Secure Safe Ambients

Michele Bugliesi
Universitá “Ca’ Foscari”, Venice
michele@dsi.unive.it

Giuseppe Castagna
C.N.R.S, École Normale Supérieure, Paris
Giuseppe.Castagna@ens.fr

Abstract. *Secure Safe Ambients* (SSA) are a typed variant of *Safe Ambients* [9], 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 powerful 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 [5] are named agents or locations that enclose collections of running processes, possibly including nested subambients. *Safe Ambients* [9] 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:

$$\textit{Mobile Ambients}\quad a[\texttt{open}\, b.\texttt{in}\, c] \mid b[\texttt{in}\, a.\texttt{in}\, d].$$

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* $\texttt{in}\, a$, and reduce to $a[\texttt{open}\, b.\texttt{in}\, c \mid b[\texttt{in}\, d]]$. Then a may dissolve the boundary provided by b by exercising $\texttt{open}\, b$, and reduce to $a[\texttt{in}\, c \mid \texttt{in}\, d]$.

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:

$$\textit{Safe Ambients}\quad a[\overline{\texttt{in}}\, a.\texttt{open}\, b.\texttt{in}\, c] \mid b[\texttt{in}\, a.\overline{\texttt{open}}\, b.\texttt{in}\, d].$$

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

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 essential 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:

$$\begin{array}{lcl} a[\overline{\texttt{in}}\, a.\texttt{open}\, b.\texttt{in}\, c] & \mid & b[\texttt{in}\, a.\overline{\texttt{open}}\, b.\texttt{in}\, d] \\ & \mid & c[\,\overline{\texttt{in}}\, c.P \mid d[\overline{\texttt{in}}\, d.Q]\,] \end{array}$$

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 $\overline{\texttt{in}}\, d$), 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 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:

$$\begin{array}{lcl} a[\overline{\texttt{in}}\, a.\texttt{in}\, c.\overline{\texttt{out}}\, a] & \mid & b[\texttt{in}\, a.\texttt{out}\, a.\texttt{in}\, d] \\ & \mid & c[\,\overline{\texttt{in}}\, c.P \mid d[\overline{\texttt{in}}\, d.Q]\,] \end{array}$$

Again, if we only look at the capabilities available for a, we are

Proc. of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages London, January 2001

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. □

1.1 Overview and Main Results

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 [9], 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 *domains*: each domain has an associated *behavior* that ambients in the domain share and must comply with, and a *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 an ambient of domain, say A, and P is a (well-typed) process exercising the term-level capability $\mathtt{in}\ a$, then the type of P traces this behavior by including the type-level capability $\mathtt{in}\ A$.

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.

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. 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.

Finally, we study 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*.

1.2 Related work

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 [6], where types are introduced to discipline the exchange of values inside ambients. In [3], Cardelli, Ghelli and Gordon extend the type system of [6] 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 [9], 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.

The type system closest to ours is the one presented by Cardelli, Ghelli, and Gordon in their recent paper on Ambient Groups [4]. 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 [4] 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 [4]: only the first of the two attacks we discussed in the examples above are detected by the type system of [4]. A further difference is the presence in [4] 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 [4], type reconstruction and the distributed version of the system (neither of which is discussed in [4]) represent further important differences between the two papers.

Further related work includes F. and H.R. Nielson's framework for control and data flow analysis for Mobile Ambients [12, 13]: 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.

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. A short section concludes the presentation. Proofs of the main results are given in a separate appendix.

2. THE LANGUAGE

The terms of our language are those of *Safe Ambients* with the only difference that the types of (ambient) names are (*protection*) *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

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

Capabilities

$$\alpha ::= \mathtt{in}\, a \;\big|\; \overline{\mathtt{in}}\, a \;\big|\; \mathtt{out}\, a \;\big|\; \overline{\mathtt{out}}\, a \;\big|\; \mathtt{open}\, a \;\big|\; \overline{\mathtt{open}}\, a$$

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 [8].

Reduction

The reduction relation for SSA derives from the one defined for *Safe Ambients*.

(in)	$b[\mathtt{in}\ a.P \mid Q] \mid a[\overline{\mathtt{in}}\ a.R \mid S] \rightarrow a[R \mid S \mid b[P \mid Q]]$
(out)	$a[\, b[\mathtt{out}\, a.P \mid Q] \mid \overline{\mathtt{out}}\, a.R \mid S] \rightarrow b[P \mid Q] \mid a[R \mid S]$
(open)	$\mathtt{open}\ a.P \mid a[\overline{\mathtt{open}}\ a.Q \mid R] \rightarrow P \mid Q \mid R$
(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 follows:

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 $\mathbf{0}$ and $\mid$ and closed under the following rules:

$!P \equiv !P \mid P$	
$(\boldsymbol{\nu} a{:}D)\mathbf{0} \equiv \mathbf{0}$	
$(\boldsymbol{\nu} a{:}A)(\boldsymbol{\nu} b{:}B)P \equiv (\boldsymbol{\nu} b{:}B)(\boldsymbol{\nu} a{:}A)P$	for $a \neq b$
$(\boldsymbol{\nu} a{:}D)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} a{:}D)Q$	for $a \notin fn(P)$
$(\boldsymbol{\nu} a{:}D)b[P] \equiv b[(\boldsymbol{\nu} a{:}D)P]$	for $a \neq b$

3. TYPE SYSTEM

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

Type Capabilities

$$M ::= \mathtt{in}\ D \mid \overline{\mathtt{in}}\ D \mid \mathtt{out}\ D \mid \overline{\mathtt{out}}\ D \mid \mathtt{open}\ D \mid \overline{\mathtt{open}}\ D$$

Process Types

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

Notation. The following conventions are used throughout. We often write $\mathtt{cap}\ D$ (resp. $\mathtt{cap}\ a$) to denote an arbitrary type-level (resp. 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 $\overline{M}$, and two sets of type capabilities $\mathbf{L}$ and $\mathbf{M}$, we write $M \in \mathtt{sync}(\mathbf{L}, \mathbf{M})$ as a shorthand for $M \in \mathbf{L}$ and $\overline{M} \in \mathbf{M}$.

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: $\mathsf{P}^{\uparrow}$ 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

[1] One problem with that solution is that types are not preserved by structural congruence. For instance, the term $(\boldsymbol{\nu} a{:}A)(\boldsymbol{\nu} b{:}B)\, a[\mathtt{in}\ b] \mid b[\overline{\mathtt{in}}\ b]$ would not be typeable, as the type A should contain all the capabilities a can exercise: yet A cannot contain $\mathtt{in}\ b$, as b is in the scope of a nested binder. If we exchange the position of the two binders, as in $(\boldsymbol{\nu} b{:}B)(\boldsymbol{\nu} a{:}A)\, a[\mathtt{in}\ b] \mid b[\overline{\mathtt{in}}\ b]$ 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).

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

of the form $a[P']$. To exemplify, given $a : A$:

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

Type Envs	E	:	*Ambient Names* $\rightarrow$ *Ambient Domains*
Type Envs	Π	:	*Ambient Domains* $\rightarrow$ *Process Types*

Type environments associate with each ambient name the domain it belongs to, while domain environments associate with 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:

$$\Pi \vdash \mathsf{P}\ \mathsf{closed} \triangleq$$
$$\mathtt{open}\ H \in \mathsf{sync}(\mathsf{P}^{=}, \Pi(H)^{=}) \Rightarrow \Pi(H) \subseteq \mathsf{P}$$

$$\Pi \vdash D\ \mathsf{bounds}\ \mathsf{P} \triangleq$$
$$\mathsf{P}^{\uparrow} \subseteq \Pi(D)^{=}\ \wedge\ \mathsf{P}^{=} \subseteq \Pi(D)^{\downarrow} \wedge$$
$$(\overline{\mathtt{open}}\ D \in \Pi(D)^{=} \Rightarrow \mathsf{P} \subseteq \Pi(D))$$

$$\Pi \vdash D\ \mathsf{closed} \triangleq$$
$$\begin{cases} \mathtt{in}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{=}) \Rightarrow \Pi \vdash H\ \mathsf{bounds}\ \Pi(D) \\ \mathtt{out}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{\downarrow}) \Rightarrow \Pi(D) \subseteq \Pi(H) \end{cases}$$ □

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).

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 $\mathtt{in}\ a$, $\mathtt{out}\ a$, $\overline{\mathtt{in}}\ a$ and $\overline{\mathtt{open}}\ a$ may be observed at the level of the enclosing ambient. Dually, $\mathtt{open}\ a$, and $\overline{\mathtt{out}}\ a$ may be observed at the level of the continuation process.

As for (AMB), the rule stipulates that an ambient $a[P]$ has the type that Π associates with the domain D of a, 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.

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

$$a[\overline{\mathtt{in}}\ a.\mathtt{open}\ b.\mathtt{in}\ c] \mid b[\mathtt{in}\ a.\overline{\mathtt{open}}\ b.\mathtt{in}\ d].$$

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

A similar analysis applies to the attack

$$a[\overline{\mathtt{in}}\ a.\mathtt{in}\ c.\overline{\mathtt{out}}\ a] \mid b[\mathtt{in}\ a.\mathtt{out}\ a.\mathtt{in}\ d].$$

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

4. TYPE SAFETY

The operational significance 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

(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)
$$\frac{\Pi, E \vdash \diamond}{\Pi, E \vdash \mathbf{0} : (\varnothing, \varnothing, \varnothing)}$$

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

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

(ACTION$^{\uparrow}$)
$$\frac{\Pi \vdash P{:}\mathsf{P} \quad \Pi, E \vdash a{:}D \quad \mathsf{cap}\ D \in \mathsf{P}^{\uparrow}}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ \mathsf{cap} \in \{\mathtt{in}, \overline{\mathtt{in}}, \mathtt{out}, \overline{\mathtt{open}}\}$$

(ACTION$^{=}$)
$$\frac{\Pi, E \vdash P{:}\mathsf{P} \quad \Pi, E \vdash a{:}D \quad \mathsf{cap}\ D \in \mathsf{P}^{=}}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ \mathsf{cap} \in \{\overline{\mathtt{out}}, \mathtt{open}\}$$

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

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

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

Figure 1: Typing Rules

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 [14]. 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.

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 the interactions considered in the Chinese Wall Security Policy [1]: in particular we can trace all the processes that "got in touch" with that ambient. Tags are propagated based on the idea of an ambient as a paint pot: any ambient exiting a tagged ambient is tagged:

$$\sharp a[\, b[\mathtt{out}\ a.P \mid Q] \mid \overline{\mathtt{out}}\ a.R \mid S\,] \ \rightarrowtail\ \sharp b[P \mid Q] \mid \sharp a[R \mid S]$$

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

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

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

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

Technically, the definition is only slightly more complex. First, 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[3]:

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

Second, 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.

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:

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

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

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[4] 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:

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

[4] 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, $|P_\Delta| = |P|_\Delta$.

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

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

Figure 2: Exhibiting a capability

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

Figure 3: Tag Propagation via Reduction

(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 $R \downarrow \alpha^{\eta}$. □

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 $a[\overline{\mathtt{in}}\, a.\mathtt{open}\ b]$. If taken in isolation, this ambient only exhibits the capabilities $\overline{\mathtt{in}}\ a$ and $\mathtt{open}\ b$. If, instead, we take the parallel composition

$$a[\overline{\mathtt{in}}\, a.\mathtt{open}\ b] \mid b[\mathtt{in}\ a.\overline{\mathtt{open}}\, b.\mathtt{in}\ c] \qquad (1)$$

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

$$\begin{array}{l}
\sharp a[\overline{\mathtt{in}}\, a.\mathtt{open}\ b] \mid b[\mathtt{in}\ a.\overline{\mathtt{open}}\, b.\mathtt{in}\ c] \\
\qquad\qquad \rightarrowtail\ \ \sharp a[\mathtt{open}\ b \mid b[\overline{\mathtt{open}}\, b.\mathtt{in}\ c]] \\
\qquad\qquad \rightarrowtail\ \ \sharp a[\mathtt{in}\ c]
\end{array}$$

Now, Theorem 4.3 ensures that if we type the process (1), the fact that the residual $a[\mathtt{in}\ c]$ of a exhibits $\mathtt{in}\ c$ 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 $\sharp a[\overline{\mathtt{in}}\, a.\overline{\mathtt{out}}\, b] \mid b[\mathtt{in}\ a.\mathtt{out}\ a.\mathtt{in}\ c]$, the result of tagged reduction is $\sharp a[\,] \mid \sharp b[\mathtt{in}\ c]$, and Theorem 4.3 ensures that the type of (the domain of) a traces the type-level capability corresponding to $\mathtt{in}\ c$, since it is exhibited by the residual $b[\mathtt{in}\ c]$.

5. ALGORITHMIC SYSTEMS

As it is usual in the presence of subsumption, the type system given in Figure 1 is not algorithmic: however, it is easy to state the type rules so that they form a syntax-directed 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 erasing the rule of subsumption, and replacing the (ACTION) and (PAR) rules with the rules in Figure 4: the only subtlety is the side condition to the rule (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).

These rules constitute the core of an algorithm that given Π, E, and P as input, returns the type P as output.

Theorem 5.1 (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 $\mathsf{P}' \subseteq \mathsf{P}$* □

Corollary 5.2 (Minimal typing). $\Pi, E \vdash_{\mathscr{A}} P : \min\{\mathsf{P} \mid \Pi, E \vdash P : \mathsf{P}\}$ *if this set is non-empty.* □

The existence of minimum 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 the behavior of every ambient (and

$$\textsc{(Par)}\quad \frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad \Pi, E \vdash_{\mathscr{A}} Q : \mathsf{Q}}{\Pi, E \vdash_{\mathscr{A}} P \mid Q : \mathsf{P} \cup \mathsf{Q}}
\qquad
\textsc{(Action}_1^{=}\textsc{)}\quad \frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \overline{\mathtt{out}}\ a.P : \mathsf{P} \cup^{=} \{\overline{\mathtt{out}}\ A\}}$$

$$\textsc{(Action}_2^{=}\textsc{)}\quad \frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathtt{open}\ a.P : \mathsf{P}'}
\qquad
\mathsf{P}' \triangleq \mathsf{P} \cup^{=} \{\mathtt{open}\ A\} \cup \begin{cases} \Pi(A) & \text{if } \overline{\mathtt{open}}\ A \in \Pi(A)^{=} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\textsc{(Action}^{\uparrow}\textsc{)}\quad \frac{\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P} \quad E(a) = A}{\Pi, E \vdash_{\mathscr{A}} \mathsf{cap}\ a.P : \mathsf{P} \cup^{\uparrow} \{\mathsf{cap}\ A\}}
\qquad
\mathsf{cap} \in \{\mathtt{in}\,, \overline{\mathtt{in}}\,, \mathtt{out}\,, \overline{\mathtt{open}}\,\}$$

Figure 4: Algorithmic Typing

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 prospective use of types to check and enforce security, as the coarser the approximation of a process' behavior, the less likely is that 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 5.3 (Closure). Let Π be a domain environment such that $fn(\Pi) \subseteq \mathsf{Dom}(\Pi)$. Then define:

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

$$\mathsf{ProcClosure}(\mathsf{P}, \Pi) \triangleq \mathsf{P} \cup \bigcup\{\Pi(A) \mid \mathtt{open}\, A \in \mathsf{sync}(\mathsf{P}^{=}, \Pi(A)^{=})\}$$

$$\mathsf{DomClosure}(\mathsf{P}, A, \Pi) \triangleq$$
$$\bigcap\{\Pi' \mid \Pi' \supseteq \Pi\,, \Pi'(A) \supseteq (\varnothing, \mathsf{P}^{\uparrow}, \mathsf{P}^{=}) \cup \mathsf{P}'\}$$
$$\text{where } \mathsf{P}' = \begin{cases} \mathsf{P} & \text{if } \overline{\mathtt{open}}\ A \in \Pi'(A)^{=} \\ \varnothing & \text{otherwise} \end{cases} \qquad \square$$

It is easy to see that all these operators are well-defined and monotone: furthermore they can be effectively computed by (always terminating) algorithms: an example is given in Figure 7.

The system for type reconstruction is defined in Figure 5: the ($\mathbf{R}$-Action) rules are the same as the corresponding algorithmic (Action) rules, ($\mathbf{R}$-Repl) and ($\mathbf{R}$-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 ($\mathbf{R}$-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 $fn(P) \subseteq \mathsf{Dom}(E)$ 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 typed restrictions of 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 5.4 (Soundness and Completeness). *Let P be a process, and E a type environment such that $fn(P) \subseteq \mathsf{Dom}(E)$. Then $\mathscr{R}_{env}(E, P), E \vdash_{\mathscr{A}} P : \mathscr{R}_{type}(E, P)$. Furthermore, for any Π and P such that $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$, one has $\mathscr{R}_{env}(E, P) \subseteq \Pi$ and $\mathscr{R}_{type}(E, P) \subseteq \mathsf{P}$.* $\square$

Corollary 5.5 (Minimality). *Let P be a process and E a type environment such that $fn(P) \subseteq \mathsf{Dom}(E)$. Then*

$$(\mathscr{R}_{env}(E, P), \mathscr{R}_{type}(E, P)) = \min\{(\Pi, \mathsf{P}) \mid \Pi, E \vdash P : \mathsf{P}\} \qquad \square$$

Accordingly, in the typed syntax it is enough to specify the domains of the ambients occurring in P, and the associated security constraints: 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 Envs Σ : *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) $\{A \mid \mathtt{in}\ D \in \mathsf{sync}(\Pi(A)^{=}, \Pi(D)^{=})\} \subseteq \Sigma(D).\mathsf{in}$, and
(ii) $\{A \mid \mathtt{out}\ D \in \mathsf{sync}(\Pi(A)^{=}, \Pi(D)^{\downarrow})\} \subseteq \Sigma(D).\mathsf{out}$.

(**R**-Dead)

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

(**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)

$$\frac{\Pi, E \vdash_{\mathscr{D}} P : \mathsf{P} \quad E(a) = A}{\Pi^\star, E \vdash_{\mathscr{D}} a[P] : \Pi^\star(A)} \qquad \Pi^\star \triangleq \bigcap\{\Pi'' \mid \Pi'' \supseteq \Pi, \Pi'' = \mathsf{EnvClosure}(\Pi'),\ \Pi' = \mathsf{DomClosure}(\mathsf{P}', A, \Pi), \mathsf{P}' = \mathsf{ProcClosure}(\mathsf{P}, \Pi'')\}$$

Figure 5: Type Reconstruction Algorithm

The security model arising from domain constraints is related to 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[5]. 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 $\mathsf{P}^{\downarrow}_{\mathsf{in}}$, $\mathsf{P}^{=}_{\mathsf{in}}$, and $\mathsf{P}^{\uparrow}_{\mathsf{in}}$ 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 $\mathsf{P}^{\uparrow}_{\mathsf{out}}$, $\mathsf{P}^{=}_{\mathsf{out}}$, and $\mathsf{P}^{\downarrow}_{\mathsf{out}}$ defined as above.

In this option $\Pi \models \Sigma$ if and only if, for all A, B in $\mathsf{Dom}(\Pi)$, $\mathtt{in}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{=})$ implies $\Pi(B) \subseteq \Sigma(A).\mathsf{in}$, and, $\mathtt{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$ implies $\Pi(B) \subseteq \Sigma(A).\mathsf{out}$ Capability constraints are loosely related to the *permission collections* used in the the JDK 1.2 architecture (a.k.a 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

$$\phi \quad ::= \quad M \in D^{\eta} \;\mid\; \neg\phi \;\mid\; \phi \wedge \phi \;\mid\; \phi \vee \phi \;\mid\; \forall D : \phi$$

Semantics

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

[5]Alternatively, we could define what ambients should not be allowed to do, but our choice complies with well-established security principles [7].

The notion of formula satisfiability is easily extended to 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 $\mathtt{cap}D \in \mathsf{sync}(\mathbf{L}, \mathbf{M})$. Based on that, we can express refined security properties: for example, the formula $\forall B, C : \mathtt{in}\ D \in \mathsf{synch}(B^{=}, D^{=}) \wedge \mathtt{in}\ B \in \mathsf{synch}(C^{=}, B^{=}) \Rightarrow \mathtt{in}\ D \in C^{=}$ 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

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

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 (*type as-*

sumptions) [10]. 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 of 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 6.

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 6:

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

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 $\Pi \vdash E(a)$ 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.[6]

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* [10].

[6] 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.

$b[\mathtt{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} : \Pi(E(b))$ **:** 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 foreing code: verification is based on the (type) information foreing code carries along with it, according to the common proof-carrying-code practice [11].

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: 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 $P_\Delta \Downarrow (\mathsf{cap}\ b)^\eta$, then $\mathsf{cap}\ E(b) \in \mathsf{P}^\eta$.* $\square$

The difference from Theorem 4.3 is that the new statement 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[7].

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

[7] 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.

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

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

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

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

Figure 6: New reduction rules for DSSA

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. CONCLUSIONS

We have showed that classical type theoretic techniques provide effective tools for characterizing behavioral properties of mobile agents. We hope to have convinced the reader that capturing implicit behavior is essential to ensure secure agent interactions: to our knowledge, our type system 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 for the calculus, one finds back features distinctive of real systems.

There are several directions for future work. A first, obvious extension is to study whether and how our techniques scale to a calculus with communications. A second interesting subject of study is the use of multi-sets (or even traces) in place of sets as basic components of process types: this would allow us to refine the analysis of process behavior and, consequently, to enforce more powerful security policies. Also interesting would be to apply these techniques to the Seal Calculus [15].

A further subject of future research is the study of a notion of “subtyping” on ambient domains. This would allow us to introduce in the system a notion of security levels and perform static analyses such as those described in [2].

Acknowledgments

Work partially supported by Franco-Italian Action “Galileo” 1999-2000. Support was also provided by the Italian MURST Project 9901403824_003 “Automatic Program Certification by Abstract Interpretation” and by the French CNRS Program *Telecommunications*: “Collaborative, distributed, and secure programming for Internet”.

9. REFERENCES

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

[2] H. R. N. C. Bodei, P. Degano and F. Nielson. Static analysis of processes for no read-up and no write-down. In *Porceedins of FoSSaCS’99*. 1999.

[3] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP’99*, LNCS 1644, pages 230–239. 1999.

[4] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *Int. Conf. IFIP TCS*, LNCS 1872, pages 333–347. 2000.

[5] L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL’98*. ACM Press, 1998.

[6] L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL’99*, pages 79–92. ACM Press, 1999.

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

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

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

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

[11] G. Necula. Proof carrying code. In A. Press, editor, *POPL ’97*, 1997.

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

[13] H. R. Nielson and F. Nielson. Shape analysis for mobile ambients. In *POPL’00*, pages 135–148. ACM Press, 2000.

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

[15] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, LNCS 1686, 1999.

APPENDIX

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 there exist $\mathsf{P}_1, \mathsf{P}_2 \subseteq \mathsf{P}$ such that $\Pi, E \vdash P : \mathsf{P}_1$ and $\Pi, E \vdash Q : \mathsf{P}_2$;*
3. *If $\Pi, E \vdash !P : \mathsf{P}$, then there exists $\mathsf{P}' \subseteq \mathsf{P}$ such that $\Pi, E \vdash P : \mathsf{P}'$;*
4. *If $\Pi, E \vdash$ cap $a.P : \mathsf{P}$, then there exists $\mathsf{P}' \subseteq \mathsf{P}$ such that $\Pi, E \vdash P : \mathsf{P}'$, and $\Pi, E \vdash a : A$ for some ambient domain A. Furthermore, either (i) cap $\in \{\mathtt{in}, \overline{\mathtt{in}}, \mathtt{out}, \overline{\mathtt{open}}\}$ and cap $A \in \mathsf{P}'^{\uparrow}$, or (ii) cap $\in \{\overline{\mathtt{out}}, \mathtt{open}\}$ and cap $A \in \mathsf{P}'^{=}$*
5. *If $\Pi, E \vdash (\boldsymbol{\nu} a{:}D)P : \mathsf{P}$, then there exists $\mathsf{P}' \subseteq \mathsf{P}$ such that $\Pi, E, a{:}D \vdash P : \mathsf{P}'$*
6. *Assume $\Pi, E \vdash a[P] : \mathsf{P}$. Then $\Pi, E \vdash a[P] : \Pi(E(a))$, $\Pi(E(a)) \subseteq \mathsf{P}$, and there exists P' such that $\Pi, E \vdash P : \mathsf{P}'$, and $\Pi \vdash E(a)$ bounds P'.*

Proof. In each case, directly by induction on the derivation of the judgment in the hypothesis. □

Lemma A.6 (Subject Congruence).

1. *If $\Pi, E \vdash P : \mathsf{P}$ and $P \equiv Q$, then $\Pi, E \vdash Q : \mathsf{P}$.*
2. *If $\Pi, E \vdash P : \mathsf{P}$ and $Q \equiv P$, then $\Pi, E \vdash Q : \mathsf{P}$.*

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

Theorem A.7 (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):
$\mathtt{open}\, a.P_1 \mid a[\overline{\mathtt{open}}\, a.P_2 \mid P_3] \rightarrow P_1 \mid P_2 \mid P_3$
From $\Pi, E \vdash \mathtt{open}\, a.P_1 \mid a[\overline{\mathtt{open}}\, a.P_2 \mid P_3] : \mathsf{P}$, 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 process types $\mathsf{P}_1 \subseteq \mathsf{P}$, and $\mathsf{P}_2, \mathsf{P}_3, \mathsf{P}_{23}$ with $\mathsf{P}_2, \mathsf{P}_3 \subseteq \mathsf{P}_{23}$ such that the following are all verified:

$$\Pi, E \vdash P_1 : \mathsf{P}_1 \tag{2}$$
$$\Pi, E \vdash \overline{\mathtt{open}}\, a.P_2 \mid P_3 : \mathsf{P}_{23} \tag{3}$$
$$\Pi, E \vdash P_2 : \mathsf{P}_2 \quad \text{and} \quad \Pi, E \vdash P_3 : \mathsf{P}_3 \tag{4}$$
$$\Pi, E \vdash a : D \quad \text{and} \quad \Pi \vdash D \text{ bounds } \mathsf{P}_{23} \tag{5}$$

From the first judgment in (4), by Lemma A.5:4, we know that $\overline{\mathtt{open}}\, D \in \mathsf{P}_{23}^{\uparrow}$. From this, and from (5), we know that $\overline{\mathtt{open}}\, D \in \Pi(D)^{=}$, and hence $\mathsf{P}_{23} \subseteq \Pi(D)$ again from (5). Then $\mathsf{P}_{23} \subseteq \mathsf{P}$ since $\Pi(D) \subseteq \mathsf{P}$. By subsumption 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):
$a[\overline{\mathtt{in}}\, a.P_1 \mid P_2] \mid b[\mathtt{in}\, a.Q_1 \mid Q_2] \rightarrow a[P_1 \mid P_2 \mid b[Q_1 \mid Q_2]]$
From $\Pi, E \vdash a[\overline{\mathtt{in}}\, a.P_1 \mid P_2] \mid b[\mathtt{in}\, a.Q_1 \mid Q_2] : \mathsf{P}$, 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 $\Pi(A), \Pi(B) \subseteq \mathsf{P}$, process types $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_{12}$ with $\mathsf{P}_1, \mathsf{P}_2 \subseteq \mathsf{P}_{12}$, and $\mathsf{Q}_1, \mathsf{Q}_2, \mathsf{Q}_{12}$ with $\mathsf{Q}_1, \mathsf{Q}_2 \subseteq \mathsf{Q}_{12}$ such that the following are all verified:

$$\Pi, E \vdash \mathtt{in}\, a.Q_1 \mid Q_2 : \mathsf{Q}_{12} \tag{1}$$
$$\Pi, E \vdash Q_1 : \mathsf{Q}_1 \quad \text{and} \quad \Pi, E \vdash Q_2 : \mathsf{Q}_2 \tag{2}$$
$$\Pi, E \vdash b : B \quad \text{and} \quad \Pi \vdash B \text{ bounds } \mathsf{Q}_{12} \tag{3}$$

$$\Pi, E \vdash \overline{\mathtt{in}}\, a.P_1 \mid P_2 : \mathsf{P}_{12} \tag{4}$$
$$\Pi, E \vdash P_1 : \mathsf{P}_1 \quad \text{and} \quad \Pi, E \vdash P_2 : \mathsf{P}_2 \tag{5}$$
$$\Pi, E \vdash a : A \quad \text{and} \quad \Pi \vdash A \text{ bounds } \mathsf{P}_{12} \tag{6}$$

From the left judgments of (2) and (6), by Lemma A.5:4, we know that $\mathtt{in}\, A \in \mathsf{Q}_{12}^{\uparrow}$. From this and from (3), $\mathtt{in}\, A \in \Pi(B)^{=}$. From the left judgment of (5), we also know that $\overline{\mathtt{in}}\, A \in \mathsf{P}_{12}^{\uparrow}$. From this and from (6), $\overline{\mathtt{in}}\, A \in \Pi(A)^{=}$. Summarizing we have, $\mathtt{in}\, A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{=})$. From this, and from $\Pi \vdash \diamond$, we know that $\Pi \vdash A$ bounds $\Pi(B)$. From this, and from the right judgment of (6), by Proposition A.1.1, we have

$$\Pi \vdash A \text{ bounds } (\Pi(B) \cup \mathsf{P}_{12}) \tag{7}$$

From the two judgments in (2), by subsumption (that can be applied by Lemma A.3) and (PAR), $\Pi, E \vdash Q_1 \mid Q_2 : \mathsf{Q}_{12}$. 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 subsumption and (PAR), $\Pi, E \vdash P_1 \mid P_2 : \mathsf{P}_{12}$. From (8) and the last judgment, by subsumption and (PAR),

$$\Pi, E \vdash P_1 \mid P_1 \mid b[Q_1 \mid Q_2] : \Pi(B) \cup \mathsf{P}_{12} \tag{9}$$

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

Case (out):

$a[\overline{\mathtt{out}}\ a.P_1 \mid P_2 \mid b[\mathtt{out}\ a.Q_1 \mid Q_2]] \rightarrow a[P_1 \mid P_2] \mid b[Q_1 \mid Q_2]$
As in the previous cases, by repeated applications of Lemma A.5 to the typing judgment of the redex, there exist process types $\mathsf{P}_1, \mathsf{P}_2$, P_a with $\mathsf{P}_1, \mathsf{P}_2 \subseteq \mathsf{P}_a$, and $\mathsf{Q}_1, \mathsf{Q}_2, \mathsf{Q}_{12}$ with $\mathsf{Q}_1, \mathsf{Q}_2 \subseteq \mathsf{Q}_{12}$, 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:

$$\Pi, E \vdash \mathtt{out}\ a.Q_1 \mid Q_2 : \mathsf{Q}_{12} \tag{1}$$
$$\Pi, E \vdash Q_1 : \mathsf{Q}_1 \quad \text{and} \quad \Pi, E \vdash Q_2 : \mathsf{Q}_2 \tag{2}$$
$$\Pi, E \vdash b : B \quad \text{and} \quad \Pi \vdash B \ \mathsf{bounds}\ \mathsf{Q}_{12} \tag{3}$$

$$\Pi, E \vdash \overline{\mathtt{in}}\ a.P_1 \mid P_2 \mid b[\mathtt{out}\ a.Q_1 \mid Q_2] : \mathsf{P}_a \tag{4}$$
$$\Pi, E \vdash P_1 : \mathsf{P}_1 \quad \text{and} \quad \Pi, E \vdash P_2 : \mathsf{P}_2 \tag{5}$$
$$\Pi, E \vdash a : A \quad \text{and} \quad \Pi \vdash A \ \mathsf{bounds}\ \mathsf{P}_a \tag{6}$$

From the left judgments of (2) and (6), by Lemma A.5:4, we know that $\mathtt{out}\ A \in \mathsf{Q}_{12}^{\uparrow}$. From this and from (3), $\mathtt{out}\ A \in \Pi(B)^{=}$. From the left judgment of (5), we also know that $\overline{\mathtt{out}}\ A \in \mathsf{P}_{12}^{=}$. From this and from (6), $\overline{\mathtt{out}}\ A \in \Pi(A)^{\downarrow}$.
Thus, $\mathtt{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$. 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 one step of tagged reduction from $\mathscr{C}[\sharp P]$: $\mathscr{C}[\sharp P] \equiv \mathscr{C}_1[\sharp R] \rightarrowtail \mathscr{C}_2[\sharp Q]$ 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 $\Pi, E \vdash |R| : \mathsf{P}$. 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 conguence 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 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, 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 suboccurrence 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) $\mathtt{open}\ a.S \mid \sharp a[\overline{\mathtt{open}}\ a.R_1 \mid R_2] \rightarrowtail S \mid \sharp(R_1 \mid R_2)$, where $R = a[\overline{\mathtt{open}}\ a.R_1 \mid R_2]$ and $Q = R_1 \mid R_2$.

From the hypothesis, we know that $\Pi, E \vdash a[\overline{\mathtt{open}}\ a.R_1 \mid R_2] : \Pi(E(a))$. From Lemma A.5:6, there exists P' such that $\Pi, E \vdash \overline{\mathtt{open}}\ a.R_1 \mid R_2 : \mathsf{P}'$ with $\Pi \vdash E(a)\ \mathsf{bounds}\ \mathsf{P}'$. By repeated applications of Lemma A.5 we also have that $\Pi, E \vdash R_1 \mid R_2 : \mathsf{Q} \subseteq \mathsf{P}'$. From this, and from $\overline{\mathtt{open}}\ E(a) \in \Pi(E(a))^{=}$, by closure it follows that $\mathsf{Q} \subseteq \mathsf{P}' \subseteq \Pi(E(a))$ as desired.

(out tag) $\sharp a[\, b[\mathtt{out}\ a.R_1 \mid R_2] \mid \overline{\mathtt{out}}\ a.R_3 \mid R_4] \rightarrowtail \sharp b[R_1 \mid R_2] \mid \sharp a[R_3 \mid R_4]$. The proof follows the pattern of the case (*out*) in the proof of Theorem 3.3.

(in) $b[\mathtt{in}\, a.S_1 \mid S_2] \mid \sharp a[\overline{\mathtt{in}}\, a.R_1 \mid R_2] \rightarrowtail \sharp a[R_1 \mid R_2 \mid b[S_1 \mid S_2]]$, where $R = a[\overline{\mathtt{in}}\ a.R_1 \mid R_2]$ 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 3.3. From the hypothesis, we know that $\Pi, E \vdash b[\mathtt{in}\ a.S_1 \mid S_2] : \Pi(E(b))$. 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
$\mathtt{in}\ E(a) \in \mathsf{sync}(\Pi(E(b))^{=}, \Pi(E(a))^{=})$. □

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. □

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. The proof 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 (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)** recutions. 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: $\Pi, E \vdash b[\mathtt{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b, E_b} : \Pi(E(b))$, that ensures that the local environment of the reductum can type its body, and $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$, 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 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 lemma on the algorithmic system.

Lemma C.3. *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.4 (Soundness and completeness). *Let P be a process, and E a type environment such that fn$(P) \subseteq \mathsf{Dom}(E)$. Then $\mathscr{R}_{env}(E, P), E \vdash_{\mathscr{A}} P : \mathscr{R}_{type}(E, P)$ (soundness). Furthermore, for any Π and P such that $\Pi, E \vdash_{\mathscr{A}} P : \mathsf{P}$, one has $\mathscr{R}_{env}(E, P) \subseteq \Pi$ and $\mathscr{R}_{type}(E, P) \subseteq \mathsf{P}$ (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 $\mathsf{cap} \in \{\mathtt{in}, \overline{\mathtt{in}}, \mathtt{out}, \overline{\mathtt{open}}\}$, 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 $\mathsf{cap} = \overline{\mathtt{out}}\ A$, by definition $\mathscr{R}_{\mathsf{env}}(E, P) = \Pi$ and $\mathscr{R}_{\mathsf{type}}(E, P) = \mathsf{P} \cup^{=} \{\mathtt{out}\ A\}$. Then the desired judgment derives from (ACTION$^{=}_{1}$).
If $\mathsf{cap} = \overline{\mathtt{open}}\ A$, 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 ($\mathbf{R}$-ACTION$^{=}_{2}$). The desired judgment derives from (ACTION$^{=}_{2}$).
In all three cases completeness follows from the induction hypothesis and monotonicity of the union.

$P = !P'$ *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.3

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

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

From (7) and (8) above, by (PAR), $\Pi, E \vdash_{\mathscr{A}} P_1 \mid P_2 : (\mathsf{P}_1^\star \cup \mathsf{P}_2^\star)$ It is now easy to check 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 $\mathsf{EnvClosure}$ and $\mathsf{ProcClosure}$ operators. Indeed for any Π' and P' such that $\Pi', E \vdash_{\mathscr{A}} P_1 \mid P_2 : \mathsf{P}'$, by induction hypothesis $\Pi_1 \subseteq \Pi'$ and $\Pi_2 \subseteq \Pi'$ hold, 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 $\mathsf{EnvClosure}$ 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 ($\mathbf{R}$-AMB) and set $\mathsf{P}^\star = \mathsf{ProcClosure}(\mathsf{P}, \Pi^\star)$. By construction of $\Pi^\star$ 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.3 we deduce

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

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

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

The result follows from (12), (13), and (14) by (AMB). For the 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 $\mathsf{EnvClosure}$, $\mathsf{ProcClosure}$, and $\mathsf{DomClosure}$. □

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.

Scope extrusion requires that extruded restrictions be 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 may then define a congruence rule for scope extrusion:

$$\sharp_E (\boldsymbol{\nu} a{:}D)P \quad \equiv \quad (\boldsymbol{\nu} a{:}D)\sharp_{E,a:D} P \tag{15}$$

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.

```
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 $M$ in $\Pi(D)^{=}$ do
          $\Pi' := \Pi$
          case $M$ of
            out $H$:
              if $\overline{\texttt{out}}\ H \in \Pi(H)^{\downarrow}$ then $\Pi(H) := \Pi(H) \cup \Pi(D)$
            in $H$:
              if $\overline{\texttt{in}}\ H \in \Pi(H)^{=}$ then
                  begin
                      $\Pi(H)^{=} := \Pi(H)^{=} \cup \Pi(D)^{\uparrow}$; $\Pi(H)^{\downarrow} := \Pi(H)^{\downarrow} \cup \Pi(A)^{=}$
                      if $\overline{\texttt{open}}\ H \in \Pi(H)^{=}$ then $\Pi(H) := \Pi(H) \cup \Pi(D)$
                  end
            open $H$:
              if $\overline{\texttt{open}}\ H \in \Pi(H)^{=}$ 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 7: A closure algorithm

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

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

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 reasons residuals will be tagged processes rather than 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:

(Type Tag)

$$\frac{\Pi, E \vdash P : \mathsf{P}}{\Pi, E \vdash \sharp_{E'} P : \mathsf{P}} \qquad \frac{P \downarrow \mathsf{cap}\ a^{\eta}}{\sharp_E P \downarrow \mathsf{cap}\ a^{\eta}}\ a \notin \mathsf{Dom}(E)$$

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 (15) a residual could exercise a capability that in the original occurrence would have been blocked by a restriction. 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 should 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 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 $\mathscr{D}$, then $\Pi, (E \cdot E_{\mathscr{D}}) \vdash \sharp_{E_2} P_2 : \mathsf{P}_1$.*

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 $\sharp P \mid \sharp Q \equiv \sharp(P \mid Q)$).
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 $E_{\mathscr{C}_1} = E_{\mathscr{C}'_1}$. □