A Confinement Criterion for Securely Executing Mobile Code

Hervé Grall

Journal of Automata, Languages and Combinatorics **u** (v) w, x–y

A CONFINEMENT CRITERION FOR SECURELY EXECUTING MOBILE CODE

Hervé Grall[1]
OBASCO (LINA), École des mines de Nantes
La Chantrerie, 4, rue Alfred Kastler, B.P. 20722, 44307 Nantes Cedex 3, France
e-mail: hgrall@emn.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 the accesses from mobile programs to local resources. The article deals with the following question: how can we 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 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 cannot be weakened by giving counter-examples for all the environment types that do not satisfy the criterion, and that it is pertinent by detailing the example of a specific security architecture. The main contribution of the article is the proof method, based on a language annotation that keeps track of code origin and that enables the study of the interaction frontier between the local code and the mobile code. The generalization of the method is finally discussed.

Keywords: typed programming languages, mobile code, language-based security, access controls, confinement

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 the control of the interactions between mobile programs and local resources, and particularly the 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

[1] This work was undertaken at CERMICS (ENPC – INRIA), école nationale des ponts et chaussées, France.

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, providing a fine-grained access control [9, 10].
A security architecture provides some means, first, of defining a security policy, second, of enforcing a given security policy. A security policy is simply a set of security rules satisfied during executions. The *confinement criterion* that we present in the article provides a useful technique for enforcing a security policy based on access control in the presence of mobile code, as the following introductory example explains.

Consider Program 1, which describes a class written in OCaml [25, 24], an implementation of an object-oriented extension of ML. Any instance of the class `resource`

```
(* class parameterized by origin *)
class resource (origin:string) =
  object
    (* method with one parameter *)
    method access (subject:string) =
      (* print to the standard output a message
       * ^ is string concatenation *)
      print_string (subject ^ " accesses "
                    ^ origin ^ " resource\n")
  end
```

Program 1: Class for resources

represents a resource. Like any resource, an instance has at least one access function, the method `access`, which prints a message to the standard output. A local resource is created as follows:

```
let res = new resource "local" ,
```

where the argument of the class `resource` gives the origin of the resource. If the mobile program can directly call the method `access` of the local resource, then it can print to the standard output the message

```
"hostile applet accesses local resource"
```

by passing "hostile applet" as argument, as follows:

```
res#access "hostile applet" (* # is method invocation *) .
```

Suppose that the accesses to local resources need to be controlled. For instance, suppose that the security policy requires that the adverb "securely" is added to every message printed to the standard output after a call to the method `access` of a local resource. The adverb symbolically represents the security checks needed to control the access to a local resource: a practical example would check the identity of the caller, its rights to the resource, the value of the arguments or the result of the access function. There are two main techniques to implement these checks in the local environment.

The first solution modifies the class `resource` by redefining the access function: the method `access` encapsulated in any instance of the class `resource` is replaced with

a secured one, which makes the security checks. The local environment can then provide a direct reference to a local resource without any danger. If we represent the environment by an OCaml module, we get Program 2. Whenever some code calls

```
module Env1 =
  struct
    (* secured class *)
    class resource (origin:string) =
      object
        (* secured method *)
        method access (subject:string) =
          print_string (subject ^ " securely accesses "
                        ^ origin ^ " resource\n")
      end
    (* direct access *)
    let secured_res = new resource "local"
  end
```

Program 2: First solution — Secured redefinition

the access function of the local resource with `Env1.secured_res#access` "subject", the following message is printed:

```
"subject securely accesses local resource".
```

This solution therefore satisfies the security requirement, but is an invasive technique.

On the contrary, the second solution does not modify the class `resource`, but checks its uses. First, in the local environment, every access needs to be instrumented to perform the security checks. Second, the environment can no longer provide a direct access to a local resource, which is not protected. Instead, it provides as a service a proxy, or surrogate, which replies to access requests by making the security checks and calling the access function encapsulated in the local resource. In our example with the class `resource`, a new class `proxy` is added to the local environment. It has a unique method, `request`, which is instrumented by using the function `check`. The instance `controller` of the class `proxy` is defined as the proxy of the local resource `confined_res`. The second solution leads to the local environment described in Program 3. It also satisfies the security requirement. Indeed, the unique call to the method `access` in the local environment is instrumented. Moreover, the mobile program cannot directly access the local resource `confined_res`. It can only access the resource via the proxy `controller`, with the call `Env2.controller#request` "hostile applet", which prints the following message:

```
"securely hostile applet accesses local resource".
```

Now, suppose that we add to the environment `Env2` a definition `danger`, with type `t`, as in Program 4. Does the instrumentation of `danger`'s definition suffice to satisfy the security policy? Clearly, we can argue that the answer depends on the type `t` of `danger`. Indeed, this type specifies the ability of the mobile program to directly access

```
module Env2 =
  struct
    (* unsecured class *)
    class resource (origin:string) =
      object
        method access (subject:string) =
          print_string (subject ^ " accesses "
                        ^ origin ^ " resource\n")
      end
    (* instrumentation function *)
    let check (res:resource) =
      function (subject:string) ->
        print_string "securely "; res#access subject
    (* secured proxy definition *)
    class proxy (res:resource) =
      object
        (* instrumented method *)
        method request (subject:string) =
          (* instrumentation of res#access subject *)
          check res subject
      end
    (* indirect access via a proxy *)
    let controller =
      let confined_res = new resource "local" in
      new proxy confined_res
  end
```

Program 3: Second solution — Secured instrumentation and proxy

```
(* environment Env2 extended *)
module Env3 =
  struct
    ...
    let danger = ...
  end
```

Program 4: Problematic environment

a local resource. For example, if `t` is equal to `resource`, then the instrumentation cannot suffice, since the local resource returned by `danger` is directly accessible from mobile code. On the contrary, if `resource` does not occur in `t`, then the instrumentation seems to suffice. Likewise, if the type `t` is the type `resource -> s` of the functions from `resource` to `s`, where `resource` does not occur in `s`. However if the type `t` is the functional type `resource ref -> s`, the environment may be unsecure for some definition of `danger`, in spite of its instrumentation. For instance, suppose that the function

`danger` and the mobile program are defined as in Program 5. The mobile program

```
(* unsecure environment *)
module Env3 =
  struct
    ...
    let danger = function (channel:resource ref) ->
      let accessible_res = new resource "local" in
      channel := accessible_res
  end
let mobile_program =
let access_channel = ref (new Env3.resource "mobile") in
Env3.danger access_channel;
!access_channel#access "hostile applet"
```

Program 5: Confinement problem

directly accesses `accessible_res` by using as a communication channel the reference passed as argument to `danger`, and succeeds in printing the following message:

```
"hostile applet accesses local resource",
```

which breaks the security policy. This is an instructive case where the local resource is not confined in the local environment.

The preceding example makes our quest precise: we are looking for a *confinement criterion* that ensures that if the local environment satisfies it, then no mobile program can directly access a local sensitive resource. It also shows that a confinement criterion is just an element of a security architecture. Hence, we begin with a review of current security architectures for controlling accesses, without considering the mobile code issue.

Language-based Security Architectures for Access Control We restrict ourselves to the approach called *language-based security*, which is particularly relevant to computer-security questions, as Harper et al. have argued [12]. In this approach, a security architecture generally adds new features to the programming language, enabling the definition of a security policy. It also provides the technical means of enforcing a given security policy: the means are implemented either at a syntactic level by a program transformation, for instance a code instrumentation that directly inserts checks in programs, or at the semantic level by a static analysis or an instrumentation of the (operational) semantics.

How are accesses represented in this approach? An *access* involves a *subject* accessing, a *resource* accessed and an *access function*.
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") [13] or in the POP system (acronym for "Programming with Object Protection") [29]. A dynamic point

of view considers not only the current subject, but also its callers, which are obtained by inspecting the call stack: see the original work of Wallach [34], the official implementation for Java [9, 10], Schneider and Erlingsson's alternative [7], Pottier et al.'s analysis [22], which replaces dynamic checks with static ones, implemented by a type system, and the semantic theory of stack inspection developed by Fournet and Gordon [8].
A resource usually corresponds to the source of an input or the target of an output. More generally, we consider that a resource is represented by a value of the programming language, whereas an access function is represented by any operation applicable to a resource. Thus, an access is just the call to an access function. In a language with references to structured data, like objects, any reference can therefore represent a resource. The operations applicable to a reference are typically content update, content selection, or method invocation. In a language with functions, any function can represent a resource. The unique operation applicable to a function is application.

Once accesses are defined, a security architecture for controlling accesses can be built. It provides the means of specifying a security policy, a set of rules that all accesses must obey during executions. The simplest form of a policy is an access control matrix, which assigns to each subject and each resource a set of authorized access functions. Since Schneider's work [28], a more sophisticated form has become usual to specify a security policy for controlling accesses: a security automaton describes how to reply to all accesses during a program execution. A security architecture also provides some means of enforcing a given security policy. For example, for a policy specified by a security automaton, different solutions have been designed, in order to make the security automaton monitor the execution. A straightforward solution is to execute the automaton in parallel with the program: when an access is to be executed, the program notifies the automaton, which then makes the corresponding transition and replies to the program according to the result of the transition. Other solutions aim at reducing the overhead of a parallel execution. They all resort to program transformations and static analyses: see Colcombet and Fradet's solution [6], Walker's [33] and Thiemann's [30].

Mobile Code Issue We now come to our specific question: how can we enforce access control in the presence of mobile code?
We assume that the means provided by the security architecture can be applied to the local environment: after a security policy has been defined to protect local resources, the local code can be fully secured 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 securing 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 checked in order to certify it according to the security policy of the host system. Modularity is required to proceed as follows: first, the local environment is checked to determine its secure calling contexts, second, the mobile program is certified by checking that the calls to the local environment are secure. Thus, Jensen et al. [3] add modularity to an analysis for whole programs [4]. When the host system receives

a certified mobile program, it just verifies the correctness of the certification. This verification may 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. Thus, Lee and Necula [19] propose to use "proof-carrying code", that is to say to add proof hints to the mobile program, in order to facilitate the proof task.

It remains that, with program certification, the mobile program can no longer 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. From the introductory example, we can distinguish two defensive techniques.
The first one is based on *encapsulation*. Access functions are replaced with secured ones, and since they are encapsulated in local resources, each time an access function is called, a secured one is actually called. The technique corresponds to our first solution (see Program 2), where the method `access` of the class `Env1.resource` has been secured. The same solution is used in the current implementation of access control in the standard application programming interface of the language Java. The second technique is based on *confinement*: access control is checked 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. The technique corresponds to our second solution (see Program 3), where in the method `request` of the class `Env2.proxy`, the call to the method `access` has been instrumented, and the local resource `confined_res` is only accessible from its proxy `controller`.
The confinement technique has to be chosen when the secured access function cannot be encapsulated in the resource, for instance, when the code of the resource is not available. Performance requirements may also justify this solution: indeed, the first technique based on encapsulation implies that each call to an access function entails a security check, whereas the second technique based on confinement requires security checks only when they are needed. The encapsulation technique may be chosen for expressivity reasons: indeed, the access function can be fully redefined, as for the method `access` in `Env1.resource`, whereas with the confinement technique, only the arguments and the result of the access function can be checked, as for the instrumentation with the function `Env2.check` in the class `Env2.proxy`.
A practical example for the confinement technique is given by Bokowski and Vitek [31], who advocate this solution for Java. They propose to confine types, that is to say to confine all the values of sensitive types. Palsberg et al. [20] give a formalization of their proposal for a Java-like language, leading to a type-based confinement criterion. We also give a type-based confinement criterion: it is more precise than theirs, to some extent, since it is only valid for a simpler language, a functional language with references. Our work actually extends Leroy and Rouaix's results [16], where code instrumentation is used in conjunction with confinement or type abstraction to control accesses. They have proved the validity of a confinement criterion that justifies our first assertion in the example of `danger`'s definition (see Program 4): if the type `resource` does not occur in the type of `danger`, then any local resource is confined in `danger`. We will prove what they have conjectured: it suffices that the type `resource` does not occur in `danger`'s type either at a positive occurrence, or under the reference type constructor **ref**. We will also prove that this criterion cannot be weakened: if the

type `resource` occurs in `danger`'s type at a dangerous place, then there effectively exist a definition for `danger` and a mobile program such that the mobile program directly accesses a local resource in `danger`. In Program 5, we have given an example of this general result for the type `resource ref -> unit`, where `resource` occurs under the type constructor **ref**.
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. The confinement criterion is valid for a high-level language, and ensures the following property: for each local environment satisfying the confinement criterion, for every mobile program executing in the local environment, the mobile code cannot directly access a local sensitive resource. 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. According to Abadi [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 low-level languages instead of high-level ones in a future work. Note that a trend is now emerging, which advocates the use of structured low-level languages, suitable for verification: see for example the "typed assembly language" [18] or the translation of Java "bytecode" in a typed term calculus [14]. Otherwise, the preservation by compilation of contextual properties can 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 Language Annotation", we define the programming language with which we work and the general technique that we use. The language corresponds to a simply typed λ-calculus enriched with references in the style of ML. In order to keep track of code origin during executions, we annotate the language, as described by Klop et al. [5, sect. 4]. We formally define the annotated language and give its static and dynamic semantics as well as some useful semantic properties.

Section 3, called "Frontier under Control: The Confinement Criterion", is an application to confinement of this annotation technique. The technique allows the confinement property to be formally defined and the validity of the confinement criterion to be proved, which solves the conjecture stated by Leroy and Rouaix [16, sect. 5.1, p. 162]. In Section 4, we also prove that this criterion cannot be weakened. In other words, if the type of the local environment does not satisfy the confinement criterion, we can effectively define a local environment and a mobile program such that the mobile program can directly access a local resource by calling the local environment.

Section 5, called "Confinement Criterion in Action: Example of Access Qualifiers", considers the relationship of the criterion with a standard security architecture. It details the example of the SLam-calculus [13], which provides a paradigmatic model

for a language with access qualifiers and with a type-based analysis for controlling accesses.

To conclude, after a brief comparison with Vitek et al.'s results [31, 20] and Leroy and Rouaix's [16], we will outline some directions for future work. It is particularly interesting to consider some extensions of the programming language that we use, like data abstraction or parametric polymorphism. Other interesting directions are to generalize the method to security architectures dealing with access control, and the mobile code issue to other security properties, like confidentiality, which is related to information flows.

2. Keeping Track of Code Origin by Language Annotation

We work with a functional language extended in order to manipulate 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). Although it is a very

$$
\begin{array}{rcll}
A & ::= & \texttt{Unit} & \text{(singleton type)} \\
& | & A \to A & \text{(functional type)} \\
& | & \texttt{Ref}(A) & \text{(reference type)} \\
\\
a & ::= & x & \text{(variable)} \\
& | & \lambda x : A.\, a & \text{(abstraction)} \\
& | & a_1\, a_2 & \text{(application of } a_1 \text{ to } a_2\text{)} \\
& | & \texttt{unit} & \text{(unit)} \\
& | & l^{\texttt{Ref}(A)} & \text{(store location } l \text{ with content of type } A\text{)} \\
& | & \texttt{ref}(a) & \text{(reference creation)} \\
& | & \texttt{get}(a) & \text{(dereferencing)} \\
& | & \texttt{set}(a_1, a_2) & \text{(assignment of } a_2 \text{ to } a_1\text{)}
\end{array}
$$

Table 1: Syntax of the programming language

simple language, it is sufficient to model 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 easily encoded (by using references). In order to keep track of code origin during executions, we resort to an annotation of the language: each operator occurring in a program becomes labeled and its label is preserved during the execution. A label is an ordered pair, whose first component is a type, called the *type label*, and whose second component is a piece of information, called the *information label*. The syntax of the annotated language is given in Table 2. In the following, `f` and `g` stand for operators in $\{\lambda x, \texttt{app}, \texttt{unit}, l, \texttt{ref}, \texttt{get}, \texttt{set}\}$: an annotated term e either is a variable, or has the form $\texttt{f}^m(e_1, \ldots, e_i)$, where $e_1, \ldots, e_i$ are the immediate annotated subterms

$$
\begin{array}{rcl}
m & ::= & (A, \iota) \quad \text{(type and information)} \\
\\
e & ::= & x \quad \text{(variable)} \\
& | & \lambda x^m\, e \quad \text{(labeled abstraction)} \\
& | & \mathtt{app}^m\,(e_1, e_2) \quad \text{(labeled application of } e_1 \text{ to } e_2) \\
& | & \mathtt{unit}^m \quad \text{(labeled unit)} \\
& | & l^m \quad \text{(labeled store location } l) \\
& | & \mathtt{ref}^m(e) \quad \text{(labeled reference creation)} \\
& | & \mathtt{get}^m(e) \quad \text{(labeled dereferencing)} \\
& | & \mathtt{set}^m(e_1, e_2) \quad \text{(labeled assignment of } e_2 \text{ to } e_1)
\end{array}
$$

Table 2: Syntax of the annotated language

$(0 \leq i \leq 2)$. 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; we also say that e is labeled with m[2]. Given an annotated term e, the set of free variables in e is denoted by $\mathrm{FV}(e)$, and the set of labeled locations in e is denoted by $\mathrm{Ref}(e)$. A *closed term* is a term without free variables, whereas a *program* is a closed term without store locations.
Note the choice that we make to build terms: variables are not labeled. We also consider a specific formation rule for terms: store locations are uniformly labeled, which means that given a location l, if l^m and l^n occur in a term, then $m = n$. 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 easily deduced from the annotated version. The *type system*, which is given in Table 3, distinguishes functions and references. A typing judgment has the form $\Gamma \vdash e : A$, which means that in the typing environment Γ, the annotated term e has type A.
Note the following points:

- since store locations are labeled with their type, typing environments only deal with variables, which are not labeled;
- in a given typing environment, a well-typed term receives a unique type;
- for each valid judgment $\Gamma \vdash e : A$, where e is not a variable, the type A is the type label of e;
- information labels do not matter in typing rules.

[2]Note the difference: below, we say that e is *entirely annotated* by ι when *every* operator of e has ι as information label.

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

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

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

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

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

Table 3: Type system

The operational semantics is defined as 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 (in Felleisen's style [35]). Since the language contains references, with side effects, we use the standard *weak call-by-value reduction strategy*: the leftmost call-by-value redex that does not appear within a λ-abstraction reduces. *Substitutions* are defined as usual: if e and e' are terms and x a variable, then $e'[e/x]$ stands for the result of replacing in e' the free occurrences of x with e, without variable capture.

A *value* is either a closed λ-abstraction, the unit value, or a store location:

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

where $\mathrm{FV}(\lambda x^m\, e) = \emptyset$.

A *redex* is either a β-redex, a reference creation, a dereferencing or an assignment:

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

where $\mathrm{FV}(\lambda x^m\, e) = \emptyset$.

An *evaluation context* is built around the hole $-$ as follows:

$$\begin{aligned} E ::= {} & - \\ \mid {} & \mathtt{app}^m\,(E, e) \mid \mathtt{app}^m\,(\lambda x^n\, b, E) \\ \mid {} & \mathtt{ref}^{\,m}(E) \mid \mathtt{get}^{\,m}(E) \mid \mathtt{set}^{\,m}(E, e) \mid \mathtt{set}^{\,m}(l^n, E)\,, \end{aligned}$$

where $\mathrm{FV}(\lambda x^n\, b) = \emptyset$ and $\mathrm{FV}(e) = \emptyset$.

Every well-typed closed term is equal to either a value, or a redex in an evaluation context, and this decomposition is unique.

A *memory store* s is a partial function from the set of labeled locations to the set of values such that:

(i) the locations in $\operatorname{dom} s$ are uniformly labeled, in other words, if l^m and l^n belong to $\operatorname{dom} s$, then $m = n$,

(ii) the set $\{l \mid \exists\, m\,.\ l^m \in \operatorname{dom} s\}$ is a segment of the set of store locations, 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. With the second condition, we can define a function for *creating* references: given a label m, the function ν_m maps a store s to l^m, where the store location l is the successor of the greatest location in the set $\{k \mid \exists\, m\,.\ k^m \in \operatorname{dom} s\}$. We also need to define a function for *updating* or *extending* a store. Suppose that s is a memory store and l^m a labeled location such that l^m is either already created, that is in the domain of s, or new, that is equal to $\nu_m(s)$. Given a value v, the store $(s, l^m \mapsto v)$ updates or extends s by mapping l^m to v, and is formally 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) $\operatorname{Ref}(e) \subseteq \operatorname{dom} s\,,$

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

The two conditions mean that every labeled location 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 $(s, e) \to (s', e')$, which means that the configuration (s, e) reduces to (s', e'). We observe that the reduction rules preserve the label of each operator along reductions.
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 location l^m in the domain of s, l^m is well-typed, which is equivalent to $m = (\mathtt{Ref}(A), \iota)$ 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.

We can now define the execution trace of each well-typed program e: it is the maximal sequence consisting of the configurations obtained by reduction from the initial

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

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

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

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

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

Table 4: Operational semantics — Reduction relation

configuration $(\emptyset, e)$, where $\emptyset$ is the store with an empty domain. Note that each configuration in the trace is well-typed, by Subject Reduction. A well-typed program either evaluates to a configuration whose control term is a value, or diverges.

Now, we study the relationship between the annotated language and the standard one. Table 5 inductively defines a stripping function, written $\downarrow$, which erases labels in an annotated term. The following propositions give useful properties of the strip

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

Table 5: Strip function

function.

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

Thanks to the second part of the proposition and to the fact that information labels do not matter for typing, we can define from an information label ι and a standard term a of type A an annotated term denoted by $\langle a \rangle^\iota$, such that $\langle a \rangle^\iota$ has type A, $\downarrow (\langle a \rangle^\iota) = a$

and $\langle a \rangle^{\iota}$ is entirely annotated by ι, which means that every operator of $\langle a \rangle^{\iota}$ has ι as information label. The strip function $\downarrow$ naturally extends to configurations and to execution traces. The extension satisfies the following property.

Proposition 4 (Strip function — Trace simulation)
Consider the set of the execution traces of the well-typed annotated programs. Its image by the strip function $\downarrow$ is exactly the set of the execution traces of the well-typed standard programs.

In other words, the execution trace of an annotated program simulates the execution of the standard program that is associated, while keeping track of code origin with labels.

3. Frontier under Control: The Confinement Criterion

In this section, we give the confinement criterion and prove its validity by applying the annotation technique. 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 $\{\mathsf{mo}, \mathsf{lo}\} = \{\iota, \bar{\iota}\}$.

We begin by modeling the execution of a mobile program in the local environment. The local environment is simply a well-typed program L of type T, whereas the mobile program is modeled as a well-typed functional abstraction $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$, which is applied to the local environment. The local environment and the mobile program are then annotated according to their origin, local or not, in order to study confinement by executing the annotated program. More precisely, first, the local environment L is entirely annotated with lo, whereas the mobile program is entirely annotated with mo. Second, we study the execution of the well-typed annotated program

$$\mathsf{app}^{(\mathsf{R},\mathsf{mo})}\left(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}}\right),$$

which is represented by the tree in Figure 1.

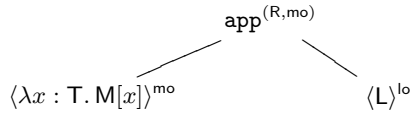

Figure 1: Modeling the mobile program calling the local environment

Recall from the introduction that an access is represented by a redex, either a β-redex, a dereferencing operation or an assignment operation. A rigorous formalization of confinement requires *frontier redexes* to be defined.

Definition 5 (Frontier redex) *A* frontier redex *is a redex such that there exists* ι *in* $\{\mathsf{mo}, \mathsf{lo}\}$ *such that*

(i) the resource accessed has ι *as origin label,*

(ii) the access operator, `app`*,* `get` *or* `set`*, has* $\bar{\iota}$ *as origin label.*

Formally, all the frontier redexes are described in Table 6, where the definitions are parameterized by an origin label ι *belonging to* $\{\mathsf{mo}, \mathsf{lo}\}$*.*

Frontier Redex	Resource accessed
$\mathtt{app}^{(B,\bar{\iota})}\,(\lambda x^{(A\to B,\iota)}\,e, v)$	$\lambda x^{(A\to B,\iota)}\,e$
$\mathtt{get}^{\,(A,\bar{\iota})}(l^{(\mathtt{Ref}(A),\iota)})$	$l^{(\mathtt{Ref}(A),\iota)}$
$\mathtt{set}^{\,(\mathtt{Unit},\bar{\iota})}(l^{(\mathtt{Ref}(A),\iota)}, v)$	$l^{(\mathtt{Ref}(A),\iota)}$

Table 6: Frontier redexes

A frontier redex therefore corresponds to a special access: the call comes from a certain origin whereas the resource accessed comes from the other origin. Since a resource type is confined in the local environment if no resource of this type coming from the local environment can be accessed by mobile code, confinement actually means that some frontier redexes must not reduce.

Definition 6 (Confinement)
The resource type A *is* confined *in the local environment* L *of type* T *if, for every mobile program* $\lambda x : \mathsf{T}.\,\mathsf{M}[x]$ *of type* $\mathsf{T} \to \mathsf{R}$*, no frontier redex accesses a resource of the type* A *and of the origin label* lo *during the execution of the annotated program*

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

We now come to the core of the section. The most accurate 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 T for the local environment and a resource type A, determine whether for every local environment L of type T, the type A is confined in L. Its proof is based on an accurate analysis of what happens at the frontier between the mobile code and the local code during an execution.
Indeed, 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 T 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 $\lambda g\, h\, (g\, v)$, where h is a λ-abstraction, g a functional variable and v 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. We can actually distinguish two kinds of edges at the frontier, according

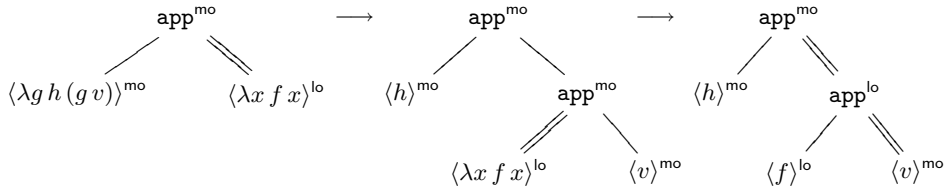

Figure 2: Mobile code execution and frontier — Example

to the origin label of the lower node. If the lower node is labeled with (A, lo), we say that A occurs at the outgoing frontier (with respect to the local environment), since a resource labeled with (A, lo) goes out from the local environment when an access function coming from the mobile program is applied to it. Dually, if the lower node is labeled with (A, mo), we say that A occurs at the incoming frontier, since a resource labeled with (A, mo) comes in the local environment when an access function coming from the local environment is applied to it. If we determine from the type T an upper bound of the set of the types occurring at the outgoing frontier during any execution, then we can easily obtain a confinement criterion. If a resource type A does not belong to the upper bound, then A does not belong to the outgoing frontier during any execution, hence no frontier redex accessing a local resource of type A reduces. In other words, the type A is confined in every local environment of type T.

In the following, we formalize the preceding argument leading to a confinement criterion.

After the definition of frontier redexes, which are redexes occurring at the frontier, we now define frontiers. Besides the formal definition, we give an intuitive representation of frontiers, which helps with the reasoning.

As the example in Figure 2 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 store location is thus followed by its content (as a tree).

$$(s, e) \stackrel{def}{=} \begin{pmatrix} l_1 & \cdots & l_n & e \\ | & & | & \\ s(l_1) & & s(l_n) & \end{pmatrix}$$

Figure 3: Configuration as a forest

Now, given an origin label ι, the type A belongs to the ι-*frontier* of the configuration (s, e) if there exists in the forest representation of (s, e) an edge of the form described in Figure 4.

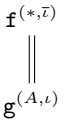

Figure 4: A at the ι-frontier

This intuitive definition leads to the following formal definition.

Definition 7 (Frontier)
Let ι be any origin label. Consider a configuration (s, e).
The ι-frontiers *of the memory store s, denoted by $\mathcal{F}_\iota(s)$, and of the control term e, denoted by $\mathcal{F}_\iota(e)$, are inductively generated by the inference system described in Table 7. The* ι-frontier *of the configuration (s, e), denoted by $\mathcal{F}_\iota(s, e)$, is equal to the union $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(e)$ of the ι-frontiers of s and e.*
The lo*-frontier and the* mo*-frontier are respectively called the* outgoing frontier *and the* incoming frontier.

$$\frac{\emptyset}{A \in \mathcal{F}_\iota(s)} \quad (\exists\, l, v\,.\ s(l^{(\mathtt{Ref}(A),\overline{\iota})}) = v^{(A,\iota)})$$

$$\frac{A \in \mathcal{F}_\iota(s(l^m))}{A \in \mathcal{F}_\iota(s)} \quad (l^m \in \operatorname{dom} s)$$

$$\frac{\emptyset}{A \in \mathcal{F}_\iota(e)} \quad (\exists\, \mathtt{f}, B, e_k\,.\ e = \mathtt{f}^{(B,\overline{\iota})}(\ldots, e_k^{(A,\iota)}, \ldots))$$

$$\frac{A \in \mathcal{F}_\iota(e_k)}{A \in \mathcal{F}_\iota(e)} \quad (\exists\, \mathtt{f}, m, e_k\,.\ e = \mathtt{f}^m(\ldots, e_k, \ldots))$$

Table 7: Frontier — Inductive definition

We now deal with frontier boundedness. Actually, although the preceding argument leading to the confinement criterion does not refer to the incoming frontier, frontier boundedness needs to apply to both frontiers, since during an execution, types at the outgoing frontier may generate types at the incoming frontier, and conversely, as the example in Figure 2 shows. Consider the type T for the local environment. We are going to determine from T two sets of types, $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{A}_{\mathsf{mo}}(\mathsf{T})$, which will be upper bounds of the outgoing and incoming frontiers resulting from executing mobile code:

more precisely, given any local environment L of type T and any mobile program $\lambda x : \mathsf{T}.\,\mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$, for every configuration (s, e) in the execution trace of

$$\mathtt{app}^{(\mathsf{R},\mathsf{mo})}\left(\langle \lambda x : \mathsf{T}.\,\mathsf{M}[x]\rangle^{\mathsf{mo}}, \langle \mathsf{L}\rangle^{\mathsf{lo}}\right),$$

the outgoing frontier of (s, e) will be included in $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and the incoming frontier of (s, e) will be included in $\mathcal{A}_{\mathsf{mo}}(\mathsf{T})$.
We begin by giving some stability properties for these upper bounds. First, note that T, initially at the lo-frontier, must belong to $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$. Second, the following examples describe the cases that generate the most frontiers. Let ι be an origin label.
Suppose that $A \to B$ belongs to $\mathcal{A}_\iota(\mathsf{T})$. Consider a well-typed configuration

$$\left(\emptyset, \langle E\rangle^{\bar\iota}[\mathtt{app}^{(B,\bar\iota)}\left(\langle \lambda x : A.\, b\rangle^{\iota}, \langle v\rangle^{\bar\iota}\right)]\right),$$

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

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

which contains A at the $\bar\iota$-frontier if b has 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 5, 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 8 (p. 19) are therefore valid.

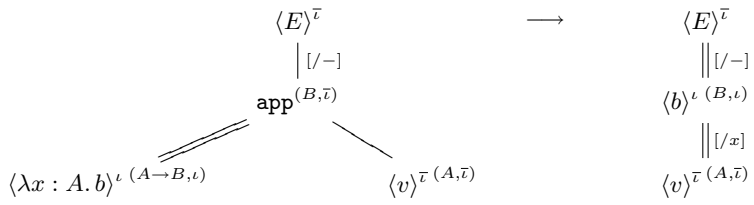

Figure 5: Frontier and β-reduction

Suppose that $\mathtt{Ref}(A)$ belongs to $\mathcal{A}_\iota(\mathsf{T})$. Let $l^{(\mathtt{Ref}(A),\iota)}$ be a location of type $\mathtt{Ref}(A)$ and of origin ι.
Consider a well-typed configuration

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

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

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

which contains A at the ι-frontier if E is different from the hole $-$: see Figure 6.
Consider a well-typed configuration

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

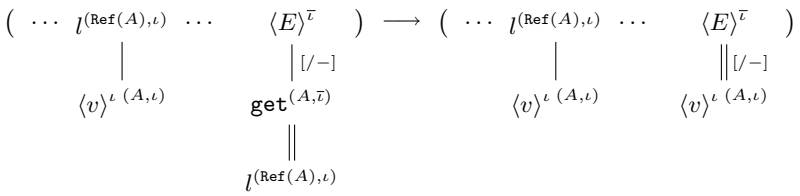

Figure 6: Frontier and dereferencing

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

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

which contains A at the $\bar{\iota}$-frontier ($\langle v\rangle^{\bar{\iota}}$ being under $l^{(\mathtt{Ref}(A),\iota)}$ in the updated store): see Figure 7.

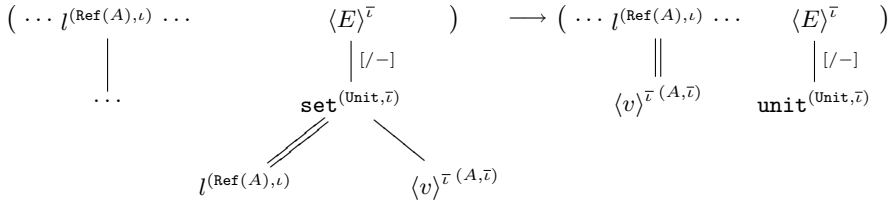

Figure 7: Frontier and assignment

The inference rules $[\mathtt{Ref}(+)]$ and $[\mathtt{Ref}(-)]$ in Table 8 are therefore valid.

$$\frac{\emptyset}{\mathsf{T} \in \mathcal{A}_{\mathsf{lo}}(\mathsf{T})}$$

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

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

Table 8: Frontier upper bounds — Inductive definition

Natural candidates for the two upper bounds $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{A}_{\mathsf{mo}}(\mathsf{T})$ are the sets inductively generated by the inference system in Table 8, which easily gives:

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

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

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

Definition 8 (Outgoing and incoming types)
Let T be a type. Let $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{A}_{\mathsf{mo}}(\mathsf{T})$ be the sets inductively generated by the inference system described in Table 8. We say that

(i) a type A is an outgoing type *of T if A belongs to $\mathcal{A}_{\mathsf{lo}}(\mathsf{T})$,*

(ii) a type A is an incoming type *of T if A belongs to $\mathcal{A}_{\mathsf{mo}}(\mathsf{T})$.*

The intuition behind this definition is right: a type that is not an outgoing type of T is confined in T, as the following theorem precisely shows, thus solving the conjecture stated by Leroy and Rouaix [16, sect. 5.1, p. 162].

Theorem 9 (Confinement criterion)
Let T be the type for the local environment, and A a resource type. If the type A is not an outgoing type of T, then for every local environment L of type T, the type A is confined in L.

The theorem is an immediate corollary of the following proposition, if we can prove its premise asserting the boundedness of the outgoing frontier.

Proposition 10 (Frontier boundedness implies confinement)
Let T be the type for the local environment, and A a resource type. Suppose that, for any local environment L of type T, for any mobile program $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$, for each configuration (s, e) in the execution trace of the annotated program $\mathsf{app}^{(\mathsf{R},\mathsf{mo})}\,(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})$, the outgoing frontier of (s, e) is included in the set of the outgoing types of T:

$$\mathcal{F}_{\mathsf{lo}}(s, e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})\,.$$

Then, if the type A is not an outgoing type of T, for every local environment L of type T, the type A is confined in L.

Proof. Suppose that A is not an outgoing type of T. Let L be any local environment of type T, and $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ any mobile program of type $\mathsf{T} \to \mathsf{R}$. Suppose that some frontier redex accesses a resource of the type A and of the origin label lo during the execution of the annotated program $\mathsf{app}^{(\mathsf{R},\mathsf{mo})}\,(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})$. Then, by definition of a frontier redex, A occurs at the outgoing frontier of some configuration (s, e) in the execution trace. Since by hypothesis $\mathcal{F}_{\mathsf{lo}}(s, e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})$, we deduce that A is an outgoing type of T: this is a contradiction. A is therefore confined in L. □

We now prove the premise in Proposition 10, which asserts the boundedness of the outgoing frontier. Actually, we also prove the boundedness of the incoming frontier, as said above.
We first show the preservation of frontier boundedness by reduction. 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. The example described in Figure 8 shows why this hypothesis is required. Initially, the incoming frontier,

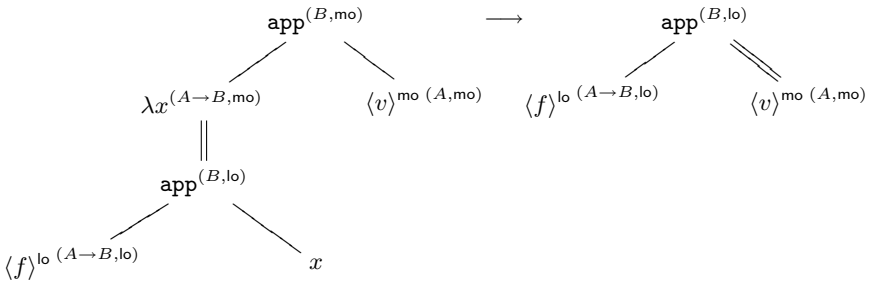

Figure 8: Origin incoherence — Example

empty, and the outgoing frontier, equal to $\{B\}$, are respectively included in the set of the incoming types of B and in the set of the 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 not be an incoming type of B: there is no preservation.
We begin by formally defining the property of origin coherence and then show its preservation by reduction.

Definition 11 (Origin coherence)
An annotated term e is origin coherent *if it belongs to the set inductively generated by the inference system in Table 9.*
A configuration (s, e) is origin coherent *if*

(i) the control term e is origin coherent,
(ii) for every labeled location 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 that is origin coherent, all the operators between its root and an occurrence of any free variable have the same origin, likewise all the 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

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

is not origin coherent, whereas the terms

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

$$\frac{\emptyset}{x} \quad (x \text{ variable})$$

$$\frac{e_1 \dots e_j}{\mathtt{f}^{(A,\iota)}(e_1,\dots,e_j)} \quad \left(\begin{array}{l} \forall\, k \in \{1,\dots,j\}\,. \\ \quad (e_k = \mathtt{g}^{(A_k,\iota_k)}(\dots) \wedge \mathrm{FV}(e_k) \neq \emptyset) \\ \quad \Rightarrow \iota = \iota_k \end{array} \right)$$

Table 9: Origin coherence — Inductive definition

and

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

are (see Figure 9, where the operators with a wrong origin label are boxed). We now

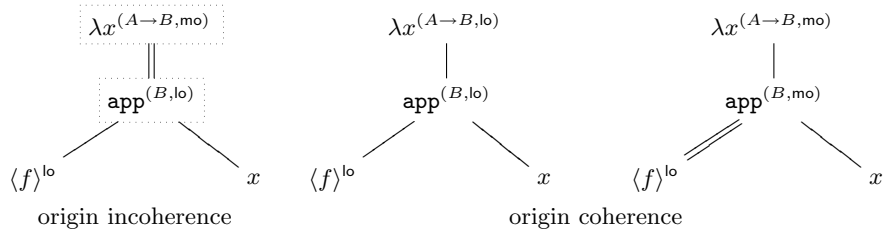

Figure 9: Origin incoherence and coherence - Examples

show that origin coherence is preserved by reduction.

Lemma 12 (Origin coherence — Decomposition lemma)
Let E be an evaluation context and e a closed term. If $E[e]$ is origin coherent, then so are E and e.

Note that in order to determine whether an evaluation context is origin coherent, the hole $-$ is not considered as a free variable.

Lemma 13 (Origin coherence — Substitution lemma)
Let e and e' be two terms that are origin coherent. If e is closed, then $e'[e/x]$ is origin coherent.

Note that if the free variable is not replaced with a closed term, the preservation may fail, as the following example shows:

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

We can now prove the preservation by reduction.

Proposition 14 (Origin coherence — Preservation by reduction)
Let (s, e) be a configuration reducing to (s', e'). If (s, e) is origin coherent, then so is (s', e').

Proof. By induction on the proof of $(s, e) \to (s', e')$, using Lemmas 12 and 13. □

We now give some lemmas useful to compute the frontier after a reduction. We need some preliminary definitions. Given two origin labels ι and ι', we define a function, $\mathcal{L}_\iota^{\iota'}$, which maps each annotated term e to a set of types, as follows:

$$\mathcal{L}_\iota^{\iota'}(e) \stackrel{def}{=} \begin{cases} \{A\} & \text{if } e = \mathtt{f}^{(A,\iota)}(\ldots) \text{ and } \iota' = \bar{\iota}\,, \\ \emptyset & \text{otherwise.} \end{cases}$$

This function locally computes the ι-frontier at the top of any term of the form $\mathtt{g}^{(\ldots,\iota')}(\ldots, e, \ldots)$, as shown in Figure 10. An evaluation context is said to *reduce under*

$\mathcal{L}_\iota^{\bar{\iota}}(e) = \{A\}, \mathcal{L}_\iota^{\iota}(e) = \mathcal{L}_{\bar{\iota}}^{\bar{\iota}}(e) = \mathcal{L}_{\bar{\iota}}^{\iota}(e) = \emptyset$ $\qquad$ $\mathcal{L}_{\bar{\iota}}^{\bar{\iota}}(e) = \mathcal{L}_\iota^{\iota}(e) = \mathcal{L}_\iota^{\bar{\iota}}(e) = \mathcal{L}_{\bar{\iota}}^{\iota}(e) = \emptyset$

Figure 10: Local Frontier

the origin label ι if it contains as a subterm $\mathtt{f}^{(\ldots,\iota)}(\ldots, -, \ldots)$, for some operator $\mathtt{f}$. We sometimes write E_ι for an evaluation context E reducing under ι. In the following lemmas, we suppose that an origin label ι, either lo or mo, is given.

Lemma 15 (Frontier — Composition lemma)
Let E be an evaluation context reducing under ι' and let e be a closed term. Then we have:

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

Lemma 16 (Frontier — Substitution lemma)
Let e be a term, e' a term with origin label ι', and x a variable. If e' is origin coherent and x is free in e', then we have:

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

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

Lemma 17 (Frontier — Memory store update)
Let s be a memory store, v a value, n a label with ι' as origin label and l a location such that l^n belongs to $\operatorname{dom} s \cup \{\nu_n(s)\}$. *Then we have:*

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

We now prove that frontier boundedness is preserved by reduction.

Proposition 18 (Frontier boundedness — Preservation by reduction) *Let T be a type. Consider a reduction $(s,e) \to (s',e')$ such that*

(i) the configuration (s,e) is well-typed and origin coherent,

(ii) the outgoing frontier and the incoming frontier of (s,e) are respectively included in the sets of the outgoing types and of the incoming types of T:

$$\mathcal{F}_{\mathsf{lo}}(s,e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})\,,$$
$$\mathcal{F}_{\mathsf{mo}}(s,e) \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})\,.$$

Then the outgoing frontier and the incoming frontier of (s',e') are respectively included in the sets of the outgoing types and of the incoming types of T:

$$\mathcal{F}_{\mathsf{lo}}(s',e') \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})\,,$$
$$\mathcal{F}_{\mathsf{mo}}(s',e') \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})\,.$$

We proceed by induction on the proof of the reduction. In the following proofs, each one corresponding to a reduction rule, we suppose that an origin label ι is given.

Proof — $[\beta]$. Suppose that the reduction $(s,e) \to (s',e')$ is

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

In order to show $\mathcal{F}_\iota(s,b[v/x]) \subseteq \mathcal{A}_\iota(\mathsf{T})$, we examine the different cases for b.

- $b = x$

We have $b[v/x] = v$; since $\mathcal{F}_\iota(s,v) \subseteq \mathcal{F}_\iota(s,e) \subseteq \mathcal{A}_\iota(\mathsf{T})$, we can conclude.

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

If $x \notin \mathrm{FV}(b)$, then $b[v/x] = b$; since $\mathcal{F}_\iota(s,b) \subseteq \mathcal{F}_\iota(s,e) \subseteq \mathcal{A}_\iota(\mathsf{T})$, we can conclude. We now suppose $x \in \mathrm{FV}(b)$.
Since $\lambda x^{(A\to B,\iota_2)}\,b$ is origin coherent and $x \in \mathrm{FV}(b)$, we have $\iota_2 = \iota_3$.
Since b is origin coherent, from Lemma 16, we deduce

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

Since by hypothesis $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(b) \cup \mathcal{F}_\iota(v) \subseteq \mathcal{A}_\iota(\mathsf{T})$, it remains to prove $\mathcal{L}_\iota^{\iota_3}(v) \subseteq \mathcal{A}_\iota(\mathsf{T})$.
Suppose $\mathcal{L}_\iota^{\iota_3}(v) \neq \emptyset$: we then have that v is labeled with (A,ι), $\mathcal{L}_\iota^{\iota_3}(v) = \{A\}$ and $\iota_3 = \bar{\iota}$. The reduction is represented in Figure 11, where the created frontier is indicated with a double line.
If $\iota_1 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(e)$, which is included in $\mathcal{A}_\iota(\mathsf{T})$ by hypothesis.
If $\iota_1 = \iota$, then $A \to B$ belongs to $\mathcal{F}_{\bar{\iota}}(e)$, which is included in $\mathcal{A}_{\bar{\iota}}(\mathsf{T})$ by hypothesis. By the inference rule $[- \to]$ (see Table 8, p. 19), we obtain that A belongs to $\mathcal{A}_\iota(\mathsf{T})$. □

Proof — [REF−?]. Suppose that the reduction $(s,e) \to (s',e')$ is

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

By Lemma 17 and since $\mathcal{F}_\iota(\mathtt{unit}^{(\mathtt{Unit},\iota_1)}) = \emptyset$, we have

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

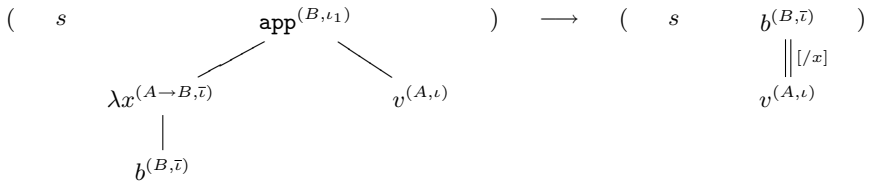

Figure 11: Proof — [β] — Interesting case

Since by hypothesis $\mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(v) \subseteq \mathcal{A}_\iota(\mathsf{T})$, it remains to prove $\mathcal{L}_\iota^{\iota_2}(v) \subseteq \mathcal{A}_\iota(\mathsf{T})$.
Suppose $\mathcal{L}_\iota^{\iota_2}(v) \neq \emptyset$: we then have that v is labeled with (A, ι), $\mathcal{L}_\iota^{\iota_2}(v) = \{A\}$ and $\iota_2 = \bar{\iota}$. The reduction is represented in Figure 12, where the created frontier is indicated with a double line.

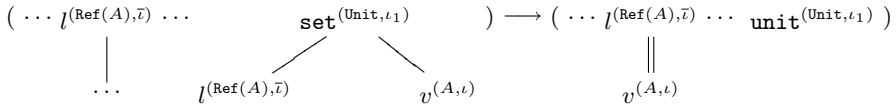

Figure 12: Proof — [REF−?] — Interesting case

If $\iota_1 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(e)$, which is included in $\mathcal{A}_\iota(\mathsf{T})$ by hypothesis.
Otherwise, $\iota_1 = \iota$ and $\mathtt{Ref}(A)$ belongs to $\mathcal{F}_{\bar{\iota}}(e)$, which is included in $\mathcal{A}_{\bar{\iota}}(\mathsf{T})$ by hypothesis; by the inference rule [$\mathtt{Ref}(-)$] (see Table 8 , p. 19), we obtain that A belongs to $\mathcal{A}_\iota(\mathsf{T})$. □

Proof — [REF] *or* [REF−!]. It can be easily shown that $\mathcal{F}_\iota(s', e') \subseteq \mathcal{F}_\iota(s, e)$. □

All the axioms are therefore satisfied. We now consider the inference rule [RED], involving an evaluation context.

Proof — [RED]. Suppose the reduction is $(s, E[r]) \to (s', E[r'])$, where E is an evaluation context that reduces under the origin label ι_0 and r is a redex. We assume the inductive hypothesis for the reduction $(s, r) \to (s', r')$.
By Lemma 15, we have:

$$\mathcal{F}_\iota(s, E[r]) = \mathcal{F}_\iota(s) \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r) \cup \mathcal{L}_\iota^{\iota_0}(r)\,,$$
$$\mathcal{F}_\iota(s', E[r']) = \mathcal{F}_\iota(s') \cup \mathcal{F}_\iota(E) \cup \mathcal{F}_\iota(r') \cup \mathcal{L}_\iota^{\iota_0}(r')\,.$$

By hypothesis, $\mathcal{F}_\iota(s, E[r]) \subseteq \mathcal{A}_\iota(\mathsf{T})$, hence $\mathcal{F}_\iota(s, r) \subseteq \mathcal{A}_\iota(\mathsf{T})$. From the inductive hypothesis, we deduce $\mathcal{F}_\iota(s', r') \subseteq \mathcal{A}_\iota(\mathsf{T})$. Since by hypothesis, $\mathcal{F}_\iota(E) \subseteq \mathcal{A}_\iota(\mathsf{T})$, it remains to prove $\mathcal{L}_\iota^{\iota_0}(r') \subseteq \mathcal{A}_\iota(\mathsf{T})$. We examine the different cases for the reduction premise $(s, r) \to (s', r')$.

Case [β] + [RED]. Suppose that the reduction $(s, E[r]) \to (s', E[r'])$ is

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

Suppose $\mathcal{L}^{\iota_0}_{\iota}(b[v/x]) \neq \emptyset$: we then have that $b[v/x]$ is labeled with (B, ι), $\mathcal{L}^{\iota_0}_{\iota}(b[v/x]) = \{B\}$ 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, B belongs to

$$\mathcal{F}_{\iota}(E_{\bar{\iota}}[\mathtt{app}^{(B,\iota_1)}\,(\lambda x^{(B\to B,\iota_2)}\,x, v^{(B,\iota)})])\,,$$

which is included in $\mathcal{A}_{\iota}(\mathsf{T})$ by hypothesis.

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

Since $b[v/x]$ is labeled with (B, ι), we have $\iota_3 = \iota$. The reduction is represented in Figure 13, where the created frontier is indicated with a double line.

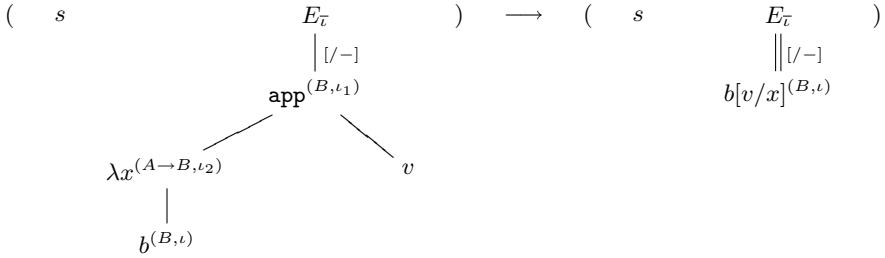

Figure 13: Proof — $[\beta]$ + [RED] — Interesting case

If $\iota_1 = \iota$, then B belongs to $\mathcal{F}_{\iota}(E[r])$, which is included in $\mathcal{A}_{\iota}(\mathsf{T})$ 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 $\mathcal{A}_{\iota}(\mathsf{T})$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \iota$, then $A \to B$ belongs to $\mathcal{F}_{\iota}(r)$, included in $\mathcal{A}_{\iota}(\mathsf{T})$ by hypothesis; by the inference rule $[\to +]$ (see Table 8, p. 19), we obtain that B belongs to $\mathcal{A}_{\iota}(\mathsf{T})$. ◇

Case [REF−!] + [RED]. Suppose that the reduction $(s, E[r]) \to (s', E[r'])$ is

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

Suppose $\mathcal{L}^{\iota_0}_{\iota}(s(l^{(\mathtt{Ref}(A),\iota_2)})) \neq \emptyset$: we then have that $s(l^{(\mathtt{Ref}(A),\iota_2)})$ is labeled with (A, ι), $\mathcal{L}^{\iota_0}_{\iota}(s(l^{(\mathtt{Ref}(A),\iota_2)})) = \{A\}$ and $\iota_0 = \bar{\iota}$. The reduction is represented in Figure 14, where the created frontier is indicated with a double line.

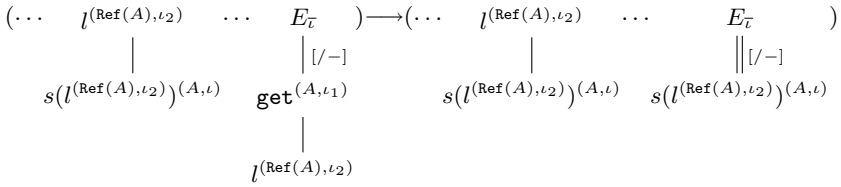

Figure 14: Proof — [REF−!] + [RED] — Interesting case

If $\iota_1 = \iota$, then A belongs to $\mathcal{F}_\iota(E[r])$, which is included in $\mathcal{A}_\iota(\mathsf{T})$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \bar{\iota}$, then A belongs to $\mathcal{F}_\iota(s)$, which is included in $\mathcal{A}_\iota(\mathsf{T})$ by hypothesis.
If $\iota_1 = \bar{\iota}$ and $\iota_2 = \iota$, then $\mathtt{Ref}(A)$ belongs to $\mathcal{F}_\iota(r)$, included in $\mathcal{A}_\iota(\mathsf{T})$ by hypothesis; by the inference rule [$\mathtt{Ref}(+)$] (see Table 8, p. 19), we obtain that A belongs to $\mathcal{A}_\iota(\mathsf{T})$. ◇

Case ([REF] or [REF−?]) + [RED]. By inspecting the reduction rules [REF] and [REF−?], we remark that $\mathcal{L}^{\iota_0}_\iota(r') = \mathcal{L}^{\iota_0}_\iota(r)$, which gives $\mathcal{L}^{\iota_0}_\iota(r') \subseteq \mathcal{A}_\iota(\mathsf{T})$. ◇

□

We can conclude by proving the premise of Proposition 10, which achieves the proof of Theorem 9.

Proposition 19 (Frontier boundedness)
Let T *be the type for the local environment. Consider any local environment* L *of type* T*, and any mobile program* $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ *of type* $\mathsf{T} \to \mathsf{R}$*.*
Then, for each configuration (s, e) *in the execution trace of the annotated program* $\mathsf{app}^{(\mathsf{R},\mathsf{mo})}\,(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})$*, the outgoing frontier of* (s, e) *is included in the set of the outgoing types of* T*:*

$$\mathcal{F}_{\mathsf{lo}}(s, e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})\,.$$

Proof. Let L be any local environment of type T, and let $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ be any mobile program of type $\mathsf{T} \to \mathsf{R}$. We show by induction on the execution trace of the annotated program $\mathsf{app}^{(\mathsf{R},\mathsf{mo})}\,(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}})$ that any configuration (s, e) in the trace is well-typed, origin coherent and satisfies $\mathcal{F}_{\mathsf{lo}}(s, e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{F}_{\mathsf{mo}}(s, e) \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})$.
Let c be the initial configuration

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

The configuration c is well-typed, origin coherent and satisfies $\mathcal{F}_{\mathsf{lo}}(c) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{F}_{\mathsf{mo}}(c) \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})$, since $\mathcal{F}_{\mathsf{lo}}(c) = \{\mathsf{T}\}$ and $\mathcal{F}_{\mathsf{mo}}(c) = \emptyset$.
Let $(s, e) \to (s', e')$ be a reduction belonging to the execution trace. Suppose that (s, e) is well-typed, origin coherent and satisfies $\mathcal{F}_{\mathsf{lo}}(s, e) \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{F}_{\mathsf{mo}}(s, e) \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})$. By Subject Reduction (see Proposition 1), (s', e') is well-typed. By Proposition 14, (s', e') is origin coherent. Finally, we can apply Proposition 18, to conclude $\mathcal{F}_{\mathsf{lo}}(s', e') \subseteq \mathcal{A}_{\mathsf{lo}}(\mathsf{T})$ and $\mathcal{F}_{\mathsf{mo}}(s', e') \subseteq \mathcal{A}_{\mathsf{mo}}(\mathsf{T})$. □

4. Completeness for the Confinement Criterion

The confinement criterion that we have given cannot be weakened since the converse of Theorem 9 (p. 20) is valid: an outgoing type is not confined.

Theorem 20 (Criterion completeness)
Let T *be the type for the local environment, and* A *a resource type. If the type* A *is an outgoing type of* T*, then there exists a local environment* L *of type* T *such that the type* A *is not confined in* L*.*

In other words, given an environment type T and a resource type A such that A is an outgoing type of T, we can define a local environment L of type T and a mobile program $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$ in order that a frontier redex accesses a resource of the type A and of the origin label lo during the execution of the annotated program

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

The proof of Theorem 20, which follows, is mainly a programming exercise. The reader can find in Appendix A some examples that illustrate the techniques involved in the proof and show how a hostile mobile program and an accomplice local environment can cooperate. For the proof, we use the programming language defined in Table 1 (p. 9). Its operational semantics can be deduced from the reduction relation described in Table 4 (p. 13) for the annotated language. The following notations facilitate programming.
A local definition is defined as follows:

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

where the variable x is not free in e_1, a sequential composition as follows:

$$e_1\,;\, e_2 \stackrel{def}{=} (\lambda x : A_1.\, e_2)\, e_1\,,$$

where e_1 has type A_1 and the variable x is not free in e_2, and a functional composition as follows:

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

where f and g have types $A_1 \to A_2$ and $A_2 \to A_3$ respectively, and the variable x is free neither in f nor in g.
Finally, we equip the set of memory stores with a concatenation operation. If the memory stores s_1 and s_2 are such that

$$\{l \mid \exists\, m\,.\ l^m \in \operatorname{dom} s_1\} = \{l \mid i \le l < j\}\,,$$
$$\{l \mid \exists\, m\,.\ l^m \in \operatorname{dom} s_2\} = \{l \mid j \le l < k\}\,,$$

for some locations i, j and k, then the memory store $s_1 \cdot s_2$ has domain $\operatorname{dom} s_1 \cup \operatorname{dom} s_2$ and is defined as follows:

$$s_1 \cdot s_2(l^m) \stackrel{def}{=} \begin{cases} s_1(l^m) \text{ if } l^m \in \operatorname{dom} s_1\,, \\ s_2(l^m) \text{ if } l^m \in \operatorname{dom} s_2\,. \end{cases}$$

Of course, the concatenation operation is associative.
We also need to define default values for each type. Precisely, each type is inhabited by a convergent program that can be effectively defined.

Lemma 21 (Inhabited types)
For each type A, there effectively exist a program a_A of type A, a value u_A and a memory store s_A, such that the configuration $(\emptyset, a_A)$ evaluates to (s_A, u_A).

The proof is straightforward, by induction on A.

The proof of Theorem 20 is by induction on the environment type. Consider the case where the environment type is a functional type $\mathsf{T}_1 \to \mathsf{T}_2$. If A is an outgoing type of $\mathsf{T}_1 \to \mathsf{T}_2$, then there exist two possibilities when A is different from $\mathsf{T}_1 \to \mathsf{T}_2$:

- either A is an outgoing type of T_2,
- or A is not, hence is an incoming type of T_1.

Indeed, we can use the alternative definition of the frontier upper bounds described in Table 10. This definition is equivalent to the one given in Table 8 (p. 19), since the

$$\frac{\emptyset}{\mathsf{T} \in \mathcal{A}_{\mathsf{lo}}(\mathsf{T})}$$

$$\frac{A \in \mathcal{A}_\iota(\mathsf{T}_1)}{A \in \mathcal{A}_{\bar{\iota}}(\mathsf{T}_1 \to \mathsf{T}_2)}\,[- \to] \qquad \frac{A \in \mathcal{A}_\iota(\mathsf{T}_2)}{A \in \mathcal{A}_\iota(\mathsf{T}_1 \to \mathsf{T}_2)}\,[\to +]$$

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

Table 10: Frontier upper bounds — Alternative inductive definition

sets inductively generated by the inference system in Table 10 are stable under the rules of the inference system in Table 8, and conversely, which can be easily shown by induction on proofs. For the former possibility, where A is an outgoing type of T_2, the inductive hypothesis can be applied, whereas for the latter, where A is an incoming type of T_1, it cannot, so that we proceed differently: we need to decompose $\mathsf{T}_1 \to \mathsf{T}_2$ some steps further. Precisely, we consider the proof of $A \in \mathcal{A}_{\mathsf{mo}}(\mathsf{T}_1)$ in the alternative inference system. Since the axiom of the proof is $A \in \mathcal{A}_{\mathsf{lo}}(A)$, there exists at least one application of the rule $[- \to]$ or $[\mathtt{Ref}(-)]$. The first application met from the conclusion defines two types T'_1 and T''_1 as follows:

$$\frac{A \in \mathcal{A}_{\mathsf{lo}}(\mathsf{T}''_1)}{A \in \mathcal{A}_{\mathsf{mo}}(\mathsf{T}'_1)}\,[\mathtt{Ref}(-)] \text{ or } [- \to]\,.$$

In order to apply the inductive hypothesis for T''_1, we use the following technical lemma. The conditions *(i)* to *(iv)* correspond to the preceding proof decomposition. The conditions *(v)* and *(vi)* define two functions used in the construction of the mobile program and the local environment.

Lemma 22 (Incoming types — Effective decomposition)
Let T *and* A *be two types such that* A *is an incoming type of* T*. Then there effectively exist two types* T' *and* T''*, and two programs* δ *and* γ*, of type* $\mathsf{T} \to \mathsf{T}'$ *and* $\mathsf{T}' \to \mathsf{T}$ *respectively, such that:*

(i) T' *is a type occurring in* T*,*

(ii) T' *is equal to* $\mathtt{Ref}(\mathsf{T}'')$ *or* $\mathsf{T}'' \to B$ *for some type* B,

(iii) A *is an incoming type of* T',

(iv) A *is an outgoing type of* T'',

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

(vi) for any configuration (s_1, v)*, where* s_1 *is a memory store and* v *a value of type* T'*, there exist a value* u *of type* T *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 using the two programs first to go from T' to T and then to come back, we can apply the inductive hypothesis in T'', as shown in Figure 15. Before describing how

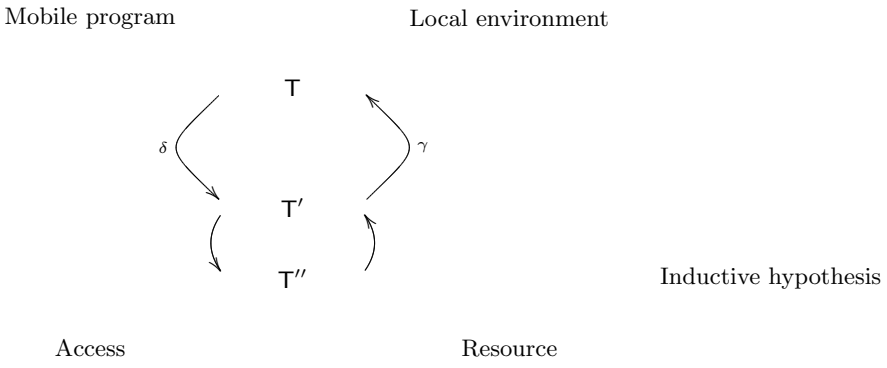

Figure 15: Technical lemma – Application of the inductive hypothesis

to proceed, we prove this lemma.

Proof. We proceed by induction on T. Let A be a type. We show that if A is an incoming type of T, then two types, T' and T'', and two programs, δ and γ, satisfying all the preceding conditions, can be defined.

• $\mathsf{T} = \mathtt{Unit}$

T has no incoming types.

• $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$

Suppose that A is an incoming type of T. There exist two possibilities, which we detail.

◦ A is an outgoing type of T_1.

We define T' as T, T'' as T_1, δ and γ as the identity function $\lambda x : \mathsf{T}.\, x$.

◦ A is not an outgoing type of T_1.

A is therefore an incoming type of T_2. The inductive hypothesis applied to T_2 gives T', T'', δ of type $\mathsf{T}_2 \to \mathsf{T}'$ and γ of type $\mathsf{T}' \to \mathsf{T}_2$.

Write F for $\lambda f : \mathsf{T}.\, f\, a_{\mathsf{T}_1}$ of type $\mathsf{T} \to \mathsf{T}_2$ and G for $\lambda y : \mathsf{T}_2.\, \lambda x : \mathsf{T}_1.\, y$ of type $\mathsf{T}_2 \to \mathsf{T}$.

The types T' and T'', and the programs $\delta \circ F$ and $G \circ \gamma$ are suitable.

- $\mathsf{T} = \mathtt{Ref}(\mathsf{T}_1)$

Suppose that A is an incoming type of T. There exist two possibilities, which we detail.

- A is an outgoing type of T_1.

We define T' as T, T'' as T_1, δ and γ as the identity function $\lambda x : \mathsf{T}.\, x$.

- A is not an outgoing type of T_1.

A is therefore an incoming type of T_1. The inductive hypothesis applied to T_1 gives T', T'', δ and γ.
Write F for $\lambda x : \mathsf{T}.\, \mathtt{get}(x)$ of type $\mathsf{T} \to \mathsf{T}_1$ and G for $\lambda x : \mathsf{T}_1.\, \mathtt{ref}(x)$ of type $\mathsf{T}_1 \to \mathsf{T}$.
The types T' and T'', and the programs $\delta \circ F$ and $G \circ \gamma$ are suitable. □

We can now prove Theorem 20. We give an effective construction of the mobile programs and the local environments, but 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 22, which we have chosen in order to simplify the verification, and on some intuitive properties of execution traces. A full formalization would be very long.

Proof — Theorem 20 — Criterion completeness.
We proceed by induction on the environment type T. Given a resource type A, we show that if A is an outgoing type of T, then we can define a local environment L of type T and a mobile program $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$ (for some type R) such that a frontier redex accesses a resource of the type A and of the origin label lo during the execution of the annotated program

$$\mathtt{app}^{(\mathsf{R},\mathsf{mo})}\left(\langle \lambda x : \mathsf{T}.\, \mathsf{M}[x] \rangle^{\mathsf{mo}}, \langle \mathsf{L} \rangle^{\mathsf{lo}}\right),$$

denoted by $\mathcal{M}(\mathsf{M}, \mathsf{L})$ in the following.
Let T be an environment type and A be a resource type such that A is an outgoing type of T. The case where A is equal to T being trivial, we suppose that A is different from T.

- $\mathsf{T} = \mathtt{Unit}$

T has no outgoing types, except itself.

- $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$

There exist two possibilities, which we detail.

- A is an outgoing type of T_2.

The inductive hypothesis applied to T_2 gives a local environment L' of type T_2 and a mobile program $\lambda x : \mathsf{T}_2.\, \mathsf{M}'[x]$ such that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended access. We define the local environment and the body of the mobile program as follows:

$$\mathsf{L} \,:\, \mathsf{T} \stackrel{def}{=} \lambda x : \mathsf{T}_1.\, \mathsf{L}',$$
$$\mathsf{M}[f \,:\, \mathsf{T}] \stackrel{def}{=} \left(\lambda x : \mathsf{T}_2.\, \mathsf{M}'[x]\right)\left(f\, a_{\mathsf{T}_1}\right).$$

- A is not an outgoing type of T_2.

A is therefore an incoming type of T_1. Lemma 22 gives two types T' and T'', and two

programs δ and γ, of type $\mathsf{T}_1 \to \mathsf{T}'$ and $\mathsf{T}' \to \mathsf{T}_1$ respectively, satisfying the different conditions. Since A is an outgoing type of T'', the inductive hypothesis applied to T'' gives a local environment L' of type T'' and a mobile program $\lambda x : \mathsf{T}''.\, \mathsf{M}'[x]$ such that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended access.
We consider two cases, according to the relationship between T'' and T' (see condition *(ii)* of Lemma 22).
◦ $\mathsf{T}' = \mathsf{T}'' \to B$
We define the local environment and the body of the mobile program as follows:

$$\begin{aligned} \mathsf{L} \,:\, \mathsf{T} &\stackrel{def}{=} \lambda x : \mathsf{T}_1.(\delta\, x\, \mathsf{L}'\,;\, a_{\mathsf{T}_2})\,, \\ \mathsf{M}[f \,:\, \mathsf{T}] &\stackrel{def}{=} f\,(\gamma\ \lambda x : \mathsf{T}''.(\mathsf{M}'[x]\,;\, a_B))\,. \end{aligned}$$

◦ $\mathsf{T}' = \mathtt{Ref}(\mathsf{T}'')$
We define the local environment and the body of the mobile program as follows:

$$\begin{aligned} \mathsf{L} \,:\, \mathsf{T} &\stackrel{def}{=} \lambda x : \mathsf{T}_1.(\mathtt{set}(\delta\, x, \mathsf{L}')\,;\, a_{\mathsf{T}_2})\,, \\ \mathsf{M}[f \,:\, \mathsf{T}] &\stackrel{def}{=} \mathtt{let}\ z \,:\, \mathsf{T}' = a_{\mathsf{T}'}\ \mathtt{in} \\ &\qquad (f\,(\gamma\, z)\,;\, (\lambda x : \mathsf{T}''.\, \mathsf{M}'[x])\,\mathtt{get}(z))\,. \end{aligned}$$

• $\mathsf{T} = \mathtt{Ref}(\mathsf{T}_1)$
There exist two possibilities, which we detail.
◦ A is an outgoing type of T_1.
The inductive hypothesis applied to T_1 gives a local environment L' of type T_1 and a mobile program $\lambda x : \mathsf{T}_1.\, \mathsf{M}'[x]$ such that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended access. We define the local environment and the body of the mobile program as follows:

$$\begin{aligned} \mathsf{L} \,:\, \mathsf{T} &\stackrel{def}{=} \mathtt{ref}(\mathsf{L}')\,, \\ \mathsf{M}[z \,:\, \mathsf{T}] &\stackrel{def}{=} \mathtt{let}\ x \,:\, \mathsf{T}_1 = \mathtt{get}(z)\ \mathtt{in}\ \mathsf{M}'[x]\,. \end{aligned}$$

◦ A is not an outgoing type of T_1.
A is therefore an incoming type of T_1. Lemma 22 gives two types T' and T'', and two programs δ and γ, of type $\mathsf{T}_1 \to \mathsf{T}'$ and $\mathsf{T}' \to \mathsf{T}_1$ respectively, satisfying the different conditions. Since A is an outgoing type of T'', the inductive hypothesis applied to T'' gives a local environment L' of type T'' and a mobile program $\lambda x : \mathsf{T}''.\, \mathsf{M}'[x]$ such that the execution of $\mathcal{M}(\mathsf{M}', \mathsf{L}')$ entails the intended access.
We consider two cases, according to the relationship between T'' and T' (see condition *(ii)* of Lemma 22).
◦ $\mathsf{T}' = \mathsf{T}'' \to B$
We define the local environment and the body of the mobile program as follows:

$$\begin{aligned} \mathsf{L} \,:\, \mathsf{T} &\stackrel{def}{=} \mathtt{let}\ z \,:\, \mathsf{T} = a_{\mathsf{T}}\ \mathtt{in} \\ &\qquad (\mathtt{set}(z, \gamma\ \lambda x : \mathsf{T}''.\, \delta\, \mathtt{get}(z)\, \mathsf{L}')\,;\, z)\,, \\ \mathsf{M}[z \,:\, \mathsf{T}] &\stackrel{def}{=} \mathtt{let}\ y \,:\, \mathsf{T}_1 = \mathtt{get}(z)\ \mathtt{in} \\ &\qquad (\mathtt{set}(z, \gamma\ \lambda x : \mathsf{T}''.(\mathsf{M}'[x]\,;\, a_B))\,;\, \delta\, y\, a_{\mathsf{T}''})\,. \end{aligned}$$

◦ $\mathsf{T}' = \mathtt{Ref}(\mathsf{T}'')$

We define the local environment and the body of the mobile program as follows:

$$\mathsf{L} : \mathsf{T} \stackrel{def}{=} \mathtt{let}\ y : \mathsf{T}' = \mathtt{ref}(\mathsf{L}')\ \mathtt{in}\ \mathtt{ref}(\gamma\, y)\,,$$
$$\mathsf{M}[z : \mathsf{T}] \stackrel{def}{=} \mathtt{let}\ x : \mathsf{T}'' = \mathtt{get}(\delta\, \mathtt{get}(z))\ \mathtt{in}\ \mathsf{M}'[x]\,.$$

□

5. Confinement Criterion in Action: Example of Access Qualifiers

Whereas the confinement criterion as stated in Theorem 9 (p. 20) is a general property of the programming language that does not refer to any particular security architecture, it is interesting to consider the relationship of the criterion with standard security architectures. We detail the example of access qualifiers, whose addition to a programming language provides a form of access control. Access qualifiers are found in many popular object-oriented programming languages (like C++ and Java) where they provide a simple mechanism to hide information (about data representation): each object contains a private state and a public interface. From a security perspective, this addition requires two steps: first, each resource of a program receives an access control list, which registers the access rights that the listed subjects have with respect to the resource, second, each call to an access function is mapped to a subject accessing. Finally, a static analysis ensures the following property: only authorized accesses happen during the execution of a program.
In the presence of mobile code, we study the following scenario: the local environment contains access qualifiers and is fully analyzed since its code is available, whereas the mobile program is a standard program without access qualifiers. We can therefore decompose the verification process into two steps. First, analyze the local environment, as a whole program, second, determine a criterion which enables the following inference: from the analysis result for the local environment, infer the analysis result for any mobile program calling the local environment. The criterion is just some extra condition that the local environment must satisfy in order to make the inference valid. Given a static analysis for controlling accesses, whether the associated criterion is equivalent to our confinement criterion is an open question that we now study in the particular case of access qualifiers.

We begin by adding access qualifiers to our introductory example using the class `resource` (see Program 1, p. 2). To simplify, we only consider two subjects, the former with high privileges, the latter with low privileges, and two access control lists, the former for sensitive resources, which can only be accessed by the subject with high privileges, the latter for non-sensitive resources, which can be accessed by any subject. This situation can be faithfully implemented as follows. First, we associate to each subject a method, the method `access_high` for the subject with high privileges and the method `access` for the subject with low privileges. Second, we associate to each access control list a resource class that contains the authorized methods. We thus obtain the class definitions in Program 6. Consider a standard program, that is to say a program using the original class `resource`, which has a unique method `access`. With

```
(* sensitive resources *)
class resource_high (qualifier:string) =
  object
    (* privileged access *)
    method access_high (subject:string) =
      print_string (subject
                    ^ " with high privileges accesses "
                    ^ qualifier ^ " resource\n")
  end
(* non-sensitive resources *)
class resource (qualifier:string) =
  object
    (* privileged access *)
    method access_high (subject:string) =
      print_string (subject
                    ^ " with high privileges accesses "
                    ^ qualifier ^ " resource\n")
    (* non-privileged access *)
    method access (subject:string) =
      print_string (subject
                    ^ " with low privileges accesses "
                    ^ qualifier ^ " resource\n")
  end
(* unsecure implies ill-typed *)
let unsecure_code =
  let res = new resource_high "sensitive" in res#access
(* well-typed implies secure*)
let secure_code =
  let res = new resource_high "sensitive" in res#access_high
```

Program 6: Adding access qualifiers

the class definitions in Program 6, in order to add access qualifiers to the program, we can proceed as follows, by using a simple annotation of the program. First, a resource becomes an instance of the class `resource_high` if we want to qualify it as sensitive and a call `res#access` becomes a call `res#access_high` when done by the subject with high privileges. Then, type checking provides the static analysis to enforce the security policy: with a well-typed program, only authorized accesses happen during the execution. If we adopt the convention that a sensitive resource is always created with **new** `resource_high` "sensitive" and that a non-sensitive resource is always created with **new** `resource` "non−sensitive", then type checking ensures the following security property: it is impossible to print the following message

```
"... with low privileges accesses sensitive resource" .
```

See the examples of `secure_code` and `unsecure_code` in Program 6.
In the presence of mobile code, we actually meet the following scenario. The local en-

vironment contains access qualifiers and is well-typed in the type system enriched with `resource_high`, whereas the mobile program is a standard program. A possible annotation of the local environment described in Program 3 (p. 4) is given in Program 7. First, the instance `controller` of the class `proxy` is now defined as the proxy of the sen-

```
(* local environment with access qualifiers *)
module Env4 =
  struct
    (* sensitive resources *)
    class resource_high (qualifier:string) =
      object
        (* privileged access *)
        method access_high (subject:string) = ...
      end
    (* non-sensitive resources *)
    class resource (qualifier:string) =
      object
        (* privileged access *)
        method access_high (subject:string) = ...
        (* non-privileged access *)
        method access (subject:string) = ...
      end
    (* secure proxy definition *)
    class proxy (res:resource_high) =
      object
        (* secure method *)
        method request (subject:string) =
          res#access_high subject
      end
    (* indirect access via a proxy *)
    let controller =
      let confined_res =
        new resource_high "sensitive" in
        new proxy confined_res
  end
```

Program 7: Local environment with access qualifiers

sitive resource `confined_res`. Then, the call `res#access` in the method `request` needs to be annotated in order to get a well-typed program. A mobile program can indirectly access the resource `confined_res` with a call `Env4.controller#request` "applet", which prints to the standard output:

```
"applet with high privileges accesses sensitive resource" .
```

Thus, the local environment locally grants high privileges to the applet. Now, suppose that we add a definition `danger` to `Env4`, as in Program 4 (p. 4): in Program 8, the definition of `danger` is supposed to be annotated and well-typed, so that the extended

local environment `Env5` is still well-typed. Is a mobile program calling `Env5.danger`

```
(* environment Env4 extended *)
module Env5 =
  struct
    ...
    (* well-typed definition *)
    let danger = ...
  end
```

Program 8: Problematic environment

secure? The answer is affirmative if it is well-typed. If the type `resource_high` does not occur in the type `t` of `danger`, then any mobile program calling `Env5.danger` is well-typed, hence secure. But we can refine this result. Indeed, since the type `resource` is equivalent to the type

```
< access_high : string -> unit; access : string -> unit>
```

and the type `resource_high` to the type

```
< access_high : string -> unit > ,
```

we can deduce that the type `resource` is a subtype of the type `resource_high`. If the annotated type `t` can be subsumed to a standard type `t'`, where `resource_high` does not occur, then a mobile program calling `Env5.danger` becomes well-typed if we coerce `danger` to the type `t'`. For instance, if the type `t` is equal to `resource_high -> s`, where `resource_high` does not occur in `s`, then by contravariance, the type `t` is a subtype of `resource -> s`, which contains no occurrence of `resource_high`. On the contrary, if the type `t` is equal to `resource_high` **ref** `-> s`, we cannot subsume `t` into `resource` **ref** `-> s`. Actually, we claim that the subsumption from `t` to `t'` is valid if, and only if, the annotated type `resource_high` does not occur in the type `t'` either at a positive occurrence, or under the reference type constructor **ref**: this criterion, deduced from the subtyping properties of the annotated type system, is exactly our confinement criterion.

The rest of the section is devoted to a formal proof of the preceding claim. We adopt Heintze and Riecke's SLam-calculus ("Secure Lambda-calculus") [13]: indeed, it is a paradigmatic model for a language with access qualifiers and with a type-based analysis for controlling accesses. We restrict the SLam-calculus to access control, whereas it also deals with information flows.

In the following, we use a set of *security labels*, which is a partial order. In order to simplify, while permitting a straightforward generalization, we use the pair $\{\bot, \top\}$, ordered by $\bot < \top$. A security label corresponds first to a subject, second to an access control list defined as follows:

> the subject σ is authorized to access the resource with the access control list α if $\alpha \leq \sigma$.

In other words, the subject $\top$ can access any resource, whereas the subject $\bot$ can only access the resources with the access control list $\bot$, or alternatively, a resource with the access control list $\top$, called a sensitive resource, can only be accessed by the subject $\top$, whereas a resource with the access control list $\bot$ can be accessed by any subject.
In order to assign to each resource an access control list and to each access a subject, we again resort to an annotated language that preserves labels, as defined in Section 2. 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: the label of an access operator represents the subject making the access, whereas the label of a constructor represents the access control list assigned to the resource built from the constructor. Our example is thus modeled as follows: the calls ...#`access` and ...#`access_high` correspond to access operators labeled with $\bot$ and $\top$ respectively, and the instances of the classes `resource` and `resource_high` correspond to resources labeled with $\bot$ and $\top$ respectively.
The static analysis enforcing access control is based on a system of annotated types, an extension of the standard type system. Precisely, the types are generated by the grammar in Table 11. How can we define the labels of the annotated language? Recall

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

Table 11: Annotated types

that the label of an operator is an ordered pair whose first component is a standard type and whose second component is a piece of information (see Section 2). If the information label simply was the security label, then we could not deduce from the label of a term its annotated type: for instance, since the type system ensures that the type of a sensitive resource is labeled with $\top$, the identity program $\lambda x^{(A,\top)}\, x$, where A is a standard type, would have any annotated type $A' \xrightarrow{\top} A'$, where A' results from an annotation of the type A. That is the reason why we define the information label as a security label associated with an annotated type. Since an annotated type gives the underlying standard type if we erase labels, we can simplify: a label becomes an ordered pair (A, τ), where A is an annotated type, called the *type label*, and where τ belongs to $\{\bot, \top\}$ and is called the *security label*. Thus, the preceding program becomes $\lambda x^{(A',\top)}\, x$, with type $A' \xrightarrow{\top} A'$. This label definition gives a least type for a term, as we will see.
The annotated type system is described by the inference system in Table 12. A typing judgment has the form $\Gamma \vdash e : A$, where Γ is a typing environment, e an annotated term and A an annotated type. The type system follows two main design principles:

- a resource (or a redex creating a new reference) labeled with α in $\{\bot, \top\}$ has a type labeled with α;

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

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

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

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

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

Table 12: Annotated type system

- elimination rules enforce an access control: the security label of an access operator is greater than the security label of the resource accessed.

These principles lead to the fundamental property, detailed in Corollary 24: a program that is well-typed in the system is secure, which means that only authorized accesses happen during its execution.
A form of subtyping can be added to the type system: given any type constructor $\mathtt{F}$, a type $\mathtt{F}^{\perp}(\ldots)$ becomes a subtype of $\mathtt{F}^{\top}(\ldots)$. Indeed, the substitution principle, as defined by Liskov and Wing [17], is valid in the following form: if a program is secure when it uses a value of type $\mathtt{F}^{\top}(\ldots)$, then it is also secure when it uses instead any value of type $\mathtt{F}^{\perp}(\ldots)$. The subtyping relation is inductively generated by the inference system given in Table 13: judgments are inequalities $A \leq B$, meaning that A is a subtype of B. Note the following properties, which are standard:

$$\frac{\emptyset}{\mathtt{Unit} \leq \mathtt{Unit}} \qquad \frac{A_2 \leq A_1 \quad B_1 \leq B_2}{A_1 \xrightarrow{\tau_1} B_1 \leq A_2 \xrightarrow{\tau_2} B_2} \quad (\tau_1 \leq \tau_2) \qquad \frac{\emptyset}{\mathtt{Ref}^{\,\tau_1}(A) \leq \mathtt{Ref}^{\,\tau_2}(A)} \quad (\tau_1 \leq \tau_2)$$

Table 13: Subtyping relation

- 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);
- for the type constructor $\mathtt{Ref}(-)$, the subtyping rule is invariant on the unique component;
- 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 annotated type system equipped with the conversion rule ensures that the type label of a term well-typed in some typing environment is the least type that the term receives in the environment.

Following Heintze and Riecke [13, Th. 2.1, 3.1], we now state the main property, which relates the static and the dynamic semantics. A memory store s is said to be *well-typed* (in the annotated type system) if, for all locations l^m in $\operatorname{dom} s$, l^m is well-typed, which is equivalent to $m = (\mathtt{Ref}^{\,\tau}(A), \tau)$ for some type A and some security label τ, 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 23 (Subject reduction — Type decreasing)
Let (s, e) be a well-typed configuration reducing to (s', e'). Then (s', e') is well-typed and moreover the least type of e' is a subtype of the least type of e.

The proposition can be easily shown by induction on the proof of the reduction $(s, e) \to (s', e')$. It implies that a program is secure if it is well-typed in the annotated type system.

Corollary 24 (Type soundness)
If an annotated program is well-typed in the annotated type system, then, during its execution, the subject $\bot$ does not access a sensitive resource.

Proof. By Subject Reduction, every configuration in the execution trace is well-typed. Suppose that during the execution, an access operator labeled with $\bot$ accesses a resource labeled with $\top$. Then the corresponding redex is ill-typed; this is a contradiction by the decomposition lemma, which asserts that any subterm of a well-typed term is also well-typed. □

In other words, the static verification of typing entails the dynamic verification of the security policy.

In the presence of mobile code, we study the following scenario: the local environment can contain access qualifiers and be fully analyzed since its code is available, whereas the mobile program cannot. We therefore consider that

- the local environment is annotated and well-typed in the annotated type system,
- the mobile program is not annotated, but just well-typed in the standard type system.

Actually, if we entirely annotate the mobile program with $\bot$, then we can assign to it a type entirely annotated with $\bot$ in the annotated type system. This annotation with $\bot$ corresponds to two natural assumptions:

- only local resources need to be protected;

- the mobile program receives the least privileges and therefore acts on behalf of the subject $\bot$.

Formally, some notations are needed. If A is an annotated type and e an annotated term, we denote by $\downarrow(A)$ and $\downarrow(e)$ the standard type and the standard term resulting from erasing the labels in A and e; if A is a standard type and e a standard term, we denote by $\langle A\rangle^\alpha$ and $\langle e\rangle^\alpha$ the type and the term resulting from an entire annotation of A and e with α.
The local environment is therefore a well-typed annotated program L, with T as least type, whereas the mobile program $\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot$ is entirely annotated with $\bot$, with $\langle\downarrow(\mathsf{T})\to\mathsf{R}\rangle^\bot$ as least type. The question becomes: is the annotated mobile program calling the local environment secure? The answer is affirmative if it is well-typed in the annotated type system. Actually, as shown in Figure 16, the annotated mobile program calling the local environment is well-typed if, and only if, the local environment can receive a type entirely annotated with $\bot$, in other words, $\mathsf{T}\le\langle\downarrow(\mathsf{T})\rangle^\bot$. If the annotated type of the local environment contains the label $\top$, it may be sub-

$$\dfrac{\dfrac{\dfrac{\vdots}{\emptyset\vdash\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot\ :\ \langle\downarrow(\mathsf{T})\to\mathsf{R}\rangle^\bot}}{\emptyset\vdash\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot\ :\ T\stackrel{\bot}{\to}\langle\mathsf{R}\rangle^\bot}\,(T\le\langle\downarrow(\mathsf{T})\rangle^\bot) \qquad \dfrac{\dfrac{\vdots}{\emptyset\vdash\mathsf{L}\ :\ \mathsf{T}}}{\emptyset\vdash\mathsf{L}\ :\ T}\,(\mathsf{T}\le T)}{\emptyset\vdash\mathtt{app}^{(\langle\mathsf{R}\rangle^\bot,\bot)}\,(\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot,\mathsf{L})\ :\ \langle\mathsf{R}\rangle^\bot}$$

Figure 16: Mobile code calling the local environment — Typing proof

sumed in another type entirely annotated 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 $\mathtt{Ref}^{\,\top}(A)\stackrel{\bot}{\to}B$ is a subtype of $\mathtt{Ref}^{\,\bot}(A)\stackrel{\bot}{\to}B$ because of the rule of domain contravariance. Finally, this technique of converting the local environment leads to the following criterion, which also uses the outgoing types (see Definition 8, p. 20).

Theorem 25 (Mobile code — Security criterion)
Let T be an annotated type for the local environment. If each outgoing type of T is labeled with $\bot$, then, for every local environment L of type T, for every mobile program $\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot$, the subject $\bot$ does not access a sensitive resource during the execution of the mobile program calling the local environment,

$$\mathtt{app}^{(\langle\mathsf{R}\rangle^\bot,\bot)}\,(\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot,\mathsf{L})\,.$$

Proof. We can easily prove by induction on the type T the following equivalences:

(i) $\mathsf{T}\le\langle\downarrow(\mathsf{T})\rangle^\bot$ if, and only if, each outgoing type of T is labeled with $\bot$,
(ii) $\langle\downarrow(\mathsf{T})\rangle^\bot\le\mathsf{T}$ if, and only if, each incoming type of T is labeled with $\bot$.

Suppose that each outgoing type of T is labeled with $\bot$. We deduce $\mathsf{T}\le\langle\downarrow(\mathsf{T})\rangle^\bot$. Hence, the program

$$\mathtt{app}^{(\langle\mathsf{R}\rangle^\bot,\bot)}\,(\langle\lambda x :\downarrow(\mathsf{T}).\,\mathsf{M}[x]\rangle^\bot,\mathsf{L})$$

is well-typed in the annotated type system. By Corollary 24, we can conclude. □

The criterion inferred from the annotated type system and its subtyping properties turns out to be equivalent to our confinement criterion (see Theorem 9, p. 20). Indeed, the typing of the local environment provides half of the verification: confinement for local sensitive resources is also needed, since mobile code could otherwise directly access a sensitive resource.

6. Conclusion

The confinement criterion that we have defined is type-based: it takes as inputs a resource type A and a type T for the local environment and determines whether, for each local environment L of type T, the type A is confined in L, which means that no resource of type A belonging to the local environment 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 does not occur in the environment type T either at a positive occurrence, or 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 [16], which forbids the resource type to occur in the environment type. Moreover, we have proved that our criterion cannot be weakened: if the resource type A occurs in the environment type T at a positive occurrence or under the reference type constructor $\mathtt{Ref}(-)$, then there exists a local environment L of type T such that A is not confined in L.

It remains that the criterion is only valid for a functional language with references. Vitek et al.'s works about confinement [31, 20] 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 (see rules $\mathcal{C}_2$ and $\mathcal{C}_3$ [20, 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 first on the annotation of the programming language, in order to keep track of code origin, second on the study of the interaction frontier between the mobile code and the local code. It allows confinement to be rigorously defined, which seems to be its decisive advantage over the previous methods. Future works will apply our method to different features of object-oriented programming languages, like data abstraction, subtyping and inheritance. We anticipate that our method is suitable for a lot of data types, particularly for ordered pair, record and list types, and also recursive types. Since objects can be represented by references containing records of pre-methods, which are functions parameterized by the object itself (following the self-application model of Kamin [15]), and object types can be represented by recursive types, objects could be added to our programming language without difficulty. One step further, we could adopt the imperative object calculus of Abadi and Cardelli [2, chap. 10–11] instead of the λ-calculus. Further investigations are needed to confirm these expectations and could benefit from Vitek et al.'s works [31, 20]. 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 sensitive resources (see rule $\mathcal{C}_5$ [20, p. 137]).

Another possible extension is parametric polymorphism: a program fragment can be parameterized with a type, and used in a polymorphic way, for any instantiation of the type parameter. Parametric polymorphism enables the confinement of a local resource by making its type abstract in the mobile code, as shown by Leroy and Rouaix [16, sect. 5.2]. The transition from a monomorphic type system to a polymorphic type system requires our annotation technique to be improved: indeed, it becomes impossible to label an operator by a unique type, since any term receives a type from inside, according to its subterms, and another type from outside, according to its use, and these types may differ because of type parameterization. For instance, if the type of a local resource is made abstract in the mobile code, the type of the local resource, known in the local code, becomes unknown in the mobile code where the resource is therefore used in a restricted way. Grossman et al. [11] and Rossberg [26] provide solutions to this problem that could be adapted to our specific question.

We have also studied the relationship of the confinement criterion with a standard security architecture with access qualifiers, modeled with the SLam-calculus [13]. First, the security architecture ensures that the local environment enforces a security policy specified with an access control matrix. Then, we have proved that if the local environment satisfies the confinement criterion, no mobile program calling the local environment can violate the security policy, although the security architecture does not apply to the mobile program. An interesting question consists in extending the result to other security architectures enforcing access control. For example, the preceding method would be useful for security policies specified by security automata, as defined by Schneider [28], which improve the policy expressivity since access rights can depend on the history of the program execution.

Besides access control, security requirements may also deal with information flows. For instance, the content of a local resource, like a cryptographic key or a password, may be confidential, which means that the mobile code cannot obtain information about the content during its execution in the local environment. The absence of information flows between a local resource and the mobile program implies that the resource is confined. However, its confinement does not imply the absence of information flows, which can result from indirect accesses: the mobile program requests the local environment, which accesses the resource and replies to the mobile program by returning the confidential piece of information. Confinement is therefore a weaker security property than confidentiality. Language-based techniques for ensuring confidentiality are presented in the exhaustive survey of Myers and Sabelfeld [27]: they all track information flows in whole programs. Is it possible to define confidentiality criteria analogous to our confinement criterion? Suppose that a technique enforcing a confidentiality policy is applied to the local environment. A confidentiality criterion would ensure that if the local environment satisfies it, then no mobile program calling the local environment could violate the confidentiality policy. This interesting question, already raised by Leroy and Rouaix [16, sect. 8], is still open.

A. Access Flaws: Some Examples in OCaml

The following examples illustrate the proof of Theorem 20 (p. 27), which shows the completeness of the confinement criterion described in Theorem 9 (p. 20): given an environment type T and a resource type A such that A is an outgoing type of T, we can define a local environment L of type T and a mobile program $\lambda x : \mathsf{T}.\, \mathsf{M}[x]$ of type $\mathsf{T} \to \mathsf{R}$ in order that the mobile program directly accesses a local resource during its execution in the local environment.

The examples are developed using OCaml (see [25, 24]). Although the language permits polymorphism, we only define monomorphic programs, by using the Church notation: each parameter receives a monomorphic type, which ensures that any expression receives a unique monomorphic type. We pursue the introductory example with the class `resource` (see Program 1, p. 2). The local environment is represented by an OCaml module called `Env`. The module provides the definition of the class `resource` and of an expression `danger`. The type of `danger` does not satisfy the confinement criterion: the type `resource` occurs either at a positive occurrence, or under the type constructor **ref**. The expression `danger` begins with the local definition of a local resource `accessible_res`: see Program 9. Our goal is to define the expression

```
(* local environment *)
module Env =
  struct
    class resource (origin:string) =
      object
        method access (subject:string) =
          print_string (subject ^ " accesses "
                        ^ origin ^ " resource\n")
      end
    (* type not satisfying the confinement criterion *)
    let danger =
      (* local definition of a local resource *)
      let accessible_res = new resource "local" in
      ...
```

Program 9: Local environment

`danger` and a mobile program calling the module `Env` such that the mobile program directly accesses the local resource `accessible_res`. More precisely, the mobile program must call the following auxiliary function, called `attack`, with `accessible_res` as the argument:

```
let attack (res:Env.resource) = (* attack function *)
  res#access "hostile applet"  ,
```

which prints to the standard output:

```
"hostile applet accesses local resource" .
```

We now describe some interesting cases. We suppose that a local resource of type A is given. For each given type T for the local environment, we informally define a local environment of type T 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`. The definitions of the local environment `Env` and of the mobile programs `mobile_program` 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 #.
If $\mathsf{T} = (A \to B) \to C$, then A occurs at a positive occurrence in T. The mobile program passes to the environment a function that realizes an access to its unique argument; the environment applies this function to the resource: see `danger1` and `mobile_program1` in Program 10 for details.
If $\mathsf{T} = \mathtt{Ref}(A) \to B$, 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 `danger2` and `mobile_program2` in Program 10 for details.
If $\mathsf{T} = \mathtt{Ref}(A \to B)$, then A again occurs under the type constructor $\mathtt{Ref}(-)$. It turns out that the situation is more difficult. The local environment evaluates to a location l^{T} of type T, which is initialized with a particular function f: when f is applied to an argument, it first reads the content of l^{T} to obtain a function of type $A \to B$, and then applies this function to the local resource of type A, which may give some value of type B. Of course, as long as the content of l^{T} is f, the function f diverges for any argument. As for the mobile program, it first reads the content of its argument l^{T} in order to obtain f, then assigns to l^{T} a function of type $A \to B$ that 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^{T}, now the function realizing the access, is thus applied to the local resource. See `danger3` and `mobile_program3` in Program 10 for details.

Acknowledgements

Thanks are due to Gilbert Caplain, Didier Le Botlan and Carol Robins for helpful comments and very thorough readings of different versions of the manuscript. Thanks also to the programme committee of the workshop "Security Analysis of Systems: Formalism and Tools" for inviting me to submit this article, whose first version was presented during the workshop. I am grateful to the members of my thesis jury, Didier Caucal, Norbert Cot, Thomas Jensen, René Lalement and Xavier Leroy, who have encouraged me to publish this work, extracted from my doctoral dissertation at the "École nationale des ponts et chaussées". I am also grateful to the anonymous referees, who have greatly contributed to improve the presentation of this work.

```
# module Env =
  struct
    class resource (origin:string) = ...
    let danger1 =
      let accessible_res = new resource "local" in
      function (f:resource -> unit) -> f accessible_res
    let danger2 =
      let accessible_res = new resource "local" in
      function (l:resource ref) -> l := accessible_res
    let danger3 =
      let accessible_res = new resource "local" in
      let l = ref (function (res:resource) -> ()) in
      let f = function (res:resource) -> !l accessible_res in
      l := f ; l
  end;;
module Env :
  sig
    class resource : string ->
      object method access : string -> unit end
    val danger1 : (resource -> unit) -> unit
    val danger2 : resource ref -> unit
    val danger3 : (resource -> unit) ref
  end

# let attack (res:Env.resource) = (* attack function *)
  res#access "hostile applet" ;;
val attack : Env.resource -> unit = <fun>

# let mobile_program1 = (* attack with danger1 *)
  Env.danger1 attack ;;
hostile applet accesses local resource
val mobile_program1 : unit = ()

# let mobile_program2 = (* attack with danger2 *)
  let l = ref (new Env.resource "mobile") in
  Env.danger2 l ; attack !l ;;
hostile applet accesses local resource
val mobile_program2 : unit = ()

# let mobile_program3 = (* attack with danger3 *)
  let f = !Env.danger3 in
  let r = new Env.resource "mobile" in
  Env.danger3 := attack ; f r ;;
hostile applet accesses local resource
val mobile_program3 : unit = ()
```

Program 10: Attacks on `resource`

References

[1] Martin Abadi. Protection in programming-language translations. In Vitek and Jensen [32], pages 19–34.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In PPDP '02 [23], pages 76–87.

[4] Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model-checking security properties of control-flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

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

[6] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In POPL '00 [21], pages 54–66.

[7] 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.

[8] Cédric Fournet and Andrew Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

[9] 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.

[10] 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.

[11] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, 2000.

[12] 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.

[13] Nevin Heintze and Jon Riecke. The SLam calculus: Programming with security and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '98)*, pages 365–377. ACM Press, 1998.

[14] Tomoyuki Higuchi and Atsushi Ohori. Java bytecode as a typed term calculus. In PPDP '02 [23], pages 201–211.

[15] Samuel Kamin and Uday Reddy. Two semantic models of object-oriented languages. In Carl Gunter and John Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 463–495. MIT Press, 1994.

[16] Xavier Leroy and François Rouaix. Security properties of typed applets. In Vitek and Jensen [32], pages 147–182.

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

[18] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

[19] George Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–89. Springer-Verlag, 1998.

[20] 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.

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

[22] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.

[23] *Proceedings of the 4th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '02)*. ACM Press, 2002.

[24] Didier Rémy. Using, understanding, and unraveling the OCaml language. In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 413–537. Springer-Verlag, 2002.

[25] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[26] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '03)*, pages 241–252. ACM Press, 2003.

[27] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal of Selected Areas in Communications*, 21(1):5–19, 2003.

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

[29] Christian Skalka and Scott Smith. Static use-based object confinement. *International Journal of Information Security*, 4(1–2):1–18, 2005.

[30] Peter Thiemann. Enforcing security properties by type specialization. In *Programming Languages and Systems: 10th European Symposium on Programming,*

ESOP 2001, Proceedings, volume 2028 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 2001.

[31] 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.

[32] Jan Vitek and Christian Jensen, editors. *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[33] David Walker. A type system for expressive security policies. In POPL '00 [21], pages 254–267.

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

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