On Declassification and the Non-Disclosure Policy (*)

Ana Almeida Matos (†) and Gérard Boudol

INRIA Sophia Antipolis, France

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. 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. Besides dealing with declassification, our type system improves over previous systems for checking information flow in two directions: first, we show that the typing of terminations leaks can be largely improved, by particularizing the case where the alternatives in a conditional branching both terminate. Moreover, we also provide a quite precise way of approximating the confidentiality level of an expression, by ignoring the level of values that are only used for side-effects.

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 [14, 23, 35, 41] 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.

Since Bell and La Padula and Denning's pioneering works [5, 14], the classical approach to secure information flow is to use a *lattice of security levels* (see for instance the survey [45] for the use of security lattices). The "objects" – information containers – 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 [12], and is also referred to as *non-interference* according to the terminology used by Goguen and Meseguer in [19].

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

(†) Current affiliation: SQIGT-IT and IST Lisbon, Portugal

A lot of work has been devoted to the design of methods for analyzing information flow in programs (see for instance [4] 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 [41]). More importantly, the failure of a run-time check can serve as a covert channel [14, 22, 33]. 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 [55]. 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, culminating with Jif (or JFlow, see [32]) and Flow Caml [46] as regards the size of the language. See e.g. [8, 13, 20, 37, 47, 48, 52, 55, 60], and further references in the survey [41] (which contains a complete review of results and issues, up to year 2002, in the area of language-based information-flow security). In this paper we shall base our study on Core ML [31, 56], a call-by-value λ-calculus extended with imperative constructs that we enrich with concurrent threads. In this setting, the "information containers," to which security levels are assigned, are memory locations – references, in ML's jargon (in an extended setting, that could also be files, entries in a database, or library functions).

Approaching declassification

As suggested above, language-based information-flow security is a well established theory. However, this theory is not yet widely used, and there are still a number of issues to investigate in order to make it practically useful – see [59] for a review of some of the challenges. One of the well-known challenges is to design new security properties and static analysis techniques to accept programs that intentionally *declassify* information. Declassifying programs are quite common and very useful. 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 (though with no profit, unless one has the decryption key). These programs are, by definition, ruled out by the non-interference requirement, which is therefore too strong to be used in practice (this observation was made very early, e.g. in [22]. See also [40]). To support programming software using declassification, we need to devise new security policies that are more flexible than non-interference, and then to provide the programmer with means to check that his code implements only the intended information flows. This is what we plan to do here.

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 in Section 6. In this paper we propose 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, e.g. 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?

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 [10, 15, 25, 26, 51, 54], 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. More precisely, our purpose is to provide the programmer with the help of static analysis techniques for developing security-minded software that involves declassification operations. In this way, he will be able to detect programming errors, like violating security policies.

Clearly, following a programming language approach, one cannot hope to provide much help as regards the semantical justification of programs. For instance, from a programming language perspective a wrong password checking procedure that always returns "ok" should be as acceptable as a correct one, that returns a "yes/no" answer (or as another incorrect one, that also returns a "yes/no" answer, but based on an incorrect checking of the association of passwords with users names). Again, it is the programmer's task, not the language's one, to justify the semantical correctness of the code, and in this paper we do not address the issue of developing tools for this purpose. Still, the question remains of defining what is secure information flow when declassification is a feature of the language.

The flow declaration construct

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 $A(lice)$ and $B(ob)$, then – using the ML notation $!\,x$ for dereferencing, and x_A, y_B for memory locations with confidentiality level A and B – a program like:

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

is legal, since in the context of the relation $A \prec B$, information is allowed to flow from A to B[1]. 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 $y_B := (\mathsf{flow}\ A \prec B\ \mathsf{in}\ !\,x_A)$ is also legal, and has the same meaning as the previous one. Another example is

$$(\mathsf{flow}\ A \prec B\ \mathsf{in}\ M)\,;(\mathsf{flow}\ B \prec C\ \mathsf{in}\ N) \tag{2}$$

that shows a way to achieve a kind of non-transitive flow relation (see [6, 38, 39]). In this case, B acts as a trusted "downgrader" principal, mediating information flow between A and C which are otherwise unrelated, or such that $C \prec A$ (*cf.* [39]).

[1] In what follows, security levels will be sets of principals, and flows like $A \prec B$ will relate principals. Then the example above should really be written ($\mathsf{flow}\ A \prec B\ \mathsf{in}\ y_{\{B\}} := !\,x_{\{A\}}$).

To very briefly compare our local flow declaration construct with other proposals for declassification (more complete comparisons are given in Section 6), let us first notice that a similar construct 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 [49] under the name of "delegation", and a "distributed banking" example is given which illustrates its use, but it was not formally studied there. The $\mathsf{declassify}(M, \ell)$ operation, that is used in some languages (see [32, 42] for instance) to downgrade the value of M to the confidentiality level ℓ, may be represented as – again using ML notations:

$$\mathsf{let}\ x = (\mathsf{ref}_H\ M)\ \mathsf{in}\ (\mathsf{flow}\ H \prec \ell\ \mathsf{in}\ !\,x)$$

where $(\mathsf{ref}_H\ M)$ creates a new memory address with security level H and contents M (that is, the value of M), and 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. Our flow declaration construct is more general than $\mathsf{declassify}(M, \ell)$ in two respects: first, with $(\mathsf{flow}\ F\ \mathsf{in}\ \cdots)$, the scope of the local flow policy F may enclose a whole computation (see [49] for an example), not just a value; second, the flow declaration construct allows us to express more precise ways of declassifying, by specifying the levels from which information may flow, like for instance in the examples above. This also allows us to encode a construct that is more general than $\mathsf{declassify}$, and offers more control on the security level of the declassified information, namely the *coercion* construct $[\ell_1 \preceq \ell_2]M$ that was considered in [21] (though not for declassification purposes), as follows:

$$\mathsf{let}\ x = (\mathsf{ref}_{\ell_1}\ M)\ \mathsf{in}\ (\mathsf{flow}\ \ell_1 \prec \ell_2\ \mathsf{in}\ !\,x)$$

Notice that an expression $\mathsf{declassify}(M, \ell)$ – that is $[H \preceq \ell]M$ – is always accepted in our language (provided that M is acceptable), while for instance $[\ell_0 \preceq \ell](!\,x_{\ell_1})$ is only accepted if $\ell_1 \preceq \ell_0$. Indeed, $[\ell_1 \preceq \ell_2]M$ casts M as having security level ℓ_2, provided that the level of M is lower than ℓ_1.

The non-disclosure policy

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 $(P, \mu) \to (P', \mu')$ 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:

$$(P, \mu) \xrightarrow[F]{} (P', \mu') \tag{3}$$

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.* (We could also call this "local non-interference.") More precisely, given that P performs the transition (3) 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 should be a sequence of transitions $(P, \nu) \xrightarrow{*} (P'', \nu')$ 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 $A \prec B$.

Outline

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. This was presented, without proofs, in the workshop paper [3]. 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. This system, improving upon the one of [3] by accepting more programs while still preserving the security property, was introduced in [7]. It provides a new way to type termination leaks, and a refined notion of the confidentiality level of an expression, showing that the level of sub-expressions used for side-effect only is irrelevant. 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

The information flow analysis we are aiming at relies, as usual, on a notion of security level, and on assigning such levels to memory locations – also called *references*. (More generally, one would classify in this way any "container" in which information is stored, like files, database entries, libraries, and so on.) As we said in the Introduction, we will use dynamically evolving flow relations between security levels 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

$$x \curlywedge y \preceq x \qquad\qquad x \preceq x \curlyvee y$$

$$x \curlywedge y \preceq y \qquad\qquad y \preceq x \curlyvee y$$

$$z \preceq x \ \&\ z \preceq y \ \Rightarrow\ z \preceq x \curlywedge y \qquad\qquad x \preceq z \ \&\ y \preceq z \ \Rightarrow\ x \curlyvee y \preceq z$$

It is easy to see that the quotient of a pre-lattice by the equivalence relation $\simeq$ given by $x \simeq y \ \Leftrightarrow\ x \preceq y \ \&\ y \preceq x$ is a lattice with respect to the quotient ordering.

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. Reverse inclusion of security levels may be interpreted as indicating allowed flows of information: if a reference u is labelled ℓ, and $\ell \supseteq \ell'$ then the value of u may be transferred to a reference v labelled ℓ', since the programs allowed to read this value from v were already allowed to read it from u.

The dynamically varying information flow policies are determined by relations on principals. Namely, 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:

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

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

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

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*

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

We notice that, if we only consider labels with only one "owner" in the DLM (decentralized label model) of Myers and Liskov [33], then our security lattices (quotients of security pre-lattices) coincide with the ones introduced in [34], where the "acts for" hierarchy is what we call here a flow policy. The following observation should be useful for type inference purposes:

Remark 2.2. *For any G and F, and for any security level ℓ, there exists a level $\bar{\ell}$ that is minimal with respect to $\preceq_G$ among the ℓ' such that $\ell \preceq_{F \cup G} \ell'$, namely $\bar{\ell} = \bigcup\{\, \ell' \mid \ell \preceq_{F \cup G} \ell' \,\}$.*

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

Figure 1: Syntax

2.2 Syntax and operational semantics

The language we consider is a higher-order language with mutable state à la ML, extended with a construct for dynamically creating concurrent threads, and a construct for 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 (which we would normally assume to be non-empty, but such an assumption is not used in the technical developments that follow). 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 explained in Section 4, we do not regard sequential composition as a derived construct, and we do not regard the imperative constructs ref, ! and := as first-class functions. Indeed, the typing of $(\mathsf{ref}_{\ell,\theta}\,N)$ will generally be different from, that is, more permissive than 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 be denoted $(\mathsf{let\ rec}\ f = \lambda x M\ \mathsf{in}\ f)$ in an ML-like notation. We denote by loop the expression $\varrho x x$, and we may use the following standard abbreviation:

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

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 reduction 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 $v_{\ell,\theta} \in \mathsf{dom}(\mu) \ \Rightarrow \ v \neq u$. In what follows we shall only consider *well-formed* configurations, that is pairs (P, μ) such that $\mathsf{loc}(P) \subseteq \mathsf{dom}(\mu)$ and for any $u_{\ell,\theta} \in \mathsf{dom}(\mu)$ we have $\mathsf{loc}(\mu(u_{\ell,\theta})) \subseteq \mathsf{dom}(\mu)$ (this property will be preserved by the operational semantics). The operational semantics consists of a small-step transition relation $(P, \mu) \to (P', \mu')$ between (well-formed) configurations. This is defined by means of an auxiliary transition relation $(M, \mu) \xrightarrow{N}\!\!\!\!\rightarrow (M', \mu')$, as follows:

$$\frac{(M,\mu) \xrightarrow{()}\!\!\!\!\rightarrow (M',\mu')}{(M,\mu) \to (M',\mu')} \qquad \frac{(M,\mu) \xrightarrow{N}\!\!\!\!\rightarrow (M',\mu') \quad N \neq ()}{(M,\mu) \to ((M' \parallel N),\mu')}$$

$$\frac{(P,\mu) \to (P',\mu')}{((P \parallel Q),\mu) \to ((P' \parallel Q),\mu')} \qquad \frac{(P,\mu) \to (P',\mu')}{((Q \parallel P),\mu) \to ((Q \parallel P'),\mu')}$$

As usual we denote by $\xrightarrow{*}$ the reflexive and transitive closure of $\to$. The meaning of a transition $(M, \mu) \xrightarrow{N}\!\!\!\!\rightarrow (M', \mu')$ is that the expression M, in the context of the memory μ, makes a computing step, possibly spawning a thread with body N (which, by convention, is $()$ if no thread is actually spawned at this step), and reconfigures itself as M', while updating the memory into μ'. In order to define this auxiliary transition system, we introduce evaluation contexts:

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

and we denote by $\lceil \mathbf{F} \rceil$ the flow policy enforced by the context $\mathbf{F}$, defined as follows:

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

The relation $\xrightarrow{N}\!\!\!\!\rightarrow$ is defined by means of yet another transition relation, which we denote with the same symbol, as follows:

$$\frac{(U,\mu) \xrightarrow{()}\!\!\!\!\rightarrow (M,\mu')}{(\mathbf{F}[U],\mu) \xrightarrow{()}\!\!\!\!\rightarrow (\mathbf{F}[M],\mu')} \qquad \frac{(U,\mu) \xrightarrow{N}\!\!\!\!\rightarrow (M,\mu') \quad N \neq ()}{(\mathbf{F}[U],\mu) \xrightarrow{(\mathsf{flow}\ \lceil \mathbf{F} \rceil\ \mathsf{in}\ N)}\!\!\!\!\rightarrow (\mathbf{F}[M],\mu')}$$

where U is a *redex*, that is a reducible expression, written according to the following grammar:

$$\begin{array}{lcl}
U & ::= & (\mathsf{if}\ \mathit{tt}\ \mathsf{then}\ M\ \mathsf{else}\ N) \ \mid \ (\mathsf{if}\ \mathit{ff}\ \mathsf{then}\ M\ \mathsf{else}\ N) \ \mid \ (\lambda x M V) \\
 & \mid & V\,;N \ \mid \ (\mathsf{ref}_{\ell,\theta}\,V) \ \mid \ (!\,u_{\ell,\theta}) \ \mid \ (u_{\ell,\theta} := V) \\
 & \mid & (\mathsf{thread}\ M) \ \mid \ (\mathsf{flow}\ F\ \mathsf{in}\ V) \ \mid \ \varrho x W
\end{array}$$

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

Figure 2: Reduction

Finally the rules for reducing redexes are given in Figure 2. One should notice that the local flow declarations do not introduce any constraint on the run-time behaviour of a program: the behaviour of $(\mathsf{flow}\ F\ \mathsf{in}\ M)$ is the same as the one of M. Notice also that, when M terminates, that is, when M reduces to a value V, the flow declaration vanishes, by means of the rule $((\mathsf{flow}\ F\ \mathsf{in}\ V), \mu) \xrightarrow{()}\!\!\!\!\rightarrow (V, \mu)$. This expresses the local character of such declarations. We shall use, somewhat abusively, the same notation $\xrightarrow{P}\!\!\!\!\rightarrow$ for a transition relation which is a kind of reflexive and transitive closure of $\xrightarrow{N}\!\!\!\!\rightarrow$, given by

$$
\frac{}{(M, \mu) \xrightarrow{()}\!\!\!\!\rightarrow (M, \mu)} \qquad \frac{(M, \mu) \xrightarrow{P}\!\!\!\!\rightarrow (M'', \mu'') \xrightarrow{()}\!\!\!\!\rightarrow (M', \mu')}{(M, \mu) \xrightarrow{P}\!\!\!\!\rightarrow (M', \mu')}
$$

$$
\frac{(M, \mu) \xrightarrow{P}\!\!\!\!\rightarrow (M'', \mu'') \xrightarrow{N}\!\!\!\!\rightarrow (M', \mu') \quad N \neq ()}{(M, \mu) \xrightarrow{(P \| N)}\!\!\!\!\rightarrow (M', \mu')}
$$

Then we introduce the *strong convergence* predicate $M\!\downarrow$, meaning that in the context of any memory μ, the reduction of M, as the main thread, terminates on a value, while possibly spawning some new threads:

$$
M\!\downarrow \quad \Leftrightarrow_{\mathrm{def}} \quad \forall \mu\, \exists P\, \exists V \in \mathcal{V}al\, \exists \mu'.\ (M, \mu) \xrightarrow{P}\!\!\!\!\rightarrow (V, \mu')
$$

It is easy to see that the expressions given by the following grammar have no infinite computations, in the sense of the transition relations $\xrightarrow{N}\!\!\!\!\rightarrow$:

$$
\begin{array}{rcl}
T & ::= & V \mid (\mathsf{if}\ T\ \mathsf{then}\ T_0\ \mathsf{else}\ T_1) \mid T\,;T' \mid (\mathsf{ref}_{\ell,\theta}\, T) \mid (!\, T) \mid (T := T') \\
& \mid & (\mathsf{thread}\ M) \mid (\mathsf{flow}\ F\ \mathsf{in}\ T)
\end{array}
$$

Then, with the help of a type safety theorem, one could see that the closed expressions of this class are strongly converging when they are typable – that is, T should be of boolean type in $(\mathsf{if}\ T\ \mathsf{then}\ T_0\ \mathsf{else}\ T_1)$, and T should be of reference type in $(!\, T)$ and $(T := T')$.

3. The non-disclosure policy

3.1 Definition

In this section we introduce our security property, which is relative to a given global flow policy G (determining a lattice of security levels, as usual when dealing with secure information flow). Roughly speaking, it says that a process is secure if, at each step, it satisfies a non-interference property with respect to the flow policy that holds for this particular step. This policy is obtained by extending the global flow policy G with the flow relations introduced by the flow declarations in the scope of which the computational step is performed. To state this formally, we first introduce a new transition relation $(P,\mu) \underset{F}{\longrightarrow} (P',\mu')$, where F is the local flow policy that holds for this step. This is defined exactly as $(P,\mu) \rightarrow (P',\mu')$, using an auxiliary transition relation $(M,\mu) \overset{N}{\longrightarrow\!\!\!\!\!\rightarrow}_{F} (M',\mu')$ given by

$$\frac{(U,\mu) \overset{\emptyset}{\longrightarrow\!\!\!\!\!\rightarrow} (M,\mu')}{(\mathbf{F}[U],\mu) \overset{\emptyset}{\underset{\lceil\mathbf{F}\rceil}{\longrightarrow\!\!\!\!\!\rightarrow}} (\mathbf{F}[M],\mu')} \qquad \frac{(U,\mu) \overset{N}{\longrightarrow\!\!\!\!\!\rightarrow} (M,\mu') \quad N \neq \emptyset}{(\mathbf{F}[U],\mu) \overset{(\mathsf{flow}\ \lceil\mathbf{F}\rceil\ \mathsf{in}\ N)}{\underset{\lceil\mathbf{F}\rceil}{\longrightarrow\!\!\!\!\!\rightarrow}} (\mathbf{F}[M],\mu')}$$

where U is any redex. It should be clear that we have only decorated the operational semantics by introducing these transitions, that is $(M,\mu) \overset{N}{\longrightarrow\!\!\!\!\!\rightarrow} (M',\mu')$ if and only if $(M,\mu) \overset{N}{\underset{F}{\longrightarrow\!\!\!\!\!\rightarrow}} (M',\mu')$ for some (unique) F. The meaning of F in a transition $(P,\mu) \underset{F}{\longrightarrow} (P',\mu')$ is that what is read by P at confidentiality level ℓ at this step can be regarded as having level ℓ' if information is allowed to flow from ℓ to ℓ' by the global policy G extended with the local flow policy F, that is if $\ell \preceq_{G\cup F} \ell'$. Then P is secure if, for any confidentiality level ℓ, it does not reveal, by writing in the memory at a level lower than ℓ, information that it reads at levels not lower than ℓ. As we just said, from the reading point of view, confidentiality levels are compared with respect to $G \cup F$, whereas from the writing point of view, the levels are to be compared with respect to the global policy G only, because the updates in the memory are read by other expressions that are not, a priori, in the scope of the declarations introducing F.

As usual, the property that a process does not transfer information from a "high" part (not below ℓ) of the memory to a "low" part (below ℓ) is formalized by requiring that the process preserves in a sense the "low equality" of memories. Two memories are equal up to level ℓ if they assign the same value to every location with security level lower than ℓ. Here we have to explicitly indicate which is the flow policy that is used to compare confidentiality levels. Furthermore, we will compare two memories only with respect to the references they share. The "low equality" of memories is thus defined:

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

This relation is not transitive, but it is reflexive and symmetric [2]. We shall use without notice the fact that

$$F \subseteq F' \ \&\ \mu \mathrel{\dot{=}}^{F',\ell} \nu \ \Rightarrow\ \mu \mathrel{\dot{=}}^{F,\ell} \nu$$

Our security property is defined in terms of *bisimulations* (see [8, 17, 43, 47] for the use of bisimulations in stating security properties, and [28] 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 [43], here we shall make use

[2] In the following we however still use the terminology "low equal", in quotes.

of this notion in a slightly non-standard way, since we shall call "bisimulation" a relation between processes P, rather than configurations (P, μ). The notion of a bisimulation we use here is relative not only to a confidentiality level ℓ, determining what is regarded as being "low", but also to the global flow policy G. The definition is rather abstract, as it could be used in any framework where decorated transitions $(P, \mu) \underset{F}{\longrightarrow} (P', \mu')$ are available, and where confidentiality levels are assigned to memory locations.

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

$$\left.\begin{array}{l} P \,\mathcal{R}\, Q \quad \& \\ (P, \mu) \underset{F}{\longrightarrow} (P', \mu') \quad \& \\ \mu \doteq^{F \cup G, \ell} \nu \quad \& \\ u_{\ell', \theta} \in \mathsf{dom}(\mu' - \mu) \;\Rightarrow\; u \text{ is fresh for } \nu \end{array}\right\} \quad \Rightarrow \quad \exists Q', \nu'. \left\{\begin{array}{l} (Q, \nu) \xrightarrow{*} (Q', \nu') \quad \& \\ P' \,\mathcal{R}\, Q' \quad \& \\ \mu' \doteq^{G, \ell} \nu' \end{array}\right.$$

(with implicit universal quantifications over all the free meta-variables in this formula).

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

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

has to be matched by a sequence of transitions from (Q, ν), whenever $P \,\mathcal{R}\, Q$, and the memories μ and ν satisfy some conditions. We have already explained the meaning of the first one, namely $\mu \doteq^{F \cup G, \ell} \nu$. 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'), such that there has been no information leakage, that is $\mu' \doteq^{G, \ell} \nu'$, and no mismatch will occur in future computations, that is $P' \,\mathcal{R}\, Q'$. As we explained above, 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"). The reader may have noticed that there is no condition on the flow policies involved in the matching moves for Q – they are not even mentioned. This means in particular that all the *pure* programs, that never touch the memory, are bisimilar. More generally, there is a bisimulation that relates all the expressions that never modify the memory.

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 detect an illegal flow in the program

$$(\mathsf{if}\ !\, w_X\ \mathsf{then}\ (\mathsf{if}\ !\, w_X\ \mathsf{then}\ ()\ \mathsf{else}\ v_L := !\, u_H)\ \mathsf{else}\ ()) \tag{4}$$

because the initial memory, assigning *tt* to w_X for instance, may have been changed (by another thread) into another one where the content of w_X is *ff*, before evaluating the sub-expression $(\mathsf{if}\ !\, w_X\ \mathsf{then}\ ()\ \mathsf{else}\ v_L := !\, u_H)$. This demanding definition for bisimulations seems also appropriate for dealing with a mobile code scenario, where the shared memory of a system of threads could 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 $\leftrightarrows^{G,\ell}$. This is the union of all such bisimulations.*

One should observe that the relation $\leftrightarrows^{G,\ell}$ 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 [43], 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 $P \leftrightarrows^{G,\ell} P$ for all ℓ. We then write $P \in \mathcal{NDC}(G)$.*

It is easily seen that the set $\mathcal{NDC}(G)$ is non-empty. For instance, any value is secure, as well as any pure expression. As a matter of fact, any "mute" expression, that does not update the memory (like in particular an expression written without using the assignment construct) is secure, and this is intuitively quite natural, since such an expression cannot disclose any information. We should point out here that our notion of a secure program elaborates upon the standard approach regarding imperative and concurrent languages, based on an input/outpout semantics in the case of sequential programs [12, 55], and on a small-steps semantics regarding concurrent ones [43]. We could say, following the terminology of [13], that we have followed a *state-oriented* approach to secure information flow. Then we get a rather extensional notion of secure programs, which in particular does not mention any type system, nor any particular syntax. This is in contrast with most studies concerning security for functional languages (e.g. [13, 20, 37, 57, 60]). We shall come back to this point when introducing the type system.

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 *G-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:

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

To show that for sequential programs, non-disclosure with respect to a given flow policy G implies non-interference (w.r.t. the same policy), let us denote by $\mathcal{DE}xpr$ the set of expressions written without using thread, flow and ref. The big-step semantics for expressions in $\mathcal{DE}xpr$ can be defined as follows:

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

It is easy to see that the evaluation mechanism is deterministic for $M \in \mathcal{DE}xpr$, and that if $(M,\mu) \Rightarrow \mu'$ then $\mathsf{dom}(\mu') = \mathsf{dom}(\mu)$. Now assume that $M \in \mathcal{DE}xpr \cap \mathcal{NDC}(G)$, $(M,\mu) \xrightarrow{*} (V,\mu')$ and $(M,\nu) \xrightarrow{*} (V',\nu')$ with $\mu \doteq^{G,\ell} \nu$. Then there exist M' and ν'' such that $(M,\nu) \xrightarrow{*} (M',\nu'')$, $V \leftrightarrows^{G,\ell} M'$ and $\mu' \doteq^{G,\ell} \nu''$. 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 $\mu' \doteq^{G,\ell} \nu'$.

3.2 Examples of secure and unsecure programs

Now let us see some examples. We assume given two principals H and L, and a global 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 $\mathcal{DE}xpr$, it is obvious that the standard examples of explicit (or direct) and implicit (or indirect) flow, namely:

$$v_L := !\, u_H \tag{5}$$

$$(\mathsf{if}\ !\, u_H\ \mathsf{then}\ v_L := tt\ \mathsf{else}\ v_L := ff) \tag{6}$$

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

$$(\mathsf{if}\ !\, u_H\ \mathsf{then}\ ()\ \mathsf{else}\ \mathsf{loop})\, ;\, v_L := tt \tag{7}$$

where writing at level L depends on reading at level H (we refer to [4, 8, 20, 41, 47, 52] for discussions about this kind of leak). Following [8, 47], we shall put a constraint on sequential composition in the type system to rule out such a program. 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:

$$(w_H := !\, u_H)\, ;\, (v_L := tt) \tag{8}$$

Besides these classical examples of illegal information flow, there are, in our higher-order language with mutable state, some other kinds of leaks, which arise in particular from the fact that functional values may wrap side-effects inside a λ-abstraction. Then for instance we have functional variants of the various kinds of leaks, like in

$$(\lambda x(v_L := x)(!\, u_H)) \tag{9}$$

Slightly more subtle leaks arise from the fact that functional values may be stored in the memory. For instance, a secret reference u_H may contain a functional value that, when applied to an argument, writes at a low confidentiality level in the memory, like $\lambda x(v_L := V)$, with different values for V in different memories. Then the expression

$$((!\, u_H)())) \tag{10}$$

implements an indirect flow of information from level H to level L, since the value assigned to v_L while reducing this expression depends on the value read from the memory for u_H (see [58] for a similar example). A more explicit way of writing a similar computing effect is

$$(\mathsf{if}\ !\, w_H\ \mathsf{then}\ u_H := \lambda x(v_L := tt)\ \mathsf{else}\ u_H := \lambda x(v_L := ff))\, ;\, ((!\, u_H)())$$

Clearly, reading a functional value from the memory may cause the reduction of an expression like $((!\, u_H)())$ to diverge or not, depending on the value read for u_H. Indeed, this value could be λxx or $\lambda x\, \mathsf{loop}$ for instance. Then another example of a termination leak is

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

Similarly, there is a termination leak in

$$(\lambda x(x())(!\,u_H))\,;v_L := tt \tag{12}$$

One should also observe that the reduction order matters. For instance, the following expression implements a termination leak, but it would be secure if we had adopted a right to left order:

$$(((!\,u_H)())(v_L := tt))$$

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

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

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

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ (\mathsf{flow}\ H \prec L\ \mathsf{in}\ v_L := tt)\ \mathsf{else}\ ()) \tag{14}$$

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.

3.3 Preliminary results

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

PROOF: by case on the transition $(M,\mu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (M',\mu')$. 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 $M = \mathbf{F}[(\mathsf{ref}_{\ell',\theta}\, V)]$, then $M' = \mathbf{F}[u_{\ell',\theta}]$, $F = \lceil \mathbf{F} \rceil$ and $\mu' = \mu \cup \{u_{\ell',\theta} \mapsto V\}$. Since u is fresh for ν, we have $(M,\nu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (M', \nu \cup \{u_{\ell',\theta} \mapsto V\})$. If $M = \mathbf{F}[(!u_{\ell',\theta})]$ then $M' = \mathbf{F}[\mu(u_{\ell',\theta})]$, $F = \lceil \mathbf{F} \rceil$ and $\mu' = \mu$. Since the configurations we consider are assumed to be well-formed, we have $(M,\nu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (\mathbf{F}[\nu(u_{\ell',\theta})], \nu)$. If $M = \mathbf{F}[(u_{\ell',\theta} := V)]$ then $M' = \mathbf{F}[()]$, $F = \lceil \mathbf{F} \rceil$ and $\mu' = \mu[u_{\ell',\theta} := V]$. In this case $(M,\nu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (M', \nu[u_{\ell',\theta} := V])$. ❑

Now we define "operationally high" processes, those that leave the memory constant, up to "low equality":

DEFINITION (OPERATIONALLY HIGH PROCESSES) 3.5. *A set* $\mathcal{H}$ *of processes is said to be a set of* operationally (G,ℓ)-high *processes if the following holds for any* $P \in \mathcal{H}$*:*

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

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 (4) for instance would be high, and Lemma 3.6 below would therefore not hold. One should remark that there are sets satisfying Definition 3.5. For instance, any value is a high process, as well as any reference creation ($\mathsf{ref}_{\ell,\theta}V$). Moreover, the union of a family of such $\mathcal{H}$'s is a set of (G,ℓ)-high processes, and so there exists a largest such set, which we denote by $\mathcal{H}_{G,\ell}$, and call the set of *operationally* (G,ℓ)*-high* processes. Notice that

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

LEMMA and NOTATION 3.6. *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 $\asymp^{G,\ell}$ the (G,ℓ)-bisimulation $\mathcal{H}_{G,\ell}\times\mathcal{H}_{G,\ell}$.*

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

The following is easy to see:

LEMMA 3.7. $M,N_0,N_1\in\mathcal{H}_{G,\ell} \;\Rightarrow\; (\mathsf{if}\ M\ \mathsf{then}\ N_0\ \mathsf{else}\ N_1)\in\mathcal{H}_{G,\ell}$

4. The type and effect system

4.1 The system

In the store-oriented approach that we have adopted, information flow is performed by means of side-effects. It is therefore not surprising that our static analysis technique takes the form of an *effect* system [29] where, as noted in [13], confidentiality levels play the role of *regions*, and where the security effects are, roughly speaking, confidentiality levels at which the program reads or writes. The types are then 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, which is the confidentiality level at which the reference is classified. Since a functional value wraps a possibly effectful computation, its type records this *latent effect* [29], which is the effect the function may have when applied to an argument. It also records the "latent flow policy", which is assumed to hold when the function is called. The types do not otherwise mention the security framework. Then there is no type of "secret booleans" or "public booleans" for instance, as this would be the case with the "value-oriented" approach to security which is often adopted regarding functional languages (in our view the purely functional programs, written without using any i/o facility, are all secure). The syntax of types is

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

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 reducing M;
- ℓ_0, also denoted by $s.\mathsf{c}$, is the *confidentiality level* of M. This is an upper bound (up to the current flow relation) of the confidentiality levels of the references the expression M reads that may influence its resulting *value*;

- 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 the levels of the references the expression M reads that may influence its *termination*. We call this the *termination effect* of the expression.

In the following we shall denote $s.\mathsf{c} \curlyvee_G s.\mathsf{t}$ by $s.\mathsf{r}$. There is actually an implicit parameter in the type system, which is a set $\mathcal{T}$ of expressions that is used in the typing of conditional branching. We could make it apparent, writing for instance the judgements as $G;\Gamma \vdash_{\mathcal{T}} M : s, \tau$. The single property that we will assume about this set in our proof of type soundness is that it only contains strongly converging expressions, that is:

$$M \in \mathcal{T} \;\Rightarrow\; M\dagger \qquad (*)$$

We recall that the predicate $M\dagger$, meaning that M converges to a value, independently of the memory, was defined at the end of Section 2, where we have given an instance of a class $\mathcal{T}$ of expressions satisfying $(*)$.

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 state 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 [49]). The original type system of [55] only recorded the writing effect for commands, so that a type w cmd (which is more an effect than a type) in the system of [55] is similar to the pair $(\bot, w, \bot), \mathsf{unit}$ in our setting. The termination effect is similar to the "guard level" of [8] and to the "running time level" of [47]. With respect to the system presented in [3], we shall adopt the improvements introduced in [7], that concern the way we type conditional branching, and the way we built the confidentiality level and the termination effect.

According to the intuition above, the security effects $s = (c, w, t)$ are ordered componentwise, in a covariant manner as regards the confidentiality level c and the termination effect t, and in a contravariant way as regard the writing effect w. 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:

$$s \curlyvee_G s' \;=_{\text{def}}\; (s.\mathsf{c} \curlyvee_G s'.\mathsf{c}, s.\mathsf{w} \cup s'.\mathsf{w}, s.\mathsf{t} \curlyvee_G s'.\mathsf{t})$$

as well as the following convention:

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 $\curlyvee_G$. We recall that by $s.\mathsf{r}$ we mean $s.\mathsf{c} \curlyvee_G s.\mathsf{t}$.*

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. In particular, there is no subtyping rule.

4.2 Comments and examples

Now let us comment on some of the typing rules, justifying the constraints on the flow of information (that is, the inequations involving $\preceq$ in the premises), as well as the resulting effect of the expression. We see that, in accordance with the intuitive explanations above, the confidentiality level and writing effect of an expression are respectively introduced by the constructions for dereferencing and updating the memory – rules (Deref) and (Assign). The termination level is

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

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

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

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

where

$$t \;=\; \begin{cases} \bot & \text{if } N_0, N_1 \in \mathcal{T} \\ s.\mathsf{c} & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

Figure 3: The Type and Effect System

introduced in typing conditional branching, rule (Cond), and application, rule (App). We shall comment on this below. The constraints on information flow are implemented in the rules (Cond), (App), (Seq), (Ref) and (Assign).

(Cond) In this rule, 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 confidentiality level and termination effect of the predicate (recall that $s.\mathsf{r}$ stands for $s.\mathsf{c} \curlyvee_G s.\mathsf{t}$). Regarding the confidentiality level, this is to prevent indirect flows, like in example (6). We shall see below why we also need the constraint $s.\mathsf{t} \preceq_G s_0.\mathsf{w} \cup s_1.\mathsf{w}$ in the (Cond) rule. In the conclusion of (Cond), we record the confidentiality level of the predicate in the termination level of the whole expression, but only if one of the two branches is not known to terminate (that is, to belong to the

class $\mathcal{T}$ of expressions). Clearly, for instance, the program

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

is secure, and should not be rejected by the type system (although it is rejected by the systems of [3, 8], and also by the system of [47] if we modify the second branch into $();\, w_H := ff$ for instance). But we have to reject the program of example (7), and this is indeed the case, thanks to condition $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ in the (Seq) rule, since, by the $(*)$ requirement, we have $\mathsf{loop} \notin \mathcal{T}$. This example is essentially the same as

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

which is ruled out by the condition $s.\mathsf{t} \preceq_G s''.\mathsf{w}$ in the (App) rule.

(App) In this rule, the condition $s''.\mathsf{r} \preceq_G s'.\mathsf{w}$ is, regarding the confidentiality part $s''.\mathsf{c}$, to prevent a direct flow, like in Example (9). The condition $s.\mathsf{r} \preceq_G s'.\mathsf{w}$, or more precisely $s.\mathsf{c} \preceq_G s'.\mathsf{w}$, 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 the expression of Example (10). Examples (11) and (12) respectively show why the confidentiality 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 $(\mathsf{let}\ x = N\ \mathsf{in}\ M) = (\lambda x M N)$, namely:

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

(Seq) We have already seen a justification for the constraint $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ in this rule, with the example (7). Notice that, in contrast to the rule for sequential composition in [3], we do not record the confidentiality level of the expression M in the effect of the compound expression $M\,;N$. The reason is that the value of this expression, if any, is the one of N, and therefore does not depend on the particular value of M (indeed, this construct is generally used with M of type unit). We can now explain the constraints $s.\mathsf{t} \preceq_G s_0.\mathsf{w} \cup s_1.\mathsf{w}$ in the (Cond) rule, and $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ and $s''.\mathsf{t} \preceq_G s'.\mathsf{w}$ in the (App) rule. It is easy to check that, if we assume that the reference u_H contains values of type $(\mathsf{unit} \xrightarrow[\emptyset]{\bot} \tau)$, the expression $((!\,u_H)())$, used in example (11), has a security effect $(H, \bot, H)$, and therefore for any (typable) value V the expression $((!\,u_H)())\,;V$ has effect $(\bot, \top, H)$. Then, thanks to the constraint $s.\mathsf{t} \preceq_G s_0.\mathsf{w} \cup s_1.\mathsf{w}$ in the (Cond) rule, the type system rejects for instance

$$(\mathsf{if}\ ((!\,u_H)())\,;\, tt\ \mathsf{then}\ v_L := tt\ \mathsf{else}\ ()) \tag{15}$$

which is indeed unsecure, if the current flow policy does not say $H \prec L$, for the same reason as in example (11). Similarly, one can see that the constraints $s.\mathsf{t} \preceq_G s'.\mathsf{w}$ and $s''.\mathsf{t} \preceq_G s'.\mathsf{w}$ in the (App) rule allow us to reject respectively the unsecure programs

$$(((!\,u_H)())\,;\lambda z.v_L := V)()$$

and

$$(\lambda z.v_L := V)(((!\,u_H)())\,;())$$

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

confidentiality level 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 (8) is accepted. This explains our syntax for the imperative part of the language.

(Ref) As for sequential composition, in typing a reference creation $(\mathsf{ref}_{\ell,\theta}\,M)$ we do not have to record the confidentiality level of M, since the particular value of this expression will not influence the result of the statement $(\mathsf{ref}_{\ell,\theta}\,M)$, which is a new reference. The condition $s.\mathsf{r} \preceq_G \ell$ in the typing rule (Ref)[3] is meant to rule out for instance the expression

$$(\mathsf{let}\ x = (\mathsf{ref}_L\,(!\,u_H))\ \mathsf{in}\ v_L := !\,x)$$

We let the reader find an example illustrating the need for $s.\mathsf{t} \preceq_G \ell$ – hint: try with $(\mathsf{let}\ x = (\mathsf{ref}_L\,M)\ \mathsf{in}\ v_L := !\,x)$ where M has effect $(\bot, \top, H)$.

(Assign) The condition $s'.\mathsf{c} \preceq_G \ell$ in this rule is to prevent a direct flow, like in example (5). Again justifying $s.\mathsf{t} \curlyvee s'.\mathsf{t} \preceq_G \ell$ is left to the reader. With the condition $s.\mathsf{c} \preceq_G \ell$ 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

$$(((!\,u_H)())\,;w_X) := (v_L := tt)$$

If the assignment $M := N$ terminates, its value is $()$, and this explains why we do not record the confidentiality level of the sub-expressions M and N in the security effect of the assignment.

These examples – plus some others that we left to the reader to find – 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.

We now comment the rules (Thread), (Rec) and (Flow). One can see that an expression $(\mathsf{thread}\,M)$ has no termination effect, since its reduction terminates in one step. Similarly, the value of this expression, namely $()$, does not depend on the one of M, and therefore $(\mathsf{thread}\,M)$ has a trivial confidentiality level. We only record the writing effect here, because we have to reject a program like

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

which implements an indirect flow from u_H to v_L. Regarding recursion, the reader can check for instance that a derived typing for the while construct is

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

As in [8, 47], 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 (a different design choice is made in

[3] This was missing in [3].

[2]). For instance, one can check that the following is a valid proof of typing (leaving out the type annotation of references):

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

and therefore one can see that the type system accepts the expression of example (13). Using Remark 2.2, we could choose c and t in the (FLOW) rule to be G-minimal respectively such that $s.\mathsf{c} \preceq_{G \cup F} c$ and $s.\mathsf{t} \preceq_{G \cup F} t$. Subsumption in the (FLOW) rule will be used in the proof of the Subject Reduction property [4]. However, one should notice that in the typing rule for flow declaration we do not allow subsumption for the writing level. Example (14) shows why it would be wrong to do so.

As suggested in the Introduction, we can express the declassification operation of [32, 42] as follows (omitting the explicit type anotation for reference creation):

$$\mathsf{declassify}(M, \ell) =_{\mathrm{def}} (\mathsf{let}\ x = (\mathsf{ref}_H M)\ \mathsf{in}\ (\mathsf{flow}\ H \prec \ell\ \mathsf{in}\ !x))$$

where H is a principal that is higher (w.r.t. the current flow policy) than any other one, and $H \prec \ell$ is an abbreviation for $H \prec p_1, \ldots, H \prec p_n$ if $\ell = \{p_1, \ldots, p_n\}$ [5]. The derived typing for this expression is

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

One should notice that there is no specific constraint on M here. In [3], we proposed, for lack of a fine enough typing of $(\mathsf{ref}_H\, M)$, the expression $(\mathsf{flow}\ H \prec \ell\ \mathsf{in}\ M)$ as a possible encoding of the declassification operation. The encoding above seems better, because it does not include the side-effects in reducing the declassified expression M inside the scope of the declassifying flow policy $H \prec \ell$. Then, for instance $\mathsf{declassify}((v_L := !u_H)\,;!w_H, L)$ is rejected, whereas $(\mathsf{flow}\ H \prec L\ \mathsf{in}\ (v_L := !u_H)\,;!w_H)$ is accepted by the type system. Another example of the use of the flow declaration construct is given in the Introduction. Namely, we showed how to encode the coercion construct $[\ell \preceq \ell']M$ of [21], as

$$(\mathsf{let}\ x = (\mathsf{ref}_\ell M)\ \mathsf{in}\ (\mathsf{flow}\ \ell \prec \ell'\ \mathsf{in}\ !\, x))$$

where $\ell \prec \ell'$ means $\{\, p \prec q \mid p \in \ell\ \&\ q \in \ell' \,\}$. One can check that a derived typing for this construct is

$$\dfrac{G; \Gamma \vdash M : s, \tau \quad s.\mathsf{r} \preceq_G \ell}{G; \Gamma \vdash [\ell \preceq \ell']M : (\ell', s.\mathsf{w}, s.\mathsf{t} \curlyvee \ell'), \tau}$$

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:

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

[4] The proof we have for our Soundness Theorem uses all the features of the type system.

[5] For this example and the next one, we assume that the confidentiality levels attached to references are non-empty.

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 in examples (6) and (15), the constraint $s.\mathsf{r} \preceq_G \{q, r\}$ is to prevent implicit flows and termination leaks. 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.

4.3 Comparison with other security type systems

The type system we have presented, which was introduced in [7], improves over the one of [3] in two respects, namely the typing of termination leaks, and the typing of side effects. A typical example of termination leak, where a low assignment depends on a high read, is given by example (6), which could also be written

$$(\mathsf{while}\ !u_H\ \mathsf{do}\ ())\ ;\ v_L := \mathit{ff} \tag{16}$$

Many research works avoid the issue of typing termination leaks by only considering *weak*, or *termination insensitive* non-interference. Although this may be acceptable when dealing with sequential programs, one has to seriously take into account termination leaks when dealing with concurrent threads, where a program such as (16) is dangerous (see [8, 20, 47, 48, 52]). Moreover, our way of dealing with declassification relies on a termination sensitive security property. Termination leaks may be ruled out using drastic restrictions, like allowing only predicates of the lowest confidentiality level in the while loops [48, 52]. In our language, where while loops are recursively defined by means of conditional branching, such a restriction would be obtained by restricting the predicate in the branching construct to be of the lowest confidentiality level only – an unacceptable constraint. Here we improve on a solution to this issue introduced in [8] (and independently in [47]), which we followed in [3], that consists in using the "termination effect" to control information flow in sequential composition. In this way, a program such as (16) is rejected, but loops or conditional branchings on high predicates are allowed. However, this typing discipline is still quite inflexible, since it rejects any program of the form

$$(\mathsf{if}\ !u_H\ \mathsf{then}\ P\ \mathsf{else}\ Q)\ ;\ v_L := \mathit{ff} \tag{17}$$

for instance. Here we show that a program such as (17) is secure in the case where both branches P and Q are known to terminate – provided, obviously, that $(\mathsf{if}\ !u_H\ \mathsf{then}\ P\ \mathsf{else}\ Q)$ is secure. Then we get a refined typing for termination leaks which, as far as we can see, has not been previously suggested (a preliminary step was however taken in [47], where it is shown that a program such as (17) is secure if both P and Q terminate in the same number of computing steps). One may observe that, as opposed to the one of [3] for instance, our type system does not rule out expressions that are regarded as involving a *timing leak* (see [22, 41, 48]), like

$$(\mathsf{if}\ !\,u_H\ \mathsf{then}\ M\ \mathsf{else}\ ())\ ;\ v_L := \mathit{ff} \tag{18}$$

where M is an expression that takes some time to compute, like $() ; \cdots ; ()$. Indeed, this program is secure, according to Definition 3.3. In our view, 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 leak should rather be tracked down when dealing with the code produced by an optimizing compiler for instance.

Another contribution we make in this paper is in the way we extend the standard typing of information flow in a simple imperative language to a more expressive one. In the simple "while" language of [55], which is very often considered as a kernel language in the information flow literature, an expression has for "type" (this is actually an effect) an upper bound of the confidentiality level of memory locations that are read to compute the value of the expression, and a program has for its "type" a lower bound of the confidentiality level of memory locations it may update. In the system of [8, 47], one adds, as we have seen, a "termination level". In a language like Core ML, an expression is a program, and therefore it may have side-effects, and its reduction may diverge. Then the "security effect" of an expression in such a language is a triple (r, w, t) where r is the reading effect, w is the writing effect and t is the termination effect of the expression. This is the basis on which we built our type system in [3]. However, in this paper the reading effect is an upper bound of the confidentiality level of all the memory locations that are read while reducing the expression, and this is rather coarse. As we show here, we may exploit the fact that some read operations do not contribute to the specific value an expression may have to obtain a refined typing (a similar observation is made in [13] to motivate the "informativeness" predicate). One should then observe that, since both the confidentiality level and the termination effect of an expression are always smaller in our system than in the one of [3], the flow constraints where these are involved – in the rules (Cond), (App), (Seq), (Ref) and (Assign) – are less constraining here. Indeed, for each of these constraints one can find an expression that is accepted in our system but rejected by the system of [3].

It is not easy to compare the type systems that have been proposed in the literature to deal with information flow in functional languages with the one we have introduced here, for various reasons. A first remark that we already made is that most type systems for functional languages are not – with the exception of [13], which deals with a "monadic" language – "store-oriented", but "value-oriented". That is, they most often assign a security level to *values*, whereas we have taken the point of view that the "pure" fragment of the language is secure (with the advantage of not having to decorate the types with security levels). Indeed, we think that some communication medium (the references in our case) is needed for implementing the idea that information flows from one place to another. Assigning security levels to values amounts to regarding the formal parameters of a function (that is, the λ-variables) as such communication media, and to regarding the meta-operation of substitution as implementing a flow of information. With the state-oriented approach, we are assuming in a sense that the λ-variables of a (closed) expression cannot be inspected by an attacker. The two approaches have been related in [13], where it is shown that a typically "value-oriented" approach to typing information flow, as it can be found in [58], can be encoded in a "store-oriented" approach, simply by making any value available as the contents of a reference (a similar result regarding the SLam calculus of [20] is presented in the workshop version of [13]). Another difference with type systems for functional languages is that these, including the one of [13], are most often only dealing with weak, termination insensitive non-interference. Then such type systems cannot ensure our "non-disclosure policy", and therefore they are not well-suited to support declassification, as we have formulated it. Our type system is actually closer to the ones that are used when dealing with termination sensitive non-interference, that is, with type systems for concurrent and imperative languages, although we have to cope with many more implicit flows. 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.

4.4 Preliminary results

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

To prove Subject Reduction we shall follow the usual steps [56], thus omitting the details of the proofs. We need, in particular, some standard weakening and strengthening properties:

LEMMA 4.2.

(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 for any* F *there exists* s' *such that* $F,G;\Gamma \vdash M : s',\tau$ *with* $s'.\mathsf{c} \preceq_{F\cup G} s.\mathsf{c}$, $s'.\mathsf{w} = s.\mathsf{w}$ *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) 4.3. $G;\Gamma \vdash W : \bot,\tau\ \&\ G;\Gamma, x:\tau \vdash M : s,\sigma\ \Rightarrow\ G;\Gamma \vdash \{x \mapsto W\}M : s,\sigma$

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 $s'.\mathsf{c} \preceq_{G\cup F} s.\mathsf{c}$, $s'.\mathsf{w} = s.\mathsf{w}$ 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) 4.4. *If* $G;\Gamma \vdash \mathbf{F}[M] : s,\tau$ *is a valid judgement, with a proof where* M *has the typing* $\lceil\mathbf{F}\rceil, G;\Gamma \vdash M : s',\sigma$, *and if* $\lceil\mathbf{F}\rceil, G;\Gamma \vdash N : s'',\sigma$ *with* $s''.\mathsf{c} \preceq_{G\cup\lceil\mathbf{F}\rceil} s'.\mathsf{c}$, $s'.\mathsf{w} \preceq s''.\mathsf{w}$ *and* $s''.\mathsf{t} \preceq_{G\cup\lceil\mathbf{F}\rceil} s'.\mathsf{t}$, *then* $G;\Gamma \vdash \mathbf{F}[N] : s_0,\tau$ *for some* s_0 *such that* $s_0.\mathsf{c} \preceq_G s.\mathsf{c}$, $s.\mathsf{w} \preceq s_0.\mathsf{w}$ *and* $s_0.\mathsf{t} \preceq_G s.\mathsf{t}$.

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

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

with $s_1.\mathsf{c} \preceq_{G \cup F} s.\mathsf{c}$, $s_1.\mathsf{w} = s.\mathsf{w}$ and $s_1.\mathsf{t} \preceq_{G \cup F} s.\mathsf{t}$. By induction hypothesis there exists s_0 such that the judgement $F, G; \Gamma \vdash \mathbf{F}'[N] : s_0, \tau$ is provable, and $s_0.\mathsf{c} \preceq_{G \cup F} s_1.\mathsf{c}$, $s_1.\mathsf{w} \preceq s_0.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_{G \cup F} s_1.\mathsf{t}$. Then by the (FLOW) rule $G; \Gamma \vdash \mathbf{F}[N] : (s.\mathsf{c}, s_0.\mathsf{w}, s.\mathsf{t}), \tau$ 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) 4.5. *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* $s'.\mathsf{c} \preceq_G s.\mathsf{c}$, $s.\mathsf{w} \preceq s'.\mathsf{w} \cup s''.\mathsf{w}$ *and* $s'.\mathsf{t} \preceq_G s.\mathsf{t}$.

PROOF: by cases on the proof of the transition $(M, \mu) \xrightarrow[F]{N} (M', \mu')$. We only examine some cases. If $M = \mathbf{F}[(\lambda x M_0 V)]$, with $F = \lceil \mathbf{F} \rceil$, $N = \langle\rangle$, $\mu' = \mu$ and $M' = \mathbf{F}[\{s \mapsto V\} M_0]$, then we use the lemmas 4.3 and 4.4. The argument is similar if $M = \mathbf{F}[\varrho x W]$ and $M' = \mathbf{F}[\{x \mapsto \varrho x W\} W]$. If $M = \mathbf{F}[(\mathsf{flow}\ F'\ \mathsf{in}\ V)]$ and $M' = \mathbf{F}[V]$, with $F = \lceil \mathbf{F} \rceil \cup F'$, $N = \langle\rangle$ and $\mu' = \mu$, then in the proof of the judgement $G; \Gamma \vdash M : s, \tau$, the expression $(\mathsf{flow}\ F'\ \mathsf{in}\ V)$ has the typing

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

with $s.\mathsf{w} = \bot$. By Remark 4.1 we have $\lceil \mathbf{F} \rceil, G; \Gamma \vdash V : \bot, \tau$, and we conclude using Lemma 4.4 in this case. If $M = \mathbf{F}[(\mathsf{thread}\, M_0)]$ and $M' = \mathbf{F}[\langle\rangle]$, with $F = \lceil \mathbf{F} \rceil$, $N = (\mathsf{flow}\ \lceil \mathbf{F} \rceil\ \mathsf{in}\ M_0)$ and $\mu' = \mu$, and if in the proof of $G; \Gamma \vdash M : s, \tau$, the expression $(\mathsf{thread}\, M_0)$ has the typing

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

then we conclude using the Lemma 4.4 and the fact that

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

is a valid typing. ❑

We conclude this section with some technical definitions and results that will be used to prove our main result. First, we notice that the writing effect of an expression is indeed a lower bound of the level of references that are updated while reducing the expression:

LEMMA 4.6. $G; \Gamma \vdash \mathbf{F}[(u_{\ell,\theta} := V)] : s, \tau \Rightarrow s.\mathsf{w} \preceq \ell$.

PROOF: by induction on $\mathbf{F}$, easy. ❑

This justifies the following definition:

Definition (Syntactically High Expressions) 4.7. *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 $s'.\mathsf{w} \not\preceq_G \ell$.*

Indeed, we have:

Lemma 4.8.

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

(ii) *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:

(i) 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 $(M,\mu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (M',\mu')$ 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 $(M,\mu) \xrightarrow[F]{N}\!\!\!\!\rightarrow (M',\mu')$. The lemma is trivial in all the cases where $\mu \subseteq \mu'$. If $M = \mathbf{F}[(u_{\ell',\theta} := V)]$ and $\mu' = \mu[u_{\ell',\theta} := V]$ then $s.\mathsf{w} \preceq \ell'$ by Lemma 4.6(ii). This implies $\ell' \not\preceq_G \ell$, hence $\mu' \doteq^{G,\ell} \mu$.

(ii) 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 $M,N \in \mathcal{H}_{G,\ell}$ and M is a (G,ℓ)-high function. Assume that such an application (MN) performs the transition $((MN),\mu) \xrightarrow[F]{N'}\!\!\!\!\rightarrow (M',\mu')$. We show, by cases 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 $N' = \langle\rangle$, 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 4.8. If $M = (\mathbf{F}[U]N)$ and $M' = (\mathbf{F}[M'']N)$ with $(U,\mu) \xrightarrow[F]{N''}\!\!\!\!\rightarrow (M'',\mu')$ and $N' = (\mathsf{flow}\ \mathbf{F}\ \mathsf{in}\ N'')$ if $N \neq \langle\rangle$, and $N' = N''$ otherwise, then $\mathbf{F}[M'']$ is a (G,ℓ)-high function, by Subject Reduction, and $\mathbf{F}[M''] \in \mathcal{H}_{G,\ell}$ since $\mathbf{F}[U] \in \mathcal{H}_{G,\ell}$. This shows that $M' \in \mathcal{G}$. The argument is similar if $M = (V\mathbf{F}[U])$ and $M' = (V\mathbf{F}[M''])$ with $(U,\mu) \xrightarrow[F]{N''}\!\!\!\!\rightarrow (M'',\mu')$. ❑

We could prove, following the usual steps [56], 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. First we make a simple but crucial observation regarding the operational semantics of our language, namely that if the reduction 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 5.1. *If $(M,\mu) \xrightarrow[F]{N}\!\!\!\!\rightarrow\, (M',\mu')$ and $(M,\nu) \xrightarrow[F']{N'}\!\!\!\!\rightarrow\, (M'',\nu')$ with $M' \neq M''$ (or $N \neq N'$) and $\mathsf{dom}(\nu' - \nu) = \mathsf{dom}(\mu' - \mu)$, then $N = () = N'$ and there exist $\mathbf{F}$ and $u_{\ell,\theta}$ such that $F = \lceil \mathbf{F} \rceil = F'$, $M = \mathbf{F}[(!u_{\ell,\theta})]$, and $M' = \mathbf{F}[\mu(u_{\ell,\theta})]$, $M'' = \mathbf{F}[\nu(u_{\ell,\theta})]$ with $\mu' = \mu$ and $\nu' = \nu$.*

PROOF: by cases on the proof of the transition $(M,\mu) \xrightarrow[F]{N} (M',\mu')$. ❑

Now trying to see to which bisimulation a pair (M,M) where M is G-typable may belong, we see from this lemma that we have in particular to examine the case where $M = \mathbf{F}[(!u_{\ell',\theta})]$, so that the bisimulation we are seeking would contain the pair $(\mathbf{F}[V_0], \mathbf{F}[V_1])$ where V_0 and V_1 are respectively the values stored at location $u_{\ell',\theta}$ in memories μ and ν such that $\mu \doteq^{G \cup \lceil \mathbf{F} \rceil, \ell} \nu$, with $\ell' \not\preceq_{G \cup \lceil \mathbf{F} \rceil} \ell$. By a case analysis on the context $\mathbf{F}$, one then sees that the bisimulation we are looking for should satisfy the following clauses, as confirmed by the lemma below. Given a global flow policy G and a confidentiality level ℓ, we define inductively two mutually recursive binary relations $\mathcal{U}_{G,\ell}$ and $\mathcal{S}_{G,\ell}$ on expressions, as follows:

$M \,\mathcal{U}_{G,\ell}\, N$ if M and N are both G-typable with $s.\mathsf{r} \preceq_G \ell$, and one of the following holds:

(Clause 1) $M = N$, or

(Clause 2) $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $M_0 \,\mathcal{U}_{G,\ell}\, N_0$, or

(Clause 3) $M = (M_0 M_1)$ and $N = (N_0 N_1)$ with $M_i \,\mathcal{U}_{G,\ell}\, N_i$ $(i = 0, 1)$, or

(Clause 4) $M = M_0 \,;\, M_1$ and $N = N_0 \,;\, N_1$ with $M_0 \,\mathcal{S}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{U}_{G,\ell}\, N_1$, or

(Clause 5) $M = (\mathsf{ref}_{\ell',\theta}\, M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\, N_0)$ with
(a) either $M_0 \,\mathcal{U}_{G,\ell}\, N_0$, or
(b) $\ell' \not\preceq_G \ell$ and $M_0 \,\mathcal{S}_{G,\ell}\, N_0$, or

(Clause 6) $M = (!\, M_0)$ and $N = (!\, N_0)$ with $M_0 \,\mathcal{U}_{G,\ell}\, N_0$, and both M_0 and N_0 have type $\theta\, \mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \preceq_G \ell$, or

(Clause 7) $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with
(a) either $M_0 \,\mathcal{U}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{U}_{G,\ell}\, N_1$, or

(b) $M_0 \,\mathcal{S}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{S}_{G,\ell}\, N_1$, and both M_0 and N_0 have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 8) $M = (\mathsf{flow}\ F\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N_0)$ with $M_0 \,\mathcal{U}_{F\cup G,\ell}\, N_0$.

$M \,\mathcal{S}_{G,\ell}\, N$ if M and N are both G-typable with $s.\mathsf{t} \preceq_G \ell$, and one of the following holds:

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

(Clause 2) $M, N \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$, or

(Clause 3) $M = N$, or

(Clause 4) $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with
(a) either $M_0 \,\mathcal{U}_{G,\ell}\, N_0$, or
(b) $M_0 \,\mathcal{S}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{S}_{G,\ell}\, M_2$, or

(Clause 5) $M = (M_0 M_1)$ and $N = (N_0 N_1)$ with $M_i \,\mathcal{U}_{G,\ell}\, N_i$ $(i = 0, 1)$, or

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

(Clause 7) $M = (\mathsf{ref}_{\ell',\theta}\, M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\, N_0)$ with
(a) either $M_0 \,\mathcal{U}_{G,\ell}\, N_0$, or
(b) $\ell' \not\preceq_G \ell$ and $M_0 \,\mathcal{S}_{G,\ell}\, N_0$, or

(Clause 8) $M = (!\, M_0)$ and $N = (!\, N_0)$ with $M_0 \,\mathcal{S}_{G,\ell}\, N_0$, or

(Clause 9) $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with
(a) either $M_0 \,\mathcal{U}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{U}_{G,\ell}\, N_1$, or
(b) $M_0 \,\mathcal{S}_{G,\ell}\, N_0$ and $M_1 \,\mathcal{S}_{G,\ell}\, N_1$, and both M_0 and N_0 have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 10) $M = (\mathsf{flow}\ F\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N_0)$ with $M_0 \,\mathcal{S}_{F\cup G,\ell}\, N_0$.

REMARK 5.2. *The relations $\mathcal{U}_{G,\ell}$ and $\mathcal{S}_{G,\ell}$ are symmetric. Moreover*
(i) $M \,\mathcal{U}_{G,\ell}\, N \;\&\; M \in \mathcal{V}al \;\Rightarrow\; N = M$;
(ii) $M \,\mathcal{S}_{G,\ell}\, N \;\&\; M \in \mathcal{V}al \;\Rightarrow\; N \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$.

The next lemma relates the typing of expressions with the operational semantics, and more precisely with the branching situation as stated in Lemma 5.1:

LEMMA 5.3. *Assume that $G;\Gamma \vdash \mathbf{F}[(!u_{\ell',\theta})] : s, \tau$ with $\ell' \not\preceq_{G\cup\lceil\mathbf{F}\rceil} \ell$, and let V_0 and V_1 be any values such that $\lceil\mathbf{F}\rceil, G;\Gamma \vdash V_i : \bot, \theta$ for $i = 0, 1$. Then*
(i) $s.\mathsf{r} \preceq_G \ell \;\Rightarrow\; \mathbf{F}[V_0] \,\mathcal{U}_{G,\ell}\, \mathbf{F}[V_1]$,
(ii) $s.\mathsf{t} \preceq_G \ell \;\Rightarrow\; \mathbf{F}[V_0] \,\mathcal{S}_{G,\ell}\, \mathbf{F}[V_1]$.

PROOF: we prove simultaneously (i) and (ii) by induction on the structure of $\mathbf{F}$.

- If $\mathbf{F} = []$, then $\ell' \preceq_G s.\mathsf{c}$ by the rule (DEREF), and therefore this case is not possible for (i). For (ii) in this case we conclude using Clause 1.

- If $\mathbf{F} = (\mathsf{if}\ \mathbf{F}'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ then we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \mathsf{bool}$ and $G;\Gamma \vdash M_i : s_i, \tau$, with $s'.\mathsf{r} \preceq_G s.\mathsf{r}$, and we use the induction hypothesis and Clause 2 for (i). For (ii), we distinguish two cases. If $s'.\mathsf{r} \preceq_G \ell$ then $\mathbf{F}'[V_0]\,\mathcal{U}_{G,\ell}\,\mathbf{F}'[V_1]$ by induction hypothesis, and we conclude using Clause 4(a). Otherwise, we have $M_1 \asymp^{G,\ell} M_2$ by the Lemma 4.8 since $s'.\mathsf{r} \preceq_G s_i.\mathsf{w}$. Since $s'.\mathsf{t} \preceq_G s.\mathsf{t} \preceq_G \ell$ we cannot have $s'.\mathsf{c} \preceq_G s.\mathsf{t}$, and therefore $M_i \in \mathcal{T}$, and we conclude using the induction hypothesis and Clauses 2 and 4(b).

- If $\mathbf{F} = (\mathbf{F}'N)$ then $G = F, G'$ with $G';\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s_0, \sigma \xrightarrow[F]{s_1} \tau$ and $G';\Gamma \vdash N : s_2, \sigma$, and we have $s_0.\mathsf{r} \preceq_G s.\mathsf{t}$ and $s_2.\mathsf{r} \preceq_G s.\mathsf{t}$. Then we conclude using the induction hypothesis and Clause 3 or 5. The argument is similar if $\mathbf{F} = (V\mathbf{F}')$.

- If $\mathbf{F} = \mathbf{F}'\,;N$ we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \sigma$ and $G;\Gamma \vdash N : s'', \tau$ with $s'.\mathsf{t} \preceq_G s.\mathsf{t}$ and $s''.\mathsf{r} \preceq_G s.\mathsf{r}$. Then in case (i) we have $N\,\mathcal{U}_{G,\ell}\,N$ by Clause 1, and $\mathbf{F}'[V_0]\,\mathcal{S}_{G,\ell}\,\mathbf{F}'[V_1]$ by induction hypothesis. We conclude using Clause 4. Since $s''.\mathsf{t} \preceq_G s.\mathsf{t}$, in case (ii) we use the induction hypothesis and Clause 6 to conclude.

- If $\mathbf{F} = (\mathsf{ref}_{\ell'',\theta'}\,\mathbf{F}')$ then $\tau = \theta'\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \theta'$ with $s'.\mathsf{r} \preceq_G \ell''$ and $s'.\mathsf{t} \preceq_G s.\mathsf{t}$. For case (i), we distinguish two cases. If $\ell'' \preceq_G \ell$ then we use the induction hypothesis and Clause 5(a) to conclude. Otherwise we have $\mathbf{F}'[V_0]\,\mathcal{S}_{G,\ell}\,\mathbf{F}'[V_1]$ by induction hypothesis, and we conclude using Clause 5(b). The argument is the same regarding the case (ii).

- If $\mathbf{F} = (!\,\mathbf{F}')$ then $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \tau\,\mathsf{ref}_{\ell''}$ with $s'.\mathsf{r} \curlyvee \ell'' \preceq_G s.\mathsf{r}$ and $s'.\mathsf{t} = s.\mathsf{t}$. In both cases we use the induction hypothesis and Clauses 6 or 8 to conclude.

- If $\mathbf{F} = (\mathbf{F}' := N)$ we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \sigma\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash N : s'', \sigma$ with $s'.\mathsf{r} \curlyvee s''.\mathsf{r} \preceq_G \ell''$ and $s'.\mathsf{t} \curlyvee s''.\mathsf{t} \preceq_G s.\mathsf{t}$. We distinguish two cases. If $\ell'' \preceq_G \ell$ then we use the induction hypothesis regarding (i) and Clause 7(a) or 9(a). Otherwise, we use the induction hypothesis regarding (ii) and Clause 7(b) or 9(b). The argument is similar if $\mathbf{F} = (V := \mathbf{F}')$.

- If $\mathbf{F} = (\mathsf{flow}\ F\ \mathsf{in}\ \mathbf{F}')$ we have $F, G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \tau$ with $s'.\mathsf{r} \preceq_{F\cup G} s.\mathsf{r}$ and $s'.\mathsf{t} \preceq_{F\cup G} s.\mathsf{t}$. Since $\lceil\mathbf{F}\rceil = F \cup \lceil\mathbf{F}'\rceil$, we conclude using the induction hypothesis and Clause 8 or 10. ❑

The following proposition establishes that the relations $\mathcal{U}_{G,\ell}$ and $\mathcal{S}_{G,\ell}$ are a kind of (G,ℓ)-bisimulation. For this to hold, we assume the $(*)$ property regarding the set $\mathcal{T}$ of expressions.

PROPOSITION 5.4.

(i) *If* $M\,\mathcal{U}_{G,\ell}\,N$ *and* $(M,\mu) \xrightarrow[F]{M''} (M',\mu')$, *with* $\mu \doteq^{F\cup G,\ell} \nu$ *and* u *is fresh for* ν *if* $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$, *then there exist* N', P *and* ν' *such that* $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow (N',\nu')$ *with* $M'\,\mathcal{U}_{G,\ell}\,N'$, $P \in \mathcal{H}_{G,\ell}$ *and* $\mu' \doteq^{G,\ell} \nu'$.

(ii) *If* $M\,\mathcal{S}_{G,\ell}\,N$ *with* $M \notin \mathcal{H}_{G,\ell}$ *and* $(M,\mu) \xrightarrow[F]{M''} (M',\mu')$, *with* $\mu \doteq^{F\cup G,\ell} \nu$ *and* u *is fresh for* ν *if* $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$, *then there exist* N', P *and* ν' *such that* $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow (N',\nu')$ *with* $M'\,\mathcal{S}_{G,\ell}\,N'$, $P \in \mathcal{H}_{G,\ell}$ *and* $\mu' \doteq^{G,\ell} \nu'$.

PROOF: we prove simultaneously the two points by induction on the definitions of $\mathcal{U}_{G,\ell}$ and $\mathcal{S}_{G,\ell}$. We begin with proving point (i), that is with examining the clauses defining $\mathcal{U}_{G,\ell}$.

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 1, that is $M = N$, then by Lemma 3.4 there exist N' and ν' such that $(N,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (N',\nu')$ with $\mu' \doteq^{F\cup G,\ell} \nu'$ and $\mathsf{dom}(\nu'-\nu) = \mathsf{dom}(\mu'-\mu)$. If $M' = N'$ then $M'\,\mathcal{U}_{G,\ell}\,N'$ by Clause 1 (and Subject Reduction). Otherwise, by Lemma 5.1 we have ($M'' = ()$ and) $M = \mathbf{F}[!u_{\ell',\theta}]$ with $M' = \mathbf{F}[\mu(u_{\ell',\theta})]$ and $N' = \mathbf{F}[\nu(u_{\ell',\theta})]$, and $F = \lceil\mathbf{F}\rceil$. Since $\mu(u_{\ell',\theta}) \neq \nu(u_{\ell',\theta})$, we have $\ell' \not\preceq_{F\cup G} \ell$, and therefore $M'\,\mathcal{U}_{G,\ell}\,N'$ by Lemma 5.3(i) above (and the Subject Reduction property).

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 2, that is $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $M_0\,\mathcal{U}_{G,\ell}\,N_0$, then we distinguish two cases. If M_0 is a value, that is $M_0 \in \{tt, ff\}$ (since M is typable), then $N_0 = M_0$ by Remark 5.2(i). Assume for instance that $M_0 = tt = N_0$. Then $M' = M_1$, and we may let $N' = M_1$ and $\nu' = \nu$, since $M'\,\mathcal{U}_{G,\ell}\,N'$ by Clause 1. Otherwise, $M' = (\mathsf{if}\ M_0'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$, and we use the induction hypothesis and Clause 2 to conclude.

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 3, that is $M = (M_0M_1)$ and $N = (N_0N_1)$ with $M_0\,\mathcal{U}_{G,\ell}\,N_0$ and $M_1\,\mathcal{U}_{G,\ell}\,N_1$, then there are three cases. If both M_0 and M_1 are values, we have $N_0 = M_0$ and $N_1 = M_1$ by Remark 5.2(i). Moreover $M_0 = \lambda x M_2$ and $M' = \{x \mapsto M_1\}M_2$, and we may let $N' = M'$ and conclude using Clause 1. If $M_0 \in \mathcal{V}al$ and $M' = (M_0M_1')$ with $(M_1,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_1',\mu')$, then by induction hypothesis there exist N_1' and ν' such that $(N_1,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (N_1',\nu')$ with $M_1'\,\mathcal{U}_{G,\ell}\,N_1'$. We have $(N,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (N_0N_1')$, and we conclude using Clause 3. If $M' = (M_0'M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$, then we use the induction hypothesis and Clause 3 to conclude.

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 4, that is $M = M_0\,;M_1$ and $N = N_0\,;N_1$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and $M_1\,\mathcal{U}_{G,\ell}\,N_1$, 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 $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$. In the latter case, we use the induction hypothesis (regarding $\mathcal{S}_{G,\ell}$) and Clause 4 to conclude. Otherwise, we have $N_0 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ by Remark 5.2(ii), and therefore, by $(*)$, there exist $V \in \mathcal{V}al$, $P \in \mathcal{H}_{G,\ell}$ and ν' such that $(N_0,\nu) \xrightarrow{P}\!\!\!\!\rightarrow\, (V,\nu')$ and $\mathsf{dom}(\nu'-\nu) \cap \mathsf{dom}(\mu') = \emptyset$, hence $\mu' \doteq^{G,\ell} \nu'$. Then $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[\emptyset]{()}\!\!\!\!\rightarrow\, (N_1,\nu')$.

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 5(a), that is $M = (\mathsf{ref}_{\ell',\theta}\,M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\,N_0)$ with $M_0\,\mathcal{U}_{G,\ell}\,N_0$, then either M_0 is a value and $M' = u_{\ell',\theta}$ where u is fresh for μ, $M'' = ()$, $F = \emptyset$ and $\mu' = \mu \cup \{u_{\ell',\theta} \mapsto M_0\}$, or $M' = (\mathsf{ref}_{\ell',\theta}\,M_0')$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$. In the latter case, we use the induction hypothesis and Clause 5(a) to conclude. Otherwise, we have $N_0 = M_0$ by Remark 5.2(i), and therefore $(N,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (u_{\ell',\theta}, \nu \cup \{u_{\ell',\theta} \mapsto M_0\})$ since u is fresh for ν, and we conclude using Clause 1.

• If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 5(b), that is $M = (\mathsf{ref}_{\ell',\theta}\,M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\,N_0)$ with $\ell' \not\preceq_G \ell$ and $M_0\,\mathcal{S}_{G,\ell}\,N_0$, then either M_0 is a value and $M' = u_{\ell',\theta}$ where u is fresh for μ, $M'' = ()$, $F = \emptyset$ and $\mu' = \mu \cup \{u_{\ell',\theta} \mapsto M_0\}$, or $M' = (\mathsf{ref}_{\ell',\theta}\,M_0')$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$. In the latter case, we use the induction hypothesis regarding $\mathcal{S}_{G,\ell}$ and Clause 5(b) to conclude. Otherwise, we have $N_0 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ by Remark 5.2(ii), and therefore there exist $V \in \mathcal{V}al$, $P \in \mathcal{H}_{G,\ell}$ and ν'' such that $(N_0,\nu) \xrightarrow{P}\!\!\!\!\rightarrow\, (V,\nu'')$ and $\mathsf{dom}(\nu''-\nu) \cap \mathsf{dom}(\mu') = \emptyset$, and u is fresh for ν''. Then $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (u_{\ell',\theta},\nu')$ where $\nu' = \nu'' \cup \{u_{\ell',\theta} \mapsto V\})$, since u is fresh for ν''. We have $\mu' \doteq^{G,\ell} \nu'$ since $\ell' \not\preceq_G \ell$, and we conclude using Clause 1.

- If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 6, that is $M = (!\,M_0)$ and $N = (!\,N_0)$ with $M_0\,\mathcal{U}_{G,\ell}\,N_0$, and both M_0 and N_0 have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \preceq_G \ell$, there are two cases. If $M_0 \in \mathcal{V}al$ then M_0 is a memory location $u_{\theta,\ell'}$, and $M' = \mu(u_{\theta,\ell'})$, $M'' = ()$, $F = \emptyset$ and $\mu' = \mu$. We have $N_0 = M_0$ by Remark 5.2(i). We let $N' = \nu(u_{\theta,\ell'})$ and we conclude using Clause 1 since $\ell' \preceq_G \ell \;\Rightarrow\; M' = N'$. Otherwise, $M' = (!\,M'_0)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; (M'_0,\mu')$, and we use the induction hypothesis (and Subject Reduction) and Clause 6 to conclude.

- If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 7(a), that is $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with $M_0\,\mathcal{U}_{G,\ell}\,N_0$ and $M_1\,\mathcal{U}_{G,\ell}\,N_1$ then there are three cases. If both M_0 and M_1 are values, then M_0 is a memory location $u_{\theta,\ell'}$, and $M' = () = M''$, $F = \emptyset$ and $\mu' = \mu[u_{\theta,\ell'} := M_1]$. We have $N_0 = M_0$ and $M_1 = N_1$ by Remark 5.2(i), and therefore $(N,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; ((),\nu')$ where $\nu' = \nu[u_{\theta,\ell'} := N_1]$. We conclude using Clause 1, since $M_1 = N_1 \;\&\; \mu \mathrel{\hat{=}}^{G,\ell} \nu \;\Rightarrow\; \mu' \mathrel{\hat{=}}^{G,\ell} \nu'$. If M_0 is a value (hence also N_o, by Remark 5.2(i)) and $M' = (M_0 := M'_1)$ with $(M_1,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; (M'_1,\mu')$, or if $M' = (M'_0 := M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; (M'_0,\mu')$, then we use the induction hypothesis and Clause 7(a) to conclude.

- If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 7(b), that is $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and $M_1\,\mathcal{S}_{G,\ell}\,N_1$, and both M_0 and N_0 have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, then there are again three cases. If both M_0 and M_1 are values, then M_0 is a memory location $u_{\theta,\ell'}$, and $M' = () = M''$, $F = \emptyset$ and $\mu' = \mu[u_{\theta,\ell'} := M_1]$. By Remark 5.2(ii) we have $N_0, N_1 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$, and therefore there exist $U \in \mathcal{V}al$, $P \in \mathcal{H}_{G,\ell}$ and ν_0 such that $(N_0,\nu) \xrightarrow{P}\!\!\!\!\rightarrow\; (U,\nu_0)$ and $\mathsf{dom}(\nu_0 - \nu) \cap \mathsf{dom}(\mu') = \emptyset$. Moreover, by Subject Reduction U has type $\theta\,\mathsf{ref}_{\ell'}$, and therefore U is a memory location $v_{\theta,\ell'}$. Since $N_1 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ there exist $V \in \mathcal{V}al$, $Q \in \mathcal{H}_{G,\ell}$ and ν_1 such that $(N_1,\nu_0) \xrightarrow{P}\!\!\!\!\rightarrow\; (V,\nu_1)$ and $\mathsf{dom}(\nu_1 - \nu) \cap \mathsf{dom}(\mu') = \emptyset$. Then we have $(N,\nu) \xrightarrow{R}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; ((),\nu')$ where $\nu' = \nu_1[v_{\theta,\ell'} := V]$. We conclude using Clause 1, since $\ell' \not\preceq_G \ell \;\Rightarrow\; \mu' \mathrel{\hat{=}}^{G,\ell} \nu'$. In the other cases, where M' is obtained by reducing M_0 or M_1, we use the induction hypothesis regarding $\mathcal{S}_{G,\ell}$ and Subject Reduction to conclude using Clause 7(b).

- If $M\,\mathcal{U}_{G,\ell}\,N$ by Clause 8, that is $M = (\mathsf{flow}\ F'\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F'\ \mathsf{in}\ N_0)$ with $M_0\,\mathcal{U}_{F'\cup G,\ell}\,N_0$, we have either $M' = M_0$ if $M_0 \in \mathcal{V}al$ (and $M'' = ()$, $F = \emptyset$ and $\mu' = \mu$), or $M' = (\mathsf{flow}\ F'\ \mathsf{in}\ M'_0)$ with $(M_0,\mu) \xrightarrow[F'']{M''}\!\!\!\!\rightarrow\; (M'_0,\mu')$ and $F = F' \cup F''$. In the latter case, we simply use the induction hypothesis and Clause 8 to conclude. In the former case, we have $N_0 = M_0$ by Remark 5.2(i), hence $(N,\nu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; (M_0,\nu)$ and we conclude using Clause 1.

Now we prove point (ii), examining the clauses defining $\mathcal{S}_{G,\ell}$. Notice that it cannot be that $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 1 or 2. We shall only investigate the clauses where the argument is different from the one we used regarding $\mathcal{U}_{G,\ell}$, namely 4(b), 6 and 8.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 4(b), that is $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and $M_1\,\mathcal{S}_{G,\ell}\,M_2$, then there are two cases. If M' is obtained by reducing M_0, we use the induction hypothesis to conclude. If M_0 is a value then $M' = M_i$ for $i = 1$ or $i = 2$ with $M'' = ()$, $F = \emptyset$ and $\mu' = \mu$, then $N_0 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ by Remark 5.2(ii), and therefore there exist $V \in \mathcal{V}al$, $P \in \mathcal{H}_{G,\ell}$ and ν' such that $(N_0,\nu) \xrightarrow{P}\!\!\!\!\rightarrow\; (V,\nu')$ and $\mathsf{dom}(\nu' - \nu) \cap \mathsf{dom}(\mu') = \emptyset$. By Subject Reduction V has type bool, that is $V \in \{tt, ff\}$, and therefore $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow\; (M_j,\nu')$, and we are done in this case.

- If $M\,\mathcal{S}_{G,\ell}\,N$ by Clause 6, that is $M = M_0\,;M_1$ and $N = N_0\,;M_1$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$, then either

$M_0 \in \mathcal{V}al$ and $M' = M_1$ (and $F = \emptyset$, $M'' = ()$ and $\mu' = \mu$), or M' is obtained by reducing M_0. In this latter case, we use the induction hypothesis and Clause 6 to conclude. Otherwise, $N_0 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ by Remark 5.2(ii), and we conclude like in the case of Clause 4 of $\mathcal{U}_{G,\ell}$, observing that $M_1 \, \mathcal{S}_{G,\ell} \, M_1$ by Clause 3.

- If $M \, \mathcal{S}_{G,\ell} \, N$ by Clause 8, that is $M = (!\, M_0)$ and $N = (!\, N_0)$ with $M_0 \, \mathcal{S}_{G,\ell} \, N_0$, there are two cases. If M' is obtained by reducing M_0, then we conclude using the induction hypothesis and Clause 8. Otherwise, M_0 is a value, that is a memory location $u_{\theta,\ell'}$ since M is typable, and $M' = \mu(u_{\theta,\ell'})$ with $M'' = ()$, $F = \emptyset$ and $\mu' = \mu$. Then $N_0 \in \mathcal{H}_{G,\ell} \cap \mathcal{T}$ by Remark 5.2(ii), and therefore there exist $V \in \mathcal{V}al$, $P \in \mathcal{H}_{G,\ell}$ and ν' such that $(N_0, \nu) \xrightarrow{P} (V, \nu')$ and $\mathsf{dom}(\nu' - \nu) \cap \mathsf{dom}(\mu') = \emptyset$. Since N is typable, by Subject Reduction V must be a reference $v_{\theta',\ell''}$, and therefore $(N_0, \nu) \xrightarrow{P} \xrightarrow[F]{M''} (N', \nu')$ where $N' = \nu'(v_{\theta',\ell''})$ and we conclude using Clause 1. ❑

Now we introduce, for any given G and ℓ, another binary relation $\mathcal{R}_{G,\ell}$ on expressions that will be the basis for building the bisimulation by means of which we show type soundness.

$M \, \mathcal{R}_{G,\ell} \, N$ if M and N are both G-typable and one of the following holds:

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

(Clause 2) $M = N$, or

(Clause 3) $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with
- (a) either $M_0 \, \mathcal{U}_{G,\ell} \, N_0$, or
- (b) $M_0 \, \mathcal{R}_{G,\ell} \, N_0$ and $M_1 \asymp^{G,\ell} M_2$, or

(Clause 4) $M = (M_0 M_1)$ and $N = (N_0 N_1)$ with
- (a) either $M_0 \, \mathcal{U}_{G,\ell} \, N_0$ and $M_1 \, \mathcal{U}_{G,\ell} \, N_1$, or
- (b) $M_0 \, \mathcal{S}_{G,\ell} \, N_0$ and $M_1 \, \mathcal{R}_{G,\ell} \, N_1$ and M_0, N_0 are (G,ℓ)-high functions, or
- (c) $M_0 \, \mathcal{R}_{G,\ell} \, N_0$ and $M_1 \asymp^{G,\ell} N_1$ and M_0, N_0 are (G,ℓ)-high functions, or

(Clause 5) $M = M_0 \,;\, M_1$ and $N = N_0 \,;\, M_1$ with
- (a) either $M_0 \, \mathcal{S}_{G,\ell} \, N_0$, or
- (b) $M_0 \, \mathcal{R}_{G,\ell} \, N_0$ and $M_1 \in \mathcal{H}_{G,\ell}$, or

(Clause 6) $M = (\mathsf{ref}_{\ell',\theta}\, M_0)$ and $N = (\mathsf{ref}_{\ell',\theta}\, N_0)$ with
- (a) either $M_0 \, \mathcal{U}_{G,\ell} \, N_0$, or
- (b) $\ell' \not\preceq_G \ell$ and $M_0 \, \mathcal{R}_{G,\ell} \, N_0$, or

(Clause 7) $M = (!\, M_0)$ and $N = (!\, N_0)$ with $M_0 \, \mathcal{R}_{G,\ell} \, N_0$, or

(Clause 8) $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with
- (a) either $M_0 \, \mathcal{U}_{G,\ell} \, N_0$ and $M_1 \, \mathcal{U}_{G,\ell} \, N_1$, or
- (b) $M_0 \, \mathcal{S}_{G,\ell} \, N_0$ and $M_1 \, \mathcal{R}_{G,\ell} \, N_1$, and both M_0 and N_0 have type $\theta\, \mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or
- (c) $M_0 \, \mathcal{R}_{G,\ell} \, N_0$ and $M_1 \asymp^{G,\ell} N_1$, and both M_0 and N_0 have type $\theta\, \mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, or

(Clause 9) $M = (\mathsf{flow}\ F\ \mathsf{in}\ M_0)$ and $N = (\mathsf{flow}\ F\ \mathsf{in}\ N_0)$ with $M_0\,\mathcal{R}_{F\cup G,\ell}\,N_0$.

The relation $\mathcal{R}_{G,\ell}$ is clearly symmetric.

REMARK 5.5. $M\,\mathcal{R}_{G,\ell}\,N\ \&\ M \in \mathcal{V}al\ \Rightarrow\ N \in \mathcal{V}al$

The following lemma is analogous to Lemma 5.3. 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 5.6. *If $G;\Gamma \vdash \mathbf{F}[(!u_{\ell',\theta})] : s, \tau$ with $\ell' \not\preceq_{G\cup\lceil\mathbf{F}\rceil} \ell$ then for any values $V_0, V_1 \in \mathcal{V}al$ such that $\lceil\mathbf{F}\rceil, G;\Gamma \vdash V_i : \bot, \theta$ we have $\mathbf{F}[V_0]\,\mathcal{R}_{G,\ell}\,\mathbf{F}[V_1]$.*

PROOF: by induction on the structure of $\mathbf{F}$. The proof is mostly the same as the one of Lemma 5.3, and therefore we only examine some cases.

- If $\mathbf{F} = (\mathsf{if}\ \mathbf{F}'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ then we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \mathsf{bool}$ and $G;\Gamma \vdash M_i : s_i, \tau$, with $s'.\mathsf{r} \preceq_G s_i.\mathsf{w}$. If $s_i.\mathsf{w} \preceq_G \ell$ for $i = 1$ or $i = 2$ then we use Lemma 5.3(1), the induction hypothesis and Clause 3(a) to conclude. Otherwise, we use the induction hypothesis and Clause 3(b).

- If $\mathbf{F} = (\mathbf{F}'N)$ then we have $G = F, G'$ with $G';\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s_0, \sigma \xrightarrow[F]{s_1} \tau$ and $G';\Gamma \vdash N : s_2, \sigma$, with $s_0.\mathsf{r} \preceq_{G'} s_1.\mathsf{w}$, $s_2.\mathsf{r} \preceq_{G'} s_1.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_{G'} s_2.\mathsf{w}$, hence also $s_0.\mathsf{r} \preceq_G s_1.\mathsf{w}$, $s_2.\mathsf{r} \preceq_G s_1.\mathsf{w}$ and $s_0.\mathsf{t} \preceq_G s_2.\mathsf{w}$. By Lemma 4.2 we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s'_0, \sigma \xrightarrow[F]{s_1} \tau$ and $G;\Gamma \vdash N : s'_2, \sigma$ for some s'_0 and s'_2 such that $s'_0.\mathsf{r} \preceq_G s_0.\mathsf{r}$, $s'_2.\mathsf{r} \preceq_G s_2.\mathsf{r}$, $s'_0.\mathsf{t} \preceq_G s_0.\mathsf{t}$ and $s'_2.\mathsf{w} = s_2.\mathsf{w}$. If $s_1.\mathsf{w} \preceq_G \ell$ then $s'_0.\mathsf{r} \preceq_G \ell$ and $s'_2.\mathsf{r} \preceq_G \ell$, and we use Lemma 5.3(i), Clause 1 of the definition of $\mathcal{U}_{G,\ell}$, and conclude by Clause 4(a). Otherwise, $\mathbf{F}'[(!u_{\ell',\theta})]$ is a (G,ℓ)-high function, and the same holds for both $\mathbf{F}'[V_0]$ and $\mathbf{F}'[V_1]$. Now if $s_2.\mathsf{w} \preceq_G \ell$ we have $s_0.\mathsf{t} \preceq_G \ell$, and we use Lemma 5.3(ii) and Clause 2 to conclude, by Clause 4(b). Otherwise, we have $N \in \mathcal{H}_{G,\ell}$ by Lemma 4.8, and we conclude using the induction hypothesis and Clause 4(c). The argument is similar if $\mathbf{F} = (V\mathbf{F}')$.

- If $\mathbf{F} = \mathbf{F}'\,;N$ we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \sigma$ and $G;\Gamma \vdash N : s'', \tau$ with $s'.\mathsf{t} \preceq_G s''.\mathsf{w}$. Depending on whether $s''.\mathsf{w} \preceq_G \ell$ or not, we conclude using Lemmas 5.3(ii) and 4.8, and Clause 5(a) or 5(b).

- If $\mathbf{F} = (\mathsf{ref}_{\ell'',\theta'}\,\mathbf{F}')$ then $\tau = \theta'\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \theta'$ with $s'.\mathsf{r} \preceq_G \ell''$. Depending on whether $\ell'' \preceq_G \ell$ or not, we conclude using Lemma 5.3(i)and Clause 6(a) or 6(b).

- If $\mathbf{F} = (\mathbf{F}' := N)$ we have $G;\Gamma \vdash \mathbf{F}'[(!u_{\ell',\theta})] : s', \sigma\,\mathsf{ref}_{\ell''}$ and $G;\Gamma \vdash N : s'', \sigma$ with $s'.\mathsf{r} \curlyvee s''.\mathsf{r} \preceq_G \ell''$ and $s'.\mathsf{t} \preceq_G s''.\mathsf{w}$. If $\ell'' \preceq_G \ell$, we use Lemma 5.3(i) and Clause 1 of the definition of $\mathcal{U}_{G,\ell}$ to conclude, using Clause 8(a). Otherwise, depending on whether $s''.\mathsf{w} \preceq_G \ell$, we conclude using Lemmas 5.3(ii) and 4.8, and Clause 2, by Clause 8(b) or 8(c). The argument is similar if $\mathbf{F} = (V := \mathbf{F}')$. ❑

PROPOSITION 5.7. *If $M\,\mathcal{R}_{G,\ell}\,N$ with $M \notin \mathcal{H}_{G,\ell}$ and $(M,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow (M',\mu')$, with $\mu \doteq^{F\cup G,\ell} \nu$ and u is fresh for ν if $u_{\ell',\theta} \in \mathsf{dom}(\mu' - \mu)$, then there exist N', P and ν' such that $(N,\nu) \xrightarrow{P}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow (N',\nu')$ with $M'\,\mathcal{R}_{G,\ell}\,N'$, $P \in \mathcal{H}_{G,\ell}$ and $\mu' \doteq^{G,\ell} \nu'$.*

PROOF: by induction on the definitions of $\mathcal{R}_{G,\ell}$, using Proposition 5.4. Since the proof is very similar to the one of Proposition 5.4, we only examine some cases.

- If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 3(b), that is $M = (\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ and $N = (\mathsf{if}\ N_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M_1 \asymp^{G,\ell} M_2$, then M_0 cannot be a value, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$, by lemma 3.7. Then $M' = (\mathsf{if}\ M'_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow (M'_0,\mu')$, and we

use the induction hypothesis to conclude.

• If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 4(b), that is $M = (M_0 M_1)$ and $N = (N_0 N_1)$ with $M_0\,\mathcal{S}_{G,\ell}\,N_0$ and $M_1\,\mathcal{R}_{G,\ell}\,N_1$ and M_0 and N_0 are (G,ℓ)-high functions, then by Lemma 4.8(ii) M cannot be a redex, with $M_0, M_1 \in \mathcal{V}al$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$. Then either $M' = (M_0' M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$, and we use Proposition 5.4(ii) and Clause 4(b) to conclude, or $M_0 \in \mathcal{V}al$ and $M' = (M_0 M_1')$ with $(M_1,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_1',\mu')$, and we use the induction hypothesis to conclude, by Clause 4(b).

• If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 4(c), that is $M = (M_0 M_1)$ and $N = (N_0 N_1)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M_1 \asymp^{G,\ell} N_1$ and N_0 are (G,ℓ)-high functions then, as above, M is not a redex. Moreover $M_0 \notin \mathcal{V}al$, since otherwise we would have $M \in \mathcal{H}_{G,\ell}$, and therefore it must be the case that $M' = (M_0' M_1)$ for some M_0' such that $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$. We conclude using the induction hypothesis and Clause 4(c).

• If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 5(b), 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 $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$, and we conclude using the induction hypothesis and Clause 5(b).

• If $M\,\mathcal{R}_{G,\ell}\,N$ by Clause 8(c), that is $M = (M_0 := M_1)$ and $N = (N_0 := N_1)$ with $M_0\,\mathcal{R}_{G,\ell}\,N_0$ and $M_1 \asymp^{G,\ell} N_1$, and both M_0 and N_0 have type $\theta\,\mathsf{ref}_{\ell'}$ for some θ and ℓ' such that $\ell' \not\preceq_G \ell$, then we have $M_0 \notin \mathcal{H}_{G,\ell}$ (otherwise we would have $M \in \mathcal{H}_{G,\ell}$, for $M_1 \in \mathcal{H}_{G,\ell}$ and $\ell' \not\preceq_G \ell$). Therefore, there exists M_0' such that $M' = (M_0' := M_1)$ with $(M_0,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M_0',\mu')$, and we conclude using the induction hypothesis and Clause 8(c). ❏

Finally we define the bisimulation $\mathcal{R}^\star_{G,\ell}$ we were looking for, as follows:

$$\frac{M\,\mathcal{R}_{G,\ell}\,N}{M\,\mathcal{R}^\star_{G,\ell}\,N} \qquad \frac{P \asymp^{G,\ell} Q \quad Q\,\mathcal{R}^\star_{G,\ell}\,R}{P\,\mathcal{R}^\star_{G,\ell}\,R} \qquad \frac{P\,\mathcal{R}^\star_{G,\ell}\,Q \quad Q \asymp^{G,\ell} R}{P\,\mathcal{R}^\star_{G,\ell}\,R}$$

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

and we have

Proposition 5.8. *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 $\mu \simeq^{F\cup G,\ell} \nu$ 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 $P\,\mathcal{R}^\star_{G,\ell}\,Q$. We only examine two cases.

• 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 $P' \asymp^{G,\ell} P\,\mathcal{R}_{G,\ell}\,Q$, 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, either P' is an expression and $(P,\mu) \xrightarrow[F]{\emptyset}\!\!\!\!\rightarrow\, (P',\mu')$ or $(P,\mu) \xrightarrow[F]{M''}\!\!\!\!\rightarrow\, (M',\mu')$ 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 5.7, and we have either $(Q,\nu) \xrightarrow{R}\!\!\!\!\rightarrow \xrightarrow[F]{\emptyset}\!\!\!\!\rightarrow (Q',\nu')$ or $(Q,\nu) \xrightarrow{R}\!\!\!\!\rightarrow \xrightarrow[F]{M''}\!\!\!\!\rightarrow (Q',\nu')$ for some Q', R and ν' such that $\mu' \doteq^{G,\ell} \nu'$ and $R \in \mathcal{H}_{G,\ell}$, and either $P' \,\mathcal{R}_{G,\ell}\, Q'$ or $M' \,\mathcal{R}_{G,\ell}\, Q'$. Then it is easy to see that there exists $R' \in \mathcal{H}_{G,\ell}$ such that either $(Q,\nu) \xrightarrow{*} ((Q' \parallel R'),\nu')$ or $(Q,\nu) \xrightarrow{*} (((Q' \parallel M'') \parallel R'),\nu')$. By definition of $\mathcal{R}^{\star}_{G,\ell}$, in the first case we have $P' \,\mathcal{R}^{\star}_{G,\ell}\, (Q' \parallel R')$, and $(M' \parallel M'') \,\mathcal{R}^{\star}_{G,\ell}\, ((Q' \parallel M'') \parallel R')$ in the second.

- If $P \,\mathcal{R}^{\star}_{G,\ell}\, R \asymp^{G,\ell} Q$ then by induction hypothesis there exist R' and μ'' such that $(R,\mu) \xrightarrow{*} (R',\mu'')$ with $P' \,\mathcal{R}^{\star}_{G,\ell}\, R'$, $\mu' \doteq^{G,\ell} \mu''$. We have $R' \in \mathcal{H}_{G,\ell}$, and therefore $P' \,\mathcal{R}^{\star}_{G,\ell}\, R' \asymp^{G,\ell} Q$, hence $P' \,\mathcal{R}^{\star}_{G,\ell}\, Q$, and a matching transition for Q is $(Q,\nu) \xrightarrow{*} (Q,\nu)$, since $\mathsf{dom}(\mu') \cap \mathsf{dom}(\nu) = \mathsf{dom}(\mu) \cap \mathsf{dom}(\nu)$. ❑

The Type Soundness theorem is an immediate consequence of this last result.

6. Related work

We have proposed a programming construct to deal with declassification in a controlled way, as regards the confidentiality levels involved in this operation. In our language, this construct can be used in a quite liberal way, with the only constraint that the sub-programs indeed comply with the flow policy in the scope of which they are. 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 [51, 54], 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 [32, 33, 35, 36, 49, 58] 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 (intentionally) more restrictive, since it disallows declassify operations that are not recognized as "robust". Nevertheless, 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 [36] is related to our non-disclosure property), that is, more precisely, to see how our security property could be made more accurate for dealing with robust declassification.

In another paper [42], 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 $u_H := \mathit{ff}\,;\, v_L := \mathsf{declassify}(!\,u_H, L)$ (which is accepted as secure by our type system) 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 not secure in our sense), 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 [42].

More recently, Chong and Myers introduced *declassification policies* [11], 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 [11] 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. [16], 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 [27]. 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 (with no other guarantee than the non-disclosure policy). 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 works that are the closest to ours are the recent ones by Sands & al. [9, 30]. In [30], Mantel and Sands consider, in addition to a given lattice structure of security levels, 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. This is quite similar to using a coercion operation, as in $v_{\ell'} := [\ell \preceq \ell'](!\, u_\ell)$. 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). 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. It might be that our flow declaration construct is reducible in some sense to using only coercions $[\ell \preceq \ell']M$, but the former is, from a programming point of view, obviously more flexible. Moreover, it is not immediately clear (as for declassify) which generalized notion of non-interference could support the latter. From a pragmatic point of view, the main difference we see is that in the work of Mantel and Sands, 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, especially in a context where principals are first-class values (see [49, 61]). 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). For a more complete comparison of our work with [30], showing the differences in detail, we refer to the first author's PhD Thesis [2].

In [9], Broberg and Sands introduce *flow locks*, that are used to selectively provide access to memory locations. In their language (which otherwise is very close to the one we used here, except for thread creation), they introduce constructs for manipulating locks, namely open and close. Compared to our block-structured approach to dynamically manipulating flow policies, this looks obviously more expressive, and Broberg and Sands indeed show how to encode various declassification scenarios, like our local flow declaration construct, as well as the declassification contruct used by Mantel and Sands in their work on intransitive non-interference [30]. One may expect that some semantical checks are needed to ensure correct use of the locks. For instance, one would certainly require in some cases that a lock does not remain indefinitely open. Also, one may wonder how flow locks can be used in a concurrent setting.

To conclude this section, let us mention a work by the first author of this paper, that shows that, as suggested earlier, local flow policies could be suitable for a mobile code setting. In [1, 2], the author considers a language similar to ours, but enriched with the notion of a resource locality, and a standard migration primitive. In this setting, a new kind of information leak arises, which she calls migration leaks. She then studies a generalization of the non-disclosure policy, called "non-disclosure for networks," and its enforcement using a type and effect system. The extended security policy relies on a richer notion of low equality of states, which reflects the location of resources and threads. The type system, which includes the typing of the programming constructs related to code mobility, involves some new typing conditions that are needed in order to tackle migration leaks. The generalization to a mobile code setting could be done quite smoothly, which testifies for the robustness of the flow declaration and the non-disclosure policy in other contexts.

7. Conclusion

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*" [59]. 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 (see [44] for a discussion of the various "dimensions" of declassification), we addressed the second question by proposing a simple construct for declassification based on dynamically varying flow policies. Although this idea has already been mentioned in the literature [6] (see [49]), it does not seem to have been previously studied in a formal way (in [50] 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. Moreover, we have shown that, for programs written in an expressive, higher-order imperative core language, this property can be enforced by static analysis. Then checking, piece by piece, that a program does not violate *local* flow policies provides the programmer with the same guarantee as the classical

[6] We notice that it is also similar in spirit to the dynamic granting of permissions, by means of block-structured temporary assertion of priviledges, that one finds in stack inspection mechanisms, see for instance [18].

non-interference property, while allowing him to deal with declassification. 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. Similarly, our construct for dynamically introducing flow policies can certainly be used in the context of various other programming paradigms.

Another contribution we made in this paper, which is not specifically related to declassification, lies in the type system, which we tried to make as precise as possible. The motivation is, again, pragmatic: to be useful, a type system should not reject too many programs. It is well-known that a decidable static analysis technique cannot, in general, be complete with respect to the property to ensure, like not running into an error of some kind, because the latter is generally undecidable. Then a trade-off has to be found, between the most conservative way – of not accepting any program – and a complete but non-computable method. In this respect, an issue that is specific to language-based security is the typing of termination leaks, which we had to face, because our security property is termination sensitive. We have shown that, by relying on the fact that some pieces of code belong to a given class $\mathcal{T}$ of terminating programs, we can improve a lot on the various proposals made regarding the typing of termination leaks in concurrent and imperative languages. We have shown that our type system is sound, whatever choice is made as regards the set $\mathcal{T}$ of strongly converging expressions, so that in order to obtain a practical type system, we only have to instantiate this parameter into a computable set of expressions. We could for instance choose $\mathcal{T}$ to be the class of terminating expressions defined at the end of Section 2 – or even to be $\emptyset$, which amounts to what we did in the workshop version of the paper, but one could hope to do better. Then an obvious topic of research, which is left open in this paper, is the characterization of a class $\mathcal{T}$ of strongly converging expressions which would be as large as possible, including in particular the ability of using functions. An obvious idea would be to exploit the classical results regarding strong normalization in typed λ-calculi. However, there is a technical difficulty, which is known since a long time but has, as far as we can see, no solution up to now. Namely, it is known since Landin's pioneering work on the implementation of functional languages [24] that circular higher-order references introduce non-termination, like for instance in (using ref without subscripts)

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

This particular expression cannot be typed in our type and effect system, but a variant of it like

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

is accepted, provided that V is typable. Then, to obtain a computable set of strongly converging expressions where functions can be used is still problematic. This issue is left for further work.

Acknowledgments We are grateful to the anonymous referees for their valuable suggestions.

References

[1] A. Almeida Matos, *Non-disclosure for distributed mobile code*, FST-TCS'05, Lecture Notes in Comput. Sci. 3821 (2005) 177-188.

[2] A. Almeida Matos, *Typing Secure Information Flow: Declassification and Mobility*, PhD Thesis, École Nationale Supérieure des Mines de Paris (2006).

[3] A. Almeida Matos, G. Boudol, *On declassification and the non-disclosure policy*, CSFW'05 (2005) 226-240.

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

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

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

[7] G. Boudol, *On typing information flow,* Intern. Coll. on Theoretical Aspects of Computing, Lecture Notes in Comput. Sci. 3722 (2005) 366-380.

[8] G. Boudol, I. Castellani, *Non-interference for concurrent programs and thread systems,* in "Merci, Maurice, A mosaic in honour of Maurice Nivat" (P.-L. Curien, Ed), Theoretical Comput. Sci. Vol. 281, No. 1 (2002) 109-130.

[9] N. Broberg, D. Sands, *Flow locks: towards a core calculus for dynamic flow policies,* ESOP'06, Lecture Notes in Comput. Sci. 3924 (2006) 180-196.

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

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

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

[13] 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.

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

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

[16] 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.

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

[18] C. Fournet, A. Gordon, *Stack inspection: theory and variants,* POPL'02 (2002) 307-318.

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

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

[21] M. Hicks, B. Hicks, S. Tse, S. Zdancewic, *Dynamic updating of information-flow policies,* FACS'05 (2005) 7-18.

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

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

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

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

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

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

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

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

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

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

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

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

[34] A.C. Myers, B. Liskov, *Complete, safe information flow with decentralized labels,* IEEE Symp. on Security and Privacy (1998).

[35] 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.

[36] A.C. Myers, A. Sabelfeld, S. Zdancewic, *Enforcing robust declassification and qualified robustness,* J. of Computer Security Vol. 14, No. 2 (2006) 157-196.

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

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

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

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

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

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

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

[44] A. Sabelfeld, D. Sands, *Dimensions and principles of declassification,* CSFW'05 (2005) 255-269.

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

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

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

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

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

[50] S. Tse, S. Zdancewic, *A design for a security-typed language with certificate-based declassification,* ESOP'05, Lecture Notes in Comput. Sci. 3444 (2005) 279-294.

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

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

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

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

[55] 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.

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

[57] S. Zdancewic, *Programming Languages for Information Security,* PhD Thesis, Cornell University (2002).

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

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

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

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