A security types preserving compiler in Haskell

Cecilia Manzino[a], Alberto Pardo[b]

[a] *Departamento de Ciencia de la Compuación, Universidad Nacional de Rosario, Argentina*
[b] *Instituto de Computación, Universidad de la República, Montevideo, Uruguay*

Abstract

The analysis of information flow has become a popular technique for ensuring the confidentiality of data. An end-to-end confidentiality policy guarantees that private data cannot be inferred by the inspection of public data. A security property that ensures a kind of confidentiality is the noninterference property, which can be enforced by the use of security type systems where types correspond to security levels. In this paper we show the development of a compiler (written in Haskell) between a simple imperative language and semi-structured machine code, which preserves the property of noninterference. The compiler is based on the use of typed abstract syntax (implemented in terms of Haskell GADTs and type-level functions) to encode the security type system of both the source and target language. This makes it possible to use Haskell's type checker to verify two things: that programs in both languages satisfy the security property, and that the compiler is correct by construction (in the sense that it preserves noninterference).

1. Introduction

The confidentiality of the information manipulated by computing systems has became of significant importance with the increasing use of web applications. Even though these applications are widely used, there is little assurance that there is no leakage of confidential information to public output.

A technique that has been widely used in the last years for ensuring the confidentiality of information is the analysis of information flow [1]. This technique analyses information flows between inputs and outputs of systems. Flows can be explicit or implicit. A flow from a variable x to a variable y is considered explicit if the value of x is transferred directly to y. On the other hand, it is implicit when the flow from x to y is generated by the control structure of the program.

In this paper we deal with the security property of *noninterference* [2]. This property guarantees that private data cannot be inferred by inspecting public

Email addresses: ceciliam@fceia.unr.edu.ar (Cecilia Manzino), pardo@fing.edu.uy (Alberto Pardo)

channels of information. This implies that a variation of private data does not cause a variation of public data.

There are different approaches to ensure this security property. In this paper we follow a type-based approach which relies on the use of a security type system [3]. In this setting, variables are classified in different categories according to which kind of data they can store (for example, public or confidential). Modelling security properties in terms of types has the advantage that the property can be checked at compile-time (during type checking). The overhead of checking the property at run-time is thus partially reduced or even eliminated.

Although there is an important amount of work on type systems for noninterference on high-level languages (e.g. [4, 3, 5]), there is little work on type systems for noninterference on low-level languages. This is a consequence of the lack of structure of low-level languages, which make them difficult to reason about. Some of the existing works on security type systems for low-level languages use code annotations to simulate the block structure of high-level languages at the low-level [6, 7]. Others are based on the use of a basic implicit structure present in low-level code [8].

The aim of this paper is to perform a simple, but nontrivial excercise: to show that it is possible to write in a general purpose (functional) language like Haskell a compiler that preserves the property of noninterference. We do so by using Haskell plus some minor extensions, such as GADTs[1], type families and multi-parameter type classes, which open us the possibility to perform some kind of type-level programming. The compiler we present translates programs in a simple imperative high-level language (with loops and conditionals) into programs in a semi-structured low-level language. We define the notion of noninterference for each of these languages by the specification of a security type system. Those type systems are represented in Haskell in terms of GADTs. We then prove that the compiler preserves the security property, i.e., secure programs in the source language are translated into secure programs in the target language. The novelty of the approach is that the preservation proof is automatically checked by Haskell's type system. That way we are proving, *once and for all*, that the compiler is correct with respect to the property it preserves. In other words, we are proving that the compiler is *correct-by-construction*. This is a non-standard application of Haskell which resembles a language with dependent types. We developed our implementation in the Glasgow Haskell Compiler (GHC) using some of its extensions.

The paper is organized as follows. In Section 2 we present the high-level language that serves as source language to our compiler and define a security type system for it. We also describe how to encode the security type system in Haskell. In section 3 we do the same with the low-level language that serves as target language. We also present an encoding of this language and its secure type system in Haskell. Section 4 presents the compiler and the proof that it preserves security typing. Section 5 discusses related work and Section 6

[1] Generalized Algebraic Data Types

concludes the paper.

2. Source Language

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

2.1. Syntax

As source language we consider a simple imperative language formed by expressions and statements 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 $e \in \mathbf{Exp}$ (expresions) and $S \in \mathbf{Stm}$ (statements). Variables range over identifiers ($x \in \mathbf{Var}$) whereas n ranges over integer literals ($n \in \mathbf{Num}$).

For the sake of simplicity we have not included booleans in the language, thus giving the conditions of `if` and `while` statements in terms of arithmetic expressions.

2.2. Big-step semantics

The semantics we present for this language is completely standard [9]. 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 which contains the current value of each variable.

The semantics of expressions is given by 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 shown in Figure 1.

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`.

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

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

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

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

Figure 1: Big-step semantics of statements

2.3. 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. Our language is *flow insensitive* in the sense that the security level of each variable does not changed during program execution.

For simplicity, in this paper we consider just two security levels, *low* and *high*, corresponding to *public* and *confidential* data, respectively, but the whole development can be easily generalized to a lattice of security levels ordered by their degree of confidentiality. As usual $\mathit{low} \leq \mathit{high}$. Instead of $\Gamma(x) = \mathit{low}$ (high), we will simply write x_L (x_H) to mean a variable with *low* (*high*) security level and refer to it as a low (high) variable.

Noninterference 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. The property can be formulated as a semantic condition on programs. 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 low variables).

Definition 1 (Noninterference source language). *A program $S \in \mathbf{Stm}$ is* ***noninterfering*** *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:*

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

Nowadays it is well-known that this property can be enforced statically by the definition of an information-flow type system in which security levels are

EXPRESSIONS

$$\frac{}{\vdash e : high}\ \textsc{expH} \qquad \frac{x_H \notin Vars(e)}{\vdash e : low}\ \textsc{expL}$$

STATEMENTS

$$\frac{\vdash e : low}{[low] \vdash x_L := e}\ \textsc{assL} \qquad \frac{}{[pc] \vdash x_H := e}\ \textsc{assH}$$

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

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

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

Figure 2: Security Type System with subsumption (Source Language)

used as types and referred to as *security types*. This started with the work of Volpano et al. [4, 10]. Figures 2 and 3 present two alternative security type systems for our source language. The difference between them is that the system in Figure 3 is syntax-directed.

Expressions. For typing expressions in the system of Figure 2 we use a judgement of the form $\vdash e : st$, where $st \in \{low, high\}$. Rule EXPH states that 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 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.

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.

$$y_L := x_H + 1$$

EXPRESSIONS

$$\frac{}{\vdash_{sd} n : low} \qquad \frac{}{\vdash_{sd} x_L : low} \qquad \frac{}{\vdash_{sd} x_H : high} \qquad \frac{\vdash_{sd} e : st \quad \vdash_{sd} e' : st'}{\vdash_{sd} e + e' : \texttt{max}\ st\ st'}$$

STATEMENTS

$$\frac{\vdash e : st}{[high] \vdash_{sd} x_H := e}\ \text{ASSH}^{sd} \qquad \frac{\vdash e : low}{[low] \vdash_{sd} x_L := e}\ \text{ASSL}^{sd}$$

$$\frac{}{[high] \vdash_{sd} \texttt{skip}}\ \text{SKIP}^{sd} \qquad \frac{[pc_1] \vdash_{sd} S_1 \quad [pc_2] \vdash_{sd} S_2}{[\texttt{min}\ pc_1\ pc_2] \vdash_{sd} S_1;S_2}\ \text{SEQ}^{sd}$$

$$\frac{\vdash e : st \quad [pc_1] \vdash_{sd} S_1 \quad [pc_2] \vdash_{sd} S_2 \quad st \leq \texttt{min}\ pc_1\ pc_2}{[\texttt{min}\ pc_1\ pc_2] \vdash_{sd} \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\ \text{IF}^{sd}$$

$$\frac{\vdash e : st \quad [pc] \vdash_{sd} S \quad st \leq pc}{[pc] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S}\ \text{WHILE}^{sd}$$

Figure 3: Syntax-directed Security Type System (Source Language)

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.

$$x_H := y_L$$

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:

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

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:

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

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 is based on similar systems given in [4, 3]. 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 3. The typing judgement in this new system has the form $[pc] \vdash_{sd} S$. Our interest in having this other formulation of the type system is due to an important property of the syntax-directed system, namely that the last rule in every derivation of $[pc] \vdash_{sd} S$ is uniquely determined by pc and S. This property makes this system particularly appropriate to to be considered for the implementation.

Rule ASSH^{sd} states that assignments to high variables need to be performed in *high* contexts. In this type system a *high* context does not restrict typing, since by other rules (mainly those for conditional and loop) a code with *high* context can be part of any typable code. On the other hand, rule ASSL^{sd} states that assignments to low variables can only be done in *low* contexts. Explicit flows are prevented by this rule due to the restriction to the expression to be low. The rule IF^{sd} (WHILE^{sd}) imposes a restriction between the security level of the condition and the branches of the conditional (the body of the iteration). As a consequence, if the condition is high then the branches of the conditional (body of the while) must type in *high* contexts. This restriction in conjunction with that on assignments to low variables prevent implicit flows.

The following theorem establishes the relationship between both type systems.

Theorem 1. *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 Appendix A.

A desirable property for the security type system is type soundness, which means that every typable statement satisfies noninterference.

Theorem 2 (Type soundness). *For every $S \in \mathbf{Stm}$, $s_i, s_i', s_f, s_f', pc \in \mathbf{SType}$,*

$$[pc] \vdash_{sd} S \ \wedge \ s_i \cong_L s_i' \ \wedge \ \langle S, s_i \rangle \Downarrow s_f \ \wedge \ \langle S, s_i' \rangle \Downarrow s_f' \implies s_f \cong_L s_f'$$

2.4. Implementation

Following the approach of Sheard [11] and Pasalic and Linger [12], we implement the security type system in Haskell by encoding the typing judgements as GADTs. The constructors of the GADT then encode the typing rules. A

value of the GADT thus represents a derivation of the encoded judgement. A nice property of this encoding is that, the security property that is enforced by the type system of the object language (our source language) results checked by the type system of the host language (Haskell). This is a technique widely used nowadays.

Generalized Algebraic Data Types (GADTs) [13] are a generalization of the ordinary algebraic datatypes available in functional languages such as Haskell, ML or O'Caml. We explain the features that GADTs incorporate while showing the encoding of the type judgement for the arithmetic expressions of our language. Let us start considering the datatype definition of the abstract syntax for expressions in Haskell:

```
data Exp = IntLit Int | Var V | Add Exp Exp
```

where `V` is the type of variable names. The first ingredient that GADTs introduce is an alternative syntax for datatype declarations, where an explicit type signature is given for every data constructor. So, using GADT syntax we can define `Exp` as:

```
data Exp where
  IntLit :: Int  → Exp
  Var    :: V    → Exp
  Add    :: Exp  → Exp  → Exp
```

The second feature that GADTs incorporate is even more important. GADTs remove the restriction present in parameterized algebraic datatypes by means of which the return type of every data constructor must be the same polymorphic instance of the type constructor being defined (i.e. the type constructor applied to exactly the same type variables as in the left-hand side of the datatype definition). In a GADT, the return type of the data constructors continues being an application of the same type constructor that is being defined, but, in contrast to standard datatype definitions, its arguments can be arbitrary. This is the essential feature that makes it possible to encode the type judgements as GADTs, together with the fact that the encoded type systems are syntax-directed and therefore type judgements reflect the structure of abstract syntax definitions.

We represent the type system for expressions as a GADT `Exp st`, similar to the one above for the abstract syntax, but with the addition of a type parameter `st` that denotes the security type of the encoded expression. The encoding is such that, the judgement $\vdash e : st$ in our formal type system corresponds to the typing judgement `e :: Exp st` in Haskell. We represent the security types *low* and *high* in Haskell as empty types (i.e. datatypes with no constructors):

```
data Low
data High
```

The reason for using empty types is because we are only interested in computing with them at the type level. In fact, security types are only necessary to perform

the static verification of the noninterference property on programs. They are not necessary at runtime.

The GADT for expressions is then the following:

```
data Exp st where
   IntLit :: Int     → Exp Low
   VarL   :: VL      → Exp Low
   VarH   :: VH      → Exp High
   Add    :: Exp st  → Exp st'  → Exp (Max st st')
```

where `VL` and `VH` are types for identifiers of low and high variables, respectively. The definition of disjoint sets for low and high variables (and consequently the definition of a constructor to each case) simplifies the implementation of the language, avoiding the necessity of supplying a typing environment with the security type of each variable. Notice that a separate treatment of each kind of variable had been already given in the definition of the formal type system.

In the encoding, the maximun of two security types is computed by means of the following type level function (called a *type family* [14] in Haskell's jargon):

```
type family Max st st' :: ∗
type instance Max Low x  = x
type instance Max High x = High
```

To model statements we define a GADT that is also parametrized by a security type, but in this case it represents the security level of the context in which a statement is executed.

```
data Stm pc where
   AssH  :: VH      → Exp st   → Stm High
   AssL  :: VL      → Exp Low  → Stm Low
   Skip  :: Stm High
   Seq   :: Stm pc  → Stm pc'  → Stm (Min pc pc')
   If    :: LEq st (Min pc pc')
         ⇒ Exp st  → Stm pc    → Stm pc'  → Stm (Min pc pc')
   While :: LEq st pc
         ⇒ Exp st  → Stm pc    → Stm pc
```

Each constructor corresponds to a rule of the type system shown in Figure 3. Now the typing judgement `stm :: Stm pc` in Haskell corresponds to the judgement $[pc] \vdash_{sd} S$ in the formal type system. It is worth noting that since `Stm pc` encodes the typing rules, it is only possible to write terms that correspond to secure programs. Insecure programs will be rejected by Haskell's compiler because they correspond to ill-typed terms.

The minimum of two security types is computed by means of the following type level function:

```
type family Min st st' :: ∗
type instance Min Low x  = Low
type instance Min High x = x
```

The class `LEq` is defined for modelling at type level the condition $pc \leq pc'$ that is used in the typing rules for conditional and loop. This is a class with no methods.

```
class LEq a b
instance LEq Low  b
instance LEq High High
```

Having this class it is not necessary to provide a proof of the inequality between two types. If the condition holds for given two types, the selection of the appropriate instance will be chosen by Haskell's type system.

3. Target Language

The target language of the compiler is a simple machine code that runs on a stack abstract machine in the style of the presented in [9]. In this section we describe its syntax and operational semantics and define a type system that enforces noninterference.

3.1. Syntax

The instructions of the low-level 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 high-level 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 operand stack in order to be used by operations.

3.2. 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

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

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

$$\langle \mathsf{noop},\ \sigma,\ s\rangle \rhd (\sigma,\ s)$$

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

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

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

Figure 4: Operational Semantics of the Target Language

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 4.

We define a *meaning relation*

$$\langle c, s\rangle \downarrow s' \textit{ iff } \langle c,\ \epsilon,\ s\rangle \rhd^* (\sigma', s')$$

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 what does it mean for a low-level program to be noninterfering.

Definition 2 (Noninterference Target Language).

$$\mathbf{NI}_{\mathbb{T}}(c) \stackrel{\text{df}}{=} \forall s_i, s_i'.\ s_i \cong_L s_i' \ \wedge\ \langle c, s_i\rangle \downarrow s_f \ \wedge\ \langle c, s_i'\rangle \downarrow s_f' \implies s_f \cong_L s_f'$$

3.3. Security Type System

The security type system of the target language is shown in Figure 5. 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 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 high-level language, this type system was designed in order to prevent explicit and implicit illegal flows. Rule STOREL, 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 (which must be *low*) prevents implicit flows. Rules BRANCH and LOOP also take care of implicit

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

$$\textsc{add}\quad st_1 :: st_2 :: ls \vdash add : high \rightsquigarrow \mathtt{max}\ st1\ st2 :: ls$$

$$\textsc{fetchL}\quad ls \vdash \mathsf{fetch}\ x_L : high \rightsquigarrow low :: ls$$

$$\textsc{fetchH}\quad ls \vdash \mathsf{fetch}\ x_H : high \rightsquigarrow high :: ls$$

$$\textsc{storeL}\quad low :: ls \vdash \mathsf{store}\ x_L : low \rightsquigarrow ls$$

$$\textsc{storeH}\quad st :: ls \vdash \mathsf{store}\ x_H : high \rightsquigarrow ls$$

$$\textsc{noop}\quad ls \vdash \mathsf{noop} : high \rightsquigarrow ls$$

$$\textsc{cseq}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow ls' \qquad ls' \vdash c_2 : pc_2 \rightsquigarrow ls''}{ls \vdash c_1\,;c_2 : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls''}$$

$$\textsc{branch}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow ls \qquad ls \vdash c_2 : pc_2 \rightsquigarrow ls \qquad st \leq \mathtt{min}\ pc_1\ pc_2}{st :: ls \vdash \mathsf{branch}\ (c_1, c_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls}$$

$$\textsc{loop}\ \frac{ls \vdash c_1 : pc_1 \rightsquigarrow st :: ls' \qquad ls' \vdash c_2 : pc_2 \rightsquigarrow ls' \qquad st \leq pc_2}{ls \vdash \mathsf{loop}\ (c_1, c_2) : \mathtt{min}\ pc_1\ pc_2 \rightsquigarrow ls'}$$

Figure 5: Security Type System for the Target Language

flows. Rule BRANCH, for example, requires that the security level of the program counters of the branches must be at least the security level of the value at the top of the stack (which is the value used to choose the branch to continue). Something similar happens with the loop construct in rule LOOP.

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

$$\mathsf{push}\ 1; \mathsf{branch}\ (\mathsf{push}\ 0, \mathsf{noop})$$

because of the restriction of rule BRANCH, which states that the branches of a conditional cannot change the stack of security types. In this case, the branch push 0, adds an element to the stack. If we decided to remove such a restriction, we should be careful that a code like the following one, clearly insecure, is rejected by the type system:

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

However, without the restriction of rule BRANCH this is a situation not easy to detect because the instruction store x_L occurs outside the conditional and depends on the code that comes before it. This problem is due to the way in which the storage of a value in a variable is performed in this language. In fact, at least two actions, rather than one, are required: one for allocating the value

(to be stored in the variable) in the stack, and another for moving that value to the variable.

A similar situation happens with rule LOOP. It rejects any loop $\mathsf{loop}\ (c_1, c_2)$ whose body c_2 changes the stack of security types.

In Section 4, 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.

The security type system for the low-level language is also sound.

Theorem 3 (Type soundness). *For every* $c \in$ **Code**, ls, ls', s_i, s_i', s_f, s_f', pc,

$$ls \vdash c : pc \rightsquigarrow ls' \ \wedge\ s_i \cong_L s_i' \ \wedge\ \langle c, s_i\rangle \downarrow s_f \ \wedge\ \langle c, s_i'\rangle \downarrow s_f' \implies s_f \cong_L s_f'$$

3.4. Implementation

We use type-level lists to represent the stacks of security types. Such lists can be defined by introducing the following empty types:

```
data Empty
data st :#: l
```

The type `Empty` represents the empty stack whereas a type `st :#: l` represents a stack with top element `st` and tail stack `l`.

The low-level language is encoded as a GADT that is parameterized by the security level of the context and the stacks of security types before and after the execution of the code.

```
data Code ls pc ls' where
   Push   :: Int  →  Code ls High (Low :#: ls)
   AddOp  :: Code (st1 :#: st2 :#: ls) High (Max st2 st1 :#: ls)
   FetchL :: VL   →  Code ls High (Low :#: ls)
   FetchH :: VH   →  Code ls High (High :#: ls)
   StoreL :: VL   →  Code (Low :#: ls) Low ls
   StoreH :: VH   →  Code (pc :#: ls) High ls
   Noop   :: Code ls High ls
   CSeq   :: Code ls pc1 ls'  →  Code ls' pc2 ls''
          →  Code ls (Min pc1 pc2) ls''
   Branch :: LEq  pc (Min pc1 pc2)
          ⇒ Code ls pc1 ls  →  Code ls pc2 ls
           →  Code (pc :#: ls) (Min pc1 pc2) ls
   Loop   :: LEq st pc2
          ⇒ Code ls pc1 (st :#: ls')  →  CodeS ls' pc2 ls'
          →  Code ls (Min pc1 pc2) ls'
```

The typing judgement `c :: Code ls pc ls'` in Haskell corresponds to the judgement $ls \vdash c : pc \rightsquigarrow ls'$ in the formal type system.

Expressions

$$\boldsymbol{C}_e[n] = \mathsf{push}\ n$$
$$\boldsymbol{C}_e[x] = \mathsf{fetch}\ x$$
$$\boldsymbol{C}_e[e_1 + e_2] = \boldsymbol{C}_e[e_1]\ ;\ \boldsymbol{C}_e[e_2]\ ;\ \mathsf{add}$$

Sentences

$$\boldsymbol{C}_S[x := e] = \boldsymbol{C}_e[e]\,;\mathsf{store}\ x$$
$$\boldsymbol{C}_S[\texttt{skip}] = \mathsf{noop}$$
$$\boldsymbol{C}_S[S_1;S_2] = \boldsymbol{C}_S[S_1]\,;\,\boldsymbol{C}_S[S_2]$$
$$\boldsymbol{C}_S[\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2] = \boldsymbol{C}_e[e]\,;\mathsf{branch}\ (\boldsymbol{C}_S[S_1],\,\boldsymbol{C}_S[S_2])$$
$$\boldsymbol{C}_S[\texttt{while}\ e\ \texttt{do}\ S] = \mathsf{loop}\ (\boldsymbol{C}_e[e],\,\boldsymbol{C}_S[S])$$

Figure 6: Compilation functions

4. Compilation

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 ($\boldsymbol{C}_e : \mathbf{Exp} \to \mathbf{Code}$) and the other for commands ($\boldsymbol{C}_S : \mathbf{Stm} \to \mathbf{Code}$). Figure 6 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 4 (Compiler correctness). *For any $e \in \mathbf{Exp}$, $S \in \mathbf{Stm}$, and state s it holds that:*

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

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

However, in this paper we are especially interested in another property of the compiler, namely the preservation of the noninterference by compilation. This means that, if we start with a noninterfering program in the source language, then the compiler returns a noninterfering program in the target language. This property can be expressed semantically.

Theorem 5 (Security preservation). *For any $e \in \mathbf{Exp}$ and $S \in \mathbf{Stm}$,*

i) $\mathbf{NI}_{\mathbb{T}}(\boldsymbol{C}_e[e])$

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

However, our interest in this paper is to establish this property in terms of the type systems.

Theorem 6 (Type-based security preservation). *For any $e \in \mathbf{Exp}$ and $S \in \mathbf{Stm}$,*

i) If $\vdash_{sd} e : st$ *then* $ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow st :: ls$

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

Proof. By induction on the structure of expressions and statements. See Appendix B.

4.1. Implementation

Both $\boldsymbol{C}_e$ and $\boldsymbol{C}_S$ can be easily implemented in Haskell.

```
compE :: Exp st  →  Code ls High (st :#: ls)
compE (IntLit n)  =  Push n
compE (VarL x)    =  FetchL x
compE (VarH x)    =  FetchH x
compE (Add e1 e2) =  CSeq (CSeq (compE e1) (compE e2)) Addop

compS :: Stm pc  →  Code ls pc ls
compS (AssL x e)   =  CSeq (compE e) (StoreL x)
compS (AssH x e)   =  CSeq (compE e) (StoreH x)
compS Skip         =  Noop
compS (Seq s1 s2)  =  CSeq (compS s1) (compS s2)
compS (If e s1 s2) =  CSeq (compE e) (Branch (compC s1) (compC s2))
comps (While e s)  =  Loop (compE e) (compS s)
```

We should not forget that we have represented as GADTs not only the abstract syntax of the languages but actually their secure type systems. Therefore, these translation functions turn out to be more than compilation functions. They are actually the Haskell representation of the proof terms of Theorem 6! In fact, observe that the type of these functions is exactly the Haskell encoding of the properties *i)* and *ii)* in the Theorem. In other words, when writing these functions we are actually proving this Theorem and the verification that the Theorem is valid is performed by Haskell type system. As we mentioned above, both *i)* and *ii)* are proved by structural induction. The different cases of the translation functions are actually the encoding in Haskell of the cases of those inductive proofs.

This is a common situation in languages with dependent types, like Agda [15], Coq [16], or Idris [17], but it was not so in languages like Haskell. However, with the increasing incorporation of new features to Haskell, and in particular, to its reference compiler (GHC), this sort of type level programming applications are becoming more frequent and feasible (see e.g. [18]).

5. Related Work

There has been a lot of work on information flow analysis, pioneered by Bell and LaPadula [1], and continued with the work of Denning [19]. Noninterference was introduced by Goguen and Meseguer [2]. One of the approaches to ensure this security property has been based on the use of type system [19, 3]. This is

the approach we followed in this paper. Most of the works on type systems for noninterference concentrated on high-level languages (e.g. [4, 3, 5]), but there are also some works that studied security type systems for low-level languages (e.g. [6, 7, 8]).

In this paper we used a limited form of dependently typed programming available in Haskell and in particular in the GHC compiler by the use of some extensions. The frontiers of this discipline of programming in Haskell is nowadays a subject of discussion and experimentation (see e.g. [20, 18]).

Guillemette and Monnier [21] wrote a type-preserving compiler for System F in Haskell. Their compiler is composed by phases so that Haskell's type checker can mechanically verify the typing preservation of each phase.

An example of the use of GADTs to ensure static properties of programs is presented by Sheard [11]. He encoded in Omega a simple While language that satisfies two semantic properties: scoping and type safety.

These works are similar to ours in the sense that they prove that a compiler preserves the types of object programs, or that a language satisfies some static property. However, none of them are concerned with proving the preservation of a security property of programs.

There are some works on the use of dependent types for developing type-preserving compilers. Chlipala [22], for example, developed a certified compiler from the simply-typed lambda calculus to an assembly language using the proof assistant Coq. He uses dependent types in the representation of the target language of his compiler to ensure, like we did for our object languages, that only terms satisfying the object language typing are representable.

There are many works on certified compilers. One is the work by Leroy [23] who developed a certified compiler (CompCert) for a subset of C in the proof assistant Coq. Another is the work by Barthe, Naumann and Rezk [24] who wrote a compiler for Java that preserves information flow typing, such that any typable program is compiled into a program that will be accepted by a bytecode verifier which enforces noninterference.

6. Conclusion

We presented a compiler written in Haskell that preserves the security property of noninterference. The compiler takes source code from an imperative high-level language and returns code of a low-level language that runs in a stack-based abstract machine. For each of the languages we defined the property of noninterference by means of a security type system. Those type systems were represented in Haskell by means of GADTs combined with type families for computing with security types at the type level and a multi-parameter type class for comparing security types. This encoding guarantees that we can only write terms (of the corresponding GADTs) that are the representation of secure programs.

Using this approach the type of the compiler corresponds exactly to the formulation of the property that noninterference is preserved by compilation.

The definition of the compiler function itself then corresponds to the proof term that proves that property. The rest of the work (i.e. the verification that the function is indeed a proof of that property) is done by Haskell's type system.

We are completely aware we are working in an unsound logic and therefore we must be very careful when we encode proofs in such a logic. The unsoundness of the logic comes from the fact that in Haskell all types are inhabited (they have at least the undefined value). In our development we maintained ourselves in a terminating subsubset of Haskell.

Although the target language of the compiler is simple and semi-structured, we think that a compiler to a more realistic low-level language (for example, with goto) can be constructed applying the same ideas. We are currently doing some experiments in this line using Agda [15], but we do not discard to try with Haskell as well. The decision of using Agda for developing a more realistic compiler is because the complexity of the required program properties increase and then it becomes difficult to express them in Haskell.

References

[1] D. E. Bell, L. J. LaPadula, Secure Computer Systems: Unified Exposition and Multics Interpretation, Tech. Rep. MTR-2997, The MITRE Corp., 1975.

[2] J. A. Goguen, J. Meseguer, Security Policies and Security Models, in: Symposium on Security and Privacy, IEEE Computer Society Press, 11–20, 1982.

[3] A. Sabelfeld, A. C. Myers, Language-Based Information-Flow Security, IEEE J. Selected Areas in Communications 21 (1) (2003) 5–19.

[4] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, J. Comput. Secur. 4 (2-3) (1996) 167–187.

[5] A. Banerjee, D. A. Naumann, Stack-based access control and secure information flow, J. Funct. Program. 15 (2) (2005) 131–177.

[6] G. Barthe, T. Rezk, A. Basu, Security types preserving compilation, Comput. Lang. Syst. Struct. 33 (2) (2007) 35–59.

[7] R. Medel, A. B. Compagnoni, E. Bonelli, A Typed Assembly Language for Non-interference., in: M. Coppo, E. Lodi, G. M. Pinna (Eds.), ICTCS, vol. 3701 of *Lecture Notes in Computer Science*, Springer, 360–374, 2005.

[8] A. Saabas, T. Uustalu, Compositional Type Systems for Stack-based Low-level Languages, in: Proceedings of the 12th Computing: The Australasian Theroy Symposium - Volume 51, CATS '06, Australian Computer Society, Inc., 27–39, 2006.

[9] H. R. Nielson, F. Nielson, Semantics with Applications: A Formal Introduction, John Wiley & Sons, Inc., New York, NY, USA, 1992.

[10] D. M. Volpano, G. Smith, A Type-Based Approach to Program Security, in: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, TAPSOFT '97, Springer-Verlag, London, UK, 607–621, 1997.

[11] T. Sheard, Languages of the Future, SIGPLAN Not. 39 (12) (2004) 119–132.

[12] E. Pasalic, N. Linger, Meta-programming with Typed Object-Language Representations, in: G. Karsai, E. Visser (Eds.), Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings, vol. 3286 of *Lecture Notes in Computer Science*, Springer, 136–167, 2004.

[13] S. Peyton Jones, D. Vytiniotis, S. Weirich, G. Washburn, Simple Unification-based Type Inference for GADTs, in: 11th International Conference on Functional Programming, ACM, 50–61, 2006.

[14] T. Schrijvers, S. Peyton Jones, M. Chakravarty, M. Sulzmann, Type Checking with Open Type Functions, SIGPLAN Not. 43 (9) (2008) 51–62, ISSN 0362-1340.

[15] U. Norell, Dependently Typed Programming in Agda, in: Advanced Functional Programming, vol. 5832 of *Lecture Notes in Computer Science*, Springer, 230–266, 2008.

[16] Y. Bertot, P. Casteran, G. Huet, C. Paulin-Mohring, Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions, Texts in Theoretical Computer Science, Springer, Berlin, New York, 2004.

[17] E. Brady, Idris, a general-purpose dependently typed programming language: Design and implementation, J. Funct. Program. 23 (5) (2013) 552–593.

[18] S. Lindley, C. McBride, Hasochism: the pleasure and pain of dependently typed haskell programming, in: C. Shan (Ed.), Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013, ACM, 81–92, 2013.

[19] D. E. Denning, A Lattice Model of Secure Information Flow, Commun. ACM 19 (5) (1976) 236–243.

[20] R. A. Eisenberg, S. Weirich, Dependently Typed Programming with Singletons, in: Proceedings of the 2012 Haskell Symposium, Haskell '12, ACM, New York, NY, USA, 117–130, 2012.

[21] L.-J. Guillemette, S. Monnier, A type-preserving compiler in Haskell, SIGPLAN Not. 43 (9) (2008) 75–86.

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

[23] X. Leroy, A Formally Verified Compiler Back-end, J. Autom. Reason. 43 (4) (2009) 363–446.

[24] G. Barthe, T. Rezk, D. A. Naumann, Deriving an Information Flow Checker and Certifying Compiler for Java, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, 230–242, 2006.

Appendix A. Proof of Theorem 1

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 base cases corresponding to assignment of low and high variables and skip are immediate.

- Case $S_1;S_2$

  If $S_1;S_2$ is typable in the syntax-directed type system, then by rule SEQ^{sd} we have that:

  - $[pc_1] \vdash_{sd} S_1$
  - $[pc_2] \vdash_{sd} S_2$
  - $[\texttt{min}\ pc_1\ pc_2] \vdash_{sd} S_1;S_2$

  The proof then continues by case analysis on pc_1 and pc_2.

  - When $pc_1 = pc_2$, we use rule SEQ and the induction hyphotesis (HYP) to conclude.

$$\dfrac{\dfrac{[pc_1] \vdash_{sd} S_1}{[pc_1] \vdash S_1}\ \text{HYP} \qquad \dfrac{[pc_1] \vdash_{sd} S_2}{[pc_1] \vdash S_1}\ \text{HYP}}{[pc_1] \vdash S_1;S_2}\ \text{SEQ}$$

  - When $pc_1 = low$ and $pc_2 = high$, we have the following derivation

$$\dfrac{\dfrac{[low] \vdash_{sd} S_1}{[low] \vdash S_1}\ \text{HYP} \qquad \dfrac{\dfrac{[high] \vdash_{sd} S_2}{[high] \vdash S_2}\ \text{HYP}}{[low] \vdash S_2}\ \text{SUB}}{[low] \vdash S_1;S_2}\ \text{SEQ}$$

  - When $pc_1 = high$ and $pc_2 = low$ the proof is similar.

- Case `while` e `do` S

  If `while` e `do` S is typable, then the rule WHILE^{sd} is used and we have that:

  - $\vdash_{sd} e : st$
  - $[pc] \vdash_{sd} S$
  - $st \leq pc$
  - $[pc] \vdash_{sd}$ `while` e `do` S

  The proof then continues by case analysis on st and pc.

  - When $st = pc$, we have the following derivation:

$$\frac{\vdash e : pc \qquad \dfrac{[pc] \vdash_{sd} S}{[pc] \vdash S}\ \text{HYP}}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S}\ \text{WHILE}$$

    where $\vdash e : pc$ is obtained by part (i) of this theorem from $\vdash_{sd} e : pc$.
  - Since $st \leq pc$, the other case corresponds to $st = low$ and $pc = high$. Then we have the following derivation:

$$\frac{\dfrac{}{\vdash e : high}\ \text{EXPH} \qquad \dfrac{[high] \vdash_{sd} S}{[high] \vdash S}\ \text{HYP}}{[high] \vdash \texttt{while}\ e\ \texttt{do}\ S}\ \text{WHILE}$$

- case `if` e `then` S_1 `else` S_2

  When `if` e `then` S_1 `else` S_2 is typable, the rule IF^{sd} is used and we have that:

  - $\vdash_{sd} e : st$
  - $[pc_1] \vdash_{sd} S_1$
  - $[pc_2] \vdash_{sd} S_2$
  - $st \leq \texttt{min}\ pc_1\ pc_2$
  - $[\texttt{min}\ pc_1\ pc_2] \vdash_{sd}$ `if` e `then` S_1 `else` S_2

  The proof then continues by case analysis on st, pc_1 and pc_2.

  - The case where $st = pc_1 = pc_2 = low$ follows by application of induction hypothesis:

$$\frac{\vdash e : low \qquad \dfrac{[low] \vdash_{sd} S_1}{[low] \vdash S_1}\ \text{HYP} \qquad \dfrac{[low] \vdash_{sd} S_2}{[low] \vdash S_2}\ \text{HYP}}{[low] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\ \text{IF}$$

    where $\vdash e : low$ is obtained by part (i) of this theorem from $\vdash_{sd} e : low$.

– When $st = pc_1 = low$ and $pc_2 = high$ we have the following derivation:

$$\dfrac{\vdash e : low \qquad \dfrac{[low] \vdash_{sd} S_1}{[low] \vdash S_1}\,\text{HYP} \qquad \dfrac{\dfrac{[high] \vdash_{sd} S_2}{[high] \vdash S_2}\,\text{HYP}}{[low] \vdash S_2}\,\text{SUB}}{[low] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\,\text{IF}$$

where $\vdash e : low$ is obtained by part (i) of this theorem from $\vdash_{sd} e : low$.

– The case where $st = pc_2 = low$ and $pc_1 = high$ is similar.

– When $st = low$ and $pc_1 = pc_2 = high$, we have that:

$$\dfrac{\dfrac{}{\vdash e : high}\,\text{EXPH} \qquad \dfrac{[high] \vdash_{sd} S_1}{[high] \vdash S_1}\,\text{HYP} \qquad \dfrac{[high] \vdash_{sd} S_2}{[high] \vdash S_2}\,\text{HYP}}{[high] \vdash \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2}\,\text{IF}$$

– The case where $st = pc_1 = pc_2 = high$ is similar to the previous one.

– The case $st = high$ and $pc_1 = pc_2 = low$ is impossible because it contradicts the condition $st \leq \texttt{min}\ pc_1\ pc_2$.

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 $[pc] \vdash S$.

- When the last rule used in the derivation is assL, assH or skip the proof is trivial.
- When the last rule used in the derivation is seq:

$$\dfrac{[pc] \vdash S_1 \quad [pc] \vdash S_2}{[pc] \vdash S_1;S_2}\,\text{SEQ}$$

we can apply induction hypothesis to the derivations of $[pc] \vdash S_1$ and $[pc] \vdash S_2$, obtaining that $[pc_1] \vdash_{sd} S_1$ and $[pc_2] \vdash_{sd} S_2$ for some pc_1 and pc_2 which satisfy the inequalities $pc \leq pc_1$ and $pc \leq pc_2$. Finally, we use rule SEQ^{sd} to conclude that:

$$[\texttt{min}\ pc_1\ pc_2] \vdash_{sd} S_1;S_2$$

where $pc \leq \texttt{min}\ pc'\ pc''$.

- When the last rule used in the derivation is while:

$$\dfrac{\vdash e : pc \quad [pc] \vdash S}{[pc] \vdash \texttt{while}\ e\ \texttt{do}\ S}\,\text{WHILE}$$

we can apply induction hypothesis to the derivation of $[pc] \vdash S$ obtaining $[pc'] \vdash_{sd} S$ for some pc' such $pc \leq pc'$. On the other hand, by part (ii) of the theorem we have that $\vdash_{sd} st : e$ for some st such $st \leq pc$. Therefore, since $st \leq pc'$, we can use the rule WHILE^{sd} to conclude that:

$$[pc'] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S$$

with $pc \leq pc'$.

- 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:

$$\frac{[high] \vdash S}{[low] \vdash S}\ \text{SUB}$$

then we proceed by induction on the structure of S:

  - The cases for assignment of low and high variables and ski are immediate.
  - Case $S_1;S_2$.
    By applying rule SEQ we obtain that $[high] \vdash S_1$ and $[high] \vdash S_2$, and by induction hypothesis we have that $[high] \vdash_{sd} S_1$ and $[high] \vdash_{sd} S_2$. Finally, we use rule SEQ^{sd} to conclude that $[high] \vdash_{sd} S_1;S_2$.
  - Case $\texttt{while}\ e\ \texttt{do}\ S$.
    By applying rule WHILE we obtain that $\vdash e : high$ and $[high] \vdash S$, and by induction hypothesis we have that $[high] \vdash_{sd} S$. Now, by part (ii) of the theorem we have that $\vdash_{sd} st' : e$ for some $st' \leq high$. Finally, we use rule WHILE^{sd} to conclude that $[high] \vdash_{sd} \texttt{while}\ e\ \texttt{do}\ S$.
  - The case $\texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2$ is analogous.

Appendix B. Proof of Theorem 6

Property (i). If $\vdash_{sd} e : st$ then $ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow st :: ls$.

PROOF. The proof is by induction on the structure of expressions.

- Case n

  We have that $\boldsymbol{C}_e[n] = \mathsf{push}\ n$ and $\vdash_{sd} n : low$. Then, we use rule PUSH to conclude that $ls \vdash \mathsf{push}\ n : high \rightsquigarrow low :: ls$.

- Case $e_1 + e_2$

  Recall that $\boldsymbol{C}_e[e_1 + e_2] = \boldsymbol{C}_e[e_1]\ ; \boldsymbol{C}_e[e_2]\ ; \mathsf{add}$. Since $e_1 + e_2$ is typable, there exists st_1 and st_2 such that:

- $\vdash_{sd} e_1 : st_1$
- $\vdash_{sd} e_2 : st_2$
- $\vdash_{sd} e_1 + e_2 : \mathtt{max}\ st_1\ st_2$

By induction hypothesis we have that $ls \vdash \boldsymbol{C}_e[e_1] : high \rightsquigarrow st :: ls$ and $st :: ls \vdash \boldsymbol{C}_e[e_2] : high \rightsquigarrow st' :: st :: ls$. Then, by using the rules CSEQ and ADD we obtain $ls \vdash \boldsymbol{C}_e[e_1]\ ; \boldsymbol{C}_e[e_2]\ ; \mathsf{add} : high \rightsquigarrow \mathtt{max}\ st\ st' :: ls$, which rewrites to $ls \vdash \boldsymbol{C}_e[e_1 + e_2] : high \rightsquigarrow \mathtt{max}\ st\ st' :: ls$ by definition of $\boldsymbol{C}_e$.

- The other cases are analogous.

Property (ii). If $[pc] \vdash_{sd} S$ then $ls \vdash \boldsymbol{C}_S[S] : pc \rightsquigarrow ls$

PROOF. The proof is by induction on the structure of statements.

- Case $x_L := e$.

  Since $x_L := e$ is typable we have that:

  - $\vdash_{sd} e : low$
  - $[low] \vdash_{sd} x_L := e$

  By using part (i) of the theorem we obtain that: $ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow low :: ls$. Then, we apply the following derivation:

$$\dfrac{ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow low :: ls \qquad \dfrac{}{low :: ls \vdash \mathsf{store}\ x_L : low \rightsquigarrow ls}\ \textsc{storeL}}{ls \vdash \boldsymbol{C}_e[e]\,;\mathsf{store}\ x_L : low \rightsquigarrow ls}\ \textsc{cseq}$$

  Finally, by definition of $\boldsymbol{C}_S$ the desired result follows.

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

  Since $\mathtt{while}\ e\ \mathtt{do}\ S$ is typable, there exist st and pc such that:

  - $\vdash_{sd} e : st$
  - $st \leq pc$
  - $[pc] \vdash_{sd} S$
  - $[pc] \vdash_{sd} \mathtt{while}\ e\ \mathtt{do}\ S$

  Using part (i) of the theorem we have that $ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow st :: ls$ and by induction hypothesis we obtain that $ls \vdash \boldsymbol{C}_S[S] : pc \rightsquigarrow ls$.

  Then, we can apply the following derivation:

$$\dfrac{ls \vdash \boldsymbol{C}_e[e] : high \rightsquigarrow st :: ls \qquad ls \vdash \boldsymbol{C}_S[S'] : pc \rightsquigarrow ls \qquad st \leq pc}{ls \vdash \mathsf{loop}\ (\boldsymbol{C}_e[e], \boldsymbol{C}_S[S']) : pc \rightsquigarrow ls}\ \textsc{loop}$$

  Finally, by definition of $\boldsymbol{C}_S$ the desired result follows.

- The other cases are analogous.