *If $E \vdash P : T$ and $P \rightarrow Q$, then $E \vdash Q : T$.*

Theorem 3 (Validity). *If $E \vdash P : T$, then T is valid.*

By this last theorem, we are sure that a well-typed process will never cause run-time faults, i.e. there will never be an exchange of incompatible values during execution and it cannot contain instructions requiring both mobility and immobility (like in $imm.in\ n.P$). Another desired property is that we do not want an ambient to be opened if it is locked. This property is a direct result of the type system: the instructions $open\ n$ and $n[P]$ can be typed only if we can prove that n is unlocked.

4 A First Typing Algorithm

In this Section, we are going to deduce a typing algorithm from the type system we introduced in the previous one. Then, we will see that this algorithm returns exactly all the types (in a certain sense) that could be derived.

4.1 Typing Rules

Definition 4. *An environment is said to be* well-formed *if all the names it contains are different. This is of course equivalent to $E \vdash \diamond$. Algorithmically, it just consists in checking that all the names are different.*

For any well-formed environment E, we define an algorithm returning the type of expressions and processes by the following rules. For every undefined case, we will say that the algorithm *fails*. Note that even if we write it as derivation rules for simplicity, it can also be expressed directly in an algorithmic way. Note also that the algorithm can be implemented in a parallel way when there are several recursive calls (for instance in (Type Par)).

Type of an Expression M ($Type(E, M) = W$)

$$\text{(Type In)}\ \frac{Type(E, M) = Amb^Y[T, T']}{Type(E, in\ M) = Cap[\curvearrowright \bot \rightsquigarrow \top]}$$

$$\text{(Type Out)}\ \frac{Type(E, M) = Amb^Y[T, T']}{Type(E, out\ M) = Cap[\curvearrowright \bot \rightsquigarrow \top]}$$

$$\text{(Type Open)}\ \frac{Type(E, M) = Amb^Y[T, T'] \quad Y \leq \circ}{Type(E, open\ M) = Cap[T']}$$

$$\text{(Type Imm)}\ \frac{}{Type(E, imm) = Cap[\underline{\vee} \bot \rightsquigarrow \top]}$$

$$\text{(Type } n\text{)}\ \frac{}{Type((E', n : W, E''), n) = W}$$

For each message type, we always return the minimal type required by this capability (for example, $Cap[\curvearrowright \bot \rightsquigarrow \top]$ for $in\ M$). In (Type Open), we return the maximal effects T' which can appear when opening an ambient of that name and we check that this ambient is unlocked by $Y \leq \circ$.

Type of a Process P ($Type(E,P) = T$)

$$\text{(Type Par)}\ \frac{Type(E,P) = T \quad Type(E,Q) = T' \quad T \vee T'\ valid}{Type(E,P \mid Q) = T \vee T'}$$

$$\text{(Type Action)}\ \frac{Type(E,M) = Cap[T] \quad Type(E,P) = T' \quad T \vee T'\ valid}{Type(E,M.P) = T \vee T'}$$

$$\text{(Type Zero)}\ \frac{}{Type(E,\mathbf{0}) = {}^{\lrcorner}\bot \leadsto \top} \qquad \text{(Type Repl)}\ \frac{Type(E,P) = T}{Type(E,!P) = T}$$

$$\text{(Type Res)}\ \frac{Type((E, n : Amb^Y[T_n, T'_n]), P) = T}{Type(E, (\nu n : Amb^Y[T_n, T'_n])P) = T}$$

$$\text{(Type Amb}_\circ\text{)}\ \frac{Type(E,M) = Amb^Y[T,T'] \quad Type(E,P) = T'' \quad T'' \leq T \quad Y \leq \circ}{Type(E, M[P]) = {}^{\lrcorner}\bot \leadsto \top}$$

$$\text{(Type Amb}_\bullet\text{)}\ \frac{Type(E,M) = Amb^Y[T,T'] \quad Type(E,P) = T'' \quad T'' \leq T \quad Y \leq \bullet}{Type(E, M[\![P]\!]) = {}^{\lrcorner}\bot \leadsto \top}$$

$$\text{(Type Input)}\ \frac{Type((E, n_1 : W_1, \ldots, n_k : W_k), P) = {}^Z O \leadsto I \quad O \leq W_1 \times \cdots \times W_k}{Type(E, (n_1 : W_1, \ldots, n_k : W_k).P) = {}^Z O \leadsto I \wedge W_1 \times \cdots \times W_k}$$

$$\text{(Type Output)}\ \frac{Type(E,M_1) = W_1 \quad \cdots \quad Type(E,M_k) = W_k}{Type(E, \langle M_1, \ldots, M_k \rangle) = {}^{\lrcorner}W_1 \times \cdots \times W_k \leadsto \top}$$

In (Type Par), we just take the join of the two sub-processes types, ensuring first that the resulting process is still valid.

In (Type Amb$_\circ$), we must check that the type T'' of P is accepted by this ambient ($T'' \leq T$) and that the ambient can be opened ($Y \leq \circ$).

In (Type Input), we add the information of the input instruction by returning the meet of I and $W_1 \times \cdots \times W_k$. We must also check that $O \leq W_1 \times \cdots \times W_k$ in P, to ensure that $O \leq I \wedge W_1 \times \cdots \times W_k$ in the resulting process.

In (Type Output), we just put the information of an output of type $W_1 \times \cdots \times W_k$. Since there is no continuation, there is nothing to check here.

The other rules are similar or (quite) natural.

4.2 Results

Theorem 5 (Soundness).

- *If $Type(E,M) = W$, then $E \vdash M : W$.*
- *If $Type(E,P) = T$, then $E \vdash P : T$.*

Theorem 6 (Completeness).

- *If $E \vdash M : W$, then the algorithm succeeds on M and $Type(E,M) \leq W$.*
- *If $E \vdash P : T$, then the algorithm succeeds on P and $Type(E,P) \leq T$.*

From those two theorems, we easily deduce the property of minimal type for our type system and that the algorithm is able to compute it efficiently.

Corollary 7 (Minimal Type). *The set of all possible types for a typable expression or process has a minimum and this minimum is precisely the type returned by the algorithm.*

5 Type Inference

In the previous Section, we described a deterministic typing algorithm. This is a satisfactory result, to be compared to the nondeterministic type system we had before. But, up to now, this algorithm performs only type-checking: the programmer must still annotate explicitly all ambient names and input variables with their types. To go one step further, the natural extension would be to remove these type annotations and try to design a type-inference algorithm. Unfortunately, even if at first sight this seems to require only minor modifications, some new and difficult problems appear if we want to keep the subtyping relation. So we will have to go a little back and restrict our problem to the original type system of CGG. For this system, we will show that it can be completely and efficiently solved with a Damas-Milner style algorithm.

5.1 Background

For the syntax, we will consider the calculus we studied since the beginning, that is with the two new constructs and the associated reduction rules (they do not bring any new difficulties). For the type system, we will nearly take the typing rules of CGG, as they are described in [CGG99]. We modify them only to handle the two new constructs.

Instead of simply removing the type annotations, we keep them but allow to write type variables instead. For this, we must extend the definitions of types by adding an infinite set of variables for each of them. More generally, for the same letter, the lower case one will denote a type variable and the upper case one will denote a metavariable (as before).

Now we can write expressions like $(x : w).P$ or $(\nu n : Amb^y[t])P$, or even $(x : Cap[{}^\frown u]).P$. In fact, we allow to mix both type variables and explicit types in a same term or even in a same type expression. By this mean, we get a more generic algorithm, and this property can be useful in practice: for example, if you want to check an insecure code, you should be able to constraint some of its types by specifying them explicitly before applying the type-inference algorithm. Note also that one can express equality constraints between types just by using the same variable: in $(x : w).P \mid (y : w).Q$, the input variables x and y must have the same type.

5.2 The Algorithm

We first need some classical definitions and results: a *substitution* is a total map from the set of all type variables (of any kind) to types of the same kind. We will denote them by the letters σ, θ, ρ... The empty substitution is the identity function and will be noted **id**. Finally, the composition of substitutions is defined in exactly the same way as functions. We extend naturally substitutions to complex types (and not only type variables), to processes (by replacing type annotations in input and restriction constructions) and to environments.

An *unifier* of two types X_1 and X_2 is a substitution σ such that $\sigma(X_1) = \sigma(X_2)$. Since the types in our system are simple trees, we know that there is a sound and complete unification algorithm for those types. It returns the *principal unifier* of two types (when it exists; otherwise it fails). We will call it $mgu(.,.)$ (it is not difficult to write its rules explicitly).

We can know give the rules of the typing algorithm. They are used to infer judgments of the forms $Infer(E, M) = (W, \sigma)$ and $Infer(E, P) = (T, \sigma)$, where W or T is the most generic possible type for M or P (possibly containing type variables), σ is a substitution representing the constraints on the type variables in M or P, and E is a well-formed environment.

In the following rules, the premises must be read (and applied) from left to right. We do not detail how the algorithm gets *new* type variables. We will only consider that whenever a variable is declared **new**, it is different from all type variables previously used. In practice this can be achieved by using a global counter to number new type variables.

Type-Inference for an Expression M ($Infer(E, M)$)

$$\text{(Infer In)}\ \frac{Infer(E,M) = (W,\sigma) \quad y, t\ \mathbf{new} \quad mgu(W, Amb^{y}[t]) = \rho \quad u\ \mathbf{new}}{Infer(E, in\ M) = (Cap[\frown u], \rho\sigma)}$$

$$\text{(Infer Out)}\ \frac{Infer(E,M) = (W,\sigma) \quad y, t\ \mathbf{new} \quad mgu(W, Amb^{y}[t]) = \rho \quad u\ \mathbf{new}}{Infer(E, out\ M) = (Cap[\frown u], \rho\sigma)}$$

$$\text{(Infer Open)}\ \frac{Infer(E,M) = (W,\sigma) \quad t\ \mathbf{new} \quad mgu(W, Amb^{\circ}[t]) = \rho}{Infer(E, open\ M) = (Cap[\rho(t)], \rho\sigma)}$$

$$\text{(Infer Imm)}\ \frac{u\ \mathbf{new}}{Infer(E, imm) = (Cap[\veebar u], \mathbf{id})}$$

$$\text{(Infer } n\text{)}\ \frac{}{Infer((E, n : W, E'), n) = (W, \mathbf{id})}$$

Type-Inference for a Process P ($Infer(E, P)$)

$$\text{(Infer Par)}\ \frac{\begin{array}{r} Infer(E,P) = (T,\sigma) \quad Infer(\sigma(E), \sigma(Q)) = (T', \sigma') \\ mgu(\sigma'(T), T') = \rho \end{array}}{Infer(E, P \mid Q) = (\rho(T'), \rho\sigma'\sigma)}$$

$$\text{(Infer Zero)}\ \frac{t\ \mathbf{new}}{Infer(E, \mathbf{0}) = (t, \mathbf{id})} \qquad \text{(Infer Repl)}\ \frac{Infer(E,P) = (T,\sigma)}{Infer(E, !P) = (T,\sigma)}$$

$$\text{(Infer Action)}\ \frac{\begin{array}{r} Infer(E,M) = (W,\sigma) \quad Infer(\sigma(E), \sigma(P)) = (T, \sigma') \\ mgu(\sigma'(W), Cap[T]) = \rho \end{array}}{Infer(E, M.P) = (\rho(T), \rho\sigma'\sigma)}$$

$$\text{(Infer Res)}\ \frac{Infer((E, n : Amb^{Y}[T]), P) = (T', \sigma)}{Infer(E, (\nu n : Amb^{Y}[T])P) = (T', \sigma)}$$

$$\text{(Infer Amb}_{\circ}\text{)}\ \frac{\begin{array}{r} Infer(E,M) = (W,\sigma) \quad Infer(\sigma(E), \sigma(P)) = (T, \sigma') \\ mgu(\sigma'(W), Amb^{\circ}[T]) = \rho \quad t\ \mathbf{new} \end{array}}{Infer(E, M[P]) = (t, \rho\sigma'\sigma)}$$

$$\text{(Infer Amb}_\bullet) \quad \frac{\begin{array}{c} Infer(E, M) = (W, \sigma) \quad Infer(\sigma(E), \sigma(P)) = (T, \sigma') \\ mgu(\sigma'(W), Amb^\bullet[T]) = \rho \quad t \textbf{ new} \end{array}}{Infer(E, M[\![P]\!]) = (t, \rho\sigma'\sigma)}$$

$$\text{(Infer Input)} \quad \frac{\begin{array}{c} Infer((E, n_1 : W_1, \ldots, n_k : W_k), P) = (T, \sigma) \\ z \textbf{ new} \quad mgu(T, {}^z\sigma(W_1) \times \cdots \times \sigma(W_k)) = \rho \end{array}}{Infer(E, (n_1 : W_1, \ldots, n_k : W_k).P) = (\rho(T), \rho\sigma)}$$

$$\text{(Infer Output)} \quad \frac{\begin{array}{c} Infer(E, M_1) = (W_1, \sigma_1) \quad Infer(\sigma_1(E), M_2) = (W_2, \sigma_2) \\ \cdots \quad Infer(\sigma_{k-1} \ldots \sigma_1(E), M_k) = (W_k, \sigma_k) \quad z \textbf{ new} \end{array}}{\begin{array}{l} Infer(E, \langle M_1, \ldots, M_k \rangle) = \\ \qquad ({}^z\sigma_k \ldots \sigma_2(W_1) \times \cdots \times \sigma_k(W_{k-1}) \times W_k, \sigma_k \ldots \sigma_1) \end{array}}$$

5.3 Results

Theorem 8 (Soundness). *If $Infer(E, P) = (T, \sigma)$, then $\sigma(E) \vdash_{CGG} \sigma(P) : T$. Moreover, $\sigma'\sigma(E) \vdash_{CGG} \sigma'\sigma(P) : \sigma'(T)$ for any substitution σ' (we will say that all these derivations are* solutions *returned by the inference algorithm).*

Theorem 9 (Completeness). *If there is a type T such that $\sigma(E) \vdash_{CGG} \sigma(P) : T$ (i.e. if the process P is typable in the environment E after performing some substitutions on type variables), the inference algorithm $Infer(E, P)$ succeeds and $\sigma(E) \vdash_{CGG} \sigma(P) : T$ is one of the returned solutions.*

5.4 Type Inference with Subtyping

Returning back to the original problem, can we do the same as above with the type system with subtyping ? Adding subtyping brings many problems, mainly because there is no minimal type for ambient names and because we get ordering constraints due to ambient names and valid processes. Some similar problems appeared in the type system of Abadi and Cardelli for object calculus. In this case, Jens Palsberg gave a solution in [Pal95], by building a graph of constraints and checking some properties on it. Maybe the same approach would be possible with the ambient calculus, but our attempts in this way failed. Up to now, all we could do is build a set of constraints that type variables should satisfy in order to get a solution. But solving it remains an open problem (see [Zim99] for more details and explanations).

6 Conclusion

We have extended the previous type system for mobile ambients with new types and with a subtyping relation. We gave the corresponding typing rules and deduced a type-checking algorithm. We also gave a type-inference algorithm for CGG, but the problem of solving the constraints set in the system with subtyping remains open.

These algorithms are efficient and could be implemented quite easily. To our knowledge, there are two “implementations” of ambients so far: a Java applet from L. Cardelli and a translation into the join-calculus in a modified version of Objective Caml ([FS99]). None of them use types for now.

An other primitive was introduced by Cardelli-Ghelli-Gordon in [CGG99]: the primitive *go*, which performs objective moves. To prevent some dangerous effects such moves can induce (entrapping of an ambient), they extended the type system so that the type of an ambient name says explicitly if the ambient allows them or not. We did not keep this primitive to simplify the notations for ambient names, but we checked that all our work and algorithms could be easily extended so as to include *go*.

In [LS00], Levi and Sangiorgi studied *plain and grave interferences* in the ambient calculus. They proposed a syntax extension along with a new type system to prevent grave interferences. Future work may be to extend our subtyping relation and algorithms to their system.

Acknowledgments

This work is the result of a two-months internship in the University of Turin, Italy. I would like to express my gratitude to Mariangiola Dezani, who supervised the internship and proposed the subject. Many thanks also to Paola Giannini, Ferruccio Damiani and Giorgio Ghelli for some useful suggestions and pointers. Last but not least, thanks to Daniel Hirschkoff for correcting the draft.

References

[Car99a] L. Cardelli. Abstractions for Mobile Computation. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, 1999.

[Car99b] L. Cardelli. Wide Area Computation. In *ICALP'99*, April 1999.

[CG97] L. Cardelli and A. D. Gordon. A Calculus of mobile Ambients. 1997. Slides.

[CG98] L. Cardelli and A. D. Gordon. Mobile Ambients. In *Proceedings FoSSaCS'98*, volume LNCS 1378, pages 140–155. Springer, 1998.

[CG99] L. Cardelli and A. D. Gordon. Types for Mobile Ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 79–92. ACM, January 1999.

[CGG99] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility Types for Mobile Ambients. In *Proceedings of ICALP'99*, volume LNCS, April 1999.

[FS99] C. Fournet and A. Schmitt. An Implementation of Ambients in JoCaml. In *Proceedings MOS'99*, April 1999.

[LS00] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. Draft of a paper to appear in the Proceedings of POPL'00, 2000.

[Mil91] R. Milner. The Polyadic π-Calculus: a Tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Pal95] J. Palsberg. Efficient Inference of Object Types. *Information and Computation*, 1995.

[YH99] N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher Order Processes. Technical Report 01/99, University of Sussex, May 1999.

[Zim99] P. Zimmer. Subtyping and Typing Algorithms for Mobile Ambients. Internship Report – Ecole Normale Supérieure de Lyon, 1999. Available at `http://www.ens-lyon.fr/~pzimmer/`.