Behavioural Typing for Safe Ambients[*]

Michele Bugliesi
Dipartimento di Informatica
Università Ca' Foscari di Venezia

Giuseppe Castagna
C.N.R.S.
Laboratoire d'Informatique, ENS, Paris

Abstract

We introduce a typed variant of *Safe Ambients*, named *Secure Safe Ambients* (SSA), whose type system allows behavioral invariants of ambients to be expressed and verified. The most significant aspect of the type system is its ability to capture *both* explicit *and* implicit process and ambient behavior: process types account not only for immediate behavior, but also for the behavior resulting from capabilities a process acquires during its evolution in a given context. Based on that, the type system provides for static detection of security attacks such as *Trojan Horses* and other combinations of malicious agents.

We study the type system of SSA, define algorithms for type checking and type reconstruction, define languages for expressing security properties, and study a distributed version of SSA and its type system. For the latter, we show that distributed type checking ensures security even in ill-typed contexts, and discuss how it relates to the security architecture of the Java Virtual Machine.

1 Introduction

Mobile Ambients [6] are named agents or locations that enclose collections of running processes, possibly including nested sub-ambients. *Safe Ambients* [12] are a variant of Mobile Ambients. The two calculi differ in the underlying notion of interaction: in Mobile Ambients, interaction is "one-sided", in that one of the two partners in a *move* or *open* action simply undergoes the action. In Safe Ambients, instead, the reduction relation requires actions to synchronize with corresponding co-actions. To exemplify, consider the ambients a and b described below:

Mobile Ambients $$a[\mathsf{open}\ b.\mathsf{in}\ c] \mid b[\mathsf{in}\ a.\mathsf{in}\ d].$$

The brackets $[\ldots]$ represent ambient boundaries, " $\mid$ " denotes parallel composition, and "." enforces sequential execution. Given the above configuration, the ambient b may enter a, by exercising the *capability* "$\mathsf{in}\ a$", and reduce to $a[\mathsf{open}\ b.\mathsf{in}\ c \mid b[\mathsf{in}\ d]]$. Then a may dissolve the boundary provided by b by exercising $\mathsf{open}\ b$, and reduce to $a[\mathsf{in}\ c \mid \mathsf{in}\ d]$.

Neither of the two reductions is legal in Safe Ambients. To obtain the behavior we just described, the two ambients a and b should be written as follows:

Safe Ambients $$a[\mathsf{coin}\ a.\mathsf{open}\ b.\mathsf{in}\ c] \mid b[\mathsf{in}\ a.\mathsf{coopen}\ b.\mathsf{in}\ d].$$

[*] To appear in COMPUTER LANGUAGES, Elsevier Science. Extended version of the article *Secure Safe Ambients* included in the *Proc. of the 28th ACM Symposium on Principles of Programming Languages* (POPL '01), pages 222–235, ACM Press

Now the move of b into a arises as the result of a mutual agreement between the two partners: b exercising the capability $\mathsf{in}\ a$, and a exercising the *co-capability* $\mathsf{coin}\ a$. The resulting configuration, that is, $a[\mathsf{open}\ b.\mathsf{in}\ c \mid b[\mathsf{coopen}\ b.\mathsf{in}\ d]]$, reduces to $a[\mathsf{in}\ c \mid \mathsf{in}\ d]$, again as the result of the synchronization between $\mathsf{open}\ b$ and $\mathsf{coopen}\ b$.

1.1 Motivations

Secure Safe Ambients (SSA) are a typed variant of Safe Ambients whose type system is so defined as to allow behavioral invariants of ambients to be expressed and verified. The most significant aspect of the type system is its ability to trace *both* explicit *and* implicit process behavior and ambient mobility: the type assigned to a process accounts not only for the behavior resulting from the capabilities that process possesses in isolation, but also from the capabilities the process may acquire by interacting with the surrounding environment. This degree of accuracy is useful for a sound verification of security policies, as implicit (i.e., acquired) mobility is at the core of a number of security attacks such as *Trojan Horses* or other combinations of malicious agents.

EXAMPLE 1.1. Consider again the two (safe) ambients a and b introduced above, now running in parallel with a third ambient c as in the following configuration, where P and Q are arbitrary processes:

$$a[\mathsf{coin}\ a.\mathsf{open}\ b.\mathsf{in}\ c] \quad \mid \quad b[\mathsf{in}\ a.\mathsf{coopen}\ b.\mathsf{in}\ d] \quad \mid \quad c[\ \mathsf{coin}\ c.P \mid d[\mathsf{coin}\ d.Q]\,]$$

For the purpose of the example, assume that d contains confidential data, which should be made available to ambients running *within* c (which may enter, as signaled by the co-capability $\mathsf{coin}\ d$), but not to ambients *entering* c. Given this security policy, the question is whether c should let a in without fear that a may access the confidential data in d. If we only look at explicit mobility, that is at the capabilities immediately available for a, then the move of a into c seems safe, as a does not make any direct attempt to move into d. However, a can be used as a Trojan Horse for b: a can let b in, then enter c and, once inside c, open b to gain access to d. □

EXAMPLE 1.2. A different way that a may attack c is by letting b out after having entered c. The two ambients a and b would then be written as shown below:

$$a[\mathsf{coin}\ a.\mathsf{in}\ c.\mathsf{coout}\ a] \quad \mid \quad b[\mathsf{in}\ a.\mathsf{out}\ a.\mathsf{in}\ d] \quad \mid \quad c[\ \mathsf{coin}\ c.P \mid d[\mathsf{coin}\ d.Q]\,]$$

Again, if we only look at the capabilities available for a, we are mislead to let a into c. Yet, a could let b in, then enter c, and finally let b out handing over to b the capability to enter d. □

1.2 Overview

The type system we discuss in this paper provides for static, type-driven verification of security. It allows the definition of security policies for ambients, and provides mechanisms for static detection of any attempt to break those policies. In particular, the type system detects security attacks based on implicit (and undesired or malicious) acquisition of capabilities by hostile agents such as those described in the previous examples. As argued in [12], the presence of co-capabilities is essential for an accurate static characterization of processes in the type system: our choice of Safe Ambients as the basis for our type system is motivated by the same reasons.

There are three key ingredients to the type system.

Ambient Domains. Ambients are classified by *ambient domains* (also called *protection domains* or simply *domains*): each domain has an associated *behavior* that ambients in the domain share and must comply with, and an associated *security policy* that protects the ambients in the domain from undesired interactions with the surrounding context.

Type-level capabilities and Process Types. Process types describe process behavior using domains as the unit of abstraction. The term-level capabilities available to processes are abstracted upon in the type system by resorting to type-level capabilities. Process types are defined in terms of sets of type capabilities: to exemplify, if a is, say, an ambient of domain A and P is a (well-typed) process exercising the term-level capability $\mathsf{in}\ a$, then the type of P traces this behavior by including the type-level capability $\mathsf{in}\ A$.

To gain accuracy in the description of ambient behavior, the type system traces the *nesting level* at which the effect of exercising a capability may be observed. This is accomplished by introducing chemical abstract model, where exercising a capability corresponds, in the typing rules, to releasing a type-level capability, or *molecule*. Molecules are classified as *plain*, *light*, and *heavy*: plain molecules are released at the nesting level of the process exercising the corresponding capability, light molecules at upper level (the level of the enclosing ambient), while heavy molecules are released within ambients. Molecules react with co-molecules (corresponding to co-capabilities) released at the same nesting level. Thus, in the chemical metaphor, type checking corresponds to a chromatographic analysis in which each element of different weight is precisely determined.

Security Constraints. Each ambient domain has an associated set of security constraints that define the security policy for that domain: the constraints establish the access rights for ambients crossing the boundary of any of the ambients in the domain.

1.3 Contributions

We prove two main results for our type system. The first is subject reduction, the second is a rather strong form of type safety showing that types provide a safe approximation of behavior: specifically, we show that if a process P running inside a context $\mathscr{C}$ may (after any number of reduction steps of $\mathscr{C}$) exercise a capability on some name, and $\mathscr{C}$ is well-typed, then the corresponding type capability is traced by the static type of P. For that we introduce a new and powerful notion of *residual*. As a corollary, we then deduce that well-typed processes comply with the security policies established by ambients.

We also define a type-checking algorithm that computes minimum types and, more importantly, an algorithm for type reconstruction: we prove both sound and complete. Type reconstruction is particularly important for our purposes, as it infers the behavior of ambient domains, thus leaving the programmer with the only task of specifying the domains of ambients, and their associated security policies.

We continue by studying a distributed variant of SSA, where each ambient carries its own type environment along with it, and type-checking is performed locally by the ambient at any time other ambients cross its own boundaries. The distributed variant of the calculus and its type system are particularly interesting in perspective, in view of a practical implementation. In a highly distributed system it is clearly unrealistic to rely on the assumption that type checking may access information on all the components of the system. Accordingly, in the distributed version of the calculus, we dispense with global security and type soundness, and replace them by local type checking and security analysis. A typed version of reduction complements these analyses by allowing ambient boundaries to be crossed only by ambients satisfying the type and security checks performed, *just in time*, by the ambient whose domain is being crossed.

The study of the distributed version yields, as a byproduct, a further interesting result. Looking at the dynamic checks performed upon reduction, one discovers that they correspond to the type and security checks performed by the three components of the security architecture of the Java Virtual Machine: the *Class Loader*, the *Bytecode Verifier*, and the *Security Manager*.

Finally we study the system in the presence of communication primitives. This extension, absent from [3], is non-trivial, as capabilities, as well as names, can be exchanged in communications. Therefore characterizing ambient and process behavior in the presence of communications involves tracing not only the capabilities a process may acquire by mobility interactions with the environment, but also those that may be obtained via explicit communication.

1.4 Plan of the paper

Section 2 reviews the syntax and reduction semantics of (Secure) Safe Ambients. Section 3 defines the type system, while Section 4 focuses on type soundness and safety. Section 5 introduces the algorithmic systems, and proves them sound and complete. Section 6 shows how to define a security layer on top of the type system, and how the type system may be used enforce and verify security properties. In Section 7 we define a distributed version of SSA, and discuss how it relates to the security architecture of the JVM. in Section 8 we extend our system with communications. A short section concludes the presentation. Proofs of the main results are given in separate appendixes.

2 The language

The terms of our language are those of *Safe Ambients* with the only difference that the types of (ambient) names are *domains*. These are type-level constants used to identify ambients that satisfy the same behavioral invariants and share common security policies: instead of associating such invariants and policies to each ambient we rather define them for domains, and then group ambients in domains.

Processes

$$P ::= \mathbf{0} \;\mid\; \alpha.P \;\mid\; (\boldsymbol{\nu} a{:}D)P \;\mid\; P \mid P \;\mid\; a[P] \;\mid\; !P$$

Capabilities

$$\alpha ::= \mathsf{in}\ a \;\mid\; \mathsf{coin}\ a \;\mid\; \mathsf{out}\ a \;\mid\; \mathsf{coout}\ a \;\mid\; \mathsf{open}\ a \;\mid\; \mathsf{coopen}\ a$$

Besides being a design choice, the introduction of domains is motivated by technical reasons. An alternative, and more informative, notion of ambient type could be defined by associating each ambient with the set of term-level capabilities that ambient may exercise. The resulting type system would certainly provide a more accurate characterization of process and ambient behavior, but it would also incur into a number of technical problems arising from the dependency of these types on terms[1]. On the other hand, our use of protection domains is well motivated and justified by what is nowadays common practice for languages and systems supporting code mobility [10].

[1] One problem with that solution is that types are not preserved by structural congruence. For instance, the term $(\boldsymbol{\nu} a{:}A)(\boldsymbol{\nu} b{:}B)a[\mathsf{in}\ b] \mid b[\mathsf{coin}\ b]$ would not be typeable, as the type A should contain all the capabilities a can exercise: yet A cannot contain $\mathsf{in}\ b$, as b is in the scope of a nested binder. If we exchange the position of the two binders, as in $(\boldsymbol{\nu} b{:}B)(\boldsymbol{\nu} a{:}A)a[\mathsf{in}\ b] \mid b[\mathsf{coin}\ b]$ the term becomes typeable. The use of domains resolves the problem: both terms are well-typed when A and B are domains (thus type constants rather than sets of term-level capabilities).

Reduction

The reduction relation for SSA derives from the one defined for *Safe Ambients*. We let Q, R and S range over arbitrary processes.

(in) $b[\mathsf{in}\ \ a.P \mid Q] \mid a[\mathsf{coin}\ \ a.R \mid S] \rightarrow a[R \mid S \mid b[P \mid Q]]$

(out) $a[\ b[\mathsf{out}\, a.P \mid Q] \mid \mathsf{coout}\, a.R \mid S] \rightarrow b[P \mid Q] \mid a[R \mid S]$

(open) $\mathsf{open}\ \ a.P \mid a[\mathsf{coopen}\ \ a.Q \mid R] \rightarrow P \mid Q \mid R$

(context) $P \rightarrow Q \;\Rightarrow\; \mathscr{E}[P] \rightarrow \mathscr{E}[Q]$

(struct)[2] $P' \equiv P \rightarrow Q \;\Rightarrow\; P' \rightarrow Q$

where $\mathscr{E}[\,]$ denotes an evaluation context defined as

Evaluation Contexts

$$\mathscr{E}[\,] ::= [\,] \;\Big|\; (\boldsymbol{\nu} a{:}D)\mathscr{E}[\,] \;\Big|\; P \mid \mathscr{E}[\,] \;\Big|\; \mathscr{E}[\,] \mid P \;\Big|\; a[\mathscr{E}[\,]]$$

and $\equiv$ is the standard structural equivalence relation for ambients, that is the least congruence relation that is a commutative monoid for **0** and $\mid$ and closed under the following rules:

$$\begin{array}{ll}
!P \equiv !P \mid P & \\
(\boldsymbol{\nu} a{:}D)\mathbf{0} \equiv \mathbf{0} & \\
(\boldsymbol{\nu} a{:}A)(\boldsymbol{\nu} b{:}B)P \equiv (\boldsymbol{\nu} b{:}B)(\boldsymbol{\nu} a{:}A)P & \text{for } a \neq b \\
(\boldsymbol{\nu} a{:}D)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} a{:}D)Q & \text{for } a \notin \mathit{fn}(P) \\
(\boldsymbol{\nu} a{:}D)b[P] \equiv b[(\boldsymbol{\nu} a{:}D)P] & \text{for } a \neq b
\end{array}$$

Here $\mathit{fn}(P)$ denotes the set of free names of P, defined as:

$$\begin{array}{lll}
\mathit{fn}(\mathbf{0}) = \varnothing & \mathit{fn}(\mathsf{cap}\, a.P) = \mathit{fn}(P) \cup \{a\} & \mathit{fn}((\boldsymbol{\nu} a{:}D)P) = \mathit{fn}(P) \setminus \{a\} \\
\mathit{fn}(P_1 \mid P_2) = \mathit{fn}(P_1) \cup \mathit{fn}(P_2) & \mathit{fn}(a[P]) = \mathit{fn}(P) \cup \{a\} & \mathit{fn}(!P) = \mathit{fn}(P)
\end{array}$$

Here $\mathsf{cap} \in \{\mathsf{in}, \mathsf{coin}, \mathsf{out}, \mathsf{coout}, \mathsf{open}, \mathsf{coopen}\ \}$ and, as it is costumary, we work modulo α-conversion of bound names and variables.

3 Type System

Ambient domains, ranged over by $A - H$, provide the type-level unit of abstraction: in the type system, the effect of exercising a capability is observed on domains rather than on ambients. We define process types in terms of type-level capabilities as follows:

Type Capabilities

$$\mathsf{M} ::= \mathsf{in}\ D \mid \mathsf{coin}\ D \mid \mathsf{out}\ D \mid \mathsf{coout}\ D \mid \mathsf{open}\ D \mid \mathsf{coopen}\ D$$

Process Types

$$\mathsf{P} ::= (\mathbf{L}, \mathbf{M}, \mathbf{N}) \qquad\qquad (\mathbf{L}, \mathbf{M}, \mathbf{N} \in 2^{\mathsf{M}})$$

[2] We use this definition of structural reduction instead of the more standard $P' \equiv P \rightarrow Q \equiv Q' \Rightarrow P' \rightarrow Q'$ to ease the proof of type safety (see Section 4).

Notation. The following conventions are used throughout. We often write $\mathsf{cap}\ D$ (respectively, $\mathsf{cap}\ a$) to denote an arbitrary type-level (respectively, term-level) capability. If $\mathsf{P} = (\mathbf{L}, \mathbf{M}, \mathbf{N})$, we write $\mathsf{P}^{\uparrow}$ for $\mathbf{L}$, $\mathsf{P}^{=}$ for $\mathbf{M}$, and $\mathsf{P}^{\downarrow}$ for $\mathbf{N}$, and often abuse this notation using $\mathsf{P}^{\uparrow}$, $\mathsf{P}^{=}$ and $\mathsf{P}^{\downarrow}$ both as projections of the type P, and directly as sets, as in $P : (\mathsf{P}^{\uparrow}, \mathsf{P}^{=}, \mathsf{P}^{\downarrow})$. Also, we use set-theoretic notation for various operations on process types: if P and Q are process types $\mathsf{P} \subseteq \mathsf{Q}$ denotes component-wise inclusion. Similarly $\mathsf{P} \cup \mathsf{Q}$ denotes component-wise union. Given a set $\mathbf{M}$ of type capabilities and a process type P, we define $\mathsf{P} \cup^{\downarrow} \mathbf{M}$ (respectively, $\mathsf{P} \cup^{=} \mathbf{M}$ and $\mathsf{P} \cup {}^{\uparrow}\mathbf{M}$) as the process type resulting from the union of $\mathbf{M}$ and $\mathsf{P}^{\downarrow}$ (respectively, $\mathsf{P}^{=}$ and $\mathsf{P}^{\uparrow}$): $\mathsf{P} \cup^{\downarrow} \mathbf{M} \triangleq (\mathsf{P}^{\uparrow}, \mathsf{P}^{=}, \mathsf{P}^{\downarrow} \cup \mathbf{M})$, $\mathsf{P} \cup^{=} \mathbf{M} \triangleq (\mathsf{P}^{\uparrow}, \mathsf{P}^{=} \cup \mathbf{M}, \mathsf{P}^{\downarrow})$ and $\mathsf{P} \cup {}^{\uparrow}\mathbf{M} \triangleq (\mathsf{P}^{\uparrow} \cup \mathbf{M}, \mathsf{P}^{=}, \mathsf{P}^{\downarrow})$. Finally, given a type-level capability M, a type-level co-capability $\overline{\mathsf{M}}$, and two sets of type capabilities $\mathbf{L}$ and $\mathbf{M}$, we write $\mathsf{M} \in \mathsf{sync}(\mathbf{L}, \mathbf{M})$ as a shorthand for $\mathsf{M} \in \mathbf{L}$ and $\overline{\mathsf{M}} \in \mathbf{M}$. Finally, the set of free domains (names) of a process type is defined as follows: $\mathit{fn}(\mathsf{P}) = \{D \mid \exists\, \eta \in \{\uparrow, =, \downarrow\}$ such that $\mathsf{cap}\, D \in \mathsf{P}^{\eta}\}$. □

Process types describe the capabilities that processes may exercise, and trace the *nesting level* at which the effect of exercising a capability may be observed. The three components of process types identify those levels: if P has type P, then $\mathsf{P}^{\uparrow}$ describes the effects that can be observed at the level of the ambient enclosing P, $\mathsf{P}^{=}$ describes the capabilities observed at the level of P, and finally, $\mathsf{P}^{\downarrow}$ represents the capabilities that are exercised *within* P, whenever P is an ambient of the form $a[P']$. To exemplify, given $a : A$:

- $\mathsf{in}\ a.P : \mathsf{P} \Rightarrow \mathsf{in}\ A \in \mathsf{P}^{\uparrow}$, since the effect of exercising $\mathsf{in}\ a$ is observed at the level of the ambient (if any) enclosing P
- $b[\mathsf{in}\ a.P] : \mathsf{P} \Rightarrow \mathsf{in}\ A \in \mathsf{P}^{=}$, since now it is $b[\mathsf{in}\ a.P]$ that exercises $\mathsf{in}\ a$
- $\mathsf{open}\ a.P : \mathsf{P} \Rightarrow \mathsf{open}\ A \in \mathsf{P}^{=}$, since $\mathsf{open}\ a$ is exercised (and its effect observed) at the level of the processes running in parallel with $\mathsf{open}\ a.P$
- $b[\mathsf{open}\ a.P] : \mathsf{P} \Rightarrow \mathsf{open}\ A \in \mathsf{P}^{\downarrow}$, since $\mathsf{open}\ a$ is exercised within b.

3.1 Environments and Type Rules

We define two classes of environments, namely *Type Environments*, denoted by E, and *Domain Environments*, denoted by Π:

$$\begin{array}{lcl} \textit{Type Environments} & E & : \ \textit{Ambient Names} \rightarrow \textit{Ambient Domains} \\ \textit{Domain Environments} & \Pi & : \ \textit{Ambient Domains} \rightarrow \textit{Process Types} \end{array}$$

Type environments associate to each ambient name the domain it belongs to, while domain environments associate to each domain the type that is shared by all its ambients. Thus, while type environments partition ambients into domains, domain environments convey information about potential interactions among domains, and enforce behavioral invariants for processes enclosed in ambients in each domain.

Definition 3.1 (Closure and Boundedness). Let Π be a domain environment, P a process type, and D and H be ambient domains. We define the following notation:

$$\begin{array}{lcl} \Pi \vdash \mathsf{P}\ \mathsf{closed} & \triangleq & \mathsf{open}\ H \in \mathsf{sync}(\mathsf{P}^{=}, \Pi(H)^{=}) \Rightarrow \Pi(H) \subseteq \mathsf{P} \\ \Pi \vdash D\ \mathsf{bounds}\ \mathsf{P} & \triangleq & \mathsf{P}^{\uparrow} \subseteq \Pi(D)^{=} \ \wedge\ \mathsf{P}^{=} \subseteq \Pi(D)^{\downarrow} \ \wedge\ (\mathsf{coopen}\ D \in \Pi(D)^{=} \Rightarrow \mathsf{P} \subseteq \Pi(D)) \\ \Pi \vdash D\ \mathsf{closed} & \triangleq & \left\{ \begin{array}{l} \mathsf{in}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{=}) \Rightarrow \Pi \vdash H\ \mathsf{bounds}\ \Pi(D) \\ \mathsf{out}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{\downarrow}) \Rightarrow \Pi(D) \subseteq \Pi(H) \end{array} \right. \end{array}$$

□

The closure condition on process types formalizes the intuition that processes may exercise all the capabilities of the ambients they may open. The boundedness of P by D ensures that the process type $\Pi(D)$ provides a sound approximation of the type P of any process enclosed in (ambients of) domain D. This is expressed by the first two inclusions, which reflect the different nesting level at which one may observe the behavior of ambients and their enclosed processes. The last inclusion handles the case of domains whose ambients may be opened: in that case ambient boundaries are dissolved, and consequently the behavior of the processes unleashed as a result of the open may be observed at the nesting level of the ambients where they were originally enclosed. Finally, the closure condition for domains enforces the previous invariants in the presence of mobility: the behavior of an ambient a of domain D must account for the behavior of ambients entering a, as well as for the behavior of ambients exiting a (since a lets these ambients out, then it is virtually responsible for their behavior). The import of the closure and boundedness conditions is exemplified in Section 3.2 by the typing of Example 1.1 and Example 1.2 from the introduction.

Definition 3.2 (Coherence). Let Π be a domain environment. We define the notation $\Pi \vdash \diamond$ (read Π is *coherent*) as follows:

$$\Pi \vdash \diamond \triangleq \mathit{fn}(\Pi) \subseteq \mathsf{Dom}(\Pi) \ \wedge \ \forall D \in \mathsf{Dom}(\Pi). \ (\Pi \vdash D \ \mathsf{closed} \wedge \Pi \vdash \Pi(D) \ \mathsf{closed})$$

where, with an abuse of notation, we use $\mathit{fn}(\Pi)$ to denote the set $\{D \mid \mathsf{cap}\ D \in \mathsf{Img}(\Pi)\}$. □

The typing rules are given in Figure 1. They derive judgments of the form $\Pi, E \vdash P : \mathsf{P}$, where E is a type environment, Π is a domain environment, and $\mathsf{Img}(E) \subseteq \mathsf{Dom}(\Pi)$ (that is, the *image* of E is contained in the *domain* of Π).

The rules (DEAD), (PAR), (REPL), and (RESTR) are standard. The typing of prefixes (in the (ACTION) rules) is motivated by the observations we made earlier: the effect of exercising the capabilities $\mathsf{in}\ a$, $\mathsf{out}\ a$, $\mathsf{coin}\ a$ and $\mathsf{coopen}\ a$ may be observed at the level of the enclosing ambient. Dually, $\mathsf{open}\ a$, and $\mathsf{coout}\ a$ may be observed at the level of the continuation process.

As for (AMB), the rule stipulates that an ambient $a[P]$ has *at least* the type that Π associates with the domain D of a, i.e. $\Pi(D)$, provided that D bounds the type of P in Π. The (AMB) rule is technically interesting, as, unlike its companion rule in previous type systems for Mobile (and Safe) Ambients, it establishes a precise relationship between the type of an ambient and the process running inside it. This relationship, which is essential for tracing implicit behavior, can be expressed in our type system thanks to the three-level structure of our process types.

The format of the rules (DEAD) and (AMB) could be simplified and made perhaps more intuitive, by stipulating that the types deduced in the consequences of the two rules are the types $(\varnothing, \varnothing, \varnothing)$ and $\Pi(D)$ respectively. More precisely, we could have used the following two rules instead of the respective rules in Figure 1 (note the simpler premises):

$$\text{(DEAD)}\ \frac{\Pi, E \vdash \diamond}{\Pi, E \vdash \mathbf{0} : (\varnothing, \varnothing, \varnothing)} \qquad \text{(AMB)}\ \frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash a : D \quad \Pi \vdash D \ \mathsf{bounds}\ \mathsf{P}}{\Pi, E \vdash a[P] : \Pi(D)}$$

As a matter of facts, these rules are those used for the type inference algorithm defined in Section 5. On the other hand, here this simplification would require the introduction of a subsumption rule like the following one:

$$\frac{\Pi, E \vdash P : \mathsf{P} \qquad \Pi \vdash \mathsf{Q} \qquad \mathsf{P} \subseteq \mathsf{Q}}{\Pi, E \vdash P : \mathsf{Q}}$$

(TYPE PROC)

$$\frac{\Pi \vdash \diamond \quad \mathit{fn}(\mathsf{P}) \subseteq \mathsf{Dom}(\Pi) \quad \Pi \vdash \mathsf{P}\ \mathsf{closed}}{\Pi \vdash \mathsf{P}}$$

(ENV)

$$\frac{\Pi \vdash \diamond \quad \mathsf{Img}(E) \subseteq \mathsf{Dom}(\Pi)}{\Pi, E \vdash \diamond}$$

(NAME)

$$\frac{\Pi, E \vdash \diamond \quad a \in \mathsf{Dom}(E)}{\Pi, E \vdash a : E(a)}$$

(DEAD)

$$\frac{\Pi, E \vdash \diamond \quad \Pi \vdash \mathsf{P}}{\Pi, E \vdash \mathbf{0} : \mathsf{P}}$$

(PAR)

$$\frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash Q : \mathsf{P}}{\Pi, E \vdash P \mid Q : \mathsf{P}}$$

(REPL)

$$\frac{\Pi, E \vdash P : \mathsf{P}}{\Pi, E \vdash !P : \mathsf{P}}$$

(RESTR)

$$\frac{\Pi, E, a : D \vdash P : \mathsf{P} \quad D \in \mathsf{Dom}(\Pi) \quad a \notin \mathsf{Dom}(E)}{\Pi, E \vdash (\boldsymbol{\nu} a{:}D)P : \mathsf{P}}$$

(ACTION$^\uparrow$)

$$\frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash a{:}D \quad \mathsf{cap}\ D \in \mathsf{P}^\uparrow}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ \mathsf{cap} \in \{\mathsf{in}, \mathsf{coin}, \mathsf{out}, \mathsf{coopen}\ \}$$

(ACTION$^=$)

$$\frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash a{:}D \quad \mathsf{cap}\ D \in \mathsf{P}^=}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ \mathsf{cap} \in \{\mathsf{coout}, \mathsf{open}\ \}$$

(AMB)

$$\frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash a : D \quad \Pi \vdash D\ \mathsf{bounds}\ \mathsf{P} \quad \Pi \vdash \mathsf{Q} \quad \Pi(D) \subseteq \mathsf{Q}}{\Pi, E \vdash a[P] : \mathsf{Q}}$$

Figure 1: Typing Rules

which is easily shown to be admissible in the type system, as presently defined (Lemma A.7).

We conclude with the statement of the subject reduction theorem, whose proof in Appendix A.

Theorem 3.3 (Subject Reduction). *If* $\Pi, E \vdash P : \mathsf{P}$ *and* $P \rightarrow Q$, *then* $\Pi, E \vdash Q : \mathsf{P}$. □

3.2 Examples

We illustrate the behavior of the typing rules with the two systems of Examples 1.1 and 1.2. Assume $E \equiv a{:}A, b{:}B, c{:}C, d{:}D$, and consider the attack

$$a[\mathsf{coin}\ a.\mathsf{open}\ b.\mathsf{in}\ c] \mid b[\mathsf{in}\ a.\mathsf{coopen}\ b.\mathsf{in}\ d].$$

Let P_b be the type of the process enclosed in b: it is easy to verify that $\{\mathsf{coopen}\ B, \mathsf{in}\ D\} \subseteq {\mathsf{P}_b}^\uparrow$. From $\Pi \vdash B\ \mathsf{bounds}\ \mathsf{P}_b$, one has $\mathsf{coopen}\ B \in \Pi(B)^=$, and hence $\mathsf{in}\ D \in \Pi(B)^\uparrow$. Let now P_a be the type of the process enclosed in a. Since $\mathsf{open}\ B \in \mathsf{sync}(\mathsf{P}_a^=, \Pi(B)^=)$, then a consequence of the closure of P_a is that $\Pi(B)^\uparrow \subseteq {\mathsf{P}_a}^\uparrow \subseteq \Pi(A)^=$ (the last inclusion holds because $\Pi \vdash A\ \mathsf{bounds}\ \mathsf{P}_a$). Hence $\mathsf{in}\ D \in \Pi(A)^=$ and the attack is detected.

A similar analysis applies to the attack

$$a[\mathsf{coin}\ a.\mathsf{in}\ c.\mathsf{coout}\ a] \mid b[\mathsf{in}\ a.\mathsf{out}\ a.\mathsf{in}\ d].$$

Here $\mathsf{in}\ D \in \Pi(A)^{=}$ results from $\mathsf{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$, which implies $\Pi(B) \subseteq \Pi(A)$ by closure.

4 Type Safety

The operational import of the type system is established by showing that process types provide a safe approximation of process behavior. In that direction, we introduce the relation $P \Downarrow \alpha^{\eta}$ that defines the behavior of a process P in terms of the capabilities α that P may exercise (at nesting level $\eta \in \{\uparrow, =, \downarrow\}$) while evolving in a context. Then we connect the type system with this notion of process behavior by means of a safety result stating that, given a well-typed process P in a well-typed context, for every α such that $P \Downarrow \alpha^{\eta}$, the type capability corresponding to α is traced by the type of P: in other words, no action goes untraced by the type system.

Below, we focus on a simplified case of type safety, one that assumes that processes are "normalized" to the form $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ where P contains no restriction $\boldsymbol{\nu}$. This assumption simplifies the statement and the proof of the type safety theorem: in Appendix D we show how the result can be generalized to arbitrary processes.

We start by introducing a relation of "immediate exhibition", noted $P \downarrow \alpha^{\eta}$: the relation is defined in Figure 2 by induction on the structure of the process P. Next we define a tagging mechanism for processes, by a technique similar to the one in [18]. Let us start with giving the intuition first. Given a process P, we consider its syntax tree and tag some of its nodes with the symbol $\sharp$. So for example, if P is the process $P_1 \mid a[P_2 \mid (\boldsymbol{\nu}b{:}B)P_3]$ then, say, $P_1 \mid \sharp a[P_2 \mid (\boldsymbol{\nu}b{:}B)\sharp P_3]$ denotes the process P in which we tagged the ambient a and the subprocess P_3 occurring therein.

$$\frac{\alpha \in \{\mathsf{in}\ a, \mathsf{out}\ a, \mathsf{coin}\ a, \mathsf{coopen}\ a\}}{\alpha.P \downarrow \alpha^{\uparrow}} \qquad \frac{\alpha \in \{\mathsf{open}\ a, \mathsf{coout}\ a\}}{\alpha.P \downarrow \alpha^{=}} \qquad \frac{P_i \downarrow \alpha^{\eta}}{P_1 \mid P_2 \downarrow \alpha^{\eta}}\ (i = 1, 2)$$

$$\frac{P \downarrow \alpha^{\eta}}{!P \downarrow \alpha^{\eta}} \qquad \frac{P \downarrow \mathsf{cap}\ b^{\eta}}{(\boldsymbol{\nu}a{:}D)P \downarrow \mathsf{cap}\ b^{\eta}}\ a \neq b \qquad \frac{P \downarrow \alpha^{\uparrow}}{a[P] \downarrow \alpha^{=}} \qquad \frac{P \downarrow \alpha^{=}}{a[P] \downarrow \alpha^{\downarrow}}$$

Figure 2: Exhibiting a capability

Having tagged a particular occurrence of P, we instrument reduction so that every process interacting with this occurrence gets tagged: if the tag is initially applied to an ambient, this technique allows us to trace all the processes that "got in touch" with that ambient [3]. Tags are propagated based on the idea of an ambient as a paint pot: any ambient exiting a tagged ambient is tagged:

$$\sharp a[\ b[\mathsf{out}\ a.P \mid Q] \mid \mathsf{coout}\ a.R \mid S\ ] \ \rightarrowtail\ \sharp b[P \mid Q] \mid \sharp a[R \mid S]$$

[3] This corresponds to tracing the interactions considered in the Chinese Wall Security Policy [2]

and so is every process unleashed by opening a tagged ambient:

$$\mathsf{open}\ a.P \mid \sharp a[\mathsf{coopen}\ a.Q \mid R] \ \rightarrowtail\ P \mid \sharp(Q \mid R).$$

Following the intuition that a process exercises all the capabilities of the processes it opens, we also have:

$$\sharp\mathsf{open}\ a.P \mid a[\mathsf{coopen}\ a.Q \mid R] \ \rightarrowtail\ \sharp(P \mid Q \mid R).$$

Technically, the definition is only slightly more complex. First we define *tagged processes*:

Tagged Processes

$$P ::= \mathbf{0} \ \big|\ \alpha.P \ \big|\ P \mid P \ \big|\ a[P] \ \big|\ !P \ \big|\ \sharp P$$

We use the convention that $\sharp$ bounds more than the parrallel composition. Therefore $\sharp P \mid Q$ denotes $(\sharp P) \mid Q$ (and, of course, $\sharp a[P]$ and $\sharp\alpha.P$ respectively denote $\sharp(a[P])$ and $\sharp(\alpha.P)$). We call *untagged processes* those processes in which no tag occurs.

Second, we need to extend structural congruence to tagged processes. Given our assumption that processes are in "normal" form, structural congruence is extended to tagged processes by simply adding the following additional clauses[4]:

$$\sharp\mathbf{0} \equiv \mathbf{0} \qquad \sharp(P \mid Q) \equiv \sharp P \mid \sharp Q \qquad \sharp\,!P \equiv !\,\sharp P$$

The structural congruence relation on tagged processes is then the smallest congruence on tagged processes that is a commutative monoid for **0** and $\mid$ and is closed under the rules above and those of Section 2.

Third, we define the reduction rules for all possible cases that result from whether the processes involved in a reduction step are tagged or not. To ease the definition, we indicate with $\sharp^\circ$ a possibly absent tag, and with $\sharp_i$ the i-th occurrence of the tag $\sharp$. With this notation, the tagged version of reduction is defined by the rules in Figure 3 plus the rules (context) and (struct) of Section 2.

$$\begin{array}{lrcl}
\textbf{(in)} & \sharp^\circ_1 b[\,\sharp^\circ_2\mathsf{in}\ \ a.P \mid Q] \mid \sharp^\circ_3 a[\,\sharp^\circ_4\mathsf{coin}\ \ a.R \mid S] & \rightarrowtail & \sharp^\circ_3 a[\,\sharp^\circ_4 R \mid S \mid \sharp^\circ_1 b[\sharp^\circ_2 P \mid Q]] \\
\textbf{(out)} & a[\,\sharp^\circ_1 b[\,\sharp^\circ_2\mathsf{out}\ \ a.P \mid Q] \mid \sharp^\circ_3\mathsf{coout}\ \ a.R \mid S] & \rightarrowtail & \sharp^\circ_1 b[\,\sharp^\circ_2 P \mid Q] \mid a[\sharp^\circ_3 R \mid S] \\
\textbf{(open)} & \sharp^\circ_1\mathsf{open}\ \ a.P \mid a[\,\sharp^\circ_2\mathsf{coopen}\ \ a.Q \mid R] & \rightarrowtail & \sharp^\circ_1(P \mid \sharp^\circ_2 Q \mid R) \\
\textbf{(out tag)} & \sharp a[\,\sharp^\circ_1 b[\,\sharp^\circ_2\mathsf{out}\ \ a.P \mid Q] \mid \sharp^\circ_3\mathsf{coout}\ \ a.R \mid S] & \rightarrowtail & \sharp b[\,\sharp^\circ_2 P \mid Q] \mid \sharp a[\,\sharp^\circ_3 R \mid S] \\
\textbf{(open tag)} & \sharp^\circ_1\mathsf{open}\ \ a.P \mid \sharp a[\,\sharp^\circ_2\mathsf{coopen}\ \ a.Q \mid R] & \rightarrowtail & \sharp^\circ_1 P \mid \sharp(Q \mid R)
\end{array}$$

Figure 3: Tag propagation via reduction

Now we can give a precise definition of the *residuals* of a process evolving in a context: intuitively these are all the tagged processes that result from tagging the process in question, and reducing it in the given context. The definition relies on the following notion of (restriction-free) context:

$$\mathscr{C}[\,] \ \ ::= \ \ [\,] \ \big|\ P \mid \mathscr{C}[\,] \ \big|\ \mathscr{C}[\,] \mid P \ \big|\ a[\mathscr{C}[\,]] \ \big|\ \alpha.\mathscr{C}[\,]$$

Definition 4.1 (Residuals). Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a process, with P containing no restrictions.

[4] In Appendix D the definition is refined to handle restrictions and scope extrusion.

1. An *occurrence* of P is a path Δ in the syntax tree of P. We denote with P_Δ the subprocess of P occurring at Δ, and with $\mathscr{C}^P_\Delta[\,]$ the context obtained from P by substituting a hole for the subprocess occurring at Δ. Hence $P = \mathscr{C}^P_\Delta[P_\Delta]$.
2. Given a tagged process P, we denote by $|P|$ the process obtained by erasing[5] all tags occurring in P.
3. Let Δ be an occurrence of an untagged process P. The set of *residuals of* Δ *in* P is defined as follows:
   (1) P_Δ is a residual of Δ in P
   (2) If $\mathscr{C}^P_\Delta[\sharp P_\Delta] \rightarrowtail Q$ and $Q_{\Delta'}$ is tagged (that is, $Q_{\Delta'} = \sharp R$ for some R), then every residual of Δ' in $|Q|$ is also a residual of Δ in P. □

We can finally generalize the notion of capability exhibition to process occurrences.

Definition 4.2 (Residual Behavior). Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a process, with P containing no restriction, and Δ be an occurrence of P. Then, $\Delta \Downarrow \alpha^\eta$ if and only if there exists a residual R of Δ in P such that $R \downarrow \alpha^\eta$. □

The definitions above hide several subtilities. First of all note that the definition of structural equivalence for tagged processes allows us to use the informal notation $\sharp(P \mid Q \mid R)$, to denote either $\sharp(P \mid (Q \mid R))$ or $\sharp((P \mid Q) \mid R)$. The choice of one of the two is not important as both of them are structurally equivalent to all the possible distributions of the tags over the subterms, such as in $\sharp(P \mid Q) \mid \sharp R$ or in $\sharp P \mid (\sharp Q \mid \sharp R)$. This for example is used in the following instance of the rule **(open)**:

$$\sharp\mathsf{open}\ \ a.P \mid a[\,\mathsf{coopen}\ \ a.Q \mid R] \quad \rightarrowtail \quad \sharp(P \mid Q \mid R)$$

However it is important to notice that according to Definition 4.1 while the process $(P \mid Q \mid R)$ is a residual of $\mathsf{open}\ \ a.P$ in the redex, but neither $(P \mid Q)$, nor P, nor Q, nor R are residuals of that occurence. The reason resides in the reduction rule **(struct)** in Section 2 which applies structural equivalence to the redex but not to the reductum. Even though P or Q are not residuals of $\mathsf{open}\ \ a.P$ their behavior is included in the behavior of $(P \mid Q \mid R)$. This holds thanks to the Definition 4.2 and the rules in Figure 2. We gave our definitions as such, since we wanted that in a one step reduction every tagged process had at most one residual, so that to be able to easily follow in the proofs the behavior of the residuals. In order not to loose any behavior, we defined our reduction rules in Figure 3 so that in the reductum we tagged the "most general" residual. This explains why, say, $\mathsf{open}\ \ a.P \mid \sharp a[\mathsf{coopen}\ \ a.Q \mid R]$ reduces *in one step* to $P \mid \sharp(Q \mid R)$ rather than to $P \mid \sharp Q \mid \sharp R$.

The last step consists in using this definition to state the type safety theorem, whose proof is given in Appendix B.

Theorem 4.3 (Type Safety). *Let* $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ *be a process, with* P *containing no restriction,* Δ *be an occurrence of* P *and let* $E = E', \vec{a}{:}\vec{D}$ *for a type environment* E'. *Assume that* $\Pi, E \vdash P : \mathsf{P}'$ *and* $\Pi, E \vdash P_\Delta : \mathsf{P}$. *If* $\Delta \Downarrow (\mathsf{cap}\ a)^\eta$, *then* $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$. □

To exemplify, consider the ambient $a[\mathsf{coin}\ a.\mathsf{open}\ b]$. If taken in isolation, this ambient only exhibits the capabilities $\mathsf{coin}\ a$ and $\mathsf{open}\ b$. If, instead, we take the parallel composition

$$a[\mathsf{coin}\ a.\mathsf{open}\ b] \mid b[\mathsf{in}\ a.\mathsf{coopen}\ b.\mathsf{in}\ c] \qquad (1)$$

[5] Technically, tags are annotations on the syntax tree and are not part of the syntax. Thus the notion of occurrence is preserved by tagging/untagging, that is, for every tagged P and occurrence Δ, $|P_\Delta| = |P|_\Delta$.

then the ambient $a[\dots]$ also exhibits $\mathsf{in}\ c$ as a result of the interaction with the context. In fact, if we start tagging $a[\dots]$ in (1) above, the result of tagged reduction is as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{\sharp a[\mathsf{coin}\ a.\mathsf{open}\ b] \mid b[\mathsf{in}\ a.\mathsf{coopen}\ b.\mathsf{in}\ c]} \\ \rightarrowtail & \sharp a[\mathsf{open}\ b \mid b[\mathsf{coopen}\ b.\mathsf{in}\ c]] \\ \rightarrowtail & \sharp a[\mathsf{in}\ c] \end{array}$$

Now, Theorem 4.3 ensures that if we type the process (1), the fact that the residual $a[\mathsf{in}\ c]$ of a exhibits $\mathsf{in}\ c$ is traced by the type associated to the domain of a. In fact, the result is even stronger, as it ensures that the type system traces the behavior of any process that interacts with the process occurrence of interest. For example, if we take the composition $\sharp a[\mathsf{coin}\ a.\mathsf{coout}\ b] \mid b[\mathsf{in}\ a.\mathsf{out}\ a.\mathsf{in}\ c]$, the result of tagged reduction is $\sharp a[\,] \mid \sharp b[\mathsf{in}\ c]$, and Theorem 4.3 ensures that the type of (the domain of) a traces the type-level capability corresponding to $\mathsf{in}\ c$, since it is exhibited by the residual $b[\mathsf{in}\ c]$.

5 Algorithmic Systems

The type system given in Figure 1 is not algorithmic as the rules (DEAD) and (AMB) are not syntax-directed. However, it is easy to state the type rules so that they form an algorithmic system.

5.1 Typing Algorithm

The algorithmic type system finds the minimal type of a term under a given set of domain assumptions Π and type assumptions E. The system results from the type system of Figure 1 by replacing the rules (DEAD) and (AMB) by those stated at the end of Section 3.1, and replacing the (ACTION) and (PAR) with the rules in Figure 4: the only subtlety is the side condition to the rule ($\textsc{Action}_2^=$), which defines P' as the minimum P' that contains P and is closed in Π (i.e. such that $\Pi \vdash \mathsf{P}'$ is derivable). Collectively, the new rules constitute the core of an algorithm that given $\Pi,\ E$, and P as input, returns the type P as output. The side condition to the rule ($\textsc{Action}_2^=$) uses the following closure operator for process types.

Definition 5.1 (Process Type Closure). Let Π be a domain environment such that $fn(\Pi) \subseteq \mathsf{Dom}(\Pi)$. Then define $\mathsf{ProcClosure}(\mathsf{P},\Pi) \triangleq \bigcap\{\mathsf{P}' \mid \mathsf{P}' \supseteq \mathsf{P} \cup \Pi(A)$for all A such that $\mathsf{open}\,A \in \mathsf{sync}(\mathsf{P}'^{=}, \Pi(A)^{=})\}$ □

Theorem 5.2 (Soundness and completeness). *If* $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$, *then* $\Pi, E \vdash P : \mathsf{P}$. *Conversely, if* $\Pi, E \vdash P : \mathsf{P}$, *then* $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}'$ *and* $\mathsf{P}' \subseteq \mathsf{P}$

Proof sketch. To prove soundness one first has to prove that if $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$, then $\Pi \vdash \mathsf{P}$. The only non-trivial part of this is to prove that $\Pi \vdash \mathsf{P}\ \mathsf{closed}$. The latter can be proved by induction on the deduction of $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$ and a case analysis on the last applied rule: the result holds when the last rule is ($\textsc{Action}_1^=$) and ($\textsc{Action}^{\uparrow}$) since the added capability types do not interfere with type process closure; it holds for ($\textsc{Action}_2^=$) by the very definition of P' which effectively closes P; it is a consequence of Proposition A.1(5) when the last rule is (PAR). Then, soundness can be easily proved by induction, by repeated use of the subsumption admissibility property we prove in Lemma A.7. Completeness also follows by induction on the derivation of $\Pi, E \vdash P : \mathsf{P}$. □

Corollary 5.3 (Minimal typing). $\Pi, E \vdash_{\mathscr{A}} P : \min\{\mathsf{P} \mid \Pi, E \vdash P\colon \mathsf{P}\}$ *if this set is non-empty.* □

(PAR)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad \Pi, E \vdash_{\mathscr{A}} Q : \mathsf{Q}}{\Pi, E \vdash_{\mathscr{A}} P \mid Q : \mathsf{P} \cup \mathsf{Q}}$$

(ACTION$_1^=$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{coout}\ \ a.P : \mathsf{P} \cup^{=} \{\mathsf{coout}\ \ A\}}$$

(ACTION$_2^=$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{open}\ a.P : \mathsf{P}'} \qquad \mathsf{P}' \triangleq \mathsf{ProcClosure}(\mathsf{P} \cup^{=} \{\mathsf{open}\ A\}, \Pi)$$

(ACTION$^{\uparrow}$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{cap}\ a.P : \mathsf{P} \cup^{\uparrow} \{\mathsf{cap}\ A\}} \qquad \mathsf{cap} \in \{\mathsf{in}\ , \mathsf{coin}\ , \mathsf{out}\ , \mathsf{coopen}\ \}$$

(DEAD)

$$\frac{\Pi, E \vdash_{\mathscr{A}} \diamond}{\Pi, E \vdash_{\mathscr{A}} \mathbf{0} : (\varnothing, \varnothing, \varnothing)}$$

(AMB)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad \Pi, E \vdash a : D \quad \Pi \vdash D \ \mathsf{bounds}\ \mathsf{P}}{\Pi, E \vdash_{\mathscr{A}} a[P] : \Pi(D)}$$

Figure 4: Algorithmic Typing

The existence of minimum (with respect to point-wise set containment) types and of an algorithm computing them are interesting and useful properties. Yet, leaving a programmer with the task of providing a domain environment Π as input to the type checking algorithm is a very strong requirement. Below, we show that this task can be dispensed with, as domain environments can be reconstructed automatically. In principle, providing a coherent Π for which the typing algorithm does not fail is straightforward. Given a process P, let $\mathscr{D}$ be the set of domain names occurring in P, and let E be a type environment that assigns a domain in $\mathscr{D}$ to every name in P. Now, denote by P^{sat} the process type whose components contain all the possible type capabilities over $\mathscr{D}$, and let Π^{sat} be the *saturated* type environment such that $\mathsf{Dom}(\Pi) = \mathscr{D}$ and $\Pi^{sat}(D) = \mathsf{P}^{sat}$ for all $D \in \mathsf{Dom}(\Pi)$. It is easy to verify that there always exists a process type P such that $\Pi^{sat} \vdash P : \mathsf{P}$ is derivable: to see that, observe that $\Pi^{sat}(D)$ provides a sound approximation of the behavior of every ambient (and process) occurring in P (indeed, $\Pi^{sat} \vdash P : \mathsf{P}^{sat}$ holds). On the other hand, it is also clear that Π^{sat} is not very useful as a domain environment, as it provides the coarsest possible approximation of behavior: this is problematic in view of our perspective use of types to check and enforce security, as the coarser the approximation of a process' behavior, the less likely for the process to pass the security checks imposed by its environment.

5.2 Type Reconstruction

Type reconstruction computes the minimum coherent domain environment Π such that a given term type checks. The ordering over environments derives by extending the containment relation to environments, using point-wise ordering as follows: $\Pi \subseteq \Pi'$ if and only if $\mathsf{Dom}(\Pi) = \mathsf{Dom}(\Pi')$ and for all $D \in \mathsf{Dom}(\Pi), \Pi(D) \subseteq \Pi'(D)$. We use $\bigcap\{a \mid \mathscr{P}(a)\}$ as a shorthand for $\bigcap_{e \in \{a \mid \mathscr{P}(a)\}} e$, and similarly for the union. Then we have the following definition.

Definition 5.4 (Domain Closures). Let Π be a domain environment s.t. $\mathit{fn}(\Pi) \subseteq \mathsf{Dom}(\Pi)$. Then define:

$$\mathsf{EnvClosure}(\Pi) \triangleq \bigcap \{\Pi' \mid \Pi' \supseteq \Pi\,,\ \Pi' \vdash \diamond\}$$

$$\mathsf{DomClosure}(\mathsf{P}, A, \Pi) \triangleq \bigcap \left\{\Pi' \mid \Pi' \supseteq \Pi\ , \Pi'(A) \supseteq (\varnothing, \mathsf{P}^{\uparrow}, \mathsf{P}^{=}) \cup \mathsf{P}'\right\} \text{ with } \mathsf{P}' = \begin{cases} \mathsf{P} & \text{if } \mathsf{coopen}\ A \in \Pi'(A)^{=} \\ \varnothing & \text{otherwise} \end{cases}$$

□

Both these operators, as well as the $\mathsf{ProcClosure}$ operator of Definition 5.1, are easily seen to be well-defined and monotone: furthermore they can be effectively computed by (always terminating) algorithms. An example is given in Figure 5.

```
EnvClosure($\Pi$ : **DomEnv**):**DomEnv** :=
    $\mathscr{D}$ := Dom($\Pi$);
    **while** $\mathscr{D} \neq \varnothing$ **do**
        **choose** $D$ **in** $\mathscr{D}$; $\mathscr{D} := \mathscr{D} \setminus \{D\}$
        **for** cap $H$ **in** $\Pi(D)^{=}$ **do**
            $\Pi' := \Pi$
            **case** cap **of**
              out :
                **if** coout $H \in \Pi(H)^{\downarrow}$ **then** $\Pi(H) := \Pi(H) \cup \Pi(D)$
              in :
                **if** coin $H \in \Pi(H)^{=}$ **then**
                    **begin**
                        $\Pi(H)^{=} := \Pi(H)^{=} \cup \Pi(D)^{\uparrow}$; $\Pi(H)^{\downarrow} := \Pi(H)^{\downarrow} \cup \Pi(D)^{=}$
                        **if** coopen $H \in \Pi(H)^{=}$ **then** $\Pi(H) := \Pi(H) \cup \Pi(D)$
                    **end**
              open :
                **if** coopen $H \in \Pi(H)^{=}$ **then** $\Pi(D) := \Pi(D) \cup \Pi(H)$
            **esac**
            **if** $\Pi \neq \Pi'$ **then** $\mathscr{D} := \mathscr{D} \cup \{H\}$
        **done**
    **done**
    **return**($\Pi$)
```

Figure 5: A closure algorithm

The system for type reconstruction is defined in Figure 6: the (**R**-ACTION) rules are the same as the corresponding algorithmic (ACTION) rules, (**R**-REPL) and (**R**-RESTR) are defined as their corresponding rules in Figure 1. In all the rules, the subscript $\mathscr{D}$ indicates a finite set of ambient domains: in (**R**-DEAD), $\varnothing_{\mathscr{D}}$ is the domain environment defined by $\varnothing_{\mathscr{D}}(D) = (\varnothing, \varnothing, \varnothing)$ for every $D \in \mathscr{D}$. The rules describe an algorithm that, given a process P and a type environment E such that $\mathit{fn}(P) \subseteq \mathsf{Dom}(E)$ returns a process type P and a domain environment Π. More precisely, given a process P and a type environment E, let $\mathscr{D}$ be the set of ambient domains occurring in the type assumptions of E and in the types of restrictions in P. Then, there exists one and only one process type P and environment Π such that $\Pi, E \vdash_{\mathscr{D}} P : \mathsf{P}$: we denote this process type and domain environment respectively with $\mathscr{R}_{\mathsf{type}}(E, P)$ and $\mathscr{R}_{\mathsf{env}}(E, P)$.

Theorem 5.5 (Soundness and Completeness). *Let P be a process, and E a type environment such that $\mathit{fn}(P) \subseteq \mathsf{Dom}(E)$. Then $\mathscr{R}_{\mathsf{env}}(E, P), E \vdash_{\mathscr{A}} P : \mathscr{R}_{\mathsf{type}}(E, P)$. Furthermore, for any Π and P such that $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$, one has $\mathscr{R}_{\mathsf{env}}(E, P) \subseteq \Pi$ and $\mathscr{R}_{\mathsf{type}}(E, P) \subseteq \mathsf{P}$.*

(**R**-Dead)

$$\frac{}{\varnothing_{\mathscr{D}}, E \vdash_{\mathscr{D}} \mathbf{0} : (\varnothing, \varnothing, \varnothing)}$$

(**R**-Action$_1^=$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{coout}\ \ a.P : \mathsf{P} \cup^{=} \{\mathsf{coout}\ \ A\}}$$

(**R**-Action$_2^=$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{open}\ a.P : \mathsf{P}'} \qquad \mathsf{P}' \triangleq \mathsf{ProcClosure}(\mathsf{P} \cup^{=} \{\mathsf{open}\ A\}, \Pi)$$

(**R**-Action$^{\uparrow}$)

$$\frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{cap}\ a.P : \mathsf{P} \cup {}^{\uparrow}\{\mathsf{cap}\ A\}} \qquad \mathsf{cap} \in \{\mathsf{in}\ , \mathsf{coin}\ , \mathsf{out}\ , \mathsf{coopen}\ \}$$

(**R**-Par)

$$\frac{\Pi_1, E \vdash_{\mathscr{D}} P_1 : \mathsf{P}_1 \quad \Pi_2, E \vdash_{\mathscr{D}} P_2 : \mathsf{P}_2}{\Pi, E \vdash_{\mathscr{D}} P_1 \mid P_2 : \mathsf{P}} \qquad \begin{array}{l} \Pi \triangleq \mathsf{EnvClosure}(\Pi_1 \cup \Pi_2), \\ \mathsf{P} \triangleq \mathsf{ProcClosure}((\mathsf{P}_1 \cup \mathsf{P}_2), \Pi) \end{array}$$

(**R**-Amb)

$$\frac{\Pi, E \vdash_{\mathscr{D}} P : \mathsf{P} \quad E(a) = A}{\Pi^{\star}, E \vdash_{\mathscr{D}} a[P] : \Pi^{\star}(A)} \qquad \Pi^{\star} \triangleq \bigcap \left\{ \Pi'' \;\middle|\; \begin{array}{l} \Pi'' = \mathsf{EnvClosure}(\Pi') \\ \Pi' = \mathsf{DomClosure}(\mathsf{P}', A, \Pi) \\ \mathsf{P}' = \mathsf{ProcClosure}(\mathsf{P}, \Pi'') \end{array} \right\}$$

Figure 6: Type Reconstruction Algorithm

Proof. See Appendix C. □

Corollary 5.6 (Minimal typing). *Let P be a process and E a type environment such that fn$(P) \subseteq \mathsf{Dom}(E)$. Then $(\mathscr{R}_{\mathit{env}}(E, P), \mathscr{R}_{\mathit{type}}(E, P)) = \min\{(\Pi, \mathsf{P}) \mid \Pi, E \vdash P : \mathsf{P}\}$* □

Accordingly, in the typed syntax it is enough to specify the domains of the ambients occurring in P: the type checker will then generate the minimal types for each domain and for P.

6 Security

Security policies are expressed by means of security constraints, and new environments help associate security constraints with ambient domains:

Security Environments $\quad \Sigma$: *Ambient Domains* $\rightarrow$ *Security Constraints*

A security environment establishes the security structure for a given system of processes and ambients. Given domain and type environments Π and E, and a well-typed process P, we may then verify that P is secure in Σ by checking that Π satisfies Σ. The definition of satisfaction, denoted $\Pi \models \Sigma$, requires

$\mathsf{Dom}(\Sigma) = \mathsf{Dom}(\Pi)$ and depends on the structure of the security constraints, which in turn depend on the sort of security policy one wishes to express. We discuss three options below.

Domain Constraints yield rather coarse security policies whereby one can identify *trusted* and *untrusted* domains and, for each domain, allow interactions only with trusted domains. These security constraints may be expressed by tables of the form $\mathsf{S} = \langle \mathsf{in} = \mathscr{D}_{\mathsf{in}},\ \mathsf{out} = \mathscr{D}_{\mathsf{out}} \rangle$. If D is a domain and $\Sigma(D) = \mathsf{S}$, then $\mathscr{D}_{\mathsf{in}}$ (respectively, $\mathscr{D}_{\mathsf{out}}$) is the set of trusted domains whose ambients can enter (respectively, exit) the ambients of D. In this option $\Pi \models \Sigma$ if and only if, for all D in $\mathsf{Dom}(\Pi)$, one has

(i) $\{A \mid \mathsf{in}\ D \in \mathsf{sync}(\Pi(A)^{=}, \Pi(D)^{=})\} \subseteq \Sigma(D).\mathsf{in}$, and

(ii) $\{A \mid \mathsf{out}\ D \in \mathsf{sync}(\Pi(A)^{=}, \Pi(D)^{\downarrow})\} \subseteq \Sigma(D).\mathsf{out}$.

The security model arising from domain constraints is related to the security policy of the JDK 1.1.x. In JDK 1.0.x all non local definitions are considered as insecure. The same applies under JDK 1.1.x with the difference that a class loaded from the network can become trusted if it is digitally signed by a party the user has decided to trust (in our case a domain in $\mathscr{D}_{\mathsf{in}}$).

Capability Constraints lead to finer protection policies that identify the type-level capabilities that entering and exiting ambients may exercise[6]. These constraints may be expressed by tables of the form $\mathsf{S} = \langle \mathsf{in} = \mathsf{P}_{\mathsf{in}},\ \mathsf{out} = \mathsf{P}_{\mathsf{out}} \rangle$, whose entries are process types. If D is a domain, and $\Sigma(D) = \mathsf{S}$ then:

- P_{in} defines the only capabilities that processes entering ambients of domain D have permission to exercise: the three sets ${\mathsf{P}_{\mathsf{in}}}^{\uparrow}$, ${\mathsf{P}_{\mathsf{in}}}^{=}$, and ${\mathsf{P}_{\mathsf{in}}}^{\downarrow}$ specify the capabilities that can be exercised, respectively, at the level of the entering process, at the level of the enclosing ambient, and inside the entering process. The first specification is useful to prevent information leakage, the second to control the local interactions of the entering ambient, and the third is useful when opening (or entering) the entered process.
- $\mathsf{P}_{\mathsf{out}}$ is the table defining the capabilities that are granted to processes exiting out of ambients of domain D, with the three entries ${\mathsf{P}_{\mathsf{out}}}^{\uparrow}$, $\mathsf{P}^{=}_{\mathsf{out}}$, and $\mathsf{P}^{\downarrow}_{\mathsf{out}}$ defined as above.

In this option $\Pi \models \Sigma$ if and only if, for all A, B in $\mathsf{Dom}(\Pi)$, $\mathsf{in}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{=})$ implies $\Pi(B) \subseteq \Sigma(A).\mathsf{in}$, and, $\mathsf{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$ implies $\Pi(B) \subseteq \Sigma(A).\mathsf{out}$. Capability constraints are loosely related to the *permission collections* used in the JDK 1.2 architecture (also known as Java 2) to enforce security policies based on access control and stack inspection.

Constraint Formulas. More refined policies can be expressed by resorting to a fragment of first order logic. The fragment is given below, where M ranges over type capabilities, D over ambient domain names (and domain variables), and η over $\uparrow$, $=$, and $\downarrow$.

Syntax

$$\phi \quad ::= \quad \mathsf{M} \in D^{\eta} \ \big|\ \neg\phi \ \big|\ \phi \wedge \phi \ \big|\ \phi \vee \phi \ \big|\ \forall D : \phi$$

[6] Alternatively, we could define what ambients should not be allowed to do, but our choice complies with well-established security principles [8].

Semantics

$$\begin{array}{lcl}
\Pi \models \mathsf{M} \in D^{\eta} & \Leftrightarrow & \mathsf{M} \in \Pi(D)^{\eta} \\
\Pi \models \neg\phi & \Leftrightarrow & \Pi \models \phi \text{ does not hold} \\
\Pi \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \Pi \models \phi_1 \text{ and } \Pi \models \phi_2 \\
\Pi \models \phi_1 \vee \phi_2 & \Leftrightarrow & \Pi \models \phi_1 \text{ or } \Pi \models \phi_2 \\
\Pi \models \forall D : \phi & \Leftrightarrow & \Pi \models \phi\{D := A\} \text{ for all } A \in \mathsf{Dom}(\Pi)
\end{array}$$

The notion of formula satisfiability is easily extended to the security environments, namely $\Pi \models \Sigma$ if an only if for all D in $\mathsf{Dom}(\Pi)$, $\Pi \models \Sigma(D)$. Since we work on finite models, satisfiability is always decidable. Note that the first-order fragment is powerful enough to encode quantification on actions as well as formulas such as $\mathsf{cap}D \in \mathsf{synch}(\mathbf{L}, \mathbf{M})$. Based on that, we can express refined security properties: for example, the formula $\forall B, C : \mathsf{in}\ D \in \mathsf{synch}(B^{=}, D^{=}) \wedge \mathsf{in}\ B \in \mathsf{synch}(C^{=}, B^{=}) \Rightarrow \mathsf{in}\ D \in C^{=}$ allows one to prevent arbitrary nested Trojan Horses (an ambient entering a second ambient that enters a third ambient that can enter D), since it requires that all ambients that are granted the right to enter domain D may only be entered by ambients that already have the right to enter D.

Independently of the structure of constraints, given a process P and a type environment E for the names occurring free in P, we say that E and P satisfy a security policy Σ if and only if $\mathscr{R}_{\mathsf{env}}(E, P) \models \Sigma$. As a corollary of Theorem 4.3 we have that $\mathscr{R}_{\mathsf{env}}(E, P) \models \Sigma$ implies that no ambient occurring in P can violate the security policies defined in Σ.

7 Distributed SSA

The type systems presented in the previous sections have interesting properties and significant operational impact. Yet, there is also a fundamental weakness to them, in that they rely on the assumption that global information is available on ambient domains and their types: a derivation for a typing judgment $\Pi, E \vdash P : \mathsf{P}$ requires that the environments Π and E contain assumptions for all the ambients occurring in P and for all those ambients' domains. This is clearly unrealistic for a foundational calculus for wide-area distributed computations and systems.

In this section we address the problem by presenting a distributed variant of SSA. In the distributed version, which we call DSSA, each ambient (i.e. each "location" in the system of processes) carries a type and a domain environment. The syntax of DSSA processes is defined by the following productions:

Distributed Processes

$$P ::= \mathbf{0} \;\big|\; \alpha.P \;\big|\; (\boldsymbol{\nu} a{:}D)P \;\big|\; P \mid P \;\big|\; a[P]^{\mathsf{S}}_{\Pi,E} \;\big|\; !P$$

where α, Π, and E are defined as in the previous sections, and S is a capability constraint.

To get an intuition of DSSA ambients, it is useful to think of Java `class` files. Class files include applet bytecode together with type and security information used for bytecode verification and dynamic linking. In particular a `class` file declares the types of all methods and fields the associated class defines (the *type assertions*), and the types of all the identifiers the class refers to (the *type assumptions*) [13]. When downloading a class file, the verifier checks (among other properties) that the bytecode satisfies the type assertions under the type assumptions. A DSSA ambient $a[P]^{\mathsf{S}}_{\Pi,E}$ can be understood as a class file, where $a[P]$ represents the bytecode, and the pair Π, E corresponds to the type assertions and assumptions. Intuitively, for any name b occurring in $a[P]$, the process type $\Pi(E(b))$ may be thought of as a type assertion, if $b = a$ or b is the name of an ambient contained in P, or else as a type assumption if b occurs in a capability of P but P contains no ambient named b.

7.1 Typed Reduction

The type system for DSSA is the same as that defined for SSA. DSSA ambients are typed, statically, by simply disregarding their associated environments: the latter are used in the dynamic type-checks performed upon reduction. The new reduction relation is based on structural congruence, which is defined as in Section 2 with the only exception of the following rule:

$$(\boldsymbol{\nu} a{:}D)\, b[P]^{\mathsf{S}}_{\Pi,E,a:D} \equiv b[(\boldsymbol{\nu} a{:}D)P]^{\mathsf{S}}_{\Pi,E} \qquad a \neq b$$

that replaces the corresponding rule for SSA. Typed reduction is then defined by the **(open)**, **(struct)**, and **(context)** rules of Section 2, plus the rules in Figure 7.

(in) $$b[\mathsf{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} \mid a[\mathsf{coin}\ a.R \mid S]^{\mathsf{S}_a}_{\Pi_a,E_a} \ \rightarrow\ a[R \mid S \mid b[P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b}]^{\mathsf{S}_a}_{\Pi,E} \qquad (*)$$

$(*)$ *provided that, given* $\Pi, E = \Pi_b{\cdot}\Pi_a, E_b{\cdot}E_a$, *one has*
$\Pi, E \vdash b[\mathsf{in}\ a.P \mid Q] : \Pi(E(b)),\ \ \Pi(E(b)) \subseteq \mathsf{S}_a.\mathsf{in},$ *and* $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$

(out) $$a[\mathsf{coout}\ a.P \mid Q \mid b[\mathsf{out}\ \ a.R \mid S]^{\mathsf{S}_b}_{\Pi_b,E_b}]^{\mathsf{S}}_{\Pi,E} \ \rightarrow\ b[R \mid S]^{\mathsf{S}_b}_{\Pi_b,E_b} \mid a[P \mid Q]^{\mathsf{S}}_{\Pi,E} \qquad (**)$$

$(**)$ *provided that* $\Pi, E \vdash b[\mathsf{out}\ \ a.R \mid S] : \Pi(E(b)),$ *and* $\Pi(E(b)) \subseteq \mathsf{S.out}$

Figure 7: New reduction rules for DSSA

The notation $\Pi \cdot \Pi'$ indicates the environment that results from appending Π' to Π so that assumptions in Π' hide corresponding assumptions in Π. Hence, in the rules of Figure 7:

$$(\Pi{\cdot}\Pi')(D) \ \triangleq\ \begin{cases} \Pi'(D) & \textit{if } D \in \mathsf{Dom}(\Pi') \\ \Pi(D) & \textit{otherwise} \end{cases} \qquad (E{\cdot}E')(a) \ \triangleq\ \begin{cases} E'(a) & \textit{if } a \in \mathsf{Dom}(E') \\ E(a) & \textit{otherwise} \end{cases}$$

The rule **(in)** extends the corresponding rule for SSA with additional conditions ensuring that the reduction takes place only when the local environments of the two ambients involved in the move are mutually compatible and the security constraints fulfilled. First, the rule requires the environment of a to be extended by the environment of b (in the reductum a carries the environment Π, E that extends Π_a, E_a). Second, the reduction requires the entering ambient b to (i) be well-typed in the extended environments, and (ii) to satisfy the security constraints of a. Finally, the condition $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$ requires that the entering ambient b does not modify the external behavior of a: a lets new ambients in only if they comply with its own local behavior discipline.[7]

The rule **(out)** performs similar type and security checks: note, in particular, that if a were well typed then the type check on b would be unnecessary. Yet, we cannot make any *a priori* assumption about a and its type, and therefore we must check that the exiting ambient has the type it is supposed to have (otherwise the security check would be of no use).

A closer look at the rule **(in)** shows an interesting correspondence between the constraints enforced by the target of the move and the functions implemented by the three component of the JVM security system: the *Class Loader*, the *Bytecode Verifier*, and the *Security Manager* [13].

[7] In the rules we considered that ambients are indexed by Capability Constraints. If S's were instead Domain Constraints the security requirements $\Pi(E(b)) \subseteq \mathsf{S}_a.\mathsf{in}$ and $\Pi(E(b)) \subseteq \mathsf{S.out}$ in **(in)** and **(out)** would change respectively to $E(b) \in \mathsf{S}_a.\mathsf{in}$ and $E(b) \in \mathsf{S.out}$. If instead, the constraints were expressed by formulas, we could consider fine-graded security constraints of the form $\mathsf{S} ::= \langle \mathsf{in} = \phi,\ \mathsf{out} = \phi \rangle$, and the security conditions in **(in)** and **(out)** would change to $\Pi \models \mathsf{S}_a.\mathsf{in}$ and $\Pi \models \mathsf{S.out}$, respectively.

$\Pi\,,\, E = \Pi_b{\cdot}\Pi_a\,,\, E_b{\cdot}E_a$ **:** Local (to a) assumptions on the type of each name hide remote assumptions for that name. As a consequence, the entering agent b cannot spoof a definition of the target host a. This is the security policy implemented by the JVM Class Loader, which provides name-space separation and prevents type-confusion attacks for spoofing.

$b[\mathsf{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} : \Pi(E(b))$ **:** The target of the move, ambient a, checks that the entering agent b has the type it declares to have, in case $b \notin \mathsf{Dom}(E_a)$, or that a expects it to have, when $b \in \mathsf{Dom}(E_a)$. This is the security policy enforced by the bytecode verifier.

$\Pi(E(b)) \subseteq \mathsf{S}_a.\mathsf{in}$ **:** The ambient a checks that the entering agent performs only actions that are explicitly permitted by the security constraints defined by $\mathsf{S}_a.\mathsf{in}$. This is essentially the security policy enforced by the Security Manager: the difference is that the Security Manager performs these checks dynamically (when the agent is already entered and requires to perform the action), whereas in our system they are performed at load time.

Note that, intuitively, all the above checks are performed by a, the ambient whose boundary is crossed. That ambient does not trust foreign code, it just trusts, of course, its own implementation of the type checking algorithm which is used to dynamically verify foreign code: verification is based on the (type) information foreign code carries along with it, according to the common proof-carrying-code practice [14].

7.2 Type Safety

Most of the properties relating the type system and reduction carry over from SSA to DSSA. However the key property of DSSA, where the essence of distribution resides, is the following, stronger, version of Theorem 4.3. Again, the theorem is stated for the simplified case of "normalized" distributed processes, i.e. for processes with all restrictions extruded to the outermost scope. It is based on the same definitions of residual and exhibition of the previous section (but stated for the new typed reduction): the additional information attached to ambients is simply disregarded.

Theorem 7.1 (Local Type Safety). *Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a DSSA process, with P containing no restriction, and Δ be an occurrence of P such that $P_\Delta = a[Q]^{\mathsf{S}}_{\Pi,E}$. Assume $\Pi, E \vdash P_\Delta : \mathsf{P}$ is derivable. If $\Delta \Downarrow (\mathsf{cap}\ b)^\eta$, then $\mathsf{cap}\ E(b) \in \mathsf{P}^\eta$.* □

The difference between this theorem, whose proof is sketched in Theorem B.5, and Theorem 4.3 is that the statement of the former does not require the context P to be well typed, but just that the ambient occurrence can be typed under the assumptions it comes with. Accordingly, every ambient that type-checks under the environment it carries along with it will only exhibit capabilities that are already in its static type, even though the context it interacts with is not well-typed[8].

This is an interesting result for wide-area distributed systems, where global typing may not be possible: for example, distinct subsystems may have incompatible type assumptions. Even then, typed reduction allows secure interactions provided that local type safety exists or can be ensured. Hence, an agent can confidently let another ambient in or out even if the former is evolving in a possibly ill-typed context: as long as typed reduction is respected, the security constraints that agent defines are never violated. The dual view holds as well: an agent can confidently enter or exit another ambient even if the latter is ill-typed: the reduction semantics ensures that the security constraints defined by the former are never violated.

[8] This property does not hold for the non-distributed calculus: the proof fails in the case for **(in)** as it is not possible to deduce the well-typing of the ambient *b*.

8 Communications

The analyses we developed in the previous sections were targeted to the combinatorial kernel of Safe Ambients. We now discuss their extension to the case of ambients with communication primitives: the extension is nontrivial, as communication may involve exchange of capabilities which, once received, may be exercised and thus affect the behavior of the ambient where they are received.

We first briefly introduce the constructs for communication, which are directly inherited from the corresponding constructs defined for Mobile Ambients in [6]. The type analyses for the extended calculus are developed in two steps: first we define a type system that only provides for exchange of capabilities; then we introduce a full-fledged system, where the exchange of values also includes ambient names, and study its properties in detail.

8.1 Safe Ambients and Communication

In addition to their ability to move, ambients and processes are now endowed with primitives for communication. As in the original proposal by Cardelli and Gordon, communication is anonymous and asynchronous, and takes place inside ambients. The new typed syntax is defined by the following extensions to the productions given in Section 2:

$$
\begin{array}{llcll}
\textit{Processes} & P & ::= & \ldots & \text{as in Section 2} \\
 & & & \langle M \rangle & \text{asynchronous output} \\
 & & & (x{:}V)P & \text{input} \\[1ex]
\textit{Capabilities} & \alpha & ::= & \ldots & \text{as in Section 2} \\[1ex]
\textit{Terms} & M & ::= & a, b, c, \ldots, x, y, z & \text{variables} \\
 & & \mid & \alpha & \text{capabilities} \\
 & & \mid & M.M & \text{paths}
\end{array}
$$

First, now $a, b, c, \ldots, x, y, z$ are used to range over variables, with the usual convention that constant names are variables we commit not to abstract upon. We will rather use —quite informally— $a, b, c, \ldots$ for ambient names and variables, and x, y for generic variables bound in input processes.

The productions introduce two new process forms: $(x{:}V)P$ inputs a value of type V (defined next) and then continues as P (with every free occurrence of the variable x substituted by the input term), while $\langle M \rangle$ denotes asynchronous output. The intuitive semantics of communication is that an output process $\langle M \rangle$ simply "drops" the term M which may then be input by any process running at the same nesting level, as in $(x{:}V)P \mid \langle M \rangle$. Terms that may be exchanged are names and capabilities, as well as *paths* of capabilities of the form $M.M'$.

The intuitive semantics of communication we just outline is formalized by two simple extensions of the relations of structural equivalence and reduction. Structural congruence is defined as in Section 2, with the addition of the following clause:

$$M_1.(M_2.P) \equiv (M_1.M_2).P$$

Reduction also in defined as in Section 2, with a new rule for communication, namely:

$$\textbf{(comm)} \qquad (x{:}V)P \mid \langle M \rangle \;\rightarrow\; P\{x := M\}$$

Note that communication is purely local, as it only happens when the input and output processes are at the same nesting level, hence within the same ambient. Instead, communication across ambient boundaries requires mobility and is effectively enabled by the open capability. To exemplify, consider two ambients running in parallel as in the following configuration $a[(x : V)P \mid Q] \mid b[\langle M\rangle \mid R]$. The exchange of the value M from b to the process P enclosed in a happens as a result of b first moving inside a, and then a opening b (or vice-versa, by a entering b and being opened there). Thus, if Q is the process $\mathsf{open}\ b$, and R is $\mathsf{in}\ a$, communication is the result of the following sequence of reductions:

$$\begin{array}{llll} a[(x : W)P \mid \mathsf{open}\ b] \mid b[\langle M\rangle \mid \mathsf{in}\ a] & \rightarrow & a[(x{:}V)P \mid \mathsf{open}\ b \mid b[\langle M\rangle]] & \text{by exercising } \mathsf{in}\ a \\ & \rightarrow & a[(x{:}V)P \mid \langle M\rangle] & \text{by opening } b \\ & \rightarrow & a[P\{x := M\}] & \end{array}$$

8.2 Exchanging Capabilities

As advocated by Cardelli and Gordon [6], communication of names should be rare in distributed systems, because knowing the name of an ambient gives full control over it; instead, communication of capabilities should be commonplace, as it allows controlled interaction between ambients. Our first type system takes this view to its extreme, and limits communication to the sole exchange of capabilities.

The resulting system is somewhat restrictive, but nevertheless interesting as it is based on a rather smooth and simple extension of the system discussed in Section 3. The basic observation for the new system is that capabilities and processes can be typed uniformly: in fact, given that process types describe the behavior of processes in terms of the capabilities those processes may exercise, it is natural to associate process types to capabilities as well [9]. Based on this observation, the type system is easily defined by taking the type V of exchange values to be the type P of processes, and by introducing new rules for typing capabilities in isolation. These rules, together with a new (PREFIX) rule replace the previous (ACTION$^{\uparrow}$) and (ACTION$^{=}$) rules from Section 3. In addition, of course, we have new typing rules for input and output processes.

(CAP$^{\uparrow}$)

$$\frac{\Pi, E \vdash a{:}D \quad \mathsf{cap}\, D \in \mathsf{P}^{\uparrow}}{\Pi, E \vdash \mathsf{cap}\ a : \mathsf{P}}\ \mathsf{cap} \in \{\mathsf{in}, \mathsf{coin}, \mathsf{out}, \mathsf{coopen}\}$$

(CAP$^{=}$)

$$\frac{\Pi, E \vdash a{:}D \quad \mathsf{cap}\, D \in \mathsf{P}^{=}}{\Pi, E \vdash \mathsf{cap}\ a : \mathsf{P}}\ \mathsf{cap} \in \{\mathsf{coout}, \mathsf{open}\}$$

(PATH)

$$\frac{\Pi, E \vdash M_1 : \mathsf{P} \quad \Pi, E \vdash M_2 : \mathsf{P}}{\Pi, E \vdash M_1.M_2 : \mathsf{P}}$$

(PREFIX)

$$\frac{\Pi, E \vdash M : \mathsf{P} \quad \Pi, E \vdash P : \mathsf{P}}{\Pi, E \vdash M.P : \mathsf{P}}$$

(INPUT)

$$\frac{\Pi, E, x : \mathsf{Q} \vdash P : \mathsf{P} \quad \mathsf{Q} \subseteq \mathsf{P} \quad x \notin \mathsf{Dom}(E)}{\Pi, E \vdash (x : \mathsf{Q})P : \mathsf{P}}$$

(OUTPUT)

$$\frac{\Pi, E \vdash M : \mathsf{P}}{\Pi, E \vdash \langle M\rangle : \mathsf{P}}$$

The intuition underlying the new system can be explained as follows. Process types now trace two different kinds of information: (i) the (implicit and explicit) behavior of a process, and (ii) the behavior resulting from the exchange of capabilities via communication.

The typing of capabilities, in the rules (CAP) characterizes capabilities as directly determining process behavior, observable at different nesting levels. The format of the rules is consistent with the format we used in Figure 1 and will use in the full-fledged system of Section 8.3: the algorithmic version of the rules

[9] This does not allow process exchanges and hence affect the first-order nature of the calculus. In fact, the syntax insures that only terms may be output, and hence by itself prevents process exchanges via communication.

(CAP$^{\uparrow}$) and (CAP$^{=}$) would derive the minimum types $(\{\mathsf{cap}, D\}, \varnothing, \varnothing)$ and $(\varnothing, \{\mathsf{cap}, D\}, \varnothing)$, respectively. The rule (PATH) simply collects the behavior associated with the capabilities on the path, and the rule (PREFIX) combines the behavior determined by the prefix with the behavior of the continuation process. Again, we have given the non-algorithmic versions of the rules: in the algorithmic versions of the rules (PATH) and (PREFIX) the type deduced by the conclusions would be the union of the two types deduced by the two respective premises.

The rule (INPUT) implicitly assumes that every capability input by a process may potentially be exercised: this is enforced by the constraint $\mathsf{Q} \subseteq \mathsf{P}$, requiring that the process exhibit in its type the behavior that may result from exercising any capability that is input by the process. Dually, the rule (OUTPUT) identifies the type of the capability being output with the type of the process that outputs it: this is required for type safety. To see that, assume $M : \mathsf{P}$, and consider the process $(x : \mathsf{Q})P \mid \langle M \rangle$. This process type-checks with the rules above only if $\mathsf{Q} \subseteq \mathsf{P}$, and the type assigned to P is a super-type of P, which therefore provides a safe approximation for the behavior P may acquire in the exchange of M.

Notice that the type used for the parameter of input processes can be any type, not necessarily a closed one. This is convenient, as it allows a more liberal typed syntax in which the type annotations are not necessarily closed, and their closure is automatically computed by the system. In Section 5, we showed that this can indeed be accomplished by the type reconstruction algorithm: in the new system, the type closure is implicitly computed by the rule (INPUT) which subsumes the possibly ill-formed parameter type Q, to the well-formed (i.e., closed) type P. From the last observation, it directly follows that the typing rules can be reformulated and based on an untyped syntax, by simply replacing (INPUT) rule above with the rule given next:

(CURRY-STYLE INPUT)

$$\frac{\Pi, E, x : \mathsf{P} \vdash P : \mathsf{P}}{\Pi, E \vdash (x)P : \mathsf{P}}$$

Discussion. While easily accommodated in the basic type system, the solution we just outlined is somewhat unsatisfactory. The problem is that representing behavior and exchange with process types effectively amounts to identifying the communication of a capability with the act of exercising it. Clearly, this leads to a rather coarse type analysis, because an ambient could exchange a capability without ever exercing it. A further source of unwanted approximation arises from type closure: take for instance the ambient $a[P] : \mathsf{P}$, and assume that P opens another ambient enclosing an output process $\langle M \rangle : \mathsf{Q}$. Then, type closure implies that P must subsume the type Q, even though P does not include any input process, and therefore it has no way to effectively exercise the capability.

8.3 Exchanging names and capabilities

A more effective analysis results from distinguishing the two forms of behavior a process exhibits: the capabilities it may exercise from the capabilities it may exchange. This can be accomplished by enriching the syntax of types as defined by the following productions. Let P denote the usual triples $(\mathbf{L}, \mathbf{M}, \mathbf{N})$, with $\mathbf{L}, \mathbf{M}, \mathbf{N} \in 2^{\mathsf{M}}$, and M denoting type capabilities, exactly as in Section 3. Then define:

$$\begin{array}{llcll}
\textit{Exchanges} & W & ::= & D[W] & \text{exchange of names} \\
 & & \mid & T & \text{exchange of capabilities} \\
 & & \mid & \mathsf{Shh} & \text{no exchange} \\
\textit{Processes} & T & ::= & \mathsf{P}[W] &
\end{array}$$

In addition, we define the types of values, as expected:

$$\textit{Values} \qquad V \quad ::= \quad D[W] \;\mid\; T$$

The structure of types is similar to that of the original type system for Mobile Ambients by Cardelli and Gordon [7]. Process types describe the two components of process behavior: the direct behavior resulting from exercising capabilities, traced by P, and the *exchange behavior* resulting from communication, traced by W. As we anticipated, communication can now exchange either capabilities (of type T) or ambient names (of type $D[W]$). The type $D[W]$ is assigned to ambients names of domain D whose internal exchanges, if any, are of type W. The typed syntax is similar to the previous, with the only restrictions that new names may only be declared at types of the form $D[W]$.

Processes

$$P ::= \mathbf{0} \;\mid\; M.P \;\mid\; (\boldsymbol{\nu} a{:}D[W])P \;\mid\; P \mid P \;\mid\; \langle M\rangle \;\mid\; (x{:}\,V)P \;\mid\; a[P] \;\mid\; !P$$

8.4 Environments and Typing Rules

The binding environments of the new type system are still defined as pairs of *Type Environments*, denoted by E, and *Domain Environments*, denoted by Π. Domain Environments defined as in Section 3, as finite maps from domain names to the component P of process types. Type environments, instead, have a different structure as they now map ambient names to the newly defined ambient types of the form $D[W]$, and input variables to value types V.

Interestingly the definition of closure, boundedness and coherence from Section 3 work just as well with the new structure of types. The typing rules, instead, are different: they derive five different forms of judgments, that is $\Pi \vdash \diamond$ or $\Pi, E \vdash \diamond$ or $\Pi \vdash T$ or $\Pi, E \vdash P{:}T$ or $\Pi, E \vdash M{:}W$.

Type and Environment Formation

(TYPE Shh)

$$\frac{\Pi \vdash \diamond}{\Pi \vdash \mathsf{Shh}}$$

(TYPE MESSG)

$$\frac{\Pi \vdash W \quad D \in \mathsf{Dom}(\Pi)}{\Pi \vdash D[W]}$$

(TYPE PROC)

$$\frac{\Pi \vdash W \quad \mathit{fn}(\mathsf{P}) \subseteq \mathsf{Dom}(\Pi) \quad \Pi \vdash \mathsf{P}\ \mathsf{closed}}{\Pi \vdash \mathsf{P}[W]}$$

(ENV$_1$)

$$\frac{\Pi \vdash \diamond}{\Pi, \varnothing \vdash \diamond}$$

(ENV$_2$)

$$\frac{\Pi, E \vdash \diamond \quad \Pi \vdash W \quad x \notin \mathsf{Dom}(E)}{\Pi\,,\, E, x{:}W \vdash \diamond}$$

Typing of Terms

(NAME)

$$\frac{\Pi, E \vdash \diamond \quad x \in \mathsf{Dom}(E)}{\Pi, E \vdash x : E(x)}$$

(CAP$^{\uparrow}$)

$$\frac{\Pi, E \vdash a{:}D[W'] \quad \Pi \vdash \mathsf{P}[W] \quad \mathsf{cap}\ D \in \mathsf{P}^{\uparrow}}{\Pi, E \vdash \mathsf{cap}\ a : \mathsf{P}[W]}\ \mathsf{cap} \in \{\mathsf{in}, \mathsf{coin}, \mathsf{out}, \mathsf{coopen}\ \}$$

$$(\textsc{Cap}^{=}_{1})\quad \frac{\Pi, E \vdash a{:}D[W'] \quad \Pi \vdash \mathsf{P}[W] \quad \mathsf{coout}\ D \in \mathsf{P}^{=}}{\Pi, E \vdash \mathsf{coout}\ a : \mathsf{P}[W]} \qquad (\textsc{Cap}^{=}_{2})\quad \frac{\Pi, E \vdash a{:}D[W] \quad \Pi \vdash \mathsf{P}[W] \quad \mathsf{open}\ D \in \mathsf{P}^{=}}{\Pi, E \vdash \mathsf{open}\ a : \mathsf{P}[W]}$$

$$(\textsc{Path})\quad \frac{\Pi, E \vdash M_1 : T \quad \Pi, E \vdash M_2 : T}{\Pi, E \vdash M_1.M_2 : T}$$

Typing of Processes

$$(\textsc{Dead})\quad \frac{\Pi, E \vdash \diamond \quad \Pi \vdash T}{\Pi, E \vdash \mathbf{0} : T} \qquad (\textsc{Repl})\quad \frac{\Pi, E \vdash P : T}{\Pi, E \vdash !P : T} \qquad (\textsc{Restr})\quad \frac{\Pi, E, a{:}D[W] \vdash P : T \quad \Pi \vdash D[W] \quad a \notin \mathsf{Dom}(E)}{\Pi, E \vdash (\boldsymbol{\nu} a{:}D[W])P : T}$$

$$(\textsc{Par})\quad \frac{\Pi, E \vdash P : T \quad \Pi, E \vdash Q : T}{\Pi, E \vdash P \mid Q : T} \qquad (\textsc{Prefix})\quad \frac{\Pi, E \vdash P : T \quad \Pi, E \vdash M : T}{\Pi, E \vdash M.P : T}$$

$$(\textsc{Input})\quad \frac{\Pi, E, x : V \vdash P : \mathsf{P}[V] \quad x \notin \mathsf{Dom}(E)}{\Pi, E \vdash (x : V)P : \mathsf{P}[V]} \qquad (\textsc{Output})\quad \frac{\Pi, E \vdash M : V \qquad \Pi \vdash \mathsf{P}[V]}{\Pi, E \vdash \langle M \rangle : \mathsf{P}[V]}$$

$$(\textsc{Amb})\quad \frac{\Pi, E \vdash P : \mathsf{P}[W] \quad \Pi, E \vdash a : D[W] \quad \Pi \vdash D\ \mathsf{bounds}\ \mathsf{P} \quad \Pi \vdash \mathsf{Q}[W'] \quad \Pi(D) \subseteq \mathsf{Q}}{\Pi, E \vdash a[P] : \mathsf{Q}[W']}$$

The rules for typing capabilities mimic those defined in [7] for Mobile Ambients. In particular the rule for open demands that the exchange types of the opened and the opening ambient coincide: this explains why the rule ($\textsc{Cap}^{=}$) is split into two rules. Note that all types occurring in processes are required to be well formed. This is unfortunate, as it requires the typing annotations for terms to be built around closed types, but at the same time necessary for safety.

8.4.1 Type Safety

The proof of type safety follows essentially the same argument described in Section 4, based on subject reduction. For the latter, Lemma A.6 is easily proved for the new system: one only needs an additional case for the new structural rule for paths, which follows immediately by an inspection of the typing rules. In addition, one needs the following revised form of Lemma A.7 and a substitution lemma, to handle communication. In both cases the proof is standard.

Lemma 8.1 (Subsumption Admissibility). *If* $\Pi, E \vdash P : \mathsf{P}[W]$*, then* $\Pi, E \vdash P : \mathsf{Q}[W]$ *for every* Q *such that* $\mathsf{P} \subseteq \mathsf{Q}$ *and* $\Pi, E \vdash \mathsf{Q}[W]$. □

Lemma 8.2 (Substitution). *If* $\Pi, E, x{:}V \vdash P : T$ *and* $\Pi, E \vdash M : V$*, then* $\Pi, E \vdash P\{x := M\} : T$ □

Theorem 8.3 (Subject Reduction). *If* $\Pi, E \vdash P : T$ *and* $P \rightarrow Q$*, then* $\Pi, E \vdash Q : T$.

Proof. A straightforward modification of the proof in Appendix A. □

The definition of *immediate exhibition* of a capability of Figure 2 does not change, because the input and output processes —$(x{:}V)P$ and $\langle M\rangle$— do contribute to any immediate exhibition of capabilities. The same is true of processes in prefix form $M.P$ when (the first capability of) M is a variable.

The definition of tagged reduction is directly derived from the corresponding definition in Figure 3 with the addition of the structural rule $\sharp\langle M\rangle \equiv \langle M\rangle$, and of a new reduction for communication, namely: $\sharp(x{:}V)P \mid \langle M\rangle \rightarrow \sharp P\{x := M\}$. Finally, one needs an additional context form to account for contexts built around the input construct: $(x{:}V)\mathscr{C}[\,]$.

Given these extensions, the notions of residual and residual behavior are defined exactly as in Definitions 4.1 and 4.2, respectively. Then we have:

Theorem 8.4 (Type Safety). *Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a process, with P containing no restriction, Δ be an occurrence of P and let $E = E', \vec{a}{:}\vec{D}$ for a type environment E'. Assume that $\Pi, E \vdash P : \mathsf{P}'$ and $\Pi, E \vdash P_\Delta : \mathsf{P}$. If $\Delta \Downarrow (\mathsf{cap}\ a)^\eta$, then $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$.*

Proof. (Sketch) A direct extension of the proof of Theorem 4.3. There is no change for any of definitions related to behavior types. The only novelty is that now processes may have the form $x.P$. On the other hand, such processes do not, in fact, have any immediate exhibition: they only exhibit a capability when the variable in the prefix is eventually substituted. Hence, if we can prove that the type of $x.P$ takes into account the type capabilities of all possible substitutions for x, then type safety follows. But this follows directly from subject reduction property and an inspection of the typing rule (PREFIX). □

9 Related Work

We have showed that classical type theoretic techniques provide effective tools for characterizing behavioral properties of mobile agents. Capturing implicit behavior is essential to ensure secure agent interactions: to our knowledge, ours is the first among type systems for Mobile Ambients to have this property. Also, we have showed that in the design of a distributed implementation of the calculus and its type system one finds back features distinctive of real systems. We conclude with comparisons with related work.

9.1 Type Systems for Mobile Ambients

Type systems for Mobile Ambients and related calculi have been studied in several papers. The first paper on the subject is by Cardelli and Gordon [7], where types are introduced to discipline the exchange of values inside ambients. In [4], Cardelli, Ghelli and Gordon extend the type system of [7] to account for ambient mobility. The new type system provides for a classification of ambients according to simple behavioral invariants: specifically, the type system identifies ambients that remain immobile, and ambients that may not be dissolved by their environment. In [12], Levi and Sangiorgi define a suite of type systems for their Safe Ambients, which also characterize behavioral properties of ambients, such as immobility and *single-threadedness*: based on these invariants, they prove interesting equivalences for well-typed processes. In [1], Amtoft, Kfoury and Pericas develop a type and effect system for Mobile Ambients that provides support for polymorphic exchanges within ambients. Work on combining their type system with the one presented here is part of our and their current collaborative research.

The type system closest to ours is the one presented by Cardelli, Ghelli, and Gordon in their recent paper on Ambient Groups [5]. Although their and our motivations are somewhat orthogonal —they refine previous work on static detection of ambient mobility, we give a type-theoretic account of security by defining and enforcing security policies for ambients— the two solutions have several similarities. If we disregard the security layer of our type system, our notion of ambient domain is essentially the same as their notion of *group*. Also, ambient behavior is characterized in both type systems in terms of sets built around domains (or equivalently groups). In [5] each group G is associated with sets that identify which groups ambients of group G may potentially cross or open. In our type system, we directly associate ambient domains with type-level capabilities with similar information content. However, our type system is superior in precision, as our type-capability sets are constructed in ways that allow implicit and hidden mobility to be statically detected. That is not always the case in the type system of [5]: only the first of the two attacks we discussed in the examples of Section 1.1 is detected by the type system of [5][10].

A further difference is the presence in [5] of a novel (and quite interesting) construct for dynamic group creation, a primitive that is not available for our version of mobile ambients. While we believe that this construct could be included in our type system, it would certainly complicate type reconstruction. Besides our specific interests in security issues, that are somewhat disregarded in [5], type reconstruction and the distributed version of the system (neither of which is discussed in [5]) represent further important differences between the two papers.

9.2 Static Analysis for Mobile Ambients

Although developed in a different framework, and based on different techniques, our work on type-based analysis has the same goals as F. and H.R. Nielson's study for control and data flow analysis for Mobile Ambients [15, 17] and achieves similar results.

In fact, our type reconstruction algorithm may be seen as an abstract control flow analysis where ambient behavior is abstracted upon in terms of domain behavior. In particular if we consider the work in [15] the resulting analysis is very similar to the one detailed here up to Section 5. In some respects, our analysis is more precise as we use co-capabilities and the three-levels structure of types to refine it. Furthermore, as we have shown, out analysis scales to the distributed version of the calculus, an issue that is not discussed in in [15]. In other respects, however, the analysis presented in [15] is finer than ours since they collect not only the actions emitted by an ambient, as we do, but also the set of its possible parents. This information is then used to refine the analysis as it allows one to disregard capabilities that may not be exercised: for example, the capability $\mathtt{out}\ a$ is included in an ambient's behavior only if the target ambient a is among the current ambient's parents. In fact, there seems to be no fundamental impediment in refining our system to perform the kind shape analysis proposed in [15]. Plans of future research work may include work in that direction.

The analyses of [15, 17] have been enhanced in [9, 11] by the use of abstract interpretation. In these works, as in [16] the complexity of the analyses is also studied, an issue that we completely overlooked here and leave for future work.

Acknowledgments

Work partially supported by the Italian MURST Project 9901403824_003 "Automatic Program Certification by Abstract Interpretation", by the French CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet" and by the European FET contract *MyThS*, IST-2001-32617.

[10] This is because in the system of [5] the type associated with a group G only traces the capabilities of the ambients that members of G may open, not those of the ambients exiting members of G.

References

[1] T. Amtoft, A. Kfoury, and S. Pericas-Geertsen. What are polymorphically-typed ambients? In *ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 2001.

[2] D. Brewer and M. Nash. The chinese wall security policy. In *Proc. of IEEE Symposium on Security and Privacy*, pages 206–214, 1982.

[3] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, pages 222–235, London, 2001. ACM Press.

[4] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for Mobile Ambients. In *Proceedings of ICALP '99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.

[5] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science, pages 333–347. Springer, 2000.

[6] L. Cardelli and A. Gordon. Mobile Ambients. In *Proceedings of F0SSaCS'98*, number 1378 in Lecture Notes in Computer Science, pages 140–155. Springer, 1998.

[7] L. Cardelli and A. Gordon. Types for Mobile Ambients. In *Proceedings of POPL '99*, pages 79–92. ACM Press, 1999.

[8] P. J. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, Dec. 1976.

[9] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *Eighth International Static Analysis Symposium (SAS '01)*, number 2126 in Lecture Notes in Computer Science. Springer, 2001.

[10] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[11] F. Levi and S. Maffeis. An abstract interpretation framework to analyse mobile ambients. In *Eighth International Static Analysis Symposium (SAS '01)*, number 2126 in Lecture Notes in Computer Science. Springer, 2001.

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

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Java series. Addison-Wesley, 1997.

[14] G. Necula. Proof carrying code. In *24th Ann. ACM Symp. on Principles of Programming Languages.* ACM Press, 1997.

[15] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *Proc. CONCUR '99*, number 1664 in LNCS, pages 463–477. Springer, 1999.

[16] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. ESOP'01*, number 2028 in Lecture Notes in Computer Science, pages 252–268. Springer, 2001.

[17] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL '00*, pages 142–154. ACM Press, 2000.

[18] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *13th IEEE Computer Security Foundations Workshop*, 2000.

A Subject Reduction

We first prove a few simple and useful properties for domain environments and process types. In that direction, we extend the set-theoretic notation used on processes to domain environments as follows. Given two domain environments Π_2 and Π_2 such that $\mathsf{Dom}(\Pi_1) = \mathsf{Dom}(\Pi_2)$, we define $\Pi_1 \cap \Pi_2$ (respectively, $\Pi_1 \cup \Pi_2$) to be the domain environment that maps every $D \in \mathsf{Dom}(\Pi_i)$ into the process type $\Pi_1(D) \cap \Pi_2(D)$ (respectively, $\Pi_1(D) \cup \Pi_2(D)$).

Proposition A.1 (Boundedness and Closedness). *Let Π and Π' be domain environments, D an ambient domain, and P, P' two process types.*

1. *If* $\Pi \vdash D$ bounds P *and* $\Pi \vdash D$ bounds P', *then* $\Pi \vdash D$ bounds $(\mathsf{P} \cup \mathsf{P}')$
2. *If* $\Pi \vdash D$ bounds P *and* $\mathsf{P}' \subseteq \mathsf{P}$, *then* $\Pi \vdash D$ bounds P'.
3. *If* $\Pi \vdash D$ bounds P *and* $\Pi' \vdash D$ bounds P, *then also* $\Pi \cap \Pi' \vdash D$ bounds P.
4. *If* $\Pi \vdash D$ closed *and* $\Pi' \vdash D$ closed, *then also* $\Pi \cap \Pi' \vdash D$ closed.
5. *If* $\Pi \vdash \mathsf{P}$ closed *and* $\Pi \vdash \mathsf{P}'$ closed, *then also* $\Pi \vdash \mathsf{P} \cup \mathsf{P}'$ closed.

Proof. In all cases, the proof is by a direct application of the definitions. □

Corollary A.2 (Coherence). *Let Π, Π' be domain environments. If $\Pi \vdash \diamond$ and $\Pi' \vdash \diamond$, then $\Pi \cap \Pi' \vdash \diamond$.* □

Lemma A.3 (Process Types). *Let Π be a domain environment, and $\mathsf{P}_1, \mathsf{P}_2$ be two process types such that $\Pi \vdash \mathsf{P}_1$ and $\Pi \vdash \mathsf{P}_2$. Then $\Pi \vdash \mathsf{P}_1 \cup \mathsf{P}_2$*

Proof. By Proposition A.1. □

Lemma A.4 (Type Formation). *If $\Pi, E \vdash P : \mathsf{P}$, then $\Pi \vdash \diamond$ and $\Pi \vdash \mathsf{P}$.*

Proof. By induction on the derivation of $\Pi, E \vdash P : \mathsf{P}$. □

Lemma A.5 (Generation).

1. *If* $\Pi, E \vdash a : D$, *then* $D = E(a)$.
2. *If* $\Pi, E \vdash P \mid Q : \mathsf{P}$ *then* $\Pi, E \vdash P : \mathsf{P}$ *and* $\Pi, E \vdash Q : \mathsf{P}$;
3. *If* $\Pi, E \vdash !P : \mathsf{P}$ *then* $\Pi, E \vdash P : \mathsf{P}$;
4. *If* $\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}$, *then* $\Pi, E \vdash P : \mathsf{P}$, *and* $\Pi, E \vdash a : A$ *for some ambient domain A. Furthermore, either (i)* $\mathsf{cap} \in \{\mathsf{in}\ , \mathsf{coin}\ , \mathsf{out}\ , \mathsf{coopen}\ \}$ *and* $\mathsf{cap}\ A \in \mathsf{P}^{\uparrow}$, *or (ii)* $\mathsf{cap} \in \{\mathsf{coout}\ , \mathsf{open}\ \}$ *and* $\mathsf{cap}\ A \in \mathsf{P}^{=}$
5. *If* $\Pi, E \vdash (\boldsymbol{\nu} a{:}D)P : \mathsf{P}$, *then* $\Pi\ , E, a{:}D \vdash P{:}\ \mathsf{P}$
6. *Assume* $\Pi, E \vdash a[P] : \mathsf{P}$. *Then* $\Pi(E(a)) \subseteq \mathsf{P}$ *and there exists* P' *such that* $\Pi, E \vdash P{:}\ \mathsf{P}'$, *and* $\Pi \vdash E(a)$ bounds P'.

Proof. In each case, by an inspection of the typing rules. □

Lemma A.6 (Subject Congruence). *If* $\Pi, E \vdash P : \mathsf{P}$ *and* $P \equiv Q$*, then* $\Pi, E \vdash Q : \mathsf{P}$.

Proof. By simultaneous induction on the derivations of $P \equiv Q$ and $Q \equiv P$. □

Lemma A.7 (Subsumption Admissibility). *If* $\Pi, E \vdash P : \mathsf{P}$*, then* $\Pi, E \vdash P : \mathsf{Q}$ *for every* Q *such that* $\mathsf{P} \subseteq \mathsf{Q}$ *and* $\Pi \vdash \mathsf{Q}$.

Proof. An easy induction on the derivation of $\Pi, E \vdash P : \mathsf{P}$. □

Theorem A.8 (Subject Reduction). *If* $\Pi, E \vdash P : \mathsf{P}$ *and* $P \rightarrow Q$*, then* $\Pi, E \vdash Q : \mathsf{P}$.

Proof. The proof is by induction on the depth of the derivation of the reduction, and by a case analysis on the last rule in the derivation.

Case (open) $\mathsf{open}\ a.P_1 \mid a[\mathsf{coopen}\ a.P_2 \mid P_3] \rightarrow P_1 \mid P_2 \mid P_3$

From $\Pi, E \vdash \mathsf{open}\ a.P_1 \mid a[\mathsf{coopen}\ a.P_2 \mid P_3] : \mathsf{P}$, by repeated applications of Lemma A.5:2, A.5:4, and A.5:6, there exist an ambient domain $D \in \mathsf{Dom}(\Pi)$ with $\Pi(D) \subseteq \mathsf{P}$, and a process type P_a such that the following are all verified:

$$\Pi, E \vdash P_1 : \mathsf{P} \tag{2}$$

$$\Pi, E \vdash \mathsf{coopen}\ a.P_2 \mid P_3 : \mathsf{P}_a \tag{3}$$

$$\Pi, E \vdash P_2 : \mathsf{P}_a \quad \text{and} \quad \Pi, E \vdash P_3 : \mathsf{P}_a \tag{4}$$

$$\Pi, E \vdash a : D \quad \text{and} \quad \Pi \vdash D\ \mathsf{bounds}\ \mathsf{P}_a \tag{5}$$

From (3 and the first judgment in (4), by Lemma A.5:4, we know that $\mathsf{coopen}\ D \in \mathsf{P}_a{}^{\uparrow}$. From this, and from (5), we know that $\mathsf{coopen}\ D \in \Pi(D)^{=}$, and hence $\mathsf{P}_a \subseteq \Pi(D)$ again from (5). Then $\mathsf{P}_a \subseteq \mathsf{P}$ since $\Pi(D) \subseteq \mathsf{P}$. By subsumption, which is admissible by Lemma A.7, from (2) and the two judgments in (4), we then derive $\Pi, E \vdash P_i : \mathsf{P}$ for $i = 1, 2, 3$. Then $\Pi, E \vdash P_1 \mid P_2 \mid P_3 : \mathsf{P}$ derives by two applications of (PAR).

Case (in): $a[\mathsf{coin}\ a.P_1 \mid P_2] \mid b[\mathsf{in}\ a.Q_1 \mid Q_2] \rightarrow a[P_1 \mid P_2 \mid b[Q_1 \mid Q_2]]$

From $\Pi, E \vdash a[\mathsf{coin}\ a.P_1 \mid P_2] \mid b[\mathsf{in}\ a.Q_1 \mid Q_2] : \mathsf{P}$, by Lemma A.4 we know that $\Pi \vdash \diamond$. By repeated applications of Lemma A.5:2, A.5:4, and A.5:6 there exist ambient domains $A, B \in \mathsf{Dom}(\Pi)$, with $\Pi(A), \Pi(B) \subseteq \mathsf{P}$, and process types P_a and Q_b such that the following are all verified:

$$\Pi, E \vdash \mathsf{in}\ a.Q_1 \mid Q_2 : \mathsf{Q}_b \tag{1}$$

$$\Pi, E \vdash Q_1 : \mathsf{Q}_b \quad \text{and} \quad \Pi, E \vdash Q_2 : \mathsf{Q}_b \tag{2}$$

$$\Pi, E \vdash b : B \quad \text{and} \quad \Pi \vdash B\ \mathsf{bounds}\ \mathsf{Q}_b \tag{3}$$

$$\Pi, E \vdash \mathsf{coin}\ a.P_1 \mid P_2 : \mathsf{P}_a \tag{4}$$

$$\Pi, E \vdash P_1 : \mathsf{P}_a \quad \text{and} \quad \Pi, E \vdash P_2 : \mathsf{P}_a \tag{5}$$

$$\Pi, E \vdash a : A \quad \text{and} \quad \Pi \vdash A\ \mathsf{bounds}\ \mathsf{P}_a \tag{6}$$

From (1), the left judgments of (2) and (6), by Lemma A.5:4, we know that $\mathsf{in}\ A \in \mathsf{Q}_b{}^{\uparrow}$. From this and from (3), $\mathsf{in}\ A \in \Pi(B)^{=}$. From the left judgment of (5), we also know that $\mathsf{coin}\ A \in \mathsf{P}_a{}^{\uparrow}$. From this and from (6), $\mathsf{coin}\ A \in \Pi(A)^{=}$. Summarizing we have, $\mathsf{in}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{=})$. From this, and from $\Pi \vdash \diamond$, we know that $\Pi \vdash A \ \mathsf{bounds}\ \Pi(B)$. From this, and from the right judgment of (6), by Proposition A.1.1, we have

$$\Pi \vdash A \ \mathsf{bounds}\ (\Pi(B) \cup \mathsf{P}_a) \tag{7}$$

From the two judgments in (2), by (PAR), $\Pi, E \vdash Q_1 \mid Q_2 : \mathsf{Q}_b$. From this, and (3), by (AMB)

$$\Pi, E \vdash b[Q_1 \mid Q_2] : \Pi(B) \tag{8}$$

From the two judgments in (5), by (PAR), $\Pi, E \vdash P_1 \mid P_2 : \mathsf{P}_a$. From (8) and the last judgment, by subsumption and (PAR),

$$\Pi, E \vdash P_1 \mid P_1 \mid b[Q_1 \mid Q_2] : \Pi(B) \cup \mathsf{P}_a \tag{9}$$

Now, the type of the reduct derives from (9), (7), and the left judgment of (6) by (AMB).

Case (out): $a[\mathsf{coout}\ a.P_1 \mid P_2 \mid b[\mathsf{out}\ a.Q_1 \mid Q_2]\,] \rightarrow a[P_1 \mid P_2] \mid b[Q_1 \mid Q_2]$
As in the previous cases, by repeated applications of Lemma A.5 to the typing judgment of the redex, there exist process types P_a and Q_b, and ambient domains $A, B \in \mathsf{Dom}(\Pi)$ with $\Pi(A) \subseteq \mathsf{P}$ and $\Pi(B) \subseteq \mathsf{P}_a$, such that the following are all verified:

$$\Pi, E \vdash \mathsf{out}\ a.Q_1 \mid Q_2 : \mathsf{Q}_b \tag{1}$$

$$\Pi, E \vdash Q_1 : \mathsf{Q}_b \quad \text{and} \quad \Pi, E \vdash Q_2 : \mathsf{Q}_b \tag{2}$$

$$\Pi, E \vdash b : B \quad \text{and} \quad \Pi \vdash B \ \mathsf{bounds}\ \mathsf{Q}_b \tag{3}$$

$$\Pi, E \vdash \mathsf{coout}\ a.P_1 \mid P_2 \mid b[\mathsf{out}\ a.Q_1 \mid Q_2] : \mathsf{P}_a \tag{4}$$

$$\Pi, E \vdash P_1 : \mathsf{P}_a \quad \text{and} \quad \Pi, E \vdash P_2 : \mathsf{P}_a \tag{5}$$

$$\Pi, E \vdash a : A \quad \text{and} \quad \Pi \vdash A \ \mathsf{bounds}\ \mathsf{P}_a \tag{6}$$

From the left judgments of (2) and (6), by Lemma A.5:4, we know that $\mathsf{out}\ A \in {\mathsf{Q}_b}^{\uparrow}$. From this and from (3), $\mathsf{out}\ A \in \Pi(B)^{=}$. From the left judgment of (5), we also know that $\mathsf{coout}\ A \in \mathsf{P}_a^{=}$. From this and from (6), $\mathsf{coout}\ A \in \Pi(A)^{\downarrow}$. Thus, $\mathsf{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$. From this, and from $\Pi \vdash \diamond$, we know that $\Pi(B) \subseteq \Pi(A)$. It is now easy to check that the judgments $\Pi, E \vdash b[Q_1 \mid Q_2] : \Pi(B)$ and $\Pi, E \vdash a[P_1 \mid P_2] : \Pi(A)$ are both derivable. The typing judgment for the reductum derives then by subsumption and an application of (PAR).

Case (context): Standard, by induction hypothesis.

Case (struct): by Lemma A.6 and the induction hypothesis. □

B Type Safety

Lemma B.1. *Let $\mathscr{C}[\,]$ be a restriction-free context, and P be a restriction-free process. Assume that $\Pi, E \vdash \mathscr{C}[P] : \mathsf{P}'$ and $\Pi, E \vdash P : \mathsf{P}$. Consider a generic one-step of tagged reduction from $\mathscr{C}[\sharp P]$, that is: $\mathscr{C}[\sharp P] \equiv \mathscr{C}_1[\sharp R] \rightarrowtail \mathscr{C}_2[\sharp Q]$ for some contexts $\mathscr{C}_1[\,]$ and $\mathscr{C}_2[\,]$. Then $\Pi, E \vdash |Q| : \mathsf{P}$.*

Proof. We first show that the lemma holds for the preliminary step of structural rearrangement, i.e. that $\Pi, E \vdash |R| : \mathsf{P}$. This can be done by induction on the depth of the derivation of the congruence. Since $\mathscr{C}[\sharp P]$ contains a single tagged occurrence, $\mathscr{C}_1[\sharp R]$ results from either rearranging only untagged occurrences, or from rearranging $\sharp P$. In the first case the claim is trivially true. Then, consider the case when $\sharp P$ matches either side of a congruence rule. Since P is restriction-free by hypothesis, we have only four base cases to consider, namely: $\sharp P = \sharp\mathbf{0}$, $\sharp P = \sharp(P_1 \mid P_2)$, for given P_1 and P_2, and finally $\sharp P = \sharp !P_1$, or $\sharp P = !\sharp P_1$. In all cases the claim follows by Lemma A.6. The first case is vacuous, as there is no tagged process corresponding to $\sharp\mathbf{0}$. The second case follows by the type rule (PAR) and the last two cases follow by (REPL). For the inductive cases, the only subtlety is transitivity, as the intermediate tagged process may contain more than one tagged occurrence. However, since there is only one tag in $\mathscr{C}[\sharp P]$, it is not difficult to see that $\mathscr{C}_1[\sharp R]$ can always be obtained by a sequence of rearrangements that only use the congruence law $\sharp(P_1 \mid P_2) \equiv \sharp P_1 \mid \sharp P_2$ from left to right.

Next, consider one step of tagged-reduction from $\mathscr{C}_1[\sharp R]$. If $\sharp R$ is not a sub-occurrence of the redex nor is the redex a suboccurence of $\sharp R$, then the proof is trivial. The same holds if $\sharp R$ is a sub-occurrence of the redex but it is not one of the tagged processes involved in the reduction. If the redex is a sub-occurrence of $\sharp R$, then the proof follows by subject reduction. The remaining cases are when $\sharp R$ is one of the processes involved in the reduction: we work out the interesting cases below, the remaining cases are similar and simpler.

(open tag) $\mathsf{open}\ a.S \mid \sharp a[\mathsf{coopen}\ a.R_1 \mid R_2] \rightarrowtail S \mid \sharp(R_1 \mid R_2)$, where $R = a[\mathsf{coopen}\ a.R_1 \mid R_2]$ and $Q = R_1 \mid R_2$. From the hypothesis, we know that $\Pi, E \vdash a[\mathsf{coopen}\ a.R_1 \mid R_2] : \Pi(E(a))$. From Lemma A.5:6, there exists P' such that $\Pi, E \vdash \mathsf{coopen}\ a.R_1 \mid R_2 : \mathsf{P}'$ with $\Pi \vdash E(a)\ \mathsf{bounds}\ \mathsf{P}'$. By repeated applications of Lemma A.5 we also have that $\Pi, E \vdash R_1 \mid R_2 : \mathsf{P}'$. From $\Pi, E \vdash \mathsf{coopen}\ a.R_1 \mid R_2 : \mathsf{P}'$ the typing rules tell us that $\mathsf{coopen}\ E(a) \in \mathsf{P}' \uparrow$ and by the definition of boundness this implies $\mathsf{coopen}\ E(a) \in \Pi(E(a))^{=}$. From $\Pi, E \vdash R_1 \mid R_2 : \mathsf{P}'$, and from $\mathsf{coopen}\ E(a) \in \Pi(E(a))^{=}$, by closure it follows that $\mathsf{P}' \subseteq \Pi(E(a))$ as desired.

(out tag) $\sharp a[\ b[\mathsf{out}\ a.R_1 \mid R_2] \mid \mathsf{coout}\ a.R_3 \mid R_4] \rightarrowtail \sharp b[R_1 \mid R_2] \mid\ \sharp a[R_3 \mid R_4]$. The proof follows the pattern of the case (*out*) in the proof of Theorem A.8.

(in) $b[\mathsf{in}\, a.S_1 \mid S_2] \mid \sharp a[\mathsf{coin}\, a.R_1 \mid R_2] \rightarrowtail \sharp a[R_1 \mid R_2 \mid b[S_1 \mid S_2]]$, where $R = a[\mathsf{coin}\ a.R_1 \mid R_2]$ and $Q = a[R_1 \mid R_2 \mid b[S_1 \mid S_2]]$. Again, the proof follows the pattern of the case (*in*) of Theorem A.8. From the hypothesis, we know that $\Pi, E \vdash b[\mathsf{in}\ a.S_1 \mid S_2] : \Pi(E(b))$. Hence also $\Pi, E \vdash b[S_1 \mid S_2] : \Pi(E(b))$. To conclude, it is enough to show that $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$. But this follows from the coherence of Π, given that in $E(a) \in \mathsf{sync}(\Pi(E(b))^{=}, \Pi(E(a))^{=})$.

□

Lemma B.2. *If* $\Pi, E \vdash P : \mathsf{P}$ *and* $P \downarrow (\mathsf{cap}\, a)^{\eta}$ *then* $\mathsf{cap}\, E(a) \in \mathsf{P}^{\eta}$.

Proof. By a direct inspection of the typing rules and a straightforward induction on the depth of the derivation of $P \downarrow (\mathsf{cap}\, a)^{\eta}$. □

The proof of Type Safety is a corollary of the following Lemma.

Lemma B.3. *Let P be a restriction-free process, Δ be an occurrence of P and let E a type environment. Assume that* $\Pi, E \vdash P : \mathsf{P}'$ *and* $\Pi, E \vdash P_{\Delta} : \mathsf{P}$ *are derivable. If* $\Delta \Downarrow (\mathsf{cap}\ a)^{\eta}$*, then* $\mathsf{cap}\ E(a) \in \mathsf{P}^{\eta}$.

Proof. Follows by Lemma B.1 and Lemma B.2 by induction on the number of reduction steps needed to reach the residual of P_Δ that emits $(\mathsf{cap}\ a)^\eta$. The base case is proved by Lemma B.2, while the inductive case is obtained by considering the residuals after a one step reduction. The proof of the inductive case is eased by the definition of residuals in terms of one-step reductions of processes that have at most one tag, and that structural congruence is applied only before (not after) a reduction step. □

Theorem B.4 (Type Safety). *Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a process, with P containing no restriction, Δ be an occurrence of P and let $E = E', \vec{a}{:}\vec{D}$ for a type environment E'. Assume that $\Pi, E \vdash P : \mathsf{P}'$ and $\Pi, E \vdash P_\Delta : \mathsf{P}$. If $\Delta \Downarrow (\mathsf{cap}\ a)^\eta$, then $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$.*

Proof. A corollary of the previous lemma. □

Finally let us consider the safety for the the distributed system:

Theorem B.5 (Local Type Safety). *Let $(\boldsymbol{\nu}\vec{a}{:}\vec{D})P$ be a DSSA process, with P containing no restriction, and Δ be an occurrence of P of the form $a[P']^{\mathsf{S}}_{\Pi,E}$. Assume $\Pi, E \vdash P_\Delta : \mathsf{P}$ is derivable, and $E(b) = B$. If $\Delta \Downarrow (\mathsf{cap}\ b)^\eta$, then $\mathsf{cap}\ B \in \mathsf{P}^\eta$.*

Proof. (*Sketch*) The proof is based on the analogue of Lemmas B.2 and B.3 for DSSA processes, and a different version of Lemma B.1 that handles the new form of the **(out)** and **(in)** reductions. The only critical case is the subcase of **(in)** in which $\sharp R$ (i.e., Δ) is the entered ambient. For DSSA, this case follows by two side conditions of the **(in)** rule: $\Pi, E \vdash b[\mathsf{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} : \Pi(E(b))$, that ensures that the local environment of the reductum can type its body, and $\Pi \vdash E(a)$ bounds $\Pi(E(b))$, that ensures that the behavior of the entering ambient is already accounted for by the local environments of the reductum. Then, the result follows from the observation that $\Pi(E(a)) = \Pi_a(E_a(a))$.

Note that the theorem is stated for ambient occurrences and not generic occurrences. Indeed the result does not hold for generic processes since in DSSA we did not modify the **(open)** rule to check that opened ambients are well-typed. □

C Type Reconstruction

Proposition C.1. *Let Π be a domain environment with fn$(\Pi) \subseteq \mathsf{Dom}(\Pi)$. Then $\mathsf{EnvClosure}(\Pi)$ is the least coherent domain environment containing Π.*

Proof. To prove the claim it is enough to show that $\{\Pi' \mid \Pi \subseteq \Pi' \text{ and } \Pi' \vdash \diamond\}$ is not empty and finite. The proof follows then by Corollary A.2. That this set is not empty follows by observing that the environment Π^{sat} that results from Π by saturating $\Pi(D)$ for every $D \in \mathsf{Dom}(\Pi)$ is contained in it. That the set is finite follows from the fact that $\mathsf{Dom}(\Pi)$ is finite. □

Proposition C.2. *Let Π a coherent domain environment and $A \in \mathsf{Dom}(\Pi)$. Then for every process type P,*

1. $\mathsf{EnvClosure}(\Pi) = \Pi$.
2. $\Pi \vdash \mathsf{ProcClosure}(\mathsf{P}, \Pi)$ closed.
3. $\mathsf{DomClosure}(\mathsf{P}, A, \Pi) \vdash A$ bounds P. □

To prove the reconstruction algorithm sound, we need the following additional lemmas.

Lemma C.3. *Let* Π *be a be domain environment,* P *be a process, and let* $\mathsf{P}^\star = \mathsf{ProcClosure}(\mathsf{P}, \Pi)$*. Then* $\mathsf{P}^\star = \bigcap\{\mathsf{P}' \mid \mathsf{P}' \supseteq \mathsf{P}$ *and* $\Pi \vdash \mathsf{P}'\ \mathsf{closed}\}$.

Proof. $\Pi \vdash \mathsf{P}^\star\ \mathsf{closed}$ follows by Proposition C.2. That $\mathsf{P}^\star$ is the minimum superset of P closed in Π follows by observing that $\mathsf{P}^\star$ is the minimum fixed-point of the following monotone operator: $\mathsf{pc}_\Pi(\mathsf{P}) = \mathsf{P} \cup \{\Pi(A) \mid \mathsf{open}\,A \in \mathsf{sync}(\mathsf{P}^=, \Pi(A)^=)\}$. □

Lemma C.4. *Let* Π *be a be domain environment,* P_1 *and* P_2 *process types. Then:*

$$\mathsf{ProcClosure}(\mathsf{P}_1, \Pi) \cup \mathsf{ProcClosure}(\mathsf{P}_2, \Pi) = \mathsf{ProcClosure}(\mathsf{P}_1 \cup \mathsf{P}_2, \Pi)$$

Proof. We prove the double inclusion. The direction ($\subseteq$) follows by monotonicity. The direction ($\supseteq$) follows (i) by Proposition C.2(2) by which $\Pi \vdash \mathsf{ProcClosure}(\mathsf{P}_i, \Pi)\ \mathsf{closed}$ ($i = 1, 2$), then (ii) by Proposition A.1(5) by which $\Pi \vdash \mathsf{ProcClosure}(\mathsf{P}_1, \Pi) \cup \mathsf{ProcClosure}(\mathsf{P}_2, \Pi)\ \mathsf{closed}$, and finally (iii) by Lemma C.3, as $\mathsf{ProcClosure}(\mathsf{P}_1, \Pi) \cup \mathsf{ProcClosure}(\mathsf{P}_2, \Pi) \supseteq \mathsf{P}_1 \cup \mathsf{P}_2$. □

Lemma C.5. *Assume* $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$*, and let* Π' *be any coherent domain environment containing* Π*. Then* $\Pi', E \vdash_{\mathscr{A}} P : \mathsf{P}^\star$ *where* $\mathsf{P}^\star = \mathsf{ProcClosure}(\mathsf{P}, \Pi')$.

Proof. By induction on the derivation of $\Pi \vdash_{\mathscr{A}} P : \mathsf{P}$. □

Theorem C.6 (Soundness and completeness). *Let* P *be a process, and* E *a type environment such that* $\mathit{fn}(P) \subseteq \mathsf{Dom}(E)$*. Then* $\mathscr{R}_{\mathsf{env}}(E, P), E \vdash_{\mathscr{A}} P : \mathscr{R}_{\mathsf{type}}(E, P)$ *(soundness). Furthermore, for any* Π *and* P *such that* $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$*, one has* $\mathscr{R}_{\mathsf{env}}(E, P) \subseteq \Pi$ *and* $\mathscr{R}_{\mathsf{type}}(E, P) \subseteq \mathsf{P}$ *(completeness).*

Proof. By induction on the structure of P.

$P = \mathbf{0}$ In this case $\mathscr{R}_{\mathsf{env}}(E, P) = \varnothing_{\mathscr{D}}$ and $\mathscr{R}_{\mathsf{type}}(E, P) = (\varnothing, \varnothing, \varnothing)$. By construction, $\varnothing_{\mathscr{D}} \vdash \diamond$, and $\mathsf{Img}(E) \subseteq \mathsf{Dom}(\varnothing_{\mathscr{D}})$. Hence $\varnothing_{\mathscr{D}}, E \vdash \diamond$ by (ENV), and $\mathscr{R}_{\mathsf{env}}(E, P), E \vdash_{\mathscr{A}} P : \mathscr{R}_{\mathsf{type}}(E, P)$ derives by (DEAD). Completeness is trivial.

$P = \mathsf{cap}\ a.P'$ Let $\Pi = \mathscr{R}_{\mathsf{env}}(E, P')$ and $\mathsf{P} = \mathscr{R}_{\mathsf{type}}(E, P')$. By induction hypothesis, we have $\Pi, E \vdash_{\mathscr{A}} P' : \mathsf{P}$, and for any Π' and P' such that $\Pi', E \vdash_{\mathscr{A}} P' : \mathsf{P}'$, we have $\Pi \subseteq \Pi'$ and $\mathsf{P} \subseteq \mathsf{P}'$. By construction, there exists A such that $E(a) = A$. There are now three cases, depending on the structure of cap.

If $\mathsf{cap} \in \{\mathsf{in}\ , \mathsf{coin}\ , \mathsf{out}\ , \mathsf{coopen}\ \}$, by definition $\mathscr{R}_{\mathsf{env}}(E, P) = \Pi$ and $\mathscr{R}_{\mathsf{type}}(E, P) = \mathsf{P} \cup {}^{\uparrow}\{\mathsf{cap}\ A\}$. Then the desired judgment derives from (ACTION$^{\uparrow}$).

If $\mathsf{cap} = \mathsf{coout}\ A$, by definition $\mathscr{R}_{\mathsf{env}}(E, P) = \Pi$ and $\mathscr{R}_{\mathsf{type}}(E, P) = \mathsf{P} \cup^{=} \{\mathsf{coout}\ A\}$. Then the desired judgment derives from (ACTION$^{=}_{1}$).

If $\mathsf{cap} = \mathsf{open}\ A$, by definition $\mathscr{R}_{\mathsf{env}}(E, P) = \Pi$ and $\mathscr{R}_{\mathsf{type}}(E, P) = \mathsf{P}'$ as defined by the side-condition of (**R**-ACTION$^{=}_{2}$). The desired judgment derives from (ACTION$^{=}_{2}$).

In all three cases completeness follows from the induction hypothesis and the fact that set-union is monotonic.

$P = !P'$ *and* $P = (\boldsymbol{\nu} a{:}A)P'$ Directly, by induction hypothesis.

$P = P_1 \mid P_2$ Let $\Pi_1 = \mathscr{R}_{\mathsf{env}}(E, P_1)$, $\mathsf{P}_1 = \mathscr{R}_{\mathsf{type}}(E, P_1)$, $\Pi_2 = \mathscr{R}_{\mathsf{env}}(E, P_2)$ and $\mathsf{P}_2 = \mathscr{R}_{\mathsf{type}}(E, P_2)$. By induction hypothesis, $\Pi_1, E \vdash_{\mathscr{A}} P_1 : \mathsf{P}_1$, and $\Pi_2, E \vdash_{\mathscr{A}} P_2 : \mathsf{P}_2$. Let now $\Pi = \mathscr{R}_{\mathsf{env}}(E, P) \triangleq \mathsf{EnvClosure}(\Pi_1 \cup \Pi_2)$. By Proposition C.1, $\Pi_1, \Pi_2 \subseteq \Pi$, and $\Pi \vdash \diamond$. From the last two judgments, by Lemma C.5

$$\Pi, E \vdash_{\mathscr{A}} P_1 : \mathsf{P}_1^\star \quad \textit{with} \quad \mathsf{P}_1^\star = \mathsf{ProcClosure}(\mathsf{P}_1, \Pi) \tag{7}$$

$$\Pi, E \vdash_{\mathscr{A}} P_2 : \mathsf{P}_2^\star \quad \textit{with} \quad \mathsf{P}_2^\star = \mathsf{ProcClosure}(\mathsf{P}_2, \Pi) \tag{8}$$

From (7) and (8) above, by (PAR), $\Pi, E \vdash_{\mathscr{A}} P_1 \mid P_2 : (\mathsf{P}_1^\star \cup \mathsf{P}_2^\star)$. By Lemma C.4 we know that $\mathsf{P}_1^\star \cup \mathsf{P}_2^\star = \mathsf{ProcClosure}((\mathsf{P}_1 \cup \mathsf{P}_2), \Pi)$ and hence conclude as $\mathscr{R}_{\mathsf{type}}(E, P) = \mathsf{ProcClosure}((\mathsf{P}_1 \cup \mathsf{P}_2), \Pi)$.

Completeness follows by induction hypothesis and monotonicity of the EnvClosure and ProcClosure operators. In fact, for any Π' and P' such that $\Pi', E \vdash_{\mathscr{A}} P_1 \mid P_2 : \mathsf{P}'$, by induction hypothesis one has $\Pi_1 \subseteq \Pi'$ and $\Pi_2 \subseteq \Pi'$, which implies $\Pi_1 \cup \Pi_2 \subseteq \Pi'$. Furthermore since Π' is coherent, by Proposition C.2(1) we obtain $\Pi' = \mathsf{EnvClosure}(\Pi')$. From these last two points and the monotonicity of EnvClosure we have $\mathscr{R}_{\mathsf{env}}(E, P) \triangleq \mathsf{EnvClosure}(\Pi_1 \cup \Pi_2) \subseteq \mathsf{EnvClosure}(\Pi') = \Pi'$. A similar reasoning yields $\mathscr{R}_{\mathsf{type}}(E, P) \subseteq \mathsf{P}'$.

$P = a[P']$ Let $\Pi = \mathscr{R}_{\mathsf{env}}(E, P')$ and $\mathsf{P} = \mathscr{R}_{\mathsf{type}}(E, P')$. By construction there exists A such that $E(a) = A$, and by induction hypothesis $\Pi, E \vdash_{\mathscr{A}} P' : \mathsf{P}$. Then also $\Pi \vdash \diamond$, and $\Pi \vdash \mathsf{P}$ closed. Let $\Pi^\star$ be defined as in the side condition of (**R**-AMB) and set $\mathsf{P}^\star = \mathsf{ProcClosure}(\mathsf{P}, \Pi^\star)$. By the construction of $\Pi^\star$, noting that EnvClosure is idempotent, we have

$$\mathsf{P}^\star = \mathsf{ProcClosure}(\mathsf{P}, \Pi^\star) \tag{9}$$

$$\Pi^\star = \mathsf{DomClosure}(\mathsf{P}^\star, A, \Pi) \tag{10}$$

$$\Pi^\star = \mathsf{EnvClosure}(\Pi^\star) \tag{11}$$

From (11) and Proposition C.1, we deduce $\Pi^\star \vdash \diamond$. From this, (ENV), and (NAME) we obtain:

$$\Pi^\star, E \vdash a : A \tag{12}$$

By construction $\Pi \subseteq \Pi^\star$. Thus by (9), the induction hypothesis, and Lemma C.5 we deduce

$$\Pi^\star, E \vdash_{\mathscr{A}} P' : \mathsf{P}^\star \tag{13}$$

Finally from (10) and Proposition C.2(3) we have

$$\Pi^\star \vdash A \ \mathsf{bounds}\ \mathsf{P}^\star \tag{14}$$

The result follows from (12), (13), and (14) by (AMB). For completeness, consider any Π' and P' such that $\Pi', E \vdash_{\mathscr{A}} P' : \mathsf{P}'$ and redo the proof above using Π' and P' instead of Π and P. The result follows from the monotonicity of EnvClosure, ProcClosure, and DomClosure. □

D Generalized Type Safety

The generalized version of type safety, for processes in arbitrary form, is subtler and requires more complex definitions. The problem is that restrictions may extrude tagged processes and thus inherently change the set of actions exhibited by the latter. For example consider the following process:

$$b[(\boldsymbol{\nu} c{:}C)\, a[\mathsf{out}\ b.\mathsf{in}\ c] \mid \ldots] \tag{15}$$

Imagine that we want to consider the set of residuals of b. According to the actual definition we have to tag b, that is, $\sharp b[(\boldsymbol{\nu} c{:}C)\, a[\mathsf{out}\ b.\mathsf{in}\ c] \mid \ldots]$, and apply the reduction rule **(open tag)**. But to apply this rule we must first extrude the restriction $(\boldsymbol{\nu} c{:}C)$ from b. The final result is:

$$(\boldsymbol{\nu} c{:}C)(\sharp b[\ldots] \mid \sharp a[\mathsf{in}\ c])$$

Now according to the previous definitions $a[\mathsf{in}\ c]$ is a residual of b, and the former emits $\mathsf{in}\ c$ (more precisely $a[\mathsf{in}\ c] \downarrow \mathsf{in}\ c^{=}$). However it is clear that b cannot emit $\mathsf{in}\ c^{=}$ as the extruded restrictions always blocks this action. And indeed the type system does not require $\mathsf{in}\ C$ to belong to the type of b.

The above example shows that scope extrusion requires that extruded restrictions are traced by the extruded tags. Thus, the general form of tagged processes will be $\sharp_E P$, where E is a type environment. Given the extended notion of tags, we then define a congruence rule for scope extrusion:

$$\sharp_E(\boldsymbol{\nu} a{:}D)P \quad \equiv \quad (\boldsymbol{\nu} a{:}D)\sharp_{E,a:D}P \tag{16}$$

In the following, we omit the type environment in tags unless it really matters. The tagged-reduction rules and the remaining structural congruence rules are as before, with the only exceptions that now tags carry type environments with them.

The definition of $\mathscr{C}[\,]$ must be extended to include restrictions:

$$\mathscr{C}[\,] \;::=\; [\,] \;\big|\; P \mid \mathscr{C}[\,] \;\big|\; \mathscr{C}[\,] \mid P \;\big|\; a[\mathscr{C}[\,]] \;\big|\; \alpha.\mathscr{C}[\,] \;\big|\; (\boldsymbol{\nu} a{:}D)\mathscr{C}[\,]$$

Given a context $\mathscr{C}[\,]$ we denote by $E_{\mathscr{C}}$ the type environment formed by all the declarations introduced in the context by $\boldsymbol{\nu}$'s that have the context's hole in their scope. For brevity we use E^P_Δ to denote $E_{\mathscr{C}^P_\Delta}$.

We can now state the new definition of set of *residuals*, which is modified so that type-environments annotations are traced during the reduction. For this reason residuals will be tagged processes rather than just processes:

Definition D.1 (Residuals). Let P be a process.

1. Let Δ be an occurrence of an untagged process P and E a type environment. The set of *E-residuals of Δ in P* is defined as follows:
(1) $\sharp_E P_\Delta$ is an E-residual of Δ in P
(2) If $\mathscr{C}^P_\Delta[\sharp_E P_\Delta] \rightarrowtail Q$ and $Q_{\Delta'} = \sharp_{E'} R$ for some R, then every E'-residual of Δ' in $|Q|$ is also an E-residual of Δ in P.
2. Let Δ be an occurrence of an untagged process P. The set of *residuals of* Δ is the set of $\varnothing$-residuals of Δ in P. □

We extend the type system with an additional type rule for tagged processes and define the $\downarrow$ relation also for tagged processes:

$$\begin{array}{ccc}
\text{(TYPE TAG)} & & \\
\dfrac{\Pi, E \vdash P : \mathsf{P}}{\Pi, E \vdash \sharp_{E'} P : \mathsf{P}} & \qquad & \dfrac{P \downarrow \mathsf{cap}\ a^\eta}{\sharp_E P \downarrow \mathsf{cap}\ a^\eta}\ a \notin \mathsf{Dom}(E)
\end{array}$$

The way capability exhibition is defined for tagged processes justifies why residuals are now defined as tagged processes and why tags have to store environments: if we did not, then by the rule (16) a residual could exercise a capability that in the original occurrence would have been blocked by a restriction. If we consider again the example (15) and the set of $\varnothing$-residuals of b, then this set contains $\sharp_{c:C}\, a[\mathsf{in}\ c]$ which, according to the new rule for $\downarrow$ defined above, no longer emits $\mathsf{in}\ c^{=}$.

Finally, the definition of $\Downarrow$ is as before, but now it uses the new definitions of exhibition and residual.

Definition D.2 (Residual Behavior). Let P be a process, Δ and an occurrence of P. $\Delta \Downarrow \alpha^\eta$ if and only if $Q \downarrow \alpha^\eta$, for some residual Q of Δ. □

The general version of Theorem 4.3 stated for generic processes holds for this new definition of $\Downarrow$.

Theorem D.3 (General Type Safety). *Let P be a process and Δ be an occurrence of P. Assume that $\Pi, E \vdash P : \mathsf{P}'$ and $\Pi, E \cdot E^P_\Delta \vdash P_\Delta : \mathsf{P}$. If $\Delta \Downarrow (\mathsf{cap}\ a)^\eta$, then $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$.* □

To prove it we must first lift the subject reduction theorem to tagged processes and tagged reduction.

Theorem D.4 (Tagged Subject Reduction). *Let P be a tagged process. If $\Pi, E \vdash P : \mathsf{P}$ and $P \rightarrowtail Q$, then $\Pi, E \vdash Q : \mathsf{P}$.* □

Then the General Type Safety theorem follows from an analogue of Lemma B.2 on tagged processes, and the following version of Lemma B.1.

Lemma D.5. *Let P be an untagged process and Δ an occurrence of P. Assume that $\Pi, E \vdash P : \mathsf{P}$ and $\Pi, (E \cdot E^P_\Delta) \vdash P_\Delta : \mathsf{P}_1$. If $\mathscr{C}^P_\Delta[\sharp_{E_1} P_\Delta] \rightarrowtail \mathscr{C}_1[\sharp_{E_2} P_2]$, for some context $\mathscr{C}_1$, then $\Pi, (E \cdot E_{\mathscr{C}_1}) \vdash \sharp_{E_2} P_2 : \mathsf{P}_1$.*

Proof. (Sketch) The proof is in two steps. First we prove that the claim holds for structural congruence, i.e. that if $\mathscr{C}^P_\Delta[\sharp_{E_1} P_\Delta] \equiv \mathscr{C}'_1[\sharp_{E_3} P_3]$, then $\Pi, (E \cdot E_{\mathscr{C}'_1}) \vdash \sharp_{E_3} P_3 : \mathsf{P}_1$. This follows by a case analysis on the possible occurrences of $\sharp_{E_1} P_\Delta$: the proof makes a crucial use of the assumption that P is untagged and that therefore $\sharp_{E_1} P_\Delta$ is the only tagged occurrence of the starting processes (if we had several tags then the statement would not hold because of the rule $\sharp P \mid \sharp Q \equiv \sharp(P \mid Q)$ which could then be applied from right to left making other tags "pollute" the tagged occurrence considered in the statement).

Then, we observe all possible one step reductions starting from $\mathscr{C}'_1[\sharp_{E_3} P_3]$ and ending into $\mathscr{C}_1[\sharp_{E_2} P_2]$. This part of the proof is very much the same as the corresponding part in the proof of Lemma B.1, once we note that if $\sharp_{E_3} P_3$ is directly issued from $\sharp_{E_2} P_2$, then $E_{\mathscr{C}_1} = E_{\mathscr{C}'_1}$. □