A Haskell-embedded DSL for Secure Information-flow

Cecilia Manzino and Gonzalo de Latorre

Departamento de Ciencias de la Computación, Universidad Nacional de Rosario, Argentina
ceciliam@fceia.unr.edu.ar, gdelatorre_@hotmail.com

Abstract. This paper presents a domain specific language, embedded in Haskell (EDSL), for enforcing the information flow policy *Delimited Release*. To build this language we use Haskell extensions that will allow some kind of dependently-typed programming.
Considering the effort it takes to build a language from scratch, we decided to provide an information-flow security language as an EDSL, using the infrastructure of the host language to support it.
The decision of using Haskell as the implementation language is because it has a powerful type system that makes it possible to encode the security type systems of the embedded language at the type level and also because it is a general purpose language.
The implementation follows an approach where the type of the abstract syntax of the embedded language was decorated with security type information. This way, typed programs will correspond to secure programs, and the verification of the security invariants of programs will be reduced to type-checking.
The embedded security language is designed in a way that is easy to use. We illustrate its use through two examples: an electronic purchase and a secure reading of database information.

Keywords: dependently-typed programming · Haskell · information flow type systems · declassification

1 Introduction

Ensuring the confidentiality of information manipulated by computing systems has become of significant importance in recent years [2]. Traditional security mechanisms such as access control [1] or cryptography do not provide end-to-end protection of data. To complement these security mechanisms, techniques that examine information flows between inputs and outputs of systems have become a subject of study. In this context security polices arise for guaranteeing that confidential information cannot be released to public data. Non-interference [3] is an example of an information flow policy. A program satisfies this property when the final value of any public variable is not influenced by a variation of confidential inputs during its execution.

A remarkable feature of this property is that it can be enforced statically by the definition of an information flow type system ([4], [7]).

However, some realistic applications must allow intentional information release as part of its intended behavior and non-interference is very restrictive for this. To make security languages practical, mechanisms for declassifying or releasing information in a controlled manner have been studied ([11], [8], [6]).

The critical part of these mechanisms is to ensure that declassification is used safely.

An information flow policy, named *Delimited Release* was introduced by Sabelfeld and Myers [6] to guarantee that the intentional release of information, marked in programs under expressions *declassify*, cannot be used to infer more information than it should. They also define a type system that enforces this security property on typed programs.

The results about information flow security are mostly theoretical. Two languages based on information flow were implemented: Jif [5] (a variant of Java) and Flow Caml (based on Ocaml).

Other works investigate the way of expressing restrictions on information-flow providing control-flow primitives as a library, avoiding the work of producing a new language from scratch. In this line, Li and Zdancewic [12] implemented an embedded security sublanguage which provides a security property, in Haskell. Informally, the security property can be stated as: "code running at privilege l_p cannot observe information of label l if ($l \sqsubseteq l_p$)". Their implementation is based on arrows. Whereas Russo, Claessen and Hughes [10] provide information-flow security also as a library in Haskell but their implementation is based on monads. The intention of the authors is to show that the same goals can be achieved with monads instead of arrows.

In this work, we propose an EDSL in Haskell for providing information-flow security. To achieve our goal, we follow a different approach than the one used by the forementioned authors. Our implementation is based on the use of generalized algebraic data types (GADTs), for representing the terms of the embedded language together with the typing rules that guarantee the security property. Then, programs constructed with this data type are secure by-construction. Since the constructors of the GADT are a direct implementation of the typing rules, the type system must be syntax-directed. We give a syntax-directed formulation of the type system that guarantee Delimited Release to make this implementation possible.

The rest of the paper is organized as follows. Section 2 presents the syntax, semantic and security type system of the embedded language. Section 3 presents the implementation of the EDSL, while section 4 gives some example programs that illustrate its use. Section 5 presents some related works. Section 6 concludes.

The complete Haskell code of the EDSL is available in a GitHub repository[1].

[1] https://github.com/ceciliamanzino/EDSL-DR-

2 A Security Language

In this section we will describe the syntax, semantics and type system of the language implemented as an EDSL. The language is a standard while language extended with the construct *declassify* which declassifies the security level of an expression. The type system enforces a security property named *delimited release*. This security property together with type system that enforces it were defined in [6], here we present a syntax directed version of the type system.

2.1 Syntax

The expressions and statements of the language are defined by the following abstract syntax:

$$e ::= \texttt{val} \mid x \mid e_1 \ \texttt{op} \ e_2 \mid \texttt{declassify}(e, l)$$
$$s ::= x := e \mid \texttt{skip} \mid s_1; s_2 \mid \texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \mid \texttt{while} \ e \ \texttt{do} \ S$$

where $e \in \mathbf{Exp}$ (expressions), $s \in \mathbf{Stm}$ (statements), $x \in \mathbf{Var}$ (variables), $\texttt{val} \in \mathbf{Num} \cup \{true, false\}$ (integer and boolean literals), $l \in \mathcal{L}$ (security lattice) and op ranges over arithmetic and boolean operations on expressions.

The semantics of this language is completely standard. The unique expression that is not standard is $\texttt{declassify}(e, l)$, which is used for declassifying the security level of the expression e to l. At the semantic level $\texttt{declassify}(e, l)$ is equivalent to e.

The meaning of expressions and statements is given relative to a memory $s \in M = \mathbf{Var} \to \mathbf{Num} \cup \{true, false\}$, which contains the current value of each variable. We assume that the semantics for expressions is given by an evaluation function denoted by $\langle M, e\rangle \Downarrow val$. For statements, we define a big-step semantics whose transition relation is written as $\langle M, s\rangle \Downarrow M'$, meaning that the evaluation of a statement s in an initial memory M terminates with a final memory M'.

2.2 Non-interference

We assume that each variable has associated a security level, which states the degree of confidentiality of the value it stores. A type environment $\Gamma : \mathbf{Var} \to \mathcal{L}$ maps each variable to a security type, where $(\mathcal{L}, \sqsubseteq)$, is a bounded lattice of security levels with meet ($\sqcap$) and join ($\sqcup$) operations, and top ($\top$) and bottom ($\bot$) values. The bottom value represents the least security level (public data) whereas the top value represents the highest security level (private data).

Non-interference [3] is a security property that guarantees the absence of illicit information flows during execution of a program. A program satisfies this security property when final values of low level security variables are not influenced by a variation of the initial values of the high level security variables. This property can be formulated in terms of program semantics.

Two memories M_1 and M_2 are l-equivalent, written $M_1 \approx_l M_2$, when they coincide in the variables with lower security level than l. While, two program

configurations $\langle M_1, s_1\rangle$ and $\langle M_2, s_2\rangle$ are indistinguishable at level l, written $\langle M_1, s_1\rangle \approx_l \langle M_2, s_2\rangle$, if whenever $\langle M_1, s_1\rangle \Downarrow M_1'$ and $\langle M_2, s_2\rangle \Downarrow M_2'$ for some M_1' and M_2' we have that $M_1' \approx_l M_2'$. Two expression configurations $\langle M_1, e_1\rangle$ and $\langle M_2, e_2\rangle$ are indistinguishable, written $\langle M_1, e_1\rangle \approx \langle M_2, e_2\rangle$, if $\langle M_1, e_1\rangle \Downarrow val$ and $\langle M_2, e_2\rangle \Downarrow val$ for some val.

Now, the security property can be formalized as follows:

Definition 1. *(Non-interference) A statement s satisfies non-interference if, for all level l, we have:*

$$\forall M_1\ M_2\,.\,M_1 \approx_l M_2 \Rightarrow \langle M_1, s\rangle \approx_l \langle M_2, s\rangle$$

A positive feature of non-interference is that it can be enforced statically by the definition of an information-flow type system ([4], [7]). But its use in some computing systems could be too restrictive. For example, if we consider a password checking program, on every attempt to login there is a release of information about the user's password, or if we want to send encrypted secret information to a public place there is also an information leakage.

Even though it is necessary to relax the notion of non-interference and have mechanisms that are able to release information in a controlled manner, questions about how to prevent attacks arise. For example, an attacker could take advantage of these mechanisms for releasing data to extract more information than intended.

Sabelfeld and Myers [6] introduce a new security property, named *Delimited Release* that guarantees that the realease of information (marked in programs via expressions `declassify`) cannot be used to construct attacks. They also presented a security type system that enforces this property.

2.3 Delimited Release

The intention behind this property, is to express that only explicitly declassified information is released and no other information. More specifically, the property establishes that a program is secure if updates to variables that are latter declassified, occur in a manner that an attacker cannot use them to infer more information than the data that has already been released by declassification.

The security property is formalized as follows:

Definition 2. *(Delimited Release) Suppose the statement s contains within it exactly n declassify expressions* $\texttt{declassify}(e_1, l_1) \ldots, \texttt{declassify}(e_n, l_n)$. *Statement s is secure if for all security levels l we have:*

$$\forall M_1, M_2.(M_1 \approx_l M_2 \wedge \forall i \in \{i \,|\, l_i \sqsubseteq l\}.\langle M_1, e_i\rangle \approx \langle M_2, e_i\rangle) \Rightarrow \langle M_1, s\rangle \approx_l \langle M_2, s\rangle$$

Expressions

$$\Gamma \vdash val \ : \ l, \varnothing \qquad \text{(VAL)} \qquad \frac{\Gamma \vdash e_1 \ : \ l_1, D_1 \qquad \Gamma \vdash e_2 \ : \ l_2, D_2}{\Gamma \vdash e_1 \ \texttt{op} \ e_2 \ : \ l_1 \sqcup l_2, D_1 \cup D_2} \text{(OP)}$$

$$\Gamma \vdash v \ : \ \Gamma(v), \varnothing \qquad \text{(VAR)} \qquad \frac{\Gamma \vdash e \ : \ l, D}{\Gamma \vdash \texttt{declassify}(e, l_1) \ : \ l_1, Vars(e)} \text{(DEC)}$$

Statements

$$\Gamma, pc \vdash \texttt{skip} \ : \ \varnothing, \varnothing \qquad \text{(SKIP)} \qquad \frac{\Gamma \vdash e \ : \ l, D \quad l \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e \ : \ \{x\}, D} \text{(ASS)}$$

$$\frac{\Gamma, pc_1 \vdash s_1 \ : \ U_1, D_1 \qquad \Gamma, pc_2 \vdash s_2 \ : \ U_2, D_2 \qquad U_1 \cap D_2 = \varnothing}{\Gamma, pc_1 \sqcap pc_2 \vdash s_1; s_2 \ : \ U_1 \cup U_2, D_1 \cup D_2} \text{(SEQ)}$$

$$\frac{\Gamma \vdash e \ : \ l, D \qquad \Gamma, pc_1 \vdash s_1 \ : \ U_1, D_1 \qquad \Gamma, pc_2 \vdash s_2 \ : \ U_2, D_2 \qquad l \sqsubseteq pc_1 \sqcap pc_2}{\Gamma, pc_1 \sqcap pc_2 \vdash \texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ : \ U_1 \cup U_2, D \cup D_1 \cup D_2} \text{(IF)}$$

$$\frac{\Gamma \vdash e \ : \ l, D \qquad \Gamma, pc \vdash s \ : \ U, D_1 \qquad l \sqsubseteq pc \qquad U \cap (D \cup D_1) = \varnothing}{\Gamma, pc \vdash \texttt{while} \ e \ \texttt{do} \ s \ : \ U, D \cup D_1} \text{(WHILE)}$$

Fig. 1. Syntax-directed Security Type System For Delimited Release

2.4 Security Type System for Delimited Release

This section presents a type system that enforces Delimited Release. This version of the type system is based on the one given in [6] with the characteristic of being syntax-directed. The typing rules are shown in Figure 1.

In this system, the typing judgement for expressions has the form $\Gamma \vdash e \ : \ l, D$, meaning that the expression e is typed in the security environment Γ, has security level l and effect D. The type system collects the variables that are declassified in e, in set D. Typing judgement for statements has the form $\Gamma, pc \vdash s \ : \ U, D$ which means that statement s is typable in a program counter pc, under security environment Γ and effects U and D. The program counter is the lower level of the variables assigned in s. The type system collects the variables that are declassified in s in the effect D and the updated variables in U.

The goal of the typing rules is to prevent improper information flows during program execution and to control the information that is declassified.

If a program doesn't contain the expression `declassify` and is typed in the security type system for Delimited Release, then it satisfies non-interference [3].

2.5 Safe Programs Examples

This section will show two short examples, that are common in the literature and that have already been introduced in[6]. These simple programs will help us understand the security property. In the next section we will implement these examples in our EDSL in Haskell.

Example 1. (Average salary)

Consider a simple program to calculate the average salary of n employees.

Suppose that variables $h_1, \ldots, h_n$ store the salaries of the employees. We can store the result of the average of these private variables in a public variable named avg by declassifying the information of this result but not other information (like the salary of one employee):

$$avg := \texttt{declassify}((h_1 + \cdots + h_n)/n, low)$$

If we consider a lattice of two levels $\mathcal{L} = \{low, high\}$, where $\bot = low$ and $\top = high$ and an environment Γ, such $\Gamma(h_i) = high$ for $i \in \{1, \ldots, n\}$, and $\Gamma(avg) = low$ we can easily check that the program is typable in the type system for Delimited Release.

Now, we consider a malicious use of this program for leaking information of variable h_1 to avg:

$$\begin{array}{l} h_2 := h_1; \\ \ldots \\ h_n := h_1; \\ avg := \texttt{declassify}((h_1 + \cdots + h_n)/n, low) \end{array}$$

This program is rejected by the type system, since the variables $h_2, \ldots, h_n$ which are under declassification in the last line were updated before.

Example 2. (Electronic wallet)

In this example we have a scenario where if a customer has enough money in their electronic wallet then a purchase is carried out.

Suppose that a private variable h stores the amount of money in a customer's wallet, a public variable l stores the amount of money spent in the session and a public variable k stores the cost of the purchase. The following code checks if $h \geq k$, for modifying the values of the variables l and h:

$$\texttt{if declassify}(h \geq k, low) \texttt{ then } (h := h - k; l := l + k) \texttt{ else skip}$$

This program doesn't satisfy the non-interference property, since an assignment of the public variable l occurs under a high condition, but it is typable with type system for Delimited Release.

Now, we show a program that uses this code for leaking bit-by-bit the value of the secret variable k to l:

$l := 0;$
`while` $(n \geq 0)$
$\quad k := 2^{(n-1)};$
$\quad$ `if declassify`$(h \geq k, low)$
$\quad$ `then` $(h := h - k; l := l + k)$
$\quad$ `else skip`;
$\quad n := n - 1$

This program is rejected by the type system for Delimited Release.

3 Implementation

In this section we will present the implementation of our EDSL.

In the following we will accompany the implementation of the embedded language together with concepts introduced by some Haskell extensions, like *promoted data types* [13], *singleton types* [14], *type families* [9] and GADTs. These extensions will enable us to program in Haskell in a similar way as we will do in a dependently typed language.

3.1 Security Types and Variables

For simplicity we consider just two security levels, but the implementation can be easily generalized to a lattice of security levels ordered by their degree of confidentiality. In such generalization we can use Haskell's type class mechanism for defining security lattices.

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

```
data SType = Low | High
```

This defines datatype `SType` with two values (or constructors): `Low` and `High`. In a dependently typed language, we could use the values `Low` and `High` in a GADT definition, but in Haskell, we must lift them to the type level. With the extension *promoted datatypes*, we can use them one level up. So we can use the type `SType` as a kind and the values `High` and `Low` as types. To distinguish the types from the values we must use a quote.

Then, the types `'Low` and `'High` have kind `SType` and there are only two types of kind `SType`.

However, later we will need a copy of these types as value level, so we will use a *singleton type* definition of `SType`.

```
data SeType (s :: SType) where
    L :: SeType 'Low
    H :: SeType 'High
```

The idea of *singleton types* is that we can use pattern matching on values to know at run time the type of the value, since each type in the definition has exactly one value of that type.

For example, the values `Low` and `High` have the same type `SType`, but `L` and `H` (which represent the same values as `Low` and `High`) have different types `SeType 'Low` and `SeType 'High`, respectively.

In this implementation, we will use both definitions to represent security levels. In a dependently typed language, such as Agda, we need just one definition.

The order of the security types is given by the following non-method class with instances:

```
class LEq (a :: SType) (b :: SType)
instance LEq 'Low x
instance LEq 'High 'High
```

Haskell's type system will choose the appropriate instance for a pair of types `st` and `st'` that satisfies `st` $\leq$ `st'`. If we see each instance of this class as a pair of a relation, we can observe that these pairs form an order relation. This implementation can be generalized to a lattice of security levels by defining a class with methods for meet and join operations, the order relation ($\leq$), and the bottom and top values.

In the same way we define natural numbers as types:

```
data Nat = Zero | Succ Nat
```

```
data SNat (n :: Nat) where
    SZero :: SNat 'Zero
    SSucc :: SNat n → SNat ('Succ n)
```

For example,

```
two :: SNat ('Succ ('Succ 'Zero))
two = SSucc (SSucc SZero)
```

Similarly we can define any natural number, and all of them will have different types.

We will use natural numbers to represent variables of the language and lists of pairs of security types together with naturals to represent type environments.

A *type family* is essentially a function on types or it can also be seen as parametric types. They provide a functional style for programming at type-level in Haskell.

For looking up the security type associated to a variable in a given environment we will use the *type family* `Lookup`:

```
type family Lookup (env :: [(k,st)]) (n :: k) :: a where
    Lookup ('(n, st) ': env) n = st
    Lookup ('(m, st) ': env) n = Lookup env n
```

The first line introduce the signature of the type family. The word `family` distinguishes this definition from a standard type definition.

In this definition we use the constructor for pairs promoted to type `'(,)` and the constructor for list `':` which is also promoted.

3.2 The Language

Generalized algebraic data types (GADTs) are a generalization of ordinary algebraic data types. This generalization is over the return type of the constructors for the data type, that must be an application of the data type being defined, as in any data type definition, but this application can be on an arbitrary type. This feature make GADTs useful for expressing invariants at the type level.

As example, consider the following GADT definition for terms of a small language for arithmetic and boolean expresions:

```
data Term a where
   T      :: Term Bool
   F      :: Term Bool
   Lit    :: Int  → Term Int
   Add    :: Term Int  → Term Int  → Term Int
   IsZero :: Term Int  → Term Bool
   If     :: Term Bool  → Term a  → Term a  → Term a
```

We use GADTs for representing expresions and terms of the security language.

The encoding is such that, the judgement $env \vdash e : st, d$ in our formal type system corresponds to the typing judgement `e :: Exp env st d d'` in Haskell. The parameter `d'` doesn't have its correspondence in the type system since it was added in the implementation for collecting the occurrences of variables in the expression. Each constructor of the GADT encode a typing rule.

```
data Exp :: [(Nat, SType)] → SType → [Nat] → [Nat] →  *  where
    Var :: SNat (n :: Nat)  →  Exp env (Lookup env n) '[] '[n]
    IntLit :: Int  →  Exp env 'Low '[] '[]
    BoolLit :: Bool  →  Exp env 'Low '[] '[]
    Ope :: Op  →
           Exp env st d var1  →
           Exp env st' d' var2  →
           Exp env (Join st st') (Union d d') (Union var1 var2)
    Declassify :: Exp env l' d vars  →
                  SeType l  →
                  Exp env l vars vars
```

the type `Op` represents integer and boolean operations for expresions:

```
data Op = Plus | Minus | Mult | Div | Exp | Mod | And | Or | Gt |
          GtE | Lt | LtE |  Eq  | Not Eq
```

The datatype `Exp` is parametrized by the type environment (which is of kind `[(Nat , SType)]`), the security type of the expression (of kind `SType`), the list of variables that were used under declassification and the list of variables used in the expression (both of kind `[Nat]`).

In the encoding, the maximum between two security types is computed by a type family `Join` and the append of two list is computed by a type family `Union`. Both definitions are available in the github repository.

To model statements we define the following GADT that is parametrized by the type environment, the program counter (represented as a security type) and two lists that represents the set of variables that were updated and the set of variables that were used under declassification.

Now the typing judgement $env, pc \vdash stm : u, d$ in our formal type system corresponds to the typing judgement `e :: Exp env st d d'` in Haskell. Each constructor of the data type definition corresponds to a rule of the type system shown in Figure 1. This representation is possible because the type system is syntax-directed.

Since the type `Stm env pc u d` encodes the security typing rules, it is only possible to write terms that corresponds to secure programs.

```
data Stm :: [(Nat, SType)] → SType → [Nat] → [Nat] →  *  where
    Skip :: Stm env 'High '[] '[]

    Ass ::  LEq  st (Lookup env n) ⇒
           SNat (n :: Nat)  →
           Exp env st d var  →
           Stm env (Lookup env n) '[n] d

    Seq :: Intersection u1 d2 ~ '[] ⇒
           Stm env pc u1 d1  →
           Stm env pc' u2 d2  →
           Stm env (Meet pc pc')(Union u1 u2)(Union d1 d2)

    If ::  LEq  st (Meet pc pc') ⇒
          Exp env st d vars  →
          Stm env pc u1 d1  →
          Stm env pc' u2 d2  →
          Stm env (Meet pc pc')(Union u1 u2)(Union d(Union d1 d2))

    While :: (Intersection u1 (Union d d1) ~ '[], LEq  st pc) ⇒
              Exp env st d vars  →
              Stm env pc u1 d1  →
              Stm env pc u1 (Union d d1)
```

The minimum between two security types is computed by the type family `Meet`, while the intersection of two sets is computed by the type family `Intersection`.

In the constructors `While` and `Seq` we use type equality constraints of the form `a ~ b` in their type context, which means that types `a` and `b` must be the same. The restriction $U \cap (D \cup D_1) = \varnothing$ in the WHILE rule of the type system corresponds to the constraints `(Intersection u1 (Union d d1) ~ '[]` in the

constructor `While`, and the restriction $U_1 \cap D_2 = \varnothing$ of SEQ rule corresponds to the constraint `Intersection u1 d2 ~ '[]`. These restrictions are necessary to ensure that variables that are used under declassification may not be updated before being declassified.

Some rules of the type system have restrictions of the form $l \leq l'$. In these cases these restrictions correspond to a constraint of the form `LEq l l'` in the implementation, and are used in order to prevent improper information flows.

3.3 Constructors

Programming directly with the constructors of the GADT can be cumbersome, so we define the constructors of our EDSL using the definition of `Stm`.

In order to construct programs with these constructors, we need to provide a form to construct an environment of security variables. Then, we need type-level lists of types of kind `(Nat , SType)`. We represent environments with the following definition:

```
data HList :: [(Nat , SType)] → * where
    Nil :: HList '[]
    (:-:) :: (SNat n , SeType s) →
             HList xs →
             HList ('(n , s) ': xs)
```

We use this data type to have a copy of the environment at value level.

For example, we define an environment with three variables, variable `zero` has security level Low and variables `one` and `two` have security level High.

```
zero = SZero
one = SSucc zero
two = SSucc one

env = (zero, L) :-: (one, H) :-: (two, H) :-: Nil
```

The constructor `var` is used to declare a variable in a given context. The first argument is the type environment that will be used throughout the program, while the second is a natural number associated to the variable. The expression is only typable if the natural number belongs to the environment:

```
var :: HList env →
       SNat (n :: Nat) →
       Exp env (Lookup env n) '[] (n ': '[])
var en n = Var n
```

As example we define a low variable `l` that belongs to environment `env`:

```
l = var env zero
```

The constructors `int` and `bool` are used for literal integer and boolean values:

```
int :: Int  → Exp env 'Low '[] '[]
int = IntLit
```

```
bool :: Bool  → Exp env 'Low '[] '[]
bool = BoolLit
```

For example, the expressions `int 3` and `bool True` represents the values 3 and `True` respectively.

The constructor for writing assignments is the following:

```
(=:) n  exp  = Ass n  exp
```

Now, we can write a program which assigns the value 3 to the public variable `zero`:

```
Program1 = zero =: int 3
```

The other constructors of the language are defined as follows:

```
skip = Skip

iff s e1 e2 = If s e1 e2

while s e1 = While s e1

declassify :: Exp Env l' d vars  →
              SeType l  →
              Exp Env l vars vars
declassify e l = Declassify e l

(>>>) :: (Intersection u1 d2 ~ '[]) ⇒
            Stm Env pc u1 d1  →
            Stm Env pc' u2 d2  →
            Stm Env (Meet pc pc') (Union u1 u2) (Union d1 d2)
(>>>) s1 s2  =  Seq s1 s2
```

To make programs look more concise, the list of operations was simplified:

```
plus     =  Ope Plus
minus    =  Ope Minus
mult     =  Ope Mult
gt       =  Ope Gt
gte      =  Ope GtE
lt       =  Ope Lt
lte      =  Ope LtE
eq       =  Ope  Eq
```

```
neq      =  Ope Not Eq
(/.)     =  Ope Div
(&.)     =  Ope And
(|.)     =  Ope Or
```

4 Implementation of Examples

In this section we illustrate the practicality of the EDSL through the examples presented in section 2.5. A larger example that implements a secure program for password checking can be found in the repository.

Example 3. Average salary

For writing this program we need an environment with four variables, 3 of them must be high since they will be used for storing the salaries of 3 employees, and the other must be low since it will store the average of the salaries.

```
env = (zero, L) :-: (one, H) :-: (two, H) :-: (three, H) :-: Nil
```

Then we can write the example as follows:

```
h1 = var env one
h2 = var env two
h3 = var env three
avgSalaries = zero =: declassify ((h1  +.  h2  +.  h3)
                                  /.  int 3) L
```

Now, if we try to define a program in our EDSL that is considered a laundering attack (since leaks information about the salary of one employee to the variable `zero`) as follows:

```
avgAttack = one =: h2   >>>
            three =: h2 >>>
            avgSalaries
```

we found that the program is rejected by Haskell's type system.

Example 4. Electronic wallet

To write this example we define an environment with three variables, the secret variable will be used to store the amount of money of the customer's electronic wallet, and the public variable will store the amount of money spent by the customer and the cost of the purchase.

```
env = (zero, H)  :-:  (one, L)  :-:  (two, L)  :-:  Nil
```

The secure program is written as follows:

```
h = var env zero
l = var env one
k = var env two
```

```
walletSecure = iff  (declassify (h  >.  k) L)
                    (zero =: h  -.  k  >>>  one =: l  +.  k)
                    skip
```

While, the following attack is rejected by ghc:

```
walletAttack = one =: int 0 >>>
               (while (n  >.  int 0)
                      (two =: int 2 ^. n  -.  int 1) >>>
                      (iff (declassify (h  ≥ . k) L)
                           (zero =: h  -.  k  >>> one =: l  +.  k)
                           skip ) >>>
                      three =: n  -.  int 1 )
```

This program leaks information bit-by-bit of the private variable `h` to `l`. It is rejected since the variable `h` (zero) that occurs under declassification is updated in the body of the loop.

5 Related Work

Security-typed programming languages have been studied in the last years to guarantee that confidential information cannot be released to public data. Jif [5] is a secure language that extends Java with support for information flow control and access control. While Flowcaml is a prototype implementation of an information flow analyzer for OCaml.

Rather than producing a new language, Li and Zdancewic [12] presented a library for information-flow security programming in Haskell based on arrows combinators and type classes. They use arrows for providing an interface that support programming constructs like sequential compositions, conditionals and loops. The library provides some information-flow control mechanisms likes dynamic security lattices and declassification.

Russo, Claessen and Hughes [10] showed that the same goals can be achieved using monads instead of arrows, which is a less general notion and most used by Haskell programmers. The monadic library guarantees that well-typed programs are non-interferent and also allows to specify declassification polices, which are enforced dynamically at run-time. These polices can be expressed using different combinators related to what, when, and by whom information is released.

Even if the arrow notation was eliminated in this library we found that the declassification combinators for generating escape hatches for downgrading information are difficult to use.

In this paper we provide information-flow security as a sublanguage in Haskell. A difference from the works mentioned above is that in this work the sublanguage provided is an imperative language, while in the others, information flow control is applied to programs of the host language. The approach we follow to achieve this is also different, in this work we attach security type information to the datatypes representing the abstract syntax of the sublanguage in such a

way that we only deal with well-typed terms. To make it we use some extensions of Haskell that gives us the possibility to perform some kind of type-level programming.

6 Conclusion and future work

We presented an EDSL in Haskell for writting applications that require to enforce a security property that guarantee confidentiality of the information.

In the implementation we follow an approach for representing the security language using GADT to represent terms, where each constructor of the GADT is a direct implementation of a typing rule. This encoding guarantees that we can only write terms that corresponds to secure programs and the verification of that reduce to type checking.

Although type-level programming in Haskell is a bit tedious, we were able to encode the typing rules in the term's type of the embedded language in a not so difficult way. The user of the EDSL does not have to program at type-level, using the constructors provided, the implementation of the given examples could be written easily. We conclude that the EDSL is suitable for writing applications where the confidentiality of information must be required.

The decision of using Haskell instead of a dependently programming language like Adga or Idris is because Haskell is a general purpose language, and the user of the EDSL will not have to program at type-level for writing applications.

As future work we plan to add a constuctor to the language for adding variables to the environment. We started working in this direction by defining a constructor:

```
newVar :: HList env  →                -- actual environment
          SNat (n :: Nat)  →          -- new variable
          SeType (st :: SType)  →     -- security level
          Stm ('( n , st ) ': env) 'High '[] '[]
newVar en n st = Skip
```

This constructor changes the information about the security environment at type level. To use it we must find a way to have an unfixed environment in some of the constructors of the datatype `Stm`, like `Seq`.

Another future work is to address the formalization of other security policies. The property *delimited release* can capture *what* information is released, other security properties capture *who* releases information, *where* in the system information is released, and *when* information can be released [11]. The security property *robust declassification* [8], is a good candidate since it is orthogonal to *delimited release* (in the sence that control *when* information is declassified) and can be combining with this.

References

1. B. W. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton*, 1971. Reprinted in *ACM Operating Systems Review*,vol 8, no. 1, pp 18-24, Jan. 1974.
2. Dorothy E. and Denning. A lattice model of secure information flow. *ACM*, 19(5):236-243, May 1976.
3. J. A. Goguen and J. Meseguer. Security policies and security models. *In Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
4. Volpano, Dennis M. and Smith, Geoffrey. A Type-Based Approach to Program Security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, 1997.
5. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. *Jif: Java information flow*. Software release (2001). Located at http://www.cs.cornell.edu/jif.
6. A. Sabelfeld and A. C. Myers. Model for Delimited Information Release. *ISSS 2003*: 174-191.
7. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
8. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. *In Proc. IEEE Computer Security Foundations Workshop*, June 2004.
9. M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated Types with Class. *In POPL 2005*.
10. A. Russo, K. Claessen, J. Hughes. A Library for Light-Weight Information-Flow Security in Haskell. *In Haskell*, 2008.
11. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *In Journal of Computer Security*. pp 517–548, October 2009.
12. Li and S. Zdancewic. Arrows for secure information flow. In *TCS 411*, 2010.
13. B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis and J. P. Magalhães. Giving Haskell a Promotion. *In TLDI 2012*.
14. R. A. Eisenberg and S. Weirich. Dependently Typed Programming with Singletons. *In Haskell 2012*.
15. A. Löh. Applying Type-Level and Generic Programming in Haskell, 2015.
16. C. Manzino and A. Pardo. Agda Formalization of a Security-preserving Translation from Flow-sensitive to Flow-insensitive Security Types. In *Electronic Notes in Theoretical Computer Science*. Pages 75-94, 2020.
17. G. De Latorre. EDSL en Haskell para la programación segura respecto a la propiedad Delimited Release. Final year project. National University of Rosario, Argentina, 2022.