Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 351 (2020) 75–94

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Agda Formalization of a Security-preserving Translation from Flow-sensitive to Flow-insensitive Security Types

Cecilia Manzino[1]

Departamento de Ciencias de la Computación
Universidad Nacional de Rosario
Argentina

Alberto Pardo[2]

Instituto de Computación
Universidad de la República
Montevideo, Uruguay

Abstract

The analysis of information flow is a popular technique for ensuring the confidentiality of data. It is in this context that confidentiality policies arise for giving guarantees that private data cannot be inferred by the inspection of public data. One of those policies is non-interference, a semantic condition that ensures the absence of illicit information flow during program execution by not allowing to distinguish the results of two computations when they only vary in their confidential inputs. A remarkable feature of non-interference is that it can be enforced statically by the definition of information flow type systems. In those type systems, if a program type-checks, then it means that it meets the security policy.
In this paper we focus on the preservation of non-interference through program translation. Concretely, we formalize the proof of security preservation of Hunt and Sands' translation that transforms high-level While programs typable in a flow-sensitive type system into equivalent high-level programs typable in a flow-insensitive type system. Our formalization is performed in the dependently-typed language Agda. We use the expressive power of Agda's type system to encode the security type systems at the type level. A particular aspect of our formalization is that it follows a fully internalist approach where we decorate the type of the abstract syntax with security type information in order to obtain the representation of well-typed (i.e secure) programs. A benefit of this approach is that it allows us to directly express the property of security preservation in the type of the translation relation. In this manner, apart from inherently expressing the transformation of programs, the translation relation also stands for an inductive proof of security preservation.

Keywords: non-interference, information flow type systems, dependently-typed programming, Agda, type safety

[1] Email: ceciliam@fceia.unr.edu.ar
[2] Email: pardo@fing.edu.uy

https://doi.org/10.1016/j.entcs.2020.08.005

1 Introduction

The analysis of information flow is a popular technique for ensuring the confidentiality of data. It is in this context that confidentiality policies arise for giving guarantees that private data cannot be inferred by the inspection of public data. Non-interference [3] is an example of a security policy. It is a semantic condition that ensures the absence of illicit information flow during program execution by not allowing to distinguish the results of two computations when they only vary in their confidential inputs. A remarkable feature of non-interference is that it can be enforced statically by the definition of an information flow type system [2,16,14,8]. Thus, when a program type-checks in such a type system then it means that it satisfies the security policy. In this setting, program variables are classified in different categories (types) according with the kind of information they can store (e.g., public or confidential data). The advantage of modelling security properties in terms of types is that they can be checked at compile-time, thus partially reducing or even eliminating the overhead of checking properties at run-time.

Most of the security type systems are *flow-insensitive* [14]. These are type systems in which the security level of the program variables remain unchanged. This contrasts with security type systems that are *flow-sensitive* [4,13]. In those type systems each variable can have a different security level at different points of the program. Flow-sensitive type systems are more permissive than flow-insensitive ones since they accept a larger set of secure programs.

In this paper we focus on the preservation of non-interference through program translation. Concretely, we formalize the proof of security preservation of Hunt and Sands' translation [4] that transforms high-level While programs typable in a flow-sensitive type system into equivalent high-level programs typable in a flow-insensitive type system. Our formalization is performed in the dependently-typed functional language Agda [9,1]. We use the expressive power of Agda's type system to encode the security type systems at the type level.

A particular aspect of our formalization is that it follows a fully internalist approach [10,7,12], where we decorate the type of the abstract syntax with security type information in order to obtain the representation of well-typed (i.e secure) programs. These are terms that simulaneously represent ASTs and (their associated) typing rules in the formal type system [15,11]. An interesting consequence of this representation is that it restricts the object language terms that are representable. Indeed, not every AST is representable, but only those that are well-typed according to the type system of the object language. But even more interesting is the effect that this representation has on functions between typed terms: only functions that preserve those type invariants are accepted. The positive aspect of this fact is that its verification reduces to type-checking.

The paper is organized as follows. In Section 2 we present the high-level language that serves as source and target of the translation and a flow-insensitive type system for it. Section 3 deals with the flow-sensitive type system for the language defined by Hunt and Sands. Section 4 presents the program transformation and the proof

that it preserves security typing. Section 5 concludes the paper.

The complete Agda code is available at
https://www.fceia.unr.edu.ar/~{}ceciliam/codes/.

2 Type-insensitivity

We start with a summary description of the language that serves as source and target of the translation. After that we present one of the type systems that enforces secure information flow for programs of the language. The system to be shown in this section is a flow-insensitive type system and corresponds to the system that is used to type the target programs of the translation. We decided to start by describing the endpoint of the translation because the system is more natural and easier to understand. It is also a suitable context where to introduce the internalist approach we want to pursue for the Agda implementation.

Out of differences in implementation details, the flow-insensitive type system to be shown in this section has already been presented in [7], where we developed a Haskell implementation of a security-preserving compiler also using an internalist approach.

2.1 The language

The language to be used in the translation is a standard While language with expressions and statements defined by the following abstract syntax:

$$e \ ::= n \mid x \mid e_1 + e_2$$

$$S ::= x := e \mid \texttt{skip} \mid S_1;S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S$$

where $e \in \mathbf{Exp}$ (expressions), $S \in \mathbf{Stm}$ (statements), $x \in \mathbf{Var}$ (variables) and $n \in \mathbf{Num}$ (integer literals).

The semantics is completely standard. The meaning of both expressions and statements is given relative to a state $s \in \mathbf{State} = \mathbf{Var} \to \mathbf{Num}$, which contains the current value of each variable.

We assume the semantics for expressions is given by an evaluation function $\mathcal{E} : \mathbf{Exp} \to \mathbf{State} \to \mathbf{Num}$ defined by induction on the structure of expressions. For statements, we define a big-step semantics whose transition relation is written as $\langle S, s\rangle \Downarrow s'$, meaning that the evaluation of a statement S in an initial state s terminates with a final state s'. The definition of the transition relation is presented in Figure 1.

Notice that, for simplicity, the language does not contain boolean expressions. Instead, the condition of an `if` or a `while` statement is given by an arithmetic expression such that the condition is true when the expression evaluates to zero, and false otherwise.

$$\frac{}{\langle x := e,\, s\rangle \Downarrow s[x \mapsto \mathcal{E}[\![e]\!]\, s]} \qquad \frac{}{\langle \texttt{skip},\, s\rangle \Downarrow s} \qquad \frac{\langle S_1,\, s\rangle \Downarrow s' \quad \langle S_2,\, s'\rangle \Downarrow s''}{\langle S_1;S_2,\, s\rangle \Downarrow s''}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S_1,\, s\rangle \Downarrow s'}{\langle \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \Downarrow s'} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0 \quad \langle S_2,\, s\rangle \Downarrow s'}{\langle \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2,\, s\rangle \Downarrow s'}$$

$$\frac{\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S,\, s\rangle \Downarrow s' \quad \langle \texttt{while } e \texttt{ do } S,\, s'\rangle \Downarrow s''}{\langle \texttt{while } e \texttt{ do } S,\, s\rangle \Downarrow s''} \qquad \frac{\mathcal{E}[\![e]\!]\, s \neq 0}{\langle \texttt{while } e \texttt{ do } S,\, s\rangle \Downarrow s}$$

Figure 1. Big-step semantics of statements

2.2 Security Type System

Suppose we want to model a security scenario where each program variable has associated a security level stating the degree of confidentiality of the values it stores. In such a context it is natural to implement some security mechanism in order to protect confidential data. We will do so by implementing an information flow type system [16,14,8]. The type system to be defined in this section is *flow-insensitive* in the sense that it considers that the security level of the variables is maintained unchanged during program execution. Variables with this property are called fixed-type variables. This contrasts with the so-called floating-type variables, whose security level may vary along program excecution. We will deal with floating-type variables in the *flow-sensitive* type system to be presented in Section 3.

We assume a bounded lattice of security levels $(\mathcal{L}, \leq)$ with meet ($\wedge$) and join ($\vee$) operations, and top ($\top$) and bottom ($\bot$) values. The bottom value represents the least security level (*public* data) whereas the top value represents the highest security level (*confidential* data). We also assume that the lattice comes with an equality relation $\approx$ between levels. An expression $l < l'$ means that security level l is less confidential than l'.

Non-interference is a property on programs that guarantees the absence of illicit information flows during their execution. An illicit flow occurs when information flows from variables of higher security level to variables with lower security level. A program satisfies this security property when the final value of any variable with security level l is not influenced by a variation of the initial value of variables with higher security level. This property can be formulated in terms of program semantics. We write x_l to refer to a variable with security level l. Let us say that two states s and s' are l-equivalent, written $s \cong_l s'$, when every variable with lower security level than l contains the same value in both states; i.e. $s(x_{l'}) = s'(x_{l'})$ for every $x_{l'}$ with $l' < l$. The significance of l-equivalence is that l-equivalent states are indistinguishable to a lower than l confidential observer (i.e. an observer that can only inspect data with security level less than l).

A program $S \in$ **Stm** is said to be **non-interfering** when, for any pair of l-

equivalent initial states, if the execution of S starting on each of these states terminates, then it does so in l-equivalent final states:

$$\mathbf{NI}(S) \stackrel{\text{df}}{=} \forall s_i, s_i'.\ s_i \cong_l s_i' \ \wedge\ \langle S,\ s_i\rangle \Downarrow s_f \ \wedge\ \langle S,\ s_i'\rangle \Downarrow s_f' \implies s_f \cong_l s_f'$$

This definition of non-interference is *termination-insensitive* in the sense that it does not take into account non-terminating executions of programs.

It is well-known that this property can be checked statically by the definition of an information-flow type system that enforces noninterference [16,14]. In Figure 2 we present a syntax-directed security type system for our language (alternative formulations for the type system can be found in [8,6]). Security levels are used as types and are referred to as security types. The reason for presenting a syntax-directed type system is because it is the appropriate formulation to be considered later for the implementation.

Expressions The type system for expressions uses a judgement of the form $\vdash e : st$, where $st \in \mathcal{L}$. According to this system, the security type of an expression is the maximum of the security types of its variables. Integer numerals are considered public data.

Statements The goal of secure typing for statements is to prevent improper information flows during program execution. Information flow can appear in two forms: explicit or implicit.

An *explicit flow* is observed when data are copied to less confidential variables. Consider two variables x_H and y_L, with $L < H$. For example, the assignment $y_L := x_H + 1$ is not allowed because the value of the variable x_H is copied to a less confidential variable, y_L. On the other hand, an assignment in the opposite direction, e.g. $x_H := y_L$, is authorized, since it does not represent a security violation.

Implicit information flows arise from the control structure of the program. The following is an example of an insecure program where an implicit flow occurs (again assume $L < H$):

$$\texttt{if}\ x_H\ \texttt{then}\ y_L := 1\ \texttt{else skip}$$

The reason for being insecure is because by observing the value of the variable y_L on different executions we can infer information about the value of the variable x_H. This is because we are performing the assignment of a variable with security type L in a more confidential context (in this case, the branch of a conditional statement with a condition of type H). Due to these situations it is necessary to keep track of the security level of the program counter in order to know the security level of the context in which a sentence occurs. On the other hand, a program like this:

$$\texttt{if}\ y_L + 2\ \texttt{then}\ z_L := y_L + 1\ \texttt{else}\ x_H := x_H - 1$$

is accepted because the final value of the variable z_L only depends on the initial value of y_L.

The typing judgement for statements has the form $[pc] \vdash S$ and means that S is typable in the security context pc. Rule ASS states that an assigment to a variable

EXPRESSIONS

$$\vdash n : \bot \qquad \vdash x_t : t \qquad \frac{\vdash e : st \quad \vdash e' : st'}{\vdash e + e' : st \vee st'}$$

STATEMENTS

$$[pc] \vdash \texttt{skip} \ \text{SKIP} \qquad \frac{\vdash e : st \quad st \leq t \quad pc \leq t}{[pc] \vdash x_t := e} \ \text{ASS} \qquad \frac{[pc] \vdash S_1 \quad [pc] \vdash S_2}{[pc] \vdash S_1;S_2} \ \text{SEQ}$$

$$\frac{\vdash e : st \quad [st \vee pc] \vdash S_1 \quad [st \vee pc] \vdash S_2}{[pc] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2} \ \text{IF} \qquad \frac{\vdash e : st \quad [st \vee pc] \vdash S}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S} \ \text{WHILE}$$

Figure 2. Flow-insensitive Security Type System

x_t can be done in a context lower or equal than t; explicit flows are prevented by the restriction $st \leq t$. In order to prevent implicit flows, the rules IF and WHILE impose a restriction between the security level of the condition and the branches of the conditional or the body of the while. In a context pc, if the condition has type st, then the branches (of the `if`) or the body (of the `while`) must type in a context which is the least upper bound of pc and st.

A desirable property for a security type system is type soundness, which means that every typable statement satisfies non-interference. We build up the soundness proof using two lemmas taken from [8]. The first one, called *confinement*, states that if a sentence is typable in a context pc then the execution of the sentence does not alter the value of the variables with level lower than pc.

Lemma 2.1 (CONFINEMENT) $[pc] \vdash S \ \wedge \ \langle S, s\rangle \Downarrow s' \implies s \cong_l s' \, , with \ l < pc.$

Proof The proof can be done trivially by induction on the derivations of the evaluation relation $\langle S, s\rangle \Downarrow s'$.

The second lemma, called *anti-monotonicity*, states that if a sentence is typable in a context pc, then is also typable in any context lower than pc.

Lemma 2.2 $[pc] \vdash S \ \wedge \ pc' \leq pc \implies [pc'] \vdash S.$

Proof Straightforward by induction on the derivation of $[pc] \vdash S$.

Based on these lemmas, we can now state type soundness.

Theorem 2.3 (TYPE SOUNDNESS) $[pc] \vdash S \implies \mathbf{NI}(S).$

We formalized the proof of this theorem in Agda for a lattice of two levels; the proof is available at `https://www.fceia.unr.edu.ar/~ceciliam/codes/proofs`.

2.3 Implementation

Now we present an Agda implementation of the abstract syntax and the type system. We proceed following an internalist approach where we attach type invariants (in

our case typing information) to our abstract syntax representations. Proceeding that way we obtain the respresentation of *well-typed* expressions and statements.

We represent the expressions by means of a type family Exp, which is indexed by a value of type $\mathbb{S}$ representing the security type of the expression. $\mathbb{S}$ is the carrier set of our lattice of security types.

```
data Exp  : 𝕊 → Set where
  IntVal  : ℕ → Exp ⊥
  Var     : (x : Fin n) → (st : 𝕊) → Exp st
  Add     : Exp st → Exp st' → Exp (st ∨ st')
```

The internalist reprentation of statements is given by the following type family, which is indexed by a security type representing the security level of the program context in which the statement is executed.

```
data Stm  : 𝕊 → Set where
  Skip    : Stm  pc
  Assign  : (x : Fin n) → (y : 𝕊) → st ≼ y → pc ≼ y → Exp st → Stm pc
  Seq     : Stm pc → Stm pc → Stm pc
  If0     : Exp st → Stm (pc ∨ st) → Stm (pc ∨ st) → Stm pc
  While   : Exp st → Stm (pc ∨ st) → Stm pc
```

Notice that the constructors of Exp and Stm are a direct implementation of the typing rules in Figure 2. A benefit of the internalist approach is that now, the typing judgement S : Stm pc directly corresponds to the judgement $[pc] \vdash S$ in our formal type system.

For reasons that will be clear later on when we see the flow-sensitive type system, instead of simply using naturals, we use elements from a finite set (Fin n) to represent variables. As a consequence of this, our representation of expressions and statements in parameterised by a natural n.

3 Flow-Sensitive Type System

In a flow-sensitive type system, the security type of a variable can change during program execution. This allows us to type more secure programs than with a flow-insensitive type system. For example, consider the following code, again with $L < H$,

$$x_L := y_H;\ x_L := 0$$

Although in the second assignment the variable x_L is overridden with the constant 0, this code is rejected by a flow-insensitive type system because it has an insecure statement ($x_L := y_H$). This is, however, accepted by a flow-sensitive type system because the security level of x_L is relabeled to H after the first assignment.

Another example of an intuitively secure code that is rejected by a flow-insensitive type system is the following:

$$\textsc{Skip}\ \frac{}{pc \vdash \Gamma\ \{\ \texttt{skip}\ \}\ \Gamma} \qquad \textsc{Assign}\ \frac{\Gamma \vdash e\ :\ t}{pc \vdash \Gamma\ \{\ x := e\ \}\ \Gamma[x \mapsto pc \vee t]}$$

$$\textsc{Seq}\ \frac{pc \vdash \Gamma\ \{\ S_1\ \}\ \Gamma' \quad pc \vdash \Gamma'\ \{\ S_2\ \}\ \Gamma''}{pc \vdash \Gamma\ \{\ S_1;S_2\ \}\ \Gamma''} \qquad \textsc{If}\ \frac{\Gamma \vdash e\ :\ t \quad pc \vee t \vdash \Gamma\ \{\ S_i\ \}\ \Gamma_i' \quad i = 1,2}{pc \vdash \Gamma\ \{\ \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \}\ \Gamma_1' \sqcup \Gamma_2'}$$

$$\textsc{While}\ \frac{\Gamma_i' \vdash e\ :\ t_i \quad pc \vee t_i \vdash \Gamma_i'\ \{\ S\ \}\ \Gamma_i'' \quad 0 \leq i \leq n}{pc \vdash \Gamma\ \{\ \texttt{while}\ e\ \texttt{do}\ S\ \}\ \Gamma_n'} \qquad \Gamma_0' = \Gamma,\ \Gamma_{i+1}' = \Gamma_i'' \sqcup \Gamma,\ \Gamma_{n+1}' = \Gamma_n'$$

Figure 3. Flow-sensitive Type System

$$y_H := 0;\ \textbf{if}\ y_H\ \textbf{then}\ x_L := 1\ \textbf{else}\ x_L := 2$$

This code is secure since the value of the variable y_H is overwritten with 0 before the conditional and therefore by inspecting the value of x_L we cannot know anything about the value of x_H before assigning it to 0.

3.1 Type System

Now we present the flow-sensitive type system that was defined by Hunt and Sands [4,5] for the While language. This is the type system that is used to type-check the source programs of the translation. Like before, the type system is parameterized by a lattice of security types $\mathcal{L}$.

Concerning the expressions, the typing judgement $\Gamma \vdash e\ :\ t$ now requires a type environment Γ because the type system deals with floating-type variables. Again, the type of an expression is defined as the least upper bound of the types of its variables.

For statements, the judgement is now of the form $pc \vdash \Gamma\ \{\ S\ \}\ \Gamma'$, meaning that S is typable in the security context pc and the type environments Γ and Γ'. Those environments (called pre- and post-environment, respectively) describe the security level of the variables before and after the execution of S. A post-environment Γ' is now necessary because the security level of the variables may be changed by the execution of a statement.

Figure 3 shows a syntax-directed flow-sensitive type system for statements. The formulation of the type system has an algorithmic character in the sense that $pc \vdash \Gamma\ \{S\}\ \Gamma'$ computes the least post-environment Γ' obtained after the execution of statement S in a pre-environment Γ and context pc. Like in the case of flow-insensitivity, having a syntax-directed formulation of the type system results helpful because it allows us to represent it in terms of type families.

Environments form a join-semilattice whose structure is inherited from that of the lattice of security levels. In this sense, the join of two environments ($\sqcup$) is defined

$$\Gamma'_0 = \Gamma, \Gamma''_0 = \Gamma[w \mapsto N][z \mapsto M] \qquad \Gamma \vdash w > 0 \,:\, L \qquad \frac{L \vdash \Gamma \,\{\, w := y \,\}\, \Gamma[w \mapsto N] \quad L \vdash \Gamma[w \mapsto N] \,\{\, z := x + 1 \,\}\, \Gamma''_0}{L \vdash \Gamma \,\{\, w := y; z := x + 1 \,\}\, \Gamma''_0}$$

$$\Gamma'_1 = \Gamma''_0 \sqcup \Gamma = \Gamma[w \mapsto N] \qquad \Gamma'_1 \vdash w > 0 \,:\, N \qquad \frac{N \vdash \Gamma'_1 \,\{\, w := y \,\}\, \Gamma'_1 \quad N \vdash \Gamma'_1 \,\{\, z := x + 1 \,\}\, \Gamma'_1[z \mapsto H]}{N \vdash \Gamma'_1 \,\{\, w := y; z := x + 1 \,\}\, \Gamma'_1[z \mapsto H]}$$

$$\Gamma'_2 = \Gamma''_1 \sqcup \Gamma = \Gamma[w \mapsto N] = \Gamma'_1$$

Figure 4. Example of a typing iteration

pointwise: $\Gamma \sqcup \Gamma' = \lambda x.\, \Gamma(x) \vee \Gamma'(x)$; the same with the partial order between two environments: $\Gamma \sqsubseteq \Gamma'$ iff $\forall x.\, \Gamma(x) \leq \Gamma'(x)$. The semilattice has a top element given by the environment with all variables in the top security type ($\top$).

According to rule Assign, after an assignment the type of a variable x may: (i) change to a higher value $pc \vee t$ if the assignment is performed in a context pc and the assigned expression is of type t, with $\Gamma(x) < pc \vee t$; (ii) change to a lower value if $\Gamma(x) > pc \vee t$; or (iii) remain unaltered otherwise.

As usual, the rules for If and While are designed to prevent implicit flows. The branches (of the conditional) or the body (of the while) must be typable in a context which is the least upper bound of the context pc and the type t of the condition.

For example, considering the following lattice of security types:

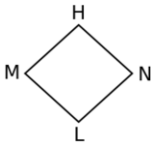

Figure 4 shows the steps of the typing iteration for the code: `while` $x > 0$ `do` $w := y; z := x + 1$ starting with the environment $\Gamma = \{w : \mathsf{L}, x : \mathsf{M}, y : \mathsf{N}, z : \mathsf{H}\}$.

Notice that in the While rule the post-environment is the result of an iterative construction which iterates until a fixed-point is obtained. The body of the loop is repeatedly typed until the post-environment does not change with respect to the last iteration. The rule can then be reformulated in terms of a least fixed point operator:

$$\textsc{While-fix}\ \frac{\Gamma_f = \mathrm{fix}(\lambda\Gamma\ .\ \text{let}\ \ \Gamma \vdash e \,:\, t \quad pc \vee t \vdash \Gamma \,\{\, S \,\}\, \Gamma' \ \text{in}\ \Gamma' \sqcup \Gamma_0)}{pc \vdash \Gamma_0 \,\{\, \texttt{while}\ e\ \texttt{do}\ S \,\}\, \Gamma_f}$$

This fixed point construction is guaranteed to terminate because it is computed on a monotone function (defined over the typing rules) and the set of environments is finite. The proof of convergence of this rule was given in [4] as part of the proof of the following theorem, which states the correctness of the type system. We refer as $\mathcal{A}^S$ to the function that calculates the least Γ' such that $pc \vdash \Gamma \,\{\, S \,\}\, \Gamma'$.

Theorem 3.1 ([4]) *For all S, pc, Γ, there exists a unique Γ' such that $pc \vdash \Gamma \,\{\, S \,\}\, \Gamma'$ and furthermore, the corresponding function $\mathcal{A}^S(pc, \Gamma) \mapsto \Gamma'$ is monotone.*

To implement this type system in Agda we need to define a function that computes the fixed point of the While-fix rule. The construction of the fixed point is based on the Agda formalization of the following theorem (Theorem 3.2), which

states the existence of a fixed point for any monotone function that satisfies certain requirements on an arbitrary partially ordered set.

In the formulation of the theorem we capture the relevant characteristics that our semilattice of environments has. Taking into account that environments are finite mappings that contain only the variables that occur in the analysed program, it turns out that the semilattice is finite. From that, what is relevant for us is that every strictly ascending chain in the semilattice has finite length; in particular, every chain that ends at the top environment. This will be reflected in the theorem by assuming the existence of a function, called *bound*, that associates a natural number to every element of the poset and is strictly decreasing with respect to the strict order:

$$x \sqsubset y \Rightarrow \mathit{bound}\, y < \mathit{bound}\, x.$$

The intuition behind the value of *bound* is that it represents the length of the longest strictly ascending chain from an element to top ($\top$). The value of *bound* at $\top$ is of course zero. Instead of requiring that the poset has a top element and that the value of *bound* at $\top$ is zero, we will equivalently require that *bound* has a unique minimal element:

$$\mathit{bound}\, x = 0 \ \wedge \mathit{bound}\, y = 0 \Rightarrow x = y.$$

Theorem 3.2 *Let $(S, \sqsubseteq)$ be a poset, $x \in S$ an element, $g : S \to S$ a monotone function over $\sqsubseteq$, and bound $: S \to \mathbb{N}$ a strictly decreasing function with respect to the strict order ($\sqsubset$) and with a unique minimal element. If there is an element $x \in S$ such that $x \sqsubseteq g\ x$ and bound $x \leq k$ for some k, then there exists $n \leq k$ such that $g^n\ x$ is a fixed-point of g.*

Proof The proof is by induction on k.

- $k = 0$: By hypothesis *bound* $x \leq 0$, and therefore *bound* $x = 0$. Since *bound* is a decreasing function and $x \sqsubseteq g\ x$, we have that *bound* $(g\ x) \leq$ *bound* x. Thus, *bound* $(g\ x) = 0$. By uniqueness of *bound*'s minimal element we conclude that $g\ x = x$, and therefore x is a fixed-point of g.
- $k = k' + 1$: By hypothesis we know that *bound* $x \leq k' + 1$ and $x \sqsubseteq g\ x$. Then we have two cases:
  case $x = g\ x$ **:** Then x is a fixed-point of g.
  case $x \sqsubset g\ x$ **:** Since *bound* is strictly decreasing wrt $\sqsubset$ we have that *bound* $(g\ x) <$ *bound* $x \leq k' + 1$, and therefore *bound* $(g\ x) \leq k'$. Since g is monotone and $x \sqsubset g\ x$ we have that $g\ x \sqsubseteq g^2\ x$. Then, by induction hyphotesis we conclude that there exists $n \leq k'$ such $g\ ((g^n\ (g\ x)) = g^n\ (g\ x)$, meaning that $g^n\ (g\ x)$ is a fixed-point of g. Therefore, we conclude that $g^{n+1}\ x$ is a fixed-point of g where $n + 1 \leq k' + 1$. □

Based on this theorem we implement in Agda a function fixS that computes the fixpoint. $\mathbb{S}$ denotes the carrier set of the poset, $\leq$ its order relation, $\approx$ the equality in $\mathbb{S}$ (which we assume is decidable). We write $\leq\mathbb{N}$ to the denote less or equal between natural numbers; $\sharp$ denotes the bound function which we assume is strictly decreasing wrt the strict order in the poset.

```
fixS :  -- boundary of iterations
        (k : ℕ)
        -- monotone function
        (g : 𝕊 → 𝕊) →
        (∀ {x y} → x ≤ y → g x ≤ g y) →
        -- initial value
        (s : 𝕊) →
        s ≤ g s →
        -- invariant
        ♯ s ≤ℕ k →
        Σ 𝕊 (λ x → x ≈ g x )
```

Function fixS turns out to be itself a monotone function wrt the elements of $\mathbb{S}$. This fact should not surprise because we are simply working with an abstraction of the fixpoint construction we had in the type system for statements. Later we will use the monotonicity of fixS for the implementation of the type system.

3.2 Implementation

For the Agda implementation of the language with a flow-sensitive type system we want to proceed in a similar way as we did for the flow-insensitive type system using an internalist approach. Again, the goal is to define type families that represent (flow-sensitive) typed terms for expressions and statements. On those typed terms we will define later the translation to flow-insensitive typed terms.

For expressions the internalist implementation is immediate. Expressions are represented by a type ExpS, which is parametrized by the type environment (given by a vector of security types), and the security type of the expression. Like before, $\mathbb{S}$ denotes the carrier set of the lattice of security types.

```
data ExpS    (Γ : Vec 𝕊 n) : 𝕊 → Set where
  IntValS    : ℕ → ExpS Γ ⊥
  VarS       : (x : Fin n) → ExpS Γ (lookup x Γ )
  AddS       : ExpS Γ st → ExpS Γ st' → ExpS Γ (st ∨ st')
```

Variable n denotes the number of program variables that occur in the program code. Each program variable is identified by a value in the finite set Fin n. That way, each variable has associated a position in a type environment of type Vec $\mathbb{S}$ n.

The internalist implementation of statements is in terms of a type family called StmS, which is indexed by the program counter and the (pre- and post-) environments. A judgement $pc \vdash \Gamma\ \{S\}\ \Gamma'$ is then represented by a typing judgement in Agda: S : StmS Γ pc Γ'. Like in Section 2.3, this is indeed possible because we have a syntax-directed type system.

The implementation of the flow-sensitive typing rules for skip, assigments, con-

ditionals and sequences is direct and poses no difficulty. For while statements, however, the situation is other. The process is a bit more laborious because we have to deal with the computation of the fixpoint. Such computation requires that we iterate over the typing judgement corresponding to the body of the while and that is impossible to be done with a decorated term of type StmS. This leads us to implement the fixpoint construction on ordinary undecorated abstract syntax terms, an undesirable situation that seems to be unavoidable. The only terms that will be represented in that undecorated form will be the term that represents the body of the while and the term corresponding to the loop condition. This will require to implement the typing relation for statements separately (in a classical externalist fashion) in order to be able to type check the body of the loop in each iteration.

We start with the definition of the undecorated abstract syntax and the relation TyStm, which implements the typing rules for statements in an externalist fashion. This requires the definition of a fix function in order to implement the WHILE-FIX rule. As second step we define the decorated type family StmS.

The undecorated abstract syntax for expressions and statements is given by the following type families. Both datatypes are indexed by the number of variables in the program code.

```
data ASTExp (m : ℕ) : Set where
  INTVAL : ℕ → ASTExp m
  VAR    : Fin m → ASTExp m
  ADD    : ASTExp m → ASTExp m → ASTExp m

data ASTStm (m : ℕ) : Set where
  ASSIGN : Fin m → ASTExp m → ASTStm m
  IF0    : ASTExp m → ASTStm m → ASTStm m → ASTStm m
  WHILE  : ASTExp m → ASTStm m → ASTStm m
  SEQ    : ASTStm m → ASTStm m → ASTStm m
```

The type system for statements is implemented by the following relation. Function tyExp computes the security type of an expression under an environment Γ.

```
data TyStm : ASTStm n → 𝕊 → Vec 𝕊 n → Vec 𝕊 n → Set where
  Skip : {Γ : Vec 𝕊 n}{ pc : 𝕊} → TyStm SKIP pc Γ Γ
  Ass : {x : Fin n}{e : ASTExp n}{Γ : Vec 𝕊 n}{pc : 𝕊} →
    TyStm (ASSIGN x e) pc Γ (change x Γ (pc ∨ (tyExp Γ e)))
  Seq : {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊} {s₁ s₂ : ASTStm n} →
    TyStm s₁ pc Γ Γ' →
    TyStm s₂ pc Γ' Γ" →
    TyStm (SEQ s₁ s₂) pc Γ Γ"
  If0 : {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊}{e : ASTExp n}{s₁ s₂ : ASTStm n} →
    TyStm s₁ (pc ∨ (tyExp Γ e)) Γ Γ' →
    TyStm s₂ (pc ∨ (tyExp Γ e)) Γ Γ" →
    TyStm (IF0 e s₁ s₂) pc Γ (Γ' ⊔ Γ")
```

```
While : {Γ : Vec 𝕊 n}{pc : 𝕊}{e : ASTExp n}{s : ASTStm n} →
  TyStm (WHILE e s) pc Γ (proj₁ (fix s e pc Γ))
```

Given an initial environment Γ_0 and a program context pc, we know that the post-environment that the rule for while computes is the result of the following fixpoint construction:

$$\mathrm{fix}(\lambda\Gamma\ .\ \text{let}\ \ \Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\ \{\ S\ \}\ \Gamma'\ \text{in}\ \Gamma' \sqcup \Gamma_0)$$

We implement the computation of this fixpoint by a function called fix with the following type:

```
fix : {n : ℕ} → (s : ASTStm n) → (e : ASTExp n) → (pc : 𝕊) → (Γ₀ : Vec 𝕊 n) →
  Σ (Vec 𝕊 n) (λ Γ → Γ ≈≈ body e s pc Γ₀ Γ)
```

where ≈≈ is the equality between type environments. Function fix is defined in terms of fixS, where function body plays the role of function g. Function body implements the body of the fixpoint construction:

```
body : ASTExp n → -- the condition
       ASTStm n → -- while's body
       𝕊  →   -- program counter's security level
       Vec 𝕊 n → -- initial environment Γ
       Vec 𝕊 n → -- environment Γ'ᵢ
       Vec 𝕊 n   -- computed environment Γ'ᵢ₊₁

body e s pc Γ Γ' =  let  st = tyExp Γ' e
                         Γ" = proj₁ (tyStm s (pc ∨ st) Γ')
                    in (Γ" ⊔ Γ)
```

Function body is defined in terms of function tyStm, a functional implementation of the type system, which computes a post-environment and a proof that it is indeed the environment that results from the type system.

```
tyStm : (s : ASTStm n) → (pc : 𝕊) → (Γ : Vec 𝕊 n) → Σ (Vec 𝕊 n) (λ Γ' → TyStm s pc Γ Γ')
tyStm SKIP pc Γ = Γ , Skip
tyStm (ASSIGN x e) pc Γ = change x Γ (pc ∨ (tyExp Γ e)) , Ass
tyStm (SEQ s s') pc Γ =    let (Γ' , tcs)    = tyStm s pc Γ
                               (Γ" , tcs') = tyStm s' pc Γ'
                           in  Γ" , Seq tcs tcs'
tyStm (IF0 e s s') pc Γ =  let pc' = pc ∨ (tyExp Γ e)
                               (Γ' , tcs)    = tyStm s pc' Γ
                               (Γ" , tcs') = tyStm s' pc' Γ
                           in  Γ' ⊔ Γ" , If0 tcs tcs'
tyStm {n} (WHILE e s) pc Γ = proj₁ (fix s e pc Γ) , While
```

The partial order relation between environments (_⊑_) is given by the pointwise ordering between the corresponding vectors:

```
data _⊑_ : Vec 𝕊 n → Vec 𝕊 n → Set ℓ₂ where
```

$$\sqsubseteq\text{-nil} \;:\; [] \sqsubseteq []$$
$$\sqsubseteq\text{-cons} : \forall\, \{st\ st' : \mathbb{S}\}\ \{\Gamma\ \Gamma' : \mathsf{Vec}\ \mathbb{S}\ n\} \to \Gamma \sqsubseteq \Gamma' \to st \leq st' \to (st :: \Gamma) \sqsubseteq (st' :: \Gamma')$$

Together with the join operation ($\sqcup$), the partial order of environments ($\sqsubseteq$) turns out to form a join semilattice.

Function body turns out to be monotone. Its monotonocity requires the monotonicity of tyStm, which in turn requires that fix is monotone. In fact, body, tyStm and fix, so as their monotonicity proofs are mutually recursive.

$$\begin{aligned}&\mathsf{bodyMonotone} : \{e : \mathsf{ASTExp}\ n\}\{s : \mathsf{ASTStm}\ n\}\ \{pc\ pc' : \mathbb{S}\}\{\Gamma\ \Gamma'\ \Gamma_1\ \Gamma_1' : \mathsf{Vec}\ \mathbb{S}\ n\} \to \\ &\quad pc \leq \mathsf{St}\ pc' \to \Gamma \sqsubseteq \Gamma_1 \to \Gamma' \sqsubseteq \Gamma_1' \to \mathsf{body}\ e\ s\ pc\ \Gamma\ \Gamma' \sqsubseteq \mathsf{body}\ e\ s\ pc'\ \Gamma_1\ \Gamma_1'\end{aligned}$$

$$\begin{aligned}&\mathsf{tcMonotone} : \{pc\ pc' : \mathbb{S}\}\{\Gamma\ \Gamma_1 : \mathsf{Vec}\ \mathbb{S}\ n\} \\ &\quad (s : \mathsf{ASTStm}\ n) \to \\ &\quad pc \leq \mathsf{St}\ pc' \to \Gamma \sqsubseteq \Gamma_1 \to \mathsf{tyStm}\ s\ pc\ \Gamma \sqsubseteq \mathsf{tyStm}\ s\ pc'\ \Gamma_1\end{aligned}$$

$$\begin{aligned}&\mathsf{fixMonotone} : \{pc\ pc' : \mathbb{S}\}\{\Gamma\ \Gamma' : \mathsf{Vec}\ \mathbb{S}\ n\} \to \\ &\quad (s : \mathsf{ASTStm}\ n) \to (e : \mathsf{ASTExp}\ n) \to \\ &\quad pc \leq \mathsf{St}\ pc' \to \Gamma \sqsubseteq \Gamma' \to \mathsf{proj}_1\ (\mathsf{fix}\ s\ e\ pc\ \Gamma) \sqsubseteq \mathsf{proj}_1\ (\mathsf{fix}\ s\ e\ pc'\ \Gamma')\end{aligned}$$

In order to define fix in terms of fixS we need that the semilattice of environments posesses a *bound* function. We use function sumDist for that end. Given an environment, sumDist returns the sum of the distances to the top type ($\top$) that have the types of the variables in the environment. For that we need to assume that the lattice of security types also has associated a bound function that we call $\sharp$.

$$\begin{aligned}&\mathsf{sumDist} : \{n : \mathbb{N}\} \to \mathsf{Vec}\ \mathbb{S}\ n \to \mathbb{N} \\ &\mathsf{sumDist}\ [] = 0 \\ &\mathsf{sumDist}\ (st :: \Gamma) = \sharp\ st + \mathsf{sumDist}\ \Gamma\end{aligned}$$

sumDist fits well as a bound function because in each step of the fixpoint computation the security type of each variable in the computed environment is possibly higher. Therefore, if at least one of the security types change to a higher type then the global distance to the top type decreases. This function also has a unique minimal value because only the top environment (that with all variables with the highest security type) has global distance equal zero.

$$\mathsf{sumDistDecr} : \{\Gamma\ \Gamma' : \mathsf{Vec}\ \mathbb{S}\ n\} \to \Gamma \sqsubset \Gamma' \to \mathsf{sumDist}\ \Gamma' <\mathbb{N}\ \mathsf{sumDist}\ \Gamma$$

$$\mathsf{minimalEnv} : \{\Gamma\ \Gamma' : \mathsf{Vec}\ \mathbb{S}\ n\} \to \mathsf{sumDist}\ \Gamma \equiv 0 \to \mathsf{sumDist}\ \Gamma' \equiv 0 \to \Gamma \equiv \Gamma'$$

Finally, we have all the elements to define fix.

$$\begin{aligned}&\mathsf{fix} : \quad (s : \mathsf{ASTStm}\ n) \to \\ &\qquad (e : \mathsf{ASTExp}\ n) \to \\ &\qquad (pc : \mathbb{S}) \to\end{aligned}$$

```
      (Γ : Vec 𝕊 n) →
      Σ (Vec 𝕊 n) (λ Γ' → Γ' ≈≈ body e s pc Γ Γ')
fix s e pc Γ =
  let  Γ₀ = Γ
       Γ₁ = body e s pc Γ Γ₀
  in fixS  (n * ♯ ⊥)
           (body e s pc Γ) -- function g
           (bodyMonotone {e = e} {s = s} reflℕ≤ (reflℕ⊑ {Γ = Γ}))
           Γ₀
           Γ⊑Γ'⊔Γ
           (initInv Γ₀)
```

where function initInv is the proof that for any environment Γ, sumDist $\Gamma \leq n$ * ♯ $\bot$. This corresponds to the initial state of the invariant that fixS requires to the iteration boundary, which is initialized in n * ♯ $\bot$ (the global distance to the top of the bottom environment).

The decorated type StmS is indexed by a program context and the pre- and post- environment. The constructors SkipS, AssignS, SeqS and IfS are direct implementations of the typing rules. This contrasts to the WhileS constructor, which needs to deal with undecorated ASTs to compute the fixpoint.

```
data StmS : Vec 𝕊 n → 𝕊 → Vec 𝕊 n → Set c where

AssignS :  {Γ : Vec 𝕊 n}{pc st : 𝕊} →
           (x : Fin n) →
           ExpS Γ st →
           StmS Γ pc (change x Γ (pc ∨ st))
SkipS   :  {Γ : Vec 𝕊 n}{pc : 𝕊} →
           StmS Γ pc Γ
SeqS    :  {Γ Γ' Γ" : Vec 𝕊 n}{pc : 𝕊} →
           StmS Γ pc Γ' →
           StmS Γ' pc Γ" →
           StmS Γ pc Γ"
IfS     :  {pc st : 𝕊}{Γ Γ' Γ" : Vec 𝕊 n} →
           ExpS Γ st →
           StmS Γ (pc ∨ st) Γ' →
           StmS Γ (pc ∨ st) Γ" →
           StmS Γ pc (Γ' ⊔ Γ")
WhileS  :  {Γ : Vec 𝕊 n}{pc : 𝕊}
           (e : ASTExp n) →
           (s : ASTStm n ) →
           StmS Γ pc (proj₁ (fix s e pc Γ))
```

Finally, the following lifting function summarizes the internalization process.

```
liftS : {Γ Γ' : Vec 𝕊 n}{pc : 𝕊} (s : ASTStm n) → TyStm s pc Γ Γ' → StmS Γ pc Γ'
```

$$\frac{}{pc \vdash \Gamma\ \{\texttt{skip} \rightsquigarrow \texttt{skip}\}\ \Gamma} \qquad \frac{\Gamma \vdash E : t \qquad s = pc \vee t}{pc \vdash \Gamma\ \{x := E \rightsquigarrow x_s := E_\Gamma\}\ \Gamma[x \mapsto s]}$$

$$\frac{pc \vdash \Gamma\ \{S_1 \rightsquigarrow D_1\}\ \Gamma' \qquad pc \vdash \Gamma'\ \{S_2 \rightsquigarrow D_2\}\ \Gamma''}{pc \vdash \Gamma\ \{S_1;S_2 \rightsquigarrow D_1;D_2\}\ \Gamma''}$$

$$\frac{\Gamma \vdash E : t \qquad pc \vee t \vdash \Gamma\ \{S_1 \rightsquigarrow D_1\}\ \Gamma_1' \qquad pc \vee t \vdash \Gamma\ \{S_2 \rightsquigarrow D_2\}\ \Gamma_2' \qquad \Gamma' = \Gamma_1' \sqcup \Gamma_2'}{pc \vdash \Gamma\ \{\texttt{if}\ E\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \rightsquigarrow \texttt{if}\ E_\Gamma\ \texttt{then}\ (D_1\ ;\ \Gamma' := \Gamma_1')\ \texttt{else}\ (D_2\ ;\ \Gamma' := \Gamma_2')\}\ \Gamma'}$$

$$\frac{\Gamma_i' \vdash E : t_i \qquad pc \vee t_i \vdash \Gamma_i'\ \{S \rightsquigarrow D_i\}\ \Gamma_i'' \qquad 0 \le i \le n}{pc \vdash \Gamma\ \{\texttt{while}\ E\ \texttt{do}\ S \rightsquigarrow \Gamma_n' := \Gamma\ ;\texttt{while}\ E_{\Gamma_n'}\ \texttt{do}\ (D_n\ ;\ \Gamma_n'' := \Gamma_n')\}\ \Gamma_n'}$$
$$\Gamma_0' = \Gamma, \Gamma_{i+1}' = \Gamma_i'' \sqcup \Gamma, \Gamma_{n+1}' = \Gamma_n'$$

Figure 5. Translation rules

```
liftS SKIP          Skip          = SkipS
liftS (ASSIGN x e)  Ass           = AssignS x (liftE e)
liftS (SEQ s1 s2)   (Seq t1 t2)   = SeqS (liftS s1 t1) (liftS s2 t2)
liftS (IF0 e s1 s2) (If0 t1 t2)   = IfS   (liftE e) (liftS s1 t1) (liftS s2 t2)
liftS (WHILE e s)   While         = WhileS e s
```

4 Translation to flow-insensitive

Now we turn to the formalization of Hunt and Sands's translation in Agda. It converts programs typable in the flow-sensitive type system to equivalent programs typable in the flow-insensitive type system. The transformation rules, shown in Figure 5, are defined as an extension to the flow-sensitive type system and are expressed in terms of the judgement $pc \vdash \Gamma\ \{S \rightsquigarrow D\}\ \Gamma'$, where as before pc is the security context, and Γ and Γ' are type environments. S is a statement with floating-type variables, which is typable in the flow-sensitive type system; D is an equivalent statement (or statements) produced by the program translation, with fixed-type variables and typable in the flow-insensitive type system.

To transform a program typable in the flow-sensitive system to another typable in the flow-insensitive system we need to transform floating-type variables into fixed-type variables. The set of fixed-type variables of a program, $FVar$, is obtained from the set of floating variables, Var, by annotating each variable name with a security type:

$$FVar = \{x_t \mid x \in Var,\ t \in \mathcal{L}\}$$

Each time a floating-type variable x raises its security type from t to s, with $t < s$, the translation will reflect this fact by constructing an assignment that moves information from x_t to x_s. Therefore, the transformed code may include a sequence of variable assignments (between annotated variables) that make explicit the variables that changed their security level. We write $\Gamma := \Gamma'$ to represent an

appropriate sequentialisation of the following set of variable assignments:

$$\{x_s := x_t \mid \Gamma(x) = s, \Gamma'(x) = t, s \neq t\}$$

A sequence of assignments $\Gamma := \Gamma'$ is used with environments Γ, Γ' such that $\Gamma' \sqsubseteq \Gamma$. This gives rise to well-typed assignments as information flows in the appropriate direction. The security context in which a sequence of assigments must type is the minimun security type of the assigned variables. We define a funtion min that calculates that context:

```
min: (Γ : Vec 𝕊 n ) → (Γ' : Vec 𝕊 n ) → 𝕊
min [] [] = ⊤
min (st :: Γ) (st' :: Γ') with st ≟ st'
... | yes st≈st' = min Γ Γ'
... | no st≉st' = st ∧ (min Γ Γ')
```

where $\stackrel{?}{=}$ denotes the decidability of equality in the set $\mathbb{S}$ of security types. A sequence of assignments has then the following type:

```
Γ:=Γ' : (Γ : Vec 𝕊 n) → (Γ' : Vec 𝕊 n) → Γ' ⊑ Γ → Stm ( min Γ Γ' )
```

This function generates a sequence of assignments by traversing the environments Γ and Γ'. For each i it adds an assigment $\Gamma(i) := \Gamma'(i)$ if $\Gamma(i) \sqsupset \Gamma'(i)$.

Let us now see the translation function. Given an environment Γ, every expression E with floating-type variables can be transformed to an expression E_Γ with fixed-type variables by replacing each floating-type variable x by a fixed-type variable x_s, where $s = \Gamma(x)$. The following function implements the transformation, where Exp is the type of expressions presented in Section 2.

```
transExp : { st : 𝕊}{Γ : Vec 𝕊 n} → ExpS Γ st → Exp st
transExp (IntValS y) = IntVal y
transExp {Γ = Γ} (VarS x) = Var x (lookup x Γ)
transExp (AddS y y') = Add (transExp y) (transExp y')
```

The translation for statements follows the rules shown in Figure 5:

```
translate : {pc : 𝕊} {Γ Γ' : Vec 𝕊 n} → StmS Γ pc Γ' → Stm pc
translate {pc = pc} {Γ = Γ } (AssignS {st = st} x e) =
  let x≤x∨y , x≤y∨x , _ = supremum pc st
  in Assign x (pc ∨ st) x≤y∨x    x≤x∨y (transExp e)
translate SkipS = Skip

translate (SeqS s s') = Seq (translate s) (translate s')

translate (IfS {pc} {st} {Γ} {Γ₁'} {Γ₂'} e s s') =
  let    ass₁ = Γ:=Γ' (Γ₁' ⊔ Γ₂') Γ₁' Γ⊑Γ⊔Γ'
         ass₂ = Γ:=Γ' (Γ₁' ⊔ Γ₂') Γ₂' Γ⊑Γ'⊔Γ
         p≼m : (pc ∨ st) ≼ min (Γ₁' ⊔ Γ₂') Γ₁'
         p≼m = propMin' s s'
         ass₁' = lemmaA2A {pc' = pc ∨ st} ass₁ p≼m    -- Γ' := Γ₁'
         p≼m₂ : (pc ∨ st) ≼ min (Γ₁' ⊔ Γ₂') Γ₂'
```

```
        p≼m₂ = propMin'' s s'
        ass₂' = lemmaA2A {pc' = pc ∨ st} ass₂ p≼m₂    -- Γ' := Γ₂'
    in If0 (transExp e) (Seq (translate s) ass₁') (Seq (translate s') ass₂')

translate (WhileS {Γ} {pc}      e s)
  =   let   Γn'     =   proj₁ (fix s e pc Γ)
            stn' , eΓn'   =   fromAstE Γn' e
            Γn'' , prf    =   tyStm s (pc ∨ stn') Γn'
            dn      =   translateASTStm   (pc ∨ stn') Γn' s
            ass     =   lemmaA2A    {pc' = pc}    ------------- Γn' := Γ
                                    (Γ:=Γ' Γn' Γ (Γ⊑Γn' {Γ = Γ} e s ))
                                    (lemmaA2C {pc = pc}
                                    (WhileS {Γ = Γ} {pc = pc} e s))

            ass₂    =   lemmaA2A    {pc' = pc ∨ stn'} -------------- Γn' := Γn''
                                    (Γ:=Γ' Γn'' Γn' (Γn'⊑Γn'' e s) )
                                    (lemmaA2C    {pc = pc ∨ stn'} (liftS s prf) )
        in Seq ass (While (transExp eΓn') (Seq dn ass₂))
```

Function translateASTStm is like translate, but it works on undecorated ASTs for statements:

$$\text{translateASTStm} : (pc : \text{A})\ (\Gamma : \text{Vec A } n) \to \text{ASTStm } n \to \text{Stm } pc$$

The functions lemmaA2A and lemmaA2C are the formalization of the lemmas given in the paper [4] as part of proof of the static soundness property for the translation. Here we show just the type of these functions:

$$\text{lemmaA2A} : \{\ pc\ pc' : \mathbb{S}\} \to \text{Stm } pc \to pc' \preccurlyeq pc \to \text{Stm } pc'$$

$$\text{lemmaA2C} : \{pc : \mathbb{S}\}\{\Gamma\ \Gamma' : \text{Vec } \mathbb{S}\ n\} \to \text{StmS } \Gamma\ pc\ \Gamma' \to pc \preccurlyeq (\text{min } \Gamma'\ \Gamma\ )$$

It is worth noticing that our translate not only implements the desired translation but it also ensures that the statement that results from the translation is typable in the flow-insensitive type system. That is, we are using Agda's type system to enforce the preservation of non-interference during translation. In other words, translate can be understood as an implementation of the following theorem, where $[pc] \vdash D$ is the typing judgement for statements presented in Section 2.

Theorem 4.1 ([4]) *If* $pc \vdash \Gamma\ \{S \rightsquigarrow D\}\ \Gamma'$ *then* $[pc] \vdash D$.

5 Conclusions

We presented the formalization of Hunt and Sands's translation for high-level secure programs. translation uses a type-based approach to noninterference to converts programs in a While language, typable in a flow-sensitive type system, into programs in the same language typable in a flow-insensitive type system. Agda was our formalization framework; we used it both as a functional language and as a proof assistant.

In both versions of the language we introduced an internalist, typed representations of the abstract syntax both for expressions and statements. In the target

version of the language, the internalist representation was the result of a direct implementation of the typing rules. In the source version of the language, however, it was necessary to combine the internalist representation with an externalist one in order to deal with the computation of a fixpoint.

A distinguishing feature of our Agda formalization was the systematic use we did of typed representations of the abstract syntax for the two versions of the object language. This was possible thanks to the syntax-directed formulations of the security type systems and the translation relation. As a result of this encoding, only terms corresponding to non-interfering programs can be written in the Agda implementations of the language. This has also consequences on the functions we can define between those typed representations. In particular, this happens with the implementation of Hunt and Sands's translation. In our formalization we defined the translation as a function that preserves well-typed representations of abstract syntax terms. The equations of the translation function then correspond to the proof steps of the preservation property. The verification that the proof is correct is then performed by Agda's type-checker.

References

[1] Bove, A. and P. Dybjer, *Dependent types at work*, in: A. Bove, L. S. Barbosa, A. Pardo and J. S. Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, Lecture Notes in Computer Science **5520** (2008), pp. 57–99.
URL https://doi.org/10.1007/978-3-642-03153-3_2

[2] Denning, D. E., *A lattice model of secure information flow*, Commun. ACM **19** (1976), pp. 236–243.
URL http://doi.acm.org/10.1145/360051.360056

[3] Goguen, J. A. and J. Meseguer, *Security policies and security models*, in: *Symposium on Security and Privacy* (1982), pp. 11–20.

[4] Hunt, S. and D. Sands, *On flow-sensitive security types*, SIGPLAN Not. **41** (2006), pp. 79–90.
URL http://doi.acm.org/10.1145/1111320.1111045

[5] Hunt, S. and D. Sands, *From exponential to polynomial-time security typing via principal types*, in: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, 2011, pp. 297–316.
URL https://doi.org/10.1007/978-3-642-19718-5_16

[6] Manzino, C., "Security preserving program translations," Master's thesis, PEDECIBA Informática, Universidad de la República, Uruguay (2018).

[7] Manzino, C. and A. Pardo, *A Security Types Preserving Compiler in Haskell*, in: F. M. Q. Pereira, editor, *Proceedings of the 18th Brazilian Symposium on Programming Languages - SBLP 2014, Maceio, Brazil, October 2-3, 2014.*, Lecture Notes in Computer Science **8771** (2014), pp. 16–30.
URL https://doi.org/10.1007/978-3-319-11863-5_2

[8] Nipkow, T. and G. Klein, "Concrete Semantics: With Isabelle/HOL," Springer Publishing Company, Incorporated, 2014.

[9] Norell, U., *Dependently typed programming in Agda*, in: *4th international workshop on Types in Language Design and Implementation*, TLDI '09 (2009), pp. 1–2.
URL http://doi.acm.org/10.1145/1481861.1481862

[10] Pardo, A., E. Gunther, M. Pagano and M. Viera, *An internalist approach to correct-by-construction compilers*, in: D. Sabel and P. Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018* (2018), pp. 17:1–17:12.
URL http://doi.acm.org/10.1145/3236950.3236965

[11] Pasalic, E. and N. Linger, *Meta-programming with typed object-language representations*, in: *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, 2004, pp. 136–167.

[12] Poulsen, C. B., A. Rouvoet, A. Tolmach, R. Krebbers and E. Visser, *Intrinsically-typed definitional interpreters for imperative languages*, Proc. ACM Program. Lang. **2** (2018), pp. 16:1–16:34.
URL https://doi.org/10.1145/3158104

[13] Russo, A. and A. Sabelfeld, *Dynamic vs. static flow-sensitive security analysis*, in: *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, pp. 186–199.
URL https://doi.org/10.1109/CSF.2010.20

[14] Sabelfeld, A. and A. C. Myers, *Language-based information-flow security*, IEEE J. Selected Areas in Communications **21** (2003), pp. 5–19.

[15] Sheard, T., *Languages of the future*, SIGPLAN Not. **39** (2004), pp. 119–132.
URL http://doi.acm.org/10.1145/1052883.1052897

[16] Volpano, D. M. and G. Smith, *A type-based approach to program security*, in: *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97 (1997), pp. 607–621.
URL http://dl.acm.org/citation.cfm?id=646620.697712