A confinement criterion for securely executing mobile code

Hervé Grall
CERMICS (ENPC - INRIA)
6 et 8 av. Blaise Pascal, Cité Descartes, Champs/Marne
77455 Marne-La-Vallée Cedex 2, France
herve.grall@free.fr

Abstract

Mobile programs, like applets, are not only ubiquitous, but also potentially malicious. We study the case where mobile programs are executed by a host system in a secured environment, in order to control all accesses from mobile programs to local resources. The article deals with the following question: how to ensure that the local environment is secure? We answer by giving a *confinement criterion*: if the type of the local environment satisfies it, then no mobile program can directly access to a local resource. The criterion, which is type-based and hence decidable, is valid for a functional language with references. By proving its validity, we solve a conjecture stated by Leroy and Rouaix at POPL '98; moreover, we show that the criterion is optimal. The article mainly presents the proof method, based on a language annotation which keeps track of code origin and thus enables studying the interaction frontier between the local code and the mobile code, and it finally discusses the generalization of the method.

TYPED PROGRAMMING LANGUAGES, MOBILE CODE, LANGUAGE-BASED SECURITY, ACCESS CONTROLS, CONFINEMENT

Contents

1 Introduction **2**

2 Keeping track of code origin by annotating the language **5**

3 Frontier under control: The confinement criterion **9**

4 Conclusion **15**

Bibliography **17**

Appendices **20**

A Validity of the confinement criterion **20**

B Optimality of the confinement criterion **28**

C The SLam-calculus: From access controls to confinement **33**

1 Introduction

Mobile programs, like applets, are not only ubiquitous, but also potentially malicious. It is thus usual that a host system executes mobile programs in a secured local environment. The environment acts as an interface for local resources and thus enables controlling interactions between mobile programs and local resources, and particularly accesses from mobile programs to local resources. A typical example is provided by the language Java, which was designed to support the construction of applications that import and execute untrusted code from across a network, like Internet; a Java applet, which is a mobile program, is executed in a secured environment by a virtual machine, which can be embedded in a web browser, or in a smart card. Since the original security model, the "sandbox model", which provided a very restricted environment, the security architecture has evolved towards a greater flexibility, with fine-grained access controls [7, 8].

The article defines a *confinement criterion*, which is new and optimal; the criterion can help to enforce a security policy based on access controls in the presence of mobile code, as the following introductory example explains.

Let us consider an object which represents a resource like a file and is equipped with an access function, and suppose that accesses to the resource need to be controlled. There are two techniques to implementing these controls. Firstly, the local environment can provide a direct reference to the resource; in order to protect the resource, the encapsulated access function must be replaced with a secured one, which makes some security checks before accessing to the resource. Secondly, the local environment can provide as a service a function which makes some security checks before calling the access function encapsulated in the resource; of course, it must also prevent mobile programs from directly calling the unsecured access function: in other words, the resource must be *confined* in the local environment.

A confinement criterion ensures that if the local environment satisfies it, then no mobile program can directly call the access function of a resource. As the example shows, confinement is used in conjunction with access controls in the local environment. It is therefore interesting that before studying confinement, we review some current implementations of access controls.

Access controls in programming languages We restrict ourselves to the approach called *language-based security*, which is particularly relevant to computer-security questions, as Harper et al. have argued in [9]. In this approach, security controls are implemented either at a syntactic level, for instance, by instrumenting code, which leads to directly insert checks in programs, or at a semantic level, by statically analyzing code or by modifying the (operational) semantics. An access involves a subject, also called a principal, an object, which we prefer to call a resource, and an access function, so that the question is: how are they represented in this approach? An identification mechanism usually assigns each piece of code to a subject. A static point of view only considers the current subject, the caller of the access function, as in the SLam-calculus (acronym for "Secure Lambda-calculus") [10] or in the POP system (acronym for "Programming with Object Protection") [23]; a dynamic point of view leads to consider not only the current subject, but also its callers, which are obtained by inspecting the call stack: see the original work of Wallach [27], the official implementation for Java [7, 8], Schneider and Erlingsson's alternative [5], Pottier et al.'s analysis [20], which replaces some dynamic checks with static ones, implemented by a type system, and the semantic theory of stack inspection developed by Fournet and Gordon in [6]. A resource, which usually corresponds to the source of an input or the target of an output, is represented by a value of the programming language: any object in Java can therefore be considered as a resource, likewise any function in a functional language like ML. An access function is any operation which can be applied to a value: any method call in Java and any functional application in ML can therefore be considered as accesses. A security policy defines the rules that all accesses must obey during executions. Since Schneider's work [22], it has become usual to specify a security policy for controlling ac-

cesses with a security automaton: see Colcombet and Fradet's implementation [4], Walker's [26] and Thiemann's [24], which all mix program transformations with static analyses.

Mobile code issue We now come to our specific question: how can we enforce access controls in the presence of mobile code?
We assume that the code of the local environment can be fully controlled by using one of the preceding techniques, since it is available on the host system. As for mobile programs, two limit cases can be drawn.
The first one corresponds to limiting mobility effects by controlling not only the local environment but also mobile programs; since the mobile code is not available on the host system, *program certification* is needed. Before being sent, a mobile program is analyzed or transformed by using one of the preceding techniques, in order to certify it according to the security policy of the host system. Modularity is required to proceed as follows: firstly, the local environment is analyzed to determine its secure calling contexts, secondly, the code of the mobile program is certified by controlling that calls to the local environment are secure. A good example is [2], which adds modularity to the analysis proposed by Jensen et al. in [12]. When the host system receives a mobile program certified, it suffices to verify the correctness of the certification. This verification may just be an authentication process, but it may also be the proof that the execution of the mobile program in the local environment will satisfy the security policy. In [16], Lee and Necula propose to use "proof-carrying code", that is to say to add proof hints to the mobile program, in order to ease the proof task.
It remains that with program certification, the mobile program can no more be regarded as hostile; on the contrary, for the second limit case, no assumption is made about mobile programs, since only the local environment plays a defensive role. We can distinguish two defensive techniques, as in the introductory example. The first one is based on *encapsulation*: access functions are replaced with secured ones, and since they are encapsulated in local resources, each time a mobile program calls an access function, it actually calls a secured one. A good example is the current implementation of access controls in Java system classes. The second one is based on *confinement*: accesses are controlled in the local environment and resources are confined in the local environment, so that there will be no direct accesses from outside the local environment. This technique may be chosen when the secured access function cannot be encapsulated in the resource, as in the two following examples: firstly, when the access function itself is not encapsulated in the resource, as in Java, for the cast operation (`(Type)obj`) and the reference equality (`=`), which can be applied to references, but are not methods; secondly, when the code of the resource is not available. Performance reasons may also justify this solution. A practical example for this technique is [25], where Bokowski and Vitek advocate for Java this solution by confinement. They propose to confine types, that is to say to confine all the values of sensitive types; their proposal is formalized in [17] for a Java-like language. We also give a type-based confinement criterion, but for a simpler language, a functional language with references; our work actually extends Leroy and Rouaix's results in [13], where code instrumentation is used in conjunction with confinement or type abstraction to control accesses. All these confinement criteria turn out to assume that the programming language is typed and that its type system is sound: "well-typed programs do not go wrong". Otherwise, it would be impossible to give a confinement criterion, since a violation of type soundness by the mobile program could lead to an uncontrolled access (for example, by converting a string to a resource reference).

To conclude this introduction, we must mention a limitation of our work. We give the confinement criterion for a high-level language, in order to ensure the following property: for each local environment satisfying the confinement criterion, for every mobile program, the mobile code cannot directly access to a sensitive local resource during its execution in the local environment. The local environment and the mobile program are expressed in the high-level language; actually, their implementations use a low-level language, like the "bytecode" language for Java, for example. According to Abadi in [1], compilation is not

a fully abstract translation, which means that contextual properties (using the universal quantification "for every mobile program") are not preserved by compilation, since low-level languages are richer than high-level ones. This is a good reason for studying in a future work low-level languages instead of high-level ones; note that a trend is now emerging, which leads to using structured low-level languages, suitable for verification: see for example the "typed assembly language" [15] or the translation of Java Bytecode in a typed term calculus [11]. Otherwise, the preservation by compilation of contextual properties may be obtained by certifying mobile programs in order to execute them only if they result from a compilation.

Overview of the paper In Section 2, called "Keeping track of code origin by annotating the language", we define the language with which we work. It is a functional language which also enables manipulating objects in the memory heap: it corresponds to a simply typed λ-calculus with references in the style of ML. Although it is a very simple language, it is sufficient for modeling complex interactions between the mobile program and the local environment, resulting from the so-called side effects. It is also expressive enough, since for each functional type, a fixpoint combinator can be encoded (by using references). In order to rigorously define confinement, we need to keep track of code origin during executions. As described by Klop et al. in [3, sect. 4], a solution is to annotate the language: every operator of a program becomes labeled with its origin, in our case, either the mobile program, or the local environment, and preserves its label during the execution. Executions of labeled programs are described by an operational semantics, defined by a reduction relation; this relation includes the following β-reduction, which clearly preserves labels:

$$@^m(\lambda x^n\, b, a) \to b[a/x]\,,$$

where $@^m(-,-)$ stands for the application operator.
Section 3, called "Frontier under control: The confinement criterion", is an application to confinement of this annotation technique. We start by modeling the execution of a mobile program in the local environment. The local environment is simply a well-typed program, denoted by L, whereas the mobile program is modeled by a well-typed functional abstraction, $\lambda x\, \mathsf{M}[x]$, which is applied to the local environment. We then define what it means for a type to be confined in the local environment: a resource type A is confined in the local environment if for every mobile program, no operator coming from the mobile program is applied to a local resource of the type A during the execution. The general confinement criterion would answer the following question: given a local environment and a resource type A, determine whether the type A is confined in the local environment. Of course, a question like this is undecidable; in order to obtain decidability we resort to an approximation by considering the type of the local environment and not its value. The approximate confinement criterion that we will give exactly answers the following question: given a type L for the local environment and a resource type A, determine whether for every local environment L of type L, the type A is confined in L. The proof is based on an accurate analysis of what happens at the frontier between the mobile code and the local code during an execution. This result first solves a conjecture stated by Leroy and Rouaix in [13, sect. 5.1, p. 397]. They have proved that a type A is confined in a local environment if it does not occur in the type of the environment, and we prove what they have conjectured: it suffices that the type A occurs in the type of the environment neither at a positive occurrence, nor under the reference type constructor $\texttt{Ref}(-)$. We also prove that this criterion is optimal, that is to say as weak as possible: if the resource type A occurs in the environment type L at a dangerous place, then there exists a local environment L of type L such that the type A is not confined in L.
Finally, after a brief comparison with Vitek et al.'s results in [25, 17] and Leroy and Rouaix's in [13], we will outline some directions for future works. It is particularly interesting to consider some extensions of the programming language that we use and to better understand the relationship between the confinement criterion and standard security analyses.
Note that the article ends with three appendices. Appendices A and B give the proof details

of our main results about the confinement criterion. Appendix C describes a particular security analysis and its relationship with our confinement criterion.

2 Keeping track of code origin by annotating the language

We work with a functional language which also enables manipulating objects in the memory heap: it corresponds to the Church version of a simply typed λ-calculus enriched with references in the style of ML (see Table 1).

Table 1: Syntax of the programming language

$$
\begin{array}{lcll}
A & ::= & \texttt{Unit} & \text{(singleton type)} \\
 & | & A \to A & \text{(functional type)} \\
 & | & \texttt{Ref}(A) & \text{(reference type)} \\
 & & & \\
a & ::= & x & \text{(variable)} \\
 & | & \lambda x : A.\, a & \text{(abstraction)} \\
 & | & a\, a & \text{(application)} \\
 & | & \texttt{unit} & \text{(unit)} \\
 & | & l^{\texttt{Ref}(A)} & \text{(reference, with identifier } l \text{ and content of type } A) \\
 & | & \texttt{ref}(a) & \text{(reference creation)} \\
 & | & \texttt{get}(a) & \text{(dereferencing)} \\
 & | & \texttt{set}(a, a) & \text{(reference assignment)}
\end{array}
$$

In order to keep track of code origin during executions, we resort to an annotation of the language: operators become labeled. A label is an ordered pair, whose first component is a type, called the *type label*, and second component a piece of information, called the *information label*. The syntax of the annotated language is given in Table 2. In the following, $\mathtt{f}$ and $\mathtt{g}$ stand for operators in $\{\lambda x, @, \texttt{unit}, l, \texttt{ref}, \texttt{get}, \texttt{set}\}$: a labeled term e either is a variable, or has the form $\mathtt{f}^m(e_1, \ldots, e_i)$, where $e_1, \ldots, e_i$ are the immediate subterms $(0 \leq i \leq 2)$. If e is equal to $\mathtt{f}^m(\ldots)$ for some label m and some operator $\mathtt{f}$, then we sometimes write e^m for e in order to stress $\mathtt{f}$'s label, which is called the label of e. Given a labeled term e, the set of free variables in e is denoted by $\mathbf{FV}(e)$, and the set of labeled references in e is denoted by $\mathbf{Ref}(e)$. A *closed term* is a term without free variables, whereas a *program* is a closed term without references.
Note the two choices that we make: variables are not labeled and references are uniformly labeled. Indeed, we suppose that for any labeled term e, the family $(l)_{l^m \in \mathbf{Ref}(e)}$ is one-to-one, in other words, we suppose that the labeled references occurring in e can be identified by their identifier. In the following, although we only use terms satisfying this formation rule, we omit to verify its preservation (by reduction, etc.), which is always obvious.

The static and dynamic semantics of the annotated language closely follow the standard definitions for the programming language, which are not recalled because they can be deduced from the annotated version. The *type system*, which is given in Table 3, enables distinguishing between functions and references. Note the following points:

- for each valid judgment $\Gamma \vdash e : A$, where e is not a variable, the type A is the type label of e, therefore e receives a unique type in Γ;

Table 2: Syntax of the annotated language

$$
\begin{array}{rcll}
m & ::= & (A, \iota) & \text{(type and information)} \\
\\
e & ::= & x & \text{(variable)} \\
 & | & \lambda x^m \, e & \text{(labeled abstraction)} \\
 & | & @^m(e, e) & \text{(labeled application)} \\
 & | & \texttt{unit}^m & \text{(labeled unit)} \\
 & | & l^m & \text{(labeled reference, with identifier } l \text{ and label } m\text{)} \\
 & | & \texttt{ref}^{\,m}(e) & \text{(labeled reference creation)} \\
 & | & \texttt{get}^{\,m}(e) & \text{(labeled dereferencing)} \\
 & | & \texttt{set}^{\,m}(e, e) & \text{(labeled reference assignment)}
\end{array}
$$

- only type labels matter in typing rules;
- since references are labeled with their type, typing environments only deal with variables, which are not labeled.

Table 3: Type system

$$\frac{\emptyset}{\Gamma \vdash x \,:\, \Gamma(x)} \quad (x \in \mathrm{dom}\,\Gamma)$$

$$\frac{\Gamma.(x \,:\, A) \vdash e \,:\, B}{\Gamma \vdash \lambda x^{(A \to B, \iota)} \, e \,:\, A \to B} \qquad \frac{\Gamma \vdash e_1 \,:\, A \to B \quad \Gamma \vdash e_2 \,:\, A}{\Gamma \vdash @^{(B,\iota)}(e_1, e_2) \,:\, B}$$

$$\frac{\emptyset}{\Gamma \vdash \texttt{unit}^{(\texttt{Unit},\iota)} \,:\, \texttt{Unit}}$$

$$\frac{\emptyset}{\Gamma \vdash l^{(\texttt{Ref}(A),\iota)} \,:\, \texttt{Ref}(A)} \qquad \frac{\Gamma \vdash e \,:\, A}{\Gamma \vdash \texttt{ref}^{\,(\texttt{Ref}(A),\iota)}(e) \,:\, \texttt{Ref}(A)}$$

$$\frac{\Gamma \vdash e \,:\, \texttt{Ref}(A)}{\Gamma \vdash \texttt{get}^{\,(A,\iota)}(e) \,:\, A} \qquad \frac{\Gamma \vdash e_1 \,:\, \texttt{Ref}(A) \quad \Gamma \vdash e_2 \,:\, A}{\Gamma \vdash \texttt{set}^{\,(\texttt{Unit},\iota)}(e_1, e_2) \,:\, \texttt{Unit}}$$

The operational semantics is given by a reduction relation, and is presented by using the standard decomposition of every closed term either into a value, or into a redex in an evaluation context (cf. [28]). We use a *lazy call-by-value reduction strategy*: the leftmost call-by-value redex which does not appear within a λ-abstraction reduces.
Substitutions are defined as usual: if e and a are terms and x a variable, then $e[a/x]$ stands for the result of replacing in e the free occurrences of x with a, without variable capture.
A *value* is either a closed abstraction, the unit value, or a reference:

$$v \quad ::= \quad \lambda x^m \, e \mid \texttt{unit}^m \mid l^m \,,$$

where $\mathbf{FV}(\lambda x^m \, e) = \emptyset$. A *redex* is either a β-redex, a reference creation, a dereferencing or

an assignment:

$$r \quad ::= \quad @^n(\lambda x^m\, e, v) \mid \texttt{ref}^{\,m}(v) \mid \texttt{get}^{\,m}(l^n) \mid \texttt{set}^{\,m}(l^n, v)\,,$$

where $\mathbf{FV}(\lambda x^m\, e) = \emptyset$. An *evaluation context* is built around the hole $-$ as follows:

$$\begin{array}{rcl} E & ::= & - \\ & \mid & @^m(E, e) \mid @^m(\lambda x^n\, b, E) \\ & \mid & \texttt{ref}^{\,m}(E) \mid \texttt{get}^{\,m}(E) \mid \texttt{set}^{\,m}(E, e) \mid \texttt{set}^{\,m}(l^n, E)\,, \end{array}$$

where $\mathbf{FV}(\lambda x^n\, b) = \mathbf{FV}(e) = \emptyset$. Every well-typed closed term is equal to either a value, or a redex in an evaluation context, and this decomposition is unique.
A *store* s is a partial function from the the set of labeled references to the set of values such that:

(i) the family $(l)_{l^m \in \mathrm{dom}\, s}$ is one-to-one, in other words, the labeled references belonging to the domain of s can be identified by their identifier,

(ii) the set of identifiers $\{l \mid \exists\, m\,.\; l^m \in \mathrm{dom}\, s\}$ is an initial segment of the set of reference identifiers, which is supposed to be isomorphic to the set of natural numbers, and thus to be well-ordered.

The first condition is coherent with the formation rule for terms. The second condition enables defining a function for *creating* references: given a label m, the function ν_m maps each store s to some reference l^m such that l is the least identifier of the complement of $\{l \mid \exists\, m\,.\; l^m \in \mathrm{dom}\, s\}$. Another useful function enables *updating* or *extending* a store. Suppose that s is a store and l^m a labeled reference such that l^m is either already created, that is in $\mathrm{dom}\, s$, or is new, that is equal to $\nu_m(s)$; given a value v, a new store, denoted by $(s, l^m \mapsto v)$, is then defined as follows:

$$(s, l^m \mapsto v)(k^n) \stackrel{def}{=} \begin{cases} s(k^n) & \text{if } k^n \neq l^m\,, \\ v & \text{otherwise.} \end{cases}$$

A *configuration* is an ordered pair (s, e), where s is a memory store and e a closed term, called the *control term*, such that

(i) $\mathbf{Ref}(e) \subseteq \mathrm{dom}\, s\,,$

(ii) $\forall\, l^m \in \mathrm{dom}\, s\,.\; \mathbf{Ref}(s(l^m)) \subseteq \mathrm{dom}\, s\,.$

The two conditions mean that every labeled reference occurring in e or in a value belonging to the range of s has a well-defined content.
The *reduction relation* is defined by an inference system, in Table 4. A judgment has the form $(s, a) \to (s', a')$, which means that the configuration (s, a) reduces to (s', a').
The following propositions give the two main properties of the reduction relation. A store s is said to be *well-typed* if for any labeled reference l^m in the domain of s, l^m is well-typed, which is equivalent to $m = (\texttt{Ref}(A), \iota)$ for some type A and some information label ι, and the value $s(l^m)$ has type A; a configuration (s, e) is said to be *well-typed* if s and e are.

Proposition 1 (Subject reduction)
Let (s, e) be a well-typed configuration. If (s, e) reduces to (s', e'), then (s', e') is a well-typed configuration and e' has the same type as e.

Proposition 2 (Totality and determinism)
Let (s, e) be a well-typed configuration. If e is a value, then (s, e) does not reduce. Otherwise, (s, e) reduces to a unique configuration.

These propositions enable defining the execution trace of each well-typed program e: it is the maximal sequence of the configurations obtained by reduction from the initial configuration $(\emptyset, e)$, where $\emptyset$ is the store with an empty domain. A well-typed program either evaluates to a configuration, whose control term is a value, or diverges.

Table 4: Operational semantics — Reduction relation

$$\frac{\emptyset}{\big(s, @^m(\lambda x^n\, e, v)\big) \to \big(s, e[v/x]\big)}\ [\beta]$$

$$\frac{\emptyset}{\big(s, \texttt{ref}^{\,m}(v)\big) \to \big((s, l^m \mapsto v), l^m\big)}\ [\text{REF}] \quad (l^m = \nu_m(s))$$

$$\frac{\emptyset}{\big(s, \texttt{get}^{\,m}(l^n)\big) \to \big(s, s(l^n)\big)}\ [\text{REF}-!]$$

$$\frac{\emptyset}{\big(s, \texttt{set}^{\,m}(l^n, v)\big) \to \big((s, l^n \mapsto v), \texttt{unit}^m\big)}\ [\text{REF}-?]$$

$$\frac{\big(s, r\big) \to \big(s', r'\big)}{\big(s, E[r]\big) \to \big(s', E[r']\big)}\ [\text{RED}] \quad \left(\begin{array}{l} r \text{ redex} \\ E \text{ evaluation context } \neq - \end{array}\right)$$

Table 5: Strip function

$$\begin{array}{rcl}
\downarrow(x) & = & x \\
\downarrow(\lambda x^{(A\to B,\iota)}\, e) & = & \lambda x : A. \downarrow(e) \\
\downarrow(l^{(\texttt{Ref}(A),\iota)}) & = & l^{\texttt{Ref}(A)} \\
\downarrow\big(\texttt{f}^m(e_j)_j\big) & = & \texttt{f}\big(\downarrow(e_j)\big)_j
\end{array}$$

Of course, a strip function, denoted by $\downarrow$, which erases labels, can be defined: it suffices to proceed by structural induction as Table 5 shows.
The following propositions explain the relationship between the annotated language and the programming one.

Proposition 3 (Strip function - Typing preservation)
If the labeled term e has type A in the environment Γ, then $\downarrow(e)$ has also type A in Γ. Conversely, if the unlabeled term a has type A in Γ, then there exists a labeled term e of type A in Γ such that $\downarrow(e) = a$.

The second part of the proposition and the fact that information labels does not matter for typing enable defining from an information label ι and an unlabeled term a of type A a unicolorous term, denoted by $\langle a\rangle^\iota$, such that $\langle a\rangle^\iota$ has type A, $\downarrow(\langle a\rangle^\iota) = a$ and every operator of $\langle a\rangle^\iota$ has ι as its information label.

Proposition 4 (Strip function - Reduction preservation)
If T is the execution trace of a well-typed labeled program T_0, then $\downarrow(T)$ is the execution trace of $\downarrow(T_0)$. Conversely, if S is the execution trace of a well-typed program, then there exists an execution trace T of a well-typed labeled program such that $\downarrow(T)$ is equal to S.

In other words, the annotated language permits simulating the programming language while keeping track of code origin with labels.

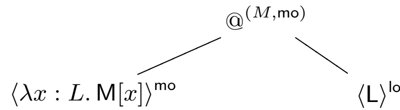
3 Frontier under control: The confinement criterion

In this section, since we want code to carry information about its origin, we use the pair $\{\mathsf{mo}, \mathsf{lo}\}$ as information labels: an operator labeled with mo comes from the mobile program, whereas one labeled with lo comes from the local environment. The labels mo and lo are called *origin labels*. In the following, if ι belongs to $\{\mathsf{mo}, \mathsf{lo}\}$, then $\bar{\iota}$ denotes the other origin label in order to obtain $\{\mathsf{mo}, \mathsf{lo}\} = \{\iota, \bar{\iota}\}$.
Initially, the local environment L of type L is labeled with lo, whereas the mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$, which is applied to the local environment, is labeled with mo. We thus study the execution of the well-typed labeled program

$$@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x]\rangle^{\mathsf{mo}}, \langle \mathsf{L}\rangle^{\mathsf{lo}}),$$

which is represented by the tree in Figure 1. Note that it is just a convention to label the application operator with mo.

Figure 1: Modeling the mobile program calling the local environment



In order to rigorously define confinement, we introduce *frontier redexes* in Table 6; the definition is parametrized by an origin label ι, belonging to $\{\mathsf{mo}, \mathsf{lo}\}$. From our perspective, an access is represented by a redex: the operator, @, `get` or `set`, which we call a destructor, is the access function, whereas the value destructed is the resource accessed. A frontier redex therefore corresponds to the application of an access function to a resource of a different origin. A type is confined in the local environment if no value of this type coming from the local environment can be destructed by mobile code. In other words, confinement actually means that some frontier redexes must not reduce.

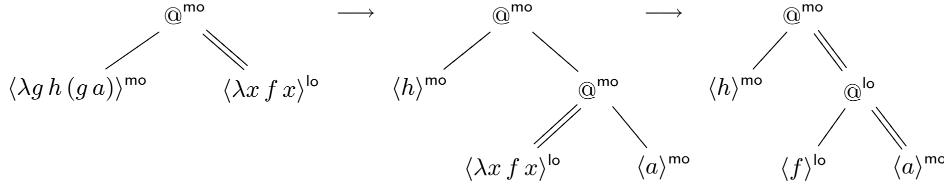
Table 6: Frontier redexes

Frontier Redex	Type of the value destructed
$@^{(B,\bar{\iota})}(\lambda x^{(A\to B,\iota)}\, e, v)$	$A \to B$
$\mathtt{get}^{\,(A,\bar{\iota})}(l^{(\mathtt{Ref}(A),\iota)})$	$\mathtt{Ref}(A)$
$\mathtt{set}^{\,(\mathtt{Unit},\bar{\iota})}(l^{(\mathtt{Ref}(A),\iota)}, v)$	$\mathtt{Ref}(A)$

Definition 5 (Confinement)
The type A is confined *in the local environment* L *of type L if for every mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$, no frontier redex destructs a value of the type A and of the origin label* lo *during the execution of the labeled program* $@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x]\rangle^{\mathsf{mo}}, \langle \mathsf{L}\rangle^{\mathsf{lo}})$.

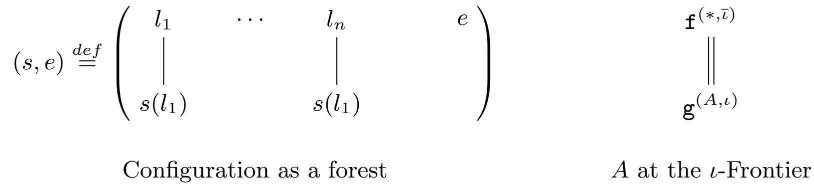
To ensure the confinement of a resource type, it suffices that it does not occur at the frontier between the mobile code and the local code in a configuration of the execution trace. Initially, there is only one type at the frontier, the type L of the local environment. Then, during the execution, the frontier becomes more complex, as is exemplified in Figure 2, where we omit to represent types and the empty store to simplify; in this example, the mobile program is equal to $\lambda g\, h\,(g\, a)$, where h is an abstraction, g a functional variable and a a value, and the local environment is equal to the η-expansion of some program f, $\lambda x\, f\, x$; the frontier is stressed with a double line.

Figure 2: Mobile code execution and frontier - Example



As the example shows, a tree representation is useful in order to make the frontier visible. Whereas we consider that a term is naturally equivalent to a tree, how can we represent a configuration? Suppose that (s, e) is a configuration, with $\operatorname{dom} s = \{l_1, \ldots, l_n\}$ (labels are omitted to simplify). Then the configuration is represented by the forest described in Figure 3: each reference is thus followed by its content (as a tree).

Figure 3: Configuration and frontier



Configuration as a forest $\qquad$ A at the ι-Frontier

The example also shows that we can actually distinguish two kinds of edges at the frontier, according to the origin label of the lower node, which leads to the following definition.

Definition 6 (Frontier)
Let ι be any origin label and let $\bar{\iota}$ be such that $\{\iota, \bar{\iota}\} = \{\mathsf{mo}, \mathsf{lo}\}$. Let us consider a configuration (s, e). The type A belongs to the ι-frontier *of (s, e) if there exists in the representation*

of (s, e) *as a forest an edge of the form described in Figure 3. The* lo*-frontier and the* mo*-frontier are respectively called the* outgoing frontier *and the* incoming frontier *(with respect to the local environment).*

Consider the type L for the local environment. We are going to determine from L two sets of types, $\mathcal{A}_{\mathsf{lo}}(L)$ and $\mathcal{A}_{\mathsf{mo}}(L)$, which are upper bounds of the outgoing and incoming frontiers resulting from executing mobile code: more precisely, given any local environment L of type L and any mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$, for every configuration (s, e) in the execution trace of

$$@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}}) \,,$$

the outgoing frontier of (s, e) will be included in $\mathcal{A}_{\mathsf{lo}}(L)$ and the incoming frontier of (s, e) will be included in $\mathcal{A}_{\mathsf{mo}}(L)$. Indeed, these upper bounds of the two frontiers lead to a confinement criterion: if a resource type A does not belong to $\mathcal{A}_{\mathsf{lo}}(L)$, then no frontier redex destructing a local value of type A reduces however the local environment and the mobile program are defined, in other words, the type A is confined in every local environment of type L.

Let us give some stability properties for these upper bounds. Firstly, note that L, initially at the lo-frontier, must belong to $\mathcal{A}_{\mathsf{lo}}(L)$. Secondly, it suffices to examine the worst cases, that is to say the cases which generate the most frontiers. Let ι be an origin label.

Suppose that $A \to B$ belongs to $\mathcal{A}_\iota(L)$. Consider a well-typed configuration

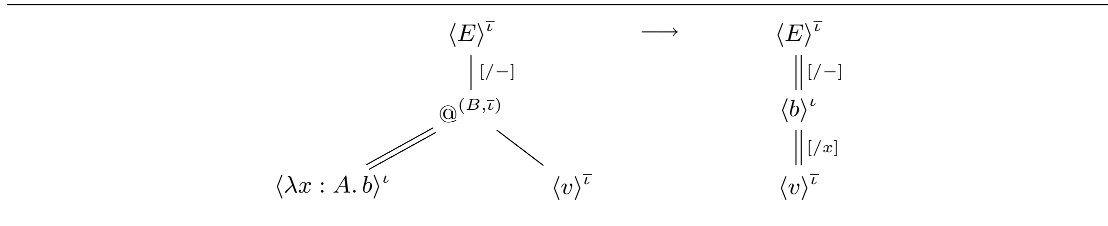
$$\big(\emptyset, \langle E \rangle^{\bar\iota}[@^{(B,\bar\iota)}(\langle \lambda x : A.\, b \rangle^{\iota}, \langle v \rangle^{\bar\iota})\big) \,,$$

where $\langle E \rangle^{\bar\iota}$ is a unicolorous evaluation context, $\langle \lambda x : A.\, b \rangle^{\iota}$ a unicolorous abstraction of type $A \to B$ and $\langle v \rangle^{\bar\iota}$ a unicolorous value of type A. The configuration contains $A \to B$ at the ι-frontier and reduces to

$$\big(\emptyset, \langle E \rangle^{\bar\iota}\big[\langle b \rangle^{\iota}[\langle v \rangle^{\bar\iota}/x]\big]\big) \,,$$

which contains A at the $\bar\iota$-frontier if b owns a free occurrence of x without being equal to x, and contains B at the ι-frontier if E is different from the hole $-$ and b from x: see Figure 4, where the empty store is omitted and where for any variable x, an edge labeled with $[/x]$ represents the substitution of the lower term for x in the upper term. The inference rules $[- \to\,]$ and $[\,\to +]$ in Table 7 (p. 12) are therefore valid.

Figure 4: Frontier and β-reduction



Suppose that $\texttt{Ref}(A)$ belongs to $\mathcal{A}_\iota(L)$. Let $l^{(\texttt{Ref}(A),\iota)}$ be a reference of type $\texttt{Ref}(A)$. Consider a well-typed configuration

$$\big(s, \langle E \rangle^{\bar\iota}[\texttt{get}^{\,(A,\bar\iota)}(l^{(\texttt{Ref}(A),\iota)})]\big) \,,$$

where $\langle E \rangle^{\bar\iota}$ is a unicolorous evaluation context and s is a memory store such that $s(l^{(\texttt{Ref}(A),\iota)})$ is equal to a unicolorous value $\langle v \rangle^{\iota}$ of type A. The configuration contains $\texttt{Ref}(A)$ at the ι-frontier and reduces to

$$\big(s, \langle E \rangle^{\bar\iota}[\langle v \rangle^{\iota}]\big) \,,$$

which contains A at the ι-frontier if E is different from the hole $-$: see Figure 5.

Figure 5: Frontier and dereferencing

$$\left(\ \cdots\ l^{(\texttt{Ref}(A),\iota)}\ \cdots\quad \langle E\rangle^{\bar\iota}\ \right) \longrightarrow \left(\ \cdots\ l^{(\texttt{Ref}(A),\iota)}\ \cdots\quad \langle E\rangle^{\bar\iota}\ \right)$$

Left configuration: $l^{(\texttt{Ref}(A),\iota)}$ — $\langle v\rangle^{\iota}$; $\langle E\rangle^{\bar\iota}$ —[/−]— $\texttt{get}^{(A,\bar\iota)}$ = $l^{(\texttt{Ref}(A),\iota)}$

Right configuration: $l^{(\texttt{Ref}(A),\iota)}$ — $\langle v\rangle^{\iota}$; $\langle E\rangle^{\bar\iota}$ =[/−]= $\langle v\rangle^{\iota}$

Consider a well-typed configuration

$$\big(s, \langle E\rangle^{\bar\iota}[\texttt{set}^{(\texttt{Unit},\bar\iota)}(l^{(\texttt{Ref}(A),\iota)}, \langle v\rangle^{\bar\iota})]\big)\ ,$$

where s is a memory store, $\langle E\rangle^{\bar\iota}$ a unicolorous evaluation context and $\langle v\rangle^{\bar\iota}$ a unicolorous value of type A. The configuration contains $\texttt{Ref}(A)$ at the ι-frontier and reduces to

$$\big((s, l^{(\texttt{Ref}(A),\iota)} \mapsto \langle v\rangle^{\bar\iota}), \langle E\rangle^{\bar\iota}[\texttt{unit}^{(\texttt{Unit},\bar\iota)}]\big)\ ,$$

which contains A at the $\bar\iota$-frontier ($\langle v\rangle^{\bar\iota}$ being under $l^{(\texttt{Ref}(A),\iota)}$ in the new store): see Figure 6.

Figure 6: Frontier and assignment

$$\left(\ \cdots\ l^{(\texttt{Ref}(A),\iota)}\ \cdots\quad \langle E\rangle^{\bar\iota}\ \right) \longrightarrow \left(\ \cdots\ l^{(\texttt{Ref}(A),\iota)}\ \cdots\quad \langle E\rangle^{\bar\iota}\ \right)$$

Left configuration: $l^{(\texttt{Ref}(A),\iota)}$ — $\cdots$; $\langle E\rangle^{\bar\iota}$ —[/−]— $\texttt{set}^{(\texttt{Unit},\bar\iota)}$, with children $l^{(\texttt{Ref}(A),\iota)}$ (double edge) and $\langle v\rangle^{\bar\iota}$

Right configuration: $l^{(\texttt{Ref}(A),\iota)}$ = $\langle v\rangle^{\bar\iota}$; $\langle E\rangle^{\bar\iota}$ —[/−]— $\texttt{unit}^{(\texttt{Unit},\bar\iota)}$

The inference rules $[\texttt{Ref}(+)]$ and $[\texttt{Ref}(-)]$ in Table 7 are therefore valid.

Table 7: Stability rules for frontier upper bounds

$$\frac{A \to B \in \mathcal{A}_\iota(L)}{A \in \mathcal{A}_{\bar\iota}(L)}\,[-\to] \qquad \frac{A \to B \in \mathcal{A}_\iota(L)}{B \in \mathcal{A}_\iota(L)}\,[\to +]$$

$$\frac{\texttt{Ref}(A) \in \mathcal{A}_\iota(L)}{A \in \mathcal{A}_{\bar\iota}(L)}\,[\texttt{Ref}(-)] \qquad \frac{\texttt{Ref}(A) \in \mathcal{A}_\iota(L)}{A \in \mathcal{A}_\iota(L)}\,[\texttt{Ref}(+)]$$

Natural candidates for the two upper bounds $\mathcal{A}_{\mathsf{lo}}(L)$ and $\mathcal{A}_{\mathsf{mo}}(L)$ are the sets inductively generated by the inference rules in Table 7 from the axiom $L \in \mathcal{A}_{\mathsf{lo}}(L)$, which easily gives:

(i) A belongs to $\mathcal{A}_{\mathsf{lo}}(L)$ if and only if A occurs in L at a positive occurrence or under the type constructor $\texttt{Ref}(-)$,

(ii) A belongs to $\mathcal{A}_{\mathsf{mo}}(L)$ if and only if A occurs in L at a negative occurrence or under the type constructor $\texttt{Ref}(-)$.

For example, the type A occurs at a positive occurrence in $* \to A$ or in $(A \to *) \to *$, at a negative occurrence in $A \to *$ or in $(* \to A) \to *$, and under the type constructor $\texttt{Ref}(-)$ in $\texttt{Ref}(* \to A) \to *$ (each occurrence of $*$ standing for any type). The following definition gives a name to these natural candidates.

Definition 7 (Outgoing and incoming types)
Let L be a type. We say that

(i) a type A is an outgoing type *of L if A occurs in L at a positive occurrence or under the type constructor* `Ref`$(-)$*,*

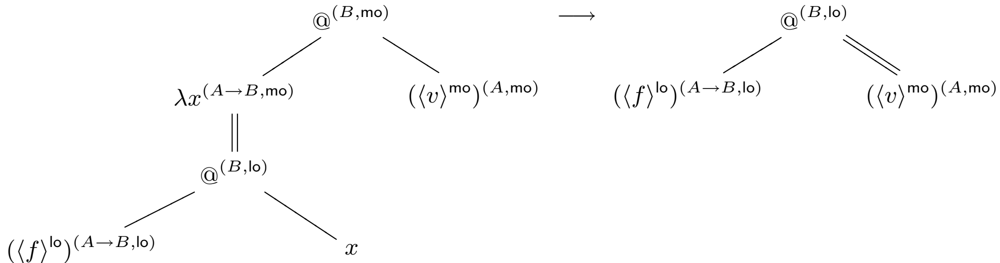
(ii) a type A is an incoming type *of L if A occurs in L at a negative occurrence or under the type constructor* `Ref`$(-)$*.*

The intuition behind these definitions is right: a type which is not an outgoing type of L is confined in L, as the following theorem precisely shows, thus solving the conjecture stated by Leroy and Rouaix in [13, sect. 5.1, p. 397].

Theorem 8 (Confinement criterion)
Let L be the type for the local environment, and A a resource type. If the type A is not an outgoing type of L, then for every local environment L of type L, the type A is confined in L.

Proof. Suppose that the resource type A is not an outgoing type of L. Let L be any local environment of type L, and $\lambda x : L.\, \mathsf{M}[x]$ be any mobile program of type $L \to M$. We must show that no frontier redex destructs a value of the type A and of the origin label lo during the execution of the labeled program $@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x]\rangle^{\mathsf{mo}}, \langle \mathsf{L}\rangle^{\mathsf{lo}})$. Let $(T_i)_i$ be its execution trace. It suffices to show that for every configuration T_i in the trace, the outgoing frontier does not contain A. Since by hypothesis, A is not an outgoing type of L, it thus suffices to show that a type belonging to the outgoing frontier is an outgoing type of L and one belonging to the incoming frontier is an incoming type of L. We proceed by induction on the position i in the trace.
Initially, the incoming frontier is empty whereas the outgoing frontier contains a unique type, L, which is an outgoing type of itself.
Then we can conclude from the fact that the two frontier upper bounds are preserved by reduction, which is shown in Appendix A (Th. 23): for any well-typed configuration t reducing to t', if the outgoing frontier and the incoming frontier of t are respectively included in the set of outgoing types and in the set of incoming types, then so are the frontiers of t'.
Note that a further hypothesis about origins is needed to obtain preservation: in any configuration, for each occurrence of any bound variable, all operators between the occurrence and its binder (including the binder) must have the same origin. This property of origin coherence is initially satisfied and is also preserved by reduction (cf. Appendix A (Prop. 13)).
The example described in Figure 7 shows why this hypothesis is required: indeed, initially, the incoming frontier, empty, and the outgoing frontier, equal to $\{B\}$, are respectively included in the set of incoming types of B and in the set of outgoing types of B; finally, the outgoing frontier is empty and the incoming frontier is equal to $\{A\}$, which can be any type and therefore may be not an incoming type of B: there is not preservation. □

Figure 7: Origin incoherence - Example



The confinement criterion that we have given is optimal since the converse of Theorem 8 is valid: an outgoing type is not confined.

Theorem 9 (Criterion optimality)
Let L be the type for the local environment, and A a resource type. If the type A is an outgoing type of L, then there exists a local environment L of type L such that the type A is not confined in L.

The proof, which can be found in Appendix B, is by induction on the type L: for each type L of whom the resource type A is an outgoing type, a local environment L of type L and a mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$ can be defined in order that a frontier redex destructs a value of the type A and of the origin label lo during the execution of the labeled program $@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})$.
The following cases sketch the proof by showing how the mobile program and the local environment can cooperate. We also illustrate these cases with concrete examples, using OCAML, an implementation of ML. These examples are presented as transcripts of sessions with the interactive system: an entry starting with `#` and finishing with `;;` represents a user input, an OCAML phrase; the system response is printed below, without a leading `#`. Note that we actually modify the type outputs of the interpreter by removing polymorphism in order to ease reading.
Program 1 gives some basic definitions. The resource type is defined by the class `resource`;

Program 1: Class for resources, instantiation for a local resource and mobile code access

```
# class resource origin = (* class resource parametrized by origin *)
    object
      method access subject = print_string (subject^" onto "^origin)
  end;;
class resource : string -> object method access : string -> unit end
# let res = new resource "local";; (* instantiation *)
val res : resource = <obj>
# let attack r = r#access "mobile";; (* mobile code access *)
val attack : resource -> unit = <fun>
```

the class is parametrized by `origin`, which we use to indicate the origin of created instances; the variable `res`, which is an instance of the class `resource` created with `"local"` as the origin, represents the sensitive local resource. The goal of our mobile programs is to call the method `access` of `res` with `"mobile"` as the argument, which will write `"mobile onto local"` onto the standard output: the function `attack` actually realizes this access.
We now describe some interesting cases. We suppose that a local resource of type A is given; for each given type L for the local environment, we informally define a local environment of type L and a well-typed mobile program whose execution entails the forbidden access to the local resource. We also give the concrete examples in OCAML, where A is represented by `resource` and the other types by `unit`.
If $L = (A \to B) \to C$, then A occurs at a positive occurrence in L. The mobile program passes to the environment a function which realizes an access to its unique argument; the environment applies this function to the resource: see Program 2.
If $L = \mathtt{Ref}(A) \to B$, then A occurs under the type constructor $\mathtt{Ref}(-)$. The mobile program passes a reference of type $\mathtt{Ref}(A)$ to the environment; the environment assigns to this reference a new content, the local resource, which becomes available to the mobile program: see Program 3.
If $L = \mathtt{Ref}(A \to B)$, then A again occurs under the type constructor $\mathtt{Ref}(-)$. It turns out that the situation is more difficult. The local environment is a reference l^L of type L, which is initialized with a particular function f: when f is applied to an argument, it first reads the content of l^L to obtain a function of type $A \to B$, then applies this function to the local

Program 2: Attack onto `resource` in `(resource -> unit) -> unit`

```
# let mobile e = e attack;; (* mobile program *)
val mobile : ((resource -> unit) -> unit) -> unit = <fun>
# let env f = f res;; (* local environment *)
val env : (resource -> unit) -> unit = <fun>
# mobile env;; (* mobile program calling the local environment *)
mobile onto local- : unit = ()
```

Program 3: Attack onto `resource` in `resource ref -> unit`

```
# let mobile e =
    let l = ref (new resource "mobile") in
    ( e l ; attack !l );; (* mobile program *)
val mobile : (resource ref -> unit) -> unit = <fun>
# let env l = (l := res );; (* local environment *)
val env : resource ref -> unit = <fun>
# mobile env;; (* mobile program calling the local environment *)
mobile onto local- : unit = ()
```

resource of type A, and finally returns some value of type B. As long as the content of l is f, the function f diverges for any argument. As for the mobile program, it first reads the content of its argument l^L in order to obtain f, then assigns to l^L a function of type $A \to B$ which realizes an access to its argument of type A; finally, the mobile program applies f to an argument of type A: the content of l^L, yet the function realizing the access, is thus applied to the local resource. See Program 4 for details.

Program 4: Attack onto `resource` in `(resource -> unit) ref`

```
# let mobile e = let f = !e in
                 let r = new resource "mobile" in
                 (e := attack ; f r);; (* mobile program *)
val mobile : (resource -> unit) ref -> unit = <fun>
# let f_init r = ();; (* initialization function *)
val f_init : resource -> unit = <fun>
# let env = let l = ref f_init in
            let f x = !l res in
            (l := f ; l);; (* local environment *)
val env : (resource -> unit) ref = {contents = <fun>}
# mobile env;;
mobile onto local- : unit = ()
```

4 Conclusion

As mobile programs, like applets, are potentially malicious, we have studied the case where mobile programs are executed by a host system in a secured local environment, in order to control all accesses from mobile programs to local resources. This article actually deals with the following question: how to ensure that the local environment is secure? We answer by giving a *confinement criterion*, such that if the local environment satisfies it, then no mobile program can directly access to a local resource.
The criterion is type-based: it takes as inputs a resource type A and a type L for the local environment, and determines whether for each local environment $\mathtt{L}$ of type L, the type A

is confined in $\mathtt{L}$, which means that no resource of type A belonging to the local environment $\mathtt{L}$ can be directly accessed by a well-typed mobile program. More precisely, we have proved that the resource type A is confined if it occurs in the environment type L neither at a positive occurrence, nor under the reference type constructor $\mathtt{Ref}(-)$, which can be computed with a polynomial-time complexity. This criterion improves the one given by Leroy and Rouaix in [13], which forbids the resource type to occur in the environment type; moreover, we have proved that our criterion is optimal: if the resource type A occurs in the environment type L at a positive occurrence or under the reference type constructor $\mathtt{Ref}(-)$, then there exists a local environment $\mathtt{L}$ of type L such that A is not confined in $\mathtt{L}$. It remains that the criterion is only valid for a functional language with references. Vitek et al.'s works about confinement [25, 17] deal with an object-oriented language like Java. If we try to apply their confinement criterion to mobile code, then we find that the criterion also forbids any resource type to occur in the environment type (cf. rules $\mathcal{C}_2$ and $\mathcal{C}_3$ in [17, p. 137]). It is thus interesting to determine whether our method, which is different from the methods of these previous works, can be extended to a richer language, particularly to an object-oriented language, in order to get the best possible confinement criterion. Our method is based firstly on the annotation of the programming language, in order to keep track of code origin, secondly on the study of the interaction frontier between the mobile code and the local code; it enables rigorously defining confinement, which seems to be its decisive advantage over the previous methods. Future works will confront our method to different features of object-oriented programming languages, like data abstraction, subtyping and inheritance. We actually expect that our method is suitable for a lot of data types, particularly with ordered pair, record and list types, and recursive types; since objects can be represented as references containing records, object types being recursive because of self-reference, objects could actually be added to our programming language without difficulty. As for other features, further investigations are needed and could benefit from Vitek et al.'s works [25, 17]. For example, a specific rule is needed for subtyping, since it becomes possible to convert a local resource into any supertype: not only the resource types must be confined, but also their supertypes, when they are used to convert local resources (cf. rule $\mathcal{C}_5$ in [17, p. 137]).

Finally, we come back to security concerns. As we have seen in the introduction, especially in the introductory example, we use confinement in conjunction with local access controls. The security certification of the local environment is therefore based on two elements, firstly, a security analysis of the code of the local environment, secondly, a confinement criterion.

A security analysis may directly lead to a confinement criterion. Indeed, it may enable inferring from a local result, only dealing with the local environment, a global result, dealing with any mobile program calling the local environment: an inference like this is valid when the local result satisfies some conditions, which define the confinement criterion. Whether this associated confinement criterion is equivalent to ours is an interesting question for a type-based analysis.

Let us give an illustration, developed in Appendix C. Suppose that the local environment contains sensitive resources. Since the local code is available, we can secure access functions in the local environment by adding run-time checks. This code instrumentation must ensure that no unsecure access function is applied to a sensitive resource during an execution. To verify this property, we can analyze the local environment; it remains that this analysis provides half of the verification: confinement for sensitive resources is also needed, since mobile code could directly access to a sensitive resource. We can therefore decompose the problem into two parts: firstly, conceive a security analysis which leads to control accesses in a whole program; secondly, determine a confinement criterion which enables inferring from the analysis result for the local environment the analysis result for any mobile program calling the local environment.

Heintze and Riecke answer in [10] the former problem by defining a type system which

statically ensures access controls for a language called the SLam-calculus (acronym for "Secure Lambda-calculus"). Note that we just consider access controls whereas their solution is more general since it also deals with information flows. An annotated language which keeps track of origin is used; labels mark sensitive resources and secured access functions. More precisely, two labels are used, $\bot$ and $\top$, with the following meaning:

	constructor	destructor
$\bot$	not sensitive	not secured
$\top$	sensitive	secured

Recall that resources are terms generated by constructors, and access functions are destructors. A labeled program is secure if during its execution, no destructor labeled with $\bot$ is applied to a constructor labeled with $\top$. Heintze and Riecke propose a system of labeled types to satisfy this property: a labeled program well-typed in this system is secure, as their subject reduction theorems show (cf. [10, Th. 2.1, 3.1]). In this type system, a value labeled with $\top$ (representing therefore a sensitive resource) has a type labeled with $\top$, whereas a value labeled with $\bot$ has a type labeled with $\bot$; elimination rules enforce the preceding security property, which gives for the application operator (with σ_1 and σ_2 belonging to $\{\bot, \top\}$, ordered by $\bot < \top$):

$$\frac{\Gamma \vdash f : A \xrightarrow{\sigma_1} B \quad \Gamma \vdash a : A}{\Gamma \vdash @^{\sigma_2}(f, a) : B} \quad (\sigma_1 \leq \sigma_2)\,.$$

The type system allows subtyping: given any type constructor $\mathtt{F}$, a type $\mathtt{F}^{\bot}(\ldots)$ is a subtype of $\mathtt{F}^{\top}(\ldots)$. Indeed, the substitution principle, as defined by Liskov and Wing in [14], is valid under the following form: if a program is secure when it uses any sensitive resource of type $\mathtt{F}^{\top}(\ldots)$, then it is also secure when it uses instead any value of type $\mathtt{F}^{\bot}(\ldots)$.
The subtyping relation enables solving the confinement problem, by using the technique of converting to confine. Consider the local environment: since it is available on the host system, we can assume that it is well-typed in the annotated type system, and therefore secure. As for a mobile program, since it is not available on the host system, it is just assumed to be typed in the standard type system, without annotation; actually, since the mobile program contains no sensitive resource and is not secured, it also receives a type only labeled with $\bot$ in the annotated type system. Here is the question: is the mobile program calling the local environment secure? The answer is affirmative if the local environment can receive a type completely labeled with $\bot$: the mobile program calling the local environment then receives a labeled type and is therefore secure. If the labeled type of the local environment contains $\top$, it may be subsumed in another type completely labeled with $\bot$: for instance, the type $\mathtt{Ref}^{\top}(A)$ is not a subtype of $\mathtt{Ref}^{\bot}(A)$, so that the label $\top$ cannot be removed, whereas the type $\mathtt{Ref}^{\top}(A) \xrightarrow{\bot} B$ is a subtype of $\mathtt{Ref}^{\bot}(A) \xrightarrow{\bot} B$ because of the rule of domain contravariance. Finally, this technique of converting the local environment leads to the following confinement criterion (cf. Appendix C (Prop. 30)):

> given a labeled environment type L, if every labeled type which occurs in L at a positive occurrence or under the reference type constructor $\mathtt{Ref}(-)$ is labeled with $\bot$, then every mobile program calling a local environment of the labeled type L is secure.

It turns out that the confinement criterion inferred from the type system and its subtyping properties is equivalent to ours. It remains to review other type-based security analyses.

Acknowledgements I thank Gilbert Caplain for helpful comments, the advisors of my doctoral thesis, Norbert Cot and René Lalement, and the members of my thesis jury, Didier Caucal, Thomas Jensen and Xavier Leroy, for encouraging me to publish my results.

References

[1] Martin Abadi. Protection in programming-language translations. *Lecture Notes in Computer Science*, 1443:868–883, 1998.

[2] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Proceedings of the 4th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '02)*, pages 76–87. ACM Press, 2002.

[3] Inge Bethke, Jan Willem Klop, and Roel de Vrijer. Descendants and origins in term rewriting. *Information and Computation*, 159:59–124, 2000.

[4] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In POPL '00 [18], pages 54–66.

[5] Ulaf Erlingsson and Fred Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy (S& P '00)*, pages 246–255. IEEE Computer Society Press, 2000.

[6] Cédric Fournet and Andrew Gordon. Stack inspection: Theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '02)*, volume 37(1) of *ACM SIGPLAN Notices*, pages 307–318. ACM Press, 2002.

[7] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 103–112. USENIX, 1997.

[8] Li Gong and Roland Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the 1998 Network and Distributed System Security Symposiums (NDSS '98)*, pages 125–134. Internet Society, 1998.

[9] Robert Harper, Greg Morrisett, and Fred Schneider. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, 2001.

[10] Nevin Heintze and Jon Riecke. The SLam calculus: Programming with security and integrity. In POPL '98 [19], pages 365–377.

[11] Tomoyuki Higuchi and Atsushi Ohori. Java bytecode as a typed term calculus. In *Proceedings of the 4th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '02)*, pages 201–211. ACM Press, 2002.

[12] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of control flow based security properties. In *Proceedings of the IEEE Symposium on Security and Privacy (S& P '99)*, pages 89–105. IEEE Computer Society Press, 1999.

[13] Xavier Leroy and François Rouaix. Security properties of typed applets. In POPL '98 [19], pages 391–403.

[14] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[15] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52. Springer-Verlag, 1998.

[16] George Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[17] Jens Palsberg, Jan Vitek, and Tian Zhao. Lightweight confinement for featherweight java. In *Proceedings of the 2003 ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, volume 38(11) of *ACM SIGPLAN Notices*, pages 135–148. ACM Press, 2003.

[18] *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '00)*. ACM Press, 2000.

[19] *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '98)*. ACM Press, 1998.

[20] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In Sands [21], pages 30–45.

[21] David Sands, editor. *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[22] Fred Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[23] Christian Skalka and Scott Smith. Static use-based object confinement. In Iliano Cervesato, editor, *Proceedings - Foundations of Computer Security (FCS '02)*, volume 02-12 of *DIKU technical reports*, pages 117–126, 2002.

[24] Peter Thiemann. Enforcing security properties by type specialization. In Sands [21], pages 62–76.

[25] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 82–96. ACM Press, 1999.

[26] David Walker. A type system for expressive security policies. In POPL '00 [18], pages 254–267.

[27] Dan Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, Department of Computer Science, 1999.

[28] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendices

Appendices A and B give the proof details of our main results about the confinement criterion (cf. Theorems 8 (p. 13) and 9 (p. 14)). Appendix C describes a particular security analysis, which has been developped by Heintze and Riecke in [10], and its relationship with our confinement criterion. It develops the second part of the conclusion (cf. p. 16–17).

A Validity of the confinement criterion

We finish off the proof of Theorem 8 (p. 13). We begin by formally defining origin coherence and studying the preservation of this property by reduction. Then, we prove that the frontier upper bounds are preserved by reduction, which allows concluding.

Definition 10 (Origin coherence)
A labeled term e is origin coherent *if it is inductively generated by the following rules:*

(i) e is a variable,

(ii) e is equal to $\mathtt{f}^{(A,\iota)}(e_1,\ldots,e_j)$, where for all integers k in $\{1,\ldots,j\}$,

(a) e_k is origin coherent,

(b) if e_k contains a free variable and is not a variable, then e_k is not at the frontier in e:

$$(\mathbf{FV}(e_k) \neq \emptyset \wedge e_k = \mathtt{g}^{(A_k,\iota_k)}(\ldots)) \Rightarrow \iota = \iota_k\,.$$

A configuration (s,e) is origin coherent *if*

(i) the control term e is origin coherent,

(ii) for every labeled reference l^m belonging to the domain of s, the value $s(l^m)$ is origin coherent.

This definition corresponds to the following intuitive characterization: in a term which is origin coherent, all operators between its root and an occurrence of any free variable have the same origin, likewise all operators between an occurrence of a bound variable and its binder (including the binder) have the same origin. For instance, given a closed term f, the term

$$\lambda x^{(A\to B,\mathsf{mo})}\ @^{(B,\mathsf{lo})}(\langle f\rangle^{\mathsf{lo}},x)$$

is not origin coherent, whereas the terms

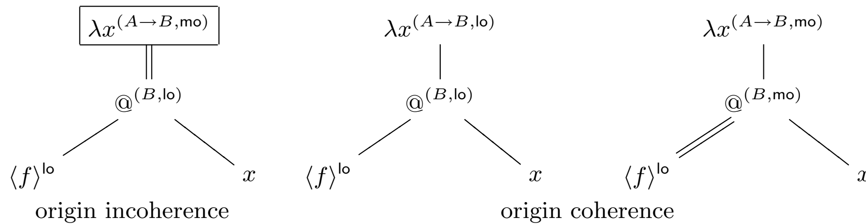
$$\lambda x^{(A\to B,\mathsf{lo})}\ @^{(B,\mathsf{lo})}(\langle f\rangle^{\mathsf{lo}},x)$$

and

$$\lambda x^{(A\to B,\mathsf{mo})}\ @^{(B,\mathsf{mo})}(\langle f\rangle^{\mathsf{lo}},x)$$

are (see Figure 8, where the operator with the wrong origin label is boxed). In the following,

Figure 8: Origin incoherence and coherence - Examples



we only use terms which are origin coherent. We must therefore show that this property is preserved by reduction.
We first examine the decomposition in an evaluation context. Note that in order to determine whether an evaluation context is origin coherent, we consider that the hole $-$ is not a variable, but an operator with a null arity. It is easy to show that evaluation contexts which are origin coherent are inductively generated by the following grammar:

$$\begin{array}{lll} E & ::= & - \\ & | & @^n(E,e) \mid @^n(v,E) \\ & | & \texttt{ref}^{\,m}(E) \mid \texttt{get}^{\,m}(E) \mid \texttt{set}^{\,m}(E,e) \mid \texttt{set}^{\,m}(v,E)\,, \end{array}$$

where the value v and the closed term e are origin coherent.
Let us give the first lemma, which will be used for the reduction rule [RED].

Lemma 11 $\left(\begin{array}{l}\textbf{Origin coherence}\\ \textbf{Decomposition lemma}\end{array}\right)$
Let E be a labeled evaluation context and e a labeled closed term. If $E[e]$ is origin coherent, then so are E and e.

Proof. It is straightforward, by induction on E. □

Let us consider substitutions, which are used in β-reductions; if the free variable is replaced with a closed term, origin coherence is preserved. Otherwise, the preservation may fail, as the following example shows:

$$\texttt{ref}^{\,\mathsf{lo}}(x)[\texttt{ref}^{\,\mathsf{mo}}(y)/x] = \texttt{ref}^{\,\mathsf{lo}}(\texttt{ref}^{\,\mathsf{mo}}(y))\,.$$

Lemma 12 $\left(\begin{array}{l}\textbf{Origin coherence}\\ \textbf{Substitution lemma}\end{array}\right)$
Let e and a be two labeled terms. If a is closed and if e and a are origin coherent, then $e[a/x]$ is origin coherent.

Proof. Let a be a labeled closed term which is origin coherent. We show by induction on e that if e is origin coherent, then so is $e[a/x]$. We deal with the only interesting case, where e is equal to $\mathtt{f}^{(A,\iota)}(e_1,\ldots,e_j)$. Let us suppose that e is origin coherent.
We show that for all integers k between 1 and j,

(a) $e_k[a/x]$ is origin coherent,

(b) if $e_k[a/x]$ contains a free variable and is not a variable, then $e_k[a/x]$ is not at the frontier in $e[a/x]$.

Let $k \in \{1,\ldots,j\}$.
The first condition follows from the inductive hypothesis, since e_k is origin coherent.
As for the second one, let us examine the different cases for e_k.
If $e_k = x$, then $e_k[a/x] = a$ and by hypothesis, $\mathbf{FV}(a) = \emptyset$: the second condition is trivially satisfied.
If e_k is equal to a variable y different from x, then $e_k[a/x] = y$: the second condition is trivially satisfied.
Otherwise, e_k is equal to $\mathtt{g}^{(A_k,\iota_k)}(\ldots)$ (for some operator $\mathtt{g}$), so that

$$e[a/x] = \mathtt{f}^{(A,\iota)}(e_1[a/x],\ldots,\mathtt{g}^{(A_k,\iota_k)}(\ldots),\ldots,e_j[a/x])\,.$$

If $e_k[a/x]$ contains a free variable, then since $\mathbf{FV}(e_k[a/x]) \subseteq \mathbf{FV}(e_k)$, a being closed, e_k contains a free variable; since e is origin coherent, we have $\iota = \iota_k$ and we can conclude. □

We can now prove that origin coherence is preserved by reduction.

Proposition 13 $\left(\begin{array}{l}\textbf{Origin coherence}\\ \textbf{Preservation by reduction}\end{array}\right)$
Let t_1 be a configuration reducing to t_2. If t_1 is origin coherent, then so is t_2.

Proof. We proceed by induction on the proof of $t_1 \rightarrow t_2$.

• $[\beta]$

We use Lemma 12 to conclude.

• [REF], [REF−!], [REF−?]

It is straightforward.

• [RED]

t_1 and t_2 are respectively equal to $(s, E[r])$ and $(s', E[r'])$, where E is an evaluation context and r a redex such that $(s, r) \rightarrow (s', r')$.

If t_1 is origin coherent, then so are E and (s, r), by using Lemma 11, so is (s, r') by the inductive hypothesis, and finally t_2 is origin coherent by Lemma 12. □

We now prove that the two frontier upper bounds given in Definition 7 (p. 13) are preserved by reduction. We only consider well-typed configurations in the following: note that by the subject reduction property (cf. Prop. 1, p. 7), if a well-typed configuration t reduces to t', then t' is well-typed. We recall that given an environment type L, we denote by $\mathcal{A}_{\mathsf{lo}}(L)$ the set of outgoing types of L and by $\mathcal{A}_{\mathsf{mo}}(L)$ the set of incoming types of L.
Some preliminary lemmas are useful. In these lemmas, we suppose that ι is an origin label, either lo or mo. The ι-frontier of a configuration (s, e) (cf. Definition 6, p. 10) is denoted by $\mathcal{F}_\iota(s, e)$; it is useful to decompose $\mathcal{F}_\iota(s, e)$ in $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(e)$, where $\mathcal{F}_\iota(s)$ and $\mathcal{F}_\iota(e)$ are the ι-frontiers of the memory store s and of the term e, respectively, and are defined as for configurations in Definition 6 (p. 10). Given a label (A_1, ι_1), let us define a useful function, $\mathcal{L}_\iota^{(A_1,\iota_1)}$, which maps each labeled term e to a set of types, as follows:

$$\mathcal{L}_\iota^{(A_1,\iota_1)}(e) \stackrel{def}{=} \begin{cases} \{A_2\} & \text{if } e = \mathtt{f}^{(A_2,\iota_2)}(\ldots) \text{ and } \iota = \iota_2 = \overline{\iota_1}\,, \\ \emptyset & \text{otherwise.} \end{cases}$$

This function, which computes the ι-frontier at the top of term patterns $\mathsf{g}^{(A_1,\iota_1)}(\ldots, e, \ldots)$, leads to the following formal definition of the term frontier by induction:

$$\begin{array}{rcl} \mathcal{F}_\iota(x) & = & \emptyset\,, \\ \mathcal{F}_\iota(\mathtt{f}^n(e_j)_j) & = & \bigcup_j \mathcal{F}_\iota(e_j) \cup \mathcal{L}_\iota^n(e_j)\,. \end{array}$$

As for a memory store s, the frontier is formally defined as follows:

$$\mathcal{F}_\iota(s) \quad = \bigcup_{l^m \in \mathrm{dom}\, s} \mathcal{F}_\iota(s(l^m)) \cup \mathcal{L}_\iota^m(s(l^m))\,.$$

We begin by computing the frontier of a closed term in an evaluation context. An evaluation context is said to *reduce under* the label n if it contains as a subterm $\mathtt{f}^n(\ldots, -, \ldots)$, for some operator $\mathtt{f}$. This definition is justified by the following lemma.

Lemma 14 (Frontier and reduction)
Let E be a labeled evaluation context reducing under n and let e be a labeled closed term. Then we have:

$$\mathcal{F}_\iota(E[e]) = \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(e) \cup \mathcal{L}_\iota^n(e)\,.$$

Proof. It is straightforward by induction on E. □

Let us consider substitution effects.

Lemma 15 (Frontier and substitution)
Let e be a term labeled with n, a a labeled term and x a variable, such that e is origin coherent. If x is free in e, then we have:

$$\mathcal{F}_\iota(e[a/x]) = \mathcal{F}_\iota(e) \cup \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^n(a)\,.$$

Note that we assume the origin coherence of e in order to use on the right hand side its label, n.

Proof. We show by induction on e that if e is origin coherent, is not a variable and contains a free occurrence of x, then the expected equality is satisfied.
The case where e is a variable being trivial, we only consider the interesting case where the labeled term e is equal to $\mathtt{f}^n(e_1, \ldots, e_j)$ for some operator $\mathtt{f}$ and some label n equal to (A_0, ι_0). We suppose that e is origin coherent and contains a free occurrence of x.
By definition, we have

$$\mathcal{F}_\iota(e[a/x]) = \bigcup_{1 \leq k \leq j} \mathcal{F}_\iota(e_k[a/x]) \cup \mathcal{L}_\iota^n(e_k[a/x])\,.$$

We must show

$$\mathcal{F}_\iota(e[a/x]) = \big(\bigcup_{1 \leq k \leq j} \mathcal{F}_\iota(e_k) \cup \mathcal{L}_\iota^n(e_k)\big) \cup \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^n(a)\,.$$

Let k be an integer in $\{1, \ldots, j\}$. We consider two cases according to whether x is a free variable in e_k or not.
- $x \in \mathbf{FV}(e_k)$

If $e_k = x$, then

$$\begin{array}{rcl} \mathcal{F}_\iota(e_k[a/x]) \cup \mathcal{L}_\iota^n(e_k[a/x]) & = & \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^n(a)\,, \\ \mathcal{F}_\iota(e_k) \cup \mathcal{L}_\iota^n(e_k) & = & \emptyset\,. \end{array}$$

Let us suppose $e_k \neq x$. Since x is free in e_k, the term e_k cannot be a variable and is thus labeled, say with (A_k, ι_k).
Since e is origin coherent, e_k is origin coherent and we have $\iota_0 = \iota_k$. From the inductive hypothesis applied to e_k, we obtain

$$\mathcal{F}_\iota(e_k[a/x]) = \mathcal{F}_\iota(e_k) \cup \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^{(A_k,\iota_k)}(a)\,.$$

Since $\iota_0 = \iota_k$, we have $\mathcal{L}_\iota^{(A_k,\iota_k)}(a) = \mathcal{L}_\iota^n(a)$. Since $\mathcal{L}_\iota^n(e_k[a/x]) = \mathcal{L}_\iota^n(e_k)$, we finally deduce:

$$\mathcal{F}_\iota(e_k[a/x]) \cup \mathcal{L}_\iota^n(e_k[a/x]) = \mathcal{F}_\iota(e_k) \cup \mathcal{L}_\iota^n(e_k) \cup \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^n(a)\,.$$

- $x \notin \mathbf{FV}(e_k)$

We then have $e_k[a/x] = e_k$, and therefore:

$$\mathcal{F}_\iota(e_k[a/x]) \cup \mathcal{L}_\iota^n(e_k[a/x]) = \mathcal{F}_\iota(e_k) \cup \mathcal{L}_\iota^n(e_k)\,.$$

Finally, by gathering all the cases, and by noting that there exists at least some k such that $x \in \mathbf{FV}(e_k)$, we obtain:

$$\mathcal{F}_\iota(e[a/x]) = \mathcal{F}_\iota(e) \cup \mathcal{F}_\iota(a) \cup \mathcal{L}_\iota^n(a)\,.$$

□

We now study how the frontier evolves when the memory store is updated.

Lemma 16 (Frontier and memory store updating)
Let s be a memory store and l^n a labeled reference belonging to $\operatorname{dom} s \cup \{\nu_n(s)\}$. *Then we have:*

$$\mathcal{F}_\iota((s, l^n \mapsto v)) \subseteq \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \cup \mathcal{L}_\iota^n(v)\,.$$

Proof. It follows from the definitions of the updating function and of the frontier. □

We can now prove that the frontier upper bounds are preserved by reduction. We are going to proceed by induction on the proof of the reduction. Firstly, we consider base cases.

Lemma 17 $\left(\begin{array}{l}\textbf{Preservation of frontier upper bounds}\\ \boldsymbol{\beta}\textbf{-reduction}\end{array}\right)$
Let L be a type. Consider a β-reduction $(s,r) \to (s,r')$ such that

(i) the configuration (s,r) is origin coherent,

(ii) for all origin labels ι, the ι-frontier of (s,r) is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s,r) \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of (s,r') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s,r') \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. Suppose that the reduction $(s,r) \to (s,r')$ satisfying the preceding conditions is

$$\big(s, @^{(B,\iota_1)}(\lambda x^{(A\to B,\iota_2)}\, b, v)\big) \to (s, b[v/x])\,.$$

Let ι be an origin label. In order to show $\mathcal{F}_\iota(s,b[v/x]) \subseteq \mathcal{A}_\iota(L)$, we examine the different cases for b.

- $b = x$

Since $b[v/x] = v$, it is straightforward.

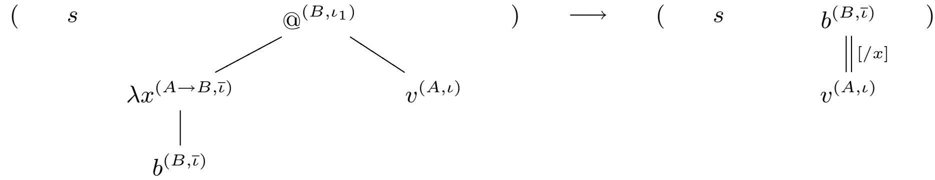
- $b = \mathtt{f}^{(B,\iota_3)}(\dots)$

If $x \notin \mathbf{FV}(b)$, then $b[v/x] = b$; we can conclude. Let us suppose $x \in \mathbf{FV}(b)$.
Since b is origin coherent, from Lemma 15, we deduce

$$\mathcal{F}_\iota(b[v/x]) = \mathcal{F}_\iota(b) \cup \mathcal{F}_\iota(v) \cup \mathcal{L}_\iota^{(B,\iota_3)}(v)\,.$$

It remains to prove $\mathcal{L}_\iota^{(B,\iota_3)}(v) \subseteq \mathcal{A}_\iota(L)$.
Suppose $\mathcal{L}_\iota^{(B,\iota_3)}(v) \neq \emptyset$: we then have that v is labeled with (A,ι), $\mathcal{L}_\iota^{(B,\iota_3)}(v) = \{A\}$ and $\iota_3 = \bar{\iota}$.
Since r is origin coherent, we have $\iota_2 = \iota_3$. The reduction is represented in Figure 9, where the created frontier is indicated with a double line.

Figure 9: Proof of Lemma 17 - Interesting case



If $\iota_1 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(r)$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.
If $\iota_1 = \iota$, then $A \to B$ belongs to $\mathcal{F}_{\bar{\iota}}(r)$, which is included in $\mathcal{A}_{\bar{\iota}}(L)$ by hypothesis. By the inference rule $[- \to]$ (cf. Table 7, p. 12), we obtain that A belongs to $\mathcal{A}_\iota(L)$. □

Lemma 18 $\left(\begin{array}{l}\textbf{Preservation of frontier upper bounds}\\ [\text{REF}-?]\ \textbf{reduction}\end{array}\right)$
Let L be a type. Consider a $[\text{REF}-?]$ reduction $(s,r) \to (s',r')$ such that for all origin labels ι, the ι-frontier of (s,r) is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s,r) \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of (s',r') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s',r') \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. Suppose that the [REF−?] reduction $(s,r) \to (s',r')$ satisfying the preceding conditions is

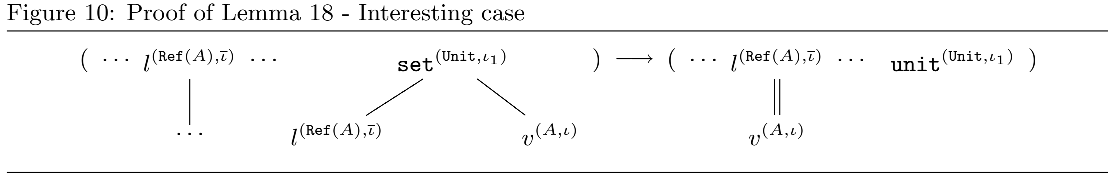
$$\big(s, \texttt{set}^{(\texttt{Unit},\iota_1)}(l^{(\texttt{Ref}(A),\iota_2)}, v)\big) \to \big((s, l^{(\texttt{Ref}(A),\iota_2)} \mapsto v), \texttt{unit}^{(\texttt{Unit},\iota_1)}\big)\,.$$

Let ι be an origin label. By Lemma 16, we have

$$\mathcal{F}_\iota(s',r') \subseteq \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \cup \mathcal{L}_\iota^{(\texttt{Ref}(A),\iota_2)}(v)\,.$$

It thus remains to prove $\mathcal{L}_\iota^{(\texttt{Ref}(A),\iota_2)}(v) \subseteq \mathcal{A}_\iota(L)$.
Suppose $\mathcal{L}_\iota^{(\texttt{Ref}(A),\iota_2)}(v) \neq \emptyset$: we then have that v is labeled with (A,ι), $\mathcal{L}_\iota^{(\texttt{Ref}(A),\iota_2)}(v) = \{A\}$ and $\iota_2 = \bar{\iota}$. The reduction is represented in Figure 10, where the created frontier is indicated with a double line.

Figure 10: Proof of Lemma 18 - Interesting case



If $\iota_1 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(r)$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.
Otherwise, $\iota_1 = \iota$ and $\texttt{Ref}(A)$ belongs to $\mathcal{F}_{\bar{\iota}}(r)$, which is included in $\mathcal{A}_{\bar{\iota}}(L)$ by hypothesis; by the inference rule [$\texttt{Ref}(-)$] (cf. Table 7 , p. 12), we obtain that A belongs to $\mathcal{A}_\iota(L)$. □

Lemma 19 $\left(\begin{array}{c}\textbf{Preservation of frontier upper bounds}\\ \text{[REF] } \textbf{or} \text{ [REF−!] } \textbf{reduction}\end{array}\right)$
Let L be a type. Consider a [REF] *or* [REF−!] *reduction* $(s,r) \to (s',r')$ *such that for all origin labels ι, the ι-frontier of (s,r) is included in $\mathcal{A}_\iota(L)$:*

$$\mathcal{F}_\iota(s,r) \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of (s',r') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s',r') \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. It can be easily shown that $\mathcal{F}_\iota(s',r') \subseteq \mathcal{F}_\iota(s,r)$. □

Lemmas 17, 18 and 19 allow concluding that all base cases are satisfied. Secondly, we consider the inference rule [RED], involving an evaluation context. We examine the different cases for the reduction premise, which is a base case; for every following lemma, we assume the inductive hypothesis.

Lemma 20 $\left(\begin{array}{c}\textbf{Preservation of frontier upper bounds}\\ \beta\textbf{-reduction in evaluation context}\end{array}\right)$
Let L be a type. Consider a [RED] *reduction rule*

$$\frac{(s,r) \to (s',r')}{(s,E[r]) \to (s',E[r'])} \quad \text{(evaluation context } E \neq -)$$

such that

(i) $(s,r) \to (s',r')$ *is a β-reduction,*

(ii) for all origin labels ι, the ι-frontier of $(s,E[r])$ is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s,E[r]) \subseteq \mathcal{A}_\iota(L)\,,$$

(iii) for all origin labels ι, the ι-frontier of (s', r') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of $(s', E[r'])$ is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s', E[r']) \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. Suppose that the reduction $(s, E[r]) \to (s', E[r'])$ satisfying the preceding conditions is

$$\big(s, E[@^{(B,\iota_1)}(\lambda x^{(A\to B,\iota_2)}\, b, v)]\big) \to (s, E[b[v/x]])\,.$$

We also assume that the evaluation context reduces under m, which has ι_0 as its origin label.
Let ι be an origin label. By Lemma 14, we have

$$\mathcal{F}_\iota(s, E[b[v/x]]) = \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(b[v/x]) \cup \mathcal{L}^m_\iota(b[v/x])\,.$$

From the hypotheses, we deduce

$$\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(b[v/x]) \subseteq \mathcal{A}_\iota(L)\,.$$

It remains to prove $\mathcal{L}^m_\iota(b[v/x]) \subseteq \mathcal{A}_\iota(L)$.
Suppose $\mathcal{L}^m_\iota(b[v/x]) \neq \emptyset$: we then have that $b[v/x]$ is labeled with (B, ι), $\mathcal{L}^m_\iota(b[v/x]) = \{B\}$ and $\iota_0 = \bar{\iota}$. We consider two cases according to whether b is equal to x or not.

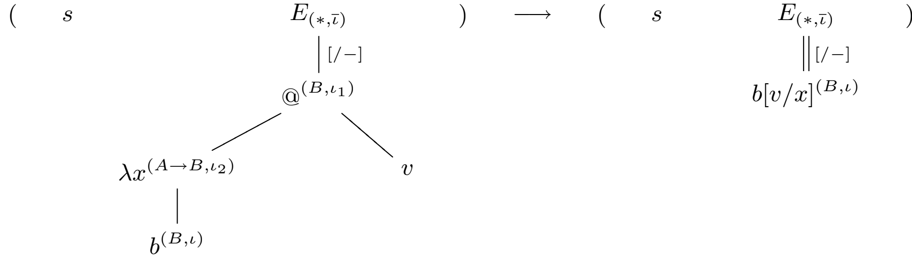
- $b = x$

We have $b[v/x] = v$ and $A = B$. Whatever ι_1 is, A belongs to $\mathcal{F}_\iota(E[r])$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.

- $b = \mathtt{f}^{(B,\iota_3)}(\dots)$

Since $b[v/x]$ is labeled with (B, ι), we have $\iota_3 = \iota$. The reduction is represented in Figure 11, where the created frontier is indicated with a double line.

Figure 11: Proof of Lemma 20 - Interesting case



If $\iota_1 = \iota$, then B belongs to $\mathcal{F}_\iota(E[r])$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \bar{\iota}$, then B belongs to $\mathcal{F}_\iota(r)$, which is also included in $\mathcal{A}_\iota(L)$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \iota$, then $A \to B$ belongs to $\mathcal{F}_\iota(r)$, included in $\mathcal{A}_\iota(L)$ by hypothesis; by the inference rule $[\to +]$ (cf. Table 7, p. 12), we obtain that B belongs to $\mathcal{A}_\iota(L)$. □

Lemma 21 $\left(\begin{array}{c}\textbf{Preservation of frontier upper bounds}\\ \text{[REF}-!\text{]}\ \textbf{reduction in evaluation context}\end{array}\right)$
Let L be a type. Consider a [RED] *reduction rule*

$$\frac{(s, r) \to (s', r')}{(s, E[r]) \to (s', E[r'])} \quad (\text{evaluation context } E \neq -)$$

such that

(i) $(s,r) \to (s',r')$ *is a* [REF−!] *reduction,*

(ii) for all origin labels ι*, the* ι*-frontier of* $(s, E[r])$ *is included in* $\mathcal{A}_\iota(L)$*:*

$$\mathcal{F}_\iota(s, E[r]) \subseteq \mathcal{A}_\iota(L)\,,$$

(iii) for all origin labels ι*, the* ι*-frontier of* (s',r') *is included in* $\mathcal{A}_\iota(L)$*:*

$$\mathcal{F}_\iota(s',r') \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι*, the* ι*-frontier of* $(s', E[r'])$ *is included in* $\mathcal{A}_\iota(L)$*:*

$$\mathcal{F}_\iota(s', E[r']) \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. Suppose that the reduction $(s, E[r]) \to (s', E[r'])$ satisfying the preceding conditions is

$$(s, E[\mathtt{get}^{\,(A,\iota_1)}(l^{(\mathtt{Ref}(A),\iota_2)})]) \to (s, E[s(l^{(\mathtt{Ref}(A),\iota_2)})])\,.$$

We also assume that the evaluation context reduces under m, which has ι_0 as its origin label.
By Lemma 14, we have

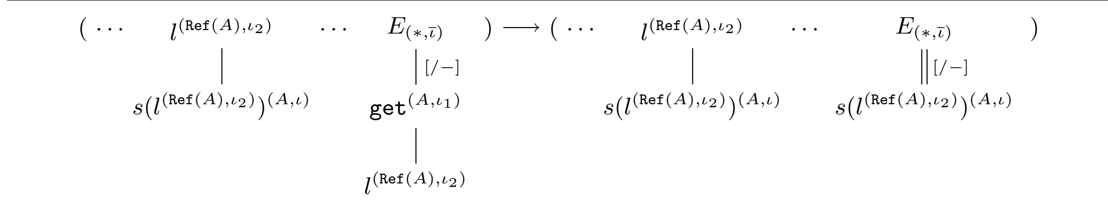
$$\mathcal{F}_\iota(s, E[s(l^{(\mathtt{Ref}(A),\iota_2)})]) = \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)})) \cup \mathcal{L}^m_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)}))\,.$$

From the hypotheses, we deduce

$$\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)})) \subseteq \mathcal{A}_\iota(L)\,.$$

It remains to prove $\mathcal{L}^m_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)})) \subseteq \mathcal{A}_\iota(L)$.
Suppose $\mathcal{L}^m_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)})) \neq \emptyset$: we then have that $s(l^{(\mathtt{Ref}(A),\iota_2)})$ is labeled with (A,ι), $\mathcal{L}^m_\iota(s(l^{(\mathtt{Ref}(A),\iota_2)})) = \{A\}$ and $\iota_0 = \bar{\iota}$. The reduction is represented in Figure 12, where the created frontier is indicated with a double line.

Figure 12: Proof of Lemma 21 - Interesting case



If $\iota_1 = \iota$, then A belongs to $\mathcal{F}_\iota(E[r])$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(s)$, which is included in $\mathcal{A}_\iota(L)$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \iota$, then $\mathtt{Ref}(A)$ belongs to $\mathcal{F}_\iota(r)$, included in $\mathcal{A}_\iota(L)$ by hypothesis; by the inference rule [$\mathtt{Ref}(+)$] (cf. Table 7, p. 12), we obtain that A belongs to $\mathcal{A}_\iota(L)$. □

Lemma 22 $\left(\begin{array}{c}\textbf{Preservation of frontier upper bounds}\\ \text{[REF] } \textbf{or} \text{ [REF−?] } \textbf{reduction in evaluation context}\end{array}\right)$
Let L *be a type. Consider a* [RED] *reduction rule*

$$\frac{(s,r) \to (s',r')}{(s, E[r]) \to (s', E[r'])} \quad \text{(evaluation context } E \neq -\text{)}$$

such that

(i) $(s,r) \to (s',r')$ *is a* [REF] *or* [REF−?] *reduction,*

(ii) for all origin labels ι, the ι-frontier of $(s, E[r])$ is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s, E[r]) \subseteq \mathcal{A}_\iota(L)\,,$$

(iii) for all origin labels ι, the ι-frontier of (s', r') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s', r') \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of $(s', E[r'])$ is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s', E[r']) \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. Assume that the evaluation context reduces under m. We have:

$$\begin{array}{rcl} \mathcal{F}_\iota(s, E[r]) & = & \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r) \cup \mathcal{L}^m_\iota(r)\,, \\ \mathcal{F}_\iota(s', E[r']) & = & \mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r') \cup \mathcal{L}^m_\iota(r')\,. \end{array}$$

From the hypotheses, we deduce $\mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(r') \cup \mathcal{F}_\iota(E) \subseteq \mathcal{A}_\iota(L)$; by inspection of the reduction rules, we remark that $\mathcal{L}^m_\iota(r') = \mathcal{L}^m_\iota(r)$, which gives by hypothesis $\mathcal{L}^m_\iota(r') \subseteq \mathcal{A}_\iota(L)$. We can therefore conclude. □

We can conclude that the frontier upper bounds $\mathcal{A}_{\mathsf{lo}}(L)$ and $\mathcal{A}_{\mathsf{mo}}(L)$ are preserved by reduction.

Theorem 23 $\left(\begin{array}{l}\textbf{Frontier upper bounds}\\ \textbf{Preservation by reduction}\end{array}\right)$
Let L be a type. Consider a reduction $(s, e) \to (s', e')$ between well-typed configurations such that

(i) the configuration (s, e) is origin coherent,

(ii) for all origin labels ι, the ι-frontier of (s, e) is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s, e) \subseteq \mathcal{A}_\iota(L)\,.$$

Then for all origin labels ι, the ι-frontier of (s', e') is included in $\mathcal{A}_\iota(L)$:

$$\mathcal{F}_\iota(s', e') \subseteq \mathcal{A}_\iota(L)\,.$$

Proof. We proceed by induction on the reduction by using Lemmas 17, 18, 19, 20, 21 and 22. It is straightforward. □

B Optimality of the confinement criterion

We prove Theorem 9 (p. 14), which is the converse of Theorem 8 (p. 13): given a resource type A and an environment type L of whom the resource type A is an outgoing type (cf. Definition 7, p. 13), we must define a local environment L of type L and a mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$ in order that a frontier redex destructs a value of the type A and of the origin label lo during the execution of the labeled program

$$@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})\,.$$

We define the local environment and the mobile program accessing to the local resource by induction on the environment type. Consider the case where the environment type is a functional type $L_1 \to L_2$. If A is an outgoing type of $L_1 \to L_2$, then there are two possibilities when A is different from $L_1 \to L_2$:

- either A is an incoming type of L_1,
- or A is an outgoing type of L_2.

For the latter possibility, the inductive hypothesis can be applied, whereas for the former, it cannot, so that we proceed differently: we show that it is possible to decompose L_1 in order to obtain a type of whom A is an outgoing type.

We begin by formalizing the preceding properties about outgoing and incoming types.

Proposition 24
Let C and A be two distinct types. We have:

(i) if C is equal to $D_1 \to D_2$, then A is an outgoing (resp. incoming) type of C if and only if A is an outgoing (resp. incoming) type of D_2 or an incoming (resp. outgoing) type of D_1.

(ii) if C is equal to $\texttt{Ref}(D)$, then A is an outgoing (resp. incoming) type of C if and only if A is an outgoing (resp. incoming) type of D or an incoming (resp. outgoing) type of D.

Proof. From left to right, it is straightforward by induction on the proof that A is an outgoing (resp. incoming) type of C (cf. Table 7 (p. 12) for the definition of the inference rules).
From right to left, it is better to use the following equivalent form: if A is an outgoing (resp. incoming) type of D, then

(i) for all types D', A is an incoming (resp. outgoing) type of $D \to D'$,

(ii) for all types D', A is an outgoing (resp. incoming) type of $D' \to D$,

(iii) A is an outgoing type and an incoming type of $\texttt{Ref}(D)$.

It is straightforward by induction on the proof that A is an outgoing (resp. incoming) type of D. □

The proof of Theorem 9 (p. 14) is mainly a programming exercise. We use the programming language defined in Table 1 (p. 5). Its operational semantics can be deduced from the reduction relation defined in Table 4 (p. 8) for the annotated language; we then need two useful functions for creating references and for updating memory stores, as described for the annotated language (cf. p. 7): given a reference type A and a memory store s, $\nu_A(s)$ creates a new reference of type A in the memory store s, and given a memory store s, a reference l^A in $\operatorname{dom} s \cup \nu_A(s)$ and a value v, then $(s, l^A \mapsto v)$ updates or extends s by assigning v to l^A.
We first show that each type is inhabited by a convergent program, which is useful to define default values.

Lemma 25
For each type A, there exists a program a_A, a value u_A and a memory store s_A, such that the configuration $(\emptyset, a_A)$ evaluates to (s_A, u_A).

Proof. We proceed by induction on A.
- $A = \texttt{Unit}$

Put $a_A = u_A = \texttt{unit}$ and $s_A = \emptyset$.
- $A = A_1 \to A_2$

Put $a_A = u_A = \lambda x : A_1 .\, a_{A_2}$ and $s_A = \emptyset$.
- $A = \texttt{Ref}(A_1)$

Put $a_A = \texttt{ref}(a_{A_1})$, $u_A = \nu_A(s_{A_1})$ and $s_A = (s_{A_1}, u_A \mapsto u_{A_1})$. □

The following notations ease programming.
If e_1 and e_2 are two terms and x a variable, which is not free in e_1 and has type A in e_2, then the expression

$$\texttt{let}\ A\, x = e_1\ \texttt{in}\ e_2$$

declares x as a local variable of type A in e_2, initializes x with e_1 and returns e_2; it can be defined as follows:

$$\texttt{let}\ A\, x = e_1\ \texttt{in}\ e_2 \stackrel{def}{=} (\lambda x : A.\, e_2)\, e_1\,.$$

If e_1 and e_2 are two terms, then $e_1\,;\,e_2$ is the sequential composition of e_1 and e_2, and is defined as follows:

$$e_1\,;\,e_2 \stackrel{def}{=} (\lambda x\, e_2)\, e_1\,,$$

where x is not free in e_2 (the type of x is omitted).
If f et g have types $A_1 \to A_2$ and $A_2 \to A_3$ respectively, then $g \circ f$ is the functional composition of f and g and is defined as follows:

$$g \circ f \stackrel{def}{=} \lambda x : A_1.\, g\,(f\,x)\,,$$

where x is free neither in f nor in g.
Finally, $s_1 \cdot \ldots \cdot s_n$ denotes the memory store s if the family $(s_i)_i$ and the memory store s satisfy all the following conditions:

(i) for each i, s is an extension of s_i,

(ii) $(\mathrm{dom}\, s_i)_i$ is a partition of s,

(iii) for each i, the set $\{l \mid \exists j \leq i\,.\, \exists A\,.\, l^A \in \mathrm{dom}\, s_j\}$ is an initial segment of the set of reference identifiers.

In other words, $s_1 \cdot \ldots \cdot s_n$ is a well-ordered decomposition of s.
The following lemma gives a fundamental property of incoming types, as explained after its statement.

Lemma 26
Let L and A be two types such that A is an incoming type of L. Then there exists two types L' and L'', and two programs δ and et γ, of type $L \to L'$ and $L' \to L$ respectively, such that:

(i) L' is a subtree of L,

(ii) L' is equal to $\mathtt{Ref}(L'')$ or $L'' \to B$ for some type B,

(iii) A is an incoming type of L',

(iv) A is an outgoing type of L'',

(v) the programs δ and γ are both abstractions,

(vi) for any configuration (s_1, v), where s_1 is a memory store and v a value of type L', there exists a value u of type L and a memory store s such that

(a) $(s_1, \gamma\, v)$ evaluates to $(s_1 \cdot s, u)$,

(b) for any configuration $(s_2 \cdot s \cdot s_3, \delta\, u)$, where s_2 and s_3 are memory stores, there exists s_4 such that $(s_2 \cdot s \cdot s_3, \delta\, u)$ evaluates to $(s_2 \cdot s \cdot s_3 \cdot s_4, v)$.

If we forget the side effects, the lemma asserts that $\delta \circ \gamma$ is the identity function. By allowing going from L' to L and then coming back, the two programs enable using the inductive hypothesis in L''. Before describing how to proceed, let us prove this lemma.

Proof. We proceed by induction on L. Let A be a type. We show that if A is an incoming type of L, then two types, L' et L'', and two programs, δ et γ, satisfying all the preceding conditions, can be defined.
• $L = \mathtt{Unit}$
L has no incoming types.
• $L = L_1 \to L_2$
Suppose that A is an incoming type of L. By Proposition 24, there exists two possibilities, which we detail.
◦ A is an outgoing type of L_1
Let us define L' as L, L'' as L_1, δ and γ as the identity function $\lambda x : L.\, x$.
◦ A is an incoming type of L_2
The inductive hypothesis applied to L_2 gives L', L'', δ of type $L_2 \to L'$ and γ of type $L' \to L_2$.

Write F for $\lambda f : L.\, f\, a_{L_1}$ of type $L \to L_2$ and G for $\lambda y : L_2.\, \lambda x : L_1.\, y$ of type $L_2 \to L$. It is easy to prove that L', L'', $\delta \circ F$ and $G \circ \gamma$ are suitable.

• $L = \mathtt{Ref}(L_1)$

Suppose that A is an incoming type of L. By Proposition 24, there exists two possibilities, which we detail.

∘ A is an outgoing type of L_1

Let us define L' as L, L'' as L_1, δ and γ as the identity function $\lambda x : L.\, x$.

∘ A is an incoming type of L_1

The inductive hypothesis applied to L_1 gives L', L'', δ and γ.

Write F for $\lambda x : L.\, \mathtt{get}(x)$ of type $L \to L_1$ and G for $\lambda x : L_1.\, \mathtt{ref}(x)$ of type $L_1 \to L$. It is easy to prove that L', L'', $\delta \circ F$ and $G \circ \gamma$ are suitable. □

We can now prove Theorem 9 (p. 14). We do not actually show that the execution traces of the programs that we define satisfy the intended property. Informally, the reader should be convinced that it works: the proof would be based on the conditions *(v)* and *(vi)* of Lemma 26, which we have chosen in order to simplify the verification, and on some intuitive properties of execution traces. Without computer aid, it should be very long to fully formalize the proof.

Proof. **(Theorem 9 - Criterion optimality)**

We proceed by induction on L. Given a type A different from $\mathtt{Unit}$, we show that if A is an outgoing type of L, then we can define a local environment L of type L and a mobile program $\lambda x : L.\, \mathsf{M}[x]$ of type $L \to M$ (for some type M) such that a frontier redex destructs a value of the type A and of the origin label lo during the execution of the labeled program

$$@^{(M,\mathsf{mo})}(\langle \lambda x : L.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}}),$$

denoted by $\mathcal{M}(\mathsf{M}, \mathsf{L})$ in the following.

The case where A is equal to L being trivial, we suppose that A is different from L.

• $L = \mathtt{Unit}$

L has no outgoing types, except itself.

• $L = L_1 \to L_2$

Suppose that A is an outgoing type of L. Proposition 24 gives two possibilities, which we detail.

∘ A is an outgoing type of L_2

The inductive hypothesis applied to L_2 gives a local environment L of type L_2 and a mobile program $\lambda x : L_2.\, \mathsf{M}[x]$ such that the execution of $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entails the intended destruction. One easily verify that

$$\lambda x : L_1.\, \mathsf{L}$$

and

$$\lambda f : L.(\lambda x : L_2.\, \mathsf{M}[x])\,(f\, a_{L_1})$$

are suitable as a local environment and a mobile program respectively.

∘ A is an incoming type of L_1

Lemma 26 gives two types L' and L'', and two programs δ et γ, of type $L_1 \to L'$ and $L' \to L_1$ respectively, satisfying the different conditions. Since A is an outgoing type of L'', the inductive hypothesis applied to L'' gives a local environment L of type L'' and a mobile program $\lambda x : L''.\, \mathsf{M}[x]$ such that the execution of $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entails the intended destruction. Consider two cases, according to the relationship between L'' and L' (cf. condition *(ii)* of Lemma 26).

∘ $L' = L'' \to B$

The new local environment is defined by

$$\mathsf{L}' \,:\, L \overset{def}{=} \lambda x : L_1.(\delta\, x\, \mathsf{L}\,;\, a_{L_2})$$

and the body of the new mobile program by

$$\mathsf{M}'[f \,:\, L] \overset{def}{=} f\,(\gamma\ \lambda x : L''.(\mathsf{M}[x]\,;\, a_B))\,.$$

One easily verify that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended destruction.
∘ $L' = \mathtt{Ref}(L'')$
The new local environment is defined by

$$\mathsf{L}' : L \stackrel{def}{=} \lambda x : L_1.(\mathtt{set}(\delta\, x, \mathsf{L}) \,;\, a_{L_2})$$

and the body of the new mobile program by

$$\mathsf{M}'[f : L] \stackrel{def}{=} \mathtt{let}\ L'\, z = a_{L'}\ \mathtt{in}\ (f\,(\gamma\, z) \,;\, (\lambda x : L''.\, \mathsf{M}[x])\, \mathtt{get}(z))\,.$$

One easily verify that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended destruction.
• $L = \mathtt{Ref}(L_1)$
Suppose that A is an outgoing type of L. Proposition 24 gives two possibilities, which we detail.
∘ A is an outgoing type of L_1
The inductive hypothesis applied to L_1 gives a local environment L of type L_1 and a mobile program $\lambda x : L_1.\, \mathsf{M}[x]$ such that the execution of $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entails the intended destruction. One easily verify that

$$\mathtt{ref}(\mathsf{L})$$

and

$$\lambda z : L.\, \mathtt{let}\ L_1\, x = \mathtt{get}(z)\ \mathtt{in}\ \mathsf{M}[x]$$

are suitable as a local environment and a mobile program respectively.
∘ A is an incoming type of L_1
Lemma 26 gives two types L' and L'', and two programs δ et γ, of type $L_1 \to L'$ and $L' \to L_1$ respectively, satisfying the different conditions. Since A is an outgoing type of L'', the inductive hypothesis applied to L'' gives a local environment L of type L'' and a mobile program $\lambda x : L''.\, \mathsf{M}[x]$ such that the execution of $\mathcal{M}(\mathsf{M}, \mathsf{L})$ entails the intended destruction. Consider two cases, according to the relationship between L'' and L' (cf. condition *(ii)* of Lemma 26).
∘ $L' = L'' \to B$
The new local environment is defined by

$$\mathsf{L}' : L \stackrel{def}{=} \mathtt{let}\ L\, z = a_L\ \mathtt{in}\ (\mathtt{set}(z, \gamma\ \lambda x : L''.\, \delta\, \mathtt{get}(z)\, \mathsf{L}) \,;\, z)$$

and the body of the new mobile program by

$$\mathsf{M}'[z : L] \stackrel{def}{=} \mathtt{let}\ L_1\, y = \mathtt{get}(z)\ \mathtt{in}\ (\mathtt{set}(z, \gamma\ \lambda x : L''.(\mathsf{M}[x] \,;\, a_B)) \,;\, \delta\, y\, a_{L''})\,.$$

One easily verify that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended destruction.
∘ $L' = \mathtt{Ref}(L'')$
The new local environment is defined by

$$\mathsf{L}' : L \stackrel{def}{=} \mathtt{let}\ L'\, y = \mathtt{ref}(\mathsf{L})\ \mathtt{in}\ \mathtt{ref}(\gamma\, y)$$

and the body of the new mobile program by

$$\mathsf{M}'[z : L] \stackrel{def}{=} \mathtt{let}\ L''\, y = \mathtt{get}(\delta\, \mathtt{get}(z))\ \mathtt{in}\ \mathsf{M}[y]\,.$$

One easily verify that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended destruction.

□

C The SLam-calculus: From access controls to confinement

In the second and last paragraph of the conclusion (cf. p. 16–17), we have described the relationship between a standard security analysis and our confinement criterion. We now formalize this relationship: we adapt Heintze and Riecke's solution [10] to our programming language, which is a restriction of their language, the SLam-calculus ("Secure Lambda-calculus"), and show how their type system, which enforces security controls, leads to a confinement criterion equivalent to ours. We restrict ourselves to access controls, whereas the SLam-calculus also deals with information flows.

We again resort to an annotated language which keeps track of origin, as defined in Section 2 (p. 5). However, the purpose is different: a label does not indicate the origin of code, the mobile program or the local environment, but its security status. We thus introduce two *security labels*, $\bot$ and $\top$, ordered by $\bot < \top$: the security label $\top$ is intended to mark sensitive resources and secured access functions. Types are also labeled and are generated by the following grammar:

$$\begin{array}{rcll} A & ::= & \mathtt{Unit} & \text{(singleton type)} \\ & \mid & A \xrightarrow{\bot} A \mid A \xrightarrow{\top} A & \text{(functional types)} \\ & \mid & \mathtt{Ref}^{\bot}(A) \mid \mathtt{Ref}^{\top}(A) & \text{(reference types)}\,. \end{array}$$

In the following, the type A, labeled by σ in $\{\bot, \top\}$, is also denoted by A^σ in order to stress the security label. We also need to define the set of labels for the annotated language. Recall that a label annotating an operator is an ordered pair whose first component is a standard type and second component a piece of information (cf. Sect. 2, p. 5). If the information label simply was the security label, then we could not deduce from the label of a term its labeled type: for instance, the identity program $\lambda x^{(A,\top)}\, x$, where A is a standard type, would have any labeled type $A' \xrightarrow{\top} A'$, where A' results from a labeling of the type A. That is the reason why we define the information label as a security label associated with a labeled type. Since a labeled type gives the underlying standard type by erasing labels, we can simplify: a label becomes an ordered pair (A, σ), where A is a labeled type, called the *type label*, and where σ belongs to $\{\bot, \top\}$ and is called the *security label*. Thus, the preceding program becomes $\lambda x^{(A',\top)}\, x$, with type $A' \xrightarrow{\top} A'$; we will see that this label definition enables determining a least type for a term.
The annotated type system is described by the inference system in Table 8; a typing judgment has the form $\Gamma \vdash e : A$, where Γ is an environment, e a labeled term and A a labeled type. Note the following points:

- given a term, its type label gives its labeled type,
- a value (or a redex creating a new reference) labeled by σ in $\{\bot, \top\}$ has a type labeled by σ,
- elimination rules enforce a security control: the security label of a destructor is greater than the security label of the term to which it is applied.

The subtyping relation is inductively generated by the inference system given in Table 9: judgments are inequalities $A \leq B$, meaning that A is a subtype of B. The subtyping relation enforces the substitution principle (cf. Liskov and Wing's definition [14]) stated as follows: given any type constructor $\mathtt{F}$, equal to $\mathtt{Ref}(-)$ or $- \to -$, for any context $C_{[\,]}$, if for all values v of type $\mathtt{F}^{\top}(\ldots)$, the program $C[v]$ is secure, then for all values v of type $\mathtt{F}^{\bot}(\ldots)$, the program $C[v]$ is secure. Note the following points:

- for the type constructor $- \to -$, the subtyping rule is contravariant on the left component (the domain type) and covariant on the right component (the codomain type),

Table 8: Annotated type system

$$\frac{\emptyset}{\Gamma \vdash x : \Gamma(x)} \quad (x \in \mathrm{dom}\,\Gamma)$$

$$\frac{\Gamma.(x : A) \vdash e : B}{\Gamma \vdash \lambda x^{(A \xrightarrow{\sigma} B, \sigma)}\, e : A \xrightarrow{\sigma} B} \qquad \frac{\Gamma \vdash e_1 : A \xrightarrow{\sigma_1} B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash @^{(B,\sigma_2)}(e_1, e_2) : B} \quad (\sigma_1 \leq \sigma_2)$$

$$\frac{\emptyset}{\Gamma \vdash \mathtt{unit}^{(\mathtt{Unit},\sigma)} : \mathtt{Unit}}$$

$$\frac{\emptyset}{\Gamma \vdash l^{(\mathtt{Ref}^{\,\sigma}(A),\sigma)} : \mathtt{Ref}^{\,\sigma}(A)} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \mathtt{ref}^{\,(\mathtt{Ref}^{\,\sigma}(A),\sigma)}(e) : \mathtt{Ref}^{\,\sigma}(A)}$$

$$\frac{\Gamma \vdash e : \mathtt{Ref}^{\,\sigma_1}(A)}{\Gamma \vdash \mathtt{get}^{\,(A,\sigma_2)}(e) : A} \quad (\sigma_1 \leq \sigma_2) \qquad \frac{\Gamma \vdash e_1 : \mathtt{Ref}^{\,\sigma_1}(A) \quad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathtt{set}^{\,(\mathtt{Unit},\sigma_2)}(e_1, e_2) : \mathtt{Unit}} \quad (\sigma_1 \leq \sigma_2)$$

Table 9: Subtyping relation

$$\frac{\emptyset}{\mathtt{Unit} \leq \mathtt{Unit}}$$

$$\frac{A_2 \leq A_1 \quad B_1 \leq B_2}{A_1 \xrightarrow{\sigma_1} B_1 \leq A_2 \xrightarrow{\sigma_2} B_2} \quad (\sigma_1 \leq \sigma_2)$$

$$\frac{\emptyset}{\mathtt{Ref}^{\,\sigma_1}(A) \leq \mathtt{Ref}^{\,\sigma_2}(A)} \quad (\sigma_1 \leq \sigma_2)$$

- for the type constructor $\texttt{Ref}(-)$, the subtyping rule is invariant on the unique component.

Finally, it is easy to show that the subtyping relation is a partial order. In order to benefit from this relation in the type system, we add the following conversion rule:

$$\frac{\Gamma \vdash e \,:\, A}{\Gamma \vdash e \,:\, B} \quad (A \leq B)\,.$$

The type system satisfy standard properties. Note that typing uniqueness is replaced with typing minimality.

Proposition 27 (Properties of the annotated type system)
Let Γ be a typing environment. Consider a labeled term a of labeled type A in Γ :

$$\Gamma \vdash a \,:\, A\,.$$

Weakening lemma
Let us consider an environment Γ' extending Γ. Then the typing judgment $\Gamma' \vdash a \,:\, A$ is valid.
Basis lemma
Consider the environment $\Gamma_{|\mathbf{FV}(a)}$ equal to the restriction of Γ to the free variables of a. Then the typing judgment $\Gamma_{|\mathbf{FV}(a)} \vdash a \,:\, A$ is valid.
Typing minimality
If a is equal to a variable x, then $\Gamma(x)$ is the least type (with respect to the subtyping relation) of a in Γ. Otherwise, if a is labeled by (M, σ), then M is the least type of a in Γ.
Substitution lemma
Let x be a variable in the domain of Γ, and b a labeled term of type $\Gamma(x)$. Then the term $a[b/x]$ has type A in Γ :

$$\frac{\Gamma \vdash a \,:\, A \quad \Gamma \vdash b \,:\, \Gamma(x)}{\Gamma \vdash a[b/x] \,:\, A} \quad (x \in \operatorname{dom}\Gamma)\,.$$

Decomposition lemma
Let us suppose that a is equal to $E[r]$, where E is an evaluation context and r a redex. Then r is well-typed, and if we assign to the hole $-$ the least type of r, then E has type A.

Proof.
- Weakening and basis lemmas, typing minimality, substitution lemma

It is easy (and standard) by induction on the proof of the typing judgment $\Gamma \vdash a \,:\, A$.
- Decomposition lemma

It is straightforward by induction on E. □

We now define the operational semantics. In order to simplify, we do not modify the operational semantics defined by a reduction relation in Table 4 (p. 8). This choice corresponds to an accessibility interpretation: the security label for a destructor represents an access right, granted for $\top$ and denied for $\bot$, to sensitive resources, marked by $\top$. A more realistic approach would consider a code instrumentation: a secured destructor could first check the access permission, then according to the result, either realize the expected destruction, or raise an exception.
We now state the main property, which allows the static and the dynamic semantics to meet. A memory store s is said to be *well-typed* (in the annotated type system) if for all references l^m in $\operatorname{dom} s$, l^m is well-typed, which is equivalent to $m = (\texttt{Ref}^{\,\sigma}(A), \sigma)$ for some type A and some security label σ, and the value $s(l^m)$ has type A; a configuration (s, e) is said to be *well-typed* if s and e are.

Theorem 28 (Subject reduction: Type decreasing)
Let (s_1, e_1) be a well-typed configuration reducing to (s_2, e_2). Then (s_2, e_2) is well-typed and moreover the least type of e_2 is a subtype of the least type of e_1.

Proof. We proceed by induction on the proof of the reduction $(s_1, e_1) \to (s_2, e_2)$. We suppose that the configuration (s_1, e_1) is well-typed. Let us denote by M the least type of e_1.

• $[\beta]$

The reduction is

$$(s_1, @^m(\lambda x^n\, a, v)) \to (s_1, a[v/x])\,.$$

The proof of $\emptyset \vdash e_1 \,:\, M$ has the following form:

$$\dfrac{\dfrac{\dfrac{\emptyset.(x \,:\, A') \vdash a \,:\, M'}{\emptyset \vdash \lambda x^n\, a \,:\, A' \xrightarrow{\sigma'} M'}}{\begin{array}{c}\vdots\\ \emptyset \vdash \lambda x^n\, a \,:\, A \xrightarrow{\sigma} M\end{array}} \quad \begin{array}{c}\vdots\\ \emptyset \vdash v \,:\, A\end{array}}{\emptyset \vdash @^m(\lambda x^n\, a, v) \,:\, M}\,,$$

where $A \leq A'$, $M' \leq M$ and $\sigma' \leq \sigma$.

From these inequalities and the substitution lemma, we deduce that $a[v/x]$ has type M, and therefore its least type is a subtype of M. Moreover, $(s_1, a[v/x])$ is well-typed.

• [REF]

The reduction is

$$(s_1, \mathtt{ref}^{\,m}(v)) \to ((s_1, l^m \mapsto v), l^m)\,,$$

where l^m is equal to $\nu_m(s)$. The proof of $\emptyset \vdash \mathtt{ref}^{\,m}(v) \,:\, M$ has the following form:

$$\dfrac{\begin{array}{c}\vdots\\ \emptyset \vdash v \,:\, M_1\end{array}}{\emptyset \vdash \mathtt{ref}^{\,(\mathtt{Ref}^{\,\sigma}(M_1),\sigma)}(v) \,:\, \mathtt{Ref}^{\,\sigma}(M_1)}\,,$$

where $\mathtt{Ref}^{\,\sigma}(M_1) = M$ and $(\mathtt{Ref}^{\,\sigma}(M_1), \sigma) = m$. We deduce that $((s_1, l^m \mapsto v), l^m)$ is well-typed. Moreover, $\mathtt{ref}^{\,m}(v)$ and l^m have the same least type, M.

• [REF-!]

The reduction is

$$(s_1, \mathtt{get}^{\,m}(l^n)) \to (s_1, s_1(l^n))\,.$$

The configuration $(s_1, s_1(l^n))$ is obviously well-typed. We show that $s_1(l^n)$ has type M.
The proof of $\emptyset \vdash \mathtt{get}^{\,m}(l^n) \,:\, M$ has the following form:

$$\dfrac{\dfrac{\emptyset}{\emptyset \vdash l^n \,:\, \mathtt{Ref}^{\,\sigma'}(M)} \atop {\vdots \atop \emptyset \vdash l^n \,:\, \mathtt{Ref}^{\,\sigma}(M)}}{\emptyset \vdash \mathtt{get}^{\,m}(l^n) \,:\, M}\,,$$

where $\sigma' \leq \sigma$. We deduce that n is equal to $(\mathtt{Ref}^{\,\sigma'}(M), \sigma')$. Since s_1 is well-typed, $s_1(l^n)$ has type M.

• [REF-?]

The reduction is

$$(s_1, \mathtt{set}^{\,m}(l^n, v)) \to ((s_1, l^n \mapsto v), \mathtt{unit}^m)\,.$$

The proof of $\emptyset \vdash \mathtt{set}^{\,m}(l^n, v) \,:\, \mathtt{Unit}$ has the following form:

$$\dfrac{\dfrac{\dfrac{\emptyset}{\emptyset \vdash l^n \,:\, \mathtt{Ref}^{\,\sigma'}(B)}}{\begin{array}{c}\vdots\\ \emptyset \vdash l^n \,:\, \mathtt{Ref}^{\,\sigma}(B)\end{array}} \quad \begin{array}{c}\vdots\\ \emptyset \vdash v \,:\, B\end{array}}{\emptyset \vdash \mathtt{set}^{\,m}(l^n, v) \,:\, \mathtt{Unit}}\,,$$

where $\sigma' \leq \sigma$. We deduce that n is equal to $(\mathtt{Ref}^{\sigma'}(B), \sigma')$. Since $(s_1, l^n \mapsto v)(v)$, which is equal to v, has type B, the memory store $(s_1, l^n \mapsto v)$ is well-typed, and so is the configuration $((s_1, l^n \mapsto v), \mathtt{unit}^m)$.

• [RED]

The reduction is $(s_1, E[r_1]) \to (s_2, E[r_2])$, where E is an evaluation context different from the hole $-$ and r_1 a redex such that (s_1, r_1) reduces to (s_2, r_2).

By the decomposition lemma, we obtain that r_1 is well-typed and that E has type M if we assign to the hole $-$ the least type of r_1. By the inductive hypothesis applied to $(s_1, r_1) \to (s_2, r_2)$, since (s_1, r_1) is well-typed, we have that s_2 is well-typed and that the least type of r_2 is a subtype of r_1. From the substitution lemma, we deduce that $E[r_2]$ has type M, and we can conclude. □

This fundamental theorem implies that a program is secure if it is well-typed in the annotated type system.

Corollary 29 (Type soundness)
If a labeled program is well-typed in the annotated type system, then, during its execution, no destructor labeled by $\perp$ is applied to a value labeled by $\top$.

Proof. By the subject reduction theorem, every configuration in the execution trace is well-typed. Suppose that during the execution, a destructor labeled by $\perp$ is applied to a value labeled by $\top$. Then the corresponding redex is ill-typed, contradiction by the decomposition lemma. □

In other words, the static verification of typing entails the dynamic verification of the security policy asserting that only secured access functions are applied to sensitive resources.

We now show how to deduce a confinement criterion from the annotated type system. We first model the mobile program calling the local environment. Since the local environment is available on the host machine, it is assumed to be secure: after the sensitive local resources have been marked and access rights have been assigned to access functions, the local environment is assumed to receive a labeled type in the annotated type system. We denote by $\mathsf{L_S}$ the labeled local environment and by L_S its least labeled type. As above, we denote by L the unlabeled type obtained by erasing the labels in L_S. As for the mobile program, since it is not local by definition, it cannot contain sensitive local resources, and since it may be hostile, no access rights are assigned to its access functions: therefore, we only assume that the mobile program is an unlabeled program, $\lambda x : L.\, \mathsf{M}[x]$, of type $L \to M$ in the standard type system (without annotation); however, we can label the mobile program, provided the labeling only uses the security label $\perp$, since the mobile program is nor local, neither secured. More precisely, given an unlabeled type A, we denote by $\langle A \rangle^\perp$ the labeling of A such that each type constructor, $- \to -$ or $\mathtt{Ref}(-)$, is labeled by $\perp$. Given an unlabeled term a, we denote by $\langle\!\langle a \rangle\!\rangle^\perp$ the term a completely labeled by $\perp$, that is the labeled term equal to $G(\langle a \rangle^\perp)$, where G is inductively defined as follows (cf. p. 9 for the definition of $\langle a \rangle^\perp$):

$$\begin{aligned} G(x) &= x\,, \\ G(\mathtt{f}^{(A,\perp)}(e_j)_j) &= \mathtt{f}^{(\langle A \rangle^\perp,\perp)}(G(e_j))_j\,. \end{aligned}$$

It is now easy to show that the labeled mobile program

$$\langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^\perp$$

is well-typed (in the annotated type system). Since it is labeled by

$$(\langle L \rangle^\perp \xrightarrow{\perp} \langle M \rangle^\perp, \perp)\,,$$

its least type is $\langle L \rangle^\perp \xrightarrow{\perp} \langle M \rangle^\perp$. Finally, the mobile program calling the local environment is soundly modeled by

$$@^{(\langle M \rangle^\perp,\perp)}(\langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^\perp, \mathsf{L_S})\,.$$

The question becomes: is this program secure? By Corollary 29, it is secure if it is well-typed in the annotated type system. In that case, the typing proof has the following form:

$$\dfrac{\begin{array}{c}\vdots\\ \emptyset \vdash \langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^{\perp} \,:\, \langle L \rangle^{\perp} \xrightarrow{\perp} \langle M \rangle^{\perp}\\ \vdots\\ \emptyset \vdash \langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^{\perp} \,:\, K \xrightarrow{\perp} \langle M \rangle^{\perp}\end{array} \qquad \begin{array}{c}\vdots\\ \emptyset \vdash \mathsf{L_S} \,:\, L_S\\ \vdots\\ \emptyset \vdash \mathsf{L_S} \,:\, K\end{array}}{\emptyset \vdash @^{(\langle M \rangle^{\perp},\perp)}(\langle\!\langle \lambda x : L.\, \mathsf{M}[x] \rangle\!\rangle^{\perp}, \mathsf{L_S}) \,:\, \langle M \rangle^{\perp}}\ ,$$

where $L_S \leq K \leq \langle L \rangle^{\perp}$. We can easily conclude that the mobile program calling the local environment is well-typed if and only if the type L_S is a subtype of the type $\langle L \rangle^{\perp}$. The following proposition gives the corresponding characterization. Given a labeled type, we denote by $\downarrow (C)$ the unlabeled type obtained by erasing the labels.

Proposition 30 (Subtyping and outgoing types)
Let C be a labeled type. Then C is a subtype of $\langle \downarrow (C) \rangle^{\perp}$ if and only if every outgoing type of C different from `Unit` *is labeled by $\perp$.*

Recall that a type A is an outgoing type of C if it occurs either at a positive occurrence, or under a reference type constructor $\texttt{Ref}^{\,*}(-)$; likewise, a type A is an incoming type of C if it occurs either at a negative occurrence, or under a reference type constructor $\texttt{Ref}^{\,*}(-)$.

Proof. We show not only the property of the statement, but also a dual property. More precisely, we show the two following equivalences:

$$\begin{array}{lcl} C \leq \langle \downarrow (C) \rangle^{\perp} & \Leftrightarrow & \forall\, A^{\sigma} \neq \texttt{Unit}\,.\ A^{\sigma} \text{ outgoing type of } C \Rightarrow \sigma = \perp\,,\\ \langle \downarrow (C) \rangle^{\perp} \leq C & \Leftrightarrow & \forall\, A^{\sigma} \neq \texttt{Unit}\,.\ A^{\sigma} \text{ incoming type of } C \Rightarrow \sigma = \perp\,. \end{array}$$

The proof is straightforward by induction on C. □

How to secure the local environment?
Firstly, we instrument the local code by marking sensitive resources and assigning access rights to access functions, according to a given security policy; secondly, we verify the instrumentation by determining whether the labeled local environment is well-typed in the annotated type system; finally, by using Proposition 30, we determine whether the labeled type of the local environment is a subtype of a labeled type completely labeled by $\perp$: this is exactly our confinement criterion, as stated in Theorem 8 (p. 13). If the answer is positive, then any mobile program calling the local environment is secure.