PEDECIBA Informática
Universidad de la República
Uruguay

Tesis de Maestría en Informática

Security preserving program translations

Cecilia Manzino

Director Académico y Director de Tesis:
Alberto Pardo
Instituto de Computación
Universidad de la República
Uruguay

2018

Tribunal:

Dr. Miguel Pagano (revisor)
Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
Argentina

Dra. Nora Szasz
Facultad de Ingeniería
Universidad ORT Uruguay

Dr. Carlos Luna
Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Abstract

The analysis of information flow has become 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 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. In such a type system, if a program type-checks, then it means that it meets the security policy.

In this thesis we focus on an important usage of the non-interference property: its preservation through program translation. We are interested in analysing techniques that make it possible the development of security-preserving program translations in the sense of code conversions that produce non-interfering output programs out of non-interfering input programs. This is a topic with significant practical relevance as can be seen in, for example, the context of program compilation: if for certain applications it is essential that the source code meets the security property, it is even more important that the corresponding compiled, low-level code, which is the one that will be actually executed, is also secure.

We pursue a formal methods approach to this topic, performing an analysis of type-based, security-preserving program translations in the context of dependently-typed programming. We use Agda, a functional language with dependent types, as the formalization language. In Agda we represent the (abstract syntax of the) object languages, their security type systems, as well as the translations between them. The importance of using Agda resides in its powerful type system that makes it possible to encode object invariants. In our case this is reflected in the ability to define the security type systems of the involved languages in terms of Agda's inductive families thus reducing the verification of security preservation by translation to type-checking.

We analyse the formalization of two cases. First, we develop a compiler between a simple imperative language and a semi-structured machine code. For each language, we define a sound information flow type system and we prove that the compiler preserves non-interference. The type systems of both languages are flow-insensitive in the sense that the security level of program

variables is not allowed to change during program execution. Second, we perform the formalization of Hund & Sands security-preserving translation that transforms programs in a high-level language typable in a flow-sensitive type system into equivalent high-level programs typable in a flow-insensitive type system. Since the source language of the compiler coincides with the target language of Hund & Sands translation, by composing the two components we get as result a security-preserving compiler for a language with a flow-sensitive type system.

Contents

Chapter 1

Introduction

The confidentiality of the information manipulated by computing systems has become of significant importance with the increasing use of applications through internet. Such applications usually give access to sensitive data and the critical part is then to assure that they do not leak confidential information to unautorized third parties.

Traditional security mechanisms like access control (used to prevent unauthorized access to information) and cryptography (used to exchange data across a non-secure channel) do not provide end-to-end protection of data. For example, with access control, once the application is authorized to access certain information there is no control of how that information is used or if it is propagated. Similarly, with cryptography, once data is decrypted there is no guarantee about its use.

To complement these security mechanisms, information flow policies come up into play for establishing guarantees that private data cannot be inferred by inspecting public data. Among many proposed ones, *information flow analysis* [3, 20] is a widely used technique that examines information flows between inputs and outputs of systems.

Non-interference [7] is an example of an information flow policy. A system is said to enjoy this property if any variation of confidential data causes no variation of public data. Non-interference is a semantic condition, with the nice property that it can be enforced statically by the definition of an information flow type system that guarantees the absence of illicit information flows during program execution [6, 22, 20]. 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* [20]. 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* [9, 17]. 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 thesis we focus on the study of a relevant application of the non-interference property: its preservation through program translation. Our interest is the analysis of program translations that preserve this property in the sense that they deliver non-interfering programs when they are provided with non-interfering programs as input. This is a topic with practical relevance, especially in the context of program compilation: if it is important that the source code of an application can be proved secure, it is even more important that the target code generated by compilation satisfies to be secure as well.

We adopt a formal approach to this topic. In this sense, we develop the formalization of type-based, security-preserving program translations in Agda [15, 4], a dependently-typed functional language developed at Chalmers University. We formalize in Agda the abstract syntax of the different object languages, their security type systems, as well as the translation between them. The importance of using Agda resides in its powerful type system which enables us the encoding of object invariants. In our case this is reflected in the features Agda provides us to define typed representations of abstract syntax terms (ASTs) of the involved object languages, following the approach of Sheard [21] and Pasalic and Linger [16]. These are terms that simulaneously represent ASTs and (their) typing rules in the type system. An interesting consequence of this representation is that it restricts the kind of terms of the object languages that are representable. In fact, 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 provide evidences that they respect the typing encoded in the manipulated typed terms are accepted. The positive aspect of this fact is that the verification that this is indeed satisfied by a function is reduced to perform type-checking in Agda. In this thesis we take advantage of this technique to verify that certain program translations are security preserving.

In concrete, we analyse two program translations. First, we develop a compiler between a simple imperative language and low-level code of a stack machine. After defining flow-insensitive, security type systems for both languages, which we prove sound with respect to the semantic definition of non-interference, we build corresponding typed representation of the ASTs as Agda GADTs. We then write the compiler as a function that preserves security typing when translating programs between the GADTs of the languages. Second, we formalize Hund & Sands [9] security-preserving translation in Agda. This is an interesting security-preserving translation that transforms high-level programs typable in a flow-sensitive type system into equivalent programs typable in a flow-insensitive type system. Interestingly, the target language of this translation coincides with the source language of the compiler described above. Thus, by composing the two components we get as result a security-preserving compiler for programs typeable in a flow-sensitive type system.

1.1 Organization of the thesis

The rest of the thesis is organized in three chapters.

- In Chapter 2 we present a security-preserving compiler from a simple imperative high-level language to low-level code.

  The chapter is a revised and extended version of the paper "A security types preserving compiler in Haskel" presented at the XVIII Brazilian Symposium of Programming Languages (SBLP 2014). In contrast to this thesis, where we use Agda, in the paper the development of the compiler is performed in Haskell.

  The chapter improves the mentioned paper in two main aspects: it presents the soundness proof of the involved security type systems, and it corrects some bugs in the formulation of the security type system of the low-level language with respect to the one showed in the paper.

- In Chapter 3 we present the formalization in Agda of Hund & Sands' security-preserving translation [9] which transforms secure programs typeable in a flow-sensitive type system to secure programs typeable in a flow-insensitive type system.

- Chapter 4 concludes the thesis with a summary of the contributions and with the description of some future work.

Chapters 2 and 3 contain only excrepts of the Agda code. The complete formalization is available at `www.fceia.unr.edu.ar/˜ceciliam/Tesis/codes`.

Chapter 2

A simple security types preserving compiler

In this chapter we develop in Agda a correct-by-construction compiler that preserves the non-interference property. The compiler translates programs written in a simple imperative language that includes loops and conditionals to programs in a stack-based low-level language. We define the notion of non-interference associated to each of these languages and introduce type systems tailored to check this security policy statically.

By representing the security type systems in terms of Agda GADTs we are then able to write the compiler as a function that translates terms between GADTs and has expressed in its type the security-preservation property. Working with such a representation of the type systems has the benefit that the proof of security-preservation is performed automatically by Agda's type system.

The chapter is organized as follows. In Section 2.1 we present the source language of the compiler and show different versions of security type systems for it. We also describe the encoding of a syntax-directed version of the security type system as a generalized algebraic data type (GADT) in Agda. Section 2.2 decribes the target language of the compiler. As for the source language, we define versions of security type systems for this language and represent a syntax-directed version in terms of a GADT. Section 2.3 presents the compiler and a proof that it preserves security typing.

2.1 Source Language

In this section, we introduce the high level language that is used as source language in our compiler. We start by describing its abstract syntax. Then we define its semantics and versions of type systems used to enforce secure information flow within programs. Finally, we show an implementation both of the syntax and type system in Agda.

2.1.1 Syntax

The source language we consider is a simple imperative language with expressions and sentences defined by the following abstract syntax:

$$\begin{array}{lcl} e & ::= & n \mid x \mid e_1 + e_2 \\ S & ::= & x := e \mid \texttt{skip} \mid S_1;S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ e\ \texttt{do}\ S \end{array}$$

where $e \in \mathbf{Exp}$ and $S \in \mathbf{Stm}$. Variables range over identifiers ($x \in \mathbf{Var}$) whereas n ranges over integer literals ($n \in \mathbf{Num}$).

2.1.2 Big-step semantics

We present a semantics of the language which is completely standard [13]. In the semantics, the meaning of both expressions and statements is given relative to a state $s \in \mathbf{State} = \mathbf{Var} \to \mathbb{Z}$, a mapping from variables to integer values that contains the current value of each variable.

The semantics of expressions is given in terms of an evaluation function $\mathcal{E} : \mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ defined by induction on the structure of expressions:

$$\begin{array}{lcl} \mathcal{E}[\![n]\!]\,s & = & \mathcal{N}[\![n]\!] \\ \mathcal{E}[\![x]\!]\,s & = & s\,x \\ \mathcal{E}[\![e_1 + e_2]\!]\,s & = & \mathcal{E}[\![e_1]\!]\,s + \mathcal{E}[\![e_2]\!]\,s \end{array}$$

where $\mathcal{N} : \mathbf{Num} \to \mathbb{Z}$ is a function that associates an integer value to each integer literal.

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 2.1.

Notice that the language does not contain boolean expressions. In fact, the condition of an `if` as well as a `while` statement is given by an arithmetic expression. According to the semantics, the condition of an `if` statement is true when it evaluates to zero, and false otherwise. The same happens with the condition of a `while`.

2.1.3 Security Type System

We assume that each variable has associated a security level, which states the degree of confidentiality of the values it stores. A type environment $\Gamma : \mathbf{Var} \to \mathbf{SType}$ maps each variable to a security type.

For simplicity, in this thesis we consider just two security levels, low and $high$, corresponding to *public* and *confidential* data, respectively, but the whole development can be generalized to a lattice of security levels ordered by their degree of confidentiality. As usual $low \leq high$.

We assume that the security level of each variable is maintained unchanged during program execution. A language with this characteristic is said to be *flow*

$$\langle x := e,\, s\rangle \Downarrow s[x \mapsto \mathcal{E}[\![e]\!]\, s] \qquad \langle \mathtt{skip},\, s\rangle \Downarrow s$$

$$\frac{\langle S_1,\, s\rangle \Downarrow s' \quad \langle S_2,\, s'\rangle \Downarrow s''}{\langle S_1;S_2,\, s\rangle \Downarrow s''}$$

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

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

Figure 2.1: Big-step semantics of statements

insensitive. We write x_L (x_H) to mean a variable with low ($high$) security level and refer to it as a low (high) variable.

Non-interference is a property on programs that guarantees the absence of illicit information flows during execution. A program satisfies this security property when the final value of the low variables is not influenced by a variation of the initial value of the high variables. This property can be formulated in terms of the semantics of the language. Let us say that two states s and s' are L-equivalent, written $s \cong_L s'$, when they coincide in the low variables, i.e., $s\ x_L = s'\ x_L$ for every low variable $x_L \in \mathbf{Var}$. In other words, L-equivalent states are indistinguishable to a low observer (i.e. to an observer that can only inspect public data).

Definition 2.1 (Non-interference source language)**.** *A program $S \in \mathbf{Stm}$ is **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:*

$$\mathbf{NI}_{\mathbb{S}}(S) \overset{\text{df}}{=} \forall s_i, s_i'.\, s_i \cong_L s_i' \ \wedge\ \langle S,\, s_i\rangle \Downarrow s_f \ \wedge\ \langle S,\, s_i'\rangle \Downarrow s_f' \implies s_f \cong_L s_f'$$

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

Nowadays it is well-known that non-interference can be enforced statically by the definition of an information-flow type system in which security levels are used as types and referred to as *security types*. This started with the work of Volpano et al. [22, 23]. Figures 2.2 and 2.3 present two alternative security type systems for our source language. The difference between them is that the system in Figure 2.3 is syntax-directed.

Expressions For typing expressions in the system of Figure 2.2 we use a judgement of the form $\vdash e \ :\ st$, where $st \in \{low, high\}$. Rule EXPH states that

Expressions

$$\frac{}{\vdash e \ : \ high}\ \textsc{exp}H \qquad \frac{x_H \notin Vars(e)}{\vdash e \ : \ low}\ \textsc{exp}L$$

Statements

$$\frac{\vdash e \ : \ low}{[low] \vdash x_L := e}\ \textsc{ass}L \qquad \frac{}{[pc] \vdash x_H := e}\ \textsc{ass}H$$

$$\frac{}{[pc] \vdash \texttt{skip}}\ \textsc{skip} \qquad \frac{[pc] \vdash S_1 \quad [pc] \vdash S_2}{[pc] \vdash S_1;S_2}\ \textsc{seq}$$

$$\frac{\vdash e \ : \ pc \quad [pc] \vdash S_1 \quad [pc] \vdash S_2}{[pc] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\ \textsc{if}$$

$$\frac{\vdash e \ : \ pc \quad [pc] \vdash S}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S}\ \textsc{while} \qquad \frac{[high] \vdash S}{[low] \vdash S}\ \textsc{sub}$$

Figure 2.2: Security Type System with subsumption (Source Language)

any expression can have type *high*. On the contrary, rule expL specifies that the only expressions that can have type *low* are those that do not contain *high* variables.

The type system for expressions shown in Figure 2.3 is syntax-directed. It uses a judgement of the form $\vdash_{sd} e : st$. According to this system, the security type of an expression is the maximum of the security types of its variables. We denote by $\texttt{max}\ st\ st'$ the maximum of two security types st and st'. Integer numerals are considered public data.

The rules for expressions remain unchanged if, instead of simply two security types, we considered a generalization in which the security level of variables belong to a lattice of security levels.

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

An *explicit flow* is observed when confidential data are copied to public variables. For example, the following assignment is not allowed because the value of a high variable is copied to a low variable.

$$y_L := x_H + 1$$

On the other hand, an assignment like the following is authorized, since copying the content of a low variable to high variable does not represent a security violation.

$$x_H := y_L$$

EXPRESSIONS

$$\vdash_{sd} n : low \qquad \vdash_{sd} x : \Gamma(x) \qquad \frac{\vdash_{sd} e : st \quad \vdash_{sd} e' : st'}{\vdash_{sd} e + e' : \mathtt{max}\ st\ st'}$$

STATEMENTS

$$\frac{\vdash_{sd} e : st \quad st \leq \Gamma(x) \quad pc \leq \Gamma(x)}{[pc] \vdash_{sd} x := e}\ \text{ASS}_{sd} \qquad \frac{[pc] \vdash_{sd} S_1 \quad [pc] \vdash_{sd} S_2}{[pc] \vdash_{sd} S_1; S_2}\ \text{SEQ}_{sd}$$

$$\frac{\vdash_{sd} e : st \quad [\mathtt{max}\ st\ pc] \vdash_{sd} S_1 \quad [\mathtt{max}\ st\ pc] \vdash_{sd} S_2}{[pc] \vdash_{sd} \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2}\ \text{IF}_{sd}$$

$$\frac{\vdash_{sd} e : st \quad [\mathtt{max}\ st\ pc] \vdash_{sd} S}{[pc] \vdash_{sd} \mathtt{while}\ e\ \mathtt{do}\ S}\ \text{WHILE}_{sd} \qquad [pc] \vdash_{sd} \mathtt{skip}\ \ \text{SKIP}_{sd}$$

Figure 2.3: Syntax-directed Security Type System (Source Language)

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:

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

The reason for being insecure is because by observing the value of the low variable y_L on different executions we can infer information about the value of the high variable x_H. This is a consequence of the assignment of a low variable in a branch of a conditional upon a high variable. Due to situations like this 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:

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

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

The type system for statements shown in Figure 2.2 is based on similar systems given in [22, 20]. In that system, the typing judgement has the form $[pc] \vdash S$ and means that statement S is typable in the security context pc. Observe that the system includes a *subsumption rule* (SUB) which states that if a statement is typeable in a high context then it is also typeable in low context. As a consequence of this rule the system is not syntax-directed.

We then reformulate the type system to turn it syntax-directed. This gives rise to the system shown in Figure 2.3. The typing judgement in this new system has the form $[pc] \vdash_{sd} S$.

Rule ASS$_{sd}$ states that assignments to low variables can only be done in *low* context while assignments to high variables can be performed in any context. Explicit flows are prevented by this rule by the restriction $st \leq \Gamma(x)$.

The rules IF$_{sd}$ and WHILE$_{sd}$ impose a restriction between the security level of the condition and the branches (of the conditional) or the body (of the while). As a consequence, if the condition is high then the branches (of the conditional) or the body (of the while) must type in *high* contexts, since $\texttt{max}\ high\ pc = high$, for any pc. This restriction prevents implicit flows.

Consider the following code that contains an implicit flow from a high variable to a low variable:

$$\texttt{if}\ x_H\ \texttt{then}\ x_L := 1\ \texttt{else}\ x_L := 2$$

This code is clearly insecure because from inspecting the value of the variable x_L we can infer if x_H was equal, or not, to 0. The restriction of the rule that is violated by this statement is the one that states that the security level of the branches of a conditional must be high if the condition is high; that restriction thus indicates that only high variables can be updated in the branches, something not satisfied by the statement of the example.

The following theorem establishes the relationship between both type systems.

Theorem 2.2. *For every $e \in \mathbf{Exp}$, $S \in \mathbf{Stm}$, and $st, pc \in \mathbf{SType}$:*

(i) If $\vdash_{sd} e : st$ then $\vdash e\ :\ st$.

(ii) If $\vdash e\ :\ st$ then there exists st' such that $\vdash_{sd} e : st'$ and $st' \leq st$.

(iii) If $[pc] \vdash_{sd} S$ then $[pc] \vdash S$.

(iv) If $[pc] \vdash S$ then there exists pc' such that $[pc'] \vdash_{sd} S$ and $pc \leq pc'$.

Proof. See 5.1.

□

The type systems presented in Figures 2.2 and 2.3 are called top-down systems. Another way of defining the security type system is by using a bottom-up formulation, where the program counter of the commands is determined by choosing the greatest pc that is consistent with variable assignments and by adding restrictions in the rules for conditionals and loops.

A bottom-up type system for statements is shown in Figure 2.4. In that system, the typing judgement has the form $[pc] \vdash' S$ and means that statement S is typable in the security context pc.

Observe that this type system includes a subsumption rule. We also reformulate this type system to turn it syntax-directed. This gives rise to the system shown in Figure 2.5. The typing judgement in this system has the form $[pc] \vdash'_{sd} S$.

The following theorem establishes the relationship between the bottom-up type system with subtyping and the top-down formulation of the syntax-directed type system.

$$\frac{\vdash_{sd} e : st \quad st \leq \Gamma(x) \quad pc \leq \Gamma(x)}{[pc] \vdash' x := e}\ \textsc{ass'} \qquad \frac{[pc_1] \vdash' S_1 \quad [pc_2] \vdash' S_2}{[\mathtt{min}\ pc_1\ pc_2] \vdash' S_1;S_2}\ \textsc{seq'}$$

$$\frac{\vdash_{sd} e : st \quad [pc_1] \vdash' S_1 \quad [pc_2] \vdash' S_2 \quad st \leq \mathtt{min}\ pc_1\ pc_2}{[\mathtt{min}\ pc_1\ pc_2] \vdash' \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2}\ \textsc{if'}$$

$$\frac{\vdash_{sd} e : st \quad [pc] \vdash' S \quad st \leq pc}{[pc] \vdash' \mathtt{while}\ e\ \mathtt{do}\ S}\ \textsc{while'} \qquad [pc] \vdash' \mathtt{skip}\ \ \textsc{skip'}$$

$$\frac{[high] \vdash' c}{[low] \vdash' c}\ \textsc{sub'}$$

Figure 2.4: Bottom Up Security Type system with Subtyping of the Source Language

$$\frac{\vdash_{sd} e : st \quad st \leq \Gamma(x) \quad pc \leq \Gamma(x)}{[pc] \vdash'_{sd} x := e}\ \textsc{ass'}_{sd} \qquad \frac{[pc_1] \vdash'_{sd} S_1 \quad [pc_2] \vdash'_{sd} S_2}{[\mathtt{min}\ pc_1\ pc_2] \vdash'_{sd} S_1;S_2}\ \textsc{seq'}_{sd}$$

$$\frac{\vdash_{sd} e : st \quad [pc_1] \vdash'_{sd} S_1 \quad [pc_2] \vdash'_{sd} S_2 \quad st \leq \mathtt{min}\ pc_1\ pc_2}{[\mathtt{min}\ pc_1\ pc_2] \vdash'_{sd} \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2}\ \textsc{if'}_{sd}$$

$$\frac{\vdash_{sd} e : st \quad [pc] \vdash'_{sd} S \quad st \leq pc}{[pc] \vdash'_{sd} \mathtt{while}\ e\ \mathtt{do}\ S}\ \textsc{while'}_{sd} \qquad [high] \vdash'_{sd} \mathtt{skip}\ \ \textsc{skip'}_{sd}$$

Figure 2.5: Syntax-directed, Bottom Up Security Type System of the Source Language

Theorem 2.3. *For every $S \in \mathbf{Stm}$, and $pc \in \mathbf{SType}$:*

i) If $[pc] \vdash' S$ then $[pc] \vdash_{sd} S$

ii) If $[pc] \vdash_{sd} S$ then $[pc] \vdash' S$

Proof. See 5.2. □

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. The first one, called *confinement*, states that if a sentence is typable in a high context then the execution of the sentence does not alter the value of the low variables.

Lemma 2.4 (Confinement)**.**

$$[high] \vdash_{sd} S \ \wedge\ \langle S, s\rangle \Downarrow s' \implies s \cong_L s'$$

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 states that the evaluation of an expression of type low (i.e. an expression that does not contain high variables) is the same in L-equivalent states.

Lemma 2.5.

$$\vdash_{sd} e : low \;\wedge\; s \cong_L s' \implies \mathcal{E}[\![e]\!]\, s = \mathcal{E}[\![e]\!]\, s'$$

Proof. Straightforward by induction on the derivation of $\vdash_{sd} e : low$. □

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

Theorem 2.6 (Type soundness). *For every $S \in \mathbf{Stm}$ and $pc \in \mathbf{SType}$,*

$$[pc] \vdash_{sd} S \implies \mathbf{NI}_S(S)$$

Proof. See 5.3. □

2.1.4 Implementation

We represent in Agda the terms of the language together with the typing rules of Figure 2.3 using generalized algebraic data types (GADTs).

The security types low and $high$ are represented by the following datatype definition:

```
data St : Set where
  Low : St
  High : St
```

The expressions of the language are then represented by a datatype Exp, indexed by a value of type St that corresponds to the security type of the expression:

```
data Exp : St → Set where
  IntVal : ℕ → Exp Low
  Var    : {st : St} → VarT st →    Exp st
  Add    : {st st' : St} → Exp st → Exp st' → Exp (st ∪ st')
```

where _∪_ : St → St → St is the function that calculates the maximun between two security types and VarT is the type to represent typed variables:

```
data VarT : St → Set where
  VarL : ℕ → VarT Low
  VarH : ℕ → VarT High
```

To model the statements of the language we define the following datatype, indexed by a security type representing the security level of the program point in which the statement is executed.

```
data Stm : St → Set where
  Skip     : {pc : St} → Stm pc

  Assign   : {st st' pc : St} →
             st' ≤St st → pc ≤St st → VarT st → Exp st' → Stm pc

  Seq      : {pc : St} → Stm pc → Stm pc → Stm pc

  If0      : {st pc : St} →
             Exp st → Stm (st ∪ pc) → Stm (st ∪ pc) → Stm pc

  While    : {st pc : St} →
             Exp st → Stm (st ∪ pc) → Stm pc
```

The constructors Skip, Assign, Seq, If0 and While are direct implementations of the rules for statements of the type system in Figure 2.3. Now, the typing judgement *S* : Stm *pc* corresponds to the judgement $[pc] \vdash_{sd} S$ in our formal type system.

2.2 Target Language

The target language of the compiler is a low-level code of a stack machine in the style of the presented in [13]. In this section we describe its syntax and operational semantics and define a type system that guarantees non-interference.

2.2.1 Syntax

The instructions of the target language are given by the following abstract syntax:

c	::=	$push\ n$	pushes the value n on top of the stack
	|	add	addition operation
	|	$fetch\ x$	pushes the value of variable x onto the stack
	|	$store\ x$	stores the top of the stack in variable x
	|	$noop$	no operation
	|	$c_1\,;c_2$	code sequence
	|	$branch\ (c_1, c_2)$	conditional
	|	$loop\ (c_1, c_2)$	looping

where $c \in \mathbf{Code}$, $x \in \mathbf{Var}$ and $n \in \mathbf{Num}$. Like the source language, this language also manipulates program variables that have associated a security level. As usual in this kind of low-level languages, values are placed in an evaluation stack in order to be used by certain operations.

$$\langle push\ n,\ \sigma,\ s\rangle \rhd (\mathcal{N}[\![n]\!] : \sigma,\ s) \qquad \langle add,\ z_1 : z_2 : \sigma,\ s\rangle \rhd ((z_1 + z_2) : \sigma,\ s)$$

$$\langle fetch\ x,\ \sigma,\ s\rangle \rhd ((s\ x) : \sigma,\ s) \qquad \langle store\ x,\ z : \sigma,\ s\rangle \rhd (\sigma, s[x \mapsto z])$$

$$\langle noop,\ \sigma,\ s\rangle \rhd (\sigma,\ s)$$

$$\frac{\langle c_1,\ \sigma,\ s\rangle \rhd (\sigma', s')}{\langle c_1 ; c_2,\ \sigma,\ s\rangle \rhd \langle c_2,\ \sigma',\ s'\rangle} \qquad \frac{\langle c_1,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle}{\langle c_1 ; c_2,\ \sigma,\ s\rangle \rhd \langle c' ; c_2,\ \sigma',\ s'\rangle}$$

$$\frac{z = 0}{\langle branch\ (c_1, c_2),\ z : \sigma,\ s\rangle \rhd \langle c_1,\ \sigma,\ s\rangle} \qquad \frac{z \neq 0}{\langle branch\ (c_1, c_2),\ z : \sigma,\ s\rangle \rhd \langle c_2,\ \sigma,\ s\rangle}$$

$$\langle loop\ (c_1, c_2),\ \sigma,\ s\rangle \rhd \langle c_1 ; branch\ (c_2 ; loop\ (c_1, c_2), noop),\ \sigma,\ s\rangle$$

Figure 2.6: Operational Semantics of the Target Language

2.2.2 Operational semantics

A code c is executed on an abstract machine with configurations of the form $\langle c, \sigma, s\rangle$ or (σ, s), where σ is an evaluation stack and $s \in State$ is a state that associates values to variables. The operational semantics is given by a transition relation between configurations that specifies an individual execution step. The transition relation is of the form $\langle c,\ \sigma,\ s\rangle \rhd \gamma$, where γ may be either a new configuration $\langle c',\ \sigma',\ s'\rangle$, expressing that remaining execution steps still need to be performed, or a final configuration (σ', s'), expressing that the execution of c terminates in one step. As usual, we write $\langle c,\ \sigma,\ s\rangle \rhd^* \gamma$ to indicate that there is a finite number of steps in the execution from $\langle c,\ \sigma,\ s\rangle$ to γ. The operational semantics of the language is shown in Figure 2.6.

We can define a *meaning relation* $\langle c, s\rangle \downarrow s'$ *iff* $\langle c,\ \epsilon,\ s\rangle \rhd^* (\sigma', s')$ which states that a given code c, and states s and s' are in the relation whenever the execution of c starting in s and the empty stack ϵ terminates with state s'. It can be proved that this is in fact a partial function as our semantics is deterministic. Based on this relation, we can define the notion of non-interference for low-level programs:

$$\mathbf{NI}_T(c) \overset{\mathrm{df}}{=} \forall s_i, s_i'. s_i \cong_L s_i' \ \wedge\ \langle c, s_i\rangle \downarrow s_f \ \wedge\ \langle c, s_i'\rangle \downarrow s_f' \implies s_f \cong_L s_f'$$

Like it happened for the source language, this definition of non-interference is *termination-insensitive*.

2.2.3 Security Type System

The security type system of the target language is shown in Figure 2.7. It is defined in terms of a transition relation that relates a program code with the security level of the program counter and the state of the *stack type* (stack of security types) before and after the execution of that code. The typing

$$\textsc{push}\quad ls \vdash push\ n : pc \rightsquigarrow low :: ls$$

$$\textsc{add}\quad st_1 :: st_2 :: ls \vdash add : pc \rightsquigarrow \mathtt{max}\ st_1\ st_2 :: ls$$

$$\textsc{fetch}\quad ls \vdash fetch\ x : pc \rightsquigarrow \Gamma(x) :: ls$$

$$\textsc{store}\ \frac{st \leq \Gamma(x) \qquad pc \leq \Gamma(x)}{st :: ls \vdash store\ x : pc \rightsquigarrow ls}$$

$$\textsc{noop}\quad ls \vdash noop : pc \rightsquigarrow ls$$

$$\textsc{cseq}\ \frac{ls \vdash c_1 : pc \rightsquigarrow ls' \qquad ls' \vdash c_2 : pc \rightsquigarrow ls''}{ls \vdash c_1 ; c_2 : pc \rightsquigarrow ls''}$$

$$\textsc{branch}\ \frac{\mathcal{B}\ (st,\ ls,\ c_1,\ c_2,\ pc, ls')}{st :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls'}$$

$$\textsc{B}_1\ \frac{ls \vdash c_1 : pc \rightsquigarrow ls' \qquad ls \vdash c_2 : pc \rightsquigarrow ls'}{\mathcal{B}\ (low,\ ls,\ c_1,\ c_2,\ pc, ls')}$$

$$\textsc{B}_2\ \frac{[\,] \vdash c_1 : high \rightsquigarrow [\,] \qquad [\,] \vdash c_2 : high \rightsquigarrow [\,]}{\mathcal{B}\ (high,\ ls,\ c_1,\ c_2,\ pc, ls)}$$

$$\textsc{loop}\ \frac{\mathcal{L}\ (ls,\ c_1,\ c_2,\ pc,\ st,\ ls')}{ls \vdash loop\ (c_1, c_2) : pc \rightsquigarrow ls'}$$

$$\textsc{L}_1\ \frac{ls \vdash c_1 : pc \rightsquigarrow low :: ls' \qquad ls' \vdash c_2 : pc \rightsquigarrow ls''}{\mathcal{L}\ (ls,\ c_1,\ c_2,\ pc,\ low,\ ls'')}$$

$$\textsc{L}_2\ \frac{[\,] \vdash c_1 : pc \rightsquigarrow high :: [\,] \qquad [\,] \vdash c_2 : high \rightsquigarrow [\,]}{\mathcal{L}\ (ls,\ c_1,\ c_2,\ pc,\ high,\ ls)}$$

Figure 2.7: Security Type System for the Target Language

judgement is then of the form $ls \vdash c : pc \rightsquigarrow ls'$, where ls and ls' are stack types. This judgement states that a program c is typable when, starting in the security environment given by the stack type ls and with program counter pc, it ends up with stack type ls'. This type system is syntax-directed.

Like for the source language, this type system was designed in order to prevent both explicit and implicit illegal flows. Rule STORE, for example, prevents explicit flows by requiring that the value to be stored in a variable low has also security level low, while the requirement on the context prevents implicit flows.

Rules BRANCH and LOOP also take care of implicit flows. Rule BRANCH uses an auxiliary relation $\mathcal{B}$. Depending on the security type of the value at the top of the stack (value used to choose the branch to continue) relation $\mathcal{B}$ requires a different typing to the codes at the branches. If this value is low, then the codes at the branches type in any context and the stack type can be changed (in both branches to the same stack type). On the other hand, if the value at the top is of type high, then the codes at the branches can only be typed in a high context and are not allowed to use the stack. This is enforced by typing the branches with the empty stack.

To see some examples of the kind of cases we want to avoid associated with *branch*, let us consider the following codes:

$fetch\ x_H;$
$branch\ (push\ 1\,;store\ x_L, push\ 2\,;store\ x_L)$

$fetch\ x_H;$
$branch\ (push\ 0, push\ 3);$
$store\ x_L$

Although in these cases we are storing a low value in a low variable (an action, in principle, completely legal), the reason for rejecting these codes is because the low value to store depends on the value of the high variable x_H and therefore, as a result, we are implicitly informing in x_L the status of x_H (i.e. whether it is, or not, zero).

In the code on the right hand side, we have a situation that is not easy to detect because the instruction $store\ x_L$ occurs outside the conditional and its action depends on the code that comes before it. This is a problem originated by the way the storage of a value in a variable is performed in this language. In fact, to do so at least two independent actions are required: one (or many) for computing and setting (in the stack) the value to be stored, and another for storing that value in the variable.

Therefore, in order to reject this kind of implicit flows, we must enforce that if the value at the top of the stack, used by a conditional to choose the branch, is high, then the branches of that conditional are not allowed to change the stack of security types. Roughly speaking, what we are stating could be expressed by a rule like this:

$$\frac{ls \vdash c_1 : high \rightsquigarrow ls \qquad ls \vdash c_2 : high \rightsquigarrow ls}{high :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls}$$

However, a rule like the one above is not enough to guarantee noninterference, since an insecure code like the following would be accepted:

$push\ 1;$
$fetch\ x_H;$
$branch\ (store\ x_H\,;push\ 2, store\ x_H\,;push\ 3);$
$store\ x_L$

This code is insecure because after its execution the variable x_L contains the value 2 or 3 depending on whether x_H is, or not, zero. However, it passes the rule above because both branches of the conditional change the starting stack type during their execution, but deliver the starting stack type at the end. The origin of the insecurity is the fact that the codes of the branches were able both to use and modify values contained in the original stack and finish with the same stack type. That way, it was possible to replace a value in the stack with another one (of same security type) that depended on the branch taken (and hence, on a high value). Therefore, when a conditional is over a high value we must prevent the codes of the branches to change the values contained in the starting stack as well as to deliver a stack different from the starting one. To enforce this requirement we require the branches of the conditional to be typed in the empty stack. In some sense, this is a stronger condition, because doing so we are preventing the branches to even use the values contained in the starting stack. This is stated by relation $\mathcal{B}$ in rule B$_2$.

We could have written two rules for the conditional directly, instead of defining the auxiliary relation $\mathcal{B}$, but in that case the type system would not continue being syntax-directed.

A similar situation happens with rule LOOP, now using the auxiliary relation $\mathcal{L}$. By similar considerations to those given for conditionals, when in a loop $loop\ (c_1, c_2)$ the condition c_1 puts a high value at the top of the stack, then the body c_2 is required to be typed in the empty stack.

We note that this type system rejects some secure programs. For example, the following program is not accepted:

$$\begin{array}{l} push\ 1; \\ branch\ (push\ 0, noop) \end{array}$$

because of the restriction of rule B$_1$, which states that the branches of a conditional must deliver the same stack type.

In Section 2.3, we show that the programs (secure or not) that are rejected by these restrictions of the type system are not the ones generated by compilation.

Figure 2.8 shows an alternative, bottom-up security type system with subtyping for the low-level language.

The following theorem establishes the relationship between the bottom-up type system and the top-down formulation of Figure 2.7.

Theorem 2.7. *For every $c \in \mathbf{Code}$, pc, ls and ls':*

i) $ls \vdash_b c : pc \rightsquigarrow ls' \Rightarrow ls \vdash c : pc \rightsquigarrow ls'$

ii) $ls \vdash c : pc \rightsquigarrow ls' \Rightarrow ls \vdash_b c : pc \rightsquigarrow ls'$

Proof. See 5.4. □

Now, let us analyze type soundness for the type system shown in Figure 2.7. It requires the proof of some previous properties.

Definition 2.8. *Two evaluation stacks σ and σ' with same stack type ls are said to be L-*equivalent*, written $\sigma \cong_L \sigma'$, when they coincide in the low values, i.e., for every position i, if $ls[i] = low$, then $\sigma[i] = \sigma'[i]$.*

One of the properties we need is *preservation*, which states that typing is preserved through reduction.

$$\textsf{PUSH}\quad ls \vdash_b push\ n : pc \rightsquigarrow low :: ls$$

$$\textsf{ADD}\quad st_1 :: st_2 :: ls \vdash_b add : pc \rightsquigarrow \texttt{max}\ st1\ st2 :: ls$$

$$\textsf{FETCH}\quad ls \vdash_b fetch\ x : pc \rightsquigarrow \Gamma(x) :: ls$$

$$\textsf{STORE}\ \frac{st \leq \Gamma(x) \qquad pc \leq \Gamma(x)}{st :: ls \vdash_b store\ x : pc \rightsquigarrow ls}$$

$$\textsf{NOOP}\quad ls \vdash_b noop : pc \rightsquigarrow ls$$

$$\textsf{CSEQ}\ \frac{ls \vdash_b c_1 : pc_1 \rightsquigarrow ls' \qquad ls' \vdash_b c_2 : pc_2 \rightsquigarrow ls''}{ls \vdash c_1 ; c_2 : \texttt{min}\ pc_1\ pc_2 \rightsquigarrow ls''}$$

$$\textsf{BRANCH}\ \frac{\mathcal{B}\ (st,\ ls,\ c_1,\ c_2,\ pc, ls')}{st :: ls \vdash_b branch\ (c_1, c_2) : pc \rightsquigarrow ls'}$$

$$\textsf{B1}_b\ \frac{ls \vdash_b c_1 : pc_1 \rightsquigarrow ls' \qquad ls \vdash_b c_2 : pc_2 \rightsquigarrow ls'}{\mathcal{B}\ (low,\ ls,\ c_1,\ c_2,\ \texttt{min}\ pc_1\ pc_2, ls')}$$

$$\textsf{B2}_b\ \frac{[\,] \vdash_b c_1 : high \rightsquigarrow [\,] \qquad [\,] \vdash_b c_2 : high \rightsquigarrow [\,]}{\mathcal{B}\ (high,\ ls,\ c_1,\ c_2,\ pc, ls)}$$

$$\textsf{LOOP}_b\ \frac{\mathcal{L}\ (ls,\ c_1,\ c_2,\ pc,\ st,\ ls')}{ls \vdash_b loop\ (c_1, c_2) : pc \rightsquigarrow ls'}$$

$$\textsf{L1}_b\ \frac{ls \vdash_b c_1 : pc_1 \rightsquigarrow low :: ls' \qquad ls' \vdash_b c_2 : pc_2 \rightsquigarrow ls''}{\mathcal{L}\ (ls,\ c_1,\ c_2,\ \texttt{min}\ pc_1\ pc_2,\ low,\ ls'')}$$

$$\textsf{L2}_b\ \frac{[\,] \vdash_b c_1 : pc \rightsquigarrow high :: [\,] \qquad [\,] \vdash_b c_2 : high \rightsquigarrow [\,]}{\mathcal{L}\ (ls,\ c_1,\ c_2,\ pc,\ high,\ ls)}$$

$$\textsf{SUB}\ \frac{ls \vdash_b c : high \rightsquigarrow ls'}{ls \vdash_b c : low \rightsquigarrow ls'}$$

Figure 2.8: Bottom Up Security Type System for the Target Language with Subtyping

Theorem 2.9 (Preservation).

$$ls \vdash c : pc \rightsquigarrow ls' \;\wedge\; \langle c,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle \;\Rightarrow\; \exists ls''.\ ls'' \vdash c' : pc \rightsquigarrow ls'$$

Proof. By induction on a derivation of $\langle c,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle$. See 5.5. □

Another property that is necessary is *confinement*, which states that a code executed in a high context does not alter low variables.

Lemma 2.10. *For every* $c \in \mathbf{Code}$, ls, ls', σ, *and* s *such that*

$$ls \vdash c : high \rightsquigarrow ls'$$

then either

$$\exists\, \sigma', s', c'.\langle c,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle \;\wedge\; s \cong_L s'$$

or

$$\exists\, \sigma', s'.\langle c,\ \sigma,\ s\rangle \rhd (\sigma',\ s') \;\wedge\; s \cong_L s'$$

Proof. See 5.6 □

Theorem 2.11 (Confinement). *For every* $c \in \mathbf{Code}$, ls, ls', σ, *and* s,

$$ls \vdash c : high \rightsquigarrow ls' \;\wedge\; \exists\, \sigma'\ s'.\langle c,\ \sigma,\ s\rangle \rhd^* (\sigma',\ s') \;\Rightarrow\; s \cong_L s'$$

Proof. By using *preservation* and Lemma 2.10. □

Finally, we state the following property: if a typable code is executed in L-equivalent stacks of values and stores, then after n steps L-equivalent stacks and stores are reached.

Lemma 2.12. *For every* $c \in \mathbf{Code}$, ls, ls', s_1, s_2, σ_1, σ_2, pc, *such that* $\sigma_1 \cong_L \sigma_2$, $s_1 \cong_L s_2$ *and* $ls \vdash c : pc \rightsquigarrow ls'$, *then either*

$$\begin{aligned}\exists c', s_1', s_2', \sigma_1', \sigma_2'.\ &\langle c,\ \sigma_1,\ s_1\rangle \rhd^* \langle c',\ \sigma_1',\ s_1'\rangle \wedge\\ &\langle c,\ \sigma_2,\ s_2\rangle \rhd^* \langle c',\ \sigma_2',\ s_2'\rangle \wedge\\ &\sigma_1' \cong_L \sigma_2' \;\wedge\; s_1' \cong_L s_2'\end{aligned}$$

or

$$\begin{aligned}\exists s_1', s_2', \sigma_1', \sigma_2'.\ &\langle c,\ \sigma_1,\ s_1\rangle \rhd^* (\sigma_1',\ s_1') \wedge\\ &\langle c,\ \sigma_2,\ s_2\rangle \rhd^* (\sigma_2',\ s_2') \wedge\\ &\sigma_1' \cong_L \sigma_2' \;\wedge\; s_1' \cong_L s_2'\end{aligned}$$

Proof. By structural induction on the code c. See 5.7. □

Now, we can prove soundness of the security type system. Soundness states that every low-level code typable in the security type system is non-interfering.

Theorem 2.13 (Type soundness). *For every* $c \in \mathbf{Code}$, ls, ls' *and* pc,

$$ls \vdash c : pc \rightsquigarrow ls' \implies \mathbf{NI}_T(c)$$

Proof. The proof is a consequence of Lemma 2.12. □

2.2.4 Implementation

The low-level language is encoded by the datatype Code, indexed by the security level of the context and the stack types (represented as lists) before and after the execution of the code.

```
mutual
  data Code : List St → St → List St → Set where

    Push : {Σ : List St}{pc : St} →
      ℕ → Code Σ pc (Low :: Σ)

    Fetch : {Σ : List St}{st pc : St} →
      VarT st → Code Σ pc (st :: Σ)

    Store : {Σ : List St}{st st' pc : St} →
      st' ≤St st →
      pc ≤St st →
      VarT st    →
      Code (st' :: Σ) pc Σ

    Sum : {Σ : List St}{pc st st' : St} →
      Code (st :: st' :: Σ) pc (st' ∪ st :: Σ)

    App : {Σ Σ₁ Σ₂ : List St}{pc : St} →
      Code Σ pc Σ₁ →
      Code Σ₁ pc Σ₂ →
      Code Σ pc Σ₂

    Branch : {Σ Σ' : List St}{st pc : St}    →
      B st Σ pc Σ' →
      Code (st :: Σ) pc Σ'

    Loop : {Σ Σ' : List St}{pc st : St} →
      L Σ pc st Σ' →
      Code Σ pc Σ'

    Noop : {Σ : List St}{pc : St} →
      Code Σ pc Σ


  data B : St → List St → St → List St → Set where
    BLow : {Σ Σ' : List St}{pc : St} →
      Code Σ pc Σ' →
      Code Σ pc Σ' →
      B Low Σ pc Σ'

    BHigh : {Σ : List St}{pc : St} →
      Code [] High [] →
      Code [] High [] →
```

```
        B High Σ pc Σ

data L : List St → St → St → List St → Set where
  LLow : {Σ Σ1 Σ2 : List St}{pc : St} →
    Code Σ pc (Low :: Σ1 ) →
    Code Σ1 pc Σ2 →
    L Σ pc Low Σ2

  LHigh : {Σ : List St}{pc : St} →
    Code [] pc (High :: []) →
    Code [] High [] →
    L Σ pc High Σ
```

The typing judgement *c* : Code Σ *pc* Σ' corresponds to the judgement $ls \vdash c : pc \rightsquigarrow ls'$ and the data types B and L correspond to the relations $\mathcal{B}$ and $\mathcal{L}$ of the formal type system.

2.3 Compiler

The compiler is a function that converts terms of the source language into terms of the target language. Since the terms of our source language are of two syntax categories, we have to define two compilation functions, one for expressions ($\boldsymbol{C}_e : \mathbf{Exp} \rightarrow \mathbf{Code}$) and the other for statements ($\boldsymbol{C}_S : \mathbf{Stm} \rightarrow \mathbf{Code}$). Figure 2.9 shows the definition of both functions.

It is not difficult to prove that this compiler is correct with respect to the semantics of the source and target languages.

Theorem 2.14 (COMPILER CORRECTNESS)**.** *For any expression e, statement S of the source language, and state s it holds that:*

i) $\langle \boldsymbol{C}_e[e],\ \epsilon,\ s\rangle \rhd^* (\mathcal{E}[\![e]\!]s, s)$

ii) if $\langle S,\ s\rangle \Downarrow s'$ *then* $\langle \boldsymbol{C}_S[S],\ \epsilon,\ s\rangle \rhd^* (\epsilon, s')$

Expressions

$\boldsymbol{C}_e[n] = push\ n$

$\boldsymbol{C}_e[x] = fetch\ x$

$\boldsymbol{C}_e[e_1 + e_2] = \boldsymbol{C}_e[e_1]\ ;\boldsymbol{C}_e[e_2]\ ; add$

Sentences

$\boldsymbol{C}_S[x := e] = \boldsymbol{C}_e[e]\ ; store\ x$

$\boldsymbol{C}_S[\texttt{skip}] = noop$

$\boldsymbol{C}_S[S_1;S_2] = \boldsymbol{C}_S[S_1]\ ; \boldsymbol{C}_S[S_2]$

$\boldsymbol{C}_S[\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2]$
$\quad = \boldsymbol{C}_e[e]\ ; branch(\boldsymbol{C}_S[S_1], \boldsymbol{C}_S[S_2])$

$\boldsymbol{C}_S[\texttt{while}\ e\ \texttt{do}\ S] = loop(\boldsymbol{C}_e[e], \boldsymbol{C}_S[S])$

Figure 2.9: Compilation functions

In this work we are especially interested in another property of the compiler, namely the preservation of non-interference through compilation. Preservation in this context means that the compiler converts non-interfering programs in the source language into non-interfering programs in the target language. This property can be established semantically.

Theorem 2.15 (SEMANTIC-BASED PRESERVATION OF NONINTERFERENCE)**.** *For any* $e \in \mathbf{Exp}$ *and* $S \in \mathbf{Stm}$*,*

i) $\mathbf{NI}_T(\boldsymbol{C}_e[e])$

ii) if $\mathbf{NI}_S(S)$ *then* $\mathbf{NI}_T(\boldsymbol{C}_S[S])$

Our interest, however, is to establish the property at the type level.

Theorem 2.16 (TYPE-BASED PRESERVATION OF NONINTERFERENCE)**.** *For any* $e \in \mathbf{Exp}$ *and* $S \in \mathbf{Stm}$*,*

i) If $\vdash_{sd} e : st$ *then for any* ls *and* pc*,* $ls \vdash \boldsymbol{C}_e[e] : pc \rightsquigarrow st :: ls$

ii) If $[pc] \vdash_{sd} S$ *then for any* ls*,* $ls \vdash \boldsymbol{C}_S[S] : pc \rightsquigarrow ls$

Proof. The proof of *i)* proceeds by induction on the structure of the expression e. We analyse the code that returns the compiler in each case and construct a derivation tree for this code using the type system for the target language. The proof of *ii)* proceeds similarly by induction on the structure of statement S. □

2.3.1 Implementation

The compiler function $\boldsymbol{C}_e$ can be easily implemented in Agda as follows:

```
compileExp : {st pc : St}{Σ : List St} →
  Exp st → Code Σ pc (st :: Σ)

compileExp (IntVal n) = Push n

compileExp (Var    x) = Fetch    x

compileExp (Add e₁ e₂) = App (App (compileExp e₁) (compileExp e₂)) Sum
```

and $\boldsymbol{C}_S$ can be implemented as:

```
compiler : {pc : St} → Stm pc → Code [] pc []

compiler (Assign st'≤st pc≤st    var e) =
  App (compileExp e) (Store st'≤st pc≤st var)

compiler (Seq c₁ c₂) = App (compiler c₁) (compiler c₂)
compiler (If0 e c₁ c₂ ) =
   branch (compileExp e) (compiler c₁ ) (compiler c₂ )

compiler (While e c) = loop (compileExp e) (compiler c)

compiler Skip = Noop
```

where the functions branch and loop are defined as follows:

```
branch : {st pc : St} →
   Code [] pc (st :: []) →   -- result of the compilation of the guard
   Code [] (st ∪ pc) [] →   -- result of compilation of the first branch
   Code [] (st ∪ pc) [] →   -- result of compilation of the second branch
   Code [] pc []

branch {st = Low} ce c₁ c₂ = App ce (Branch (BLow c₁ c₂ ))

branch {st = High} ce c₁ c₂ = App ce (Branch (BHigh c₁ c₂ ))


loop : {st pc : St} →
   Code [] pc (st :: []) →   -- result of compilation of the guard
   Code [] (st ∪ pc) [] →   -- result of compilation of the sentence
   Code [] pc []

loop {st = Low} ce c = Loop (LLow ce c)

loop {st = High} ce c = Loop (LHigh ce c)
```

These functions turn out to be more than simply compilation functions. They are actually the Agda representation of the proof terms of Theorem 2.16. In fact, observe that the type of these functions is exactly the encoding of the properties *i)* and *ii)* in that theorem. In other words, when writing these functions we are actually proving the theorem and the preservation of noninterference is being verified by Agda's type system. As we mentioned above, both *i)* and *ii)* are proved by structural induction. The equations of the compiler functions encode the cases of the inductive proofs.

2.4 Summary

In this chapter we developed a simple compiler in Agda that preserves non-interference. The compiler takes programs in an imperative high-level language and returns code in a low-level language that runs in a stack-based abstract machine. A security type system was defined for checking non-interference on each language. Each type system was proved sound with respect to the semantic definition of non-interference and implemented in Agda in terms of data families. Interestingly, the fact of having syntax-directed formulations of the type systems produces that the Agda representations at the same time encode the AST constructors of the languages together with their typing rules. As a consequence of that, the encoding enforces that we can only write non-interfering programs in the Agda implementation of the languages.

Because of this approach the type of the compiler then corresponds exactly to the formulation of the preservation of non-interference through compilation. The definition of the compiler itself then corresponds to the definition of the proof terms that prove the preservation property. The rest of the job (i.e. the verification that the function is indeed a proof of the property) is performed by Agda's type system.

Chapter 3

Translation from flow-sensitive to flow-insensitive

Most security type systems are flow-insensitive. One of the reasons is that they are more intuitive than flow-sensitive ones. Flow-insensitive type systems do not take into account the order of excecution. This means that the security of a program remains unaltered if we, for example, reorder the statements in a sequence. In a flow-insensitive type system, like the ones presented in the previous chapter, the security level of the program variables is fixed from the beginning. In contrast, in a flow-sensitive type system the security level of the variables may be different at different program points. Flow-sensitive type systems use to be more permissive, accepting more secure programs than the flow-insensitive ones.

For example, consider following code,

$$\begin{array}{l} x := y; \\ x := 0 \end{array}$$

where x is a low variable and y is a high variable. Although in the second assignment the variable x is overridden with the constant 0, this code is rejected by a flow-insensitive type system because it has an insecure subprogram (the instruction $x := y$). This is, however, accepted by a flow-sensitive type system (e.g., the one presented in [9]) because the security level of the variable y is relabeled to low after the first assignment.

Another example of an intuitively secure code that is rejected by a flow-insensitive type system is the following, where x and y have the same security level as above:

$$\begin{array}{l} y := 0; \\ \textbf{if } y \textbf{ then } x := 1 \textbf{ else } x := 2 \end{array}$$

Hunt and Sands [9] show that any program in a While language which is typable in a flow-sensitive type system can be translated to an equivalent program that is typable in a simple flow-insensitive type system. The translation is formulated as an extension of the flow-sensitive type system.

$$\Gamma \vdash n \ : \ low \qquad \Gamma \vdash x \ : \ \Gamma(x) \qquad \frac{\Gamma \vdash e \ : \ st \quad \Gamma \vdash e' \ : \ st'}{\Gamma \vdash e + e' \ : \ st \sqcup st'}$$

Figure 3.1: Flow-Sensitive Typing Rules for expressions

The aim of this chapter is to show a formalization of that translation in Agda. As source and target language of the translation we consider the same While language presented in the previous chapter. We encode the abstract syntax of the While language and both security type systems (the flow-sensitive and the flow-insensitive one) in terms of inductive families. We then implement the translation as a function between such inductive families. Like in the previous chapter, by writing this function in Agda we are proving that the translation preserves non-interference. Again, the validation of that preservation is being performed by Agda's type checker.

3.1 Flow-Sensitive Type System

In a flow-sensitive language, the security type of a variable can change during program execution.

We use the flow-sensitive type system given by Hunt and Sand [9]. It is defined for a lattice with two elements (*low* and *high*). In this type system the judgement for sentences is 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', which describe the security level of the variables before and after the execution of S, respectively (we call them pre and post environment). The judgement for expressions has the form:

$$\Gamma \vdash e \ : \ t$$

meaning that e has type t in the type environment Γ.

The typing rules for expressions and commands are shown in Figures 3.1 and 3.2.

For any expression e, we say that e is typable in an environment Γ iff its type is the least upper bound of the types of its free variables. The rules shown in Figure 3.1 are similar to the rules of Figure 2.3.

Rule Assign states that an assignment is typable in any context. The type of the variable x in the post-environment can change to high if the assigment is performed in a high context or if the assigned expression is high. It is changed to low if the context and the type of the expression are low. In other cases, the post-environment doesn't change.

The rules If and While were designed to prevent implicit flows. If the condition is high then the branches (of the conditional) or the body (of the while) must be typable in a high context.

$$\textsc{Skip}\ \frac{}{pc \vdash \Gamma \ \{\ \mathtt{skip}\ \}\ \Gamma'} \qquad \textsc{Assign}\ \frac{\Gamma \vdash e \ :\ t}{pc \vdash \Gamma \ \{\ x := e\ \}\ \Gamma[x \mapsto pc \sqcup t]}$$

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

$$\textsc{While}\ \frac{\Gamma \vdash e \ :\ t \quad pc \sqcup t \vdash \Gamma \ \{\ S\ \}\ \Gamma}{pc \vdash \Gamma \ \{\ \mathtt{while}\ e\ \mathtt{do}\ S\ \}\ \Gamma}$$

$$\textsc{Sub}\ \frac{pc_1 \vdash \Gamma_1 \ \{\ S\ \}\ \Gamma_1'}{pc_2 \vdash \Gamma_2 \ \{\ S\ \}\ \Gamma_2'} \qquad pc_2 \leq pc_1,\ \Gamma_2 \sqsubseteq \Gamma_1,\ \Gamma_1' \sqsubseteq \Gamma_2'$$

Figure 3.2: Flow-Sensitive Typing Rules for commands

The rule Sub states that if a program is typable in a high context, pre-environment Γ_1 and post-environment Γ_1', then it is also typable in a low context, and in pre-environments that are smaller than Γ_1 and post-environment greater than Γ_1'. So, the variables of the pre-environment can vary to low, and the variables that are in the post-environment can vary to high, without affecting the typing.

Using this type system we can type for example the program presented in the introduction of this chapter, which does not type in a flow-insensitive type system:

$$\begin{array}{l} x := y \\ x := 0 \end{array}$$

The first assigment changes the security type of x from low to high. The second assigment backs the type of x to low.

A syntax-directed version of the type system, taken also from [9], is shown in Figure 3.3. In this new type system the subsumtion rule (SUB) was eliminated. It is an algorithmic type system, in the sense that it calculates the smallest Γ' such that $pc \vdash \Gamma\ \{S\}\ \Gamma'$ for a given sentence S, an environment Γ and program counter pc. The rules Skip, Assign and Seq remain unchanged.

The difference between the rule If of the system of Figure 3.2 and the one in the syntax-directed version is the post-environment of the conditional. In the syntax-directed system this environment is the smallest post-environment in which the conditional can be typable.

In the new system, the rule While is a fixed-point construction. The body of the loop will be typed repeatedly until the post-environment does not change with respect to the last iteration.

Another way to present this rule is by using the least fixed-point operator,

$$\textsc{If}\ \frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma \{ S_i \} \Gamma'_i \quad i = 1,2}{pc \vdash \Gamma \{ \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \} \Gamma'_1 \sqcup \Gamma'_2}$$

$$\textsc{While}\ \frac{\Gamma'_i \vdash e : t_i \quad pc \sqcup t_i \vdash \Gamma'_i \{ S \} \Gamma''_i \quad 0 \le i \le n}{pc \vdash \Gamma \{ \texttt{while}\ e\ \texttt{do}\ S \} \Gamma'_n} \quad \Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma'_{n+1} = \Gamma'_n$$

Figure 3.3: Syntax-directed Flow-Sensitive Typing Rules

denoted as fix, as follows:

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

This fixed-point construction guarantees to terminate since it is computed on a monotone function (defined over the typing rules) and also because the set of environments is finite (since the set of security levels is finite). The proof of convergence of this rule was given in [9] as part of the proof of the following theorem, that states the correctness of the type system. Here we include a complete proof of the theorem since it is crucial for understanding the implementation of the rule.

The function $\mathcal{A}^S$ calculates the smallest Γ' such that $pc \vdash \Gamma \{ S \} \Gamma'$.

Theorem 3.1. *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.*

Proof. The proof is by induction on the structure of the sentence S. We have the following cases:

- $S = x := e$

  Assuming $\Gamma \vdash e : t$, by the Assign rule we have that

  $$\Gamma' = \Gamma[x \mapsto pc \sqcup t]$$

  To prove that $\mathcal{A}^S$ is monotone when $S = x := e$ we need to show that:

  $$pc_1 \sqsubseteq pc_2 \ \wedge \Gamma_1 \sqsubseteq \Gamma_2 \Rightarrow \Gamma_1[x \mapsto pc_1 \sqcup t_1] \sqsubseteq \Gamma_2[x \mapsto pc_2 \sqcup t_2]$$

  where $\Gamma_1 \vdash e : t_1$ and $\Gamma_2 \vdash e : t_2$. This consequence can be proved trivially.

- $S = S_1;S_2$

  By induction hypothesis we have that there exists unique Γ' such that:

  $$pc \vdash \Gamma \{ S_1 \} \Gamma'$$

for a given pc and Γ. Using induction hypothesis on S_2 we have that there exists a unique Γ'' such that:

$$pc \vdash \Gamma' \; \{ \; S_2 \; \} \; \Gamma''$$

By the Seq rule we have that

$$pc \vdash \Gamma \; \{ \; S_1 \; ; S_2 \; \} \; \Gamma''$$

The proof of monotonocy for this case follows from the monotonocy for the sentences S_1 and S_2.

- $S = \texttt{if}\; e \;\texttt{then}\; S_1 \;\texttt{else}\; S_2$

  By induction hypothesis we have that there exists unique Γ_1' and Γ_2' such that:

$$pc \sqcup t \vdash \Gamma \; \{ \; S_1 \; \} \; \Gamma_1' \quad \wedge \quad pc \sqcup t \vdash \Gamma \; \{ \; S_2 \; \} \; \Gamma_2'$$

  for a given pc and Γ, where $\Gamma \vdash e \; : \; t$. Then, by the If rule, we have that

$$pc \vdash \Gamma \; \{ \; \texttt{if}\; e \;\texttt{then}\; S_1 \;\texttt{else}\; S_2 \; \} \; \Gamma_1' \sqcup \Gamma_2'$$

  The proof of monotonocy for this case follows from the monotonocy for the sentences S_1 and S_2.

- $S = \texttt{while}\; e \;\texttt{do}\; S_1$

  Given pc and Γ, by induction hiphotesis we have that there exists unique Γ_0'' and Γ_1'' such that:

$$pc \sqcup t \vdash \Gamma \; \{ \; S_1 \; \} \; \Gamma_0'' \quad \wedge \quad pc \sqcup t_0 \vdash \Gamma_0'' \sqcup \Gamma \; \{ \; S_1 \; \} \; \Gamma_1''$$

  where $\Gamma \vdash e \; : \; t$ and $\Gamma_0'' \sqcup \Gamma \vdash e \; : \; t_0$.
  Then, since $\Gamma \sqsubseteq \Gamma_0'' \sqcup \Gamma$ we have that $t \sqsubseteq t_0$ and also $pc \sqcup t \sqsubseteq pc \sqcup t_0$.
  From the monotonocy of $\mathcal{A}^{S_1}$, $\Gamma \sqsubseteq \Gamma_0'' \sqcup \Gamma$ and $pc \sqcup t \sqsubseteq pc \sqcup t_0$ follows that

$$\Gamma_0'' \sqsubseteq \Gamma_1''$$

  Then, by applying a similar reasoning we have that there exists a chain

$$\Gamma_0'' \sqsubseteq \Gamma_1'' \sqsubseteq \Gamma_2'' \sqsubseteq ...$$

  where $pc \sqcup t_i \vdash \Gamma_i'' \sqcup \Gamma \; \{ \; S_1 \; \} \; \Gamma_{i+1}''$ and $\Gamma_i \sqcup \Gamma \vdash e \; : \; t_i$ and also there exists n such $\Gamma_n'' = \Gamma_m''$ for all $m > n$, since the set of environments is finite. The existence of this chain with finite length guarantees the convergence of the While rule, and therefore there exists a unique Γ_n'' such

$$pc \vdash \Gamma \; \{ \; \texttt{while}\; e \;\texttt{do}\; S_1 \; \} \; \Gamma_n'' \sqcup \Gamma$$

We call G the function that calculates the elements of the chain $\Gamma''_0 \sqsubseteq \Gamma''_1 \sqsubseteq \Gamma''_2 \sqsubseteq ...$, defined as:

$$G\ (pc, \Gamma, \Gamma''_i) = \mathcal{A}^{S_1}(pc \sqcup t_i, \Gamma''_i \sqcup \Gamma)$$

where $\Gamma''_i \sqcup \Gamma \vdash e\ :\ t_i$. Function G is monotone since it is derived from $\mathcal{A}^{S_1}$ which is monotone. This allows us to conclude that $\mathcal{A}^S$ is monotone when $S = \mathtt{while}\ e\ \mathtt{do}\ S_1$.

□

The previous theorem stated the correctness of the algorithmic type system. To implement this type system in Agda we need to define a function that computes, in finite time, the fixed-point of function G. The following theorem is used for the definition of such function in Agda.

Theorem 3.2. *Given a partial order $(S, \sqsubseteq)$, an element $x \in S$, a monotone function $g : S \to S$ over $\sqsubseteq$, and a strictly decreasing function $bound : S \to \mathbb{N}$ with respect to the strict order $\sqsubset$ $(x \sqsubset y \Rightarrow bound\ y < bound\ x)$ with unique minimal element $(bound\ x = 0\ \wedge bound\ y = 0 \Rightarrow x = y)$, if $x \sqsubseteq g\ x$ 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.

- $k = n + 1$

  By hypothesis we know that $bound\ x \leq n + 1$ 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 $bound\ (g\ x) < bound\ x\ \leq n + 1$, and therefore $bound\ (g\ x) \leq n$. 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 $i \leq n$ such $g\ ((g^i\ (g\ x)) = g^i\ (g\ x)$, meaning that $g^i\ (g\ x)$ is a fixed-point of g. Therefore, we conclude that $g^{i+1}\ x$ is a fixed-point of g where $i + 1 \leq n + 1$. □

Based on this theorem we implement an Agda function fixS, which returns a value of the set and a proof that it is a fixpoint of g. Within this function we use a record definition PartialOrder to express the properties of the relation $\sqsubseteq$.

```
record PartialOrder {X} (_≼_ : X → X → Set ) : Set where
  field
    refl      : ∀ {x}     → x ≼ x
    antisym   : ∀ {x y}   → x ≼ y → y ≼ x → x ≡ y
    trans     : ∀ {x y z} → x ≼ y → y ≼ z → x ≼ z
```

Function fixS is defined as follows:

```
fixS :  (X : Set) →
        (k : ℕ) →    -- maximun number of iterations
        (_⊏_ : Rel X) →
        (_⊑_ : Rel X) →
        (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
        (str : {x y : X} → x ⊏ y → x ⊑ y) →
        (par : PartialOrder _⊑_ ) →

        (bound : X → ℕ) →    -- bound function
        -- the bound function is decreasing
        (∀ {x y : X} → x ⊏ y → bound y ¡ bound x) →
        -- the bound function has a unique minimal value
        (∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →

        (g : X → X) →
        (∀ {x y} → x ⊑ y → g x ⊑ g y) → -- g is monotone

        (x : X) →           -- inicial value
        x ⊑ g x →
        k ≥ bound x →       -- invariant
        Σ X (λ x → x ≡ g x )

fixS X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g gmono x x⊑gx p
  with minimal x (g x) (n≡0 p)
       (n≡0 (trans (propDec' bound rel⊏ rel⊑ boundDec x⊑gx) p))

fixS X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g gmono x x⊑gx p |
  x≡gx = x , x≡gx

fixS X (suc n') rel⊏ rel⊑ sum str par bound boundDec minimal g gmono
    x x⊑gx p with sum x⊑gx
... | inl x≡gx = x , x≡gx
... | inr x⊏gx =
        let r = p≤p (trans (boundDec x⊏gx) p)
        in fixS X n' rel⊏ rel⊑ sum str par
          bound boundDec minimal g
          gmono (g x) (gmono x⊑gx) r
```

In contrast to Theorem 3.2, fixS does not show that the solution is $g^n\ \Gamma_0$ for some $n \leq k$. Notice that when the number of iterations is 0 we use properties of the bound function that are passed to the function to help the type checker to infer that $x = g\ x$.

Function fixS turns out to be itself a monotone function. The monotonicity of fixS will be required later for the impementation of the type system for sentences, which uses a fix operator that is defined in terms of fixS. Here we show only the type of this property:

```
fixSmonotone :   (X : Set) →
   (k : ℕ) →                -- maximun number of iterations
   (_⊏_ : X → X → Set) →
   (_⊑_ : X → X → Set) →
   (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
   (str : {x y : X} → x ⊏ y → x ⊑ y) →

   (par : PartialOrder _⊑_ ) →

   (bound : X → ℕ) →   -- bound function
   -- the bound function is decreasing
   (bDec : ∀ {x y : X} → x ⊏ y → bound y < bound x) →
   -- the bound function has a unique minimal value
   (minimal : ∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →

   (g : X → X) →
   -- g is monotone
   (gmono : ∀ {x y} → x ⊑ y → g x ⊑ g y) →

   (x : X) →                -- inicial value x
   (x⊑gx : x ⊑ g x) →
   (inv : k ≥ bound x) →    -- invariant
   (x' : X) →               -- inicial value x'
   (x'⊑gx' : x' ⊑ g x') →
   (inv' : k ≥ bound x') →   -- invariant
   x ⊑ x' → g x ⊑ g x' →
   proj₁ (fixS X k _⊏_ _⊑_ sum str par bound bDec minimal g gmono x x⊑gx inv) ⊑
   proj₁ (fixS X k _⊏_ _⊑_ sum str par bound bDec minimal g gmono x' x'⊑gx' inv')
```

3.2 Agda Implementation

For the Agda implementation of the language with flow-sensitive type system we want to proceed in a similar way as we did for the same language in Chapter 2 when the type system was flow-insensitive. The goal then is to define GADTs of flow-sensitive typed terms for expressions and statements. Those typed terms will be useful later when we define their translation to flow-insensitive typed terms.

For the expressions of the language the implementation is immediate. The expressions with the typing rules of Figure 3.1 are represented by the type

ExpS, which is parametrized by a vector of security types corresponding to the type environment, and a value of type St corresponding to the security type of the expression:

```
data ExpS    {m : ℕ} (Γ : Vec St m) : St → Set where
  IntValS    : ℕ → ExpS Γ Low
  VarS       : (n : Fin m) → ExpS Γ (lookup n Γ )
  AddS       : {st st' : St} → ExpS Γ st → ExpS Γ st' → ExpS Γ (st ∪ st')
```

By representing variables as naturals numbers taken from a finite set we make them correspond to the positions of a vector (type environment) Γ. The type Fin *m* represents the type of finite sets of size *m*.

In the case of statements we proceed in two stages. First, we define a relation that implements the typing rules of Figures 3.2 and 3.3 and as second step we define a datatype that represents the abstract syntax of statemets decorated with their flow-sensitive typing rules. Like in Chapter 2, this is indeed possible because we have a syntax-directed type system.

The relation that implements the typing rules for statements is defined on ordinary, non-decorated abstract syntax terms given by datatypes ASTExp and ASTCom, which capture the structure of expressions and sentences, respectively.

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

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

Concerning expressions, function typeAstExp computes the security type of an expression under an environment Γ according to the typing rules of Figure 3.1.

```
typeAstExp : {n : ℕ} → Vec St n → ASTExp n → St
typeAstExp Γ (INTVAL n)    = Low
typeAstExp Γ (VAR n)       = lookup n Γ
typeAstExp Γ (ADD e e')    = (typeAstExp Γ e) ∪ typeAstExp Γ e'
```

The type system for statements is then implemented by the following relation:

```
data Tc {n : ℕ} : ASTCom n → St → Vec St n → Vec St n → Set where
  Skip : {Γ : Vec St n}{pc : St} →
    Tc SKIP pc Γ Γ
```

```
Ass : {x : Fin n}{e : ASTExp n}{Γ : Vec St n}{pc : St} →
   Tc (ASSIGN x e) pc Γ (change x Γ (pc ∪ (typeAstExp Γ e)))

Seq : {Γ Γ' Γ" : Vec St n}{pc : St} {s₁ s₂ : ASTCom n} →
   Tc s₁ pc Γ Γ'   →
   Tc s₂ pc Γ' Γ" →
   Tc (SEQ s₁ s₂) pc Γ Γ"

If0 : {Γ Γ' Γ" : Vec St n}{pc : St}{e : ASTExp n}{s₁ s₂ : ASTCom n} →
   Tc s₁ (pc ∪ (typeAstExp Γ e)) Γ Γ' →
   Tc s₂ (pc ∪ (typeAstExp Γ e)) Γ Γ" →
   Tc (IF0 e s₁ s₂) pc Γ (Γ' ⊔ Γ")

While : {Γ : Vec St n}{pc : St}{e : ASTExp n}{s : ASTCom n} →
   Tc (WHILE e s) pc Γ (proj₁ (fix {n} s e pc Γ))
```

For the While case, recall that, given an initial environment Γ_0 and a program counter pc, an statement `while` e `do` S types in the post-environment that results from computing the following fixpoint construction:

$$\mathrm{fix}(\lambda\Gamma\ .\ \mathrm{let}\ \ \Gamma\vdash e\ :\ t \quad pc\sqcup t\vdash\Gamma\ \{\ S\ \}\ \Gamma'\ \mathrm{in}\ \Gamma'\sqcup\Gamma_0) \tag{3.1}$$

To obtain the post-environment we define a function fix that computes the fixed-point of G, the function used in the proof of the While rule in Theorem 3.1:

$$G\ (pc,\Gamma,\Gamma_i'') = \mathcal{A}^S(pc\sqcup t_i,\Gamma_i''\sqcup\Gamma)$$

where $\Gamma_i''\sqcup\Gamma\vdash e\ :\ t_i$. Function fix

```
fix : {n : ℕ} →
   (s : ASTCom n) →
   (e : ASTExp n) →
   (pc : St) →
   (Γ : Vec St n) →
   Σ (Vec St n) (λ Γ' → Γ' ≡ body e s pc Γ Γ')
```

takes a statement *s* and an expression *e* (body and condition of a loop, resp.), a program context `pc`, and an initial type environment Γ and computes the environment Γ' that satisfies to be the fixpoint of the function body *e s pc* Γ, which implements the body of the fixpoint construction (3.1):

```
body : {n : ℕ} (e : ASTExp n ) →
   (s : ASTCom n ) →
   (pc : St ) →
   (Γ : Vec St n ) →
   (Γ' : Vec St n ) → Vec St n

body e s pc Γ Γ' =   let   st = typeAstExp Γ' e
```

```
             Γ” = tc s (pc ∪ st) Γ’
         in Γ” ⊔ Γ
```

Function body is defined in terms of another function, called tc, which is a functional implementation of the algorithmic type system. Given a statement *s*, a program context *pc*, and an initial type environment Γ, tc *s pc* Γ computes the post-environment Γ’ that satisfies the type system.

$$\text{tc}\ s\ pc\ \Gamma = \Gamma' \quad\Longleftrightarrow\quad \text{Tc}\ s\ pc\ \Gamma\ \Gamma'$$

Function tc returns a dependent pair consisting of the final type environment and a proof that it satisfies the Tc relation.

```
tc : {n : ℕ} →
   (s : ASTCom n) →
   (pc : St) →
   (Γ : Vec St n) →
   Σ (Vec St n) (λ Γ’ → Tc s pc Γ Γ’)

tc (SEQ s s’) pc Γ =   let (Γ’ , tcs)    = tc s pc Γ
                           (Γ” , tcs’) = tc s’ pc Γ’
                       in   Γ” , Seq tcs tcs’


tc (IF0 e s s’) pc Γ =   let pc’ = pc ∪ (typeAstExp Γ e)
                             (Γ’ , tcs)    = tc s pc’ Γ
                             (Γ” , tcs’) = tc s’ pc’ Γ
                         in   Γ’ ⊔ Γ” , If0 tcs tcs’

tc {n} (WHILE e s) pc Γ = proj₁ (fix {n} s e pc Γ) , While

tc {n} SKIP pc Γ = Γ , Skip
```

Function fix is defined in terms of fixS, the function that implements Theorem 3.2 in Agda. As partial order we use the following partial order ⊑ (and its strict version ⊏) between security type vectors:

```
mutual
  data _⊑_ : {n : ℕ} → Vec St n → Vec St n → Set where
     xs≡ys : ∀ {n} {xs : Vec St n} → xs ⊑ xs
     xs⊏ys : ∀ {n} {xs ys : Vec St n} → xs ⊏ ys → xs ⊑ ys

  data _⊏_ : {n : ℕ} → Vec St n → Vec St n → Set where
     Γ⊏Γ’< : ∀ {n} {Γ Γ’ : Vec St n} → Γ ⊑ Γ’ → (Low :: Γ) ⊏ (High :: Γ’)
     Γ⊏Γ’≡ : ∀ {n} {st : St} {Γ Γ’ : Vec St n} → Γ ⊏ Γ’ → (st :: Γ) ⊏ (st :: Γ’)
```

Function body plays the role of function g in fixS. The proof that body is monotone uses the property that tc is also monotone, and since tc is defined in terms of fix, then the monotonicity of tc requires that fix is monotone. Here we show only the types of these facts, the complete proofs can be found in Appendix B.

```
bodyMonotone : {n : ℕ}{e : ASTExp n}{s : ASTCom n}
    {pc pc' : St}{Γ Γ' Γ₁ Γ₁' : Vec St n} →
    pc ≤St pc' →
    Γ ⊑ Γ₁ → Γ' ⊑ Γ₁' →
    body e s pc Γ Γ' ⊑ body e s pc' Γ₁ Γ₁'
```

```
tcMonotone : {n : ℕ}{pc pc' : St}{Γ Γ₁ : Vec St n}
    (s : ASTCom n) →
    pc ≤St pc' → Γ ⊑ Γ₁ → tc s pc Γ ⊑ tc s pc' Γ₁
```

```
fixMonotone : {n : ℕ}{pc pc' : St}{Γ Γ' : Vec St n} →
    (s : ASTCom n) → (e : ASTExp n) →
    pc ≤St pc' → Γ ⊑ Γ' →
    proj₁ (fix s e pc Γ) ⊑ proj₁ (fix s e pc' Γ')
```

We use function sumLow, which computes the number of low values in an environment, to play the role of function *bound*.

```
sumLows : {n : ℕ} → Vec St n → ℕ
sumLows [] = zero
sumLows (Low :: xs) = suc (sumLows xs)
sumLows (High :: xs) = sumLows xs
```

This function fits as a bound function because in each step of the computation of the post-environment in the While rule the number of low variables in the environment is possibly decreasing.

We prove two properties about sumLows: (i) it is decreasing, and (ii) it has a unique minimal value. Here, we show just the types of these properties:

```
sumLowDec : {n : ℕ} {Γ Γ' : Vec St n} → Γ ⊏ Γ' → sumLows Γ < sumLows Γ'

minimalEnv : {n : ℕ} → {Γ Γ' : Vec St n} →
    sumLows Γ ≡ 0 → sumLows Γ' ≡ 0 → Γ ≡ Γ'
```

Finally, we have all the elements necessary to define fix in terms of fixS:

```
fix : {n : ℕ} →
    (s : ASTCom n) →
    (e : ASTExp n) →
    (pc : St) →
    (Γ : Vec St n) →
    Σ (Vec St n) (λ Γ' → Γ' ≡ body e s pc Γ Γ')
```

```
fix {n} s e pc Γ =
   let Γ₀ = Γ
       Γ₁ = body e s pc Γ Γ₀
   in fixS (Vec St n)
       n _⊏_ _⊑_ ≡∨⊏ xs⊏ys
       parOrd⊑
       -- bound functions and properties
       sumLows sumLowDec minimalEnv
       (body e s pc Γ) -- function g
       (bodyMonotone {e = e} {s = s} (≤St-refl {st = pc}) (refl⊑ {Γ = Γ}))
       Γ₀ lema1 (lemasL Γ₀)
```

Observe that the maximal number of iterations is n, the length of the environments. The record parOrd⊑ contains the properties that the relation _⊑_ over vectors of security types is a partial order.

We conclude the section by presenting the implementation of typed abstract syntax terms for statements:

```
data ComS {m : ℕ} : Vec St m → St → Vec St m → Set where

   AssignS : {Γ : Vec St m}{pc st : St}
      (n : Fin m) →
      ExpS Γ st →
      ComS Γ pc (change n Γ (pc ∪ st))

   SkipS : {Γ : Vec St m}{pc : St} →
      ComS Γ pc Γ

   SeqS    : {Γ Γ' Γ" : Vec St m}{pc : St} →
      ComS Γ pc Γ' →
      ComS Γ' pc Γ" →
      ComS Γ pc Γ"

   IfS    : {pc st : St}{Γ Γ' Γ" : Vec St m} →
      ExpS Γ st →
      ComS Γ (pc ∪ st) Γ' →
      ComS Γ (pc ∪ st) Γ" →
      ComS Γ pc (Γ' ⊔ Γ")

   WhileS : {Γ : Vec St m}{pc tn : St}
      (s : ASTCom m)
      (e : ASTExp m) →
      ComS Γ pc (proj₁ (fix {m} s e pc Γ) )
```

Like relation Tc, ComS is indexed by two type environments of the same lenght which contain the security level of the variables before and after the execution of the sentence. The other parameter represents the security type of the program context.

The constructors AssignS, SeqS and IfS can be seen as direct implementations of the rules Assign, Seq and If of the algorithmic version of the type system, shown in Figures 3.2 and 3.3.

Notice that, unlike the other constructors, the While constructor only considers ordinary ASTs of expressions and statements. This is because, as we already saw, the type information of *s* and *e* is unnecessary for the computation of the fixpoint.

The following definition connects ComS with the type system by stating that ComS is a correct implementation of typable statements.

```
liftS : {m : ℕ}{Γ Γ' : Vec St m}{pc : St}
  (s : ASTCom m)    →
  Tc s pc Γ Γ' →
  ComS Γ pc Γ'

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
```

The codes that complete the definitions presented in this section can be found in Appendix B.

3.3 The translation to flow-insensitive

Now we turn to the formalization of Hunt and Sands program translation [9] in Agda. It converts programs typable in the flow-sensitive type system introduced in this chapter to equivalent programs typable in a flow-insensitive type system. The translation rules, shown in Figure 3.4, are defined as an extension of the flow-sensitive type system and are expressed in terms of the following 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 sentence with floating-type variables, typable in the flow-sensitive type system, and D is the equivalent sentence (or sentences) produced by the translation, with fixed-type variables and typable in the flow-insensitive type system.

We start with the introduction of some notation and constructions used in the translation.

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

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

$$\frac{\Gamma \vdash E \ : \ t \qquad pc \sqcup t \vdash \Gamma \ \{S_1 \rightsquigarrow D_1\} \ \Gamma'_1 \qquad pc \sqcup t \vdash \Gamma \ \{S_2 \rightsquigarrow D_2\} \ \Gamma'_2 \qquad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2}{pc \vdash \Gamma \ \{\texttt{if} \ E \ \texttt{then} \ S_1 \ \texttt{else} \ S_2 \rightsquigarrow \texttt{if} \ E_\Gamma \ \texttt{then} \ (D_1 \ ; \ \Gamma' := \Gamma'_1) \ \texttt{else} \ (D_2 \ ; \ \Gamma' := \Gamma'_2)\} \ \Gamma''}$$

$$\frac{\Gamma'_i \vdash E \ : \ t_i \qquad pc \sqcup t_i \vdash \Gamma'_i \ \{S \rightsquigarrow D_i\} \ \Gamma''_i \qquad 0 \le i \le n}{pc \vdash \Gamma \ \{\texttt{while} \ E \ \texttt{do} \ S \rightsquigarrow \Gamma'_n := \Gamma \ ; \texttt{while} \ E_{\Gamma'_n} \ \texttt{do} \ (D_n \ ; \ \Gamma'_n := \Gamma''_n)\} \ \Gamma'_n}$$
$$\Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma'_{n+1} = \Gamma'_n$$

Figure 3.4: Translation rules

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 $FVar$ is defined from the set of floating variables, Var, by annotating each variable name with a security type:

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

This means that for each variable x, two fixed-type variables are introduced: x_L and x_H.

Each time a floating-type variable x raises its security type, the translation will reflect this fact by constructing an assignment that moves information from x_L to x_H. 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 $\Gamma := \Gamma'$ will be used with environments Γ, Γ' such that $\Gamma' \sqsubseteq \Gamma$. This gives rise to well-typed assignments as information flows in the appropriate direction (from low to high variables).

The relation _⊑_ is defined by the following datatype:

```
data _⊑_ : {n : ℕ} → Vec St n → Vec St n → Set where
  []⊑[] : [] ⊑ []
  xs⊑ys : ∀ {n} {x y : St} {xs ys : Vec St n} →
    x ≤St y → xs ⊑ ys → (x :: xs) ⊑ (y :: ys)
```

A sequence of assignments is modeled as the following function in Agda:

```
Γ:=Γ' : {n : ℕ} {pc : St} →
  (Γ : Vec St n) → (Γ' : Vec St n) → Γ' ⊑   Γ → Stm pc
Γ:=Γ' {n = n} Γ Γ' Γ'⊑Γ = assigments 0 Γ Γ' Γ'⊑Γ
```

It is based on a function assigments which generates the sequence of assignments by recursively traversing the environments Γ and Γ'. At each i it adds an assigment $\Gamma(i) := \Gamma'(i)$ if $\Gamma(i) = High$ and $\Gamma'(i) = Low$.

```
assigments : {n : ℕ} {pc : St} →
  ℕ → (Γ : Vec St n) → (Γ' : Vec St n) → Γ' ⊑   Γ   → Stm pc

assigments i .[] .[] []⊑[] = Skip

assigments i .(Low :: ys) .(Low :: xs)
(xs⊑ys {n} {.Low} {Low} {xs} {ys} l≤x xs≤ys)   =
  assigments (suc i) ys xs xs≤ys

assigments i .(High :: ys) .(Low :: xs)
  (xs⊑ys {n} {.Low} {High} {xs} {ys} l≤x xs≤ys)   =
    Seq (Assign ≤StHigh ≤StHigh (VarH i) (Var (VarH i)))
      (assigments (suc i) ys xs xs≤ys )

assigments i .(High :: ys) .(High :: xs)
  (xs⊑ys {n} {.High} {.High} {xs} {ys} h≤h xs≤ys)   =
    assigments (suc i) ys xs xs≤ys
```

Given an environment Γ, every expression E with floating-type variables can be translated to an expression E_Γ with fixed-type variables by replacing each floating-type variable x by a fixed-type variable x_s whose security level s is that given in Γ for x, i.e. $s = \Gamma(x)$. The following function implements this transformation, where Exp is the type of expressions presented in Chapter 2.

```
transExp : {n : ℕ}{st : St}{Γ : Vec St n} → ExpS Γ st → Exp st

transExp (IntValS y) = IntVal y
transExp {Γ = Γ} (VarS n') with lookup n' Γ
... | Low = Var (VarL (toℕ n'))
... | High = Var (VarH (toℕ n'))

transExp (AddS y y') = Add (transExp y) (transExp y')
```

Now we are in conditions to present the Agda implementation of the translation shown in Figure 3.4.

```
translate : {n : ℕ}{pc : St} {Γ Γ' : Vec St n} →
  ComS Γ pc Γ' → Stm pc
```

```
-- Asignment
translate {pc = pc} (AssignS {st = st} v e) =
   assign pc st (toℕ v) (transExp e)

-- Skip
translate SkipS = Skip

-- Sequencing
translate (SeqS s s’) with translate s | translate s’
... | s₁ | s₂ = Seq s₁ s₂

-- Conditional
translate (IfS {pc} {st} {Γ} {Γ₁’} {Γ₂’} e s s’) with translate s | translate s’
... | s₁ | s₂    =
   If0 (transExp e)
      (Seq s₁ (Γ:=Γ’ {pc = pc ∪ st} (Γ₂’ ⊔ Γ₁’) Γ₁’ (Γ⊑Γ’⊔Γ Γ₁’ Γ₂’)))
      (Seq s₂ (Γ:=Γ’ {pc = pc ∪ st} (Γ₁’ ⊔ Γ₂’) Γ₂’ (Γ⊑Γ’⊔Γ Γ₂’ Γ₁’)))

-- Iteration
translate (WhileS {Γ } {pc} e s) with fromAstE (proj₁ (fix s e pc Γ) ) e
... | st , en with tc s (pc ∪ st ) (proj₁ (fix s e pc Γ) )
... | Γ” , sn =
   let   Γn’ = Γ” ⊔ Γ
         dn = translateAst (pc ∪ st) Γ s
   in Seq (Γ:=Γ’ Γn’ Γ (Γ⊑Γ’⊔Γ Γ Γ”))
      (While (transExp en)
         (Seq dn (Γ:=Γ’ Γn’ Γ” (Γ⊑Γ⊔Γ’ Γ” Γ))))
```

Function assign groups the different cases that occur in assignments:

```
assign : (pc : St) → (st : St) → ℕ → Exp st → Stm pc
assign Low Low x e     = Assign l≤x l≤x (VarL x) e
assign Low High x e    = Assign h≤h l≤x (VarH x) e
assign High st x e     = Assign ≤StHigh h≤h (VarH x) e
```

Function translateAst is like translate, but it works on ordinary ASTs for statements instead of typable statements.

```
translateAst : {n : ℕ}( pc : St) (Γ : Vec St n) → ASTCom n → Stm pc
```

It is worth noticing that 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. Again, this is a situation where we are using Agda's type system to enforce the preservation of the security property. In other words, translate can be understood as an implemention of the following theorem, where $[pc] \vdash_{sd} D$ states for the typing judgement of the syntax-directed type system presented in Figure 2.3 (Chapter 2).

Theorem 3.3 ([9]). *If* $pc \vdash \Gamma\ \{S \rightsquigarrow D\}\ \Gamma'$ *then* $[pc] \vdash_{sd} D$*.*

3.4 Summary

In this chapter we formalized Hunt and Sands translation that converts programs typable in a flow-sensitive type system into programs typable in a flow-insensitive type system.

Like in Chapter 2, we introduced typed representations of expressions and statements, now for the flow-sensitive type system. In most of the cases those representations were the result of a direct implementation of the typing rules. The interesting case was that of the While statement. Since it is based on a fixpoint construction, it was necessary to establish the conditions that should be met for the existence of the fixpoint. We stated the conditions in Theorem 3.2 (they are similar to the conditions that are required to guarantee the termination of a while sentence in Hoare Logic). Based on this theorem we defined function fixS, which was an essential ingredient for the implementation of the fixpoint construction.

In the context of our formalization we defined the translation as a function between flow-sensitive typed terms and flow-insensitive typed terms.

Chapter 4

Conclusions

In this thesis we dealt with the formalization of program translations that preserve the security property of noninterference. We used a type-based approach to noninterference on programming languages with different characteristics. Agda was our formalization framework; we used it both as a functional language and as a proof assistant.

The first case we analyzed was the development of a simple compiler between a While language and semi-structured machine code. We defined a (flow-insensitive) security type system for each of the languages, which we proved sound with respect to the corresponding semantic definition of noninterference. The definition of the security type system for the high level language did not present any particular difficulty since it is something rather standard [22, 20, 14]. This contrasts with the situation of low-level languages, where works presenting security type systems for those languages are scarce (e.g. [2, 1, 12, 19]). Despite its simplicity and the fact of being semi-structured (without jumps), the definition of a security type system for the target language was not absent of some complications.

The second program translation we adressed was the one introduced by Hunt & Sands [9] and focuses on transforming programs in a While language which are typable in a flow-sensitive type system to equivalent programs typable in a flow-insensitive type system. An aspect that captured our interest in formalizing this transformation is that it contains the computation of a final environment as the result of a fixpoint construction.

A distinguishing feature of our Agda formalization was the systematic use we did of typed representations of the abstract syntax for the object languages in terms of inductive families. This of course demanded the introduction of syntax-directed formulations of all security type systems. As a result of this encoding, only terms corresponding to non-interfering programs can be written in the Agda implementation of the object languages. This has also consequences on the functions we can define between those typed representations. For example, concerning the compiler developed in Chapter 2, the use of typed representations of the languages makes that the type of the compiler by itself expresses the preservation of non-interference through compilation.

Therefore, the definition of the compiler plays two roles simultaneously: as compiler function (as expected) and as proof term of the preservation property. The verification that it is indeed a valid proof of the property is then performed by Agda's type system. The same happens with the implementation of Hunt & Sands' translation presented in Chapter 3. Summing up, the two program translations we analyzed are nice examples of function definitions that are designed following a sort of *correct-by-construction* discipline. In this respect, Agda resulted a more than suitable framework for experiment with such programming methodology.

Future work

In this thesis, we only dealt with the noninterference property. It is our interest to explore the analysis of other security properties and security-preserving program translations following a similar approach as the one of this thesis.

It would be interesting to use another, more realistic, low-level language as target language of the compiler. However, a problem of real low-level languages is their absence of structure, which complicates the definition of syntax-directed security type systems. We are currently experimenting in the development of a compiler in Agda that takes programs in a similar While language and returns code in a low-level language with jumps defined by Saabas & Uustalu [19, 18]. This low-level language has the important feature of being modular, which is a key aspect for defining a syntax-directed type system. In this language, modularity is achieved by defining a code as the finite union of non-overlapping pieces of code. A security type system for this language is presented in [19].

Another alternative could be to consider a richer language as target language of the compiler. For example, the one developed by Zdancewic [24], which is a security-typed language that includes recursion, higher-order functions, structured state, and concurrency. Zdancewic also considers more practical security policies than noninterference.

Chapter 5

Proofs from Chapter 2

5.1 Proof of Theorem 2.2

We include here the proof of properties (iii) and (iv) only. Properties (i) and (ii) can be proved similarly by induction.

Property (iii) If $[pc] \vdash_{sd} S$ then $[pc] \vdash S$.

Proof. The proof is by induction on the structure of statements. The cases corresponding to the sentences skip and sequence are immediate.

- Case $S = x := e$

  We have two cases:

  - Case $\Gamma(x) = low$
    Since $st \leq low$ and $pc \leq Low$ by rule $\textsc{ass}_{sd}$, we have that $st = low$ and $pc = low$. We conclude using rule $\textsc{ass}$L.
  - Case $\Gamma(x) = high$
    By rule $\textsc{ass}$H we conclude that $[pc] \vdash x := e$

- Case $S = \texttt{while}\ e\ \texttt{do}\ S_1$

  If $\texttt{while}\ e\ \texttt{do}\ S_1$ is typable, then the rule $\textsc{while}_{sd}$ is used and we have that:

  - $\vdash_{sd} e : st$
  - $[\texttt{max}\ st\ pc] \vdash_{sd} S_1$
  - $[pc] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S_1$

  By induction hypothesis we have that $[\texttt{max}\ st\ pc] \vdash S_1$ and by the property (i) we have that $\vdash e\ :\ st$.

  We can prove using the subsumption rule that if $[\texttt{max}\ st\ pc] \vdash S$ then $[pc] \vdash S$, for any statement S and security types st and pc. Then, we have that $[pc] \vdash S_1$ and by rule $\textsc{while}$ we conclude that $[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S_1$.

- The conditional case is similar to the previous case.

□

Property (iv) If $[pc] \vdash S$ then there exists pc' such that $[pc'] \vdash_{sd} S$ and $pc \leq pc'$.

Proof. The proof is by induction on the typing derivation of $[pc] \vdash S$.

- The cases the sentence skip and sequence are immediate.
- If the last rule used in the derivation is ASSL, we have that $\vdash e : low$. By property (ii) we have that $\vdash_{sd} e : low$.

  Then, we use rule ASS_{sd} to conclude that $[low] \vdash_{sd} x_L := e$.
- If the last rule used in the derivation is ASSH, we have that $[pc] \vdash x_H := e$.

  Since $pc \leq high$ and $st \leq high$ for any pc and st, and any expression is typable, by rule ASS_{sd} we have that $[pc] \vdash_{sd} x_H := e$.
- When the last rule used in the derivation is WHILE, we have that

  - $\vdash e : pc$
  - $[pc] \vdash S$
  - $[pc] \vdash \texttt{while}\ e\ \text{do}\ S$

  By property (ii) we have that exists pc' such $\vdash_{sd} e : pc'$ with $pc' \leq pc$. By induction hypothesis we have that $[pc''] \vdash_{sd} S$ for any $pc'' \geq pc$. Then, since $\texttt{max}\ pc''\ pc' = pc''$, using rule WHILE_{sd} we conclude that:

  $$[pc''] \vdash_{sd} \texttt{while}\ e\ \text{do}\ S$$
- When the last rule used in the derivation is IF, the proof is analogous to the previous case.
- When the last rule used in the derivation is SUB we have that:

  - $[high] \vdash S$
  - $[low] \vdash S$

  then we proceed by induction on the structure of S:

  - The cases for assignment of low and high variables and skip are immediate.

- Case $S = S_1;S_2$.

  The unique rule that can be used as last rule in the derivation of $[high] \vdash S_1;S_2$ is the rule SEQ.Then, we have that $[high] \vdash S_1$ and $[high] \vdash S_2$.

  By induction hypothesis we can conclude that $[high] \vdash_{sd} S_1$ and $[high] \vdash_{sd} S_2$. Finally, we use SEQ$_{sd}$ rule to conclude that

  $$[high] \vdash_{sd} S_1;S_2$$

  .
- Case $S = \texttt{while}\ e\ \texttt{do}\ S_1$.

  The unique rule that can be used as last rule in the derivation of $[high] \vdash \texttt{while}\ e\ \texttt{do}\ S_1$ is the rule WHILE.Then, we have that $\vdash high : e$ and $[high] \vdash S_1$.

  By induction hypothesis we have that $[high] \vdash_{sd} S_1$ and by property (ii) we have that there exists st such $\vdash_{sd} st : e$.

  Then, since $\texttt{max}\ st\ high = high$, by rule WHILE$_{sd}$ we conclude that $[high] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S_1$
- The case for conditionals is analogous.

□

5.2 Proof of Theorem 2.3

Proof. The proof of part *i)* is by induction on the structure of commands.
We have the following cases.

- Case $S = S_1;S_2$

  If $S_1;S_2$ is typable in the type system of figure 2.3, then by rule SEQ' we have that:

  i) $[pc_1] \vdash' S_1$
  ii) $[pc_2] \vdash' S_2$
  iii) $[\texttt{min}\, pc_1\ pc_2] \vdash' S_1;S_2$

  Since $\texttt{min}\ pc_1\ pc_2 \leq pc_1$ and $\texttt{min}\ pc_1\ pc_2 \leq pc_2$, using rule SUB' and the items i) and ii) we conclude that:

  $[\texttt{min}\, pc_1\ pc_2] \vdash' S_1$ and $[\texttt{min}\, pc_1\ pc_2] \vdash' S_1$

  By induction hypothesis we have that:

$[\mathtt{min}\; pc_1\; pc_2] \vdash_{sd} S_1$ and $[\mathtt{min}\; pc_1\; pc_2] \vdash_{sd} S_1$

Then, using rule SEQ$_{sd}$ we conclude that $[\mathtt{min}\; pc_1\; pc_2] \vdash_{sd} S_1;S_2$.

- Case $S = \mathtt{while}\; e\; \mathtt{do}\; S$

  Assuming that S is typable in type system of figure 2.3, then by rule WHILE' we have that:

  i) $\vdash_{sd} e : st$
  ii) $[pc] \vdash' S$
  iii) $st \leq pc$
  iv) $[pc] \vdash' \mathtt{while}\; e\; \mathtt{do}\; S$

  By the item iii) we have that $\mathtt{max}\; st\; pc = pc$. Then by induction hypothesis:

$$[\mathtt{max}\; st\; pc] \vdash_{sd} S$$

  By rule WHILE$_{sd}$ we conclude that $[pc] \vdash_{sd} \mathtt{while}\; e\; \mathtt{do}\; S$

- The conditional case is similar to the previous case.

- The cases for assigments and the skip sentence are immediate.

Now, we prove the part *ii)* also by induction on the structure of commands. We have the following cases.

- Case $S = S_1;S_2$

  Since $S_1;S_2$ is typable, by rule SEQ$_{sd}$ we have that:

  i) $[pc] \vdash_{sd} S_1$
  ii) $[pc] \vdash_{sd} S_2$
  iii) $[pc] \vdash_{sd} S_1;S_2$

  By induction hypothesis on items i) and ii), and the rule SEQ' we conclude that $[pc] \vdash' S_1;S_2$.

- Case $S = \mathtt{while}\; e\; \mathtt{do}\; S_1$

  Assuming that S is typable, by rule WHILE$_{sd}$ we have that:

  i) $\vdash_{sd} e : st$

ii) $[\texttt{max}\ st\ pc] \vdash_{sd} S_1$
iii) $[pc] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S_1$

By induction hypothesis we have that $[\texttt{max}\ st\ pc] \vdash' S_1$.

Since $st \leq \texttt{max}\ st\ pc$, by rule WHILE' we have that $[\texttt{max}\ st\ pc] \vdash' \texttt{while}\ e\ \texttt{do}\ S_1$. Then, we use that $pc \leq \texttt{max}\ st\ pc$ and rule SUB' to conclude that $[pc] \vdash' \texttt{while}\ e\ \texttt{do}\ S_1$.

- The conditional case is similar to the previous case.
- The cases for assigments and the skip sentence are immediate.

□

5.3 Proof of Theorem 2.6

Proof. To prove soundness we will use the bottom-up syntax directed version of the type system.

The proof is by induction on the derivation $\langle S, s\rangle \Downarrow s'$.

We have the following cases:

- If the rule for assigments is used we have that:

i) $\langle x := e, s\rangle \Downarrow s[x \mapsto \mathcal{E}[\![e]\!]\, s]$
ii) $\langle x := e, t\rangle \Downarrow t[x \mapsto \mathcal{E}[\![e]\!]\, t]$
iii) $[pc] \vdash_{sd} x := e \quad \vdash_{sd} e : st \quad st \leq \Gamma(x) \quad pc \leq \Gamma(x)$

Since the states s and t agree in all lows variables (by assumption $s \cong_L t$), to prove that $s[x \mapsto \mathcal{E}[\![e]\!]\, s] = t[x \mapsto \mathcal{E}[\![e]\!]\, t]$ we just need to prove that if $\Gamma(x) = low$ then $\mathcal{E}[\![e]\!]\, s = \mathcal{E}[\![e]\!]\, t$. We suppose that $\Gamma(x) = low$, then $st = low$. This means that the expression e not contain high variables. Then, $\mathcal{E}[\![e]\!]\, s = \mathcal{E}[\![e]\!]\, t$ follows from $s \cong_L t$ and Lemma 2.5.

- The case for sequences can be solved easily using induction hypothesis.
- If the first rule for the if sentence is used we have that:

i) $\mathcal{E}[\![e]\!]\, s = 0 \quad \langle S_1, s\rangle \Downarrow s' \quad \langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Downarrow s'$
ii) $\langle \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2, t\rangle \Downarrow t'$

iii) $\vdash_{sd} e : st \quad [pc_1] \vdash'_{sd} S_1 \quad [pc_2] \vdash'_{sd} S_2 \quad st \leq \mathtt{min}\, pc_1\, pc_2$
$[\mathtt{min}\, pc_1\, pc_2] \vdash'_{sd} \mathtt{if}\ e\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2$

We continue the proof by cases:

If $st = low$, then $[pc_1] \vdash'_{sd} S_1$ and $[pc_2] \vdash'_{sd} S_2$. Now, since $s \cong_L t$ we use Lemma 2.5 to conclude that $\mathcal{E}[\![e]\!]\ t = 0$. By semantic definition we have that $\langle S_1, t\rangle \Downarrow t'$. Then, by induction hypothesis we conclude that $s' \cong_L t'$.

If $st = high$, then $[high] \vdash'_{sd} S_1$ and $[high] \vdash'_{sd} S_2$. As opposite to the previous case we do not know if $\mathcal{E}[\![e]\!]\ t = 0$ or $\mathcal{E}[\![e]\!]\ t \neq 0$, since the expression e can contain high variables.

Using confinement (2.4) on the first sentence ($[high] \vdash'_{sd} S_1$) we conclude that $s \cong_L s'$. And on the second sentence ($[high] \vdash'_{sd} S_2$) we have that $t \cong_L t'$. Since $s \cong_L t$ by transitivity we conclude that $s' \cong_L t'$.

- The case of the second rule for the if sentence is analogous.

- The case for the while sentence is similar to the if sentence.

□

5.4 Proof of Theorem 2.7

Proof. The proof of part *i)* is by induction on the structure of instructions.
We have the following cases.

- Case $c = i_1;i_2$

  Since $i_1;i_2$ is typable, then by rule $cseq_b$ we have that:

  i) $ls \vdash_b i_1 : pc_1 \rightsquigarrow ls'$
  ii) $ls' \vdash_b i_2 : pc_2 \rightsquigarrow ls''$
  iii) $ls \vdash_b i_1\, ;\, i_2 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls''$

  Using that $\mathtt{min}\, pc_1\, pc_2 \leq pc_1$ and $\mathtt{min}\, pc_1\, pc_2 \leq pc_2$, the rule SUB and the items i) and ii) we conclude that:

  $ls \vdash_b i_1 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls'$ and $ls' \vdash_b i_2 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls''$

  Then, by induction hypothesis we have that

  $ls \vdash i_1 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls'$ and $ls' \vdash i_2 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls''$

  Using rule CSEQ we conclude that $ls \vdash i_1\, ;\, i_2 : \mathtt{min}\, pc_1\, pc_2 \rightsquigarrow ls''$.

- Case $c = \mathit{branch}\ (i_1, i_2)$
  Assuming that c is typable, by the rules of the type system we have that:

  i) $ls \vdash_b i_1 : pc_1 \rightsquigarrow ls'$
  ii) $ls \vdash_b i_2 : pc_2 \rightsquigarrow ls'$
  iii) $low :: ls \vdash_b \mathit{branch}\ (i_1, i_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'$

  or

  i) $[\,] \vdash_b i_1 : high \rightsquigarrow [\,]$
  ii) $[\,] \vdash_b i_2 : high \rightsquigarrow [\,]$
  iii) $high :: ls \vdash_b \mathit{branch}\ (i_1, i_2) : pc \rightsquigarrow ls$

  If the first option is true then using the subtyping rule and induction hypothesis we have that:

  $ls \vdash i_1 : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'$ and $ls \vdash i_2 : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'$

  Then, we conclude that

  $$low :: ls \vdash \mathit{branch}\ (i_1, i_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'$$

  If the second option is true, by induction hypothesis we have that:

  $[\,] \vdash_b i_1 : high \rightsquigarrow [\,]$ and $[\,] \vdash_b i_2 : high \rightsquigarrow [\,]$

  Using the rule LOOP and B2 we conclude.

- Case $c = \mathit{loop}\ (i_1, i_2)$
  Similar to the previous case.

- The cases $c = \mathit{fetch}\ x$, $c = \mathit{store}\ x$ $c = \mathit{add}$ and $c = \mathit{noop}$ are trivial.

The proof of part *ii)* is also by induction on the structure of commands.
We have the following cases.

- Case $c = i_1;i_2$
  Since $i_1;i_2$ is typable, we have that:

  i) $ls \vdash i_1 : pc \rightsquigarrow ls'$
  ii) $ls' \vdash i_2 : pc \rightsquigarrow ls''$

iii) $ls \vdash i_1 ; i_2 : pc \rightsquigarrow ls''$

We conclude that $ls \vdash_b i_1 ; i_2 : pc \rightsquigarrow ls''$ using first induction hypothesis and then $cseq_b$ rule.

- case $c = branch\ (i_1, i_2)$

  Since c is typable, we have that:

  i) $ls \vdash i_1 : pc \rightsquigarrow ls'$
  ii) $ls \vdash i_2 : pc \rightsquigarrow ls'$
  iii) $low :: ls \vdash branch\ (i_1, i_2) : pc \rightsquigarrow ls'$

  or

  i) $[\,] \vdash i_1 : high \rightsquigarrow [\,]$
  ii) $[\,] \vdash i_2 : high \rightsquigarrow [\,]$
  iii) $high :: ls \vdash branch\ (i_1, i_2) : pc \rightsquigarrow ls$

  If the first option is true by induction hypothesis we have that:
  $ls \vdash_b i_1 : pc \rightsquigarrow ls'$ and $ls \vdash_b i_2 : pc \rightsquigarrow ls'$

  Then, we conclude by the BRANCH$_b$ rule that:

  $$low :: ls \vdash branch\ (i_1, i_2) : pc \rightsquigarrow ls$$

  If the second option is true we conclude similarly that:

  $$high :: ls \vdash branch\ (i_1, i_2) : pc \rightsquigarrow ls$$

- case $c = loop\ (i_1, i_2)$

  Similar to the previous case.

- The other cases are trivial.

□

5.5 Proof of Theorem 2.9

Proof. We will prove this property by induction on a derivation of $\langle c,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle$.

- If the last rule used in the derivation is E-SEQ1, then we know that c has the form $c_1\,;c_2$ and:

  i) $\langle c_1,\ \sigma,\ s\rangle \rhd (\sigma', s')$
  ii) $\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c_2,\ \sigma',\ s'\rangle$

  Since $c_1\,;c_2$ is typable, by rule CSEQ we have that:

  iii) $ls \vdash c_1 : pc \rightsquigarrow ls'$
  iv) $ls' \vdash c_2 : pc \rightsquigarrow ls''$
  v) $ls \vdash c_1\,;c_2 : pc \rightsquigarrow ls''$

  We conclude by item iv).

- If the last rule used in the derivation is E-SEQ2, then we know that c has the form $c_1\,;c_2$ and:

  i) $\langle c_1,\ \sigma,\ s\rangle \rhd \langle c_1',\ \sigma',\ s'\rangle$
  ii) $\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c_1'\,;c_2,\ \sigma',\ s'\rangle$

  Since $c_1\,;c_2$ is typable we also have the hipothesis iii), iv) and v) of the previous item.

  Using i), iii) we conclude by induction hypothesis that:

  $$\exists\ ls''' \cdot \quad ls''' \vdash c_1' : pc \rightsquigarrow ls'$$

  Then, by rule CSEQ we conclude that $ls''' \vdash c_1'\,;c_2 : pc \rightsquigarrow ls''$.

- If the last rule used in the derivation is E-BRANCH1, then we know that c has the form $branch\ (c_1, c_2)$ and:

  $$\langle branch\ (c_1, c_2),\ 0 : \sigma,\ s\rangle \rhd \langle c_1,\ \sigma,\ s\rangle$$

  Since the code is typable, we have that:

i) $ls \vdash c_1 : pc \rightsquigarrow ls'$

ii) $ls \vdash c_2 : pc \rightsquigarrow ls'$

iii) $low :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls'$

or

i) $[\,] \vdash c_1 : high \rightsquigarrow [\,]$

ii) $[\,] \vdash c_2 : high \rightsquigarrow [\,]$

iii) $high :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls$

If the first option is true we conclude using i).

If the second option is true, then using i) and that $pc \leq High$ we conclude that

$ls \vdash c_1 : pc \rightsquigarrow ls$

- The case where the last rule used is E-Branch2 is similar to the previous case.

- When the last rule used in the derivation is E-Loop, then the code must be of the form $loop\ (c_1, c_2)$ and we have that:

$$\langle loop\ (c_1, c_2),\ \sigma,\ s\rangle \rhd \langle c_1\,; branch\ (c_2\,; loop\ (c_1, c_2), noop),\ \sigma,\ s\rangle$$

Since $loop\ (c_1, c_2)$ is typable, we conclude by rule loop that:

i) $[\,] \vdash c_1 : pc \rightsquigarrow high :: [\,]$

ii) $[\,] \vdash c_2 : high \rightsquigarrow [\,]$

iii) $ls \vdash loop\ (c_1, c_2) : high \rightsquigarrow ls$

or

i) $ls \vdash c_1 : pc \rightsquigarrow low :: ls'$

ii) $ls' \vdash c_2 : pc \rightsquigarrow ls''$

iii) $ls \vdash loop\ (c_1, c_2) : pc \rightsquigarrow ls''$

If the first case is true, then using i) y ii) and the type system we conclude that:

$$[\,] \vdash loop\ (c_1, c_2) : high \rightsquigarrow [\,]$$

Then by the item ii) and the rule CSEQ we have that

$$[\,] \vdash c_2\,; loop\ (c_1, c_2) : high \rightsquigarrow [\,]$$

The, we can conclude that

$$high :: [\,] \vdash branch\ (c_2\,; loop\ (c_1, c_2), noop) : high \rightsquigarrow [\,]$$

By item i) and the rule CSEQ we have that

$$[\,] \vdash c_1\,; branch\ (c_2\,; loop\ (c_1, c_2), noop) : high \rightsquigarrow [\,]$$

Then we can conclude trivially that

$$ls \vdash c_1\,; branch\ (c_2\,; loop\ (c_1, c_2), noop) : high \rightsquigarrow ls$$

If the second case is true the proof is similar.

□

5.6 Proof of Lemma 2.10

Proof. The proof is by structural induction on the code c.

- For the codes $push\ n$, $fetch\ x$, $add\ branch\ (c_1, c_2)$ and $loop\ (c_1, c_2)$ we conclude trivially since $s' = s$.

- If $c = store\ x$ then

$$\langle store\ x,\ z : \sigma,\ s\rangle \triangleright (\sigma, s[x \mapsto z])$$

Since c type in a high context by the type system we have that

$st :: ls \vdash store\ x : high \rightsquigarrow ls$ and $\Gamma(x) = high$

Then, $s \cong_L s[x \mapsto z]$.

- If $c = c_1\,;c_2$, we have that c_1 and c_2 also type in high contexts.
  By induction hypothesis exists $\sigma'\ s'\ c'$ such that,

  $$\langle c_1,\ \sigma,\ s\rangle \rhd \langle c',\ \sigma',\ s'\rangle\ \wedge\ s \cong_L s'$$

  or exists $\sigma'\ s'$ such,

  $$\exists\, \sigma'\ s'.\langle c_1,\ \sigma,\ s\rangle \rhd (\sigma',\ s')\ \wedge\ s \cong_L s'$$

  If the first case is true by the semantic definition we have that
  $\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c'\,;c_2,\ \sigma',\ s'\rangle$
  If the second case is true then
  $\langle c_1\,;c_2,\ \sigma,\ s\rangle \rhd \langle c_2,\ \sigma',\ s'\rangle$

□

5.7 Proof of Lemma 2.12

Proof. The proof is by structural induction on the code c.

- If $c = push\ n$, then

  $$\langle push\ n,\ e_1,\ s_1\rangle \rhd (\mathcal{N}[\![n]\!] : e_1,\ s_1)$$

  and

  $$\langle push\ n,\ e_2,\ s_2\rangle \rhd (\mathcal{N}[\![n]\!] : e_2,\ s_2)$$

  We conclude trivially that $\mathcal{N}[\![n]\!] : e_1 \cong_L \mathcal{N}[\![n]\!] : e_2$ and $s_1 \cong_L s_2$.

- If $c = fetch\ x$ we have that:

  $$\langle fetch\ x,\ e_1,\ s_1\rangle \rhd ((s_1\ x) : e_1,\ s_1)$$

  and

  $$\langle fetch\ x,\ e_2,\ s_2\rangle \rhd ((s_2\ x) : e_2,\ s_2)$$

  We have two cases, if $\Gamma(x) = low$ then $s_1\ x = s_2\ x$ and $e_1' \cong_L e_2'$ hold. If $\Gamma(x) = high$ we have that $e_1' \cong_L e_2'$ since $e_1 \cong_L e_2$. We conclude also trivially that $s_1 \cong_L s_2$

- If $c = store\ x$ then

  $$\langle store\ x,\ z_1 : e_1,\ s_1\rangle \rhd (e_1, s_1[x \mapsto z_1])$$

  $$\langle store\ x,\ z_2 : e_2,\ s_2\rangle \rhd (e_2, s_2[x \mapsto z_2])$$

  and

  $$st :: \Sigma \vdash store\ x : pc \rightsquigarrow \Sigma$$

  with $st \le \Gamma(x)$ and $pc \le \Gamma(x)$. We have that $e'_1 \cong_L e'_2$ trivially hold. Now, if $st = low$, we have that $z_1 = z_2$ and conclude that $s_1[x \mapsto z_1] \cong_L s_2[x \mapsto z_2]$. If $st = high$, then $\Gamma(x) = high$ holds and $s'_1 \cong_L s'_2$ is consequence of $s_1 \cong_L s_2$.

- If $c = branch\ (c_1, c_2)$ then since the code is typable, we have that:

  i) $ls \vdash c_1 : pc \rightsquigarrow ls'$
  ii) $ls \vdash c_2 : pc \rightsquigarrow ls'$
  iii) $low :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls'$

  or

  i) $[\,] \vdash c_1 : high \rightsquigarrow [\,]$
  ii) $[\,] \vdash c_2 : high \rightsquigarrow [\,]$
  iii) $high :: ls \vdash branch\ (c_1, c_2) : pc \rightsquigarrow ls$

  If the first case is true, then we have that $z_1 = z_2$ (since $st = low$ and $z_1 :: e_1 \cong_L z_2 :: e_2$), by the evaluation rules we have that:

  $$\langle branch\ (c_1, c_2),\ z_1 :: e_1,\ s_1\rangle \rhd \langle c_1,\ e_1,\ s_1\rangle$$
  $$\langle branch\ (c_1, c_2),\ z_2 :: e_2,\ s_2\rangle \rhd \langle c_1,\ e_2,\ s_2\rangle$$

  or

  $$\langle branch\ (c_1, c_2),\ z_1 :: e_1,\ s_1\rangle \rhd \langle c_2,\ e_1,\ s_1\rangle$$
  $$\langle branch\ (c_1, c_2),\ z_2 :: e_2,\ s_2\rangle \rhd \langle c_2,\ e_2,\ s_2\rangle$$

  Then, we can conclude trivially that the lemma hold.

  If the second case is true we have that $st = high$. The evaluation of $\langle branch\ (c_1, c_2),\ z_1 :: e_1,\ s_1\rangle$ and $\langle branch\ (c_1, c_2),\ z_2 :: e_2,\ s_2\rangle$ depend on z_1

and z_2. We will prove just the case where $z_1 = 0$ and $z_2 \neq 0$, the other cases are similar to previous cases.

By the evaluation rules we have that:

$$\langle branch\ (c_1, c_2),\ z_1 : e_1,\ s_1\rangle \rhd \langle c_1,\ e_1,\ s_1\rangle$$
$$\langle branch\ (c_1, c_2),\ z_2 : e_2,\ s_2\rangle \rhd \langle c_2,\ e_2,\ s_2\rangle$$

We use the item i) and Confinment to conclude that exists s' such that:

$$\langle c_1,\ [\,],\ s_1\rangle \rhd^* ([\,],\ s') \ \wedge\ s_1 \cong_L s'$$

Then, we can conclude that

$$\langle c_1,\ e_1,\ s_1\rangle \rhd^* (e_1,\ s') \ \wedge\ s_1 \cong_L s'$$

We also use Confinment and item ii) to conclude that exists s'' such that:

$$\langle c_2,\ [\,],\ s_2\rangle \rhd^* ([\,],\ s'') \ \wedge\ s_2 \cong_L s''$$

Then, we conclude that

$$\langle c_2,\ e_2,\ s_2\rangle \rhd^* (e_2,\ s'') \ \wedge\ s_2 \cong_L s''$$

By transitivity we have that $s' \cong_L s''$.

- If $c = c_1\,;c_2$, we have that

  i) $ls \vdash c_1 : pc \rightsquigarrow ls'$
  ii) $ls' \vdash c_2 : pc \rightsquigarrow ls''$
  iii) $ls \vdash c_1\,;c_2 : pc \rightsquigarrow ls''$

  By induction hypothesis we have that exists s'_1, s'_2, e'_1, e'_2 such,

$$\langle c_1,\ e_1,\ s_1\rangle \rhd^* (e'_1,\ s'_1) \ \wedge\ \langle c_1,\ e_2,\ s_2\rangle \rhd^* (e'_2,\ s'_2) \ \wedge\ e'_1 \cong_L e'_2 \ \wedge\ s'_1 \cong_L s'_2$$

  or, exists $c', s'_1, s'_2, e'_1, e'_2$ such that:

$$\langle c_1,\ e_1,\ s_1\rangle \rhd^* \langle c',\ e'_1,\ s'_1\rangle \wedge \langle c_1,\ e_2,\ s_2\rangle \rhd^* \langle c',\ e'_2,\ s'_2\rangle \wedge e'_1 \cong_L e'_2 \wedge s'_1 \cong_L s'_2$$

  In the first case we use the auxiliary lemma 5.1 to conclude that:

$$\langle c_1\,;c_2,\ e_1,\ s_1\rangle \rhd^* \langle c_2,\ e'_1,\ s'_1\rangle \ \wedge\ \langle c_1\,;c_2,\ e_2,\ s_2\rangle \rhd^* \langle c_2,\ e'_2,\ s'_2\rangle$$

And in the second case we use the auxiliary lemma 5.2 to conclude that:

$$\langle c_1 ; c_2,\ e_1,\ s_1 \rangle \rhd^* \langle c' ; c_2,\ e'_1,\ s'_1 \rangle \wedge \langle c_1 ; c_2,\ e_2,\ s_2 \rangle \rhd^* \langle c' ; c_2,\ e'_2,\ s'_2 \rangle$$

- If $c = loop\ (c_1, c_2)$, we can do the proof for this case using a big step semantic and induction on the derivation. For simplicity, we omit this proof.

$\square$

Lemma 5.1. *For every $c \in \textbf{Code}$, e, e', and s, s', if $\langle c,\ e,\ s \rangle \rhd^* (e',\ s')$ then for any code c_2 we have that $\langle c ; c_2,\ e,\ s \rangle \rhd^* \langle c_2,\ e',\ s' \rangle$.*

Lemma 5.2. *For every $c \in \textbf{Code}$, e, e', and s, s', if $\langle c,\ e,\ s \rangle \rhd^* \langle c',\ e',\ s' \rangle$ then for any code c_2 we have that $\langle c ; c_2,\ e,\ s \rangle \rhd^* \langle c' ; c_2,\ e',\ s' \rangle$.*

Chapter 6

Complete code of Chapter 3

6.1 Monotony of fixS

```
---------------------------------------------------------------------------------
-- This lemma lema is neccesary to prove that if we have two chains of the form:
-- ⊥ Γ0''_1 Γ1''_1 Γ1''_1
-- ⊥ Γ0''_2 Γ1''_2 Γ2''_2 Γ3''_2 ..  Γn''_2
-- then if Γ_1 ⊑ Γ_2, Γ1''_1 ⊑ Γn''_2 is true

-- So, we have that Γi'' ⊑ Γn''
-----------------------------------------------------------------------------------

lema'' :        (X : Set) →
        (k : ℕ) →                  -- maximun number of iterations
        (_⊏_ : X → X → Set) →
        (_⊑_ : X → X → Set) →
        (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
        (str : {x y : X} → x ⊏ y → x ⊑ y) →
        (par : PartialOrder _⊑_ ) →

        (bound : X → ℕ) →    -- bound function
        -- the bound function is decreasing
        (boundDec : ∀ {x y : X} → x ⊏ y → bound y ¡ bound x) →
        -- the bound function has a unique minimal value
        (minimal : ∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →

        (g_1 : X → X) →
        (gmono_1 : ∀ {x y} → x ⊑ y → g_1 x ⊑ g_1 y) → -- g is monotone

        (g_2 : X → X) →
        (gmono_2 : ∀ {x y} → x ⊑ y → g_2 x ⊑ g_2 y) → -- g is monotone

        (x : X) →                 -- inicials values
        (x' : X) →                 -- inicials values
        (x'⊑g₂x' : x' ⊑ g₂ x') →
        (inv' : k ≥ bound x') →    -- invariant
        x ⊑ x' →
        (prop : ∀ {x₁ x₂} → x₁ ⊑ x₂ → g₁ x₁ ⊑ g₂ x₂) →
        x      ⊑ proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal
                      g₂ gmono₂ x' x'⊑g₂x' inv')

lema'' X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂
        x x' x'⊑g₂x' inv' x⊑x' prop
           with minimal x' (g₂ x') (n≡0 inv')
              (n≡0 (trans (propDec' bound rel⊏ rel⊑ boundDec x'⊑g₂x') inv'))
lema'' X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂
        x      x' x'⊑y' inv' x⊑x' prop | l = x⊑x'

lema'' X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂
        x x' x'⊑g₂x' inv' x⊑x' prop with sum x'⊑g₂x'
lema'' X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂
        x x' x'⊑g₂x' inv' x⊑x' prop | inl x'≡g₂x' = x⊑x'

... | inr x'⊏g₂x' =
   let r : k ≥ bound (g₂ x')
        r = p≤p (trans (boundDec x'⊏g₂x') inv')

        x⊑g₂x' : rel⊑ x (g₂ x')
        x⊑g₂x' = PartialOrder.trans₁ par x⊑x' x'⊑g₂x'
   in lema'' X k rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂
                       gmono₂ x (g₂ x') (gmono₂ x'⊑g₂x') r x⊑g₂x' prop


----------------------------------------------------------------------------
-- This lemma lema is neccesary to prove that if we have two chains of the form:
-- ⊥ Γ0''₁ Γ1''₁ Γ2''₁ Γ3''₁ ..  Γn''₁
-- ⊥ Γ0''₂ Γ1''₂ Γ1''₂
-- then if Γ₁ ⊑ Γ₂, Γn''₁ ⊑ Γ1''₁ is true
----------------------------------------------------------------------------

lema' :    (X : Set) →
        (k : ℕ) →                -- maximun number of iterations
        (_⊏_ : X → X → Set) →
        (_⊑_ : X → X → Set) →
        (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
        (str : {x y : X} → x ⊏ y → x ⊑ y) →
        (par : PartialOrder _⊑_ ) →

        (bound : X → ℕ) →   -- bound function
        -- the bound function is decreasing
        (boundDec : ∀ {x y : X} → x ⊏ y → bound y ¡ bound x) →
        -- the bound function has a unique minimal value
        (minimal : ∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →
```

```
            (g₁ : X → X) →
            (gmono₁ : ∀ {x y} → x ⊑ y → g₁ x ⊑ g₁ y) → -- g is monotone

            (g₂ : X → X) →
            (gmono₂ : ∀ {x y} → x ⊑ y → g₂ x ⊑ g₂ y) → -- g is monotone

            (x : X) →                 -- inicial value
            (x⊑g₁x : x ⊑ g₁ x) →
            (inv : k ≥ bound x) →     -- invariant

            (x' : X) →                -- inicial value
            x ⊑ x' →
            x' ≡ g₂ x' →
            (prop : ∀ {x₁ x₂} → x₁ ⊑ x₂ → g₁ x₁ ⊑ g₂ x₂) →
            proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal
               g₁ gmono₁ x x⊑g₁x inv) ⊑ x'

lema' X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
          x⊑g₁x inv x' x⊑x' x'≡g₂x' prop with minimal x (g₁ x) (n≡0 inv)
                       (n≡0 (trans (propDec' bound rel⊏   rel⊑ boundDec x⊑g₁x) inv))
lema' X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
          x⊑g₁x inv x' x⊑x' x'≡g₂x' prop | l = x⊑x'

lema' X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
          x⊑g₁x inv x' x⊑x' x'≡g₂x' prop   with sum x⊑g₁x
lema' X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
          x⊑g₁x inv x' x⊑x' x'≡g₂x' prop | inl x≡g₁x = x⊑x'
... | inr x⊏g₁x =
   let   inv₁ : k ≥ bound (g₁ x)
         inv₁ = p≤p (trans (boundDec x⊏g₁x) inv)

         g₁x⊑x' : rel⊑ (g₁ x) x'
         g₁x⊑x' = PartialOrder.trans₁ par (prop x⊑x')
                     (subst (λ y → rel⊑ y x') x'≡g₂x' (PartialOrder.refl₁ par { x = x'}))

   in lema' X k rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂
      gmono₂ (g₁ x) (gmono₁ x⊑g₁x) inv₁ x' g₁x⊑x' x'≡g₂x' prop


----------------------------------------------------------------
-------- This function is a generalization of fixSmonotone,
-------- it uses two monotone functions g1 and g2 instead of one
-----------------------------------------------------------------


fixSmonotone2 :   (X : Set) →
            (k : ℕ) →                 -- maximun number of iterations
            (_⊏_ : X → X → Set) →
```

```
        (_⊑_ : X → X → Set) →
        (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
        (str : {x y : X} → x ⊏ y → x ⊑ y) →
        (par : PartialOrder _⊑_ ) →

        (bound : X → ℕ) →   -- bound function
        -- the bound function is decreasing
        (boundDec : ∀ {x y : X} → x ⊏ y → bound y ¡ bound x) →
        -- the bound function has a unique minimal value
        (minimal : ∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →

        (g₁ : X → X) →
        (gmono₁ : ∀ {x y} → x ⊑ y → g₁ x ⊑ g₁ y) → -- g is monotone

        (g₂ : X → X) →
        (gmono₂ : ∀ {x y} → x ⊑ y → g₂ x ⊑ g₂ y) → -- g is monotone

        (x : X) →                    -- inicial value
        (x⊑g₁x : x ⊑ g₁ x) →
        (inv : k ≥ bound x) →        -- invariant
        (x' : X) →                   -- inicial value
        (x'⊑g₂x' : x' ⊑ g₂ x') →
        (inv' : k ≥ bound x') →      -- invariant
        x ⊑ x' → g₁ x ⊑ g₂ x' →
        (prop : ∀ {x₁ x₂} → x₁ ⊑ x₂ → g₁ x₁ ⊑ g₂ x₂) →
        proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal g₁ gmono₁ x x⊑g₁x inv) ⊑
        proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal g₂ gmono₂ x' x'⊑g₂x' inv')

fixSmonotone2 X 0 rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
            x⊑g₁x inv x' x'⊑g₂x' inv' x⊑x' g₁x⊑g₂x' prop with minimal x (g₁ x) (n≡0 inv)
               (n≡0 (trans (propDec' bound rel⊏ rel⊑ boundDec x⊑g₁x) inv))

... | x≡g₁x with minimal x' (g₂ x') (n≡0 inv')
                    (n≡0 (trans (propDec' bound rel⊏ rel⊑ boundDec x'⊑g₂x') inv'))
...    | x'≡g₂x' = x⊑x'

fixSmonotone2 X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
            x⊑g₁x inv x' x'⊑g₂x' inv' x⊑x' g₁x⊑g₂x' prop with sum x⊑g₁x
...    | inl x≡g₁x    with sum x'⊑g₂x'
...                     | inl x'≡g₂' = x⊑x'
...                     | inr x'⊏g₂x' =
   let x⊑g₂x' : rel⊑ x (g₂ x')
        x⊑g₂x' = PartialOrder.trans₁ par x⊑x' x'⊑g₂x'

        r :    k ≥ bound (g₂ x')
        r = p≤p (trans (boundDec x'⊏g₂x') inv')

   in    lema" X k rel⊏ rel⊑ sum str par bound boundDec minimal g₁
      gmono₁ g₂ gmono₂ x (g₂ x') (gmono₂ x'⊑g₂x') r x⊑g₂x' prop
```

```
fixSmonotone2 X (suc k) rel⊏ rel⊑ sum str par bound boundDec minimal g₁ gmono₁ g₂ gmono₂ x
            x⊑g₁x inv x' x'⊑g₂x' inv' x⊑x' g₁x⊑g₂x' prop | inr x⊏g₁x with sum x'⊑g₂x'
... | inl x'≡g₂x' =
   let      inv₁ : k ≥ bound (g₁ x)
            inv₁ = p≤p (trans (boundDec x⊏g₁x) inv)

            g₁x⊑x' :    rel⊑ (g₁ x) x'
            g₁x⊑x' = PartialOrder.trans₁ par g₁x⊑g₂x'
                   (subst (λ y → rel⊑ y x') x'≡g₂x'
                   (PartialOrder.refl₁ par {x = x'}))

   in lema' X k rel⊏ rel⊑ sum str par bound boundDec minimal g₁
                         gmono₁ g₂ gmono₂ (g₁ x) (gmono₁ x⊑g₁x) inv₁ x' g₁x⊑x' x'≡g₂x' prop

... | inr x'⊏g₂x' =
   let inv₁ : k ≥ bound (g₁ x)
       inv₁ = p≤p (trans (boundDec x⊏g₁x) inv)

       inv₂ : k ≥ bound (g₂ x')
       inv₂ = p≤p (trans (boundDec x'⊏g₂x') inv')

   in fixSmonotone2 X k rel⊏ rel⊑ sum str par bound boundDec
                          minimal g₁ gmono₁ g₂ gmono₂ (g₁ x) (gmono₁ x⊑g₁x) inv₁ (g₂ x')
                          (gmono₂ x'⊑g₂x') inv₂ (prop x⊑x') (prop (prop x⊑x')) prop


------------------------------------------
-------- Monotony of fixS
------------------------------------------

fixSmonotone :    (X : Set) →    -- partial order Set
        (k : ℕ) →                -- maximun number of iterations
        (_⊏_ : X → X → Set) →
        (_⊑_ : X → X → Set) →
        (sum : {x y : X} → x ⊑ y → (x ≡ y) ∨ (x ⊏ y)) →
        (str : {x y : X} → x ⊏ y → x ⊑ y) →

        (par : PartialOrder _⊑_ ) →

        (bound : X → ℕ) →        -- bound function
        -- the bound function is decreasing
        (boundDec : ∀ {x y : X} → x ⊏ y → bound y ¡ bound x) →
        -- the bound function has a unique minimal value
        (minimal : ∀ (x y : X) → bound x ≡ 0 → bound y ≡ 0 → x ≡ y) →

        (g : X → X) →
        (gmonotone : ∀ {x y} → x ⊑ y → g x ⊑ g y) → -- g is monotone
```

```
           (x : X) →                -- inicial value x
           (x⊑gx : x ⊑ g x) →
           (inv : k ≥ bound x) →    -- invariant
           (x' : X) →               -- inicial value x'
           (x'⊑gx' : x' ⊑ g x') →
           (inv' : k ≥ bound x') →   -- invariant
           x ⊑ x' → g x ⊑ g x' →
           proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal g gmonotone x x⊑gx inv) ⊑
           proj₁ (fixS X k _⊏_ _⊑_ sum str par bound boundDec minimal g gmonotone x' x'⊑gx' inv')

fixSmonotone X k rel⊏   rel⊑ sum str par bound boundDec minimal g gmonotone x
            x⊑gx inv x' x'⊑gx' inv' x⊑x' gx⊑gx' =
  fixSmonotone2 X k rel⊏ rel⊑ sum str par bound boundDec
                       minimal g gmonotone g gmonotone x x⊑gx inv x' x'⊑gx' inv' x⊑x'
                       gx⊑gx' gmonotone
```

6.2 Function fix (used in the While rule)

```
mutual
---------------------------------------------------
-------- Relation that implement the type system
--------------------------------------------------
    data Tc {n : ℕ} : ASTCom n → St → Vec St n → Vec St n → Set where
      Skip    : {Γ : Vec St n}{pc : St} →
         Tc SKIP pc Γ Γ

      Ass     : {x : Fin n}{e : ASTExp n}{Γ : Vec St n}{pc : St} →
         Tc (ASSIGN x e) pc Γ (change x Γ (pc ∪ (typeAstExp Γ e)))

      Seq     : {Γ Γ' Γ" : Vec St n}{pc : St} {s₁ s₂ : ASTCom n} →
         Tc s₁ pc Γ Γ'          →
         Tc s₂ pc Γ' Γ" →
         Tc (SEQ s₁ s₂) pc Γ Γ"

      If0     : {Γ Γ' Γ" : Vec St n}{pc : St}{e : ASTExp n}{s₁ s₂ : ASTCom n} →
         Tc s₁ (pc ∪ (typeAstExp Γ e)) Γ Γ' →
         Tc s₂ (pc ∪ (typeAstExp Γ e)) Γ Γ" →
         Tc (IF0 e s₁ s₂) pc Γ (Γ' ⊔ Γ")

      While : {Γ : Vec St n}{pc : St}{e : ASTExp n}{s : ASTCom n} →
         Tc (WHILE e s) pc Γ (proj₁ (fix {n} s e pc Γ))


---------------------------------------------------
-------- Function that implement the type system
--------------------------------------------------

    tc : {n : ℕ} → (s : ASTCom n) → (pc : St) → (Γ : Vec St n) →
      Σ (Vec St n) (λ Γ' → Tc s pc Γ Γ')

    tc (ASSIGN m e) pc Γ = change m Γ (pc ∪ (typeAstExp Γ e)) , Ass

    tc (SEQ s s') pc Γ = let (Γ' , tcs)    = tc s pc Γ
                             (Γ" , tcs') = tc s' pc Γ'
                         in  Γ" , Seq tcs tcs'


    tc (IF0 e s s') pc Γ = let pc' = pc ∪ (typeAstExp Γ e)
                                 (Γ' , tcs)    = tc s pc' Γ
                                 (Γ" , tcs') = tc s' pc' Γ
                           in    Γ' ⊔ Γ" , If0 tcs tcs'

    tc {n} (WHILE e s) pc Γ = proj₁ (fix {n} s e pc Γ) , While

    tc {n} SKIP pc Γ = Γ , Skip
```

```
----------------------
--- Monotony of tc
------------------------

tcMonotone : {n : ℕ}{pc pc' : St}{Γ Γ₁ : Vec St n}
             (s : ASTCom n) →
             pc ≤St pc' → Γ ⊑ Γ₁ → proj₁ (tc s pc Γ) ⊑ proj₁ (tc s pc' Γ₁)

tcMonotone   (ASSIGN m x₁) pc≤pc' Γ⊑Γ₁ =
  lemaChange {m = m} (lema≤St pc≤pc' (typeAstExp≤ x₁ Γ⊑Γ₁)) Γ⊑Γ₁

tcMonotone (IF0 e s₁ s₂) pc≤pc' Γ⊑Γ₁ =
  let pc₁≤pc₁' = lema≤St pc≤pc' (typeAstExp≤ e Γ⊑Γ₁)
  in lema⊑⊔ (tcMonotone s₁ pc₁≤pc₁' Γ⊑Γ₁)
             (tcMonotone s₂ pc₁≤pc₁' Γ⊑Γ₁)

tcMonotone (WHILE e s) pc≤pc' Γ⊑Γ₁ = fixMonotone s e pc≤pc' Γ⊑Γ₁

tcMonotone (SEQ s₁ s₂) pc≤pc' Γ⊑Γ₁ =
  let Γ₁'⊑Γ₂' = tcMonotone s₁ pc≤pc' Γ⊑Γ₁
  in tcMonotone s₂ pc≤pc' Γ₁'⊑Γ₂'

tcMonotone SKIP pc≤pc' Γ⊑Γ₁ = Γ⊑Γ₁

---------------------------
------ function body
------------------------

body : {n : ℕ} (e : ASTExp n) →
       ASTCom n →
       St →    Vec St n → Vec St n →    Vec St n

body e s pc Γ Γ' = let st = typeAstExp Γ' e
                       Γ" = proj₁ (tc s (pc ∪ st) Γ')
                       in Γ" ⊔ Γ


----------------------------------
----------------Monotony of body
---------------------------------

bodyMonotone : {n : ℕ}
  {e : ASTExp n}
  {s : ASTCom n}
  {pc pc' : St}
  {Γ Γ' Γ₁ Γ₁' : Vec St n} →
  pc ≤St pc' →
```

```
    Γ ⊑ Γ₁ → Γ' ⊑ Γ₁' →
    body e s pc Γ Γ' ⊑ body e s pc' Γ₁ Γ₁'


bodyMonotone {n}{e}{s}{pc}{pc'}{Γ}{Γ'}{Γ₁}{Γ₁'}
    pc≤pc' Γ⊑Γ₁ Γ'⊑Γ₁' =
    let pc₁≤pc₂ : pc ∪ typeAstExp Γ' e ≤St pc' ∪ typeAstExp Γ₁' e
        pc₁≤pc₂ = lema≤St pc≤pc' (typeAstExp≤ e Γ'⊑Γ₁')
    in  lema⊑⊔ (tcMonotone s pc₁≤pc₂ Γ'⊑Γ₁') Γ⊑Γ₁


-----------------------------------------
------ The function used in the While rule
-------------------------------------------


fix : {n : ℕ} →
    (s : ASTCom n) →
    (e : ASTExp n) →
    (pc : St) →
    (Γ : Vec St n) →
    Σ (Vec St n) (λ Γ' → Γ' ≡ body e s pc Γ Γ')

fix {n} s e pc Γ =
    let Γ₀ = Γ
          Γ₁ = body e s pc Γ Γ₀
    in fixS (Vec St n)
            -- partial order
            n _⊏_ _⊑_ ≡∨⊏ xs⊏ys parOrd⊑
            -- bound functions and properties
            sumLows sumLowDec minimalEnv
            (body e s pc Γ) -- function g
            (bodyMonotone {e = e} {s = s} (≤St-refl {st = pc}) (refl⊑ {Γ = Γ}))
            Γ₀ lema1 (lemasL Γ₀)

-----------------------
--- Monotony of fix
----------------------

    fixMonotone : {n : ℕ}{pc pc' : St}{Γ Γ' : Vec St n} →
        (s : ASTCom n) →
        (e : ASTExp n) →
        pc ≤St pc' →
        Γ ⊑ Γ' →
        proj₁ (fix s e pc Γ) ⊑ proj₁ (fix s e pc' Γ')

    fixMonotone {n} {pc}{pc'}{Γ}{Γ'} s e pc≤pc' Γ⊑Γ' =
        fixSmonotone2 (Vec St n) n _⊏_ _⊑_ ≡∨⊏ xs⊏ys parOrd⊑
            sumLows sumLowDec minimalEnv
            (body e s pc Γ)     -- function g₁
```

```
(bodyMonotone {e = e} {s = s} (≤St-refl {st = pc}) (refl⊑ {Γ = Γ}))
(body e s pc' Γ')    -- function g₂
(bodyMonotone {e = e} {s = s} (≤St-refl {st = pc'}) (refl⊑ {Γ = Γ'}))
Γ lema1 (lemasL Γ)
Γ' lema1 (lemasL Γ') Γ⊑Γ'
(bodyMonotone {e = e} {s = s} pc≤pc' Γ⊑Γ' Γ⊑Γ')
(bodyMonotone {e = e} {s = s} pc≤pc' Γ⊑Γ')
```

Bibliography

[1] Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In J. Gregory Morrisett and Manuel Fähndrich, editors, *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Long Beach, CA, USA, January 10, 2005*, pages 103–112. ACM, 2005.

[2] Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Comput. Lang. Syst. Struct.*, 33(2):35–59, July 2007.

[3] D. E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp., 1975.

[4] Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer, 2008.

[5] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.

[6] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[8] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. *SIGPLAN Not.*, 43(9):75–86, September 2008.

[9] Sebastian Hunt and David Sands. On flow-sensitive security types. *SIGPLAN Not.*, 41(1):79–90, January 2006.

[10] Sebastian Hunt and David 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, pages 297–316, 2011.

[11] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

[12] Ricardo Medel, Adriana B. Compagnoni, and Eduardo Bonelli. A typed assembly language for non-interference. In Mario Coppo, Elena Lodi, and G. Michele Pinna, editors, *ICTCS*, volume 3701 of *Lecture Notes in Computer Science*, pages 360–374. Springer, 2005.

[13] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[15] Ulf Norell. Dependently typed programming in Agda. In *4th international workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[16] Emir Pasalic and Nathan Linger. Meta-programming with typed object-language representations. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 136–167. Springer, 2004.

[17] Alejandro Russo and Andrei 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*, pages 186–199, 2010.

[18] Ando Saabas. *Logics for Low-Level Code and Proof-Preserving Program Transformations*. PhD thesis, Tallinn University of Technology, 2008.

[19] Ando Saabas and Tarmo Uustalu. Compositional type systems for stack-based low-level languages. In *Proceedings of the 12th Computing: The Australasian Theroy Symposium - Volume 51*, CATS '06, pages 27–39, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[21] Tim Sheard. Languages of the future. *SIGPLAN Not.*, 39(12):119–132, 2004.

[22] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[23] Dennis M. Volpano and Geoffrey 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, pages 607–621, London, UK, UK, 1997. Springer-Verlag.

[24] Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.