Secure Information Flow for a Concurrent Language with Scheduling

Gilles Barthe, Leonor Prensa Nieto

Secure Information Flow for a Concurrent Language with Scheduling

Gilles Barthe
INRIA Sophia-Antipolis, France
Gilles.Barthe@inria.fr

Leonor Prensa Nieto
LORIA, France
Leonor.Prensa@loria.fr

Abstract

Information flow type systems provide an elegant means to enforce confidentiality of programs. Using the proof assistant Isabelle/HOL, we have specified an information flow type system for a concurrent language featuring primitives for scheduling, and shown that typable programs are non-interfering for a possibilistic notion of non-interference. The development, which constitutes to our best knowledge the first machine-checked account of non-interference for a concurrent language, takes advantage of the proof assistant facilities to structure the proofs about different views of the programming language and to identify the relationships among them and the type system.

Our language and type system generalize previous work of Boudol and Castellani [13], in particular by including arrays and lifting several convenient but unnecessary conditions in the syntax and type system of [13]. We illustrate the generality of our language and the usefulness of our type system with a medium size example.

1 Introduction

Security mechanisms for mobile and embedded code, such as the Java Virtual Machine and the Common Language Runtime, guarantee that downloaded applications are innocuous in the sense that they comply with some basic policies related to typing, initialization or access control, etc. However, these mechanisms are often too weak to provide effective protection against untrusted code: see for example [30] for a critical analysis of the JavaCard firewall mechanism and [1] for a critical analysis of the Java stack inspection mechanism.

In order to enforce stronger security properties that are increasingly required in the context of security-sensitive applications, platforms for mobile and embedded code should be equipped with appropriate mechanisms that guarantee end-to-end security. One such mechanism for confidentiality is provided by information flow type systems, which perform at compile-time a static analysis that tracks the flow of information in a program execution, and provide a means to enforce statically that no information leakage will result from executing the

program. In fact, such type systems guarantee non-interference [18], a high-level security property that characterizes programs whose execution does not revel secret information.

The definition of non-interference is conditioned by the attacker model which describes the capabilities of the attacker, concerning for instance which observations it can make. In a sequential setting, there are several well-established variants of non-interference, that typically consider the input/output behaviors of programs. While the definitions of non-interference for sequential programs are well understood, providing appropriate definitions of non-interference in a concurrent setting is a difficult problem that currently remains under investigation; existing works either consider possibilistic security conditions that consider all possible outputs of concurrent programs (instead of the output in a sequential setting), see e.g. [13], or probabilistic security conditions that consider the probability distribution for the possible outputs of a program execution, see e.g. [43, 46, 51].

Scheduling is an other delicate issue that must be considered. At the level of the language definition, one can either extend the programming language with primitives for scheduling and synchronization, or deal with scheduling at the semantic level, i.e. at the level of the transition system. At the level of information flow, scheduling introduces additional difficulties, because programs that are deemed secure by possibilistic non-interference are subject to refinement attacks. Indeed, using a scheduler to execute non-deterministic programs may result in secure programs leaking confidential information. For example, the program

$$(\textbf{if } h = 0 \textbf{ then } \mathsf{skip} \textbf{ else } \mathit{sleep}(100)); l := 0 \parallel l := 1$$

is likely to terminate with $l = 0$ if $h = 0$ and a round robin scheduler is used. In order to avoid such refinement attacks, several approaches have been developed to account for schedulers.

One common approach for handling scheduling in information flow analyses is to focus on probabilistic non-interference, which deals with probabilistic parallel composition (or a generalization of it that allows to compose an arbitrary number of programs in parallel) and considers the probability distribution for the possible outputs of a program execution. For example, one can show probabilistic non-interference of programs assuming that probabilities in parallel composition are uniform [46, 51]. Another alternative is to adopt a stronger, scheduler independent, notion of security; for example, one can isolate a large class of schedulers, potentially probabilistic, and require programs to be secure for all the schedulers in this class [43].

Another approach, which we will follow, is to extend the programming language with primitives for scheduling, as e.g. in [13, 29]. In this approach, scheduling policies are represented by a concurrent program that is type-checked using the same rules as other concurrent programs. (Such a scenario of schedule-carrying code has been pursued independently in the context of embedded systems [20].)

In this paper, we focus on possibilistic non-interference for a concurrent

programming language $\mathcal{L}$ that features primitives for synchronization. The language, which is inspired from earlier work by G. Boudol and I. Castellani [13], features instructions to manipulate variables and arrays, sequential and parallel composition, the latter equipped with an interleaving semantics, branching statements and loops, as well as two constructs for scheduling and monitoring programs:

- the first construct `P⟦ Q⟧` combines two processes: the controller `P` and the controlled process `Q`: the intuitive semantics is that `Q` can only proceed, i.e. make one step of execution, if `P` can also make one step of execution. However, `P` is also allowed to proceed without `Q` making any step;
- the second construct **when** `e` **do** `P` is built from an expression `e` and a process `P`: the intuitive semantics is that **when** `e` **do** `P` can only proceed if `e` evaluates to `true` and if `P` can proceed to `P'`, in which case **when** `e` **do** `P` evaluates to **when** `e` **do** `P'`. Moreover, if `P` is the terminated program `skip`, then **when** `e` **do** `P` evaluates to `skip`.

The combination of these constructs is expressive enough to capture a variety of standard schedulers such as round robin schedulers or schedulers that select randomly a thread to execute.

One further issue with concurrency is the attacker model. While definitions of non-interference in sequential settings can be concerned with an input/output view of the program behavior, definitions of non-interference in concurrent settings usually aim at guaranteeing that confidential data is protected throughout the entire program execution, and are therefore based on suitable notions of bisimulation. There are two basic motivations for adopting definitions of non-interference based on bisimulation: first of all, one wants to prevent that a malicious thread can observe the behavior of other threads and adapt its behavior accordingly. Second, the termination behavior or timing behavior of program fragments can lead to undesired information leakage, as illustrated by the following example due to Smith and Volpano [47]. In this example, the program is constituted of the following three threads:

$$\begin{array}{lcl} \gamma & : & \textbf{if } PIN = 0 \textbf{ then } t_\alpha := tt \textbf{ else } t_\beta := tt \\ \alpha & : & \textbf{while } t_\alpha \neq tt \textbf{ do } \mathsf{skip}\ ;\ r := 0 \\ \beta & : & \textbf{while } t_\beta \neq tt \textbf{ do } \mathsf{skip}\ ;\ r := 1 \end{array}$$

where PIN, t_α, t_β are high variables and r is a low variable. The parallel composition of the three threads is not secure, since the value of PIN is copied into r through the control flow of the program.

Turning back to information flow type systems, enforcing non-interference for concurrent programs is considerably more complex that enforcing non-interference for sequential programs, in particular because the parallel composition of secure programs may fail to be secure, as illustrated by the above example due to Smith and Volpano [47] (indeed, each thread above is non-interfering and typable according to the information flow type system of [49] for sequential languages).

There are several possibilities to design a sound and compositional information flow type system that rejects the above example. One first possibility, developed by Smith and Volpano in [47] is to deal with typing judgments of the form $\vdash$ P $\triangleright$ t where t is a lower bound to the assignments that can potentially be performed by P, and to reject programs that contain a loop whose guard depends on a high variable. Another possibility, advocated by G. Boudol and I. Castellani [13] and adopted in this paper, is to consider a refined type system that deals with typing judgments of the form $\vdash$ P $\triangleright$ t s where t is a lower bound to the level of variables that can be assigned by P and s is an upper bound on the levels of guards that occur in P, and to reject programs that perform a low assignment after checking a high guard.

The main contribution of this paper is a machine-checked specification of an information flow type system for the language $\mathcal{L}$, and a machine-checked proof of soundness for the type system; all the work is conducted using Isabelle [32], a general purpose proof assistant that has been previously used successfully to formalize mathematics and programming language theory. The type system is inspired from the work of G. Boudol and I. Castellani [13], and its validity is established using their tools and definitions, including various notions of bisimulations. However, our formalization improves on the results of [13] in three significant ways: first, we consider a more general language that features unrestricted sequential composition, whereas [13] requires that the first process of a sequential composition is sequential. We allow an unrestricted use of controlling and scheduling constructs, whereas [13] imposes a very specific shape to concurrent thread systems. Second, our semantics of programs improves on the semantics of [13] by providing a more complete set of reduction rules, whereas [13] does not provide reduction rules for programs such as skip; skip or skip || skip. As a result, we can achieve our third, and most significant improvement over [13], namely to prove the soundness of our type system for a more general language.

Finally, formal proofs can also facilitate modular proofs. Many proof assistants, including Isabelle, offer support for modular developments, through which additional insights might be achieved. For example, we have exploited the *locales* facility of Isabelle to provide a modular proof of the language presented in [13], where they adopt a layered view of thread systems as a list of threads (each thread being a guarded sequential program) controlled by a parallel scheduler. We have also used locales to define various models of the memory.

Our work contributes to the growing evidence that proof assistants can be used to machine check research papers in programming language theory, and shows in particular that proof assistants are mature for verifying state-of-the-art type systems for information flow. Furthermore, our work shows that proof assistants allow to discard convenient but unnecessary assumptions in proofs (according to its authors, the assumption in [13] that the first process of a sequential composition is sequential was only made to keep proofs manageable). Perhaps less importantly, our work reveals a few minor imprecisions in the definitions and proofs of [13].

Contents of the paper

The remainder of the paper is organized as follows: Section 2 gives an introduction to Isabelle/HOL. Section 3 presents the programming language $\mathcal{L}$, its operational semantics, and illustrate its expressiveness by showing how common schedulers can be programmed in $\mathcal{L}$. Section 4 introduces the security condition to be enforced, defines an information flow type system for this purpose, and shows that the type system is sound in the sense that typable programs are non-interfering. Section 5 discusses some benefits and difficulties of machine-checking formalizations. Section 6 provides an example of a program that is typable with our type system. We conclude in Section 7 with related work and directions for further work.

This paper is an extended version of [10].

2 Modular formalization in Isabelle/HOL

Isabelle [32] is a generic interactive theorem prover which can be instantiated with different object logics. Isabelle/HOL is the instance for Higher-Order Logic.

In this section we present basic notions of Isabelle generally used for formalizing programming language theory and then give a brief introduction to the use of locales and interpretations. Locales provide a logical mean of structuring elements and proofs of a theory in modules. Interpretations allow us to formally prove relationships among these modules.

In this work we have modularized in two directions:

- The first direction concerns the memory model. We have performed several formalizations which differ on the degree of abstraction used to define the memory model. Sections 3 and 4 describe the formalization of the proof of non-interference using an abstract memory model based on lookup and update functions. In Section 6 we show an application of the type system to prove non-interference for a concrete example. For this we have used a concrete model of the memory where arrays are implemented as lists in Isabelle and where evaluation of variables and expressions is modeled as function evaluation. Passing from an abstract model to a more concrete one ensuring the consistency of the transformation can be formally done in Isabelle using locales and interpretations.

- The second direction consists in using modules to structure the programming language in several layers. We have used locales to model sequential programs, parallel programs and thread systems separately. We have then identified conditions on the base languages which are necessary to establish non-interference for the top-level layer. Thus, any implementation of the base languages can be accepted as long as these conditions are satisfied. Locales act in this setting as a sort of interface for the different levels. This formalization is further discussed in Section 4.4.

In this paper we have tried to present the formalization such that a non-expert in Isabelle is able to understand the notation without any problem. The rest of this section can thus be skipped. It serves as an introduction to the notation used in the actual Isabelle theories [35].

2.1 Basics

When formalizing a programming language in a theorem prover, one has to decide between using a *deep embedding*, where first the (abstract) syntax is represented via an inductive datatype and then a semantics is assigned to it, and a *shallow embedding*, where a term in the language is essentially an abbreviation of its semantics. Deep embeddings are useful when meta-theoretic reasoning (usually by induction over the syntax) is required. Shallow embeddings on the other hand, simplify reasoning about individual programs because one may work directly with the semantics avoiding the extra syntactic level. We use a combination of both styles that has become quite established. We formalize as much as possible using a shallow embedding and use a deep embedding only where needed in order to perform the meta-theoretic proofs we are interested in.

The program syntax is defined in Isabelle/HOL via a **datatype** definition. A free datatype is defined by listing its constructors together with their argument types, separated by '|'. Laws about datatypes, such as distinctness of constructors, are automatically included in the simplification tactics for future proofs. An induction principle, namely, structural induction over the constructors of the datatype is also generated with each datatype declaration. To use it in proofs it has to be explicitly invoked. Functions about datatypes are usually defined by primitive recursion. They are introduced by the keyword **primrec**.

Constants are declared with **consts** followed by their name and type, separated by '::'. Non-recursive definitions are declared by the keyword **constdefs**. The introduced constant and its definition are separated by '≡'.

The operational semantics and type rules of commands are inductively defined via a set of rules. Such inductively defined sets represent the least set which is closed under the formation rules. From each inductive definition Isabelle generates the corresponding induction principle, called *rule induction*, which represents the most powerful proof method used in this work. The so-called *inductive cases* proof principle is also automatically generated by the system for any inductive definition. It can be understood as the counterpart of (structural) case distinction on inductively generated elements. Whenever we have an assumption stating that an element belongs to an inductively defined set, we can distinguish on the last rule used for its derivation. When we speak of case analysis on an inductively generated element we refer to this proof principle.

Statements that we want to prove are preceded by **theorem** or **lemma**. There is no formal difference between them; we use one or the other depending on the importance we attach to the stated proposition. Proofs are done by applying *tactics* to the stated goals. The basic tactics are based on *resolution*,

i.e. by applying inference rules (backwards or forwards) in a natural deduction style, and *rewriting*, i.e. by applying (conditional) directed equalities. As a result, goals are reduced to simpler subgoals until they become trivial. When all subgoals are solved the proposition is proven and stored under some name given by the user.

Some tactics are based on natural deduction (forward and back-chaining of rules) where search with backtracking is automated using the so-called *classical reasoner*. Other tactics, called *simplifiers*, compose rewriting steps. More powerful tactics (like *auto*) combine both systems and are able to automatically prove complicated goals.

In an interactive theorem prover like Isabelle, if a statement cannot be proved automatically, the user is able to direct the proof by explicitly using induction, case distinction, instantiating variables, etc. Very often, automatic tools do succeed if they are supplied with suitable lemmas obtained from the Isabelle library or previously proven by the user.

We use **typedecl** to declare types without defining them. Type abbreviations in Isabelle are declared by the keyword **types**. They follow the syntax of ML, except that function types are denoted by $\Rightarrow$. The formalization uses some types and constants defined in the standard Isabelle/HOL library. In this paper we use two base types (*bool* and *nat*) and construct others using type constructors like *list*, *set* or the product type ($\times$) or function type ($\Rightarrow$).

2.2 Locales and Interpretations

Locales are an extension of Isabelle which provide support for modular reasoning. The use of locales provides a structured way of stating and managing facts but does not modify the language for proofs. In this section we introduce only those aspects of locales which are useful for our purposes. For a more complete description see [5].

A named locale is declared with the keyword **locale** followed by a name. Locales are (internally) lists of *context elements*. These are of four kinds, identified by the keywords **fixes**, **assumes**, **defines** and **notes**. Parameters of a locale correspond to the context element **fixes**, and assumptions may be declared with **assumes**. The following is a simple locale named loc_$\mathcal{M}$ which simply fixes a parameter eqmem with infix syntax and assumes that it is an equivalence relation.

typedecl $\mathcal{M}$

locale loc_$\mathcal{M}$ =
  fixes eqmem :: $\mathcal{M} \Rightarrow \mathcal{M} \Rightarrow$ bool (_ $\simeq$ _)
  assumes eqmem_refl: $\mu \simeq \mu$
  and eqmem_sym: $\mu \simeq \mu' \Longrightarrow \mu' \simeq \mu$
  and eqmem_trans: $\mu \simeq \mu' \wedge \mu' \simeq \mu'' \Longrightarrow \mu \simeq \mu''$

The context element **defines** adds a definition of the form $p\ x_1 \ldots x_n \equiv t$ as an assumption, where p is a parameter of the locale and t a term that may

contain the x_i. It is possible to add facts to a locale and to export facts from it. The difference between **assumes** and **defines** concerns the way parameters are treated on export. This feature is, however, not relevant for our case study. Adding facts, which must be theorems, to a locale is done by using **notes**.

Locales can be combined. The locale expresion $e_1 + e_2$ denotes the locale obtained by merging the locales e_1 and e_2. This locale contains the context elements of e_1 followed by the context elements of e_2.

Locale expressions may be instantiated. This requires a proof of the instantiated specification and is called *locale interpretation*. There are various ways of interpreting locales. We use the form **interpretation** `loc1` ⊆`loc2` which interprets `loc2` in `loc1` and requires a proof that the specification of `loc1` implies the specification of `loc2`. More details about locales instantiation and interpretation can be found in [33].

3 The programming language

3.1 Language definition

The definition of processes is parameterized by four types: a type $\mathcal{V}$ of variables, a type $\mathcal{T}$ of array variables, a type $\mathbb{A}$ of arithmetic expressions, and a type $\mathbb{B}$ of boolean expressions. These types are introduced at the onset of the formalization with the following declarations:

```
typedecl 𝒱
typedecl 𝒯

typedecl 𝔸
typedecl 𝔹
```

In our formalization, parameterization is understood as a dependence relation and amounts to declaring variables and expressions before declaring the type of processes. Processes are defined inductively via a **datatype** definition with nine constructors. Enclosed in parentheses we give the concrete syntax for each one. We use `":=="` for assignments and `";;"` for sequential composition to avoid clashes with the predefined `":="` and `";"` of Isabelle.

```
datatype ℙ =
   Skip                    ( skip )
 | Assign 𝒱 𝔸               ( _ :== _ )
 | Aassign 𝒯 𝔸 𝔸            ( _ [_]:==_ )
 | Seq ℙ ℙ                 ( _ ;; _ )
 | Cond 𝔹 ℙ ℙ              ( if _ then _ else _ )
 | While 𝔹 ℙ               ( while _ do _ )
 | Par ℙ ℙ                 ( _ ∥ _ )
 | When 𝔹 ℙ                ( when _ do _ )
 | Control ℙ ℙ             ( _ ⟦ _ ⟧)
```

Defining processes as a datatype allows us to reason about properties of processes using an induction principle on the structure of processes. Such an in-

duction principle is used extensively in our proofs.

In [13], processes are introduced in three successive layers: sequential processes, concurrent processes, and thread systems. While the language of [13] is strictly more restrictive than ours, e.g. because it does not consider processes of the form `(P ∥ Q);; R`, it is possible to recover the three families of processes with appropriate predicates. For example, the predicate `issqtl` is defined as the following recursive function in Isabelle:

```
consts
  issqtl :: ℙ ⇒ bool
primrec
  issqtl (skip ) = True
  issqtl (x:== a) = True
  issqtl (y[e]:== a) = True
  issqtl (P;; Q) = (issqtl P ∧ issqtl Q)
  issqtl ( if  b then P  else  Q) = (issqtl P ∧ issqtl Q)
  issqtl (while b do P) = issqtl P
  issqtl (P  ∥  Q) = False
  issqtl (when b do P) = False
  issqtl (P⟦Q⟧) = False
```

Sequential processes enjoy a stronger security property than arbitrary processes, as shown in Section 4.4.

3.2 Operational semantics

The operational semantics of processes is defined relative to the definition of processes and to a memory model. For the memory we use an abstract model based on lookup and update functions with the usual properties.

```
typedecl 𝓜

consts 𝒱_lookup:: 𝓜 ⇒ 𝒱 ⇒ nat
consts 𝒯_lookup:: 𝓜 ⇒ 𝒯 ⇒ nat ⇒ nat

consts 𝒱_update:: 𝓜 ⇒ 𝒱 ⇒ nat ⇒ 𝓜
consts 𝒯_update:: 𝓜 ⇒ 𝒯 ⇒ nat ⇒ nat ⇒ 𝓜
```

Evaluation of arithmetic and boolean expressions is defined via functions that take as input a memory and return a natural number (for arithmetic expressions) or a boolean (for boolean expressions).

```
consts ℰ𝔸:: 𝓜 ⇒ 𝔸 ⇒ nat
consts ℰ𝔹:: 𝓜 ⇒ 𝔹 ⇒ bool
```

We define an abstract function that returns the size of an array variable.

```
consts 𝒯_size:: 𝓜 ⇒ 𝒯 ⇒ nat
```

Before proceeding with the formal definition, we briefly comment on some hidden assumptions of our memory model w.r.t. the operational semantics of expressions: modeling expressions as total functions implicitly carries three important facts: first of all, expressions can always be evaluated (termination)

and will always return a result (no exception). Furthermore, evaluating an expression will always return a single result (determinacy). The first and third assumptions are already present in [13, 47]; the second assumption already appears in [15] where it is assumed that array lookups never raise an exception and return an arbitrary value if the lookup is performed at an index that is out of bounds.

The operational semantics of processes is defined inductively via transition rules between configurations. A configuration is a pair of the form (P, μ). For clarity, we declare the following type abbreviation for configurations:

types $\mathbb{C} = \mathbb{P} \times \mathcal{M}$

We use a readable infix syntax for transition rules (most of the definitions presented in this paper are endowed with a readable infix syntax whose formal declaration is not always explicitly shown here). The transition rules are shown in Figure 1.

In the first rule for the control construct (*Control1*), the controlling command and the controlled command may each have altered memory. Thus, the effect of the composite command must combine these alterations. The `merge` function is an auxiliary that describes how these effects are to be merged.

consts $\texttt{merge\_mem} :: \mathcal{M} \Rightarrow \mathcal{M} \Rightarrow \mathcal{M} \Rightarrow \mathcal{M} \quad (\_ \sqcup \_ \_)$

Some properties about the merging of memories are needed in the proofs. We state them as axioms. The following one concerns the evaluation of the size of an array in a merged memory:

axioms $\mathcal{T}\texttt{\_size\_merge}$:
$$\mathcal{T}\texttt{\_size}\ \mu\ \texttt{y} = \mathcal{T}\texttt{\_size}\ \mu'\ \texttt{y} \wedge \mathcal{T}\texttt{\_size}\ \mu'\ \texttt{y} = \mathcal{T}\texttt{\_size}\ \mu''\ \texttt{y} \Longrightarrow \mathcal{T}\texttt{\_size}\ \mu\ \texttt{y} = \mathcal{T}\texttt{\_size}\ (\mu' \sqcup_\mu \mu'')\ \texttt{y}$$

Three more axioms concerning memory merging are stated in Section 4.1. These are the only four axioms governing merge. It is worth to notice how little these axioms really require. In Section 6 we give an implementation of the merging function that satisfies these four properties.

The rules for arrays are similar to the rules of [15]. There are two cases: either the array assignment is performed within the bounds of the array, in which case the update is performed in the usual way, or it is performed outside the bounds of the array, in which case it has no effect and leaves the memory unchanged.

Concentrating on the fragment of the language without arrays, observe that there are several differences with respect to [13]. First of all, we have introduced several rules for enabling reduction of processes that would be irreducible otherwise, namely the rules *ParLN*, *ParRN*, *WhenSkip*, and *ControlSkip*. Furthermore, the rule *Seq1* for sequential composition has been modified to allow reduction of processes of the form **skip**`;;... ;;` **skip** which were irreducible when using the rule of [13]:

$$\frac{(P, \mu) \longrightarrow_1 (P', \mu')}{(\textbf{skip};;P,\ \mu) \longrightarrow_1 (P', \mu')}$$

Incidentally, this modification of the rule `Seq1` also solves a minor flaw in the proof of non-interference presented in [13].

Our operational semantics also differs from the one of [13] in its treatment of scheduling constructs. Firstly, we do not require **when** processes whose guard evaluates to true, and whose body is "stuck" to reduce to themselves in one step, whereas [13] considers the rather unintuitive rule:

$$\frac{\texttt{b}\ \mu \qquad \texttt{stuck P}}{(\textbf{when b do P},\ \mu) \longrightarrow_1 (\textbf{when b do P},\ \mu)}$$

where `stuck P` is defined as $\forall$ `Q` ν. $\neg$`(P,`μ`)`$\longrightarrow_1$`(Q,`ν`)`.

Secondly, we allow the possibility of the controller to progress independently of the controlled program, even though the latter is not "stuck". One may argue that this latter departure from [13] is undesirable because it allows for "unfair" execution traces: for example the program `x:==0;; P` $[\![$`x:==1` $]\!]$ may terminate with `x` being equal to `0`. This departure from the operational semantics of [13] is required for showing the soundness of the type system introduced in Section 4; it is however possible to keep the rule of [13]

$$\frac{(\texttt{P},\ \mu) \longrightarrow_1 (\texttt{P}',\ \mu') \qquad \texttt{stuck Q}}{(\texttt{P}[\![\texttt{Q}]\!],\ \mu) \longrightarrow_1 (\texttt{P}'[\![\texttt{Q}]\!],\ \mu')}$$

at the price of modifying the typing rule for controlled processes.

In order to define the security condition and prove the soundness of the information flow type sytem, one needs to introduce the reflexive closure and reflexive transitive closure of $\longrightarrow_1$. Both are defined inductively and are denoted respectively by $\longrightarrow_{0,1}$ and $\longrightarrow^\star$.

4 Information flow type system

In this section, we introduce an information flow type system for our language and show that it enforces a security condition based on bisimulation and inspired from [13]. We also discuss soundness proofs for type systems that are closer to [13], as well as a proof of a stronger security condition for sequential programs.

4.1 The security condition

The definition of non-interference presupposes a preordered set of security levels. Furthermore, the security condition is defined relative to a downwards-closed set of "low" security levels which we notate $\mathbb{L}$. Formally, we declare a new type `level` as an instance of the predefined axiomatic class `quasi_order` that declares a binary relation $\sqsubseteq$ satisfying reflexivity and transitivity, and declare a downwards-closed predicate over levels.

```
typedecl level
instance level :: quasi_order
```

$$\textit{Assign}\quad \frac{}{(\texttt{x :== a},\ \mu)\longrightarrow_1(\textbf{skip},\ \mathcal{V}\texttt{\_update}\ \mu\ \texttt{x}\ (\mathcal{E}_{\mathbb{A}}\ \mu\ \texttt{a}))}$$

$$\textit{Array}1\quad \frac{\mathcal{E}_{\mathbb{A}}\ \mu\ \texttt{e}{=}\epsilon \quad \mathcal{E}_{\mathbb{A}}\ \mu\ \texttt{a}{=}\alpha \quad \epsilon < \mathcal{T}\texttt{\_size}\ \mu\ \texttt{y}}{(\texttt{y[e] :== a},\ \mu)\longrightarrow_1(\textbf{skip},\ \mathcal{T}\texttt{\_update}\ \mu\ \texttt{y}\ \epsilon\ \alpha)}$$

$$\textit{Array}2\quad \frac{\mathcal{T}\texttt{\_size}\ \mu\ \texttt{y} \leq \mathcal{E}_{\mathbb{A}}\ \mu\ \texttt{e}}{(\texttt{y[e] :== a},\ \mu)\longrightarrow_1(\textbf{skip},\ \mu)}$$

$$\textit{Seq}1\quad \frac{}{(\textbf{skip};;\ \texttt{P},\ \mu)\longrightarrow_1(\texttt{P},\ \mu)}$$

$$\textit{Seq}2\quad \frac{(\texttt{P},\ \mu)\longrightarrow_1(\texttt{P'},\ \mu')}{(\texttt{P};;\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{P'};;\ \texttt{Q},\ \mu')}$$

$$\textit{CondT}\quad \frac{\mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b}}{(\ \textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{P}\ \textbf{else}\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{P},\ \mu)}$$

$$\textit{CondF}\quad \frac{\neg\ \mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b}}{(\ \textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{P}\ \textbf{else}\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{Q},\ \mu)}$$

$$\textit{WhileT}\quad \frac{\mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b}}{(\textbf{while}\ \texttt{b}\ \textbf{do}\ \texttt{P},\ \mu)\longrightarrow_1(\texttt{P};;\ \textbf{while}\ \texttt{b}\ \textbf{do}\ \texttt{P},\ \mu)}$$

$$\textit{WhileF}\quad \frac{\neg\ \mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b}}{(\textbf{while}\ \texttt{b}\ \textbf{do}\ \texttt{P},\ \mu)\longrightarrow_1(\textbf{skip},\ \mu)}$$

$$\textit{ParL}\quad \frac{(\texttt{P},\ \mu)\longrightarrow_1(\texttt{P'},\ \mu')}{(\texttt{P}\ \|\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{P'}\ \|\ \texttt{Q},\ \mu')}$$

$$\textit{ParLN}\quad \frac{}{(\textbf{skip}\ \|\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{Q},\ \mu)}$$

$$\textit{ParR}\quad \frac{(\texttt{Q},\ \mu)\longrightarrow_1(\texttt{Q'},\ \mu')}{(\texttt{P}\ \|\ \texttt{Q},\ \mu)\longrightarrow_1(\texttt{P}\ \|\ \texttt{Q'},\ \mu')}$$

$$\textit{ParRN}\quad \frac{}{(\texttt{P}\ \|\ \textbf{skip},\ \mu)\longrightarrow_1(\texttt{P},\ \mu)}$$

$$\textit{When}1\quad \frac{\mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b} \quad (\texttt{P},\ \mu)\longrightarrow_1(\texttt{P'},\ \mu')}{(\textbf{when}\ \texttt{b}\ \textbf{do}\ \texttt{P},\ \mu)\longrightarrow_1(\textbf{when}\ \texttt{b}\ \textbf{do}\ \texttt{P'},\ \mu')}$$

$$\textit{WhenSkip}\quad \frac{\mathcal{E}_{\mathbb{B}}\ \mu\ \texttt{b}}{(\textbf{when}\ \texttt{b}\ \textbf{do skip},\ \mu)\longrightarrow_1(\textbf{skip},\ \mu)}$$

$$\textit{Control}1\quad \frac{(\texttt{P},\ \mu)\longrightarrow_1(\texttt{P'},\ \mu') \qquad (\texttt{Q},\ \mu)\longrightarrow_1(\texttt{Q'},\ \mu'')}{(\texttt{P}[\![\texttt{Q}]\!],\ \mu)\longrightarrow_1(\texttt{P'}[\![\texttt{Q'}]\!],\ \mu' \sqcup_\mu \mu'')}$$

$$\textit{Control}2\quad \frac{(\texttt{P},\ \mu)\longrightarrow_1(\texttt{P'},\ \mu')}{(\texttt{P}[\![\texttt{Q}]\!],\ \mu)\longrightarrow_1(\texttt{P'}[\![\texttt{Q}]\!],\ \mu')}$$

$$\textit{ControlSkip}\quad \frac{}{(\textbf{skip}[\![\texttt{Q}]\!],\ \mu)\longrightarrow_1(\textbf{skip},\ \mu)}$$

Figure 1: Operational semantics

```
consts 𝕃 :: level ⇒ bool
axioms Ldown: 𝕃 x' ∧ x ⊑ x' ⟹ 𝕃 x
```

The definition of non-interference presupposes that levels are attached to variables: in the case of standard variables, one attaches a single security level, whereas in the case of array variables we follow [15] and distinguish between the security level of the array elements and of the array length. Formally, we assume given functions:

```
consts 𝓛_v : 𝒱 ⇒ level
consts 𝓛_t : 𝒯 ⇒ (level × level)
```

The function $\mathcal{L}_t$ returns pairs of levels, the first one corresponding to the security level of the array elements, and the second one corresponding to the security level of the length of the array. In order to avoid information leakage for typable programs, Deng and Smith [15] require that the length of the array is assigned a security level that is lower than the security level of the array elements, which is achieved by introducing the axiom:

```
axioms level_size_leq_contents: let (𝓛_t y)= (l₁,l₂) in l₂ ⊑ l₁
```

The above functions discriminate between variables that are visible by an attacker that has access to the $\mathbb{L}$-part of the memory. One fundamental ingredient in the security condition is the definition of equality of memories, taken from the point of view of the attacker; such an equality on memories is defined relative to $\mathbb{L}$, and is termed $\mathbb{L}$-equality. For such a notion of equality, two memories are equal if they coincide on low variables. In the case of array variables, the latter is to be understood as pointwise equality concerning the contents of arrays for arrays with low content, and as equality on their length for arrays whose length is tagged as low. Formally,

```
constdefs ≃_v :: ℳ ⇒ ℳ ⇒ bool
 μ ≃_v μ' ≡ ∀ x. 𝕃 (𝓛_v x) ⟹ 𝒱_lookup μ x = 𝒱_lookup μ' x

constdefs ≃_t :: ℳ ⇒ ℳ ⇒ bool
 μ ≃_t μ' ≡ let 𝓛_t y=(l₁,l₂) in
       ∀ y. 𝕃 l₁ ⟹ 𝒯_lookup μ y = 𝒯_lookup μ' y
          ∧ 𝕃 l₂ ⟹ 𝒯_size μ y = 𝒯_size μ' y

constdefs ≃ :: ℳ ⇒ ℳ ⇒ bool
 μ ≃ μ' ≡ μ ≃_v μ' ∧ μ ≃_t μ'
```

Three axioms involving equivalence of memories and merged memories as announced in Section 3.2 are needed in the proofs.

```
axioms
 eqmem_merge1: μ ≃ ν ∧ μ' ≃ ν' ∧ μ'' ≃ ν'' ⟹ μ' ⊔_μ μ'' ≃ ν' ⊔_ν ν''

 eqmem_merge2: μ ≃ μ' ∧ μ ≃ μ'' ⟹ μ ≃ μ' ⊔_μ μ''

 eqmem_merge3: ∀y. 𝒯_size μ y = 𝒯_size μ' y ∧ μ ≃ μ''
               ⟹ μ' ⊔_μ μ'' ≃ μ'
```

The predicate `bisim` defines when a relation `R` on configurations is a *bisimulation*. This definition uses the notion of $\mathbb{L}$-equality between memories.

```
constdefs  bisim :: (ℂ ⇒ ℂ ⇒ bool) ⇒ bool
bisim R ≡ ∀P μ Q ν. R (P, μ) (Q, ν) ⟹
 R (Q, ν) (P, μ) ∧ μ ≃ ν ∧
 (∀P' μ'. (P, μ) ⟶₁ (P', μ') ⟹ (∃Q' ν'. (Q, ν) ⟶₀,₁ (Q', ν')
 ∧ R (P', μ') (Q', ν')))
```

Observe that the definition implies that the relation `R` is symmetric.

Given a relation as a set of pairs of programs we define a function that returns the corresponding relation (as a predicate) between configurations.

```
constdefs p2c :: (ℙ × ℙ) set ⇒ (ℂ ⇒ ℂ ⇒ bool)
 p2c S ≡ (λ (P, μ) (Q, ν). (P, Q) ∈ S ∧ μ ≃ ν)
```

The domain of bisimulations characterizes *secure* programs.

```
constdefs secure :: ℙ ⇒ bool
 secure P ≡ ∃S. bisim (p2c S) ∧ (P, P) ∈ S
```

4.2 Type system

Type judgments are defined relative to a context Γ which provides information about the security level of program variables, and are of the form $\vdash$ `P ▷ t s`, where:

- `t` is a *lower bound* on the level of the assigned variables of `P`;
- `s` is the *guard type*, i.e. an *upper bound* on the level of the loop and conditional guards occurring in `P`.

The following subsumption rule

$$\frac{\vdash \texttt{P} \triangleright \texttt{t s} \qquad \texttt{t'} \sqsubseteq \texttt{t} \qquad \texttt{s} \sqsubseteq \texttt{s'}}{\vdash \texttt{P} \triangleright \texttt{t' s'}}$$

is a valid derived rule in our system.

The type system is parameterized by functions that provide the security levels of variables and arrays (already required to formulate the security condition) and functions $\mathcal{L}_a$ and $\mathcal{L}_b$ that provide the security level of arithmetic and boolean expressions, respectively.

```
consts ℒa :: 𝔸 ⇒ level
       ℒb :: 𝔹 ⇒ level
```

In the sequel we assume that the security level of expressions is correct in the sense that evaluating a low expression with low equal memories should yield the same result. Such an assumption corresponds to [13, Assumption 3.3.] and is formalized by the following axioms about the behaviour of arithmetic and boolean expressions:

axioms
  beh_$\mathbb{A}$: $\mu \simeq \mu' \wedge \mathbb{L}$ ($\mathcal{L}_a$ a) $\Longrightarrow$ ($\mathcal{E}_{\mathbb{A}}$ μ a)=($\mathcal{E}_{\mathbb{A}}$ μ' a)
  beh_$\mathbb{B}$: $\mu \simeq \mu' \wedge \mathbb{L}$ ($\mathcal{L}_b$ b) $\Longrightarrow$ ($\mathcal{E}_{\mathbb{B}}$ μ b)=($\mathcal{E}_{\mathbb{B}}$ μ' b)

Formally, the type system is defined inductively using inference rules. The set of rules is shown in Figure 2. We say that a program is *typable* if there exist types such that a typing judgment can be derived in the type system.

constdefs typable :: $\mathbb{P} \Rightarrow$ bool
  typable P $\equiv \exists$t s. $\vdash$ P $\triangleright$ t s

We found it convenient to use a syntax-directed set of rules, i.e. a set of rules in which there is only one rule applicable for any construct. The main advantages of syntax-directedness are to avoid the use of the meet and join operators as well as dealing with the subsumption rule.

In order to achieve syntax-directedness, one must avoid rules that can be applied to different constructs, such as the subsumption rule shown above, that apply at any point in the derivation, and "include" the effects of such rules in the rules attached to a specific construct by including the necessary subtyping relations in those rules.

Our type system coincides with the type system of [13] for its concurrent fragment, but adopts quite different rules for the scheduling constructs `Control` and `When`. In fact, our typing rules from *Control* and *When* are identical to the typing rules for *Seq* and *Cond* whereas the typing rules of [13] require:

$$\textit{When}\quad \frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \qquad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{s}' \qquad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{t} \qquad \mathtt{t}' \sqsubseteq \mathtt{t}}{\vdash \textbf{when}\ \mathtt{b}\ \textbf{do}\ \mathtt{P} \triangleright \mathtt{t}'\ \mathtt{s}'}$$

$$\textit{Control}\quad \frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \qquad \vdash \mathtt{Q} \triangleright \mathtt{t}\ \mathtt{s} \qquad \mathtt{s} \sqsubseteq \mathtt{t} \qquad \mathtt{t}' \sqsubseteq \mathtt{t} \qquad \mathtt{s} \sqsubseteq \mathtt{s}'}{\vdash \mathtt{P}[\![\mathtt{Q}]\!] \triangleright \mathtt{t}'\ \mathtt{s}'}$$

These rules do not reflect the intuitive interpretation of the typing judgments, namely that $\vdash$ P $\triangleright$ t s implies that t is a lower bound to the level of variables that can be assigned by P and that s is an upper bound on the levels of guards that occur in P. Furthermore, the rule for `When` is unsound if applied to our extended syntax, as illustrated by the program P below, where e and z are low, and y is high:

when e **do** (**when** y **do skip**) ;; z:==1

The rules of [13] allow us to derive $\vdash$ P $\triangleright$ low low, whereas the program is insecure. Note that, in contrast to [13], the earlier paper [12] uses the same rule as ours.

The typing rule for assignments to arrays imposes that both the security level of the indexing expression and the security level of the array's length be bounded by the security level of guards. This might be surprising since, at first sight, assignments to arrays are neither a conditional nor a loop instruction.

However, by observing the operational semantics of assignments to arrays we notice that an assignment of the form `y[e]:==a` behaves exactly like the guarded command

if $\mathcal{E}_{\mathbb{A}}\ \mu$ `e` < $\mathcal{T}$`_size` μ `y` **then** `y[e]:==a` **else** **skip**

Thus, these conditions on the security levels of the indexing expression and of the length of an array correspond to the condition in the typing rule *Cond*, which imposes that the security level of guards must be an upper bound of the security level of the boolean expression. Intuitively, the security level of a boolean (or arithmetic) expression should be un upper bound of the security levels of its components. The rule for array assignment is concerned with a particular case of boolean expression.

4.3 Soundness

In this section we formally define and prove some properties of typable programs. The first property, *subject reduction*, states that types are preserved along execution.

lemma subject_reduction:
$\vdash$ P $\triangleright$ t s $\wedge$ (P, μ) $\longrightarrow_1$ (P', μ') $\Longrightarrow$ $\vdash$ P' $\triangleright$ t s

The proof is by rule induction on the typing judgment $\vdash$ P $\triangleright$ t s.

We define the relation *Q is a derivative of P*, written $P \rightsquigarrow Q$, as the least set closed under the following rules:

der_refl: P $\rightsquigarrow$ P
der_step: (P, μ) $\longrightarrow_1$ (P', μ') $\wedge$ P' $\rightsquigarrow$ Q $\Longrightarrow$ P $\rightsquigarrow$ Q

Using this definition we define *(semantically) high programs*. A program is semantically high if none of its derivatives can cause a low-visible change in any state.

constdefs SH :: $\mathbb{P}$ $\Rightarrow$ bool
SH P $\equiv$ $\forall$P' μ Q μ'. (P $\rightsquigarrow$ P' $\wedge$ (P', μ) $\longrightarrow_1$ (Q, μ')) $\Longrightarrow$ $\mu \simeq \mu'$

From this definition we prove the lemma

lemma SH_sr: (P,μ) $\longrightarrow_1$ (P',μ') $\wedge$ SH P $\Longrightarrow$SH P'

and from this the following lemma is straightforward.

lemma SH_sr_der: P $\rightsquigarrow$ P' $\wedge$ SH P $\Longrightarrow$SH P'

Let us mention that this definition of semantically high programs is a correction of the one given in [13]. The problem comes from the definition of derivative. In [13], Q is defined as a derivative of P, if for some μ and μ' we have (P, μ) $\longrightarrow^*$(Q, μ'), where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow_1$. This definition does not allow for arbitrary changes of the memory throughout the execution and thus the lemma `SH_sr_der` cannot be proved. However, using our definition of derivative, any program Q that can be obtained by reducing P via the operational semantics, allowing *arbitrary* changes

Skip

$$\frac{}{\vdash \textbf{skip} \triangleright \mathtt{t}\ \mathtt{s}}$$

Assign

$$\frac{\mathcal{L}_a\,\mathtt{a} \sqsubseteq \mathcal{L}_v\,\mathtt{x} \quad \mathtt{t} \sqsubseteq \mathcal{L}_v\,\mathtt{x}}{\vdash \mathtt{x{:=}{=}a} \triangleright \mathtt{t}\ \mathtt{s}}$$

Array

$$\frac{\mathcal{L}_t\,\mathtt{y} = (\tau_1, \tau_2) \quad \mathtt{t} \sqsubseteq \tau_1 \quad \mathcal{L}_a\,\mathtt{e} \sqsubseteq \tau_1 \quad \mathcal{L}_a\,\mathtt{a} \sqsubseteq \tau_1 \quad \mathcal{L}_a\,\mathtt{e} \sqsubseteq \mathtt{s} \quad \tau_2 \sqsubseteq \mathtt{s}}{\vdash \mathtt{y[e]{:=}{=}a} \triangleright \mathtt{t}\ \mathtt{s}}$$

Seq

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \vdash \mathtt{Q} \triangleright \mathtt{t'}\ \mathtt{s'} \quad \mathtt{s} \sqsubseteq \mathtt{t'} \quad \mathtt{t''} \sqsubseteq \mathtt{t} \quad \mathtt{t''} \sqsubseteq \mathtt{t'} \quad \mathtt{s} \sqsubseteq \mathtt{s''} \quad \mathtt{s'} \sqsubseteq \mathtt{s''}}{\vdash \mathtt{P}\ ;;\ \mathtt{Q} \triangleright \mathtt{t''}\ \mathtt{s''}}$$

Cond

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \vdash \mathtt{Q} \triangleright \mathtt{t}\ \mathtt{s} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{t} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{s'} \quad \mathtt{s} \sqsubseteq \mathtt{s'} \quad \mathtt{t'} \sqsubseteq \mathtt{t}}{\vdash \textbf{if}\ \mathtt{b}\ \textbf{then}\ \mathtt{P}\ \textbf{else}\ \mathtt{Q} \triangleright \mathtt{t'}\ \mathtt{s'}}$$

While

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{t} \quad \mathtt{s} \sqsubseteq \mathtt{t} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{s'} \quad \mathtt{s} \sqsubseteq \mathtt{s'} \quad \mathtt{t'} \sqsubseteq \mathtt{t}}{\vdash \textbf{while}\ \mathtt{b}\ \textbf{do}\ \mathtt{P} \triangleright \mathtt{t'}\ \mathtt{s'}}$$

Par

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \vdash \mathtt{Q} \triangleright \mathtt{t}\ \mathtt{s}}{\vdash \mathtt{P} \parallel \mathtt{Q} \triangleright \mathtt{t}\ \mathtt{s}}$$

When

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{t} \quad \mathcal{L}_b\,\mathtt{b} \sqsubseteq \mathtt{s'} \quad \mathtt{s} \sqsubseteq \mathtt{s'} \quad \mathtt{t'} \sqsubseteq \mathtt{t}}{\vdash \textbf{when}\ \mathtt{b}\ \textbf{do}\ \mathtt{P} \triangleright \mathtt{t'}\ \mathtt{s'}}$$

Control

$$\frac{\vdash \mathtt{P} \triangleright \mathtt{t}\ \mathtt{s} \quad \vdash \mathtt{Q} \triangleright \mathtt{t'}\ \mathtt{s'} \quad \mathtt{s} \sqsubseteq \mathtt{t'} \quad \mathtt{t''} \sqsubseteq \mathtt{t} \quad \mathtt{t''} \sqsubseteq \mathtt{t'} \quad \mathtt{s} \sqsubseteq \mathtt{s''} \quad \mathtt{s'} \sqsubseteq \mathtt{s''}}{\vdash \mathtt{P[\![Q]\!]} \triangleright \mathtt{t''}\ \mathtt{s''}}$$

Figure 2: Typing rules

in the memory throughout the reduction, is a derivative of P. This gives us the right definition of semantically high programs.

Matos et al. [29] give another corrected version of the definition which is equivalent to ours. They define semantically high programs coinductively as the greatest formula such that

```
SH P ⟹ (∀μ.  (P, μ) ⟶₁ (P', μ') ⟹ μ ≃ μ' ∧ SH P')
```

The following are several lemmas about semantically high programs which are needed in the soundness proof:

```
lemma SH_skip: SH skip
lemma SH_seq: SH P ∧ SH Q ⟹ SH (P;;Q)
lemma SH_par: SH P ∧ SH Q ⟹ SH (P ∥ Q)
lemma SH_when: SH P ⟹ SH (when b do P)
lemma SH_control: SH P ∧ SH Q ⟹ SH (P⟦Q⟧)
```

A program P is *bounded* if the infimum of the levels of variables assigned in it is $\mathbb{L}$.

```
constdefs bnd :: ℙ ⇒ bool
  bnd P ≡ ∀t s.  ⊢  P  ▷  t s ⟹ 𝕃 t
```

And we prove the following property of bounded programs:

```
lemma bnd_seq: bnd Q ⟹ bnd (P;;Q)
```

We also define the complementary notion of *nonbounded* which is often used:

```
constdefs
  notbnd :: ℙ ⇒ bool
  notbnd P ≡ ∃t s.  ⊢  P  ▷  t s ∧ ¬(𝕃 t)
```

Nonboundedness is subterm closed:

```
lemma nb_seq: notbnd(P;;Q) ⟹ notbnd Q ∧ notbnd P
lemma nb_if: notbnd( if  b then P else Q) ⟹ notbnd P ∧ notbnd Q
lemma nb_while: notbnd( while b do P) ⟹ notbnd P
lemma nb_par: notbnd(P ∥ Q) ⟹ notbnd P ∧ notbnd Q
lemma nb_when: notbnd(when b do P) ⟹ notbnd P
lemma nb_control: notbnd(P⟦T⟧) ⟹ notbnd P ∧ notbnd T
```

All these lemmas are automatically proven by the Isabelle *auto* tactic considering that the pool of rules used by *auto* has been previously fed with introduction rules and with elimination rules for case analysis on the operational semantics and on the typing rules.

A program which is notbounded cannot write on variables of low level, and therefore such a program is high.

```
lemma nb_SH : notbnd P ⟹SH P
```

However, a high program is not necessarily notbounded (see counterexample in [13]). We now define *guarded* programs. A program is guarded if the supremum of variables it reads is low.

```
constdefs guarded :: ℙ ⇒ bool
  guarded P ≡ ∃t s.  ⊢  P  ▷  t s ∧ (𝕃 s)
```

As a consequence of subject reduction, both nonboundedness and guardedness are preserved by execution.

```
lemma nb_sr: notbnd P ∧  (P, μ) ⟶₁ (Q, ν)  ⟹ notbnd Q
lemma guarded_sr: guarded P ∧  (P, μ) ⟶₁ (Q, ν)  ⟹ guarded Q
```

The next is an important lemma that characterizes the behaviour of guarded programs.

```
lemma behaviour_of_guarded_programs:
  guarded P ∧  μ ≃ ν ∧  (P, μ) ⟶₁ (P', μ')
  ⟹ ∃ν'. (P, ν) ⟶₁ (P', ν') ∧ μ' ≃ ν'
```

Proof. By structural induction on `P`. We examine the case of arrays which will help us to clarify the intuition behind the typing rule.

```
⊢  y[e]:==a  ▷  t s ∧ 𝕃 s ∧ μ ≃ ν ∧ (y[e]:==a, μ) ⟶₁ (P', μ')
⟹ ∃ν'. (y[e] :== a, ν) ⟶₁ (P', ν') ∧ μ' ≃ ν'
```

By case analysis on the transition `(y[e]:==a, μ)` $\longrightarrow_1$`(P', μ')` we obtain two cases. We detail the case where the transition is proved by means of *Array*1. For clarity, let μ=`(`μ_1`, `μ_2`)`, ν=`(`ν_1`, `ν_2`)` and $\mathcal{L}_t$ `y=(`τ_1`, `τ_2`)`. Then, we have:

```
𝕃 s ∧ μ ≃ ν ∧  ℰ𝔸 μ e < 𝒯_size μ₂ y ∧ t  ⊑  τ₁
∧ ℒₐ e  ⊑  τ₁ ∧ ℒₐ a  ⊑  τ₁ ∧ ℒₐ e  ⊑  s ∧ τ₂  ⊑  s
⟹ ∃ν'. (y[e]:==a, ν) ⟶₁ (skip, ν')
  ∧ 𝒯_update μ y (ℰ𝔸 μ e) (ℰ𝔸 μ a) ≃ ν'
```

We instantiate the quantified ν' in the conclusion by unifying with the *Array*1 rule. For this we have to prove that $\mathcal{E}_{\mathbb{A}}\,\nu$ `e < `$\mathcal{T}$`_size `ν_2 `y`. Notice that the typing rule of arrays requires the upper bound of guards be un upper bound of the security level of the indexing expression and of the array's length: $\mathcal{L}_a$`e` $\sqsubseteq$ `s` $\wedge$ τ_2 $\sqsubseteq$ `s`. From this and using axioms `Ldown` and `beh`$_{\mathbb{A}}$ we obtain $\mathcal{E}_{\mathbb{A}}\,\mu$ `e =` $\mathcal{E}_{\mathbb{A}}$ ν `e`, and by definition of equivalence of memories of arrays, we have $\mathcal{T}$`_size `μ_2 `y =` $\mathcal{T}$`_size `ν_2 `y`. The proofs of the other cases are fairly automatic using Isabelle. □

Following [13] we prove bisimilarity of high programs using the relation `S0`.

```
constdefs S0 :: (ℙ × ℙ) set
 S0 ≡ {(P, Q). SH P ∧ SH Q}

theorem S0_is_bisim: bisim (p2c S0)
```

The proof follows easily from lemma `SH_sr`.

In order to prove non-interference for our type system we define a relation called $\mathcal{S}$ between programs and prove that it, or rather `p2c` $\mathcal{S}$ is a bisimulation. Due to our generalization of the language, where we only have one syntactical level, a single relationship, that is $\mathcal{S}$, suffices (see section 4.4). The relation $\mathcal{S}$ is inductively defined by the rules of Figure 3.

$cl1$ $\dfrac{\mathtt{SHP}\quad \mathtt{SHQ}\quad \mathtt{typable\ P}\quad \mathtt{typable\ Q}}{\mathtt{P}\ \mathcal{S}\ \mathtt{Q}}$

$cl2$ $\dfrac{\mathtt{bnd\ P}\quad \mathtt{typable\ P}}{\mathtt{P}\ \mathcal{S}\ \mathtt{P}}$

$cl3$ $\dfrac{\mathtt{P}\ \mathcal{S}\ \mathtt{Q}\quad \mathtt{notbnd\ R}\quad \mathtt{typable\ (P;;R)}\quad \mathtt{typable(Q;;R)}}{\mathtt{(P;;R)}\ \mathcal{S}\ \mathtt{(Q;;R)}}$

$cl4$ $\dfrac{\mathtt{P1}\ \mathcal{S}\ \mathtt{P2}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{typable(P1}\parallel\mathtt{Q1)}\quad \mathtt{typable(P2}\parallel\mathtt{Q2)}}{\mathtt{(P1}\parallel\mathtt{Q1)}\ \mathcal{S}\ \mathtt{(P2}\parallel\mathtt{Q2)}}$

$cl4s$ $\dfrac{\mathtt{P1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{P2}\quad \mathtt{typable(P1}\parallel\mathtt{Q1)}\quad \mathtt{typable(P2}\parallel\mathtt{Q2)}}{\mathtt{(P1}\parallel\mathtt{Q1)}\ \mathcal{S}\ \mathtt{(P2}\parallel\mathtt{Q2)}}$

$cl5$ $\dfrac{\mathtt{SHP}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{typable\ Q1}\quad \mathtt{typable(P}\parallel\mathtt{Q2)}}{\mathtt{Q1}\ \mathcal{S}\ \mathtt{(P}\parallel\mathtt{Q2)}}$

$cl5s$ $\dfrac{\mathtt{SHP}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{typable\ Q1}\quad \mathtt{typable(P}\parallel\mathtt{Q2)}}{\mathtt{Q1}\ \mathcal{S}\ \mathtt{(Q2}\parallel\mathtt{P)}}$

$cl6$ $\dfrac{\mathtt{SHP}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{typable\ Q2}\quad \mathtt{typable(P}\parallel\mathtt{Q1)}}{\mathtt{(P}\parallel\mathtt{Q1)}\ \mathcal{S}\ \mathtt{Q2}}$

$cl6s$ $\dfrac{\mathtt{SHP}\quad \mathtt{Q1}\ \mathcal{S}\ \mathtt{Q2}\quad \mathtt{typable\ Q2}\quad \mathtt{typable(P}\parallel\mathtt{Q1)}}{\mathtt{(Q1}\parallel\mathtt{P)}\ \mathcal{S}\ \mathtt{Q2}}$

$cl7$ $\dfrac{\mathbb{L}\,(\mathcal{L}_b\mathtt{b})\ \mathtt{P}\ \mathcal{S}\ \mathtt{Q}\quad \mathtt{typable(}\mathbf{when\ b\ do\ P}\mathtt{)}\ \mathtt{typable(}\mathbf{when\ b\ do\ Q}\mathtt{)}}{\mathbf{(when\ b\ do\ P)}\ \mathcal{S}\ \mathbf{(when\ b\ do\ Q)}}$

$cl8$ $\dfrac{\mathtt{guarded\ P}\quad \mathtt{Q}\ \mathcal{S}\ \mathtt{R}\quad \mathtt{typable(P}[\![\mathtt{Q}]\!]\mathtt{)}\quad \mathtt{typable(P}[\![\mathtt{R}]\!]\mathtt{)}}{\mathtt{(P}[\![\mathtt{Q}]\!]\mathtt{)}\ \mathcal{S}\ \mathtt{(P}[\![\mathtt{R}]\!]\mathtt{)}}$

$cl9$ $\dfrac{\mathtt{SHQ}\quad \mathtt{SHR}\quad \mathtt{P}\ \mathcal{S}\ \mathtt{P'}\quad \mathtt{typable(P}[\![\mathtt{Q}]\!]\mathtt{)}\quad \mathtt{typable(P'}[\![\mathtt{R}]\!]\mathtt{)}}{\mathtt{(P}[\![\mathtt{Q}]\!]\mathtt{)}\ \mathcal{S}\ \mathtt{(P'}[\![\mathtt{R}]\!]\mathtt{)}}$

Figure 3: Relation $\mathcal{S}$

The original set of rules presented in [13] to prove noninterference for concurrent programs consisted of clauses *cl*1, *cl*2, *cl*3 and *cl*4. We first added *cl*4*s*, *cl*5, *cl*5*s*, *cl*6 and *cl*6*s* in order to prove the result for concurrent programs where the operational semantics included the rules *ParLN* and *ParRN*, providing for reduction of programs such as **skip** ∥ **skip**. And clauses *cl*7, *cl*8 and *cl*9 for the extension with `When` and `Control`.

We prove three useful properties of the relation $\mathcal{S}$. Using rule induction we prove that it is symmetric.

```
lemma S_sym: P S Q ⟹ Q S P
```

The second property

```
lemma S_refl: typable P ⟹ P S P
```

is proved by distinguishing the case where `P` is bounded, in which case we apply *cl*2, or not, in which case `SH P` by lemma `nb_SH` and thus `P S P` by clause *cl*1. The third property is proven using rule induction.

```
lemma S_skip_SH : skip S P ⟹SH P
```

Let us finally state the main result of this development saying that typable programs are secure in the sense of non-interference.

```
theorem Noninterference: typable P ⟹ secure P
```

Unfolding the definition of secure programs and instantiating with the relation $\mathcal{S}$ we obtain the following subgoals:

```
1. typable P ⟹ bisim (p2c S)
2. typable P ⟹ P S P
```

The second subgoal is solved using lemma `S_refl`. For the first subgoal we prove that the relation `p2c S`, the extended relation on $\mathcal{S}$ for configurations, is a bisimulation.

```
lemma S_is_bisim: bisim (p2c S)
```

The proof is by induction on the derivation of $\mathcal{S}$ by means of the following auxiliary lemma:

```
lemma S_is_bisim_aux: P S P' ⟹
  ∀μ ν Q μ'. μ ≃ ν ⟹ (P, μ) ⟶₁ (Q, μ')
  ⟹ (∃Q' ν'. (P', ν) ⟶₀,₁ (Q', ν') ∧ Q S Q' ∧ μ' ≃ ν')
```

The case of clause *cl*1 implies that both `SHP` and `SHQ` and the proof is trivial. For the rest of the proof we can assume that `P` and `Q` are not both high programs.

We examine the case for clause *cl*2.

```
lemma S_clause2:
 ∀μ ν Q μ'. bnd P ⟹ typable P ⟹ μ ≃ ν ⟹ (P, μ) ⟶₁ (Q, μ')
⟹ (∃Q' ν'. (P, ν) ⟶₀,₁ (Q', ν') ∧ Q S Q' ∧ μ' ≃ ν')
```

Proof. We proceed by induction on the structure of `P`. We present the proof of the most significant cases: assignment to arrays, sequential composition and

control.

(Assignment to arrays)

$$\texttt{bnd (y[e]:==a)} \wedge \texttt{ typable (y[e]:==a)} \wedge \ \mu \simeq \nu$$
$$\wedge \ \texttt{(y[e]:==a, } \mu\texttt{)} \longrightarrow_1 \texttt{(Q, } \mu'\texttt{)}$$
$$\Longrightarrow \exists \texttt{Q' } \nu' . \texttt{ (y[e]:==a, } \nu\texttt{)} \longrightarrow_{0,1} \texttt{(Q', } \nu'\texttt{)} \wedge \texttt{Q } \mathcal{S} \texttt{ Q'} \wedge \mu' \simeq \nu'$$

We distinguish two cases, according to which of the rules *Array*1 or *Array*2 is used to deduce the transition $\texttt{(y[e]:==a, } \mu\texttt{)} \longrightarrow_1 \texttt{(Q, } \mu'\texttt{)}$. Both cases are very similar so we only detail the case where the transition is proved by means of *Array*1. Let $\mu\texttt{=(}\mu_1\texttt{, } \mu_2\texttt{)}$ and $\mu'\texttt{=(}\mu_1'\texttt{, } \mu_2'\texttt{)}$, then we obtain

$$\texttt{bnd (y[e]:==a)} \wedge \texttt{ typable (y[e]:==a)}$$
$$\wedge \ \mu \simeq \nu \wedge \ \mathcal{E}_{\mathbb{A}} \ \mu \ \texttt{e < } \mathcal{T}\texttt{\_size } \mu \texttt{ y}$$
$$\Longrightarrow \exists \texttt{Q' } \nu' . \texttt{ (y[e]:==a, } \nu\texttt{)} \longrightarrow_{0,1} \texttt{(Q', } \nu'\texttt{)}$$
$$\wedge \ \textbf{skip } \mathcal{S} \texttt{ Q'} \wedge \mathcal{T}\texttt{\_update } \mu \texttt{ y } (\mathcal{E}_{\mathbb{A}} \ \mu \ \texttt{e)} \ (\mathcal{E}_{\mathbb{A}} \ \mu \ \texttt{a)} \simeq \nu'$$

There are two possibilities. Let $\mathcal{L}_t$ y be $\texttt{(}\tau_1\texttt{, } \tau_2\texttt{)}$, then

1. if $\mathbb{L}\,\tau_1$, we instantiate in the conclusion with the configuration $\texttt{(}\textbf{skip}\texttt{, } \nu\texttt{)}$ where $\nu\texttt{=}\mathcal{T}\texttt{\_update } \nu \texttt{ y } (\mathcal{E}_{\mathbb{A}} \ \nu \ \texttt{e)} \ (\mathcal{E}_{\mathbb{A}} \ \nu \ \texttt{a)}$ so that we can resolve the first conjunct with the rule *Array*1. To apply this rule we need to prove that the condition $\mathcal{E}_{\mathbb{A}} \ \nu \ \ \texttt{e < } \mathcal{T}\texttt{\_size } \nu \texttt{ y}$ holds. From `typable P` and case analysis on the typing rules we obtain $\mathcal{L}_a\,\texttt{e} \sqsubseteq \tau_1$. Since $\mathbb{L}\,\tau_1$ we have $\mathcal{E}_{\mathbb{A}} \ \mu \ \ \texttt{e = } \mathcal{E}_{\mathbb{A}} \ \nu \ \texttt{e}$ using axioms `beh_`$\mathbb{A}$ and `Ldown`. By the definition of $\simeq_t$ we obtain $\mathbb{L}\,\tau_2 \Longrightarrow \mathcal{T}\texttt{\_size } \mu \texttt{ y = } \mathcal{T}\texttt{\_size } \nu \texttt{ y}$. Using the axioms `Ldown` and `level_size_leq_contents`, we obtain $\mathbb{L}\,\tau_2$. That $\textbf{skip}\,\mathcal{S}\,\textbf{skip}$ is proved by *cl*1 and the proof of the equivalence of memories is easy.

2. If $\neg\ \mathbb{L}\,\tau_1$, we instantiate in the conclusion with the program `y[e]:==a` and the memory ν, so that the first conjunct is proven by reflexivity of $\longrightarrow_{0,1}$. We prove $\textbf{skip}\,\mathcal{S}$ `y[e]:==a` by clause *cl*1 since `SH` **skip**. From $\neg\ \mathbb{L}\,\tau_1$ we prove `SH(y[e]:==a)`

(Seq) We assume `IH P` and `IH Q` being the induction hypothesis for `P` and `Q`, respectively. Then we have

$$\texttt{bnd(P;;Q)} \wedge \texttt{ typable(P;;Q)} \wedge \mu \simeq \nu \wedge \texttt{ (P;;Q, } \mu\texttt{)} \longrightarrow_1 \texttt{(R, } \mu'\texttt{)}$$
$$\Longrightarrow \exists \texttt{Q' } \nu' . \texttt{ (P;;Q, } \nu\texttt{)} \longrightarrow_{0,1} \texttt{(Q', } \nu'\texttt{)} \wedge \texttt{R } \mathcal{S} \texttt{ Q'} \wedge \mu' \simeq \nu'$$

By rule inversion on the transition $\texttt{(P;;Q, } \mu\texttt{)} \longrightarrow_1 \texttt{(R, } \mu'\texttt{)}$ we obtain two cases:

1. If the transition is solved by *Seq1* we have `P=skip`. The required transition is $\texttt{(}\textbf{skip}\texttt{;;Q, } \nu\texttt{)} \longrightarrow_{0,1} \texttt{(Q, } \nu\texttt{)}$ solved by *Seq1*; `Q` $\mathcal{S}$ `Q` is resolved by $\mathcal{S}$`_refl` since trivially `typable(`**skip**`;;Q)` implies `typable Q` and $\mu \simeq \nu$ is among the assumptions.

2. If the transition is solved by *Seq2* we have

$$\texttt{bnd(P;;Q)} \wedge \texttt{ typable(P;;Q)} \wedge \mu \simeq \nu \wedge \texttt{ (P, } \mu\texttt{)} \longrightarrow_1 \texttt{(P', } \mu'\texttt{)}$$
$$\Longrightarrow \exists \texttt{Q' } \nu' . \texttt{ (P;;Q, } \nu\texttt{)} \longrightarrow_{0,1} \texttt{(Q', } \nu'\texttt{)} \wedge \texttt{(P';;Q) } \mathcal{S} \texttt{ Q'} \wedge \mu' \simeq \nu'$$

There are two possibilities:

(a) First we consider `bnd Q`. Since `P;;Q` is typable we have $\vdash$ `P` $\triangleright$ `t s`, $\vdash$ `Q` $\triangleright$ `t' s'` with `s` $\sqsubseteq$ `t'`. If `Q` is bounded then $\mathbb{L}$ `t'` and by `Ldown` we have $\mathbb{L}$ `s`. Thus `P` is guarded. By the lemma about the behaviour of guarded programs we have `(P, ν)` $\longrightarrow_1$`(P', ν')` with $\mu' \simeq \nu'$ for some ν'. The required transition in the conclusion goes to `(P';;Q, ν')` and we prove `(P';;Q)` $\mathcal{S}$`(P';;Q)` by *cl*2 and the following lemma

lemma `bnd_seq: bnd Q` $\Longrightarrow$ `bnd (P;;Q)`

which is a trivial consequence of the type system.

(b) If `notbnd Q` then we have `SH Q`. Since we have $\neg$ `SH(P;;Q)` (otherwise we fall into clause 1) we obtain $\neg$ `SH P` using lemma `SH_seq` and thus `bnd P` by `notbnd_SH` again. Using the induction hypothesis for `P` we obtain `(P, ν)` $\longrightarrow_{0,1}$`(Q', ν')` $\wedge$ `P'` $\mathcal{S}$ `Q'` $\wedge$ $\mu' \simeq \nu'$. The result follows using the rule *Seq2* and clause *cl*3.

(Control) We assume `IH P` and `IH Q` being the induction hypothesis for `P` and `Q`, respectively. Then we have

`bnd(P⟦Q⟧)` $\wedge$ `typable(P⟦Q⟧)` $\wedge$ $\mu \simeq \nu$ $\wedge$ `(P⟦Q⟧, μ)` $\longrightarrow_1$ `(R, μ')`
$\Longrightarrow$ $\exists$`Q' ν'. (P⟦Q⟧, ν)` $\longrightarrow_{0,1}$ `(Q', ν')` $\wedge$ `R` $\mathcal{S}$ `Q'` $\wedge$ $\mu' \simeq \nu'$

By case analysis on the transition `(P⟦Q⟧, μ)` $\longrightarrow_1$`(R, μ')` we distinguish three cases. We just show the proof for the case where the transition is proved by means of *Control*1. The cases for *Control*2 and *ControlSkip* present no further difficulties.

We have then `(P, μ)` $\longrightarrow_1$`(P', μ`$_a$`)` and `(Q, μ)` $\longrightarrow_1$`(Q', μ'')` among the premises. The conclusion becomes

$\exists$`R' ν'. (P⟦Q⟧, μ')` $\longrightarrow_{0,1}$ `(R', ν')` $\wedge$ `P'⟦Q'⟧` $\mathcal{S}$ `R'` $\wedge$ $\mu_a \sqcup_\mu \mu'' \simeq \nu'$

We distinguish two cases:

1. If `bnd Q` holds then we can prove `guarded P` as in the case for sequential composition. Thus by lemma `behaviour_of_guarded_programs` we obtain for some ν' that `(P, ν)` $\longrightarrow_1$`(P', ν')` and $\mu_a \simeq \nu'$. By the induction hypothesis we obtain `(Q, ν)` $\longrightarrow_{0,1}$`(R', ν'')` and `Q'` $\mathcal{S}$ `R'` $\wedge$ $\mu'' \simeq \nu''$. We distinguish whether `(Q, ν)` $\longrightarrow_{0,1}$`(Q, ν)` or `(Q, ν)` $\longrightarrow_1$`(R', ν'')`. In the first case we solve the conclusion by using rule *Control*2. The subgoal `P'⟦Q'⟧`$\mathcal{S}$ `P'⟦Q⟧` is proved by clause *cl*8. To prove $\mu_a \sqcup_\mu \mu'' \simeq \nu'$ we use symmetry and transitivity of $\simeq$ combined with the assumption $\mu_a \simeq \nu'$ and obtain $\mu_a \sqcup_\mu \mu'' \simeq \mu_a$. This is true if the lengths of arrays are preserved along the execution, which is easily proven by induction on the operational semantics.

   The second case is solved with rule *Control*1 and then using again *cl*8.

2. If we have notbnd Q then SH Q and since $\neg$ SH(P⟦Q⟧) (otherwise we fall into clause 1) we obtain $\neg$ SH P using lemma SH_control and thus bnd P. Then, by induction hypothesis we have for some R′ that

   $$(P', \nu) \longrightarrow_{0,1} (R', \nu') \wedge P' \ \mathcal{S} \ R' \wedge \mu_a \simeq \nu'$$

   By case analysis on $(P', \nu) \longrightarrow_{0,1} (R', \nu')$ we obtain two cases. The reflexive case is solved by reflexivity in the transition of the conclusion. The subgoal $P'⟦Q'⟧\mathcal{S}\ P⟦Q⟧$ can be proved with clause *cl*9. The second case is solved with rule *Control*2 and then using *cl*9.

The proof of the other clauses are relatively easy and fairly automatic with Isabelle. □

4.4 Layered view of thread systems

In the case of the original layered language of [13], one can give a modular proof using the 'locales' mechanism of Isabelle. In [13] thread systems are viewed as list of threads controlled by a parallel scheduler, each thread being a guarded sequential program. We have defined a locale for sequential programs and another one for parallel programs. These two locales have as parameters:

- a set of programs (defined as a type parameter), a partial order of security level (defined as an abstract partial order), an abstract class of memory (defined as a type parameter) with an equivalence relation as defined in Section 2.2, an operational semantics (depending on the memory, and defined as a parameter from configurations to configurations to booleans), and a type system (defined as a parameter of programs to a pair of security levels to booleans), from which the type system and operational semantics of thread systems can be defined;
- axioms that are sufficient to guarantee non-interference for typable thread systems.
  - For sequential programs we have identified seven axioms. The first three of them concern the typing system and the operational semantics. The remaining four assumptions are properties about a relation $\mathcal{S}_1$ between sequential programs, which is also a parameter of the locale.
    1. $\vdash$ P $\triangleright$ t s $\wedge$ $(P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow \vdash$ P′ $\triangleright$ t s
    2. notbnd P $\wedge (P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow \mu \simeq \mu'$
    3. $\neg(\exists$c. $(P, \mu) \longrightarrow_1$ c$) \Longrightarrow \neg(\exists$c. $(P, \nu) \longrightarrow_1$ c$)$
    4. P $\mathcal{S}_1$ Q $\Longrightarrow$ Q $\mathcal{S}_1$ P
    5. SHP $\wedge$ SHQ $\wedge$ typable P $\wedge$ typable Q $\Longrightarrow$ P $\mathcal{S}_1$ Q
    6. bound P $\wedge$ typable P $\Longrightarrow$ P $\mathcal{S}_1$ P
    7. The last axiom states that the relation $\mathcal{S}_1$ (more concretely, its extended relation $\mathcal{R}_1$ to configurations) is a particular form of bisimulation called quasi-strong bisimulation:

$$\mathsf{P}\ \mathcal{S}_1\ \mathsf{P'} \Longrightarrow \forall \mu\, \mu'\, \mathsf{Q}\, \nu.\ \mu \simeq \mu' \Longrightarrow (\mathsf{P},\ \mu) \longrightarrow_1 (\mathsf{Q},\ \nu) \Longrightarrow$$
$$(\exists\ \mathsf{Q'}\ \nu'.\ (\mathsf{P'},\ \mu') \longrightarrow_1 (\mathsf{Q'},\ \nu') \wedge \mathsf{Q}\ \mathcal{S}_1\ \mathsf{Q'} \wedge \nu \simeq \nu')$$
$$\vee\ (\mathsf{SH\,P} \wedge \mathsf{SH\,P'})$$

– For parallel programs we need three axioms. The first two axioms are analogous to the first two axioms of sequential programs, and the third one is the property about behaviour of guarded programs stated already in Section 4.3.

Thus, we can instantiate the result to other settings where sequential and concurrent programs enjoy the properties upon which we build. We have carried out the proofs with one particular instantiation. We have instantiated the memory as in the previous section, sequential programs as the sequential subset of the language of Section 3 and, parallel programs as the sequential sublanguage plus the parallel construct. This corresponds to the language considered in [13] but with a sequential sublanguage that includes assignments to arrays.

It should be noted that the type system for the layered language is different from the one presented in Section 4, and that the proof is not a mere reorganization of the proof of type soundness presented earlier. Indeed, in the layered setting two stronger notions of bisimulation are used, namely *quasi-strong* and *strong* bisimulations. The first one is needed to prove non-interference for the sequential sublanguage. The second one is used to prove non-interference for controlled thread systems.

The definition of a quasi-strong bisimulation is the following:

constdefs

$$\mathsf{quasi\_strong\_bisim} :: (\mathbb{C}_s \Rightarrow \mathbb{C}_s \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$$
$$\mathsf{quasi\_strong\_bisim\ R} \equiv \forall \mathsf{P}\ \mu\ \mathsf{Q}\ \nu.$$
$$\mathsf{R}\ (\mathsf{P},\ \mu)\ (\mathsf{Q},\ \nu) \Longrightarrow \mathsf{R}\ (\mathsf{Q},\ \nu)\ (\mathsf{P},\ \mu) \wedge \mu \simeq \nu \wedge$$
$$((\forall \mathsf{P'}\ \mu'.\ (\mathsf{P},\ \mu) \longrightarrow_1 (\mathsf{P'},\ \mu')$$
$$\Longrightarrow (\exists \mathsf{Q'}\ \nu'.\ (\mathsf{Q},\ \nu) \longrightarrow_1 (\mathsf{Q'},\ \nu') \wedge \mathsf{R}\ (\mathsf{P'},\ \mu')\ (\mathsf{Q'},\ \nu')))$$
$$\vee\ (\mathsf{SH\,P} \wedge \mathsf{SH\,Q}))$$

where $\mathbb{C}_s$ is the type of configurations for sequential programs. A relationship between pure sequential programs, consisting of the first three clauses of the relation $\mathcal{S}$ of Figure 3, is proved to be a quasi-strong bisimulation. This stronger result is needed to establish that a third relationship between thread systems is a strong bisimulation, where a strong bisimulation is defined like a bisimulation but replacing $\longrightarrow_{0,1}$ by $\longrightarrow_1$.

constdefs

$$\mathsf{strong\_bisim} :: (\mathbb{C}_c \Rightarrow \mathbb{C}_c \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$$
$$\mathsf{strong\_bisim\ R} \equiv \forall \mathsf{P}\ \mu\ \mathsf{Q}\ \nu.$$
$$\mathsf{R}\ (\mathsf{P},\ \mu)\ (\mathsf{Q},\ \nu) \Longrightarrow \mathsf{R}\ (\mathsf{Q},\ \nu)\ (\mathsf{P},\ \mu) \wedge \mu \simeq \nu \wedge$$
$$(\forall \mathsf{P'}\ \mu'.\ (\mathsf{P},\ \mu) \longrightarrow_1 (\mathsf{P'},\ \mu')$$
$$\Longrightarrow (\exists \mathsf{Q'}\ \nu'.\ (\mathsf{Q},\ \nu) \longrightarrow_1 (\mathsf{Q'},\ \nu') \wedge \mathsf{R}\ (\mathsf{P'},\ \mu')\ (\mathsf{Q'},\ \nu')))$$

where $\mathbb{C}_c$ is the type of configurations for controlled thread systems.

It is easy to prove that a strong bisimulation is also a quasi-strong bisimulation:

lemma sbisim_is_qsbisim: strong_bisim R $\Longrightarrow$ quasi_strong_bisim R

However, that a quasi-strong bisimulation is also a bisimulation is not true in general. It holds for any relation R satisfying the following condition:

$\forall$P Q μ ν. SH P $\Longrightarrow$SH Q $\Longrightarrow$ $\mu \simeq \nu \Longrightarrow$ R (P, μ) (Q, ν)

In order to prove the final result (i.e. typable controlled thread systems are secure) we exhibit a strong bisimulation R such that $R\ P\ P$ for all typable controlled thread systems P.

We proceed as follows: first, we define a strong bisimulation S_3 on controlled thread systems. Since this relation does not have the expected property, one defines a quasi-strong bisimulation S_4 that extends S_3 and enjoys the property that $P\ S_4\ P$ for all typable controlled thread systems P. Finally, we show that S_4 is a strong bisimulation, by proving that S_4 satisfies the sufficient condition for a quasi-strong bisimulation to be a strong bisimulation shown above. The definitions of these relations and further details about the proofs can be found in [13, 10, 35].

5 Discussion

The purpose of this section is to provide a high-level perspective on our formalizations, as well as some statistics on the Isabelle theories.

5.1 Benefits of machine-checked formalization

Machine-checked formalizations are particularly useful to rapidly detect potential difficulties in informal arguments: for example, the use of a proof assistant was useful in the early stages of our project to unveil minor flaws in the definitions of [13]. Although not often emphasized, proof assistants are equally useful when playing with variations of a result that was already proved formally. Indeed, it is possible to test the "type system" by replaying the proof script with a slightly modified semantics or type system can provide very convenient feedback. We fully took advantage of this "testing" facility offered by proof assistants to extend the results of Boudol and Castellani to our setting.

Then, machine-checked formalizations can be turned into executable prototypes at little cost. In the next section, we show how our formalization of the type system can be used to type-check programs. It is also possible to use our formalization of the operational semantics of the language to execute programs; however, our formalization is hiding implementation details on purpose, and it is necessary to provide implementations of the memory model in order to obtain an executable semantics. The benefits of executable semantics are obvious in the context of complex programming languages whose formalized semantics may contain bugs (a point much emphasized in formalizations of Java Virtual Machines, see e.g. [9]), but also to show that the abstract treatment of the memory model (or any other notion which has been formalized abstractly, e.g. the heap in the formalization of the Java Virtual Machine) is realistic.

5.2 Difficulties with machine-checked formalizations

The overall experience with the formalization is rather positive, but of course we encountered some well-known difficulties with carrying the formal developments.

The first difficulty is the limited possibilities for proof reuse. Formal proofs for programming languages are often carried incrementally, by considering increasingly large fragments of the language, or even better, by treating distinct fragments of the language in isolation. Unfortunately, such an incremental approach is not always possible, due to the possibility of undesirable interactions between language features.

The situation is even worse in the context of machine-checked verification, as there is no general mechanism to extend a proof of a property ϕ from one fragment L of the language (described as a datatype with a given set of constructors) to a larger fragment L' (described as a datatype with strictly more constructors). Indeed,the current approach for proving ϕ for L' is simply to run again the script proving that ϕ for L, and complete it in appropriate places. Of course, one drawback of this approach is that it makes proof scripts harder to maintain, because changes have to be reproduced in several places. Thus an important research topic in the field of proof assistants is to develop methods for supporting proof reuse.

This said, enforcing non-interference is notoriously more difficult in presence of concurrency or scheduling primitives, and it is unclear that one could provide a criterion on information flow type systems for sequential languages so that their extension to a multi-threaded language *with the parallel composition allowed arbitrarily deep in programs* enforces non-interference (which would be a modularity result similar to the one which we achieved for the layered language). Likewise, scheduling introduces so-called refinement attacks (i.e. a non-interfering multi-threaded program might become insecure once a scheduling policy is added to it), so that, as in the previous case, it is unclear that one can provide a criterion on multi-threaded programs so that their extension to a language with scheduling *with scheduling allowed arbitrarily deep in programs* enforces non-interference.

The second difficulty is the lack of automation. One initial objective of our work was to assess the extent to which proofs could be made automatic through appropriate tactics. We feel that the level of automation is acceptable for simple properties such as subject reduction, but more work is required for proving statements that involve existential statements; e.g. in the proof that $\mathcal{S}$ is a bisimulation, the user has to provide for each reduction step of the first element in the bisimulation relation the matching step of the second element in the relation. We believe that it is feasible, and useful, to develop heuristics that help finding the matching move and showing its correctness in many cases.

5.3 Statistics

The theories and proof scripts (for the development presented in Sections 3, 4 and the example in 6) consist of approximately 2,500 lines of Isabelle, and are

organized as follows:

- about 400 lines are concerned with the formal definition of the language, syntax and operational semantics, and of the type system;
- about 100 lines with basic facts such as subject reduction or the subterm closedness of typable terms;
- about 600 lines are concerned with properties of bound programs and semantically high programs;
- about 1,000 lines are concerned with the formal definition of bisimulations, the definition of $\mathcal{S}$ and the proof that $\mathcal{S}$ is a bisimulation.
- about 400 lines are concerned with the example.

The de Bruijn factor (i.e. the ratio of the length of formal definitions and proofs by the length of their informal counterparts) is of the order of 5, which seems very reasonable. Nevertheless, we believe that it is possible to achieve a greater degree of automation, and therefore a better de Bruijn factor, by developing dedicated tools for proving type soundness and for establishing that a relation is indeed a bisimulation. We believe that increasing proof automation is an essential ingredient for encouraging machine-checked formalizations of programming language theories.

It is difficult to estimate the effort required to complete the formalization, since a substantial part of the effort consisted in getting acquainted to the formalisms and results of [13]. Our rough estimate is that it took us about 6 months to complete the formalization.

The full theories and proof scripts are available at:

`http://www.loria.fr/~prensa`

and can be displayed with a notation similar to the one used in this paper using ProofGeneral with the X-symbol option[1].

6 Example

We illustrate the use of our type system by proving type correctness of a program that uses parallelism, arrays and a scheduler.

6.1 An implementation of the memory model

In order to obtain an executable semantics we define a concrete model for the memory by implementing the lookup and update functions. We use locales to structure the different levels of abstraction of the memory. Locales allow us to prove via the interpretation command that the concrete model is a refinement of the abstract one.

[1] See `http://proofgeneral.inf.ed.ac.uk/`.

The locales facility is quite recent in Isabelle and is still under development. Some features such as sharing of parameters have to be simulated by using a predeclared locale called *var* which simply fixes one parameter. Also, inductive definitions are not yet allowed in a locale declaration and have to be defined outside the locale and imported afterwards. As a consequence, the presentation with locales is unnecessarily complicated at present and will be improved in the near future. Hence, we give here a readable description of the refinement. The actual Isabelle formalization using locales can be found in [35].

The types of variables and array variables are left abstract as in Section 3. Memories are now defined as pairs of functions that map variables to values, in this case natural numbers, and array variables to arrays, which are modeled as list of values.

```
types 𝓜ᵥ = 𝒱 ⇒ nat
types 𝓜ₜ = 𝒯 ⇒ nat list
types 𝓜 = 𝓜ᵥ × 𝓜ₜ
```

Expressions are defined as functions that take as input a memory and return a natural number (for arithmetic expressions) or a boolean (for boolean expressions).

```
types 𝔸 = 𝓜 ⇒ nat
types 𝔹 = 𝓜 ⇒ bool
```

Hence, evaluation of arithmetic and boolean expressions (formerly $\mathcal{E}_{\mathbb{A}}$ and $\mathcal{E}_{\mathbb{B}}$) becomes function evaluation. Such a modeling of expressions as functions is known in the literature as a shallow embedding because it does not use an inductive definition of expressions, in contrast to a deep embedding, which does. In a shallow embedding, the constant `n` is modeled as the function `λ μ. n`, whereas compound expressions such as `e+e'` are modeled as the function `λ μ. (E μ) + (E' μ)`, where `E` and `E'` respectively denote the encoding of `e` and `e'`. The primary motivation for adopting a shallow embedding is the abstract treatment of expressions found in most papers on information flow type systems.

The definition of the lookup and update functions are:

```
𝒱_lookup μ x ≡ (fst μ) x
𝒱_update μ x ≡ (λa. ((fst μ)[x ↦ a], snd μ))
𝒯_lookup μ y ≡ (λe. ((snd μ) y)!e)
𝒯_update μ y ≡ (λe a. (fst μ, (snd μ)[y[e] ↦ a]))
```

and rely on the following auxiliary definitions:

- `fst` and `snd` are the projection functions that come with the product type $\alpha \times \beta$ used to define the memory.
- update of simple variables uses the predefined update function `μ[x ↦a]`, where `μ: 𝒱⇒nat`, defined as `λ z. if z=x then a else μ z`;
- lookup of the ith component of a list xs is written $xs!i$ in Isabelle lists notation.

- array update uses the function μ`{y[e] ↦a}`, where μ`: `$\mathcal{T}$`⇒nat list`, defined as

  λ `z. if z=y then y[e:=a] else` μ `z`

  where `y[e:=a]` is the list obtained from `y` by replacing the `e`th element of `y` by `a`.

The definition of $\mathcal{T}$`_size` is simply the length of the list:

$\mathcal{T}$`_size` μ `y` $\equiv$ `length ((snd` μ`) y)`

With these definitions we can easily prove the axioms about the lookup, update and size functions stated in the previous sections. It remains to give an implementation of the `merge_mem` function which satisfies the the four axioms stated in Sections 3 and 4. It is defined as follows (we overload the symbol $\sqcup$):

```
(* merging values *)
v₁ ⊔ᵥ v₂ = if v≠v₁ and v=v₂ then v₁
                 else if v≠v₂ and v=v₁ then v₂ else v

(* merging simple variable memories *)
μ'₁ ⊔μ' μ'₂ = λy. (μ'₁ y) ⊔μ'y (μ'₂ y)

(* merging array memories *)
μ''₁ ⊔μ'' μ''₂ = λy. map (λ (v, v₁, v₂). v₁ ⊔ᵥ v₂)
              (zip (μ'' y) (zip (μ''₁ y) (μ''₂ y)))

(* merging memories *)
 μ₁ ⊔μ μ₂ = let μ = (μ', μ''), μ₁ = (μ'₁, μ''₁), μ₂ = (μ'₂, μ''₂)
             in (μ'₁ ⊔μ' μ'₂ , μ''₁ ⊔μ'' μ''₂)
```

The functional *map* :: $(\alpha \Rightarrow \beta) \Rightarrow \alpha$ *list* $\Rightarrow \beta$ *list* applies a function to all elements of a list. The function *zip* :: α *list* $\Rightarrow \beta$ *list* $\Rightarrow (\alpha \times \beta)$ *list* returns the list of pairs of elements formed from the elements of two given lists. The length of the resulting list is the minimum of the input list's lengths.

The model of the memory as described in Section 3.2 has been encapsulated in a locale called `loc2_`$\mathcal{M}$ having as parameters the equivalence relation between memories ($\simeq$) defined in Section 4.1, the merging function `merge_mem`, the lookup and update functions for both simple and array variables and the function $\mathcal{T}$`_size`. All the properties about these functions needed for the proof of non-interference are stated as assumptions of the locale.

The refined model as explained in this section is encapsulated in a locale called `loc3_`$\mathcal{M}$. In the context of `loc3_`$\mathcal{M}$ we can prove the validity of the assumptions in `loc2_`$\mathcal{M}$ for the given instantiations. Proving that `loc3_`$\mathcal{M}$ is a refinement of `loc2_`$\mathcal{M}$ is done in Isabelle via the following statement:

`interpretation` `loc3_`$\mathcal{M}$ $\subseteq$ `loc2_`$\mathcal{M}$

This command interprets `loc2_`$\mathcal{M}$ in the locale `loc3_`$\mathcal{M}$. It requires a proof that the specification of `loc3_`$\mathcal{M}$ implies the specification of `loc2_`$\mathcal{M}$.

6.2 A tax calculation program

We illustrate the use of our type system by proving type correctness of a simple example that combines the use of parallelism, arrays and a scheduler. We reuse the example in [15] about a tax calculation program, where given an array of taxable incomes, the program outputs the corresponding array of income taxes. To place it in a concurrent setting, let us suppose that the array of taxable incomes is too big to be processed directly by the tax calculation program. Hence, we adopt a producer/consumer solution. The idea is to use two processes, producer and consumer, that share a common, bounded buffer. The producer puts information into the buffer and the consumer takes it out. In our present example the producer writes the values from the array of taxable incomes into the buffer, and the consumer, which corresponds to our tax calculation program, removes the values from the buffer and calculates their income taxes. Trouble may arise when the producer attempts to put a new item in a full buffer or the consumer tries to remove an item from an empty buffer. The solution used here is taken from [3]. Besides, a scheduler manages the execution times of the producer and the consumer.

We define two levels `L` and `H` with `L` being lower than `H`.

```
consts L :: level
       H :: level
axioms L_sub_H : L ⊑ H
```

We consider a simplified version of the tax calculation program where we have as inputs:

1. An array called `brackets` of the limits separating income brackets, with security type `(L, L)`,
2. an array `TaxTable` with the taxable incomes for each income bracket, with security type `(L, L)` and
3. the array of taxable incomes, `taxableIncome` with security type `(H, L)`, to be processed.

The goal is to fill in an array `incomeTax` with the tax owed for each tax return. The typing for this output array is `(H, L)`.

Given these tables, we calculate the income tax for taxable income t by using binary search to find an index b such that `brackets[b]` $\leq$`t < brackets[b+1]` and then return `TaxTable[b]`.

We define each variable as a constant in Isabelle and state its type by an axiom. For example, for `taxableIncome` we have:

```
consts  taxableIncome :: 𝒯
axioms  l_taxableIncome : getlevel_array taxableIncome = (H, L)
```

The definition of the main program is the following:

```
constdefs Parallel_TaxCalcul :: ℙ
  Parallel_TaxCalcul ≡ INIT;;
```

```
Sched ⟦ (when (λ(μ1, μ2). (μ2 s)!0=1) do Producer)  ‖
        (when (λ(μ1, μ2). (μ2 s)!1=1) do Consumer) ⟧
```

where the Isabelle notation $xs!i$ indicates the ith component of a list xs. The first part `INIT` initializes all auxiliary variables to the null function. The variable `s` is an array with two components, one for the `Producer` and one for the `Consumer`, each component can take the values 0 or 1, which blocks or enables their execution, respectively. They are initialized to zero by `INIT`, and are alternatively set to 1 by a scheduler that implements round robin with a time slice of `length buffer`.

As shown in the boolean conditions of the **when**-statements above, the encoding of boolean and arithmetic expressions uses lambda terms. For clarity, we hide in the following the lambda abstractions over the memory and simply write the names of locations or of arrays.

The definition of the scheduler is:

```
%constdefs Sched :: ℙ
 Sched ≡ n:== 0;;
             while True do
                  (k:== 0;;
                   while  k < length buffer  do
                         (s[n + 1]:== 1;;
                          s[n + 1]:== 0;;
                          k:== k + 1) ;;
                    n:== (n + 1) mod 2)
```

The `Producer` and `Consumer` are defined as follows:

```
 Producer ≡
 while i < length taxableIncome do
  x:== taxableIncome!i;;
  when ins - outs < length  buffer do
      (buffer[ins mod (length buffer)]:== x;;
      ins:== ins + 1) ;;
  i:== i + 1

 Consumer ≡
  while j < length taxableIncome do
     Remove;;
     BinarySearch;;
     incomeTax[j]:== TaxTable!lo
     j:== j + 1
```

`Remove` is the part of the program where the consumer removes its input from the buffer:

```
 Remove ≡
   when outs < ins do
     y:== buffer!(outs mod (length buffer));;
     outs:== outs + 1
```

and the `BinarySearch` part corresponds to the calculation in `lo` of the corresponding income bracket using binary search.

```
BinarySearch ≡
lo :== 0 ;; hi :== length brackets;;
while lo + 1 < hi do
  (mid:== (lo + hi) div 2;;
    if  y < brackets!mid
    then hi:== mid
     else lo:== mid)
```

To see whether the program is accepted by our type system we must find acceptable types for the auxiliary variables. Since the contents of `TaxableIncome` are high, so must be the variable `x` and also the contents of the auxiliary array `buffer`, whose type is thus `(H, L)`. By the assignment in `Remove` we deduce that `y` is also of level `H`. Thus, the `if` command within `BinarySearch` has an `H` guard, which means that the branches must not assign to `L` variables. So, `hi` and `low` must be of level `H`. From the assignment to `mid`, we deduce that `mid` must also be a high level variable. In `Consumer`, following the high-guarded conditional of `BinarySearch` there are two assignments, one to the array `incomeTax`, whose contents are high and then the update of the variable `j` which must then be also high. Since `j` is high the guard of the `Consumer`'s **while** is also high, that means that the body can only assign to high level variables. Thus, `outs` is also high. Since `outs` is high so must be the guard of the **when**-statement inside `Producer` implying that variables `ins` and `i` must also be high. Variables `k` and `n` have security type `L` and the array `s` has type `(L, L)`.

With these typings it is straightforward to verify that the program is well typed under our type system.

```
lemma Parallel_TaxCalcul_typable: ⊢ Parallel_TaxCalcul ▷ L H
```

Having Isabelle as an assistant automates the type checking and helps to identify the right types easily. We have stated the security types of arithmetic and boolean expressions as axioms. For an extensive use of Isabelle to check type correctness it would be more reliable to formalize expressions using a deep embedding and prove correctness of the corresponding type system.

7 Conclusion

We have presented the first machine-checked proof of non-interference for a concurrent language inspired from [13], and featuring primitives for scheduling. Our work demonstrates that it is possible to use proof assistants in the design or the verification of advanced type systems for programming languages, and is thus in line with the philosophy of the POPLMark Challenge [4], which aims at generalizing the use of theorem provers for checking the meta-theory of programming languages.

7.1 Related work

7.1.1 Information flow type systems for concurrent languages

Providing appropriate notions of non-interference and associated static enforcement mechanisms for concurrent programs is extremely challenging, and there have been many attempts to provide a satisfactory solution to guaranteeing confidentiality of concurrent programs. Some relevant works are discussed below; it should be noted that we have no claim to exhaustiveness and refer the reader to [40] for a detailed account of the field up to 2003.

Non-interference in presence of concurrency and distribution has been investigated thoroughly; in particular, many works have conducted an analysis and comparison of the possible notions of non-interference for non-deterministic systems, notably in the context of process calculi, see for example [16]. In an attempt to bridge the gap between (sequential) programming languages and process calculi, Focardi, Rossi and Sabelfeld [17] establish a relation between security conditions for programming languages and security conditions for process calculi. A similar endeavor was conducted earlier by Mantel and Sabelfeld [26], who provide a link between security conditions for programming languages and security conditions for event systems.

Smith and Volpano [47] provide the first information flow type system that allows static enforcement of non-interference for a multi-threaded programming language. As mentioned in the introduction, their type system is rather restrictive in that `while` loops cannot test on high expressions (in order to guarantee the soundness of the type system, and in particular to reject the example of the introduction). Later, in order to avoid the issue of refinement attacks in presence of scheduling, Volpano and Smith [51, 50] and also Smith [46] focus on probabilistic non-interference for uniform schedulers, i.e. schedulers that select the thread to execute from a finite set of threads with a uniform probability distribution. Stronger, scheduler-independent, notions of non-interference were proposed as well by Sabelfeld and Sands [42, 43] for a language with probabilistic choice. Their security condition is defined using partial equivalence relations (PERs), and is shown to be enforceable with an information flow type system that rejects `while` loops that test on high expressions.

Among the works that do not consider probabilistic non-interference, one finds the work by Zdancewic and Myers [53] on observational non-interference. In a nutshell, their notion of security considers low equivalence of traces up to prefix and stuttering, and is shown to be enforceable for a programming language with a rich set of features for concurrency, using a combination of an information flow type system and of another analysis to guarantee race freedom. Other facets of concurrency for which information flow type systems have been devised include synchronization and synchronous programming: indeed, Sabelfeld [39] proposes a type and effect system that enforces non-interference for a language with synchronization primitives, and Matos, Boudol and Castellani develop in [29] a sound information flow type system for a reactive language closely related to the language presented here. More recently, Russo and

Sabelfeld [38] have suggested that controlling the interactions between threads and the scheduler provides an adequate means to enforce permissive policies for concurrent languages.

Information flow type systems have also been thoroughly investigated in the context of process calculi. In particular, [19, 21, 22, 34] provide information flow type systems for the π-calculus [45]. In addition, Honda and Yoshida [21, 22] also establish type-preserving compilation from the information flow type system of [49] for a simple imperative language to their information flow type systems.

Several other works study characterizations of non-interference using program logics for concurrent programs: for example Joshi and Leino [24, 25] propose an equational characterization of non-interference, whereas Barthe, D'Argenio and Rezk [8] propose a characterization of non-interference in temporal logic. Recently, Huisman, Worah and Sunesen [23] extend the results of [8] to characterize observational equivalence [53]. While these characterizations can be used to verify statically that a program is non-interfering, they need to rely on user interaction if the domain of values is infinite. Recently, Dam [14] has shown that some notions of non-interference are decidable if the domain of values is finite; however, the definition of non-interference used in this paper is not decidable.

Finally, there are works that aim at combining different verification techniques rather than committing to a specific technique before hand. For example, Mantel, Sudbrock and Krausser [27] study the possibility of such combinations.

7.1.2 Machine-checked proofs of non-interference

While programming language research papers are seldom supported by machine-checked proofs, machine-checked verification of programming languages (operational semantics, type systems, compilers) is receiving increasing interest from the programming language community, and recent achievements demonstrate the feasibility of such tasks. Nevertheless, much of the work to-date has been focusing on safety type systems. In fact, only a few works have focused on information flow.

Motivated by the need to provide accurate models for verifying secure systems, Rushby [37] considers a notion of so-called "intransitive" non-interference which allows information release through a trusted downgrader, and establishes an unwinding theorem that provides a set of sufficient conditions for a system to be secure. The whole account is formalized using the verification system EHDM, a precursor to PVS.

Naumann [31] and Strecker [48] have also provided machined-checked proofs of soundness for information flow type systems for a (sequential) fragment of Java that includes objects and methods. Their language, type system and formal proofs are inspired from [7, 6], and consider a termination-insensitive, input-output based notion of non-interference.

However, we do not know of any machine-checked proof of non-interference for a concurrent programming language.

7.2 Future work

We intend to pursue our work along two different axes.

Language expressiveness: While expressive, the programming language modeled in this paper falls short of the complexity of modern programming languages. The objective of this line of work is to adapt and extend our results to richer or more realistic settings. There are several obvious steps to take: first of all, we would like to adapt our results to a programming language with procedures and exceptions; a more challenging goal would be to apply our type system to a concurrent fragment of Java. Second of all, it would seem interesting to generalize our proofs to programming languages with support for synchronous programming. In [11], Boudol proposes a core language for global computing, called ULM, with features such as instants and broadcasts which exhibit an interesting behavior from the point of view of information flow. Using the results of [29], it should be possible to adapt our formal proofs to ULM. In the medium term, it may be of interest to derive similar results for process calculi, using the works mentioned in the previous paragraph. Isabelle formalizations exist for some process calculi and could be used as a basis to formally machine-check the correctness of the information flow type systems for these process calculi: for example, there exist several machine-checked formalizations of π-calculus, see e.g. [36] for one example of such a formalization using Isabelle/HOL.

Security policy: While non-interference provides an elegant semantical condition that guarantees confidentiality, it also imposes stringent restrictions that reduce its applicability. Thus many works have focused on weaker definitions of confidentiality that admit some controlled form of information release. It would be interesting to adapt and extend our results to more realistic semantical definitions of security, such as those considered in [44]. Of course, the difficulty of the endeavor will vary greatly depending on the definition considered: for example, it should be reasonably feasible to adapt our results to account for delimited information release [41], or for the non-disclosure policy [28], whereas adapting our results to encryption, see e.g. [52] is substantially more challenging. In order to accommodate security definitions for which no information flow type system exists, it would also be interesting to investigate logical characterizations of non-interference and related definitions in the spirit of [8].

At the opposite end of the spectrum, one could also consider more stringent security conditions such as timing-sensitive non-interference, and prove the correctness of program transformations that ensure such stronger conditions, see e.g.[2].

Acknowledgments Thanks to the anonymous referees for their comments on the paper, to G. Boudol, I. Castellani, and A. Matos for discussions around [13]

and [29], and to S. Merz, C. Ballarin and F. Kammüller for discussions about the Isabelle formalization. This work was partially funded by the IST Projects Profundis and MOBIUS, by the QSL Project, by the ACI Sécurité SPOPS, and by the RNTL Project Castles.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of NDSS'03*, pages 107–121. Internet Society, 2003.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of POPL'00*, pages 40–53. ACM Press, 2000.

[3] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, New York, Berlin, 1991.

[4] B.E. Aydemir, A. Bohannon, M. Fairbairn, J.N. Foster, B.C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Tom Melham, editors, *Proceedings of TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 2005.

[5] C. Ballarin. Locales and locale expressions in isabelle/isar. In Stefano Berardi et al., editor, *Proceedings of TYPES'03*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer-Verlag, 2003.

[6] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[7] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.

[8] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.

[9] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.

[10] G. Barthe and L. Prensa-Nieto. Formally verifying information flow type systems for concurrent and thread systems. In M. Backes, D. Basin, and M. Waidner, editors, *Proceedings of FMSE'04*, pages 13–22. ACM Press, 2004.

[11] G. Boudol. ULM: A Core Programming Model for Global Computing: (Extended Abstract).W. In D.A. Schmidt, editor, *Proceedings of ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248. Springer-Verlag, 2004.

[12] G. Boudol and I. Castellani. Noninterference for concurrent programs. In F. Orejas, P.G. Spirakis, and J. van Leeuwen, editors, *Proceedings of ICALP'01*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395, 2001.

[13] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002. Preliminary version available as INRIA Research report 4254.

[14] M. Dam. Decidability and proof systems for language-based noninterference relations. In *Proceedings of POPL'06*, pages 67–78. ACM Press, 2006.

[15] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *Proceedings of CSFW'04*, pages 115–124. IEEE Press, 2004.

[16] R. Focardi and R. Gorrieri. Classification of security properties: (part i: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 331–396. Springer-Verlag, 2001.

[17] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. In V. Sassone, editor, *Proceedings of FOSSACS'05*, volume 3441 of *Lecture Notes in Computer Science*, pages 299–315. Springer-Verlag, 2005.

[18] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society Press, 1982.

[19] M. Hennessy and J. Riely. Information flow vs. resource access in the information asynchronous pi-calculus. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2000.

[20] T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In R. Alur and I. Lee, editors, *Proceedings of EMSOFT'03*, volume 2855 of *Lecture Notes in Computer Science*, pages 241 – 256, 2003.

[21] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Proceedings of ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

[22] K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002.

[23] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of CSFW'06*, pages 3–15. IEEE Computer Society Press, 2006.

[24] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[25] K. Leino and R. Joshi. A semantic approach to secure information flow. In J. Jeuring, editor, *Proceedings of MPC'98*, volume 1422 of *Lecture Notes in Computer Science*, pages 254–271, 1998.

[26] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[27] H. Mantel, H. Sudbrock, and T. Krausser. Combining different proof techniques for verifying information flow security. In G. Puebla, editor, *Pre-Proceedings of LOPSTR'06*, volume Raporta di Ricerca CS-2006-5, Università Ca' Foscari Di Venezia, 2006.

[28] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proceedings of CSFW'05*, pages 226–240. IEEE Computer Society Press, 2005.

[29] A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In A. Sabelfeld, editor, *Proceedings of FCS'04*, 2004.

[30] M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.

[31] D. Naumann. Machine-checked correctness of a secure information flow analyzer (preliminary report). Technical Report CS-2004-10, Stevens Institute of Technology, March 2003.

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[33] L.C. Paulson. *The Isabelle Reference Manual*. University of Cambridge, Computer Laboratory. `http://www.in.tum.de/~isabelle/dist/`.

[34] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proceedings of CSFW'02*, pages 320–330. IEEE Press, 2002.

[35] L. Prensa Nieto. `http://www.loria.fr/~prensa`.

[36] C. Röckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming*, 13:415–451, 2003.

[37] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-02, Computer Science Laboratory, SRI International, dec 1992.

[38] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of CSFW'06*, 2006.

[39] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of PSI'01*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, 2001.

[40] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.

[41] A. Sabelfeld and A. Myers. A model for delimited information release. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Proceedings of ISSS'03*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, 2004.

[42] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In D. Swiestra, editor, *Proceedings of ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 40–58. Springer-Verlag, 1999.

[43] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[44] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of CSFW'05*. IEEE Press, 2005.

[45] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[46] G. Smith. A New Type System for Secure Information Flow. In *Proceedings of CSFW'01*, pages 115–125, 2001.

[47] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, 1998.

[48] M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[49] D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.

[50] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of CSFW'98*, pages 34–43. IEEE Press, 1998.

[51] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7:231–253, November 1999.

[52] D. M. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings of POPL'00*, pages 268–276. ACM Press, 2000.

[53] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of CSFW'03*, pages 29–46. IEEE Press, 2003.