On Declassification and the Non-Disclosure Policy (*)

Ana Almeida Matos and Gérard Boudol

INRIA Sophia Antipolis

Abstract

We address the issue of declassification in a language-based security approach. We introduce, in a Core ML-like language with concurrent threads, a declassification mechanism that takes the form of a local flow policy declaration. The computation in the scope of such a declaration is allowed to implement information flow according to the local policy. This dynamic view of information flow policies is supported by a concrete presentation of the security lattice, where the confidentiality levels are sets of principals, similar to access control lists. To take into account declassification, and more generally dynamic flow policies, we introduce a generalization of non-interference, that we call the non-disclosure policy, and we design a type and effect system for our language that enforces this policy.

1. Introduction

This paper addresses the issue of declassification in a language-based security approach. We are therefore more generally concerned with the confidentiality aspect of security. It has often been argued (see [11, 19, 31, 38] for instance) that the standard techniques used for access control are not enough to fully protect confidential information. Ideally, one would like to have a way of controlling how this information is used by subjects having the required clearance. Indeed, it is useless to restrict access to confidential information if one does not have some guarantee that the authorized subjects will not publicly disclose a significant part of this information. In other words, one should be interested in how information flows in a computer system, especially when the "subjects" are programs roaming over the web, that are not easily amenable to non-disclosure agreement.

Since Bell and La Padula and Denning's pioneering works [3, 11], the classical approach to secure information flow is to use a lattice of security levels (see for instance the survey [40] for the use of security lattices). The objects of a system are then labelled by security levels, and information is allowed to flow from one object to another if the source object has a lower confidentiality level than the target one. That is, the ordering relation on security levels determines the legal flows, and a program is secure if, roughly speaking, it does not set up illegal flows from inputs to outputs. This was first formally stated via a notion of *strong dependency* by Cohen in [9], and is also referred to as *non-interference* according to the terminology used by Goguen and Meseguer in [16].

[(*)] Work partially supported by the CRISS project of the ACI Sécurité Informatique. The first author is supported by the PhD scholarship POSI/SFRH/BD/7100/2001.

A lot of work has been devoted to the design of methods for analyzing information flow in programs (see for instance [2] for early references), even though this has not always been related to a security property like non-interference by a soundness result. Some of these methods consist in run-time checks, and have been criticized for various reasons, like for the fact that they generally suffer from the "label creep problem" (see [38]). More importantly, the failure of a run-time check can serve as a covert channel [11, 30]. As an alternative, static analysis methods have been developed for information flow. One can highlight the use of type systems, which started with the work of Volpano, Smith and Irvine [50]. Although they offer only approximate analysis, type systems have well-known advantages, like in preventing some programming errors at an early stage. Indeed, in the context of a security policy like the one provided by a flow lattice of security levels, it is an error not to comply with the policy, and a type safety result should in this case establish that well-typed programs are secure. Type systems for secure information flow have been designed for various languages (see e.g. [6, 10, 17, 33, 42, 43, 47, 50, 54], and further references in [38]), culminating with Jif (or JFlow, see [29]) and Flow CAML [41] as regards the size of the language. In this paper we shall base our study on Core ML [28, 51], a call-by-value λ-calculus extended with imperative constructs that we enrich with concurrent threads.

The classical non-interference property has been a matter of debate from various points of view (see [36]). A fundamental observation, which was made very early (e.g. in [18]), is that non-interference rules out, by its very definition, programs that deliberately *declassify* information from a confidential level to a more public one. These programs are quite common and very useful, making it hard to use non-inteference in practice. A standard example is a password checking procedure, which delivers to any user the result of comparing a submitted password with secret information contained in a database, thus leaking a bit of confidential information. Another one is encryption, where secret information is encoded into a ciphertext that can be read by anyone. Besides these classical security issues, some new circumstances where downgrading information is required arise in the context of networked systems. A typical example is the one of a service selling electronic information, like articles in a journal for instance. The contents of an article have to be kept secret from the client of such a service until he has paid for it, or has identified himself as a subscriber. Then the designer of the electronic purchase service has to provide a procedure that dynamically declassifies information, depending on informations provided by the client. To support this kind of programming, it would be useful for the programmer to have means to check that his code implements only the intended information flows. Our goal here is to provide the programmer with such a support.

The incompatibility of information flow security (in its current setting) with declassification is a challenging problem that has motivated a lot of work. We will comment on this later. This paper intends to contribute to its solution by proposing a turn on how the problem is set. Our view is that there are two different issues to be considered:

1. How may we *justify* that a program is allowed to declassify information, i.e. that it is not actually revealing "too much"?

2. How may we *accept* such programs in a language-based security setting, while still preserving some secure information flow property?

To illustrate this distinction, let us imagine for a while a password checking procedure that returns the root password, instead of a "yes/no" answer. Although the first is most probably wrong, the two programs are no different from the point of view of information flow: they both disclose information from the security level of the secret password database. One should therefore be able to reject the first by a semantical program analysis, while the second should be accepted in a language supporting downgrading facilities.

There is clearly a quantitative aspect to the first question. Indeed, some researchers who have studied it have proposed to quantify the amount of information that a program may leak, or to use complexity-theoretic or probabilistic arguments to establish that it is not feasible to exploit the allowed information leakage (see [7, 12, 21, 22, 46, 49], to mention just a few recent papers). Justifying declassification is a very interesting research problem which is by no means easy – in fact, justifying practically sound encryption mechanisms is a whole research domain – and seems to be beyond the reach of static analysis techniques. However, in our view, it is the *programmer* who has responsibility for solving the first question, of *what*, or *how much* information he intends to declassify in his program, whereas the (designer of the) *programming language* has to provide an answer to the second question above. This is the language design issue that we address.

Given that deliberately downgrading programs are validated by the programmer, the programming language should be as flexible as possible in expressing them. To this end, we introduce in our core language a programming construct for directly manipulating flow relations, namely a *flow declaration* construct $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ where F is a *flow policy*, i.e. a binary relation on security levels, and M is any expression of the language. The meaning is that M is executed in the context of the current flow policy *extended with* F, and after termination the current policy is restored, that is, the scope of F is M. For instance, if we have security levels $A(\mathit{lice})$ and $B(\mathit{ob})$, then – using the ML notation $\mathbf{!}\,x$ for dereferencing, and x_A, y_B for memory locations with confidentiality level A and B – a program like:

$$(\mathsf{flow}\ A \prec B\ \mathsf{in}\ y_B := \mathbf{!}\,x_A) \tag{1}$$

is legal, since in the context of the relation $A \prec B$, information is allowed to flow from A to B. With respect to the current flow policy, this is a declassification operation – unless, obviously, the current policy already says that information may flow from A to B. Moreover, this expression appears to read at the confidentiality level B for the rest of the program. Then the expression $y_B := (\mathsf{flow}\ A \prec B\ \mathsf{in}\ \mathbf{!}\,x_A)$ is also legal, and has the same meaning as the previous one. It should be clear, on the other hand, that a statement like $y_B := (\mathsf{flow}\ C \prec B\ \mathsf{in}\ \mathbf{!}\,x_A)$ is not legal, unless the current policy allows information to flow from level A to C (or B). Another example is

$$(\mathsf{flow}\ A \prec B\ \mathsf{in}\ M)\ \mathbf{;}\ (\mathsf{flow}\ B \prec C\ \mathsf{in}\ N)$$

that shows a way to achieve a kind of non-transitive flow relation (see [4, 34, 35]).

A similar construct for introducing flow policies exists in Flow Caml, but with an important difference: there it adds the flow relation F to the global security policy, whereas in our case the declaration is *local*. Such a construct has been mentioned in [44] under the name of "delegation", but it was not formally studied there. The operation $\mathsf{declassify}(M, \ell)$, that is used in some languages (see [29, 39] for instance) to downgrade the value of M to the confidentiality level ℓ, may be represented as $(\mathsf{flow}\ H \prec \ell\ \mathsf{in}\ M)$, where H is a security level that is higher (w.r.t. the current flow policy) than any other one. This interpretation of the declassification operation as a local operator on the flow policies seems to have never been pointed out. Furthermore, the flow declaration construct allows us to express more precise ways of declassifying, by specifying the levels from which information may flow.

We should point out here that, contrarily to most studies concerning security for functional languages (with the exception of [10]), we do not regard values as having a security level – there is no "secret 0" or "public 100" for instance. Our standpoint is that confidentiality levels are associated with the objects in which information is stored (or the channels on which it is communicated), like files, libraries, databases or references in the case of ML, and that what is to be controlled is the access to the object (typically read or write). This is consistent with most studies dealing with imperative languages, and has the pleasant consequence that the security type system is just a standard *type and effect system* [26], with flow constraints, where the security levels play the role of *regions* (as noted in [10]). In the construct ($\mathsf{flow}\ F\ \mathsf{in}\ M$), only the effects of M, and the resulting confidentiality level of the expression are concerned with the flow relation F; the final value of M, which has no confidentiality status, is not.

Once declassification is permitted in the language, a question is: what kind of security property do we have that takes declassification into account? And what means could we have to ensure that programs have this property? Not surprisingly, here the answer to this second question will be: a type and effect system. To answer the first one, we must find an alternative to non-interference. In the language-based security approach, and more specifically in concurrent settings, this property is often based on the small-step semantics, where one specifies transitions $(P, \mu) \rightarrow (P', \mu')$ between successive states of the program and the memory. This is well suited for our approach, where declassification is based on a dynamically evolving structure of the lattice of confidentiality levels. Indeed, the scope of a local flow policy is only a portion of the computation, and this has to be reflected in the semantics in order to state a security property. The way we do this is by decorating each transition with a label, recording the flow policy in the scope of which this particular step is performed:

$$(P, \mu) \underset{F}{\longrightarrow} (P', \mu') \tag{2}$$

The intuition is that, as regards information flow, the memory μ should be considered from the point of view of the current flow policy extended with F. That is, if F says that information is allowed to flow from level ℓ to level ℓ', what is read at level ℓ at this step may be regarded as having level ℓ'. Then our new confidentiality property, which is a generalization of non-interference that we call the *non-disclosure policy*, roughly says that a program P is secure if at each (small) step it satisfies non-interference *with respect to the flow policy that holds for this step.* More precisely, given that P performs the transition (2) above under the memory μ, and given that ν is a memory which only differs from μ regarding confidential information with respect to the current flow policy extended with F, then there is a transition $(P, \nu) \underset{F'}{\longrightarrow} (P'', \nu')$ from P under memory ν such that ν' is again equal to μ' as regards public information. Moreover, since we have to check this at every possible step, we shall also require that the programs P' and P'' have similar behaviours, from the confidentiality point of view. For instance, program (1) satisfies this property, since the reference x_A may be considered as having the level B under the flow relation $A \prec B$.

The main technical contribution of this paper is a proof that the type and effect system we design for our core language with declassification enforces the non-disclosure policy. We thus provide a direct generalization of the standard result regarding type systems for information flow. We shall start by introducing dynamic flow policies, which provide a way to deal with declassification that, up to our knowledge, has never been formally explored before. This is supported by a specific notion of security (pre-)lattice, which is also new. These will appear in the next section, where we present the language and its operational semantics. Then, in Section 3, we introduce our generalization of non-interference, namely the non-disclosure policy, that takes into account dynamic flow policies. A sound type and effect system is given for the language in Section 4, featuring the notion of a "termination effect" that allows us to deal with a strong notion of security, while still providing a flexible type system. Section 5 is devoted to the proof of type soundness. We then briefly discuss related work and conclude.

2. The Language

2.1 Security (pre-)lattices

As we said in the Introduction, we will use dynamically evolving flow relations for dealing with declassification. However, the security levels associated with references that appear in the expression M are the same as those in $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ – it is only the flow policy that changes. We are then faced with the issue of maintaining a (varying) lattice structure over a given set of security levels, since we shall use the meet and join operations in the type system, as usual. As a matter of fact, a "pre-lattice" structure turns out to be more convenient for our purpose. We call *pre-lattice* a pair $(\mathcal{L}, \preceq)$ where $\preceq$ is a preorder on $\mathcal{L}$, that is a reflexive and transitive (but not necessarily anti-symmetric) relation, such that for any $x, y \in \mathcal{L}$ there exist a meet $x \curlywedge y$ and a join $x \curlyvee y$ for x and y, satisfying

$$\begin{array}{cc}
x \curlywedge y \preceq x & x \preceq x \curlyvee y \\
x \curlywedge y \preceq y & y \preceq x \curlyvee y \\
z \preceq x \ \&\ z \preceq y \ \Rightarrow\ z \preceq x \curlywedge y & x \preceq z \ \&\ y \preceq z \ \Rightarrow\ x \curlyvee y \preceq z
\end{array}$$

Now we will define our security pre-lattices, where the set of security levels is fixed, and only the flow relations may vary. We assume given a set $\mathcal{P}$ of *principals*, ranged over by p, $q \ldots$ A *confidentiality level* is any set of principals, that is any subset ℓ of $\mathcal{P}$. The intuition is that whenever ℓ is the confidentiality label of an object, i.e. a reference, it represents a set of programs that are allowed to get the value of the object, i.e. to read the reference. From this point of view, a reference labelled $\mathcal{P}$ (also denoted $\bot$) is a most public one – every program is allowed to read it –, whereas the label $\emptyset$ (also denoted $\top$) indicates a secret reference, so secret that no one is allowed to read it. We can interpret the reverse inclusion of security levels as indicating allowed flows of information: if a reference x is labelled ℓ, and $\ell \supseteq \ell'$ then the value of x may be transferred to a reference y labelled ℓ', since the programs allowed to read this value from y were already allowed to read it from x.

The dynamically varying information flow policies are determined by relations on principals. This is slightly different from what we informally presented in the Introduction, where, for simplicity, a flow relation was assumed to relate security levels. However, as we shall see, a relation on principals induces a preorder on security levels. A *flow policy* is a binary relation over $\mathcal{P}$. We let F, $G \ldots$ range over such relations. A pair $(p, q) \in F$ is to be understood as "information may flow from principal p to principal q", that is, more precisely, "*everything that principal p is allowed to read may also be read by principal q*". We must point out here that, since we are dealing with confidentiality (and not integrity) a flow policy will only affect the reading capabilities of programs (and not their writing capabilities). As a member of a flow policy, a pair (p, q) will most often be written $p \prec q$. We denote, as usual, by F^* the preorder generated by F (that is, the reflexive and transitive closure of F). Then we introduce the *preorder on confidentiality levels* determined by the flow relation F:

$$\ell \preceq_F \ell' \quad \Leftrightarrow_{\mathrm{def}} \quad \forall q \in \ell'.\ \exists p \in \ell.\ p\, F^*\, q$$

which is denoted $\preceq$ (instead of $\supseteq$) when $F = \emptyset$. We shall use without notice the fact that

$$G \subseteq F\ \&\ \ell \preceq_G \ell' \ \Rightarrow\ \ell \preceq_F \ell'$$

It is not difficult to see that the preorder $\preceq_F$ induces a pre-lattice structure on the set of confidentiality levels, where a meet is simply the union, and a join of ℓ and ℓ' is

$$\{\, q \mid \exists p \in \ell.\ \exists p' \in \ell'.\ p\, F^*\, q\ \&\ p'\, F^*\, q \,\}$$

$$\begin{array}{rcll} M,\ N \ldots \in \mathcal{E}xpr & ::= & W \mid (\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ N') \mid (MN) & \textit{expressions} \\ & \mid & M\,\mathbf{;}\,N \mid (\mathsf{ref}_{\ell,\theta}\,N) \mid (\mathbf{!}\,N) \mid (M := N) & \\ & \mid & (\mathsf{thread}\,M) \mid (\mathsf{flow}\ F\ \mathsf{in}\ M) & \\ W \in \mathcal{W} & ::= & V \mid \varrho x W & \textit{pseudo-values} \\ V \in \mathcal{V}al & ::= & x \mid u_{\ell,\theta} \mid \lambda x M \mid tt \mid f\!f \mid () & \textit{values} \end{array}$$

Figure 1: Syntax

This observation justifies the following definition.

Definition (Security Pre-Lattices) 2.1. *A* confidentiality level *is any subset ℓ of the set $\mathcal{P}$ of* principals. *Given a* flow policy *$F \subseteq \mathcal{P} \times \mathcal{P}$, the confidentiality levels are pre-ordered by the relation*

$$\ell \preceq_F \ell' \quad \Leftrightarrow_{\text{def}} \quad \forall q \in \ell'.\ \exists p \in \ell.\ p\,F^*\,q$$

The meet *and* join*, w.r.t. F, of two security levels ℓ and ℓ' are respectively given by $\ell \cup \ell'$ and*

$$\ell \curlyvee_F \ell' = \{\, q \mid \exists p \in \ell.\ \exists p' \in \ell'.\ p\,F^*\,q\ \&\ p'\,F^*\,q \,\}$$

2.2 Syntax and operational semantics

The language is a call-by-value λ-calculus extended with the imperative constructs of ML, conditional branching and boolean values. We also introduce the possibility of dynamically creating concurrent threads, and of declassifying computations. Clearly, the latter is the main novelty, and one could probably deal with other programming paradigms in a similar way, by adding local flow declarations. The syntax is given in Figure 1, where x is any variable, F is any flow policy that is most often written as a list $p_1 \prec q_1, \ldots, p_n \prec q_n$ of pairs of principals, and $u_{\ell,\theta}$ is a triple made of a memory address u – or location, or *reference* –, a type θ (see Section 4 below) and a label ℓ which is a confidentiality level. The label ℓ, most often written $p_1, \ldots, p_n$ instead of $\{p_1, \ldots, p_n\}$, is similar to an access control list. We use locations explicitly decorated with types and confidentiality labels for the purpose of the proof of type soundness. However, as we shall see, these annotations do not have any role in the operational semantics, and therefore they do not have to appear in an implementation (although in a mobile code setting, one would like to keep these annotations in order to perform security checks when loading a piece of code). We denote by $\mathsf{loc}(M)$ the set of decorated locations occurring in M. These addresses are regarded as providing the *inputs* of the expression M.

For typing reasons, we do not regard sequential composition as a derived construct, and we do not regard the imperative constructs ref, $\mathbf{!}$ and $:=$ as first-class functions. Indeed, the typing of $(\mathsf{ref}_{\ell,\theta}\,N)$ will generally be different from the typing of $(\lambda x(\mathsf{ref}_{\ell,\theta}\,x)N)$ for instance. Applying the construct $\mathsf{ref}_{\ell,\theta}$ to a value V creates a new reference with initial value V. Here, as we shall see, the value is assumed to be of type θ, and the confidentiality level ℓ will be assigned to the created reference. While the type θ could probably be inferred, as in ML, it seems natural for security purposes to explicitly assign a confidentiality level to the created reference. In a pure type and effect inference approach, with an unlabelled ref function, we would only get constraints that this level should satisfy. The construct $\varrho x W$, which is a binder for the variable x in W, provides a way to deal with recursive values. As a matter of fact, given that the set of values is quite limited in our core language, the only interesting case is that of recursive functions, i.e. $\varrho f \lambda x M$, which could

$$
\begin{array}{rcll}
((\mathsf{if}\ tt\ \mathsf{then}\ M\ \mathsf{else}\ N),\mu) & \underset{\emptyset}{\rightarrow} & (M,\mu) & \\
((\mathsf{if}\ f\!f\ \mathsf{then}\ M\ \mathsf{else}\ N),\mu) & \underset{\emptyset}{\rightarrow} & (N,\mu) & \\
((\lambda xMV),\mu) & \underset{\emptyset}{\rightarrow} & (\{x \mapsto V\}M,\mu) & \\
(V\,\mathbf{;}\,N,\mu) & \underset{\emptyset}{\rightarrow} & (N,\mu) & \\
((\mathsf{ref}_{\ell,\theta}\,V),\mu) & \underset{\emptyset}{\rightarrow} & (u_{\ell,\theta},\mu \cup \{u_{\ell,\theta} \mapsto V\}) & u \text{ fresh for } \mu \\
((\mathbf{!}\,u_{\ell,\theta}),\mu) & \underset{\emptyset}{\rightarrow} & (V,\mu) & \mu(u_{\ell,\theta}) = V \\
((u_{\ell,\theta} := V),\mu) & \underset{\emptyset}{\rightarrow} & ((),\mu[u_{\ell,\theta} := V]) & \\
(\varrho xW,\mu) & \underset{\emptyset}{\rightarrow} & (\{x \mapsto \varrho xW\}W,\mu) & \\
((\mathsf{flow}\ F\ \mathsf{in}\ V),\mu) & \underset{\emptyset}{\rightarrow} & (V,\mu) &
\end{array}
$$

$$
\frac{(M,\mu) \underset{F}{\rightarrow} (M',\mu')}{(\mathbf{E}[M],\mu) \xrightarrow[F \cup \lceil \mathbf{E} \rceil]{} (\mathbf{E}[M'],\mu')}
\qquad
\frac{}{(\mathbf{E}[(\mathsf{thread}\,M)],\mu) \underset{\emptyset}{\rightarrow} ((\mathbf{E}[()] \parallel (\mathsf{flow}\ \lceil \mathbf{E} \rceil\ \mathsf{in}\ M)),\mu)}
$$

$$
\frac{(P,\mu) \underset{F}{\rightarrow} (P',\mu')}{((P \parallel Q),\mu) \underset{F}{\rightarrow} ((P' \parallel Q),\mu')}
\qquad
\frac{(P,\mu) \underset{F}{\rightarrow} (P',\mu')}{((Q \parallel P),\mu) \underset{F}{\rightarrow} ((Q \parallel P'),\mu')}
$$

Figure 2: Operational Semantics

be denoted ($\mathsf{let\ rec}\ f = \lambda xM\ \mathsf{in}\ f$) in an ML-like notation. We denote by loop the expression ϱxx, and we may use the following standard abbreviation:

$$(\mathsf{while}\ M\ \mathsf{do}\ N) =_{\mathrm{def}} (\varrho y \lambda x(\mathsf{if}\ M\ \mathsf{then}\ N\,\mathbf{;}\,(yx)\ \mathsf{else}\ x)())$$

We let $\mathsf{fv}(M)$ be the set of variables occurring free in M, and we denote by $\{x \mapsto W\}M$ the capture-avoiding susbtitution of W for the free occurrences of x in M, where $W \in \mathcal{W}$. The evaluation relation is a transition relation between configurations of the form (P,μ) where P is a *process*, written according to the following syntax:

$$P,\ Q \ldots \in \mathcal{P}roc\ ::=\ M \ \mid\ (P \parallel Q)$$

and μ, the *memory* (or *heap*), is a mapping from a finite set $\mathsf{dom}(\mu)$ of decorated references to values. The operation of updating the value of a reference in the memory is denoted, as usual, $\mu[u_{\ell,\theta} := V]$. We say that the name u is *fresh for* μ if $v_{\ell,\theta} \in \mathsf{dom}(\mu) \ \Rightarrow\ v \neq u$. To define the operational semantics, we introduce evaluation contexts:

$$
\begin{array}{rcl}
\mathbf{E} & ::= & \Box \ \mid\ \mathbf{F}[\mathbf{E}] \ \mid\ (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}) \\
\mathbf{F} & ::= & (\mathsf{if}\ \Box\ \mathsf{then}\ M\ \mathsf{else}\ N) \ \mid\ (\Box N) \ \mid\ (V\Box) \\
 & \mid & \Box\,\mathbf{;}\,N \ \mid\ (\mathsf{ref}_{\ell,\theta}\,\Box) \ \mid\ (\mathbf{!}\,\Box) \ \mid\ (\Box := N) \ \mid\ (V := \Box)
\end{array}
$$

and we denote by $\lceil \mathbf{E} \rceil$ the flow policy enforced by the context $\mathbf{E}$. This is defined as follows:

$$
\begin{array}{rcl}
\lceil \Box \rceil & = & \emptyset \\
\lceil \mathbf{F}[\mathbf{E}] \rceil & = & \lceil \mathbf{E} \rceil \\
\lceil (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}) \rceil & = & F \cup \lceil \mathbf{E} \rceil
\end{array}
$$

The labelled transition rules are given in Figure 2. As one can see, the transitions do not depend on the types and labels of memory locations. Observe also that the flow label F of the transitions plays no role in determining the resulting configuration. To evaluate $(\mathsf{flow}\ F\ \mathsf{in}\ M)$, we simply evaluate M, until termination (that is, when M is a value). Then $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ is operationally the same as M, and the context $(\mathsf{flow}\ F\ \mathsf{in}\ [])$ should disappear at compile time. We denote somewhat abusively by $\rightarrow$ the relation given by

$$(P,\mu) \rightarrow (P',\mu') \quad \Leftrightarrow_{\mathrm{def}} \quad \exists F.\ (P,\mu) \xrightarrow[F]{} (P',\mu')$$

and we let $\overset{*}{\rightarrow}$ denote the reflexive and transitive closure of the relation $\rightarrow$. We shall actually only consider transitions from *well-formed* configurations: a configuration (P,μ) is well-formed if $\mathsf{loc}(P) \subseteq \mathsf{dom}(\mu)$ and for any $u_{\ell,\theta} \in \mathsf{dom}(\mu)$ we have $\mathsf{loc}(\mu(u_{\ell,\theta})) \subseteq \mathsf{dom}(\mu)$. It is easy to see that well-formedness is preserved by transitions.

To conclude this section, we introduce another kind of transitions which will be useful for the proof of type soundness. These transitions, denoted $(M,\mu) \xrightarrow[F]{N} (M',\mu')$, should be read as follows: the expression M, in the context of the memory μ, performs a step, assuming the local flow policy F, and resulting in the new expression M' and memory μ', while possibly spawning the expression N as a new thread to execute (with $N = ()$ if M actually does not spawn any thread). This formalizes the evaluation steps of an expression M as the "main thread". Formally, this is defined as $\xrightarrow[F]{}$, with $N = ()$, except for:

$$\frac{}{(\mathbf{E}[(\mathsf{thread}\, N)],\mu) \xrightarrow[\emptyset]{(\mathsf{flow}\ \lceil\mathbf{E}\rceil\ \mathsf{in}\ N)} (\mathbf{E}[()],\mu)}$$

The following lemma relates these transitions with the ones that we have used to describe the operational semantics:

Lemma 2.2.
(i) *If* $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ *then either* $N = ()$ *and* $(M,\mu) \xrightarrow[F]{} (M',\mu')$ *or* $(M,\mu) \xrightarrow[F]{} ((M' \parallel N),\mu')$.

(ii) *If* $(M,\mu) \xrightarrow[F]{} (P,\mu')$ *then either* P *is an expression and* $(M,\mu) \xrightarrow[F]{()} (P,\mu')$ *or* $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ *for some* M' *and* N *such that* $P = (M' \parallel N)$.

Next we show a simple but crucial property stating that, if the evaluation of an expression M differs in the context of two distinct memories while not creating two distinct references, this is because M is performing a dereferencing operation, which yields different results depending on the memory. Apart from non-deterministically choosing new references, this is the only way for computations of expressions to split.

Lemma 2.3. *If* $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ *and* $(M,\nu) \xrightarrow[F']{N'} (M'',\nu')$ *with* $M' \neq M''$ *and* $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$, *then* $N = () = N'$ *and there exist* $\mathbf{E}$ *and* $u_{\ell,\theta}$ *such that* $F = \lceil\mathbf{E}\rceil = F'$, $M = \mathbf{E}[(!u_{\ell,\theta})]$, *and* $M' = \mathbf{E}[\mu(u_{\ell,\theta})]$, $M'' = \mathbf{E}[\nu(u_{\ell,\theta})]$ *with* $\mu' = \mu$ *and* $\nu' = \nu$.

Proof: easy induction on the proof of the transition $(M,\mu) \xrightarrow[F]{N} (M',\mu')$. ❑

3. The non-disclosure policy

In this section we introduce our security property. The definitions that follow could be formulated in an abstract way, that is for any semantic framework consisting of a given labelled transition system, where the transitions have, as above, the form

$$(P,\mu) \xrightarrow[F]{} (P',\mu')$$

However, we instantiate here the definitions with the notion of a process that we have introduced in the previous section.

To state the security property, we use, as it is standard, a notion of *memory equality* relative to a given confidentiality level: two memories μ and ν are equal up to level ℓ if they assign the same value to every location with security level lower than ℓ (this is sometimes referred to as "low equality" of memories). However, there is an implicit parameter in this notion, which is the flow relation used to determine that a level is lower than ℓ. Since the flow policy is not fixed in our setting, we make it explicit in the notion of memory equality. Furthermore, we will compare two memories only with respect to the references they share. The low equality of memories is thus defined:

$$\mu \simeq^{F,\ell} \nu \quad \Leftrightarrow_{\text{def}} \quad \forall u_{\ell',\theta} \in \mathsf{dom}(\mu) \cap \mathsf{dom}(\nu).\ \ell' \preceq_F \ell \ \Rightarrow\ \mu(u_{\ell',\theta}) = \nu(u_{\ell',\theta})$$

This relation is not transitive, but it is reflexive and symmetric. We shall use without notice the fact that

$$G \subseteq F \ \&\ \mu \simeq^{F,\ell} \nu \ \Rightarrow\ \mu \simeq^{G,\ell} \nu$$

Our security property is defined in terms of *bisimulations* (see [6, 14, 37, 42] for the use of bisimulations in stating security properties, and [25] for a review of various other approaches). Bisimulations are relations on states of transition systems, that relate two states whenever any transition of either of these states can be "matched" by a transition of the other. Following [37], here we shall make use of this notion in a slightly non-standard way, since we shall call "bisimulation" a relation between processes P, rather than configurations (P,μ). Moreover, such a relation is parameterized upon a flow policy G, which is the current flow relation.

Definition (Bisimulation) 3.1. *A* (G,ℓ)-bisimulation *is a symmetric relation* $\mathcal{R}$ *on processes such that if*

$$P\,\mathcal{R}\,Q \ \&\ (P,\mu) \xrightarrow[F]{} (P',\mu') \ \&\ \mu \simeq^{F\cup G,\ell} \nu \ \&\ u_{\ell',\theta} \in \mathsf{dom}(\mu'-\mu) \ \Rightarrow\ u \textit{ is fresh for } \nu$$

then there exist Q' *and* ν' *such that*

$$(Q,\nu) \xrightarrow{*} (Q',\nu') \ \&\ P'\,\mathcal{R}\,Q' \ \&\ \mu' \simeq^{G,\ell} \nu' \ \&\ \mathsf{dom}(\nu'-\nu) \subseteq \mathsf{dom}(\mu'-\mu)$$

It is implicit in this definition that the configurations (P,μ) and (Q,ν) are well-formed. This implies in particular that $\mathsf{loc}(P) \subseteq \mathsf{dom}(\mu)$ and $\mathsf{loc}(Q) \subseteq \mathsf{dom}(\nu)$. The definition says that the transition

$$(P,\mu) \xrightarrow[F]{} (P',\mu')$$

has to be matched by a transition from (Q,ν), whenever $P\,\mathcal{R}\,Q$, and the memories μ and ν satisfy some conditions. The first one, namely that $\mu \simeq^{F\cup G,\ell} \nu$, can be interpreted as follows: since P is performing a transition within the scope of the current flow policy G extended with the local flow relation F, it is allowed to read references from the input memory according to the policy $F \cup G$. That is, in the input memories μ and ν, "low" means less than ℓ with respect to $F \cup G$. The condition "$u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu) \;\Rightarrow\; u$ is fresh for ν" simply means that if P creates a new reference, then we assume that the created name does not conflict with other names under consideration. Indeed, this new name can always be chosen so that it satisfies this constraint. The conclusion is that the state (Q, ν) should be able to evolve, possibly in several steps, into a configuration (Q', ν'), satisfying the following constraints: first, $P' \mathcal{R} Q'$ means that no mismatch should occur in future computations. Next, we require $\mu' \doteq^{G,\ell} \nu'$, thus considering the output μ' of the step $(P, \mu) \underset{F}{\longrightarrow} (P', \mu')$ from the point of view of the flow policy that is restored after it, that is G. This means in particular that the local flow policy F does not affect the level of references from the writing, or output point of view (recall that $p \prec q$ means that "everything that principal p is allowed to read may also be read by principal q"). Finally $\mathsf{dom}(\nu' - \nu) \subseteq \mathsf{dom}(\mu' - \mu)$ ensures that Q only creates a reference to match P's transition if P itself has created a reference, in which case we require the new location names to be the same. Without this condition, an expression like

$$(\mathsf{ref}_L(!\, u_H)) \tag{3}$$

would be secure. Indeed, one could argue that there is no information flow here, since we are creating a reference that does not yet exist, but the point is that, if we run this program in the context of two memories μ and ν such that $\mu(u_H) \neq \nu(u_H)$, we would have to count on the non-determinism of the semantics, as regards reference creation, to claim that such a program is secure. We think it is better to rule out such a program, even though we are here assuming in a sense that an "attacker" has the power of magically knowing the name of public references that are created – this goes beyond the power of programs, since they do not have access to the syntax of references. The reader may have also noticed that there is no condition on the flow policy of the matching moves for Q. This is because we wish to consider all the *pure* programs, written without $(!\, N)$ and $(M := N)$, to be bisimilar.

As mentioned above, the notion of bisimulation we use is stronger than the standard one, since if the transition $(P, \mu) \underset{F}{\longrightarrow} (P', \mu')$ is matched by $(Q, \nu) \xrightarrow{*} (Q', \nu')$, we restart the bisimulation game by comparing the processes P' and Q', in the context of any new low equal memories, rather than the configurations (P', μ') and (Q', ν'). This allows us to restore a more restrictive flow relation after a local flow relation has been used, as in

$$(\mathsf{flow}\ H \prec L\ \mathsf{in}\ v_L := !\, u_H)\,;\, w_L := !\, u'_H \tag{4}$$

where the second assignment implements an illegal flow (denoting simply by H, $L\ldots$ a singleton security level $\{H\}$, $\{L\}\ldots$ assigned to a reference). Defining bisimulations over processes, rather than between configurations, also allows us to detect an illegal flow in the program

$$(\mathsf{if}\ !\, w_X\ \mathsf{then}\ (\mathsf{if}\ !\, w_X\ \mathsf{then}\ ()\ \mathsf{else}\ v_L := !\, u_H)\ \mathsf{else}\ ()) \tag{5}$$

This demanding definition for bisimulations seems also appropriate for dealing with a mobile code scenario, where the shared memory of a system of threads can be modified by incoming code.

Remarks and Notation 3.2.

(i) *For any G and ℓ there exists a (G, ℓ)-bisimulation, like for instance the set $\mathcal{V}al \times \mathcal{V}al$ of pairs of values.*

(ii) *The union of a family of (G, ℓ)-bisimulations is a (G, ℓ)-bisimulation. Consequently, there is a largest (G, ℓ)-bisimulation, which we denote $\approxeq^{G,\ell}$. This is the union of all such bisimulations.*

One should observe that $\bowtie^{G,\ell}$ is a *partial* equivalence relation. That is, this relation is not reflexive. Indeed, a process which is not bisimilar to itself, like $v_L := \,!\, u_H$ if $H \not\preceq_G L$, is not secure. As in [37], our definition states that a program is secure if it is bisimilar to itself:

Definition (The Non-Disclosure Policy) 3.3. *A process P satisfies the* non-disclosure policy *(or is* secure *from the confidentiality point of view) with respect to the flow policy G if it satisfies $P \bowtie^{G,\ell} P$ for any ℓ. We then write $P \in \mathcal{ND}(G)$.*

It is easily seen that the set $\mathcal{ND}(G)$ is non-empty. For instance, any value is secure. An important property of our notion of security is that if an expression M does not violate the current flow policy G extended with a local policy F, then the expression $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ is secure with respect to the current flow relation:

Proposition 3.4.

$M \in \mathcal{ND}(F \cup G) \;\Rightarrow\; (\mathsf{flow}\ F\ \mathsf{in}\ M) \in \mathcal{ND}(G)$

Proof: it is easy to see that if $\mathcal{R}$ is a $(F \cup G, \ell)$-bisimulation, then the relation

$$\begin{array}{rl} & \{\, ((\mathsf{flow}\ F\ \mathsf{in}\ M), (\mathsf{flow}\ F\ \mathsf{in}\ N)) \mid M \,\mathcal{R}\, N \,\} \\ \cup & \{\, (V, (\mathsf{flow}\ F\ \mathsf{in}\ N)) \mid V \,\mathcal{R}\, N \ \&\ V \in \mathcal{V}al \,\} \\ \cup & \{\, ((\mathsf{flow}\ F\ \mathsf{in}\ M), V) \mid M \,\mathcal{R}\, V \ \&\ V \in \mathcal{V}al \,\} \end{array}$$

is a (G, ℓ)-bisimulation. ❏

Then for instance a program like the one of example (1) is secure. Another property is that security is compatible with parallel composition:

Proposition (Compositionality) 3.5.

$P \in \mathcal{ND}(G)\ \&\ Q \in \mathcal{ND}(G) \;\Rightarrow\; (P \parallel Q) \in \mathcal{ND}(G)$

Proof: it is easy to see that if $\mathcal{R}$ is a (G, ℓ)-bisimulation, then the relation

$$\{\, ((P \parallel Q), (P' \parallel Q')) \mid P \,\mathcal{R}\, P' \ \&\ Q \,\mathcal{R}\, Q' \,\}$$

is a (G, ℓ)-bisimulation. ❏

Our non-disclosure policy generalizes the usual non-interference property for sequential programs (without declassification). To see this point, let us first recall that the latter is based on the "big-step" semantics of programs, that is on the relation $(P, \mu) \Rightarrow \mu'$ that a program P establishes from an initial state of the memory μ to the final state μ'. Namely, P is *non-interfering* if $(P, \mu) \Rightarrow \mu'$ and $(P, \nu) \Rightarrow \nu'$, for μ and ν that differ only regarding confidential information, implies that also μ' and ν' are equal as regards public information, that is:

$$(P, \mu) \Rightarrow \mu' \ \&\ (P, \nu) \Rightarrow \nu' \ \&\ \mu \doteq^{G,\ell} \nu \;\Rightarrow\; \mu' \doteq^{G,\ell} \nu'$$

Let us denote for a while by $\mathcal{DE}xpr$ the set of expressions written without using thread, flow and ref, and let us show that the expressions in $\mathcal{DE}xpr$ satisfying the non-disclosure policy with respect to a given flow policy G are non-interfering. The big-step semantics for expressions in $\mathcal{DE}xpr$ can be defined as follows:

$$(M, \mu) \Rightarrow \mu' \quad \Leftrightarrow_{\mathrm{def}} \quad \exists V \in \mathcal{V}al.\ (M, \mu) \xrightarrow{*} (V, \mu')$$

It is easy to see that the evaluation mechanism is deterministic for $M \in \mathcal{DE}\mathit{xpr}$, and that if $(M,\mu) \Rightarrow \mu'$ then $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$. Now assume that $M \in \mathcal{DE}\mathit{xpr} \cap \mathcal{ND}(G)$, $(M,\mu) \xrightarrow{*} (V,\mu')$ and $(M,\nu) \xrightarrow{*} (V',\nu')$ with $\mu \bumpeq^{G,\ell} \nu$. Then there exist M' and ν'' such that $(M,\nu) \xrightarrow{*} (M',\nu'')$, $V \mathrel{⌑}^{G,\ell} M'$ and $\mu' \bumpeq^{G,\ell} \nu''$. Since M is deterministic, we have $(M',\nu'') \xrightarrow{*} (V',\nu')$, and from (V,μ') there must be a sequence of transitions matching the move from (M',ν'') to (V',ν'). This sequence must be empty, and we then have $\mu' \bumpeq^{G,\ell} \nu'$.

We can use the standard bisimulation technique to show that a process P is secure. Namely, for any ℓ we exhibit a symmetric relation $\mathcal{R}$ that contains the pair (P,P) and is a (G,ℓ)-bisimulation. To prove that a process is not secure may be more difficult. We may, in the case where P is actually an expression of $\mathcal{DE}\mathit{xpr}$, show that P does not satisfy the non-interference property. This applies to some of the examples given below. However, we cannot use this argument for expressions that spawn threads or use the declassification facility. We shall come back to this point at the end of Section 5, where we further discuss the non-disclosure policy and non-interference.

Now let us see some examples. We assume given two principals H and L, and a current flow relation G consisting of the pair $L \prec H$. We shall denote references with security levels $\{H\}$ or $\{L\}$ simply by u_H or v_L (leaving out the type), as usual. Since, as we have just seen, the non-disclosure policy implies the standard non-interference property for expressions of $\mathcal{DE}\mathit{xpr}$, it is obvious that the standard examples of explicit (or direct) and implicit (or indirect) flow, namely:

$$v_L := \mathbf{!}\, u_H \tag{6}$$

$$(\mathsf{if}\ \mathbf{!}\, u_H\ \mathsf{then}\ v_L := \mathit{tt}\ \mathsf{else}\ v_L := \mathit{ff}) \tag{7}$$

do not satisfy the non-disclosure policy, whereas these programs are secure in the context of the flow declaration $(\mathsf{flow}\ H \prec L\ \mathsf{in}\ \square)$. Since we follow a bisimulation approach to security, we also reject termination leaks, like for instance

$$(\mathsf{if}\ \mathbf{!}\, u_H\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{loop})\,\mathbf{;}\, v_L := \mathit{tt} \tag{8}$$

where writing at level L depends on reading at level H (we refer to [2, 6, 17, 38, 42, 47] for discussions about this kind of leaks). Another example of a termination leak is

$$((\mathbf{!}\, u_H)())\,\mathbf{;}\, v_L := \mathit{tt} \tag{9}$$

Indeed, the value of the reference u could be $\lambda y \lambda x x$ or $\lambda y\, \mathsf{loop}$. Similarly, there is a termination leak in

$$(\lambda x(x())(\mathbf{!}\, u_H))\,\mathbf{;}\, v_L := \mathit{tt} \tag{10}$$

since the value of the reference u might be $\lambda y y$ or $\lambda y\, \mathsf{loop}$. We shall put a constraint on sequential composition in the type system to rule out such programs. However, this constraint will not be as strict as "no low write after a high read", because we would like to accept for instance the following (secure) program:

$$(w_H := \mathbf{!}\, u_H)\,\mathbf{;}\,(v_L := \mathit{tt}) \tag{11}$$

Regarding the flow declaration construct, we notice for instance that the program

$$v_L := (\mathsf{flow}\ H \prec L\ \mathsf{in}\ \mathbf{!}\, u_H) \tag{12}$$

which is essentially the same as example (1), is secure, as well as

$$(\mathsf{if}\ \mathbf{!}\, u_H\ \mathsf{then}\ w_H := \mathit{tt}\ \mathsf{else}\ ())$$

whereas

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ (\mathsf{flow}\ H \prec L\ \mathsf{in}\ v_L := tt)\ \mathsf{else}\ ()) \tag{13}$$

is not. The reason is that the flow declaration $H \prec L$ is a way of giving (temporarily) the same reading capabilities to the principals H and L, whereas it does not affect the writing capabilities of a program. A type system for information flow has to take this into account.

We conclude this section with some technical properties that will be used in the type soundness proof. We begin with an operational property relative to the low equality of memories. Namely, we show that any expression has, in the context of low equal memories, similar transitions:

Lemma 3.6. *If $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ and $\mu \doteq^{G,\ell} \nu$ such that $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν, then there exist M'' and ν' such that $(M,\nu) \xrightarrow[F]{N} (M'',\nu')$ with $\mu' \doteq^{G,\ell} \nu'$ and $\mathsf{dom}(\nu' - \nu) = \mathsf{dom}(\mu' - \mu)$.*

Proof: by induction on the proof of the transition $(M,\mu) \xrightarrow[F]{N} (M',\mu')$. In most cases, this transition does not depend on (and does not modify) the memory μ, and we may let $M'' = M'$ and $\nu' = \nu$. If M creates a reference, that is $M = \mathbf{E}[(\mathsf{ref}_{\ell',\theta}\, V)]$, then $M' = \mathbf{E}[u_{\ell',\theta}]$, $F = \lceil \mathbf{E} \rceil$ and $\mu' = \mu \cup \{u_{\ell',\theta} \mapsto V\}$. Since u is fresh for ν, we have $(M,\nu) \xrightarrow[F]{N} (M', \nu \cup \{u_{\ell',\theta} \mapsto V\})$. If $M = \mathbf{E}[(!u_{\ell',\theta})]$ then $M' = \mathbf{E}[\mu(u_{\ell',\theta})]$, $F = \lceil \mathbf{E} \rceil$ and $\mu' = \mu$. Since the configurations we consider are assumed to be well-formed, we have $(M,\nu) \xrightarrow[F]{N} (\mathbf{E}[\nu(u_{\ell',\theta})], \nu)$. If $M = \mathbf{E}[(u_{\ell',\theta} := V)]$ then $M' = \mathbf{E}[()]$, $F = \lceil \mathbf{E} \rceil$ and $\mu' = \mu[u_{\ell',\theta} := V]$. In this case $(M,\nu) \xrightarrow[F]{N} (M', \nu[u_{\ell',\theta} := V])$. All the other cases are immediate. ❑

Now we define “operationally high” processes, those that do not modify the low part of the memory:

Definition (Operationally High Processes) 3.7. *A set $\mathcal{H}$ of processes is said to be a set of* operationally (G,ℓ)-high *processes if the following holds for any $P \in \mathcal{H}$:*

$$(P,\mu) \rightarrow (P',\mu') \ \Rightarrow\ \mu' \doteq^{G,\ell} \mu \ \&\ P' \in \mathcal{H}$$

One may wonder why we do not simply define a high process as one that satisfies

$$(P,\mu) \xrightarrow{*} (P',\mu') \ \Rightarrow\ \mu' \doteq^{G,\ell} \mu$$

This is because the expression of Example (5) for instance would be high, and Lemma 3.8 below would therefore not hold. One should remark that there are sets satisfying Definition 3.7. For instance, any value is a high process. Moreover, the union of a family of such $\mathcal{H}$'s is a set of (G,ℓ)-high processes, and so there exists the largest such set, which we denote by $\mathcal{H}_{G,\ell}$, and call the set of *operationally (G,ℓ)-high* processes. Notice that

$$G \subseteq F \ \Rightarrow\ \mathcal{H}_{F,\ell} \subseteq \mathcal{H}_{G,\ell}$$

and that a process which is $\top$-high never modifies the memory (which does not mean that it performs no writes).

Lemma and Notation 3.8. *If $\mathcal{H}$ is a set of (G,ℓ)-high processes, then the relation $\mathcal{H} \times \mathcal{H}$ is a (G,ℓ)-bisimulation. We denote by $\mathrel{\rangle\!\langle}^{G,\ell}$ the (G,ℓ)-bisimulation $\mathcal{H}_{G,\ell} \times \mathcal{H}_{G,\ell}$.*

PROOF: let $P, Q \in \mathcal{H}$ and $(P,\mu) \xrightarrow[F]{} (P',\mu')$ with $\mu \bumpeq^{F\cup G,\ell} \nu$, such that $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν. Then we have $\mu' \bumpeq^{G,\ell} \mu$ and $P' \in \mathcal{H}$, and therefore also $\mu' \bumpeq^{G,\ell} \nu$ since $\mathsf{dom}(\mu' - \mu) \cap \mathsf{dom}(\nu) = \emptyset$. Therefore the transition $(Q,\nu) \xrightarrow{*} (Q,\nu)$ matches $(P,\mu) \xrightarrow[F]{} (P',\mu')$. ❏

The following is easy to see:

LEMMA 3.9. $M, N_0, N_1 \in \mathcal{H}_{G,\ell} \;\Rightarrow\; (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) \in \mathcal{H}_{G,\ell}$

4. The type and effect system

The types we use in the type and effect system are quite standard. Namely, a reference type $\theta\,\mathsf{ref}_\ell$ records the type θ of values the reference contains, as well as the "region" ℓ where it is created. Here this is the *confidentiality level* of the reference, indicating who is allowed to read it. A function type records the *latent effect* [26] of a function of that type, which is the effect the function may have when applied to an argument. It also records the "latent flow relation", which is assumed to hold when the function is applied to an argument. The syntax of types is

$$\tau,\ \sigma,\ \theta \ldots \quad ::= \quad t \;\mid\; \mathsf{bool} \;\mid\; \mathsf{unit} \;\mid\; \theta\,\mathsf{ref}_\ell \;\mid\; (\tau \xrightarrow[F]{s} \sigma)$$

where t is any type variable and s is any "security effect" – see below. The judgements of the type and effect system have the form

$$G; \Gamma \vdash M : s, \tau$$

where G is a flow relation, Γ is a typing context, assigning types to variables, s is a security effect, that is a triple (ℓ_0, ℓ_1, ℓ_2) of confidentiality levels, and τ is a type. The intuition is:

- G is the current flow policy that is in force when evaluating M;
- ℓ_0, also denoted by $s.\mathsf{r}$, is the *reading effect*, that is an upper bound (up to the current flow relation) of the security levels of the references the expression M may read. This may be regarded as the security level, or more precisely *the confidentiality level of the expression M*;
- ℓ_1, also denoted $s.\mathsf{w}$, is the *writing effect*, that is a lower bound (w.r.t. the relation $\preceq$) of the level of references that the expression M may update;
- ℓ_2, also denoted $s.\mathsf{t}$, is an upper bound (w.r.t. the current flow relation) of some of the levels of dereferencing the expression M may perform. This is used to avoid termination leaks, and therefore we call this the *termination effect* – although the intention is not to guarantee termination.

With respect to the various type systems for information flow, the main novelty here is the G parameter in the typing context, which is used to relax the constraints on how information may flow in a piece of code to type (such a flow relation in the typing context also appears, under the name of a "hierarchy", in [44]). The termination effect is similar to the "guard level" of [6] and to the "running time level" of [42]. According to the intuition above, in the type system the reading and termination levels will be composed in a covariant way, whereas the writing level is contravariant, and not concerned with the flow relations between principals. Then we abusively denote by $\bot$ and $\top$ the triples $(\bot, \top, \bot)$ and $(\top, \bot, \top)$ respectively. In the typing rules for compound expressions, we will use the join operation on security effects:

$$s \curlyvee_G s' \quad =_{\mathrm{def}} \quad (s.\mathsf{r} \curlyvee_G s'.\mathsf{r},\, s.\mathsf{w} \cup s'.\mathsf{w},\, s.\mathsf{t} \curlyvee_G s'.\mathsf{t})$$

as well as the following convention:

$$\frac{}{G;\Gamma \vdash u_{\ell,\theta} : \bot, \theta\,\mathsf{ref}_{\ell}}\ (\textsc{Loc}) \qquad \frac{}{G;\Gamma, x:\tau \vdash x : \bot, \tau}\ (\textsc{Var})$$

$$\frac{F;\Gamma, x:\tau \vdash M : s, \sigma}{G;\Gamma \vdash \lambda x M : \bot, (\tau \xrightarrow[F]{s} \sigma)}\ (\textsc{Abs}) \qquad \frac{}{G;\Gamma \vdash () : \bot, \mathsf{unit}}\ (\textsc{Nil})$$

$$\frac{}{G;\Gamma \vdash tt : \bot, \mathsf{bool}}\ (\textsc{BoolT}) \qquad \frac{}{G;\Gamma \vdash \mathit{ff} : \bot, \mathsf{bool}}\ (\textsc{BoolF})$$

$$\frac{G;\Gamma \vdash M : s, \mathsf{bool} \quad G;\Gamma \vdash N_i : s_i, \tau \quad s.\mathsf{r} \preceq_G s_0.\mathsf{w} \cup s_1.\mathsf{w}}{G;\Gamma \vdash (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1) : s \curlyvee s_0 \curlyvee s_1 \curlyvee (\bot, \top, s.\mathsf{r}), \tau}\ (\textsc{Cond})$$

$$\frac{G;\Gamma \vdash M : s, \tau \xrightarrow[F]{s'} \sigma \quad G;\Gamma \vdash N : s'', \tau \quad s.\mathsf{t} \preceq_G s''.\mathsf{w} \quad s.\mathsf{r} \curlyvee s''.\mathsf{r} \preceq_G s'.\mathsf{w}}{F, G;\Gamma \vdash (MN) : s \curlyvee s' \curlyvee s'' \curlyvee (\bot, \top, s.\mathsf{r} \curlyvee s''.\mathsf{r}), \sigma}\ (\textsc{App})$$

$$\frac{G;\Gamma \vdash M : s, \tau \quad G;\Gamma \vdash N : s', \sigma \quad s.\mathsf{t} \preceq_G s'.\mathsf{w}}{G;\Gamma \vdash M\,;N : s \curlyvee s', \sigma}\ (\textsc{Seq})$$

$$\frac{G;\Gamma \vdash M : s, \theta \quad s.\mathsf{r} \preceq_G \ell}{G;\Gamma \vdash (\mathsf{ref}_{\ell,\theta}\, M) : s, \theta\,\mathsf{ref}_{\ell}}\ (\textsc{Ref}) \qquad \frac{G;\Gamma \vdash M : s, \theta\,\mathsf{ref}_{\ell}}{G;\Gamma \vdash (!\,M) : s \curlyvee (\ell, \top, \bot), \theta}\ (\textsc{Deref})$$

$$\frac{G;\Gamma \vdash M : s, \theta\,\mathsf{ref}_{\ell} \quad G;\Gamma \vdash N : s', \theta \quad s.\mathsf{t} \preceq_G s'.\mathsf{w},\ s.\mathsf{r} \curlyvee s'.\mathsf{r} \preceq_G \ell}{G;\Gamma \vdash (M := N) : s \curlyvee s' \curlyvee (\bot, \ell, \bot), \mathsf{unit}}\ (\textsc{Assign})$$

$$\frac{G;\Gamma \vdash M : s, \mathsf{unit}}{G;\Gamma \vdash (\mathsf{thread}\, M) : (\bot, s.\mathsf{w}, \bot), \mathsf{unit}}\ (\textsc{Thread})$$

$$\frac{G;\Gamma, x:\tau \vdash W : s, \tau}{G;\Gamma \vdash \varrho x W : s, \tau}\ (\textsc{Rec}) \qquad \frac{F, G;\Gamma \vdash M : s, \tau \quad s.\mathsf{r} \preceq_{G \cup F} r \quad s.\mathsf{t} \preceq_{G \cup F} t \preceq_G r}{G;\Gamma \vdash (\mathsf{flow}\ F\ \mathsf{in}\ M) : (r, s.\mathsf{w}, t), \tau}\ (\textsc{Flow})$$

Figure 3: The Type and Effect System

Convention. *In the type system, when the security effects occurring in the context of a judgement $G;\Gamma \vdash M : s, \tau$ involve the join operation $\curlyvee$, it is assumed that the join is taken w.r.t. G, i.e. it is $\curlyvee_G$.*

The typing system is given in Figure 3. Notice that this system is syntax-directed: there is exactly one rule per construction of the language. Let us comment on some of the rules, justifying the side conditions that constrain the typing of an expression, as well as the resulting effect of the expression. We see that the reading and writing effects are respectively introduced by the constructions for dereferencing and updating the memory – rules (Deref) and (Assign). We notice that an expression ($\mathsf{thread}\, M$) has no termination effect, since its evaluation terminates in one step. Indeed, the non-termination of M as a thread cannot influence the computations in the thread that spawned it. The constraints on information flow are implemented in the rules (Cond), (App), (Seq) and (Assign). In the former, the constraint $s.\mathsf{r} \preceq_G s_0.\mathsf{w} \cup s_1.\mathsf{w}$ means that the branches N_0 and N_1 may only write at a level which is greater, with respect to the current flow relation G, than the reading level of the predicate. This is to prevent indirect flows, like in example (7). A slightly more subtle example is

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ (\mathsf{thread}\ v_L := tt)\ \mathsf{else}\ ())$$

which also shows why we record the writing level of the body M of the thread in the effect of $(\mathsf{thread}\ M)$. In the conclusion of (Cond), we record the reading level of the predicate as the termination level of the whole expression. This, combined with the condition $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ in the (Seq) rule, is to prevent termination leaks as in example (8). This example is essentially the same as

$$((\mathsf{if}\ !\,u_H\ \mathsf{then}\ \lambda xx\ \mathsf{else}\ \mathsf{loop})(v_L := tt))$$

which is ruled out by the condition $s.\mathsf{t} \preceq_G s''.\mathsf{w}$ in the (App) rule. In this rule, the condition $s''.\mathsf{r} \preceq_G s'.\mathsf{w}$ is to prevent a direct flow, like in

$$(\lambda x(v_L := x)(!\,u_H))$$

The condition $s.\mathsf{r} \preceq_G s'.\mathsf{w}$ is meant to exclude expressions that read a secret function that writes in a public location, and unravel this effect by applying it. For instance, it rules out an expression like $((!\,u_{\ell,\theta})())$ where $\theta = \mathsf{unit} \xrightarrow{(\bot,\ell',\bot)} \mathsf{unit}$ and ℓ is not lower than ℓ'. Indeed, the value of the reference u might be $\lambda z(v_{\ell',\theta'} := V)$, with different values for V in different memories (see [52] for a similar example). Examples (9) and (10) respectively show why the reading levels of both the function and the argument are recorded in the termination level of the application. We can use the typing rules for abstraction and application to derive the typing of the let construct, that is $(\mathsf{let}\ x = N\ \mathsf{in}\ M) = (\lambda xMN)$, namely:

$$\frac{G;\Gamma \vdash N : s,\tau \quad G;\Gamma, x : \tau \vdash M : s',\sigma \quad s.\mathsf{r} \preceq_G s'.\mathsf{w}}{G;\Gamma \vdash (\mathsf{let}\ x = N\ \mathsf{in}\ M) : s \curlyvee s' \curlyvee (\bot,\top,s.\mathsf{r}),\sigma}$$

The reader may also check that the typing of $M\ ;\ N$ is slightly more liberal than the one of $(\mathsf{let}\ z = M\ \mathsf{in}\ N)$, where $z \notin \mathsf{fv}(N)$ – and similarly for $(\mathsf{ref}_{\ell,\theta}\, M)$, $(!\,M)$ and $(M := N)$ with respect to $(\lambda x(\mathsf{ref}_{\ell,\theta}\, x)M)$, $(\lambda x(!\,x)M)$ and $((\lambda x\lambda y(x := y)M)N)$ –, since we do not have to record the reading effect of the component M in the termination effect of $M\ ;\ N$. For instance neither $(\mathsf{let}\ x = (w_H := !\,u_H)\ \mathsf{in}\ (v_L := tt))$ nor $(\lambda x(w_H := x)(!\,u_H))\ ;\ (v_L := tt)$ can be typed, whereas the expression of example (11) is accepted. This explains our syntax for the imperative part of the language.

The condition $s.\mathsf{r} \preceq_G \ell$ in the typing rule (Ref) is meant to rule out for instance the expression of example (3). The condition $s'.\mathsf{r} \preceq_G \ell$ in the rule (Assign) is to prevent a direct flow, like in example (6). With the condition $s.\mathsf{r} \preceq_G \ell$ we rule out the expression $(!\,u_H) := tt$. Indeed, the value of the reference u might be different locations with level L in different memories. Finally the condition $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ is to prevent termination leaks, as in

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ w_H\ \mathsf{else}\ \mathsf{loop}) := (v_L := tt)$$

These examples show that all the constraints put on information flow in the typing rules for conditional branching, application, sequential composition, reference creation and assignment are in fact necessary. The completeness of this informal analysis will be established by the type soundness result. Regarding recursion, the reader can check for instance that a derived typing for the while construct is

$$\frac{G;\Gamma \vdash M : s,\mathsf{bool} \quad G;\Gamma \vdash N : s',\tau \quad s.\mathsf{r} \curlyvee s'.\mathsf{t} \preceq_G s.\mathsf{w} \curlyvee s'.\mathsf{w}}{G;\Gamma \vdash (\mathsf{while}\ M\ \mathsf{do}\ N) : s \curlyvee s' \curlyvee (\bot,\top,s.\mathsf{r}),\mathsf{unit}}$$

As in [6, 42], we record the confidentiality level of the boolean guard expression in the termination level of the while construct.

To type a flow declaration ($\mathsf{flow}\ F\ \mathsf{in}\ M$), we have to type M in the context of the current flow policy extended with F. In the (Flow) rule, we use a kind of subsumption for the security effect. Namely, the apparent reading and termination effects of the expression ($\mathsf{flow}\ F\ \mathsf{in}\ M$) are allowed to be higher, with respect to F, than the ones of M. For instance, one can check that the following is a valid proof of typing (leaving out the type annotation of references):

$$\dfrac{\dfrac{H \prec L, L \prec H; \Gamma \vdash u_H : \bot, \tau}{H \prec L, L \prec H; \Gamma \vdash (\,!\,u_H) : (H, \top, \bot), \tau}}{L \prec H; \Gamma \vdash (\mathsf{flow}\ H \prec L\ \mathsf{in}\ (\,!\,u_H)) : (L, \top, \bot), \tau}\ H \prec L$$

and therefore one can see that the type system accepts the expression of example (12). Similarly, if, for $\ell = \{p_1, \ldots, p_n\}$ we let, as suggested in the Introduction:

$$\mathsf{declassify}(M, \ell) =_{\mathrm{def}} (\mathsf{flow}\ H \prec p_1, \ldots, H \prec p_n\ \mathsf{in}\ M)$$

where H is a security level that is higher (w.r.t. the current flow policy) than any other one, then we have

$$\dfrac{G; \Gamma \vdash M : s, \tau}{G; \Gamma \vdash \mathsf{declassify}(M, \ell) : (\ell, s.\mathsf{w}, \ell), \tau}$$

Another example is

$$v_L := (\mathsf{flow}\ H \prec L\ \mathsf{in}\ \mathsf{encrypt}(!u_H, K))$$

where $\mathsf{encrypt}$ is a given encryption function, and K is the encryption key. Subsumption in the (Flow) rule will be used in the proof of the Subject Reduction property (1). However, one should notice that in the typing rule for flow declaration we do not allow subsumption for the writing level. Example (13) shows why it would be wrong to do so.

The way we build the termination effect of an expression allows us to accept the expression of example (11). This would not be the case if we had approximated $s.\mathsf{t}$ as $s.\mathsf{r}$ by dealing with security effects of the form (r, w). The classical approach to "weak" non-interference, based on a big-step semantics, actually corresponds to a variant of our type system where $s.\mathsf{t} = \bot$, but obviously this is too weak to ensure the non-disclosure policy. We could think of further refining our type system by building the termination level in a more clever way, since clearly secure programs such as

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ v_H := tt\ \mathsf{else}\ v_H := f\!f)\ ;\ w_L := tt$$

are still rejected by our type system. For instance, we would like to say that the termination level of ($\mathsf{if}\ !\,u_H\ \mathsf{then}\ M\ \mathsf{else}\ N$) may be taken as $\bot$ if we know that both M and N terminate. However, little is known on how to ensure termination in a higher-order imperative language. Indeed, it is well-known since Landin's work [20] that circular higher-order references introduce non-termination, like for instance in (using ref without subscripts)

$$(\mathsf{let}\ x = (\mathsf{ref}\ \lambda yy)\ \mathsf{in}\ x := \lambda y((!\,x)y)\ ;\ ((!\,x)V))$$

which has the type of V. Nevertheless, if we know for a fact that both M and N do not cause any trouble (see [1, 37, 48] for some ways of ensuring this in a simple language), we may compensate

(1) The proof we have for our Soundness Theorem uses all the features of the type system.

for the inflexibility of the type system as regards ($\mathsf{if}\ !\,u_H\ \mathsf{then}\ M\ \mathsf{else}\ N$) for instance by writing instead:

$$(\mathsf{flow}\ H \prec \bot\ \mathsf{in}\ (\mathsf{if}\ !\,u_H\ \mathsf{then}\ M\ \mathsf{else}\ N))$$

Our type system rejects in the same way expressions that are regarded as involving a *timing leak* (see [18, 38, 43]), like

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ M\ \mathsf{else}\ ())\ ;\ v_L := f\!f \tag{14}$$

where M is an expression that takes some time to compute, like $()\ ;\cdots ;\ ()$, although program (14) is generally secure in the sense of Definition 3.3. We notice that the notion of a timing leak does not make much sense at the abstract level at which we describe the operational semantics. This kind of leaks should rather be tracked down when dealing with the code produced by an optimizing compiler for instance. Nevertheless, a program such as (14) could be regarded as unsecure if some specific scheduling discipline were to be taken into account (*cf.* [6, 37, 43]). We leave for further investigations the question as to whether our type system is also adequate to deal with scheduling disciplines (we plan to do this for the cooperative concurrency of ULM [5]), but we observe that it does not restrict the confidentiality level of the predicate in a conditional branching to be $\bot$, as suggested in [43, 47]. In this respect, our system is close to the ones of [6, 42].

As a last example, let us examine a program showing that the flow policy under which the value of a reference will be put cannot be predicted statically. Let M be the following expression:

$$\begin{array}{l}\mathsf{let}\ f = \lambda x \lambda y\, \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{flow}\ p \prec q\ \mathsf{in}\ v_{q,\theta} := !\,y)\\ \qquad\qquad\qquad\quad \mathsf{else}\ (\mathsf{flow}\ p \prec r\ \mathsf{in}\ w_{r,\theta} := !\,y)\\ \mathsf{in}\ ((fN)u_{p,\theta})\end{array}$$

Then one can see that the following typing is admissible:

$$\frac{G;\Gamma \vdash N : s, \mathsf{bool} \quad s.\mathsf{r} \preceq_G \{q, r\}}{G;\Gamma \vdash M : s \curlyvee (p, \{q,r\}, s.\mathsf{r}), \mathsf{unit}}$$

As we explained above, the constraint $s.\mathsf{r} \preceq_G \{q, r\}$ is to prevent termination leaks, since the evaluation of N could terminate or not, depending on the values read in the memory. One should notice that there is no constraint relating p, the confidentiality level of the reference u, with q or r. This means that the value of this reference can be downgraded to either level q or r, depending on the value computed for the boolean N.

It is not obvious to compare our type system with other proposals for similar languages, e.g. [17, 33, 54], since most of them attach security levels to values (this is obviously the case when imperative features are not considered), whereas here all the pure expressions, written without $(!\,N)$ and $(M := N)$, are secure. Closer to ours is the type system of [10], which follows a "store-oriented" view of confidentiality. However, the security effects in [10] are – using our notations – assumed to satisfy $s.\mathsf{r} \preceq s.\mathsf{w}$. Here we do not reject expressions that write below their read level, like the one of Example (11) for instance. As far as we can see, the latter is accepted in the system of [10] using the "informativeness" predicate, which has no counterpart here. Another difference is that [10] only deals with "weak" non-interference, and does not consider termination leaks. These have sometimes been ruled out using very severe restrictions, like allowing only predicates of the lowest confidentiality level in the while loops – which here would mean restricting in the same way the conditional branchings –, see [43, 47] for instance. Regarding termination leaks, we have followed the approach of [6, 42], using the "termination effect". As a matter of fact, our type system appears to be a quite direct generalization of the one of [6], and, more generally, of type systems for imperative languages, like the one originally proposed in [50]. In that paper, and many others that followed, an expression has no side effect: it can read from the store, but does not create or update references. Moreover, its evaluation is assumed to terminate and produce a value (not an exception) in the context of any memory. Then the security effect of such an expression is reduced to its reading level – in our setting, it would have security effect $(r, \top, \bot)$. On the other hand, the programs, or "commands" in [50] only read the store when evaluating an expression, and therefore it is enough to record their writing level (and their termination level, if termination leaks are considered). It is then easy to see that, if we restrict our attention to the simple imperative fragment, given by the syntax

$$S,\ T\ldots\ ::=\ ()\ \mid\ (x := E)\ \mid\ S\,;T\ \mid\ (\mathsf{if}\ E\ \mathsf{then}\ S\ \mathsf{else}\ T)\ \mid\ (\mathsf{while}\ E\ \mathsf{do}\ S)$$

where E is any pure expression, having a security effect of the form $(r, \top, \bot)$, and if we forget about the reading effect of these programs (which is not involved in any flow constraint), then the type system we get is almost exactly the one of [6]. A rather cosmetic difference with type systems for imperative languages is that they often use subtyping, where we have used flow constraints which, to our view, clearly show the points where a possible flow of information may occur. A well-known technical advantage of having a syntax-directed type system is that it simplifies the proofs a little.

We now establish some properties of typed expressions, and in particular a Subject Reduction result. We start by remarking that a value, and more generally a pseudo-value, has no effect, and that this is properly reflected in the type system. Moreover, the typing of a pseudo-value does not depend on the flow policy:

Remark 4.1. $\forall W \in \mathcal{W}.\ G; \Gamma \vdash W : s, \tau\ \Rightarrow\ s = \bot\ \&\ \forall F.\ F; \Gamma \vdash W : \bot, \tau.$

It should be clear that the termination effect of a typable expression is always bounded by its reading effect:

Remark 4.2. $G; \Gamma \vdash M : s, \tau\ \Rightarrow\ s.\mathsf{t} \preceq_G s.\mathsf{r}.$

To prove Subject Reduction we shall follow the usual steps [51], thus omitting the details of the proofs. We need, in particular, some standard weakening and strengthening properties:

Lemma 4.3.

(i) *If* $G; \Gamma \vdash M : s, \tau$ *and* $x \notin \mathsf{dom}(\Gamma)$ *then* $G; \Gamma, x : \sigma \vdash M : s, \tau$.

(ii) *If* $G; \Gamma, x : \sigma \vdash M : s, \tau$ *and* $x \notin \mathsf{fv}(M)$ *then* $G; \Gamma \vdash M : s, \tau$.

(iii) *If* $G; \Gamma \vdash M : s, \tau$ *then* $F, G; \Gamma \vdash M : s', \tau$ *for some* s' *such that* $s'.\mathsf{r} \preceq_{F \cup G} s.\mathsf{r}$, $s'.\mathsf{w} = s.\mathsf{w}$ *and* $s'.\mathsf{t} \preceq_{F \cup G} s.\mathsf{t}$.

Proof: by induction on the inference of the judgements, and by case on the last rule used in this typing proof (that is, by induction on M, since the type system is syntax-directed). Regarding the last point, we get, for an expression M typable in the context of the flow relation $F \cup G$, a security effect which is lower than the one it has in the context of G, simply because in the conclusion of the typing rules we make a join with respect to $F \cup G$, instead of G. ❏

Lemma (Substitution) 4.4. $G; \Gamma \vdash W : \bot, \tau\ \&\ G; \Gamma, x : \tau \vdash M : s, \sigma\ \Rightarrow\ G; \Gamma \vdash \{x \mapsto W\}M : s, \sigma$

Proof: by induction on the inference of $G; \Gamma, x : \tau \vdash M : s, \sigma$, and by case on the last rule used in this typing proof, using the previous lemma. Let us just examine the case of the (Flow) typing rule, which is the only non-standard one. In this case we have $M = (\mathsf{flow}\ F\ \mathsf{in}\ M')$, and the typing proof must end with

$$\dfrac{\dfrac{\vdots}{F, G; \Gamma, x : \tau \vdash M' : s', \sigma}}{G; \Gamma, x : \tau \vdash (\mathsf{flow}\ F\ \mathsf{in}\ M') : s, \sigma}$$

with $s'.\mathsf{r} \preceq_{G \cup F} s.\mathsf{r}$, $s'.\mathsf{w} = s.\mathsf{w}$ and $s'.\mathsf{t} \preceq_{G\cup F} s.\mathsf{t} \preceq_G s.\mathsf{r}$. By Remark 4.1 we have $F, G; \Gamma \vdash W : \bot, \tau$, and therefore by induction hypothesis, the judgement $F, G; \Gamma \vdash \{x \mapsto W\}M' : s', \sigma$ is provable, hence also $G; \Gamma \vdash (\mathsf{flow}\ F\ \mathsf{in}\ \{x \mapsto W\}M') : s, \sigma$. ❑

Lemma (Replacement) 4.5. *If $G; \Gamma \vdash \mathbf{E}[M] : s, \tau$ is a valid judgement, with a proof where M has the typing $\lceil\mathbf{E}\rceil, G; \Gamma \vdash M : s', \sigma$, and if $\lceil\mathbf{E}\rceil, G; \Gamma \vdash N : s'', \sigma$ with $s''.\mathsf{r} \preceq_{G\cup\lceil\mathbf{E}\rceil} s'.\mathsf{r}$, $s'.\mathsf{w} \preceq s''.\mathsf{w}$ and $s''.\mathsf{t} \preceq_{G\cup\lceil\mathbf{E}\rceil} s'.\mathsf{t}$, then $G; \Gamma \vdash \mathbf{E}[N] : s_0, \tau$ for some s_0 such that $s_0.\mathsf{r} \preceq_G s.\mathsf{r}$, $s.\mathsf{w} \preceq s_0.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_G s.\mathsf{t}$.*

Proof: by induction on $\mathbf{E}$. We just examine the case where $\mathbf{E} = (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}')$. The typing proof of $\mathbf{E}[M]$ must end with

$$\dfrac{\dfrac{\vdots}{F, G; \Gamma \vdash \mathbf{E}'[M] : s_1, \tau}}{G; \Gamma \vdash \mathbf{E}[M] : s, \tau}$$

with $s_1.\mathsf{r} \preceq_{G\cup F} s.\mathsf{r}$, $s_1.\mathsf{w} = s.\mathsf{w}$ and $s_1.\mathsf{t} \preceq_{G\cup F} s.\mathsf{t} \preceq_G s.\mathsf{r}$. By induction hypothesis there exists s_0 such that the judgement $F, G; \Gamma \vdash \mathbf{E}'[N] : s_0, \tau$ is provable, and $s_0.\mathsf{r} \preceq_{G\cup F} s_1.\mathsf{r}$, $s_1.\mathsf{w} \preceq s_0.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_{G\cup F} s_1.\mathsf{t}$. Then by the (Flow) rule $G; \Gamma \vdash \mathbf{E}[N] : (s.\mathsf{r}, s_0.\mathsf{w}, s.\mathsf{t}), \tau$ is provable. ❑

The Subject Reduction property states that the type of an expression is preserved by reduction. Regarding its effects some may be performed, by reading or updating a reference, and some may be discarded, when a branch in a conditional expression is taken. Then the effects of an expression "decrease" along the computations, and, in particular, its confidentiality level becomes less critical.

Proposition (Subject Reduction) 4.6. *If $G; \Gamma \vdash M : s, \tau$ and $(M, \mu) \xrightarrow[F]{N} (M', \mu')$ with $u_{\ell,\theta} \in \mathsf{dom}(\mu) \;\Rightarrow\; G; \Gamma \vdash \mu(u_{\ell,\theta}) : \bot, \theta$ then $G; \Gamma \vdash M' : s', \tau$ and $G; \Gamma \vdash N : s'', \mathsf{unit}$ for some s' and s'' such that $s'.\mathsf{r} \curlyvee_G s''.\mathsf{r} \preceq_G s.\mathsf{r}$, $s.\mathsf{w} \preceq s'.\mathsf{w} \cup s''.\mathsf{w}$ and $s'.\mathsf{t} \preceq_G s.\mathsf{t}$.*

Proof: by induction on the proof of the transition $(M, \mu) \xrightarrow[F]{N} (M', \mu')$. We only examine some cases. If $M = (\lambda x M_0 V)$, with $F = \emptyset$, $N = ()$, $\mu' = \mu$ and $M' = \{s \mapsto V\}M_0$, then we use Lemma 4.4. The argument is similar if $M = \varrho x W$ and $M' = \{x \mapsto \varrho x W\}W$. If $M = (\mathsf{flow}\ F'\ \mathsf{in}\ V)$ and $M' = V$, with $F = \emptyset$, $N = ()$ and $\mu' = \mu$, then the proof of the judgement $G; \Gamma \vdash M : s, \tau$ must have the form

$$\dfrac{\dfrac{\vdots}{F', G; \Gamma \vdash V : \bot, \tau}}{G; \Gamma, \vdash (\mathsf{flow}\ F'\ \mathsf{in}\ V) : s, \tau}$$

with $s.\mathsf{w} = \bot$. By Remark 4.1 we have $G; \Gamma \vdash M' : \bot, \tau$, hence we are done in this case. If $M = \mathbf{E}[M_0]$ and $M' = \mathbf{E}[M_0']$ with $(M_0, \mu) \xrightarrow[F']{N} (M', \mu')$ and $F = F' \cup \lceil\mathbf{E}\rceil$ then we use the induction hypothesis and Lemma 4.5. If $M = \mathbf{E}[(\mathsf{thread}\, M_0)]$ and $M' = \mathbf{E}[()]$, with $F = \emptyset$,

$N = (\mathsf{flow}\ \lceil \mathbf{E} \rceil\ \mathsf{in}\ M_0)$ and $\mu' = \mu$, and if in the proof of $G;\Gamma \vdash M : s,\tau$, the expression $(\mathsf{thread}\, M_0)$ has the typing

$$\dfrac{\dfrac{\vdots}{\lceil \mathbf{E} \rceil, G;\Gamma \vdash M_0 : s', \mathsf{unit}}}{\lceil \mathbf{E} \rceil, G;\Gamma \vdash (\mathsf{thread}\, M_0) : (\bot, s'.\mathsf{w}, \bot), \mathsf{unit}}$$

then we conclude using the Lemma 4.5 and the fact that

$$\dfrac{\dfrac{\vdots}{\lceil \mathbf{E} \rceil, G;\Gamma \vdash M_0 : s', \mathsf{unit}}}{G;\Gamma \vdash (\mathsf{flow}\ \lceil \mathbf{E} \rceil\ \mathsf{in}\ M_0) : s', \mathsf{unit}}$$

is a valid typing (thanks to Remark 4.2). ❏

We conclude this section with some properties of typed expressions that will be used to prove our main result. We observe (in specific cases – those needed for the soundness proof) that the intuitive meaning of the reading and writing effect is indeed captured by the type system.

Lemma 4.7.

(i) $G;\Gamma \vdash \mathbf{E}[(!u_{\ell,\theta})] : s,\tau \;\Rightarrow\; \ell \preceq_{G \cup \lceil \mathbf{E} \rceil} s.\mathsf{r}$

(ii) *If* $G;\Gamma \vdash \mathbf{E}[(u_{\ell,\theta} := V)] : s,\tau$ *then* $s.\mathsf{w} \preceq \ell$

Proof: by induction on $\mathbf{E}$, easy. ❏

With the intuition that the writing effect of a typable expression is a lower bound of the level of the references the expression may update, we define:

Definition (Syntactically High Expressions) 4.8. *An expression M is* syntactically (G,ℓ)-high *if $G;\Gamma \vdash M : s,\tau$ with $s.\mathsf{w} \not\preceq_G \ell$. The expression M is a* (G,ℓ)-high function *if $G;\Gamma \vdash M : s, (\tau \xrightarrow[F]{s'} \sigma)$ with $s'.\mathsf{w} \not\preceq_G \ell$.*

This definition is justified by the following results.

Lemma 4.9. *A syntactically (G,ℓ)-high expression is operationally (G,ℓ)-high.*

Proof: we show that if M is syntactically (G,ℓ)-high, that is $G;\Gamma \vdash M : s,\tau$ with $s.\mathsf{w} \not\preceq_G \ell$, and $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ then $\mu' \doteq^{G,\ell} \mu$. This is enough since, by Subject Reduction, both N and M' are syntactically (G,ℓ)-high. We proceed by cases on the proof of the transition $(M,\mu) \xrightarrow[F]{N} (M',\mu')$. The lemma is trivial in all the cases where $\mu \subseteq \mu'$. If $M = \mathbf{E}[(u_{\ell',\theta} := V)]$ and $\mu' = \mu[u_{\ell',\theta} := V]$ then $s.\mathsf{w} \preceq \ell'$ by Lemma 4.7(ii). This implies $\ell' \not\preceq_G \ell$, hence $\mu' \doteq^{G,\ell} \mu$. ❏

Lemma 4.10. *If (MN) is typable in the context of the flow relation G, with $M,N \in \mathcal{H}_{G,\ell}$ and M is a (G,ℓ)-high function then $(MN) \in \mathcal{H}_{G,\ell}$.*

Proof: let $\mathcal{G}$ be the set containing $\mathcal{H}_{G,\ell}$, and containing expressions (MN) provided they are typable in the context of the flow relation G, and satisfy $M,N \in \mathcal{G}$ and M is a (G,ℓ)-high function. Assume that such an application (MN) performs the transition $((MN),\mu) \xrightarrow[F]{N'} (M',\mu')$. We show, by induction on the proof of this transition, that this implies $M',N' \in \mathcal{G}$ and $\mu' \doteq^{G,\ell} \mu$. If

$M' = \{x \mapsto N\}M''$ and $N' = ()$, where $M = \lambda x M''$ (and $N \in \mathcal{V}al$) and $\mu' = \mu$, then, since both M and N are values, the typing of (MN) must end with

$$\dfrac{\dfrac{\vdots}{G;\Gamma \vdash M : \bot, \tau \xrightarrow[F]{s} \sigma} \qquad \dfrac{\vdots}{G;\Gamma \vdash N : \bot, \tau}}{F, G;\Gamma \vdash (MN) : s, \sigma}$$

Since M is a (G,ℓ)-high function, we have $M' \in \mathcal{H}_{G,\ell}$ by Subject Reduction and the Lemma 4.9. If $M' = (M''N)$ with $(M,\mu) \xrightarrow[F]{N'} (M'',\mu')$ we have $M'', N' \in \mathcal{G}$ and $\mu \mathrel{\hat{=}}^{G,\ell} \mu'$ by induction hypothesis, and M'' is a (G,ℓ)-high function, by Subject Reduction, hence $M' \in \mathcal{G}$. The argument is similar if $M' = (MN'')$ where $M \in \mathcal{V}al$ and $(N,\mu) \xrightarrow[F]{N'} (N'',\mu')$. $\square$

We believe we could prove the standard Type Safety result for our system, establishing that a typable program either diverges or terminates, returning a value of the appropriate type. However, this is not the topic of our work, since our aim is rather to show a security property of typable programs.

5. Type Soundness

In this section we establish the main technical result of our paper, namely the type soundness property:

THEOREM (SOUNDNESS). *If M is typable in the context of a flow policy G, that is if for some Γ, s and τ we have $G;\Gamma \vdash M : s,\tau$, then M satisfies the non-disclosure policy with respect to G, that is $M \in \mathcal{ND}(G)$.*

To prove this result, for any security level ℓ we shall exhibit a (G,ℓ)-bisimulation that contains the pair (M,M) for any G-typable expression M. Such a relation is built by examining the possible cases for pairs (P,Q) such that $(M,\mu) \rightarrow (P,\mu')$ and $(M,\nu) \rightarrow (Q,\nu')$, where μ and ν satisfy the condition of Definition 3.1 of bisimulations. Lemma 2.3 shows that, starting from a given expression M in the context of two different memories, the evaluation process may only branch (to P and Q) when the expression comes to read a reference. This is the basis for building our bisimulations, as shown by Lemmas 5.2 and 5.5 below, which relate such a branching situation with the typing of expressions.

We first build a kind of "strong bisimulation" (see the next Proposition) from expressions that have a low termination effect. Namely, given a flow policy G and a confidentiality level ℓ, let us define inductively the relation $\mathcal{S}_{G,\ell}$ on expressions, as follows: $M \,\mathcal{S}_{G,\ell}\, N$ if both M and N are *typable with a low termination effect in the context of* G, that is more precisely $G;\Gamma \vdash M : s,\tau$ and $G;\Gamma \vdash N : s',\sigma$ for some Γ, s, s', τ and σ, with $s.\mathsf{t} \preceq_G \ell$ and $s'.\mathsf{t} \preceq_G \ell$, and one of the following holds:

(Clause 1) M and N are both values, or

(Clause 2) $M = N$, or

(Clause 3) $M = M' \,\boldsymbol{;}\, M''$ and $N = N' \,\boldsymbol{;}\, M''$ where $M' \,\mathcal{S}_{G,\ell}\, N'$, or

(Clause 4) $M = (\mathsf{ref}_{\ell',\theta}\, M')$ and $N = (\mathsf{ref}_{\ell',\theta}\, N')$ with $M' \,\mathcal{S}_{G,\ell}\, N'$ and $\ell' \not\preceq_G \ell$, or

(Clause 5) $M = (\mathbf{!}\, M')$ and $N = (\mathbf{!}\, N')$ with $M' \,\mathcal{S}_{G,\ell}\, N'$, or

(Clause 6) $M = (M_0 := M')$ and $N = (N_0 := M')$ with $M_0 \,\mathcal{S}_{G,\ell}\, N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 7) $M = (V := M')$ and $N = (V' := N')$ with $M' \,\mathcal{S}_{G,\ell}\, N'$ and V and V' both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 8) $M = (\mathsf{flow}\ F\ \mathsf{in}\ M')$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N')$ with $M' \,\mathcal{S}_{F\cup G,\ell}\, N'$.

This is clearly a symmetric relation.

Remark 5.1. $M \,\mathcal{S}_{G,\ell}\, N \ \&\ M \in \mathcal{V}al \ \Rightarrow\ N \in \mathcal{V}al$

The next lemma states that if, in the context of two different memories, the evaluation of a typable expression M with a low termination effect may split (see Lemma 2.3), then the resulting expressions are still in the relation $\mathcal{S}_{G,\ell}$.

Lemma 5.2. *If $G;\Gamma \vdash \mathbf{E}[(\mathbf{!}u_{\ell',\theta})] : s,\tau$ with $s.\mathsf{t} \preceq_G \ell$ and $\ell' \not\preceq_{G\cup\lceil\mathbf{E}\rceil} \ell$ then for any values $V_0, V_1 \in \mathcal{V}al$ such that $\lceil\mathbf{E}\rceil, G;\Gamma \vdash V_i : \bot,\theta$ we have $\mathbf{E}[V_0] \,\mathcal{S}_{G,\ell}\, \mathbf{E}[V_1]$.*

Proof: by induction on the structure of $\mathbf{E}$.

- If $\mathbf{E} = []$, we have $V_0 \,\mathcal{S}_{G,\ell}\, V_1$ by Clause 1.

- If $\mathbf{E} = (\mathsf{if}\ \mathbf{E}'\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1)$ then $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\mathsf{bool}$ with $s'.\mathsf{r} \preceq_G s.\mathsf{t}$. By Lemma 4.7(i) we have $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$, and therefore this case is not possible.

- Assume that $\mathbf{E} = (\mathbf{E}'N)$. Then $G = F, G'$ with $G';\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \sigma \xrightarrow[F]{s''} \tau$ and $s'.\mathsf{r} \preceq_G s.\mathsf{t}$, hence $s'.\mathsf{r} \preceq_G \ell$. Since we also have $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$ by Lemma 4.7(i), this case is not possible.

- If $\mathbf{E} = (V\mathbf{E}')$ then $G = F, G'$ with $G';\Gamma \vdash V : \bot, \sigma \xrightarrow[F]{s''} \tau$ and $G';\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\sigma$ with $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$ by Lemma 4.7(i). Since $s'.\mathsf{r} \preceq_G s.\mathsf{t} \preceq_G \ell$, this case is not possible.

- If $\mathbf{E} = \mathbf{E}'\ \mathbf{;}\ N$ we have $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\sigma$ with $s'.\mathsf{t} \preceq_G s.\mathsf{t}$, and we use the induction hypothesis and Clause 3 to conclude.

- If $\mathbf{E} = (\mathsf{ref}_{\ell'',\theta'}\, \mathbf{E}')$ then $\tau = \theta'\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s,\theta'$ with $s.\mathsf{r} \preceq_G \ell''$. Since $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s.\mathsf{r}$ by Lemma 4.7(i), we have $\ell'' \not\preceq_G \ell$, and we use the induction hypothesis and Clause 4 to conclude.

- If $\mathbf{E} = (\mathbf{!}\,\mathbf{E}')$ then $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\tau\,\mathsf{ref}_{\ell''}$ with $s'.\mathsf{t} = s.\mathsf{t}$, and we use the induction hypothesis and Clause 5 to conclude.

- If $\mathbf{E} = (\mathbf{E}' := N)$ we have $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\sigma\,\mathsf{ref}_{\ell''}$ with $s'.\mathsf{r} \preceq_G \ell''$ and $s'.\mathsf{t} \preceq_G s.\mathsf{t}$. Since $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$ by Lemma 4.7(i), we also have $\ell'' \not\preceq_G \ell$ (since otherwise we would have $\ell' \preceq_{G\cup\lceil\mathbf{E}\rceil} \ell$), and we use the induction hypothesis and Clause 6 to conclude.

- If $\mathbf{E} = (V := \mathbf{E}')$ we have $G;\Gamma \vdash V : \bot,\sigma\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\sigma$ with $s'.\mathsf{t} \preceq_G s.\mathsf{t}$ and $s'.\mathsf{r} \preceq_G \ell''$. Since $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s''.\mathsf{r}$ Lemma 4.7(i), we also have $\ell'' \not\preceq_G \ell$ (since otherwise we would have $\ell' \preceq_{G\cup\lceil\mathbf{E}\rceil} \ell$), and we use the induction hypothesis and Clause 7 to conclude.

- If $\mathbf{E} = (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}')$ we have $F, G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s',\tau$ with $s'.\mathsf{t} \preceq_{F\cup G} s.\mathsf{t}$, hence $s'.\mathsf{t} \preceq_{F\cup G} \ell$. Since $\lceil\mathbf{E}\rceil = F \cup \lceil\mathbf{E}'\rceil$, we conclude using the induction hypothesis and Clause 8. ❏

The next proposition states that $\mathcal{S}_{G,\ell}$ is indeed a kind of "strong bisimulation" with respect to the transition relation $\xrightarrow[F]{N}$.

Proposition 5.3. *If $M\,\mathcal{S}_{G,\ell}\,N$ and $(M,\mu) \xrightarrow[F]{M''} (M',\mu')$, with $\mu \overset{\frown}{=}^{F\cup G,\ell} \nu$ and u is fresh for ν if $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$, then there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''} (N',\nu')$ with $M'\,\mathcal{S}_{G,\ell}\,N'$ and $\mu' \overset{\frown}{=}^{G,\ell} \nu'$ with $\mathsf{dom}(\nu' - \nu) = \mathsf{dom}(\mu' - \mu)$.*

Proof: by induction on the definition of $\mathcal{S}_{G,\ell}$.

- It cannot be that $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 1.

- If $N = M$, then by Lemma 3.6 there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''} (N',\nu')$ with $\mu' \overset{\frown}{=}^{F\cup G,\ell} \nu'$. By Subject Reduction we have $G;\Gamma \vdash M' : s',\tau$ with $s'.\mathsf{t} \preceq_G s.\mathsf{t}$, and therefore $M'\,\mathcal{S}_{G,\ell}\,N'$ if $M' = N'$, by Clause 2. Otherwise, by Lemma 2.3 we have ($M'' = ()$ and) $M = \mathbf{E}[(!u_{\ell',\theta})]$ with $M' = \mathbf{E}[\mu(u_{\ell',\theta})]$ and $N' = \mathbf{E}[\nu(u_{\ell',\theta})]$, and $F = \lceil \mathbf{E} \rceil$. Since $\mu(u_{\ell',\theta}) \neq \nu(u_{\ell',\theta})$, we have $\ell' \not\preceq_{F\cup G} \ell$, and therefore $M'\,\mathcal{S}_{G,\ell}\,N'$ by Lemma 5.2 above (and the Subject Reduction property).

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 3, that is $M = M_0\,;M_1$ and $N = N_0\,;M_1$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, then either M_0 is a value and $M' = M_1$ (and $F = \emptyset$, $M'' = ()$ and $\mu' = \mu$), or $M' = M_0'\,;M_1$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. In the latter case, we use the induction hypothesis and Clause 3 to conclude. Otherwise, we have $N_0 \in \mathcal{V}al$ by Remark 5.1, hence $(N,\nu) \xrightarrow[\emptyset]{()} (M_1,\nu)$, and we conclude using Clause 2.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 4, we have $M = (\mathsf{ref}_{\ell',\theta}\,M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\,N_0)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and $\ell' \not\preceq_G \ell$. There are two cases. If M_0 is a value, then $M' = u_{\ell',\theta}$, with u fresh for μ, $M'' = ()$ and $\mu' = \mu \cup \{u_{\ell',\theta} \mapsto M_0\}$ (and therefore u is also fresh for ν). Then $N_0 \in \mathcal{V}al$ by Remark 5.1, and therefore $(N,\nu) \xrightarrow[\emptyset]{()} (u_{\ell',\theta}, \nu \cup \{u_{\ell',\theta} \mapsto N_0\})$. If we let $\nu' = \nu \cup \{u_{\ell',\theta} \mapsto N_0\}$ then $\mu' \overset{\frown}{=}^{G,\ell} \nu'$ since $\ell' \not\preceq_G \ell$, and we conclude using Clause 1. Otherwise, $M' = (\mathsf{ref}_{\ell,\theta}\,M_0')$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$, and we use the induction hypothesis and Clause 4 to conclude.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 5, that is $M = (!\,M_0)$ and $N = (!\,N_0)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, we distinguish two sub-cases. If $M_0 = u_{\ell',\theta}$ and $M' = \mu(u_{\ell',\theta})$ with $M'' = ()$ and $\mu' = \mu$ then, by Remark 5.1, N_0 is a value too, and it must be a reference $v_{\ell'',\theta'}$, since N is typable. In this case we let $N' = \nu(v_{\ell'',\theta'})$ and $\nu' = \nu$, and we conclude using Clause 1. Otherwise, $M' = (!\,M_0')$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$, and we use the induction hypothesis and Clause 5 to conclude.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 6, that is $M = (M_0 := M_1)$ and $N = (N_0 := M_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell''}$ for some θ and ℓ'' such that $\ell'' \not\preceq_G \ell$, we have $M' = (M_0' := M_1)$ for some M_0' such that $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. By induction hypothesis, there exist N_0' such that $(N_0,\nu) \xrightarrow[F]{M''} (N_0',\nu')$ with $M_0'\,\mathcal{S}_{G,\ell}\,N_0'$ and $\mu' \overset{\frown}{=}^{F\cup G,\ell} \nu'$ with $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$. If $M_0' \in \mathcal{V}al$, then also $N_0' \in \mathcal{V}al$ by Remark 5.1, and we conclude using Clause 7 (and Clause 2, as regards M_1). Otherwise, we conclude by Clause 6 (and Subject Reduction).

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 7, then $M = (V := M_0)$ and $N = (V' := N_0)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and V and

V' both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$. There are again two cases. If M_0 is a value and $M' = () = M''$ (and $F = \emptyset$) with $\mu' = \mu[V := M_0]$ then $N_0 \in \mathcal{V}al$ by Remark 5.1, and therefore $(N, \nu) \xrightarrow[\emptyset]{()} ((), \nu')$ where $\nu' = \nu[V' := N_0]$, and therefore $\mu' \bumpeq^{G,\ell} \nu'$ since $\ell' \not\preceq_G \ell$. Otherwise, the transition is due to M_0, and we use the induction hypothesis and Clause 7 to conclude.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 8, that is $M = (\mathsf{flow}\ F\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N_0)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$. There are again two cases, depending on whether M_0 is a value or not, and we use the same arguments as in the previous cases. ❑

Now given G and ℓ as above, we define the binary relation $\mathcal{R}_{G,\ell}$ on expressions inductively as follows: $M\,\mathcal{R}_{G,\ell}\,N$ if and only if both M and N are *typable in the context of the flow relation* G, and one of the following holds:

(Clause 1) M and N are both values, or

(Clause 2) $M = N$, or

(Clause 3) $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M'\ \mathsf{else}\ N')$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M'\ \mathsf{else}\ N')$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M' \mathrel{)\!(}^{G,\ell} N'$, or

(Clause 4) $M = (M_0M_1)$ and $N = (N_0N_1)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$, M_0 and N_0 are (G,ℓ)-high functions, and $M_1 \mathrel{)\!(}^{G,\ell} N_1$, or

(Clause 5) $M = (M_0M_1)$ and $N = (N_0N_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, M_0 and N_0 are (G,ℓ)-high functions, and $M_1\,\mathcal{R}_{G,\ell}\,N_1$, or

(Clause 6) $M = M_0\,\mathbf{;}\,M_1$ and $N = N_0\,\mathbf{;}\,M_1$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M_1 \in \mathcal{H}_{G,\ell}$, or

(Clause 7) $M = M_0\,\mathbf{;}\,M_1$ and $N = N_0\,\mathbf{;}\,M_1$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, or

(Clause 8) $M = (\mathsf{ref}_{\ell',\theta}\,M')$ and $N = (\mathsf{ref}_{\ell',\theta}\,N')$ with $M'\,\mathcal{R}_{G,\ell}\,N'$ and $\ell' \not\preceq_G \ell$, or

(Clause 9) $M = (\mathbf{!}\,M')$ and $N = (\mathbf{!}\,N')$ with $M'\,\mathcal{R}_{G,\ell}\,N'$, or

(Clause 10) $M = (M_0 := M')$ and $N = (N_0 := M')$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, and $M' \in \mathcal{H}_{G,\ell}$, or

(Clause 11) $M = (M_0 := M')$ and $N = (N_0 := M')$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 12) $M = (V := M')$ and $N = (V' := N')$ with $M'\,\mathcal{R}_{G,\ell}\,N'$ and V and V' both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 13) $M = (\mathsf{flow}\ F\ \mathsf{in}\ M')$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N')$ with $M'\,\mathcal{R}_{F\cup G,\ell}\,N'$.

Remark 5.4. $M\,\mathcal{R}_{G,\ell}\,N\ \&\ M \in \mathcal{V}al\ \Rightarrow\ N \in \mathcal{V}al$

The relation $\mathcal{R}_{G,\ell}$ is clearly symmetric. We shall prove that this is in turn a kind of (weak) bisimulation, see the next proposition. The following lemma is analogous to Lemma 5.2. It states that if a typable expression is reading in the "high" part of the memory, in the context of two possibly different memories, then the resulting expressions are in the relation $\mathcal{R}_{G,\ell}$.

Lemma 5.5. *If $G;\Gamma \vdash \mathbf{E}[(\mathbf{!}u_{\ell',\theta})] : s,\tau$ with $\ell' \not\preceq_{G\cup\lceil\mathbf{E}\rceil} \ell$ then for any values $V_0, V_1 \in \mathcal{V}al$ such that $\lceil\mathbf{E}\rceil, G;\Gamma \vdash V_i : \bot,\theta$ we have $\mathbf{E}[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{E}[V_1]$.*

Proof: by induction on the structure of $\mathbf{E}$.

- If $\mathbf{E} = [\,]$, we have $\mathbf{E}[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{E}[V_1]$ by Clause 1.

- If $\mathbf{E} = (\mathsf{if}\ \mathbf{E}'\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1)$ then we have $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \mathsf{bool}$ and $G;\Gamma \vdash N_i : s_i, \tau$ with $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r} \preceq_G s_i.\mathsf{w}$, by Lemma 4.7(i), and therefore $s_i.\mathsf{w} \not\preceq_G \ell$, This implies $N_0 \mathrel{)\!\!-\!\!(}^{G,\ell} N_1$ by Lemma 4.9, and we conclude using the induction hypothesis and Clause 3.

- If $\mathbf{E} = (\mathbf{E}'N)$ then we have $G = F, G'$ with $G';\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s_0, \sigma \xrightarrow[F]{s_1} \tau$ and $G';\Gamma \vdash N : s_2, \sigma$, with $s_0.\mathsf{r} \preceq_{G'} s_1.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_{G'} s_2.\mathsf{w}$, hence also $s_0.\mathsf{r} \preceq_G s_1.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_G s_2.\mathsf{w}$. By Lemma 4.3(iii) we have $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s_0', \sigma \xrightarrow[F]{s_1} \tau$ and $G;\Gamma \vdash N : s_2', \sigma$ for some s_0' and s_2' such that $s_0'.\mathsf{r} \preceq_G s_0.\mathsf{r}$, $s_0'.\mathsf{t} \preceq_G s_0.\mathsf{t}$ and $s_2'.\mathsf{w} = s_2.\mathsf{w}$. By Lemma 4.7(i) we have $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s_0'.\mathsf{r}$ and therefore $s_1.\mathsf{w} \not\preceq_G \ell$, that is, $\mathbf{E}'[(\mathbf{!}u_{\ell',\theta})]$ is a (G,ℓ)-high function. The same holds for both $\mathbf{E}'[V_0]$ and $\mathbf{E}'[V_1]$, and by induction hypothesis we have $\mathbf{E}'[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{E}'[V_1]$. Now if $s_0'.\mathsf{t} \not\preceq_G \ell$, we have $N \in \mathcal{H}_{G,\ell}$ by Lemma 4.9 since $s_0'.\mathsf{t} \preceq_G s_2.\mathsf{w} = s_2'.\mathsf{w}$, and therefore $\mathbf{E}[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{E}[V_1]$ by Clause 4. Otherwise $\mathbf{E}'[V_0]\,\mathcal{S}_{G,\ell}\,\mathbf{E}'[V_1]$ by Lemma 5.2, and therefore $\mathbf{E}[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{E}[V_1]$ by Clause 5 (and Clause 2).

- If $\mathbf{E} = (V\mathbf{E}')$ then $G = F, G'$ and $G';\Gamma \vdash V : \bot, \sigma \xrightarrow[F]{s_0} \tau$ and $G';\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s_1, \sigma$ with $s_1.\mathsf{r} \preceq_{G'} s_0.\mathsf{w}$. By Lemma 4.3(iii) we have $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s_2, \sigma$ for some s_2 such that $s_2.\mathsf{r} \preceq_G s_1.\mathsf{r}$. By Lemma 4.7(i) we have $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s_2.\mathsf{r}$ and therefore $s_0.\mathsf{w} \not\preceq_G \ell$, that is, V is a (G,ℓ)-high function. Then we conclude using the induction hypothesis and Clause 5, since $V\,\mathcal{S}_{G,\ell}\,V$ by definition of $\mathcal{S}_{G,\ell}$.

- If $\mathbf{E} = \mathbf{E}'\,\mathbf{;}\,N$ then $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \sigma$ and $G;\Gamma \vdash N : s'', \tau$ with $s'.\mathsf{t} \preceq_G s''.\mathsf{w}$. If $s'.\mathsf{t} \preceq_G \ell$, we have $\mathbf{E}[V_0]\,\mathcal{S}_{G,\ell}\,\mathbf{E}[V_1]$ by Lemma 5.2, and we conclude using Clause 7. Otherwise, we have $s''.\mathsf{w} \not\preceq_G \ell$, and therefore $N \in \mathcal{H}_{G,\ell}$ by Lemma 4.9, and we conclude using the induction hypothesis and Clause 6.

- For the case where $\mathbf{E} = (\mathsf{ref}_{\ell'',\theta'}\,\mathbf{E}')$, we argue as in Lemma 5.2. The case $\mathbf{E} = (\mathbf{!}\,\mathbf{E}')$ is immediate, using the induction hypothesis.

- If $\mathbf{E} = (\mathbf{E}' := N)$ then $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \theta'\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash N : s'', \theta'$ with $s'.\mathsf{r} \preceq_G \ell''$ and $s'.\mathsf{t} \preceq_G s''.\mathsf{w}$. By Lemma 4.7(i) we have $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$, and therefore $\ell'' \not\preceq_G \ell$. If $s'.\mathsf{t} \not\preceq_G \ell$ we have $s''.\mathsf{w} \not\preceq_G \ell$, and therefore $N \in \mathcal{H}_{G,\ell}$ by Lemma 4.9, and we conclude using the induction hypothesis and Clause 10 in this case (notice that $s'.\mathsf{t} \neq \bot \;\Rightarrow\; \mathbf{E}' \neq [\,]$). Otherwise, we have $\mathbf{E}[V_0]\,\mathcal{S}_{G,\ell}\,\mathbf{E}[V_1]$ by Lemma 5.2, and we conclude using Clause 11 if $\mathbf{E}' \neq [\,]$, and Clause 12 (and 2) otherwise.

- If $\mathbf{E} = (V := \mathbf{E}')$ then $G;\Gamma \vdash V : \bot, \theta'\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \theta'$ with $s'.\mathsf{r} \preceq_G \ell''$. Since $\ell' \preceq_{G\cup\lceil\mathbf{E}'\rceil} s'.\mathsf{r}$ by Lemma 4.7(i), we have $\ell'' \not\preceq_G \ell$, and we conclude using the induction hypothesis and Clause 12.

- If $\mathbf{E} = (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{E}')$ then we have $F, G;\Gamma \vdash \mathbf{E}'[(\mathbf{!}u_{\ell',\theta})] : s', \tau$. Since $\lceil\mathbf{E}\rceil = F \cup \lceil\mathbf{E}\rceil'$, we conclude using the induction hypothesis and Clause 13. ❏

Now we prove that the relation $\mathcal{R}_{G,\ell}$ is a kind of "weak bisimulation", with respect to the transition relation $\xrightarrow[F]{N}$:

Proposition 5.6. *If $M\,\mathcal{R}_{G,\ell}\,N$ with $M \notin \mathcal{H}_{G,\ell}$, and if $(M,\mu) \xrightarrow[F]{M''} (M',\mu')$ with $\mu \mathrel{\overset{\frown}{-}}^{F\cup G,\ell} \nu$, and $u_{\ell'} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν, then there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''} (N',\nu')$ with $M'\,\mathcal{R}_{G,\ell}\,N'$ and $\mu' \mathrel{\overset{\frown}{-}}^{G,\ell} \nu'$, with $\mathsf{dom}(\nu' - \nu) = \mathsf{dom}(\mu' - \mu)$.*

Proof: by induction on the definition of $\mathcal{R}_{G,\ell}$. The proposition is vacuously true if $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 1.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 2, that is $M = N$, then by Lemma 3.6 there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''} (N',\nu')$ with $\mu' \bumpeq^{F\cup G,\ell} \nu'$ and $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$. If $M' = N'$ then $M'\,\mathcal{R}_{G,\ell}\,N'$ by Clause 2 (and Subject Reduction). Otherwise by Lemma 2.3 we have $M = \mathbf{E}[(\mathbf{!}u_{\ell',\theta})]$ and $M' = \mathbf{E}[\mu(u_{\ell',\theta})]$ and $N' = \mathbf{E}[\nu(u_{\ell',\theta})]$ for some evaluation context $\mathbf{E}$, with $F = \lceil\mathbf{E}\rceil$. Since $\mu(u_{\ell',\theta}) \neq \nu(u_{\ell',\theta})$ we have $\ell' \not\preceq_{F\cup G} \ell$, and therefore $M'\,\mathcal{R}_{G,\ell}\,N'$ by Lemma 5.5.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 3, that is $M = (\mathsf{if}\ M_0\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $N_1 \mathrel{)\!(}^{G,\ell} N_2$ then we have $M_0 \notin \mathcal{H}_{G,\ell}$, by Lemma 3.9, since $M \notin \mathcal{H}_{G,\ell}$, and therefore $M' = (\mathsf{if}\ M_0'\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2)$ for some M_0' such that $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. Then we conclude using the induction hypothesis and Clause 3.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 4, that is $M = (M_0M_1)$ and $N = (N_0N_1)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$, M_0 and N_0 are (G,ℓ)-high functions, and $M_1 \mathrel{)\!(}^{G,\ell} N_1$, we notice that M cannot be a redex, with $M_0, M_1 \in \mathcal{V}al$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$ by Lemma 4.10. For the same reason, we have $M_0 \notin \mathcal{H}_{G,\ell}$, and therefore it must be the case that $M' = (M_0'M_1)$ for some M_0' such that $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. We easily conclude using the induction hypothesis (and Subject Reduction) and Clause 4.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 5, that is $M = (M_0M_1)$ and $N = (N_0N_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, M_0 and N_0 are (G,ℓ)-high functions, and $M_1\,\mathcal{R}_{G,\ell}\,N_1$, we notice that M cannot be a redex, with $M_0, M_1 \in \mathcal{V}al$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$ by the Lemma 4.10. Then either $M_0 \in \mathcal{V}al$ (hence also $N_0 \in \mathcal{V}al$, see Remark 5.1) and there exists M_1' such that $(M_1,\mu) \xrightarrow[F]{M''} (M_1',\mu')$ with $M' = (M_0M_1')$, or $M' = (M_0'M_1)$ for some M_0' such that $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. In the first case, we have $M_1 \notin \mathcal{H}_{G,\ell}$ (since otherwise we would have $M \in \mathcal{H}_{G,\ell}$), and we use the induction hypothesis and Clause 5 to conclude. In the second case, we use Proposition 5.3 and the induction hypothesis to conclude, by Clause 5.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 6, that is $M = M_0\,;M_1$ and $N = N_0\,;M_1$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M_1 \in \mathcal{H}_{G,\ell}$, then we have $M_0 \notin \mathcal{H}_{G,\ell}$ (otherwise we would have $M \in \mathcal{H}_{G,\ell}$). Therefore, there exists M_0' such that $M' = M_0'\ ;\ M_1$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$, and we conclude using the induction hypothesis and Clause 6.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 7, that is $M = M_0\,;M_1$ and $N = N_0\,;M_1$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, then there are two cases. If M_0 is a value and $M' = M_1$ (with $F = \emptyset$, $M'' = ()$ and $\mu' = \mu$) then $N_0 \in \mathcal{V}al$ by Remark 5.1, and $(N,\nu) \xrightarrow[\emptyset]{()} (M',\nu)$, and we conclude using Clause 2. Otherwise, there exists M_0' such that $M' = M_0'\,;M_1$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$, and we use Proposition 5.3 and Clause 7 to conclude.
- In the cases where $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 8 or 9, we argue as in Proposition 5.3.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 10, that is $M = (M_0 := M_1)$ and $N = (N_0 := M_1)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, and $M_1 \in \mathcal{H}_{G,\ell}$, then $M' = (M_0' := M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. We have $M_0 \notin \mathcal{H}_{G,\ell}$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$. Then by induction hypothesis there exist N_0' and ν' such that $(N_0,\nu) \xrightarrow[F]{M''} (N_0',\nu')$ with $M_0'\,\mathcal{R}_{G,\ell}\,N_0'$, $\mu' \bumpeq^{F\cup G,\ell} \nu'$, and $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$. If M_0' is not a value, then by Remark 5.4 N_0' is not a value, and we conclude by Clause 10 in this case. Otherwise, we conclude by Clause 12 (and 2, as regards M_1).

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 11, that is $M = (M_0 := M_1)$ and $N = (N_0 := M_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ where M_0 and N_0 are not values, and both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$ then, as in the previous case, $M' = (M_0' := M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$. By Proposition 5.3 there exist N_0' and ν' such that $(N_0,\nu) \xrightarrow[F]{M''} (N_0',\nu')$ with $M_0'\,\mathcal{S}_{G,\ell}\,N_0'$, $\mu' \bumpeq^{F\cup G,\ell} \nu'$, and $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$. If M_0' is not a value, then by Remark 5.1 N_0' is not a value, and we conclude by Clause 11 in this case. Otherwise, we conclude by Clause 12 (and 2, as regards M_1).

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 12, that is $M = (V := M_0)$ and $N = (V' := N_0)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and V and V' both have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, then $M_0 \notin \mathcal{H}_{G,\ell}$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$. Then in particular M_0 is a not value, and therefore there exists M_0' such that $M' = (V := M_0')$ with $(M_0,\mu) \xrightarrow[F]{M''} (M_0',\mu')$, and we use the induction hypothesis and Clause 12 to conclude.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 13, that is $M = (\mathsf{flow}\ F'\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F'\ \mathsf{in}\ N_0)$ with $M_0\,\mathcal{R}_{F'\cup G,\ell}\,N_0$ then $M_0 \notin \mathcal{H}_{G,\ell}$ (hence also $M_0 \notin \mathcal{H}_{F'\cup G,\ell}$) since $M \notin \mathcal{H}_{G,\ell}$, and therefore $M' = (\mathsf{flow}\ F'\ \mathsf{in}\ M_0')$ for some M_0' such that $(M_0,\mu) \xrightarrow[F'']{M''} (M_0',\mu')$ with $F = F' \cup F''$. Then we conclude using the induction hypothesis and Clause 13. ❏

To conclude the proof of the Soundness Theorem, it remains to exhibit an appropriate bisimulation. Let $\mathcal{R}^\star_{G,\ell}$ be the relation inductively defined as follows:

$$\frac{M\,\mathcal{R}_{G,\ell}\,N}{M\,\mathcal{R}^\star_{G,\ell}\,N} \qquad \frac{P \mathrel{)\!\!-\!\!(}^{G,\ell} Q \quad Q\,\mathcal{R}^\star_{G,\ell}\,R}{P\,\mathcal{R}^\star_{G,\ell}\,R} \qquad \frac{P\,\mathcal{R}^\star_{G,\ell}\,Q \quad Q \mathrel{)\!\!-\!\!(}^{G,\ell} R}{P\,\mathcal{R}^\star_{G,\ell}\,R}$$

$$\frac{P\,\mathcal{R}^\star_{G,\ell}\,P' \qquad Q\,\mathcal{R}^\star_{G,\ell}\,Q'}{(P \parallel Q)\,\mathcal{R}^\star_{G,\ell}\,(P' \parallel Q')}$$

Proposition 5.7. *The relation $\mathcal{R}^\star_{G,\ell}$ is a (G,ℓ)-bisimulation.*

Proof: first, it is easy to see, by induction on the definition of $\mathcal{R}^\star_{G,\ell}$, that this relation is symmetric. Now we show, by induction on the definition of $\mathcal{R}^\star_{G,\ell}$, that if $P\,\mathcal{R}^\star_{G,\ell}\,Q$ and $(P,\mu) \xrightarrow[F]{} (P',\mu')$ with $\mu \bumpeq^{F\cup G,\ell} \nu$ and $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν, then there is a matching transition from (Q,ν). We proceed by induction on the inference of $P\,\mathcal{R}^\star_{G,\ell}\,Q$.

- If P and Q are expressions, with $P\,\mathcal{R}_{G,\ell}\,Q$, we distinguish two cases. If $P \in \mathcal{H}_{G,\ell}$ then $P' \in \mathcal{H}_{G,\ell}$, hence $P'\,\mathcal{R}^\star_{G,\ell}\,Q$ since $P' \mathrel{)\!\!-\!\!(}^{G,\ell} P\,\mathcal{R}_{G,\ell}\,Q$, and a matching transition for Q is $(Q,\nu) \xrightarrow{*} (Q,\nu)$ since $\mathsf{dom}(\mu') \cap \mathsf{dom}(\nu) = \mathsf{dom}(\mu) \cap \mathsf{dom}(\nu)$. Otherwise, by Lemma 2.2(ii), either P' is an expression and $(P,\mu) \xrightarrow[F]{()} (P',\mu')$ or $(P,\mu) \xrightarrow[F]{M''} (M',\mu')$ for some M' and M'' such that $P' = (M' \parallel M'')$, with $M''\,\mathcal{R}_{G,\ell}\,M''$ since M'' is typable. In both cases we use Proposition 5.6, Lemma 2.2(i) and the definition of $\mathcal{R}^\star_{G,\ell}$ to conclude.

- If $P \mathrel{)\!\!-\!\!(}^{G,\ell} R\,\mathcal{R}^\star_{G,\ell}\,Q$ then $P' \in \mathcal{H}_{G,\ell}$, and therefore $P' \mathrel{)\!\!-\!\!(}^{G,\ell} R\,\mathcal{R}^\star_{G,\ell}\,Q$, hence $P'\,\mathcal{R}^\star_{G,\ell}\,Q$, and a matching transition for Q is $(Q,\nu) \xrightarrow{*} (Q,\nu)$.

- If $P\,\mathcal{R}^{\star}_{G,\ell}\,R \mathrel{)\!(}^{G,\ell} Q$ then by induction hypothesis there exist R' and μ'' such that $(R,\mu) \xrightarrow{*} (R',\mu'')$ with $P'\,\mathcal{R}^{\star}_{G,\ell}\,R'$, $\mu' \mathrel{\overset{\frown}{-}}^{G,\ell} \mu''$ and $\mathsf{dom}(\mu''-\mu) \subseteq \mathsf{dom}(\mu'-\mu)$. We have $R' \in \mathcal{H}_{G,\ell}$, and therefore $P'\,\mathcal{R}^{\star}_{G,\ell}\,R' \mathrel{)\!(}^{G,\ell} Q$, hence $P'\,\mathcal{R}^{\star}_{G,\ell}\,Q$, and a matching transition for Q is $(Q,\nu) \xrightarrow{*} (Q,\nu)$, since $\mathsf{dom}(\mu') \cap \mathsf{dom}(\nu) = \mathsf{dom}(\mu) \cap \mathsf{dom}(\nu)$.
- If $(P,\mu) \xrightarrow[F]{} (P',\mu')$ with $P = (P_0 \parallel P_1)$ and $P_i\,\mathcal{R}^{\star}_{G,\ell}\,Q_i$ then we have, for $i=0$ or $i=1$, $(P_i,\mu) \xrightarrow[F]{} (P'_i,\mu')$ with $P' = (P'_0 \parallel P_1)$ if $i=0$, and $P' = (P_0 \parallel P'_1)$ if $i=1$. Then we use the induction hypothesis and the definition of $\mathcal{R}^{\star}_{G,\ell}$ to conclude that $((Q_0 \parallel Q_1),\nu)$ has a transition matching the one of (P,μ). ❏

The Soundness Theorem is an immediate corollary of this last result.

To conclude this section, let us see some implications of what we have proved. Looking closely at Proposition 5.6, we see that we have shown a much more precise result than the one stated by the Soundness Theorem. Indeed, let us define the *strong non-disclosure policy* as follows.

Definition (Quasi-Strong Bisimulation) 5.8. *A* quasi-strong (G,ℓ)-bisimulation *is a symmetric relation $\mathcal{R}$ on* expressions *such that if $M\,\mathcal{R}\,N$ then either*

(i) *M and N are both (G,ℓ)-high, i.e. $M,N \in \mathcal{H}_{G,\ell}$, or*

(ii) *if $(M,\mu) \xrightarrow[F]{M''} (M',\mu')$ with $\mu \mathrel{\overset{\frown}{-}}^{F\cup G,\ell} \nu$, and $u_{\ell'} \in \mathsf{dom}(\mu'-\mu)$ implies that u is fresh for ν, then there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''} (N',\nu')$ with $M''\,\mathcal{R}\,M''$, $M'\,\mathcal{R}\,N'$ and $\mu' \mathrel{\overset{\frown}{-}}^{G,\ell} \nu'$, with $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$.*

This is a generalization of the quasi-strong bisimulations of [6]. We use the word "quasi-strong" because, in case a transition on one side has to be matched by the other, this must be achieved in a single step. Moreover, the thread that is spawned, if any, and the local flow relation that is in force in the two matching transitions must be the same.

Definition (The Strong Non-Disclosure Policy) 5.9. *An expression M satisfies the* strong non-disclosure policy, *or is* strongly secure *with respect to the flow policy G if for any ℓ there exists a quasi-strong (G,ℓ)-bisimulation $\mathcal{R}$ such that $M\,\mathcal{R}\,M$.*

Then we proved that if an expression is typable, it is strongly secure. Notice that, according to these definitions, any thread spawned by a strongly secure expression is strongly secure.

The definition above can be stated for any language where the semantics may be described using transitions of the form $(M,\mu) \xrightarrow[F]{N} (M',\mu')$. There are several advantages in using such a definition. First of all, since the transitions $(M,\mu) \xrightarrow[F]{N} (M',\mu')$ are deterministic (up to the choice of the name of created references), it is generally easy to prove that an expression is not strongly secure – hence not typable. For instance, if, starting with the expression $M_0 = M$, there are two sequences of transitions

$$\begin{array}{c}(M_0,\mu_0) \xrightarrow[F_0]{N_0} (M_1,\mu'_0),\ (M_1,\mu_1) \xrightarrow[F_1]{N_1} (M_2,\mu'_1),\ldots,(M_n,\mu_n) \xrightarrow[F_n]{N_n} (M_{n+1},\mu')\\ (M_0,\nu_0) \xrightarrow[F_0]{N_0} (M_1,\nu'_0),\ (M_1,\nu_1) \xrightarrow[F_1]{N_1} (M_2,\nu'_1),\ldots,(M_n,\nu_n) \xrightarrow[F_n]{N_n} (M_{n+1},\nu')\end{array}$$

with $0 \leqslant i \leqslant n \ \Rightarrow\ \mu_i \mathrel{\overset{\frown}{-}}^{F_i\cup G,\ell} \nu_i$ (with some obvious conditions on the created references) and $\mu' \not\mathrel{\overset{\frown}{-}}^{G,\ell} \nu'$, then M is not strongly secure. This applies for instance to Examples (4), (5), (8), (9),

(10) and (13) of Section 3. Moreover, if we define *strong non-interference* in a similar way, ignoring the local flow relations, that is:

Definition (The Strong Non-Interference Property) 5.10. *An expression M satisfies the* strong non-interference property *with respect to the flow policy G if for any ℓ there exists a relation $\mathcal{R}$ such that $M \mathrel{\mathcal{R}} M$, which satisfies: if $M_0 \mathrel{\mathcal{R}} M_1$ then either*

(i) *M_0 and M_1 are both (G,ℓ)-high, i.e. $M_0, M_1 \in \mathcal{H}_{G,\ell}$, or*

(ii) *if $(M_0,\mu) \xrightarrow{N} (M_0',\mu')$ with $\mu \doteq^{G,\ell} \nu$, and $u_{\ell'} \in \mathsf{dom}(\mu' - \mu)$ implies that u is fresh for ν, then there exist M_1' and ν' such that $(M_1,\nu) \xrightarrow{N} (M_1',\nu')$ with $N \mathrel{\mathcal{R}} N$, $M_0' \mathrel{\mathcal{R}} M_1'$ and $\mu' \doteq^{G,\ell} \nu'$, with $\mathsf{dom}(\nu' - \nu) = \mathsf{dom}(\mu' - \mu)$.*

then the following facts are immediate:

(i) if M satisfies the strong non-interference property, then it is strongly secure;

(ii) if M is strongly secure and does not use declassification, that is, M does not contain a subexpression (flow F in N), then it satisfies the strong non-interference property.

6. Conclusion and related work

We have proposed a way to face the "*challenge* [of] *determining what the nature of a downgrading mechanism should be and what kinds of security guarantees it permits*" [53]. Taking the view that one should distinguish the questions of *what* or *how much* can be revealed, from that of *how* it can be revealed, we addressed the second question by proposing a simple and powerful construct for declassification based on dynamically varying flow policies. Although this idea has already been mentioned in the literature (see [44]), it does not seem to have been previously studied in a formal way (in [45] it is shown that if the downgrading relations used in a program do not modify the security lattice under some level ℓ, then the program is secure up to this level). Our main achievement is the design of a security property that is a natural generalization of classical non-interference, based on dynamic flow policies. We notice that the idea of stating the security property in a way that reflects the *local* nature of declassification, that is, using a decorated small-step semantics, could perhaps be used in other settings. Moreover, we have shown that, for programs written in an expressive, higher-order imperative core language, this property can be enforced by static analysis. The idea of using a construct for dynamically introducing flow policies can certainly be applied to various other programming paradigms.

The language-based security approach is now well established – though not widely used –, and there is no question that lattices of information flow provide a good basis for ensuring end-to-end confidentiality properties. Indeed, checking that a program does not violate a given flow policy by means of a type system which enforces the extensional property of non-interference, is regarded as providing a reasonable security guarantee. Therefore we believe that checking, piece by piece, that a program does not violate *local* flow policies should provide a similar guarantee, while allowing us to deal with declassification. Borrowing an example from [52], we may then write a piece of code to release precious information held in a safe place A by *Alice*, to *Bob* who wishes to purchase it, provided a payment has indeed been done:

$$\mathsf{if}\ \mathit{paid}\ \mathsf{then}\ (\mathsf{flow}\ \mathit{Alice} \prec \mathit{Bob}\ \mathsf{in}\ B := \mathbf{!}\, A)\ \mathsf{else}\ \cdots$$

without checking further that this intended leak of information is justified. In some other cases, where a program has to be certified against high security standards for instance, one would have to provide a formal proof that a declassified portion of code satisfies some specification, like that of not releasing too much information. This justification is left to the programmer, whereas our programming language design provides him with a flexible programming construct for declassification, together with a static checking technique to prevent some errors.

In this respect, our approach contrasts with most previous works on declassification in a language-based security setting that aimed at imposing constraints, at a linguistic level, on this operation – sometimes without justifying such constraints, for lack of an extensional notion of security. For instance, in [46, 49], Volpano and Smith restrict downgrading to occur by means of specific "hard" functions. This is certainly relevant for some applications, especially those involving cryptography, but is less appropriate for applications where the programmer intends to let information leak in some places (like in the example above). Another example of constrained downgrading is *robust declassification* which was proposed, and then studied in a series of papers [29, 30, 31, 32, 44, 52] by Myers and colleagues. The idea of robust declassification is to allow this operation only for extending the reading clearances assigned by the owner of an object, and to control it by requiring that this operation runs under appropriate authority. This was first conceived as a run-time constraint, and was later approximated in a type system by means of integrity levels. Compared to our approach, robust declassification is obviously more restrictive (for instance one can only set up flows from lower to higher levels). However, it would be interesting to see whether we can accomodate our setting to deal with it (even though it is not very clear that the "robustness property" of [32] is related to our non-disclosure property). An obvious idea would be to restrict the use of $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ to the case where F only allows to declassify the contents of references that have been created by the thread executing this piece of code, and then to see how our security property can be made more accurate for this case.

In another paper [39], Sabelfeld and Myers introduce a different way of restricting declassification, with the idea that downgrading is acceptable provided that the program does not modify data if that could influence the value of declassified expressions, therefore addressing the question of *what* is downgraded. For instance, the program $u_H := \mathit{ff}\ ;\ v_L := \mathsf{declassify}(u_H, L)$ is not regarded as safe according to the definition of *delimited release.* On the other hand, a program like $v_L := \mathsf{declassify}(u_H, L)\ ;\ w_L := u_H$, which is similar to the one of example (4), is considered safe, but is ruled out by the type system. The type system, based on the idea that "*variables under declassification may not be updated prior to declassification*", might be difficult to extend to a more sophisticated language, with a less predictable order of execution than the one considered in [39].

More recently, Chong and Myers introduced *declassification policies* [8], that specify the levels through which a value can be downgraded. This also involves conditions, which are supposed to be satisfied in order to perform the declassification steps. These are used in the definition of a generalized noninterference property to mark the steps where declassification occurs. This bears some resemblance to our transitions labelled by a local flow relation, although conditions are rather used to single out sequences of steps that do not involve downgrading operations. The declassification policies of [8] look a bit inflexible since, as far as we can see, there is no possibility for a value to be used in another way than the one prescribed by the specific policy assigned to it. Therefore it seems that, with these policies, the programmer must accurately anticipate the run-time behaviour of the declassified values. By contrast, in our setting a reference can be involved in various declassification scenarios, and this does not have to be reflected in its type.

Closer to ours is the work by Ferrari & al. [13], who proposed to attach "*waivers*" to methods in an object-oriented language to provide a way of making information flow from objects to users. Although the authors claim that "*only priviledged methods*" have associated waivers, there seems to be actually no constraint on the flow of information they allow. This idea of a waiver is therefore similar to a local flow relation, though it is not clear whether the notion of "*safe information flow*" that the authors define is similar to our non-disclosure property (as far as we can see, this definition does not treat waivers as having a local scope). A work that is also close to ours, at least as regards the motivations, is the one by Li and Zdancewic [24]. After having made the initial decision that "*instead of studying* who *can downgrade the data* [like in the work on robust declassification]*, we take an orthogonal direction and study* how *data can be downgraded*", they intend to offer the programmer a way of specifying sophisticated *downgrading policies.* Therefore we can say we share the same motivations. However, the ways we take from this starting point differ considerably. Li and Zdancewic introduce a very sophisticated notion of a downgrading policy (an expression in a typed λ-calculus), where we use flow relations between principals, which look easy to use in practice. Our non-disclosure property also looks simpler than the notion of *relaxed noninterference*, which is based on program equivalence (in the language of downgrading policies). Their main result is again very close in its spirit to ours, since "*the security guarantee* [provided by relaxed noninterference] *only assures that the program respects the user's security policies*". Therefore it will be interesting to compare in greater details the two approaches, especially from the point of view of expressiveness.

Finally, the work that is the closest to ours is the one by Mantel and Sands [27]. In addition to a given lattice structure of security levels, they consider an extra relation on these levels, that can be used in specific instructions – namely, assignments of the form $v_{\ell'} := (!\, u_\ell)$ – to downgrade information: in such an instruction, the flow from ℓ to ℓ' should be allowed by the "exceptional" flow relation. In our syntax, we would write this as ($\mathsf{flow}\ \ell \prec \ell'\ \mathsf{in}\ v_{\ell'} := (!\, u_\ell)$) (where $\ell \prec \ell'$ means $\{\, p \prec q \mid p \in \ell \ \&\ q \in \ell' \,\}$). Then Mantel and Sands introduce a security property generalizing classical non-interference, defined by means of a notion of bisimulation with respect to transitions annotated by a flow relation, and they show a type soundness result. Therefore one can see that this is very close to what we did in this paper (the two works were done independently, and a precise comparison of our security properties remains to be made). There are some differences, however. A first difference is that Mantel and Sands choose to restrict declassification to very specific instructions, whereas we allow any computation to be declassified. From a pragmatic point of view, the main difference we see is that in their work, declassification is governed by a specific global flow policy – the "exceptional" flow relation – that cannot be manipulated by programs. We think that it could be useful in practice to have the ability of choosing various ways of downgrading, depending on the point in the program where this is performed, without necessarily complying with a predetermined, global downgrading policy. Moreover, such a dynamic view seems to be needed in order to deal with mobile code, where agents migrate with their own flow policy. Another noticeable – though not related to declassification – difference is that we are using a higher-order imperative language, whereas Mantel and Sands consider a simple while language with threads, where there is no interaction between commands and expressions. Moreover, our type system appears to be less restrictive (as regards the while construct for instance).

Some obvious topics for further investigations are polymorphism and type inference [29, 33], dynamic labels [44, 55], and more generally first-class security levels. One could also wish to deal with a richer set of effects, including for instance the creation and deletion of references, the creation of threads, and more generally any action that modifies the context of an expression in the (abstract) machine evaluating it. We are currently working on using the idea of local flow policies in a mobile code setting, and more precisely in the ULM language [5]. Indeed, a mobile agent may carry its own flow policy, and run in various sites, each having their own, local flow policies, and therefore this is a scenario where one has to deal with various flow relations. Regarding declassification, one may think our approach is too permissive, since it allows any program to declassify anything, provided that no other flow than the declared ones is implemented. Therefore it would be interesting to see how we could restrict the usage of the flow declaration construct in some sensible ways, and adapt the non-disclosure policy accordingly. We have mentioned a possible way of doing this in discussing the work on robust declassification. It would also be interesting to find a simple notion of "security error", that could be used as a basis for designing error messages in a type inference approach. Finally, we observe that, following Biba's remark that integrity is dual to confidentiality in some sense (see [23, 31]), we may design a framework for the integrity aspect of security in a similar way to what we did for confidentiality. It could support, in particular, dowgrading facilities like the "endorse" construct of [23] (which is also considered in [32], but with a different semantics). Although the expectations are even stronger regarding integrity than confidentiality (see [23] for instance), it would be interesting, from a practical point of view (*cf.* [53]), to have in a programming language such flexible downgrading facilities with respect to integrity.

References

[1] J. Agat, *Transforming out timing leaks,* POPL'00 (2000) 40-53.

[2] G.R. Andrews, R.P. Reitman, *An axiomatic approach to information flow in programs,* ACM TOPLAS, Vol. 2 No. 1 (1980) 56-76.

[3] D.E. Bell, L.J. La Padula, *Secure computer system: unified exposition and Multics interpretation,* Mitre Corp. Rep. MTR-2997 Rev. 1 (1976).

[4] A. Bossi, C. Piazza, S. Rossi, *Modelling downgrading in information flow security,* CSFW'04 (2004).

[5] G. Boudol, ULM*, a core programming model for global computing,* ESOP'04, Lecture Notes in Comput. Sci. 2986 (2004) 234-248.

[6] G. Boudol, I. Castellani, *Non-interference for concurrent programs and thread systems,* Theoretical Comput. Sci. Vol. 281, No. 1 (2002) 109-130.

[7] D. Clark, S. Hunt, P. Malacaria, *Quantified interference: information theory and information flow,* WITS'04 (2004).

[8] S. Chong, A.C. Myers, *Security policies for downgrading,* 11th ACM Conf. on Computer and Communications Security (2004).

[9] E. Cohen, *Information transmission in computational systems,* 6th ACM Symp. on Operating Systems Principles (1977) 133-139.

[10] K. Crary, A. Kliger, F. Pfenning, *A monadic analysis of information flow security with mutable state,* J. of Functional Programming, Vol. 15 No. 2 (2005) 249-291.

[11] D.E. Denning, *A lattice model of secure information flow,* CACM Vol. 19 No. 5 (1976) 236-243.

[12] A. Di Pierro, C. Hankin, H. Wiklicky, *Approximate non-interference,* CSFW'02 (2002) 1-15.

[13] E. Ferrari, P. Samarati, E. Bertino, S. Jajodia, *Providing flexibility in information flow control for object-oriented systems,* IEEE Symp. on Security and Privacy (1997) 130-140.

[14] R. Focardi, R. Gorrieri, *A classification of security properties for process algebras,* J. of Computer Security, Vol. 3 No. 1 (1995) 5-33.

[15] R. Focardi, S. Rossi, *Information flow security in dynamic contexts,* CSFW'01 (2001) 307-319.

[16] J.A. Goguen, J. Meseguer, *Security policies and security models,* IEEE Symp. on Security and Privacy (1982) 11-20.

[17] N. Heintze, J. Riecke, *The SLam calculus: programming with secrecy and integrity,* POPL'98 (1998) 365-377.

[18] A.K. Jones, R.J. Lipton, *The enforcement of security policies for computation,* 5th ACM Symp. on Operating Systems Principles (1975) 197-206.

[19] B.W. Lampson, *A note on the confinement problem,* CACM Vol. 16 No. 10 (1973) 613-615.

[20] P.J. Landin, *The mechanical evaluation of expressions,* Computer Journal Vol. 6 (1964) 308-320.

[21] P. Laud, *Semantics and program analysis of computationally secure information flow,* ESOP'01, Lecture Notes in Comput. Sci. 2028 (2001) 77-91.

[22] P. Laud, *Handling encryption in an analysis for secure information flow,* ESOP'03, Lecture Notes in Comput. Sci. 2618 (2003) 159-173.

[23] P. Li, Y. Mao, S. Zdancewic, *Information integrity policies,* Formal Aspects of Security and Trust Workshop (2003).

[24] P. Li, S. Zdancewic, *Downgrading policies and relaxed noninterference,* POPL'05 (2005) 158-170.

[25] G. Lowe, *Semantic models of information flow,* Theoretical Comput. Sci. 315 (2004) 209-256.

[26] J.M. Lucassen, D.K. Gifford, *Polymorphic effect systems,* POPL'88 (1988) 47-57.

[27] H. Mantel, D. Sands, *Controlled declassification based on intransitive noninterference,* APLAS'04, Lecture Notes in Comput. Sci. 3302 (2004) 129-145.

[28] R. Milner, M. Tofte, R. Harper, D. MacQueen, *The definition of Standard ML (Revised),* The MIT Press (1997).

[29] A. Myers, *JFlow: practical mostly-static information flow control,* POPL'99 (1999).

[30] A.C. Myers, B. Liskov, *A decentralized model for information flow control,* ACM Symp. on Operating Systems Principles (1997) 129-142.

[31] A.C. Myers, B. Liskov, *Protecting privacy using the decentralized label model,* ACM Trans. on Soft. Eng. and Methodology, Vol. 9 No. 4 (2000) 410-442.

[32] A.C. Myers, A. Sabelfeld, S. Zdancewic, *Enforcing robust declassification,* CSFW'04 (2004).

[33] F. Pottier, V. Simonet, *Information flow inference for ML,* ACM TOPLAS Vol. 25 No. 1 (2003) 117-158.

[34] A.W. Roscoe, M.H. Goldsmith, *What is intransitive noninterference?,* CSFW'99 (1999).

[35] J. Rushby, *Noninterference, transitivity, and channel-control security policies,* Comput. Sci. Lab. SRI International, Tech. Rep. CSL-92-02 (1992).

[36] P. Ryan, J. McLean, J. Millen, V. Gligor, *Non-interference, who needs it?,* CSFW'01 (2001).

[37] A. Sabelfeld, D. Sands, *Probabilistic noninterference for multi-threaded programs,* CSFW'00 (2000).

[38] A. Sabelfeld, A.C. Myers, *Language-based information-flow security,* IEEE J. on Selected Areas in Communications Vol. 21 No. 1 (2003) 5-19.

[39] A. Sabelfeld, A.C. Myers, *A model for delimited information release,* Intern. Symp. on Software Security, Lecture Notes in Comput. Sci. to appear (2003).

[40] R.S. Sandhu, *Lattice-based access control models,* IEEE Computer Vol. 26 No. 11 (1993) 9-19.

[41] V. Simonet, *The Flow Caml system: documentation and user's manual,* INRIA Tech. Rep. 0282 (2003).

[42] G. Smith, *A new type system for secure information flow,* CSFW'01 (2001).

[43] G. Smith, D. Volpano, *Secure information flow in a multi-threaded imperative language,* POPL'98 (1998).

[44] S. Tse, S. Zdancewic, *Run-time principals in information-flow type systems,* IEEE Symp. on Security and Privacy (2004).

[45] S. Tse, S. Zdancewic, *A design for a security-typed language with certificate-based declassification,* ESOP'05, Lecture Notes in Comput. Sci. to appear (2005).

[46] D. Volpano, *Secure introduction of one-way functions,* CSFW'00 (2000) 246-254.

[47] D. Volpano, G. Smith, *Eliminating covert flows with minimum typings,* CSFW'97 (1997) 156-168.

[48] D. Volpano, G. Smith, *Probabilistic noninterference in a concurrent language,* CSFW'98 (1998) 34-43.

[49] D. Volpano, G. Smith, *Verifying secrets and relative secrecy,* POPL'00 (2000) 268-276.

[50] D. Volpano, G. Smith, C. Irvine, *A sound type system for secure flow analysis,* J. of Computer Security, Vol. 4, No 3 (1996) 167-187.

[51] A. Wright, M. Felleisen, *A syntactic approach to type soundness,* Information and Computation Vol. 115 No. 1 (1994) 38-94.

[52] S. Zdancewic, *A type system for robust declassification,* MFPS'03, ENTCS Vol. 83 (2003).

[53] S. Zdancewic, *Challenges for information-flow security,* PLID'04 (2004).

[54] S. Zdancewic, A. C. Myers, *Secure information flow via linear continuations,* HOSC Vol. 15 No. 2-3 (2002) 209-234.

[55] L. Zheng, A. C. Myers, *Dynamic security labels and noninterference,* Formal Aspects of Security and Trust Workshop (2004).