Encoding Multithreaded Information Flow in Haskell

Graduate Thesis

to obtain the degree of Master of Science in
International Master's Program
Dependable Computer Systems
Chalmers University of Technology

January 2007

by

Ta-chung Tsai

Göteborg, Sweden.

Supervisor: Dr. John Hughes
Department of Computer Science and Engineering
Chalmers University of Technology
SE - 412 96 Göteborg, Sweden

Contents

List of Figures

List of Tables

Abstract

Li and Zdancewic recently propose a library in Haskell to guarantee information-flow policies for sequential programs. Their approach that develops a library rather than a new language from scratch permits a program written interchangeably in the library, or the sub-language, and ordinary Haskell. Their approach effectively reduces the threshold of practical adoption. However, computations in the library contain no side-effects. In practice, many algorithms could be implemented more efficiently with updateable memory, for instance, references. It is not clear how to guarantee information-flow policies in the library in presence of side-effects, such as references. On the other hands, the library have no support for multithreaded programs. The intreleavings of threads create new timing channels to leak secret information. Many researches have theoretically investigated how to specify and enforce information-flow policies for multithreaded programs. However, no practical tool exists so far.

This thesis presents a library in Haskell that supports reference manipulation and secure multithreaded programming. In the first part, we extend Li and Zdancewic's work with references. New non-trivial problems emerge due to the presence of references. This thesis shows how they can be resolved by type classes, singleton types, projection functions, and unification. A case study is implemented to evaluate the expressiveness of the library. To the best of our knowledge, it is the biggest security-typed program in functional language so far. In the second part, the support for multithreaded programming is added to the library. Another case study is conducted to show how feasible and dangerous the timing channels are in multithreaded programs, and also corroborates our proposed techniques to close the channels.

Acknowledgement

This work wouldn't be possible without kindly support from many people.

First of all, I would like to express my immense gratitude to my dear supervisor, Dr. John Hughes, who has suggested me such an exciting topic, has provided me brilliant ideas when I met difficulties, and has given me insightful advice as well as zealous encouragement. I cannot overstate my grateful to Alejandro Russo. This work wouldn't look like it is today without his clever ideas, enthusiastic discussions, and helpful comments. Every meeting with them is informative and enjoyable for me. I learn a lot not only from their professional expertise but also from their optimistic personalities when facing problems. Working with them is definitely an unforgettable experience in my life.

I would like to thank Dr. David Sands, Dr. Andrei Sabelfeld, Aslan Askarov, and people in the Multi Group at Department of Computer Science and Engineering for providing useful comments on many parts of this work. I also like to thank Börje Johansson for helping me facilitate the whole process.

I want to dedicate this work to my dearest family: my grand mother, Cho-she Tsai, who always fully supports me since I was a child, my father, Cheng-chih Tsai, and my mother, Li-ching Chao, who encourage me to pursue my dream and support me with infinite love, and my brother, Yu-chung Tsai, with whom I learn, play, and grow. I am also grateful to my lovely relatives whose constantly care and greetings make me feel warm even in the cold winter of Sweden.

I cannot continue without thanking a group of extraordinary friends. They are Liting Chao, Chien-chih Chen, Chien-huei Chen, Fei-neng Chuang, Yu-ting Liao, Yuehting Liao as well as other members of NCTU Europe. We share the happiness and the bitterness of studying in Sweden and strive for graduation together. I will miss our wonderful weekly parties and movie nights.

In the end, I like to express my gratitude to all my friends in Taiwan for helping me in every aspect and making me feel connected despite far from home. In particular, I would like to thank Cheng-dar Li, Yi-tzu Lin, and Fu-kuo Tseng.

Chapter 1

Introduction

Protecting the confidentiality of data has become an important issue nowadays. Everyday, computer systems process delicate information in order to satisfy users' requests. For instance, credit card numbers, email addresses, and other personal information are required to do on-line shopping. Therefore, it is necessary to guarantee that private data is not revealed, intentionally or unintentionally, to inappropriate entities. Otherwise, several unpleasant situations could occur, e.g. receiving extra bills in our credit cards or getting tons of spam in our in-boxes.

Some security problems, w.r.t. confidentiality of data, arise from intentional or unintentional mistakes in programming. Consider a bank system containing two procedures: *fullCCNumber* and *last4CCNumber*, which respectively reveal all and the last four digits of a given credit card. It is then possible for a programmer to call *fullCCNumber* instead of *last4CCNumber* by mistake or on purpose. Thus, full credit card numbers would be sent or displayed in places where the last four digits were required in order to protect credit card clients from fraudsters.

The problems above arise from a lack of mechanisms to enforce confidentiality policies of users. In particular, we would like to enforce that information controlled by a confidentiality policy does not flow into locations where that policy is violated. Thus, we are interested in guaranteeing *information-flow* policies. As described by Sabelfeld and Myers [15], there has been many conventional security mechanisms used in practical computing system for protecting confidential information. These mechanisms include access controls, firewalls, and anti-virus software. Unfortunately, none of them are enough to guarantee information-flow policies. Access control, for instance, allows users who have sufficient privilege to acquire secret data, but it does not control how that data is used and propagated through the whole system.

In the past few decades, researchers have shown that language techniques are very useful to guarantee information-flow policies [15], in particular, *non-interference* policies [3]. A non-interference policy basically allows manipulation of secret data as long as the public output of programs is not affected. Programs that satisfy this policy are called non-interfering. Static analysis techniques are useful to guarantee information-flow policies, e.g. non-interference, for programs. The reason is that both, static analysis techniques and information-flow policies, require reasoning on all possible branch executions of programs. At the moment, there are two full-scale language implementations: Jif [9], as a variant of Java, and FlowCaml [17], as an extension of Caml. They both adopt programming language techniques to specify and enforce information-flow policies.

Li and Zdancewic [7] present a different approach from Jif and FlowCaml. Rather than producing a new language from the scratch, they designed an embedded sub-language in Haskell to guarantee information-flow policies. This sub-language is based on *arrows* [5]. This approach seems to be promising in order to make information-flow techniques more usable in practice. Programmers can write code in the embedded sub-language when some security properties are required, while regular Haskell code can be used for the rest of the program. Moreover, many existing applications can be easily upgraded to guarantee information-flow policies instead of rewriting whole programs.

Multithreaded programs are ubiquitous in modern computing systems. For instance, network programming is one of the areas where multithreading is used. New threads are often spawned to handle new client's requests. Clients usually access some shared resources placed on the server side. Unfortunately, the presence of shared resources in concurrent settings opens new possibilities for leaking confidential data of programs. Several solutions to that problem have been discussed by different authors [1, 14, 16, 21], but it still remains an open challenge. To the best of our knowledge, there are no programming languages that support concurrency and enforce information-flow policies. Thus, it is not surprising that Jif, FlowCaml, and Li and Zdancewic's work have no support for multithreaded programs.

Contribution

The aim of this work is to provide an embedded language for programmers to write multithreaded programs that respect information-flow policies. We extend Li and Zdancewic's work to support complex security types. This allows programmers to keep track of security types for each component of data precisely rather than make an approximation by one lattice label. Reference manipulation is added in the sub-language and described by *arrows*. A unification mechanism is implemented to pass security types in the framework and alleviates the responsibility of programmers to remember security types of data. For multithreaded programming, primitives for dynamic thread creation and atomic execution are supported in the sub-language. A run-time system is developed to enforce multithreaded information-flow policies. Moreover, two full case studies were implemented to evaluate the expressiveness and effectiveness of the library.

The rest of the thesis is organized as following. Chapter 2 describes background knowledge, existing approaches for multithreaded information flow, and the problem statement of this project. Chapter 3 introduces Li and Zdancewic's work in detail. Chapter 4 presents how reference manipulation is added to our work. A new problem emerges from the security types of references. Our solution to the problem is also described in Chapter 4. Chapter 5 explains the implementation of unification in the sub-language. Chapter 6 presents a case study to evaluate the sub-language developed so far. Chapter 7 explains our approach for multithreaded information flow. Chapter 8 presents a case study to corroborate the approach. Chapter 9 discusses limitations, future work, and concludes.

Chapter 2

Information Flow

Information flow policies express what are the valid flows of information inside a program. Policies are established in such a way that they meet some security goals. For instance, *non-interference* is a policy to guarantee the confidentiality of secret data. In the rest of the chapter, we will explain *non-interference* in more detail.

2.1 Noninterference

Noninterference is a security policy to judge if the confidentiality of data is preserved in a program. Data are categorized by different levels. Assume we classify the input and output variables of a program into secret(*high*) and public(*low*) ones. Low variables are the only piece of information that an attacker can observe. *Noninterference* states that public results are not affected by any change in *high* variables.

2.1.1 Direct and Indirect Flows

There are generally two different ways to leak secret information into public output: *explicit* and *implicit* flows. On one hand, *explicit* flows leak information by assigning secret values directly to public variables. Assume we have two variables, called `h` and `l`, which store secret and public values, respectively. The assignment `l := h` is an example of an *explicit* flow. *Implicit* flows [2], on the other hand, leak information in a more indirect way. Consider the following program:

$$a_1 : \texttt{if}\,\texttt{h} > 0\,\texttt{then}\,\texttt{l} := 1\,\texttt{else}\,\texttt{l} := 0$$

The content of `l` depends on the content of `h`. An attacker can thus have information about `h` by looking at the value of `l` because value of `l` might change by modifying value of `h`. Program a_1 fails to meet the *non-interference* policy.

2.1.2 Security Conditions

Depending on the capability of attackers, different kind of security conditions are established to guarantee *non-interference* policy. The two more frequently used are called *termination-insensitive* and *termination-sensitive* noninterference. Termination-insensitive noninterference states that two terminating runs of a program, which agree on public inputs, produce the same public outputs. This security condition assumes that the attackers cannot distinguish between terminating and non-terminating execution of programs, otherwise termination-sensitive noninterference is required. Termination-sensitive noninterference states that two runs of a program, which agree on public inputs, either both diverge or produce the same public output.

2.1.3 Declassification

The non-interference policy alone is too restrictive to implement practical systems. Sometimes, it is necessary to allow public(some declassified) results of computations that involve secrets. For instance, a remote log-in program reveals whether or not the password stored in the system(secret data) matches the password entered by the user(public data). Declassification is a mechanism to decrease security level of data in a controlled way. Jif [9] has a declassification mechanism based on *decentralized label model*(DLM) [8]. In this model, the authorities of data are expressed as principals and act-for relations. The authorities of different levels form a lattice structure. Data can be declassified only if the authorities which perform declassification are higher than the authorities of the data.

2.2 Information Flow in Multithreaded Programs

Shared memory plays an essential role in multithreaded programs. Unfortunately, it also introduces new unintended ways to leak information. As an example, consider the following two thread commands:

$$\begin{array}{l} c_1 : \ (\texttt{if}\ \texttt{h}\ >\ 0\ \texttt{then}\ \texttt{skip}(120)\ \texttt{else}\ \texttt{skip}(1));\ \texttt{l} := 1 \\ c_2 : \ \texttt{skip}(60);\ \texttt{l} := 0 \end{array} \tag{2.1}$$

where h and l are high and low variables, respectively. Assume $\texttt{skip}(n)$ executes n consecutive *skip* commands. Notice that both c_1 and c_2 are secure under the notion of *noninterference* discussed in Sec. 2.1. By considering a scheduler with time slice of 80 steps, it is possible to leak information about h when c_1 and c_2 are run in parallel. To illustrate that assume that the scheduler always starts by running c_1. On one hand, assume that $\texttt{h} > 0$. Then, while running the command $\texttt{skip}(120)$, c_2 is scheduled and run until completion. After that, c_1 is scheduled again and completes its execution. The final value of l is 1. On the other hand, assume that $\texttt{h} \leq 0$. Then c_1 is scheduled and finishes its execution. Now, command c_2 is the only thread alive and it runs until completion. In this case, the final value of h is 0. An attacker can, therefore, deduce if $\texttt{h} > 0$(or not) by observing the final value of l. As a consequence of that, security policies need to be adapted for multithreaded programs.

2.2.1 Noninterference in Multithreaded Programs

Originally, non-interference is defined for deterministic programs. A way to guarantee this policy for concurrent system is by introducing non-determinism in its definition. For instance, *Possibilistic noninterference*, explored by Smith and Volpano [19], states that the possible *low* outputs of a program are independent of *high* inputs. Both c_1 and c_2 obey this definition because whatever the value of h is, the possible results of l are 0,1. This definition of noninterference assumes an attacker has no ability to infer information from probabilistic results of a program. Another example is *probabilistic noninterference* [21]. It requires that the probability distribution of *low* outputs is independent of *high* inputs. Thread c_1 and c_2 running parallel fail this definition. If $\mathtt{h} > 0$, variable `l` has a higher probability to be 1 in the end. Otherwise, variable `l` has a higher probability to be 0 in the end. Apart from the non-interferences described above, Roscoe [13] proposes *low-view determinism*. It states that whatever *high* inputs are changed, *low* program traces are the same.

2.2.2 Existing Approaches

Volpano and Smith [21] propose a language with static threads. They assume an uniform scheduler in which each thread has the same probability of being chosen. They also introduce a new primitive called `protect` to force a sequence of commands to be executed atomically. They claim that if every terminating command with guards containing *high* variables executes atomically, the program satisfies *probabilistic noninterference*. Commands wrapped by protect cannot diverge, otherwise all the program will diverge as well. The following is an example of using `protect`:

$$\begin{array}{l} d_1 : \ \mathtt{protect}(\mathtt{if}\ \mathtt{h}\ >\ 0\ \mathtt{then}\ \mathtt{skip}(120)\ \mathtt{else}\ \mathtt{skip}(0));\ \mathtt{l} := 1 \\ d_2 : \ \mathtt{skip}(60);\ \mathtt{l} := 0 \end{array} \tag{2.2}$$

Since the if-statement of d_1 is executed atomically, d_2 cannot distinguish the timing difference between the branches of the if-statement. Therefore, the probability distribution of `l` is independent of `h`. There are two drawbacks in their approach. First, the semantic of `protect` is nonstandard and it is not clear how to implement such a primitive. Second, a scheduler that chooses a thread uniformly from a threadpool is hard to achieve in practice as well.

Boudol and Castellani [1] give a type system that accepts programs satisfying *possibilistic noninterference*. Basically, the type system rejects programs with low assignments after a command with a high guard. Since the execution order of low assignments do not depend on high values, low outputs will not change as high inputs vary. The following program is rejected.

$$\begin{array}{l} e_1 : \ (\mathtt{if}\ \mathtt{h}\ >\ 0\ \mathtt{then}\ \mathtt{skip}(120)\ \mathtt{else}\ \mathtt{skip}(1));\ \mathtt{l}_1 := 1 \\ e_2 : \ \mathtt{skip}(60);\ \mathtt{l}_2 := 0 \end{array} \tag{2.3}$$

However, the execution order of low assignments does not leak information about `h`. This shows the type system potentially reject secure and useful programs.

Russo and Sabelfeld [14] separate threads of different security levels and treat them differently in the scheduler to ensure the interleaving of publicly-observable events does not depend on secret data. Threads are categorized as high threads and low threads, and are put into different threadpools. The running time of high threads is invisible to low threads. They introduce primitives such as `hide` and `unhide` to signal the scheduler to immediately suspend the execution of some threads. This approach requires interaction between threads and the scheduler. The approaches mentioned so far deal with *internal timing* leaks, which assumes an attacker can only judge timing difference by looking at low outputs.

Sabelfeld and Sands [16] provide a padding technique to make both branches of a high guard command have identical running time. This approach deals with *external timing*, which assumes an attacker has the power to count exact running time of a program. Nevertheless, it requires underlying operating system and hardware to preserve the timing property of each command. Padding techniques may also change the efficiency of a program.

Zdancewic and Myers [22] develop a type system to guarantee concurrent information-flow security based on *low-view determinism*. Programs with race conditions of *low* variables are rejected. Their approach rejects secure programs like ($l := 1 \,||\, l := 0$) and has the potential to reject useful programs.

Information flow in multithreaded program is still an open question. All the researches above gives theoretical results. There is no tool to support information flow security in multithreaded programs. Therefore, there is no case study of concurrent systems which satisfy noninterference.

2.3 Definition of Terms

FlowHaskell The previous work done by Li and Zdancewic [7].

FlowHaskellRef The secure embedded language developed in this project.

Information-flow policies The information-flow policies adopted by FlowHaskell-Ref is *termination-insensitive noninterference* with *low-view determinism*.

2.4 Problem Statement

The goal of the project is to develop a multithreaded embedded language that enforce information-flow policies in Haskell. It contains two main parts.

1. Extending reference manipulation in FlowHaskell
2. Eliminating internal timing channels in FlowHaskellRef

References are shared and should be modelled with two security labels [12] , one for the content and one for the reference's identity. A lattice label adopted in FlowHaskell as security types is insufficient. Besides, FlowHaskellRef relies heavily on Haskell functions to build interesting programs. New security types in FlowHaskell-Ref make it difficult to define input and output security types of Haskell functions. New approaches are required to handle Haskell functions.

Multithreaded programs with shared resources open new timing channels. This has been understood for a long time. However, there are still no practical tools so far. Jif [9], FlowCaml [17], and FlowHaskell [7] all have no support for multithreaded information flow. In Chapter7, we propose a run-time system for FlowHaskellRef to eliminate *internal-timing* channels.

Chapter 3

Encoding Information Flow in Haskell

This chapter begins with a brief introduction to *arrow* interface, and then we explain the previous work that our project is based on.

3.1 The Arrows Interface

Arrows is proposed by Hughes [5] to represent a set of computations. It is aimed to be a common interface for libraries. Comparing to monads, *arrows* also parameterises on input type, and thus has the possibility to choose computations based on static information. The *arrow* interface is categorized in several type classes and each one provides a different functionality to model computations. The following introduces two type classes that is used in the project. Type class `Arrow` provides lifting, sequential as well as parallel computations:

```
class Arrow a where
  pure   :: (b -> c) -> a b c
  (>>>)  :: a b c -> a c d -> a b d
  first  :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***)  :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&)  :: a b c -> a b c' -> a b (c, c')
```

where `a` is an abstract data type for *arrows*, and `b` and `c` are input and output types respectively. Operator `pure` lifts a Haskell function of type (`b` $\rightarrow$ `c`) into an arrow. Infix operator ($\ggg$) composes two *arrows* in sequence where output type of first arrow should be identical to input type of second arrow. Operator `first` composes an arrow parallel with an identity arrow. These three operators are minimum to define an `Arrow` instance and the remaining operators have default definitions based on the three. Operator `second` has an opposite effect of parallel composition to `first`. Operator ($***$) parallel composes two arbitrary *arrows*, while (&&&) first duplicates the input and does the same as operator ($***$).

The basic `Arrow` type class does not support conditional computations. Type class `ArrowChoice` provide such possibility via `Either` type in Haskell.

```
data Either a b = Left a | Right b

class Arrow a => ArrowChoice a where
  left  :: a b c -> a (Either b d) (Either c d)
  right :: a b c -> a (Either d b) (Either d c)
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b d -> a c d -> a (Either b c) d
```

Operator `left` transforms an arrow to support conditional computations based on type constructor of `Either`. If input is of constructor `Left`, value inside `Left` will be fed to the input arrow. The output is again wrapped by type constructor `Left`. Otherwise, the input passes through as the output. Other operators have default definitions based on `left`. Combinator `right` feeds input to the arrow inside if the input is of constructor `Right`. Similarly, (+++) and (|||) feed input to one of the *arrows* depending on the type constructor of the input.

Both [6] and [11] are good tutorials for further study of *arrows*.

3.2 FlowHaskell

Instead of designing a new language from the scratch, Li and Zdancewic [7] showed how to develop an embedded security language in Haskell.[1] The approach used by FlowHaskell is considerably more light-weight comparing to Jif [9] and FlowCaml [17], which drastically modify their ancestor compilers to enforce information flow policies. FlowHaskell expresses security levels as a lattice and adopts *arrows* as language interface. Constraints, related to information flow policies, are introduced by the *arrow* combinators. Those constraints, when satisfied, guarantee that no data of a higher level in the lattice flows to a place where data of a lower level is expected, unless authorized declassification happened. Constraints are generated and collected at the beginning of the execution of programs. Programs perform computations only if the constraints are satisfied. Sec. 3.2.1 explains how data are categorized into different levels in a lattice, and Sec. 3.2.2 presents example programs written in FlowHaskell. The rest sections introduce implementation of FlowHaskell in more detail.

3.2.1 Security Lattice

FlowHaskell provides a generic type class `Lattice` for programmers to define security labels and their relations.

```
class (Eq a) => Lattice a where
  label_top    :: a
  label_bottom :: a
  label_join   :: a -> a -> a
  label_meet   :: a -> a -> a
  label_leq    :: a -> a -> Bool
```

Any arbitrary type can be used as a security label as long as the operations in `Lattice` are supported. Programmers can define their own security labels according to their needs.

3.2.2 Programming in FlowHaskell

In this section, we assume a simple security type containing two security labels.

[1]From now on, we will refer to their work as FlowHaskell.

```
data TwoLabel = HIGH | LOW
instance Lattice TwoLabel where
  label_top = HIGH
  label_bottom = LOW
  label_join x y = if x `label_leq` y then y else x
  label_meet x y = if x `label_leq` y then x else y
  label_leq HIGH LOW = False
  label_leq _ _ = True
```

In FlowHaskell, a polymorphic abstract data type `Protected a` represents a computation with output type `a`. A computation of type `Protected a` cannot be performed freely.

New values are created in Haskell functions and lifted to `Protected a` by `pure`. Operator `tag` asserts certain security level in computations. It takes a security label `l` and specifies that input and output data are protected by `l`. The following programs protect data with operator `tag`.

```
t :: Protected Int
t = pure (\_ -> 3::Int)

ht = t >>> tag HIGH
lt = t >>> tag LOW
```

The result of `ht` is a 3 protected by security label `HIGH`, while the result of `lt` is a 3 protected by security label `LOW`. The following program adds the results of `ht` and `lt`.

```
sum :: Protected Int
sum = (ht &&& lt) >>> pure (\(x,y) -> x+y)
```

The result is protected by security label `HIGH` because the result of (&&&) is protected by `HIGH`, the join of security labels of `ht` and `lt`. Again, we can use `tag` to protect data as following:

```
lsum = sum >>> tag LOW
```

However, program `lsum` fails to obey information-flow policies since the result of `sum` is protected by `HIGH`. The violation of information-flow policies is not reflected until computation `lsum` is performed. To make `lsum` legal, declassification is required. Operator `declassify` takes two security labels. The first one is a security label of input and the second one is a declassified security label for output. Computation `lsum′` becomes secure with declassification as following:

```
lsum′ = sum >>> declassify HIGH LOW >>> tag LOW
```

A protected computation can only be performed by operator `cert`. Operator `cert` takes a privilege, a protected computation, and a unit as the input to the protected computation. The authority has to be higher enough to perform all declassification in the protected computation. Operator `cert` requires output security label being `LOW`. The following program performs computation `lsum′`:

```
r = cert (PR HIGH) lsum′ ()
```

Data `PR` is a type constructor of privileges. Privilege `HIGH` is required because we declassify data from `HIGH` to `LOW` in `lsum′`.

Branch computations can be constructed in FlowHaskell using `Either` type.

```
hr = pure (\x -> if x>0 then Left x else Right x) >>>
     (pure (\y -> y-1) ||| pure id)
```

If the input of `hr` is bigger than 0, it is subtracted by 1. It keeps the same, otherwise.

3.2.3 FlowArrow

`FlowArrow` is an abstract data type implementing the *arrow* interface and used as the embedded language in FlowHaskell. The definition is as follows:

```
data FlowArrow l a b c = FA
     { computation :: a b c
     , flow        :: Flow l
     , constraints :: [Constraint l] }

data Flow l = Trans l l | Flat

data Constraint l = LEQ l l | USERGEQ l
```

Type variable `l` represents a security label belonging to type class `Lattice`, and variable `a` is an *arrow* that performs desired computation. Field `computation` stores an *arrow* that takes an input of type `b`, and returns a value of type `c`. Field `flow` keeps track of the security levels of the input and output of the arrow in `computation`. Constructor `Trans` l_1 l_2 denotes a computation that expects a input security label l_1 and gives an output security label l_2. Constructor `Flat` denotes a computation where input and output have the same security level. Field `constraints` contains a list of constraints required to guarantee information flow policies when the computation is executed. Constraint (`LEQ` l_1 l_2) requires $l_1 \sqsubseteq l_2$ to be satisfied. Constraint (`USERGEQ` `l`) demands that the authority of the computation has a privilege greater than or equal to `l`.

`FlowArrow` requires underlying computations to be *arrows*. The simplest *arrow* is the function arrow ($\rightarrow$). Security flows and constraints are created at the same time that underlying computations are constructed. The following is part of the implementation of `Arrow` class:

```
instance (Lattice l, Arrow a) => Arrow (FlowArrow l a) where
  pure f = FA { computation = pure f
              , flow = Flat
              , constraints = [] }
  (FA c1 f1 t1) >>> (FA c2 f2 t2) =
      let (f,c) = flow_seq f1 f2 in
           FA { computation = c1 >>> c2
              , flow = f
              , constraints = t1 ++ t2 ++ c }
  ...
  (FA c1 f1 t1) &&& (FA c2 f2 t2) =
      FA { computation = c1 &&& c2
         , flow = flow_par f1 f2
         , constraints = t1++t2 }
  ...

flow_seq :: Flow l -> Flow l -> (Flow l, [Constraint l])
flow_seq (Trans l1 l2) (Trans l3 l4) = (Trans l1 l4,[LEQ l2 l3])
flow_seq Flat f2 = (f2,[])
flow_seq f1 Flat = (f1,[])

flow_par :: (Lattice l) => Flow l -> Flow l -> Flow l
flow_par (Trans l1 l2) (Trans l3 l4) =
         Trans (label_meet l1 l3) (label_join l2 l4)
flow_par Flat f2 = f2
flow_par f1 Flat = f1
```

In FlowHaskell, `pure` constructs an *arrow* with `Flat` flow and no constraints. Constraints are introduced during the combination of *arrows*. The ($\ggg$) combinator, for example, requires the output security label of the first *arrow* to be less than or equal to the input security label of the second *arrow*. See function `flow_seq`. Another example is when we compose two *arrows* in parallel. As showed in `flow_par`, the new input label becomes the meet of two input labels, while the new output label is the join of two output labels.

Type `protected a` in Sec. 3.2.2 is implemented as following:

```
type Protected a = FlowArrow TwoLabel (->) () a
```

It always takes a unit as input.

3.2.4 Tagging Security Label

A new Combinator, called `tag`, is provided to specify expected security labels. The definition is as follows:

```
tag :: (Lattice l, Arrow a) => l -> FlowArrow l a b b
tag l = FA { computation = pure (\x->x)
           , flow = Trans l l
           , constraints = [] }
```

The flow of `tag` is from `l` to `l`, where `l` is provided by programmers. This combinator is used mainly for two reasons. The first reason is to give a definite security label to a value, for instance, a security level to the input of `pure`. An example is as follows:

```
    ... >>> tag HIGH >>> pure (\i -> i+1) >>> ...
```

The `pure` becomes an *arrow* from input label `HIGH` to output label `HIGH` because the output label of `tag` is `HIGH`. The other reason is to ensure the security labels of data are as expected. Consider following example:

```
    pure (\i -> i+1) >>> tag LOW >>> ...
```

The `tag` asserts its input security label is `LOW` and gives `LOW` as its output security label. This eliminates the case where the output security label of the `pure` is higher than `LOW`.

3.2.5 Declassification

The declassification mechanism provided by FlowHaskell is similar to the *decentralized label model*(DLM) [8]. A declassification statement changes security labels which is below the authority of the code to any security labels. In other words, each declassification can be seen as a trusted information leak.

```
declassify :: (Lattice l, Arrow a) =>
              l -> l -> FlowArrow l a b b
declassify l1 l2 =
    FA { computation = pure (\x->x)
       , flow = Trans l1 l2
       , constraints = [USERGEQ l1] }
```

Programmers specify original security label, l_1, and also declassified security label, l_2. A constraint `USERGEQ` is generated to ensure the authority of the code has a greater or equal privilege to l_1.

3.2.6 Policy Enforcement

The programmers construct a `FlowArrow` program using all the combinators described in the previous part of the section. Meanwhile, an underlying computation, which performs computation to solve problems of programmers, is constructed in the field `computation`. To execute the program, all constraints have to be satisfied first, and then the underlying computation is returned from `FlowArrow` and can be executed. The whole process is enforced in `certify`. It is defined as follows:

```
data Priv l = PR l

certify :: (Lattice l) => l -> l -> Priv l
           -> FlowArrow l a b c -> a b c
certify l_in l_out (PR l_user) (FA c f t) =
  if not $ check_levels l_in l_out f then
      error $ "security level mismatch" ++ (show f)
  else if not $ check_constraints l_user t then
      error $ "constraints cannot be met" ++ (show t)
  else c
```

Label `l_in` is the security label of input data to the computation, and `l_out` is the security label of output data. If an `FlowArrow` has a `flow` from $\mathtt{l}_1$ to $\mathtt{l}_2$, function `check_levels` verifies $\mathtt{l\_in} \sqsubseteq \mathtt{l}_1$ and $\mathtt{l}_2 \sqsubseteq \mathtt{l\_out}$. Data type `Priv` takes a lattice label and represents privilege of the authority of the computation. It is required when checking constraint `USERGEQ`. Function `check_constraints` verifies if all constraints can be satisfied given the authority privilege. If both tests are valid, the underlying computation, `c`, is returned.

Operator `cert` in Sec. 3.2.2 is defined in terms of `certify`.

```
cert = certify label_bottom label_bottom
```

Chapter 4

Extending FlowHaskell with Reference Manipulation

As a first step towards having multithreaded information-flow secure programs, we need shared resources. We extend FlowHaskell with reference manipulation. Unfortunately, FlowHaskell cannot be naturally extended to include this feature or others like algebraic data types or exceptions.

4.1 Refinement of Security Types

Similarly to FlowHaskell, FlowHaskellRef can perform computations with any value that is typable in Haskell. However, FlowHaskellRef has more complex security types than just a label annotation.

Values handled by FlowHaskellRef have types associated with them. The types are elements of the language denoted by the following grammar.

$$\tau \ ::= \mathbf{int} \mid \tau\ \mathbf{ref} \mid (\tau, \tau) \mid \mathbf{either}\ \tau\ \tau \mid \ \ldots$$

Figure 4.1: Data type

In FlowHaskell, every type has only associated one security label as its security type. According to Pottier and Simonet [12], references require at least two security labels in their security types. Security types handled by FlowHaskellRef, denoted by s^l_l, are elements of the following grammar.

$$s^l \ ::= \ \ell \mid s^l\ \mathbf{ref}^\ell \mid (s^l, s^l) \mid (\mathbf{either}\ s^l\ s^l\ )^\ell \mid \mathbf{high}$$

Figure 4.2: Security type

where l is a provided lattice. The security types for references, pairs, and $\mathbf{either}$ values are respectively $s^l\ \mathbf{ref}^\ell$, (s^l, s^l), and $(\mathbf{either}\ s^l\ s^l)^\ell$. Security type $\mathbf{high}$, whose purpose is explained in Section 4.4, denotes any security type where all the label annotations are the top element of the lattice. Values which are not references, pairs, or $\mathbf{either}$s have security labels, denoted by ℓ, as their security types.

Security types s^l in Figure 4.2 is implemented as follows(in Lattice.hs):

$$\frac{s_1^l = s_2^l \quad \ell_1 \sqsubseteq \ell_2}{s_1^l\ \mathbf{ref}^{\ell_1} \sqsubseteq s_2^l\ \mathbf{ref}^{\ell_2}} \qquad \frac{s_1^l \sqsubseteq s_3^l \quad s_2^l \sqsubseteq s_4^l}{(s_1^l, s_2^l) \sqsubseteq (s_3^l, s_4^l)}$$

$$\frac{\ell_1 \sqsubseteq \ell_2 \quad s_1^l \sqsubseteq s_3^l \quad s_2^l \sqsubseteq s_4^l}{(\mathbf{either}\ s_1^l\ s_2^l)^{\ell_1} \sqsubseteq (\mathbf{either}\ s_3^l\ s_4^l)^{\ell_2}}$$

Figure 4.3: Subtyping

```
data SecType l = SecLabel l | SecRef (SecType l) l |
                 SecPair (SecType l) (SecType l) |
                 SecEither (SecType l) (SecType l) l |
                 SecHigh
```

where type variable `l` is instantiated with any arbitrary lattice.

Security type s^l exhibits a partial ordering among security labels of identical structure, which is described in Figure 4.3. For security labels ℓ in s^l, the sub-typing relationship is the same as in the lattice l. Data type `SecType` implements sub-typing relationship in type class `Ord`. The following is the implementation for security labels, references, and pairs(in Lattice.hs).

```
instance Lattice l => Ord (SecType l) where
  (SecLabel l1) <= (SecLabel l2) = label_leq l1 l2
  (SecRef st1 lr1) <= (SecRef st2 lr2) =
              label_leq lr1 lr2 && (st1 <= st2) && (st2 <= st1)
  (SecPair s1 t1) <= (SecPair s2 t2) = (s1 <= s2) && (t1 <= t2)
  ...
  max (SecLabel l1) (SecLabel l2) =
                          SecLabel (Lab (label_join l1 l2))
  max (SecRef t1 l1) (SecRef t2 l2) =
                          if t1 /= t2
                            then error "SecRef : no join defined."
                            else SecRef t1 (Lab (label_join l1 l2))
  max (SecPair s1 t1) (SecPair s2 t2) = SecPair (max s1 s2) (max t1 t2)
  ...
  min (SecLabel l1) (SecLabel l2) =
                          SecLabel (Lab (label_meet l1 l2))
  min (SecRef t1 l1) (SecRef t2 l2) =
                          if t1 /= t2
                            then error "SecRef : no meet defined."
                            else SecRef t1 (Lab (label_meet l1 l2))
  min (SecPair s1 t1) (SecPair s2 t2) = SecPair (min s1 s2) (min t1 t2)
```

Three methods, (<=), `max`, and `min`, are required to be implemented. Method (<=) implements the sub-typing relationship in Figure 4.3. For security labels, the sub-typing relationship is the same as in the lattice. Notice the security type of a reference's content is invariant and implemented by two (<=)s. Method `max` returns the join of two security types. Basically, it computes the join by applying `max` recursively to the sub-components of both security types. Moreover, it is only defined for references with the same content security types due to the sub-typing relationship. Method `min` returns the meet of two security types and is implemented similar to `max`.

Programmers only need to define a lattice type to achieve their goals as in FlowHaskell. Security type `SecType` is part of the security language and maintained by library providers. Any new security type can be easily extended in `SecType` by adding a new type constructor and implementations of the methods in type class `Ord`.

4.2 Programming in FlowHaskellRef

As in FlowHaskell, programmers have to define a lattice to achieve their goals. In the section, we assume `TwoLabel`, the same as in Sec. 3.2.2. Besides, programmers also need to define singleton types for each label as following:

```
data SHigh = VHigh
data SLow  = VLow
```

Each singleton type corresponds to a lattice constructor. See Sec. 4.6 for details about singleton types. Programming in FlowHaskellRef is similar to programming in FlowHaskell because they all implement *arrow* interface. A new combinator `lowerA` is used with `pure` to specify security levels of data. The following FlowHaskellRef program illustrates how to specify `ht` and `lt` as in Sec. 3.2.2:

```
t :: Protected Int
t = pure (\_ -> 3)

ht = lowerA HIGH t
lt = lowerA LOW t
```

In FlowHaskellRef, operator `tagRef` corresponds to `tag` in FlowHaskell. However, rather than permitting a fixed security label, it pushes up all security labels in the input security type to as least the same level as the given security label. Consider the following program.

```
ht' = lt >>> tagRef HIGH
```

The result of `ht′` is protected by `SecLabel` (`Lab HIGH`). Security label `Lab HIGH` can be regarded as `HIGH` for the time being. Constructor `Lab` is explained in Sec. 5.1.1.

New security types and `lowerA` provide more flexibility than that in FlowHaskell. Consider the following program.

```
fstPair :: Protected Int
fstPair = (ht &&& lt) >>> lowerA LOW (pure (\(x,y) -> y))
```

The result security type of (`ht` &&& `lt`) contains result security types of both `ht` and `lt`. Thus, value `x` is protected by `HIGH`, while value `y` is protected by `LOW`. As long as `x` is not used in the function of `pure`, the program obeys information-flow policies.

In declassification, operator `declassifyRef` takes a declassified security type. The security type is specified in the singleton types. The following program shows how to declassify `ht`.

```
lt' = ht >>> declassifyRef LOW
```

The result security type of `lt′` becomes `SecLabel` (`Lab LOW`).

Besides, FlowHaskellRef supports reference manipulation. A reference created for the result of `lt` is as following.

```
rlt = lt >>> createRefA (VSecLabel VLow) LOW
```

Security type `VSecLabel VLow`, specified in the singleton types, is the security type for the reference's content. Constructor `VSecLabel` belongs to singleton type `SSecLabel`. It corresponds to `SecLabel` and is defined in the library. Security label `LOW` is for the reference's identity. The following program reads out the reference.

```
hlt = rlt >>> readRefA
```

We can also write the result of `lt` again to the reference.

```
rlt' = (rlt *** lt) >>> writeRefA
```

Input of `writeRefA` is a pair of a reference and its new content.

To perform a FlowHaskellRef computation, operator `certifyRef` takes an input security type, an output security type, a priviledge, and a protected program. Computation `lt'` is certified in the following program.

```
certifyRef (SecLabel (Lab LOW)) (SecLabel (Lab LOW)) (PR HIGH) lt'
```

4.3 Defining a New Arrow

Reference manipulation are accompanied by side effects that cannot be tracked by the security types in FlowHaskell. However, it is enough to have an *arrow* able to handle the security types described in Sec. 4.1 in addition to new kind of constraints that guarantee information-flow policies in presence of references. Thus, we firstly need to define a new data type to represent *arrow* computations our library. It replaces `FlowArrow` and implements *arrow* interface.

4.3.1 Arrows Computation

Values that belongs to the following data type represent *arrow* computations in FlowHaskell-Ref.

```
data FlowArrowRef l a b c =
    FARef { computation :: a b c
          , flow        :: Flow (SecType l)
          , constraints :: [Constraint (SecType l)]
          , pc          :: l
          }

data Flow l = Trans l l
data Constraint l = LEQ l l | USERGEQ l | IS l l |
                    GUARD l l | GSLEQL l l | GLLEQS l l
```

`FlowArrowRef` has four type parameters, where `l` is a lattice type, `a` is an arrow that performs desired computations, `b` and `c` are the input and the output types respectively. New definition of `Flow` removes constructor `Flat` which loses structure information of a security type. Four new constraints are added to `Constraint`. Constraint `IS` requires two security type having the same structure. Constraint `GUARD` takes a security label and a security type, and requires the security level of the security type higher than or equal to the security label. Constraint `GSLEQL` takes a security type and a security label, and the security label is the upper bound of all security labels appearing in the security type. Constraint `GLLEQS` takes a security type and a security label, and the security label is the lower bound of all security labels appearing in the security type. A formal description of these constraints are given in the next section. Field `pc` keeps track of the lower bound of visible side effects produced in a computation.

4.3.2 Type System

The type system conceptually implemented by `FlowArrowRef` is depicted in Figure 4.4, Figure 4.5, and Figure 4.6. The type judgement has following form:

$$pc, \Theta \vdash f \; : \; \tau_1 \mid s_1^l \;\rightarrow\; \tau_2 \mid s_2^l$$

$$(\mathit{PURE})\frac{f\ :\ \tau_1\ \rightarrow\ \tau_2}{\top, \emptyset \vdash \texttt{pure}\ f\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid \mathbf{high}}$$

$$(\mathit{SEQ})\frac{pc_1, \Theta_1 \vdash f_1\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid s^l_2 \quad pc_2, \Theta_2 \vdash f_2\ :\ \tau_2 \mid s^l_3\ \rightarrow\ \tau_4 \mid s^l_4}{pc_1 \sqcap pc_2, \Theta_1 \cup \Theta_2 \cup \{s^l_2 \sqsubseteq s^l_3\} \vdash f_1\ \ggg\ f_2\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_4 \mid s^l_4}$$

$$(\mathit{FIRST})\frac{pc, \Theta \vdash f\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid s^l_2}{pc, \Theta \vdash \texttt{first}\ f\ :\ (\tau_1, \tau_3) \mid (s^l_1, s^l_3)\ \rightarrow\ (\tau_2, \tau_3) \mid (s^l_2, s^l_3)}$$

$$(\mathit{SECOND})\frac{pc, \Theta \vdash f\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid s^l_2}{pc, \Theta \vdash \texttt{second}\ f\ :\ (\tau_3, \tau_1) \mid (s^l_3, s^l_1)\ \rightarrow\ (\tau_3, \tau_2) \mid (s^l_3, s^l_2)}$$

$$(\mathit{PAR1})\frac{pc_1, \Theta_1 \vdash f_1\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid s^l_2 \quad pc_2, \Theta_2 \vdash f_2\ :\ \tau_3 \mid s^l_3\ \rightarrow\ \tau_3 \mid s^l_4}{pc_1 \sqcap pc_2, \Theta_1 \cup \Theta_2 \vdash f_1\ {*}{*}{*}\ f_2\ :\ (\tau_1, \tau_3) \mid (s^l_1, s^l_3)\ \rightarrow\ (\tau_2, \tau_4) \mid (s^l_2, s^l_4)}$$

$$(\mathit{PAR2})\frac{pc_1, \Theta_1 \vdash f_1\ :\ \tau_1 \mid s^l_1\ \rightarrow\ \tau_2 \mid s^l_2 \quad pc_2, \Theta_2 \vdash f_2\ :\ \tau_1 \mid s^l_3\ \rightarrow\ \tau_4 \mid s^l_4}{pc_1 \sqcap pc_2, \Theta_1 \cup \Theta_2 \vdash f_1\ \&\&\&\ f_2\ :\ \tau_1 \mid (s^l_1 \sqcap s^l_3)\ \rightarrow\ (\tau_2, \tau_4) \mid (s^l_2, s^l_4)}$$

Figure 4.4: Type system of methods in type class `Arrow`

Environment variable pc is the lower bound of side effects produced by computation `f`. Variable Θ denotes constraints of the form $s^l \sqsubseteq s^l$, $s^l \sqsubseteq \texttt{user}$, $s^l \sim s^l$,$\ell \lhd s^l$, $s^l \blacktriangleleft \ell$, and $\ell \preceq s^l$. Constraint $\sim$ corresponds to `IS` and is resolved by rules depicted in Figure 4.7. Constraints $\lhd$, $\blacktriangleleft$, and $\preceq$ are implemented by `GUARD`, `GSLEQl`, and `GLLEQS`, respectively. They are resolved by the rules in Figure 4.8, Figure 4.9, and Figure 4.10. Function e used in Figure 4.8 is defined in Figure 4.11. It returns a security label representing the security level of a security type. This is used to prevent *implicit* information flow. Computation `f` has input type τ_1 with security type s^l_1 and output type τ_2 with security type s^l_2.

Combinator `pure` lifts any Haskell function `f` into `FlowArrowRef`. Function `f` takes the input security types s^l_1. Since it is difficult to statically predict how the input of `f` is used to build the output, the output security type of (`pure f`) cannot be precisely established. We approximate its output security type by **high**. In Sec. 4.4, a more detail discussion about this decision can be found. Rule (`SEQ`) ensures output security type of the first computation is less than or equal to input security type of the second one. The resulting computation has an lower bound of side effect which is the meet between pc_1 and pc_2. In rule (`FIRST`), (`SECOND`), and (`PAR1`), both input and output types and security types are wrapped in a pair constructor. In rule (`PAR2`), both *arrow* computations should have the same input type. The meet of the two input security types becomes new input security type.

Rules for branching computations are shown in Figure 4.5. Rule (`LEFT`) and (`RIGHT`) lift the input and output types and security types in $either$ and **either** types respectively. In rule (`CHOICE1`), the constraint $(\mathbf{either} s^l_1\ s^l_3)^\ell \blacktriangleleft (pc_1 \sqcap pc_2)$ expresses that the security label of the lowest side effect in the computations f_1 and f_2 should be above or equal to all the security labels in $(\mathbf{either}\ s^l_1\ s^l_3)^\ell$. This prevents from leaking some

$$(\mathit{LEFT})\ \frac{pc, \Theta \vdash f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l}{pc, \Theta \vdash \texttt{left}\, f \ : \ \mathit{either}\, \tau_1\, \tau_3 \mid (\mathbf{either}\ s_1^l\ s_3^l)^\ell \ \rightarrow \ \mathit{either}\, \tau_2\, \tau_3 \mid (\mathbf{either}\ s_2^l\ s_3^l)^\ell}$$

$$(\mathit{RIGHT})\ \frac{pc, \Theta \vdash f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l}{pc, \Theta \vdash \texttt{right}\, f \ : \ \mathit{either}\, \tau_3\, \tau_1 \mid (\mathbf{either}\ s_3^l\ s_1^l)^\ell \ \rightarrow \ \mathit{either}\, \tau_3\, \tau_2 \mid (\mathbf{either}\ s_3^l\ s_2^l)^\ell}$$

$$(\mathit{CHOICE1})\ \frac{pc_1, \Theta_1 \vdash f_1 \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l \quad pc_2, \Theta_2 \vdash f_2 \ : \ \tau_3 \mid s_3^l \ \rightarrow \ \tau_4 \mid s_4^l}{pc_1 \sqcap pc_2, \Theta_1 \cup \Theta_2 \cup \mathit{constraint}1 \vdash f_1 +\!\!\!+\!\!\!+ f_2 \ : \ \mathit{flow}1}$$

$$\mathit{flow}1 \ = \ \mathit{either}\, \tau_1\, \tau_3 \mid (\mathbf{either}\ s_1^l\ s_3^l)^\ell \ \rightarrow \ \mathit{either}\, \tau_2\, \tau_4 \mid (\mathbf{either}\ s_2^l\ s_4^l)^\ell$$

$$\mathit{constraint}1 = \{(\mathbf{either} s_1^l\ s_3^l)^\ell \blacktriangleleft (pc_1 \sqcap pc_2)\}$$

$$(\mathit{CHOICE2})\ \frac{pc_1, \Theta_1 \vdash f_1 \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l \quad pc_2, \Theta_2 \vdash f_2 \ : \ \tau_3 \mid s_4^l \ \rightarrow \ \tau_2 \mid s_4^l}{pc_1 \sqcap pc_2, \Theta_1 \cup \Theta_2 \cup \mathit{constraint}2 \vdash f_1 \mid\mid\mid f_2 \ : \ \mathit{flow}2}$$

$$\mathit{flow}2 \ = \ \mathit{either}\, \tau_1\, \tau_3 \mid (\mathbf{either}\ s_1^l\ s_3^l)^\ell \ \rightarrow \ \tau_2 \mid \uparrow (s_2^l \sqcup s_4^l, \ell)$$

$$\mathit{constraint}2 = \{(\mathbf{either} s_1^l\ s_3^l)^\ell \blacktriangleleft (pc_1 \sqcap pc_2), (\mathbf{either}\ s_1^l\ s_3^l)^\ell \blacktriangleleft e(\uparrow (s_2^l \sqcup s_4^l, \ell))\}$$

Figure 4.5: Type system of methods in type class `ArrowChoice`

input information by writing into references.

In rule (`CHOICE2`), the output security type is lifted by function $\uparrow$ because the output is extracted from an **either** constructor. Function $\uparrow$, defined in Figure 4.12, lifts security labels in a security type to the security level the same as the second argument. There are two constraints created. The first one is the same as in rule (`CHOICE1`) and has already been explained. The second constraint requires all the security labels in $(\mathbf{either}\ s_1^l\ s_3^l)^\ell$ is less than or equal to $e(\uparrow (s_2^l \sqcup s_4^l, \ell))$. The constraint is necessary in the case s_1^l or s_3^l has security types higher than the identity security label ℓ. For instance, assume the input security type is $(\mathbf{either}\ (\mathtt{H}\ \mathbf{ref}^{\mathtt{L}})\ \mathtt{L})^{\mathtt{L}}$, where $\mathtt{L} \sqsubseteq \mathtt{H}$. Function $\uparrow$ pushes up the output security type to at least the same level as the identity security label `L`. However, the security type of the reference contains data of security type `H` and the output security type might be not higher enough.

For non-standard combinators shown in Figure 4.6, combinator `tagRef` takes a lattice label, ℓ, and produces an output with security type which has the same constructors as the input security type but with each security label higher or equal to ℓ. Combinator `declassifyRef` takes a security label, ℓ, and declassifies the input security type to a security type with security labels lower than or equal to ℓ. The output security type is computed by function $\downarrow$ and defined in Figure 4.13. Function $\downarrow$ declassifies the first security type based on the second security type and the third security label. The second security type is computed by function ρ, defined in Figure 4.14, which takes a Haskell type and a security label ℓ, and generates a security type with every security label being ℓ for the type. This is necessary since the input security type might be **high**, which has no structural inforamtion. One constraint is generated. Constraint $s_1^l \sqsubseteq \mathtt{user}$ guarantees the authority of the code, denoted by `user`, is higher than or equal to the input

$$(\mathit{TAG})\frac{}{\top, \emptyset \vdash \texttt{tagRef}\,\ell \;:\; \tau \mid s^l \;\rightarrow\; \tau \mid \uparrow(s^l, \ell)}$$

$$(\mathit{DECL})\frac{}{\top, \{s^l_1 \sqsubseteq \texttt{user}\} \vdash \texttt{declassifyRef}\,\ell \;:\; \tau \mid s^l \;\rightarrow\; \tau \mid \downarrow(s^l, \rho(\tau, \ell), \ell)}$$

$$(\mathit{LOWER})\frac{\top, \Theta \vdash f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; \tau_2 \mid s^l_2}{\top, \emptyset \vdash \texttt{lowerA}\,\ell\, f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; \tau_2 \mid \rho(\tau_2, \ell)}$$

$$(\mathit{EQUAL})\frac{\top, \Theta \vdash f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; \tau_2 \mid s^l_2}{\top, \{\ell \preceq s^l_1, s^l_1 \blacktriangleleft \ell\} \vdash \texttt{equalA}\,\ell\, f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; \tau_2 \mid \rho(\tau_2, \ell)}$$

$$(\mathit{ITERATE})\frac{pc, \Theta \vdash f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; (\tau_2, \mathit{bool}) \mid (s^l_2, s^l_3)}{pc, \Delta \cup \{e(s^l_3) \lhd s^l_2, s^l_2 \sqsubseteq s^l_1\} \vdash \texttt{iterateA}\, f \;:\; \tau_1 \mid s^l_1 \;\rightarrow\; \tau_2 \mid s^l_2}$$

Figure 4.6: Type system of new arrow combinators

$$\frac{}{\ell_1 \sim \ell_2} \qquad \frac{s^l_1 \sim s^l_2}{s^l_1\;\mathbf{ref}^{\ell_1} \sim s^l_2\;\mathbf{ref}^{\ell_2}}$$

$$\frac{s^l_1 \sim s^l_3 \quad s^l_2 \sim s^l_4}{(s^l_1, s^l_2) \sim (s^l_3, s^l_4)} \qquad \frac{s^l_1 \sim s^l_3 \quad s^l_2 \sim s^l_4}{(\mathbf{either}\; s^l_1\; s^l_2)^{\ell_1} \sim (\mathbf{either}\; s^l_3\; s^l_4)^{\ell_2}}$$

Figure 4.7: Constraint $\sim$

security type. Combinator `lowerA` takes a security label and a `FlowArrowRef`, and transform the output security type of the computation based on the output type. The detail information of combinator `lowerA` is explained in Sec. 4.5. Combinator `equalA` is similar to `lowerA` but requires all security labels of the input security type the same as the given security label, ℓ. The difference is that using `lowerA` will delay some constraint checking to the run-time of the underlying computation, but `equalA` guarantees all constraints are checked during certification of a program. Combinator `iterateA` takes a `FlowArrowRef` and transforms the underlying computation. If the second element of the output is `True`, the whole computation is repeated again by taking the first element of the output as input. Thus, constraint $s^l_2 \sqsubseteq s^l_1$ is required. The other constraint, $e(s^l_3) \lhd s^l_2$, is necessary because the outputs depends on the boolean value.

The type system for `certifyRef` is described in Figure 4.15. Type judgement $L, \ell_u \vdash f \;:\; \tau_1 \mid s^l_{in} \;\rightarrow\; \tau_2 \mid s^l_{out}$ says given a lattice L and an authority privilege label ℓ_u, computation f takes an input of type τ_1 with security label s^l_{in}, and returns an output of type τ_2 with security label s^l_{out}. Computation f passes certification if all the constraints in the premise are satisfied.

$$\frac{\ell \sqsubseteq e(s^l)}{\ell \lhd s^l}$$

Figure 4.8: Constraint $\lhd$

$$\frac{\ell' \sqsubseteq \ell}{\ell' \blacktriangleleft \ell} \qquad \frac{\ell' \sqsubseteq \ell \quad s^l \blacktriangleleft \ell}{s^l\ \mathbf{ref}^{\ell'} \blacktriangleleft \ell} \qquad \frac{s_1^l \blacktriangleleft \ell \quad s_2^l \blacktriangleleft \ell}{(s_1^l, s_2^l) \blacktriangleleft \ell}$$

$$\frac{\ell' \sqsubseteq \ell \quad s_1^l \blacktriangleleft \ell \quad s_2^l \blacktriangleleft \ell}{(\mathbf{either}\ s_1^l\ s_2^l)^{\ell'} \blacktriangleleft \ell} \qquad \frac{}{\mathbf{high} \blacktriangleleft \top}$$

Figure 4.9: Constraint $\blacktriangleleft$

$$\frac{\ell \sqsubseteq \ell'}{\ell \preceq \ell'} \qquad \frac{\ell \sqsubseteq \ell' \quad \ell \preceq s^l}{\ell \preceq s^l\ \mathbf{ref}^{\ell'}} \qquad \frac{\ell \preceq s_1^l \quad \ell \preceq s_2^l}{\ell \preceq (s_1^l, s_2^l)}$$

$$\frac{\ell \sqsubseteq \ell' \quad \ell \preceq s_1^l \quad \ell \preceq s_2^l}{\ell \preceq (\mathbf{either}\ s_1^l\ s_2^l)^{\ell'}} \qquad \frac{\ell \sqsubseteq \top}{\ell \preceq \mathbf{high}}$$

Figure 4.10: Constraint $\preceq$

$$\frac{}{e(\ell) \rightarrow \ell} \qquad \frac{}{e(s^l\ \mathbf{ref}^{\ell}) \rightarrow \ell} \qquad \frac{e(s_1^l) \rightarrow \ell_1 \quad e(s_2^l) \rightarrow \ell_2}{e((s_1^l, s_2^l)) \rightarrow \ell_1 \sqcup \ell_2}$$

$$\frac{}{e((\mathbf{either}\ s_1^l\ s_2^l)^{\ell}) \rightarrow \ell} \qquad \frac{}{e(\mathbf{high}) \rightarrow \top}$$

Figure 4.11: Function e

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow\ \ell_1\ \ell_2 \rightarrow \ell_2} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow\ \ell_1\ \ell_2 \rightarrow \ell_1}$$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\uparrow\ (s^l\ \mathbf{ref}^{\ell_1})\ \ell_2 \rightarrow s^l\ \mathbf{ref}^{\ell_2}} \qquad \frac{\ell_2 \sqsubset \ell_1}{\uparrow\ (s^l\ \mathbf{ref}^{\ell_1})\ \ell_2 \rightarrow s^l\ \mathbf{ref}^{\ell_1}}$$

$$\frac{\uparrow\ s_1^l\ \ell \rightarrow s_3^l \quad \uparrow\ s_2^l\ \ell \rightarrow s_4^l}{\uparrow\ (s_1^l, s_2^l)\ \ell \rightarrow (s_3^l, s_4^l)} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad \uparrow\ s_1^l\ \ell_2 \rightarrow s_3^l \quad \uparrow\ s_2^l\ \ell_2 \rightarrow s_4^l}{\uparrow\ (\mathbf{either}\ s_1^l\ s_2^l)^{\ell_1}\ \ell_2 \rightarrow (\mathbf{either}\ s_3^l\ s_4^l)^{\ell_2}}$$

$$\frac{\ell_2 \sqsubset \ell_1 \quad \uparrow\ s_1^l\ \ell_2 \rightarrow s_3^l \quad \uparrow\ s_2^l\ \ell_2 \rightarrow s_4^l}{\uparrow\ (\mathbf{either}\ s_1^l\ s_2^l)^{\ell_1}\ \ell_2 \rightarrow (\mathbf{either}\ s_3^l\ s_4^l)^{\ell_1}} \qquad \frac{}{\uparrow\ \mathbf{high}\ \ell \rightarrow \mathbf{high}}$$

Figure 4.12: Function $\uparrow$

$$\frac{\ell_1 \sqsubseteq \ell_2}{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_1} \qquad \frac{\ell_2 \sqsubset \ell_1}{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_2}$$

$$\frac{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_3 \quad \downarrow\ s_1^l\ s_2^l\ \ell_2\ \rightarrow\ s_3^l}{\downarrow\ (s_1^l\ \mathbf{ref}^{\ell_1})\ (s_2^l\ \mathbf{ref}^{\ell_2})\ \ell_2\ \rightarrow\ s_3^l\ \mathbf{ref}^{\ell_3}} \qquad \frac{\downarrow\ s_1^l\ s_3^l\ \ell\ \rightarrow\ s_5^l \quad \downarrow\ s_2^l\ s_4^l\ \ell\ \rightarrow\ s_6^l}{\downarrow\ (s_1^l, s_2^l)\ (s_3^l, s_4^l)\ \ell\ \rightarrow\ (s_5^l, s_6^l)}$$

$$\frac{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_3 \quad \downarrow\ s_1^l\ s_3^l\ \ell_2\ \rightarrow\ s_5^l \quad \downarrow\ s_2^l\ s_4^l\ \ell_2\ \rightarrow\ s_6^l}{\downarrow\ (\mathbf{either}\ s_1^l\ s_2^l)^{\ell_1}\ (\mathbf{either}\ s_3^l\ s_4^l)^{\ell_2}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s_5^l\ s_6^l)^{\ell_3}} \qquad \frac{}{\downarrow\ \mathbf{high}\ s^l\ \ell\ \rightarrow\ s^l}$$

Figure 4.13: Function $\downarrow$

$$\frac{}{\rho(int, \ell)\ \rightarrow\ \ell} \qquad \frac{\rho(\tau, \ell)\ \rightarrow\ s_1^l}{\rho(\tau\ ref, \ell)\ \rightarrow\ s_1^l\ \mathbf{ref}^{\ell}}$$

$$\frac{\rho(\tau_1, \ell)\ \rightarrow\ s_1^l \quad \rho(\tau_2, \ell)\ \rightarrow\ s_2^l}{\rho((\tau_1, \tau_2), \ell)\ \rightarrow\ (s_1^l, s_2^l)} \qquad \frac{\rho(\tau_1, \ell)\ \rightarrow\ s_1^l \quad \rho(\tau_2, \ell)\ \rightarrow\ s_2^l}{\rho(either\ \tau_1\ \tau_2, \ell)\ \rightarrow\ (\mathbf{either}\ s_1^l\ s_2^l)^{\ell}}$$

Figure 4.14: Function ρ

$$(CERTIFY)\frac{\begin{array}{c} pc, \Theta \vdash f\ :\ \tau_1 \mid s_1^l\ \rightarrow\ \tau_2 \mid s_2^l \quad s_{in}^l \blacktriangleleft pc \\ s_{in}^l \sqsubseteq s_1^l \quad s_2^l \sqsubseteq s_{out}^l \quad L \vdash \Theta[\ell_u/\texttt{user}] \end{array}}{L, \ell_u \vdash \texttt{certifyRef}\ f\ :\ \tau_1 \mid s_{in}^l\ \rightarrow\ \tau_2 \mid s_{out}^l}$$

Figure 4.15: Type system of `certifyRef`

4.4 Pure Problem

Combinator `pure` lifts a Haskell function into `FlowArrowRef`. Different from other combinators in FlowHaskellRef, which provide the ability to assemble existing *arrow* computations, combinator `pure` provides programmers with the ability to define new *arrow* computations based on Haskell functions. It is an essential building block when programming in FlowHaskellRef.

However, to derive the output security type of a `pure` computation is not straightforward. One difficulty arises from the output type of a Haskell function is not fixed, and neither is the output security type. It can be an integer, a pair of integers, a reference, or anything else, and they are protected by different security types. The other problem is the dependency of inputs and outputs of a Haskell function is not understood. Considering the following program.

```
pure (\(a,b) -> if a then (a,b) else (b,a))
```

Assume `a` is a `Low` variable and `b` is a `High` variable. Depending on the value of `a`, the output security type can be (`Low`,`High`) or (`High`,`Low`). A Haskell function may contain more complex expressions and makes it impossible to deduce the dependency between the inputs and the outputs.

In FlowHaskell, constructor `Flat` is used to represent the security flow of a `pure` computation. Since no declassification can happen inside of a Haskell function, the output has the same security level as the input. Moreover, only one lattice label is used to model the security level of any data type. The output security label is the same as the input security label whatever the output type is. That is why `Flat` is an adequate choice in the previous work. But in FlowHaskellRef, different data types are protected by different security types. The input and output security types may at the same security level but with different security type constructors.

We avoid the difficulties mentioned above by making an approximation. The output security label of `pure` is set to **high**, as in the type system in Figure 4.4. It means every data becomes top secret afterward. However, it severely restricts the usefulness of `pure`. Consider a simple case that programmers want to increase the content of a reference by 1 and the reference has security type ($\texttt{Low}\ \mathbf{ref}^{\texttt{Low}}$). It is illustrated in the following program.

```
... >>>
(idRef
 &&&
 (readRefA >>>
  pure (\x -> x+1))) >>>
writeRefA
```

The value $\texttt{x} + \texttt{1}$ becomes **high** after `pure` and can not be written back to the reference again. To mitigate the limitation, a new combinator `lowerA` is introduced and explained in Sec. 4.5.

4.5 Combinator `lowerA`

Combinator `lowerA` is introduced in FlowHaskellRef to alleviate the restriction posed by `pure`. It transforms a `FlowArrowRef` computation by filtering out the inputs which have security levels higher than expected and downgrading the output security type to the expected security level. The following figure illustrates how `lowerA` works:

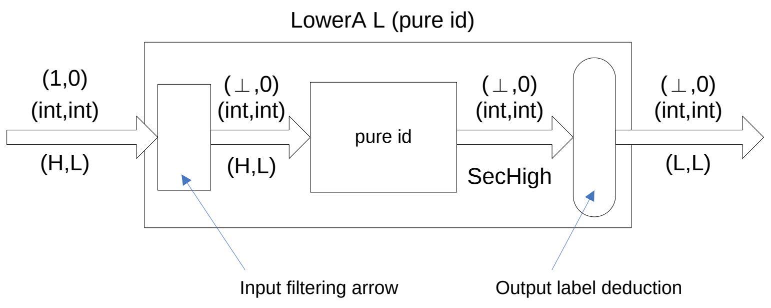

Figure 4.16: Example of `lowerA`

Assume we have a lattice of two labels, `L` and `H` where $\mathtt{L} \sqsubseteq \mathtt{H}$. A pair of integers that the first element has security label `H` and the second element has security label `L` are passed as the inputs. Because the expected label is `L` in the `lowerA`, the first element of the pair becomes `undefined` after passing through the input filter *arrow*. The implementation of input filter arrow is explained in Sec. 4.5.1. The output security type of the `lowerA` is derived from its output type and the security label, `L`. The output type is a pair of integers, so the corresponding security type $(\mathtt{L}, \mathtt{L})$ is generated. The implementation of the output security type deduction is explained in Sec. 4.5.2. If the *undefined* value is used somewhere later in the computation, the whole program aborts immediately and no information can leak.

4.5.1 Input Filtering Mechanism

The input filtering mechanism in `lowerA` is implemented by method `removeData` in type class `FilterData`. Any type used as an input type in computations created by `lowerA` requires an instance in the type class `FilterData`. The implementations for integer, pair, and reference are as following(in SecureFlow.hs):

```
class (Lattice l) => FilterData l t where
  removeData :: l -> (SecType l) -> t -> t

instance (Lattice l) => FilterData l Int where
  removeData l (SecLabel (Lab l')) t =
             if label_leq l' l then t else undefined

instance (Lattice l, FilterData l a
        , FilterData l b) =>
          FilterData l (a,b) where
  removeData l (Pair lx ly) (x,y) =
             (removeData l lx x, removeData l ly y)
```

Method `removeData` takes a lattice label `l`, a complex security type (`s l`), and a value of type `t`. In the instance of `Int`, the security label is compared with `l`. If it is higher than `l`, `undefined` is returned. Otherwise, the same value is returned. This is the same for any data type that is protected by a lattice label. When `removeData` is applied to a pair, `removeData` is applied to both elements with the corresponding security types and values.

4.5.2 Output Security Type Derivation

Function ρ in Figure 4.14 is implemented by method `buildSecType` in type class `BuildSecType`. Type class `buildSecType` deduces security types for a data type based on a type. The definition of BuildSecType and some instances are as following(in SecureFlow.hs):

```
class (Lattice l) => BuildSecType l t where
  buildSecType :: l -> t -> (SecType l)

instance (Lattice l) => BuildSecType l Int where
  buildSecType l _ = (SecLabel (Lab l))

instance (Lattice l, BuildSecType l a
         , BuildSecType l b) =>
           BuildSecType l (a,b) where
  buildSecType l _ = (SecPair (low l x) (low l y))
```

Method buildSecType takes a security label and a value of type t, and returns a corresponding security type for data type t with every label annotation being the security label. In the instance of Int, a SecLabel with label annotation l is returned. In the instance of pair, a constructor SecPair is returned with method buildSecType applied again to both elements of the pair.

The output type of a FlowArrowRef can be obtained in the type signature. Type class TakeOutputType collects output types of *arrows* from the type signatures(in SecureFlow.hs).

```
class (Lattice l, Arrow a) =>
      TakeOutputType l a b c where
  deriveSecType :: l -> (a b c) -> (SecType l)

instance (Lattice l, Arrow a,
          BuildSecType l c) =>
         TakeOutputType l a b c where
  deriveSecType l t = buildSecType l (undefined::c)
```

Type variable l is a lattice and variables b and c are the types of inputs and outputs of an *arrow*, a. Method deriveSecType takes a security label and an *arrow*, and return a security type for the output. In the instance of SecType, deriveSecType returns an output security type generated by method buildSecType, given an expected security label l and a undefined of type c.

4.6 References and Combinator lowerA

References have a different nature from other data types because the content of a reference can be shared by other reference identities. Combinator lowerA requires special treatments for references. In this section, we explain why references introduce problems and how they are resolved.

4.6.1 Filtering References

When a reference passing through the input filtering function of lowerA, it is not adequate to make the content undefined when its security types is higher than expected. Because the content of a reference can be shared by other reference identities, making it undefined will affect other reference identities as well. However, if we only filter a reference based on its identity security label, secret information may leak due to reference manipulation in combinator lowerA. Consider the following program: lowerA LOW readRefA. Assume the input has security type HIGH $\mathbf{ref}^{\mathtt{LOW}}$. The HIGH content is not filtered and read out in lowerA. Nevertheless, the output security type is LOW.

To resolve this problem, we define a reference type with a *read projection* function, which is called SRef. A read projection function makes a content undefined only when the content is read out. Other references pointing to the same content are not affected. The following is the definition of SRef.

```
data SRef a = SRef (IORef a) (a->a)
```

It parameterises on the type of its content and contains a IORef and a read projection function. When a reference is read, the read projection function is applied to the result read out from the reference. The following is the implementation of instance SRef in type class FilterData:

```
instance (Lattice l, FilterData l a) =>
         FilterData l (SRef a) where
  removeData l (SecRef t (Lab l')) (MkSRef c f) =
        if label_leq l' l
          then (MkSRef c ((removeData l t).f))
          else undefined
```

In removeData, a new projection function, based on an expected security label and a content security type, is composed with the original one. If the content has a security type higher than expected, it becomes undefined after passing through the read projection function.

On the other hand, when a reference passing out from combinator lowerA, the read projection function should be removed. Otherwise, the read projection function will continue to be effective and may make legal readRefA undefined later in the program. This is done by resetProject of type class ResetProject. It resets read projection function to id if a reference is passed to it.

4.6.2 Preserving Subtyping Invariants

The security types of references have a different sub-typing relation from other types. As described in Figure 4.3, the security type of a reference's content in the relation is invariant. However, in combinator lowerA, output security types are deduced based on output types and a given security label. The information is not enough to derive a content security type that obey the invariant relation. This may leak secret information. Assume two references r1 and r2 which refer to the same content and both have security type (SecRef (SecLabel LOW) LOW). If r1 is passed to computation lowerA HIGH (pure id), the output security type becomes (SecRef (SecLabel HIGH) HIGH). Secret information may leak by written to r1 and read out from r2. This is because combinator lowerA contains no information to preserve the subtyping invariant of references.

In FlowHaskellRef, we encode content security types in references' types so the content security types can be deduced from the types of references. To achieve this, different content security types have to be distinct types in the type system of Haskell. New types, called *singleton types*, are required for both lattice constructors and SecType constructors. We first extend SRef with a content security type(in RefOp.hs):

```
data SRef rs a = MkSRef rs (IORef a) (a->a)
```

Reference SRef parameterises on a content security type rs and a content type a. The content security type rs is constructed via *singleton types*. Therefore, SRef with different content security types are distinct types. Reference SRef contains its own content security type in singleton types as well as a IORef and a read projection function as before.

Singleton Types for Lattice

The programmers have to define new types for their lattice type. In this section, we use the same lattice as in Li and Zdancewic's paper [7]:

```
data TriLabel = LOW | MEDIUM | HIGH
```

Three new lattice singleton types are required to distinguish the three constructors, and they are as following:

```
data SLow    = VLow
data SMedium = VMedium
data SHigh   = VHigh
```

Type `SLow` corresponds to label `LOW`, while `SMedium` and `SHigh` corresponds to `MEDIUM` and `HIGH`, respectively. Constructor `VLow`, for instance, is used to construct type `SLow` for label `LOW`, and thus should be able to be transformed to label `LOW`. This is implemented in type class `STLabel`:

```
class STLabel rl l where
  toLabel :: rl -> l

instance STLabel SLow TriLabel where
  toLabel _ = LOW

instance STLabel SMedium TriLabel where
  toLabel _ = MEDIUM

instance STLabel SHigh TriLabel where
  toLabel _ = HIGH
```

Type class `STLabel` is a relation between a lattice label type, `rl`, and a lattice, `l`. It serves as a generic interface used in the singleton types of `SecType`, so programmers has to implement this class for their singleton types.

Singleton Types for `SecType`

The approach is similar for `SecType`. New types are defined for each constructor except `SecHigh`, because `SecHigh` is only used for output security type of combinator `pure`. The definitions are as following(in RLattice.hs):

```
data SSecLabel l        = VSecLabel l
data SSecRef s l        = VSecRef s l
data SSecPair s1 s2     = VSecPair s1 s2
data SSecEither s1 s2 l = VSecEither s1 s2 l
```

For the same reason, the transformation from these new types to `SecType` is provided in type class `STSecType`. The following is part of the definition(in RLattice.hs):

```
class STSecType rs l where
  toSecType :: rs -> SecType l

instance STLabel rl l => STSecType (RSecLabel rl) l where
  toSecType _ = SecLabel (Lab (undefined::rl))

instance (STSecType rs l, STLabel rl l)
         => STSecType (RSecRef rs rl) l where
  toSecType _ = SecRef (toSecType (undefined::rs))
                       (Lab (toLabel (undefined::rl)))
```

$$(CREATE)\frac{}{e(s_2^l), \{s_1^l \sqsubseteq s_2^l, \rho(\tau, \bot) \sim s_2^l\} \vdash \texttt{createRefA}\ (s_2^l)_v\ \ell\ :\ \tau \mid s_1^l\ \rightarrow\ \tau\ ref \mid s_2^l\ \mathbf{ref}^\ell}$$

$$(READ)\frac{}{\top, \emptyset \vdash \texttt{readRefA}\ :\ \tau\ ref \mid s^l\ \mathbf{ref}^\ell\ \rightarrow\ \tau \mid \uparrow (s^l, \ell)}$$

$$(WRITE)\frac{}{e(s^l), \{\ell \lhd s^l\} \vdash \texttt{writeRefA} : (\tau\ ref, \tau) \mid (s^l\ \mathbf{ref}^\ell, s^l)\ \rightarrow\ () \mid \bot}$$

Figure 4.17: Type system of reference primitives

Type class STSecType is defined between a new type, rs, and a lattice, l. This is important because sub-components of a SecType can only based on the same lattice type. The transformation is recursively defined so any type constructed from singleton types of SecType and singleton types of a lattice can be transformed to SecType. Notice how undefineds are used as arguments to define SecType. The constructors of singleton types are not required to define method toSecType. Only the type matters here.

With content security types encoded in reference types, combinator lowerA can derive content security types that preserve the subtyping invariant of references. The following shows how buildSecType is implemented for references.

```
instance (Lattice l, STSecType rs l) =>
            BuildSecType l (SRef rs a) where
  buildSecType l _ = (SecRef (toSecType (undefined::rs)) (Lab l))
```

The security type of a reference's content is generated from the type of SRef, in particular, the content security type rs. Since the value is never used in toSecType, we can simply assigned a undefined to it.

4.7 Reference Manipulation

Three standard operations for SRef are provided(in RefOp.hs).

```
newSRef :: a -> rs -> IO (SRef rs a)
newSRef a rs = do r <- newIORef a
                  return (MkSRef rs r)

readSRef :: SRef rs a -> IO a
readSRef (MkSRef rs r f) = do x <- readIORef r
                              return (f x)

writeSRef :: SRef rs a -> a -> IO ()
writeSRef (MkSRef rs r) a = writeIORef r a
```

A content security type is required when creating a reference, so the type of the reference contains information of its content security type.

Type class RefMonad specifies a common interface for any monads and corresponding references which can be used in FlowHaskellRef. The definition is as following(in RefOp.hs):

```
class RefMonad m r rs | m -> r where
  createMRef :: a -> rs -> m (r rs a)
  readMRef   :: (r rs a) -> m a
  writeMRef  :: (r rs a) -> a -> m ()
```

Type class `RefMonad` parameterises on a monad, `m`, a reference type in the monad, `r`, and a singleton type, `rs`. It provides standard operations for reference manipulation. Instance (`RefMonad IO SRef rs`) is implemented directly by the corresponding operations of `SRef`.

Three standard operations for reference manipulation, called `createRefA`, `readRefA`, and `writeRefA`, are implemented in FlowHaskellRef. They lift methods of `RefMonad` into `FlowArrowRef`. The implementation is based on the type system in Pottier and Simonet's work [12]. They are depicted as in Figure 4.17.

In reference creation, a reference with `LOW` content and `HIGH` identity is created as (`createRefA` (`VSecLabel VLow`) `HIGH`). The content security type, the first argument, is expressed as constructors of singleton types, denoted as $(s^l_2)_v$. However, programmers may specify wrong content security types, intentionally or unintentionally. Thus, two constraints are generated to guarantee the correctness of content security types. First, the security type specified by programmers should have a consistent structure w.r.t. the type of the input. For instance, if the input is a pair of integers, the top level security type constructor should be a `VSecPair`. This is expressed as $\rho(\tau, \perp) \sim s^l_2$. A security type derived from the input type is necessary because the input security type, s^l_1, could be **high**, which contains no structural information. Second, the security type specified by programmers is higher than or equal to the security type obtained in unification, which is the correct security type. This is expressed as $s^l_1 \sqsubseteq s^l_2$. See Sec. 5.1 for detail information about unification in FlowHaskellRef. In `readRefA`, the output security type is lifted by function $\uparrow$ according to the security label for a reference's identity, since the content is on longer protected by the reference identity. In `writeRefA`, the output is a unit, and thus only protected by $\perp$.

Chapter 5

Adding Unification Inside of FlowHaskellRef

5.1 Unification

Primitive `createRefA` takes a value and returns a reference, and the output security type , s^l $\mathbf{ref}^\ell$, contains two parts. Security type s^l is for the content and security label ℓ is for its own identity. The security type of the content should be identical or higher than the input security type of `createRefA` according to the type system in Figure 4.17. But how should the input security type be determined? In the framework of FlowHaskell, the only feasible way is to ask programmers to specify the security types. The following is an example using this approach:

```
pure (\_ -> 0) >>> createRefA (SecLabel HIGH) LOW >>>
readRefA (SecLabel HIGH) LOW >>> ...
```

Programmers are required to specify the security type of the input as well as the security label of the identity in `createRefA`. Similar in `readRefA`, both the security types for the content and the identity need to be specified. In many cases, however, the create and read of a reference may reside in very different parts of a program, but programmers have to remember and specify correct security types to make the program secure. This is not only tedious but error-prone. Consider a small error in `readRefA` as following:

```
pure (\_ -> 0) >>> createRefA (SecLabel HIGH) LOW >>>
readRefA (SecLabel LOW) LOW >>> ...
```

The `HIGH` content of the reference becomes `LOW` due to a mistake by specifying a wrong security type in `readRefA`. To avoid this kind of mistakes, unification is adopted in FlowHaskellRef to pass security types through computation and programmers do not have to worry about specifying wrong security types.

The main idea is to extend the possibility of passing security types in the `flow` field of `FlowArrowRef`. Unification variables can be used to define primitives and are aimed to receive security types from previous computations when combined by *arrow* combinators. To certify a computation, all the variables are unified first and the computation is certified as before.

$$\alpha ::= \omega \mid \ell$$

Figure 5.1: Extended lattice type

$$s^l \;::=\; \alpha \mid s^l\,\mathbf{ref}^\alpha \mid (s^l, s^l) \mid (\mathbf{either}\; s^l\; s^l)^\alpha \mid \mathbf{high} \mid t$$

Figure 5.2: Extended security type

5.1.1 Unification Types

Two kinds of unification variables are required. One can be unified with a security label and the other is used to unify with a security type. In Figure 5.1 a new constructor, variable ω, is extended in the original lattice type. It is implemented by a new data type `Label` as following (in Lattice.hs):

```
data Label l = Lab l | LVar String
```

To unify with a security type, s^l has a new constructor, t, for unification variables, as in Figure 5.2. Data type `SecType` is modified to reflect the new design(in Lattice.hs).

```
data SecType l = SecLabel (Label l)
               | SecRef (SecType l) (Label l)
               | SecVar String
               | ...
```

Constructors not shown above are the same as before.

The unification semantics for security labels is depicted in Figure 5.3. When two security labels unify with each other, they should be identical. If one is a variable, a new unification result is generated. Figure 5.4 shows the unification semantics for s^l. Symbol $\overset{U}{\sim}$ denotes a unification that produces some unification results, U. A unification result along with a security type, Us^l, updates the security type, s^l, according to the unification result, U. If more than one unification result are produced, the final unification result is the composition of all results in the same order as they are generated. Notice that there is no rule to unify a $\mathbf{high}$ and a ($s^l\;\mathbf{ref}^\alpha$). The reason is the security label of the content of a reference is invariant in the sub-typing relation in Figure 4.3. Unifying with $\mathbf{high}$ may break the invariant property.

Many operators takes security types and are required to generate constraints in `FlowArrowRef`, but they can only be defined for security types which contain no unification variables. Therefore, those operations should be retained and performed until all variables inside of the involving security types are resolved. A new type called unification constraint, as in Figure 5.5, represents those operators and postpones the operations until all variables are unified to some security types. Those constraints are implemented by data type `U`(in Unification.hs):

```
data U s = Meet (U s) (U s)
         | Join (U s) (U s)
         | LExtr (U s)
         | Fst (U s)
         | Snd (U s)
         | Tag (U s) (U s)
         | Decl (U s) (U s) (U s)
         | Id s
```

$$\ell \overset{\emptyset}{\sim} \ell \qquad \omega \overset{(\omega:=\alpha)}{\sim} \alpha$$

Figure 5.3: Unification of lattice

$$\frac{\alpha_1 \overset{U}{\sim} \alpha_2}{\alpha_1 \overset{U}{\sim} \alpha_2} \qquad \frac{\alpha_1 \overset{U_1}{\sim} \alpha_2 \quad U_1 s_1^l \overset{U_2}{\sim} U_1 s_2^l}{s_1^l\ \mathbf{ref}^{\alpha_1} \overset{U_2.U_1}{\sim} s_2^l\ \mathbf{ref}^{\alpha_2}} \qquad \frac{s_1^l \overset{U_1}{\sim} s_3^l \quad U_1 s_2^l \overset{U_2}{\sim} U_1 s_4^l}{(s_1^l, s_2^l) \overset{U_2.U_1}{\sim} (s_3^l, s_4^l)}$$

$$\frac{\alpha_1 \overset{U_1}{\sim} \alpha_2 \quad U_1 s_1^l \overset{U_2}{\sim} U_1 s_3^l \quad U_2 U_1 s_2^l \overset{U_3}{\sim} U_2 U_1 s_4^l}{(\mathbf{either}\ s_1^l\ s_2^l)^{\alpha_1} \overset{U_3.U_2.U_1}{\sim} (\mathbf{either}\ s_3^l\ s_4^l)^{\alpha_2}} \qquad \frac{}{\mathbf{high} \overset{\emptyset}{\sim} \mathbf{high}}$$

$$\frac{t \notin \mathtt{FV}(s^l)}{t \overset{t:=s^l}{\sim} s^l} \qquad \frac{}{\mathbf{high} \overset{\emptyset}{\sim} \top} \qquad \frac{}{\mathbf{high} \overset{\omega:=\top}{\sim} \omega}$$

$$\frac{\mathbf{high} \overset{U_1}{\sim} s_1^l \quad U_1 \mathbf{high} \overset{U_2}{\sim} U_1 s_2^l}{\mathbf{high} \overset{U_2.U_1}{\sim} (s_1^l, s_2^l)} \qquad \frac{\mathbf{high} \overset{U_1}{\sim} \alpha \quad \mathbf{high} \overset{U_2}{\sim} U_1 s_1^l \quad U_2 U_1 \mathbf{high} \overset{U_3}{\sim} U_2 U_1 s_2^l}{\mathbf{high} \overset{U_3.U_2.U_1}{\sim} (\mathbf{either}\ s_1^l\ s_2^l)^{\alpha}}$$

Figure 5.4: Unification of security type

Type variable `s` is instantiated with `SecType` ℓ in our case, where ℓ is a lattice. Constraints $\sqcap$ and $\sqcup$ correspond to the `Meet` and the `Join` constructors. The operations are performed by method `min` and `max` of `SecType` respectively. Constraint **extr** is implemented by `LExtr` and resolved by function e defined in Figure 4.11. Constraint **fst** and **snd** returns the first and the second element of a pair respectively. Constraint $\uparrow$, resovled by function $\uparrow$ in Figure 4.12, is implemented by constructor `Tag`. Constraint $\downarrow$ corresponds to constructor `Decl` and is resolved by function $\downarrow$ in Figure 4.13. The last constraint is just a `SecType` and implemented by constructor `Id`. Besides, any `Label` is wrapped in a `SecLabel` constructor so both data type `SecType` and data type `Label` can share the same constraint type. For example, a label `Lab l` in constraint `U` becomes `Id (SecLabel (Lab l))`.

The unification semantics for unification constraint is depicted in Figure 5.6. When two constraints are both `SecType`s, they are unified according to the unification semantics of s^l. The second rule says if a constraint contains no variable, the corresponding operation is performed before unification, denoted by symbol $[\![]\!]$. Otherwise, the constraint is put back to the unification set and try again later.

5.1.2 Unification in FlowArrowRef

To extend unification in FlowHaskellRef, new fields to handle unification variables are required. The following is the complete definition of `FlowArrowRef`(in FlowArrowRef.hs):

```
data FlowArrowRef l a b c =
    FARef { computation :: ((SecType l) -> (SecType l))
                           -> a b c
```

$$c^l ::= c^l \sqcap c^l \mid c^l \sqcup c^l \mid \mathbf{extr}\ c^l \mid \mathbf{fst}\ c^l \mid \mathbf{snd}\ c^l \mid \uparrow l\ c^l \mid \downarrow l\ c^l \mid s^l$$

Figure 5.5: Unification constraint type

$$\frac{s_1^l \overset{U}{\sim} s_2^l}{\mathtt{Id}\ s_1^l \overset{U}{\sim} \mathtt{Id}\ s_2^l} \qquad \frac{[\![c^l]\!] \overset{U}{\sim} \beta}{c^l \overset{U}{\sim} \beta}$$

Figure 5.6: Unification constraint elimination

```
, flow        :: Flow (SecType l)
, constraints :: [Constraint (SecType l)]
, pc          :: (SecType l)
, uniset      :: [(U (SecType l), U (SecType l))]
}
```

Field `computation` becomes a function that accepts a substitution function and returns an underlying *arrow* computation. A substitution function takes a `SecType` and returns a new `SecType` with some internal variables replaced with other `SecTypes`. The reason of introducing a substitution function is explained in Sec. 5.2. A new field, `uniset`, contains a list of pairs of constraint types which are aimed to be unified. Field `pc` becomes `SecType` rather than a lattice type. This is due to implementation issues that security types in field `uniset` should have a uniform type. The interpretation of `pc` remains the same.

When defining new primitives with unification, variables normally appear in the input security type because it is the only place to receive information from previous computations. Those variables can also be used to define constraints within the same `FlowArrowRef`. But when combining two `FlowArrowRef` computations via *arrow* combinators, it is possible that two computations have variables with identical names but are actually distinct. Treating those variables the same are likely to become inconsistent and cannot be resolved to a single value in unification algorithm. As a result, every variable is renamed to a distinct name in *arrow* combinators. To collect all variables in a `FlowArrowRef` computation, it is sufficient to collect variables in the field `flow` and `uniset`. Variables must appear either in field `flow` or field `uniset` so that they can be unified to a value afterward. After all variables are collected, a substitution containing a list of old name and new name pairs are generated. Then, variables in all fields are replaced with their new names according to the substitution.

Since security types are passed through field `flow` by unification variables, combinators need to compute new `flows` that pass security types as well as fulfill the type system described in Sec. 4.3.2. In the rest of this section, we explain how each combinator produces flow security types with the presence of unification variables, and show that they still follow the type system.

Combinator `pure` defines its `flow` to be ($t \ \rightarrow \ \mathbf{high}$) which implements the type system directly(in FlowArrowRef.hs).

```
pure f = FA { ...
              flow = Trans (SecVar "x0") SecHigh
              ... }
```

The `flow` of combinator (>>>) is defined by the following function(in FlowArrowRef.hs):

```
flow_seq :: Lattice l =>
            Flow (SecType l) -> Flow (SecType l)
            -> ( Flow (SecType l)
               , [Constraint (SecType l)]
               , [(U (SecType l),U (SecType l))])
flow_seq (Trans s1 s2) (Trans s3 s4)= (Trans s1 s4, c,u)
  where
  (c,u) = seqFlow s2 s3

seqFlow s2 s3 =
  case (s2,s3) of
    (t1@(SecVar _), t2) -> if (hasVarSecType t2)
                             then ([],[(Id t1,Id t2)])
                             else ([LEQ t1 t2], [])
    ...
    (SecPair t1 r1, SecPair t2 r2) -> let (ct,ut) = seqFlow t1 t2
                                          (cr,ur) = seqFlow r1 r2
                                      in (ct++cr,ut++ur)
    ...
```

Given input and output security types of two `FlowArrowRef`s, function `flow_seq` returns a new `flow` as `Trans s1 s4`. Function `seqFlow` takes two security types, `s2` and `s3`, and generates additional constraints and unification pairs. As in the first case pattern matching of `seqFlow`, if `s2` is a variable and `s3` contains no variable, a new constraint `s2` $\sqsubseteq$ `s3` is generated. This is the same as in the type system. On the other hand, if `s3` contains variables, it is aimed to receive information from `s2`, so `s3` should be unified with `s2`. This means `s2` $=$ `s3` and the constraint `s2` $\sqsubseteq$ `s3` is satisfied implicitly. If `s2` and `s3` contains structure information, the sub-components of them are again computed by `seqFlow`. For example, if both `s2` and `s3` are `SecPair`, the first and the second components of them are compared respectively. This fine-grained comparison still satisfies the constraint `s2` $\sqsubseteq$ `s3` in the type system.

For combinators `first`, `second`, and (***), the new flow security type is implemented by function `flow_pair`(in FlowArrowRef.hs).

```
flow_pair :: Lattice l => Flow (SecType l)
             -> Flow (SecType l) -> Flow (SecType l)
flow_pair (Trans s1 s2) (Trans s3 s4) =
                Trans (SecPair s1 s3) (SecPair s2 s4)
```

It simply puts input and output security types in pair constructors respectively. For combinator (&&&), more analyses are required in `flow_diverge`(in FlowArrowRef.hs).

```
flow_diverge :: Lattice l =>
     Flow (SecType l) -> Flow (SecType l)
     -> ( Flow (SecType l)
        , [Constraint (SecType l)]
        , [(U (SecType l), U (SecType l))])
flow_diverge (Trans s1 s2) (Trans s1' s2') =
  let (in_flow, cons, us) = meetInFlow s1 s1' in
  (Trans in_flow out_flow, cons, us)
  where
  out_flow = (Pair s2 s2')

meetInFlow s1 s2 =
  case (s1,s2) of
    (t1@(SecVar _), t2@(SecVar _)) -> (t1, [], [(Id t1,Id t2)])
    (t1@(SecVar _), t2) -> if hasVarSecType t2
                             then (t1, [], [(Id t1,Id t2)])
                             else (t1, [LEQ t1 t2], [])
    ...
    (SecLabel l1, SecLabel l2) ->
                let (l' , cons, us) = meetInFlowLabel l1 l2
                in (SecLabel l', cons, us)
    (Pair s1 t1, Pair s2 t2) ->
                let (s', scons, sus) = meetInFlow s1 s2
                    (t', tcons, tus) = meetInFlow t1 t2
                in (Pair s' t', scons++tcons, sus++tus)
    ...

meetInFlowLabel k1 k2 =
 case (k1,k2) of
  (l1@(LVar _), l2@(LVar _)) ->
                 (l1, [], [(Id (SecLabel l1),Id (SecLabel l2))])
  (l1@(LVar _), l2) -> (l1,[LEQ (SecLabel l1) (SecLabel l2)],[])
  (l1, l2@(LVar _)) -> (l2,[LEQ (SecLabel l2) (SecLabel l1)],[])
  (Lab l1, Lab l2) -> (Lab (label_meet l1 l2), [], [])
```

According to the type system, the new input security type is the meet of the two original input security types. Function `meetInFlow` traverses and compares two original input security types simultaneously, and produces a security type that approximates but still respects the type system. If two variables are compared, since they both expect to receive the security types from previous computation, they should be identical. Thus, any of them can represent new input security type and a unification pair is generated to ensure that they are the same. In the case that only one input security types is a variable, if the other security type contains no variable, the variable is regarded as new input security type. A constraint `LEQ` is required to guarantee that the resulting security type is the meet of the two input security types. However, if the other input security type contains variables, an approximation is made to force the two security types are the same. The second pattern matching of the case statement in function `meetInFlow` implements the idea described above. A constraint (`LEQ t1 t2`) is generated in the case that `t2` contains no variable. In the other case, a unification pair of `t1` and `t2` is generated to ensure that they are identical. This approximation rejects some legal programs. One way to avoid this is to provide unification constraints to extract sub-components of each security type constructors, such as `Fst` and `Snd` for pairs. Thus, the meet of the two input security types can be precisely produced after all variables are unified. This is one of the future work of FlowHaskellRef. In the case that both of input security types contain no variable, function `meetInFlow` and function `meetInFlowLabel` is applied to the sub-components of the security types. Function `meetInFlowLabel` is similar to function `meetInFlow` but compares two security labels.

Function `flow_either` defines flow for `left`, `right`, and (+++)(in FlowArrowRef.hs).

```
flow_either (Trans s1 s2) (Trans s3 s4) =
               Trans (SecEither s1 s3 (LVar "x0"))
                     (SecEither s2 s4 (LVar "x0"))
```

A security label for the identity of `SecEither` is received from previous computations and passed as the identity security label of output `SecEither`. It implements the type system directly.

Function `flow_converge` returns flow of (|||) and a list of unification pairs(in FlowArrowRef.hs):

```
flow_converge :: Lattice l =>
                 Flow (SecType l) -> Flow (SecType l)
                 -> ( Flow (SecType l)
                    , [(U (SecType l),U (SecType l))])
flow_converge (Trans s1 s2) (Trans s1' s2') =
  let l = (LVar "x0") in
  let s_out = (SecVar "x1") in
  (Trans (SecEither s1 s1' l) s_out
  ,[(Id s_out, Tag (Join (Id s2) (Id s2')) (Id (SecLabel l)))])
```

The identity security label of input, `l`, is a variable to receive a security label from previous computations. The resulting output security type, `s_out`, is the result of applying function $\uparrow$ to the join of two original output security types, `s2` and `s2'`. The type system is implemented directly.

Except field `flow`, other fields of `FlowArrowRef` implement the type system in Sec. 4.3.2 as before, although the enforcement of the type system is postponed until unification variables are unified successfully.

5.1.3 Defining New Primitives

New primitives of `FlowArrowRef` may take advantages of unification. Unification variables provide a possibility of receiving information from other computations. However, there are two rules which should be strictly obeyed to guarantee success of unification algorithm.

1. Names of new variables cannot begin with a character 'a' and followed by an integer.
2. New variables in a primitive must appear either in input security types or unification set.

First rule comes from the fact that variables are renamed to the internal format used by fresh name generating functions. A new variable may have a name collision with existing ones if not named properly. If a variable violates the second rule, the variable cannot be resolved to a security type and the unification algorithm will fail.

5.2 Implementation of `lowerA`

With unification added to `FlowArrowRef`, we are now ready to explain the implementation of `lowerA`(in FlowArrowRef.hs):

```
lowerA :: (Lattice l, Arrow ar, BuildSecType l b,
           FilterData l a, ResetProject b,
           TakeOutputType l (FlowArrowRef l ar) a b)
          => l -> FlowArrowRef l ar a b -> FlowArrowRef l ar a b
lowerA level fa@(FARef com' (Trans s_in' _) cons' pc' uniset') =
   let flow_out = (deriveSecType level fa) in
   let inputFilter upd = pure (\i -> (removeData level (upd s_in') i)) in
   let removeRead = pure (\i -> resetProject i) in
   FARef { computation = (\upd -> (inputFilter upd) >>> (com' upd)
                                  >>> removeRead)
         , flow = Trans s_in' flow_out
         , constraints = [LEQ (SecLabel (Lab label_top)) pc'] ++ cons'
         , pc = (SecLabel (Lab label_top))
         , uniset = uniset'
         }
```

Combinator `lowerA` takes a lattice label, `level`, and a `FlowArrowRef`, `fa`. The new `pc` is $\top$ and a new constraint requires `pc'` to be $\top$ as well, which means no low side effect is permitted in lowerA.

Method `deriveSecType` is applied to the lattice label `level` and computation `fa`, and returns a derived output security label `flow_out`. Function `inputFilter` defines the input filter function via `pure` of underlying *arrow* computation, and it filters the inputs by sequenced before the underlying computation `com`$'$. Input security type `s_in`$'$ in `inputFilter` may be a variable and can only be resolved to a value after unification. Moreover, during the construction of a `FlowArrowRef` program, `s_in`$'$ may be renamed several times in combinators of `FlowArrowRef`. Thus, a substitution function, called `upd`, is taken in function `inputFilter`. A substitution function takes a security type and returns a new security type which is similar to the input but with some variables inside replaced by other security types or variables. Function `upd` is applied to input security type `s_in`$'$ and the result is a security type without variables. Because the substitution function is known only from outside a computation, the definition of field `computation` is adapted to take a substitution function and returns an underlying *arrow* computation, as described in Sec. 5.1.2. Finally, underlying *arrow* `removeRead` resets the read projection function of a reference to `id`.

The substitution function is constructed in *arrow* combinators. The following is an example of combinator (>>>)(in FlowArrowRef.hs):

```
a1@(FA c1 f1 t1 g1 pc1 u1) >>> a2@(FA c2 f2 t2 g2 pc2 u2) =
      ...
      let (sub1, sub2, _) = (make_sub a1 a2 []) in
      let (f,c,u) = flow_seq (replace_flow sub1 f1)
                             (replace_flow sub2 f2) in
      FARef{
      computation =
           (\upd -> c1 (upd.(replaceSecType sub1)) >>>
                    c2 (upd.(replaceSecType sub2)))
      ...
      }

replaceSecType :: Lattice l =>
                  [(SecType l, SecType l)]
                  -> (SecType l -> SecType l)
```

Function `make_sub` collects all variables in `FlowArrowRef` `a1` and `a2`, and returns substitutions `sub1` and `sub2`. A substitution is a list of old variable name and new variable name pair. Function `replaceSecType` takes a substitution and becomes a substitution function defined in the paragraph above. In field `computation`, a new substitution function `replaceSecType sub1`, for instance, is composed with the substitution function passed from outside. By this substitution function composition, a variable inside a computation performs all renaming in the same order as other variables in other fields. At top level, a substitution function that substitute variables according to the unification result is passed. It substitutes a variable to a security type containing no variables.

Chapter 6

Case Study of FlowHaskellRef

In the case study, a cryptographic protocol between a client card, an ATM, and a bank is implemented to show that FlowHaskellRef can be used to build bigger applications. The case study is revised from Tse and Washburn's work [20], which is a case study for Jif [9].

6.1 Cryptographic Protocol

The scenario of the case study is as following. A bank card is inserted in an ATM machine and two messages are exchanged between the ATM machine and a bank server for authentication. If the authentication succeeds, another two messages are exchanged for transaction. In real situation, messages are delivered over public networks. To guarantee confidentiality, they are encrypted before sent. In the case study, we simulate the public networks as IO monad for simplicity and each message passing in the IO monad is public and should be encrypted. Besides, only the protocol involving security data is implemented. Key generation and a database in the bank server, which appear in Tse and Washburn's work [20], are skipped.

6.1.1 Protocols

The protocol contains the following four messages.

1. **Authentication request** The message is from an ATM to a bank server and contains two data. One is the account number read from a client card, $\mathtt{id}_c$, and the other is a nonce generated by the ATM, $\mathtt{n}_a$. The message is encrypted with the public key of the bank, $\mathtt{k}_b$, and as following:

$$\{\ \mathtt{id}_c,\ \mathtt{n}_a\ \}_{\mathtt{k}_b}$$

2. **Authentication response** When a authentication request is received by a bank server, it is decrypted with the private key of the bank. A session key, $\mathtt{s}$, and a bank nonce, $\mathtt{n}_b$, are generated by the bank server. Along with the ATM nonce received in the authentication request, the authentication response is encrypted

with the public key of the client, $\mathtt{k}_c$, and sent back to the ATM. The message is as following:

$$\{\, \mathtt{s},\ \mathtt{n}_a,\ \mathtt{n}_b \}_{\mathtt{k}_c}$$

3. **Transaction request** The ATM retrieves the private key of the client, $\mathtt{p}_c$, with a correct password, and it is used to decrypt the authentication response. The ATM nonce in the decrypted message is verified to be the same as the one sent in the authentication request. A transaction, `tran`, contains the account number, type of action(deposit or withdraw, `act`), amount(`mnt`), and a new ATM nonce($\mathtt{n}_a$). A signature, `sig`, is obtained by signing `tran` with the private key of the client. Together with $\mathtt{n}_b$ received in the authentication response and the new ATM nonce $\mathtt{n}_a$, the transaction request is encrypted with the session key also received in the authentication response and sent to the bank server.

$$\begin{aligned} &\mathtt{tran} = (\, \mathtt{id}_c,\ \mathtt{act},\ \mathtt{mnt},\ \mathtt{n}_a) \\ &\mathtt{sig} = \{\, \mathtt{tran} \,\}_{\mathtt{p}_c} \\ &\{\, \mathtt{tran},\ \mathtt{sig},\ \mathtt{n}_a,\ \mathtt{n}_b \,\}_{\mathtt{s}} \end{aligned}$$

4. **Transaction response** Transaction request is decrypted with the session key. The bank nonce is verified to be the same as the one sent in the authentication response. Then a new bank nonce, $\mathtt{n}_b$, is generated. Along with the ATM nonce received in the request and a response(transaction complete or failure, `res`), the transaction response is encrypted with the same session key and sent to the ATM. The message is as following:

$$\{\, \mathtt{res},\ \mathtt{n}_a,\ \mathtt{n}_b \,\}_{\mathtt{s}}$$

6.1.2 Cryptographic Library

The Haskell cryptographic library is adopted to encrypt and decrypt messages described in the previous section. Cryptosystem RSA is chosen for public-key encryption and signature signing, and the encryption and decryption primitives are in module `Codec.Encryption.RSA`. Advanced Encryption Standard(AES) is used for symmetric-key encryption. The required primitives are in module `Codec.Encryption.AES`.

6.2 Implementation in FlowHaskellRef

In the bank system simulation, a bank lattice is developed to represent different security levels of data in the program. A set of trusted computing base is identified, and then we show how information flow policies are enforced to achieve the security goals of the simulation. The rest of the section explains each topic in details.

6.2.1 Bank Lattice

A data type `BankLabel` is defined in the case study and its elements form a lattice illustrated in Figure 6.1. Label `CLIENT` classifies data that belong to a client card, e.g. the private key of the client. Label `BANK` classifies data that belong to the ATM and the bank server. For instance, session keys used in AES cryptosystem are protected by `BANK`. Label `ATM_BANK` classifies data that are generated in the ATM or sent from the ATM to the bank server. Label `BANK_ATM`, in contrast, classifies data that are generated

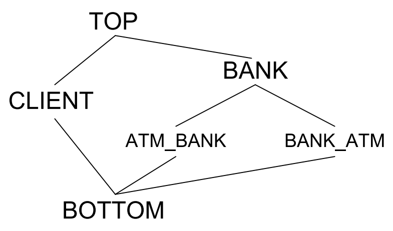

Figure 6.1: Bank lattice

in the bank server or sent from the bank server to the ATM. For example, the ATM nonce generated in the ATM is protected by `ATM_BANK`, while the ATM nonce received from the bank server in the ATM is protected by `BANK_ATM`. All data that are released to IO monad are public and thus protected by label `BOTTOM`. Label `TOP` is introduced to complete the lattice.

6.2.2 Trusted Computing Base

In the bank system simulation, secure programs constructed in FlowHaskellRef have type `Protected c`. The definition of `Protected` is as following:

```
type Protected c = FlowArrowRef BankLabel ArrowRef () c
```

Each secure program receives a unit and returns a value of type `c`. Data type `ArrowRef` is an *arrow* that supports computation in the IO monad.

Function `main` simulates network as the IO monad for message passing and is as following(in BankTCB.hs):

```
main =
  do -- Declare reference and protect them
     atm_ATMNonce    <- getNewRef ATM_BANK (1::Int) (VSecLabel VAtm_bank)
     atm_BankNonce   <- getNewRef BANK_ATM (2::Int) (VSecLabel VBank_atm)
     atm_SessionKey  <- getNewRef BANK (3::Word128) (VSecLabel VBank)
     bank_ATMNonce   <- getNewRef ATM_BANK (4::Int) (VSecLabel VAtm_bank)
     bank_BankNonce  <- getNewRef BANK_ATM (5::Int) (VSecLabel VBank_atm)
     bank_SessionKey <- getNewRef BANK bankSeKey (VSecLabel VBank)
     -- Protocol begins
     auth_req <- atm_authRequest atm_ATMNonce (PR BANK)
     auth_res <- bank_authResponse auth_req bank_ATMNonce
                   bank_BankNonce bank_SessionKey (PR BANK)
     tran_req <- atm_tranRequest auth_res atm_ATMNonce
                   atm_BankNonce atm_SessionKey
                   (client_getPriKey (PR CLIENT)) (PR BANK)
     tran_res <- bank_tranResponse tran_req bank_ATMNonce
                   bank_BankNonce bank_SessionKey (PR BANK)
     atm_resultProcess tran_res
     return ()
  where
  bankSeKey = (0x06a9214036b8a15b512e03d534120006::Word128)
  getNewRef l init rs =
    do ref <- createMRef init rs
       let pref = tagRef l >>> lowerA l (pure (\() -> ref))
       return pref
```

References are used to store all sorts of data, such as keys, ATM nonces, and bank nonces. In the first part of `main`, they are created and protected by appropriate labels. Labels `VAtm_bank`, `VBank_atm`, and `VBank` are belongs to singleton types `SAtm_bank`, `SBank_atm`, and `SBank`, respectively. This part belongs to TCB. In the second part of `main`, four messages are passed in the IO monad. For each procedure, required data references and a privilege are given. The dispatch of references and privileges belongs to TCB. Function `main` is separated in module `BankTCB` because the constructor of privilege, `PR`, is only in scope in this module. As a result, the programs built in other modules can only declassify data with the privilege given in the TCB.

To run a FlowHaskellRef program in the IO monad, function `expose` must be used to release the program from type `Protected` to type `IO`. The definition is as following(in BankSystem.hs):

```
expose :: (Privilege BankLabel) -> SecType BankLabel
         -> Protected c -> IO c
expose p r t = runArrowRef (certifyRef (SecLabel (Lab label_bottom))
                                       (ml_bottom r) p t) ()
```

Function `expose` certifies a program with an input security type $\perp$ and an output security type having the same structure as `r` but with every security label being $\perp$.

6.2.3 Security Assumptions

The following are the security assumptions of the bank system simulation.

- Data in the IO monad of function `main` are public and cannot be trusted.
- The client's password is well-protected and can not be obtained by unauthorized people.
- RSA and AES cryptosystems are unbreakable within tolerable computation power and time.
- Programs mentioned in Sec. 6.2.2 is trusted.

6.2.4 Confidentiality of Bank System

Function `expose` must be used to certify a FlowHaskellRef program before executing and releasing the result in the IO monad. It expects the output security type with all security labels being $\perp$, which means the result is public. This satisfies the first security assumption that the IO monad cannot be trusted.

There are twelve `declassifyRef` statements appearing in the program. The following justifies that each use of `declassifyRef` is adequate. They are organized by the functions using them.

- Function **encryptRSA**(in BankSystem.hs)
  The data declassified to $\perp$ is encrypted by RSA cryptosystem. According to the security assumptions, the data is secure by itself.

  ```
  encryptRSA l_owner p_owner req key =
    let enreq = req >>> tagRef l_owner >>>
                lowerA l_owner
                  (pure (\plain -> RSA.encrypt key plain))
                >>> declassifyRef BOTTOM
    ...
  ```

- Function **client_getPriKey**(in BankTCB.hs)
  The ownership of the private key is transferred from CLIENT to BANK if a correct password is provided. According to the security assumptions, the password is secure, so the declassification is proper.

```
ient_getPriKey pr p =
  if p /= clientPW
    then return Nothing
    else let key = clientPriKey >>>
                   declassifyRef BOTTOM in
         do pri <- expose pr (SecLabel (Lab BOTTOM)) key
            return $ Just (tagRef BANK >>>
             lowerA BANK (pure (\() -> (clientModulus,pri))))
```

- Function **atm_AuthResDecrypt**(in BankSystem.hs)
  There are two declassification statements in the function. The message received from the bank server is decrypted with the private key of the client. The private key pk is protected by BANK, so the first declassification is required to downgrade the decrypted data to BANK_ATM, a proper label for data received from the bank server. In the second declassification, the result of verifying the equality of two ATM nonce's is made public. This is the necessary one-bit information for the procedure to continue or to stop and reveals no other information, so the declassification is adequate.

```
...
let dres =
    pk >>>
    lowerA BANK (pure (\key -> RSA.decrypt key res)) >>>
    declassifyRef BANK_ATM >>>
    ...
    lowerA BANK (pure (\(a,b) -> if a == b then True else False))
    ...
    >>> declassifyRef BOTTOM
...
```

- Function **atm_getSignature**(in BankSystem.hs)
  The password of the client, ppw, is protected by BANK. Declassification is required to retrieve the private key of the client to sign the signature. The password is public only inside this function and is not returned. Therefore, the declassification is proper.

```
atm_getSignature tran client_prikey ppw priv =
  do let password = ppw >>> declassifyRef BOTTOM
     pw <- expose priv (SecLabel (Lab BOTTOM)) password
     cpri <- client_prikey pw
     ...
```

- Function **atm_buildTranRequest**(in BankSystem.hs)
  The declassified data is encrypted by AES cryptosystem and is secure according to the security assumptions. Thus, the declassification is adequate.

```
...
lowerA BANK (pure (\(c,key) ->
  cbc AES.encrypt sessionIV key (pkcs5 c) )) >>>
declassifyRef BOTTOM
...
```

- Function **bank_tranDecrypt**(in BankSystem.hs)
  There are two declassification statements in the function. The first one declassifies data that are decrypted with the session key to `ATM_BANK`. The justification is the same as the first declassification in function **atm_AuthResDecrypt**. The second declassification declassifies the content of the message but they are protected separately right away in the `return` statement. Therefore, the declassification is adequate.

```
...
let
dtran = sessionKey >>> readRefA >>>
        lowerA BANK (pure (\key ->
         unPkcs5 (unCbc AES.decrypt sessionIV key tran) ))
        >>> declassifyRef ATM_BANK >>>
        ...
        >>> declassifyRef BOTTOM
(trans,(sig,(atm_n,bank_n))) <- exposeL priv dtran
return (protect trans, protect sig,
        protect atm_n, protect bank_n)
where
protect t = tagRef ATM_BANK >>>
            lowerA ATM_BANK (pure (\( -> t))
```

- Function **bank_checkBankNounce**(in BankSystem.hs)
  One-bit information about the bank nonces' equality is released. This piece of information is necessary for the procedure to continue or to stop.

```
...
lowerA BANK (pure (\(x,y) -> if x == y then True else False))
>>> declassifyRef BOTTOM
...
```

- Function **bank_buildTranResponse**(in BankSystem.hs)
  The declassified data is encrypted by AES cryptosystem. According to the security assumptions, the declassification is adequate.

```
...
lowerA BANK (pure (\(dat,key)->
  cbc AES.encrypt sessionIV key $ pkcs5 dat))
>>> declassifyRef BOTTOM
...
```

- Function **bank_RSAdecrypt**(in BankSystem.hs)
  There are two declassification statements in the function. The first one declassifies data decrypted with the private key of the bank. The justification is the same in the first declassification in function **atm_AuthResDecrypt**. The second one declassifies the account number of the client, which is a public data.

```
...
bankPriKey >>> tagRef BANK >>>
lowerA BANK (pure (\prikey ->
     RSA.decrypt (bankModulus, prikey) req)) >>>
declassifyRef ATM_BANK >>>
...
-- The input of fstPair is (account number,())
>>> fstPair
>>> declassifyRef BOTTOM
...
```

Function expose satisfies the first security assumption. All declassification statements are adequate as described above, based on other security assumptions. Therefore, with the assumption that programs written in FlowHaskellRef obey information-flow policies, the bank system simulation preserves confidentiality.

6.3 Evaluation of Bank System Simulation

6.3.1 Examples of Malicious Programs

In this section, we demonstrates two malicious programs trying to violate the information-flow policies and how they are rejected by FlowHaskellRef.

- **Local comparison of nonces**
  In the protocol, local nonces and remote nonces are compared to ensure the messages are in the same session. For instance, the bank server compares the bank nonce generated by itself with the bank nonce received from the ATM in the transaction response. If a local comparison of nonces happened, secret data may be delivered to a fake ATM or a fake bank server which is set by an attacker. The one generated in the bank server is protected by BANK_ATM, while the one from the ATM is protected by ATM_BANK. If a mistake is made to compare the same nonce, as following code:

  ```
  (bank_n &&& bank_n) >>> checkEqualNonce
  ```

  Value bank_n is the local bank nonce. The program is rejected by function checkEqualNonce as following:

  ```
  ((tagRef ATM_BANK) *** (tagRef BANK_ATM)) >>>
  lowerA BANK (pure (\(x,y) -> if x == y then True else False))
  ```

  The tagRef requires two nonces protected by ATM_BANK and BANK_ATM, respectively.

- **Using unauthorized private keys** The private key of the client is protected by label CLIENT. If a malicious program trying to use it directly without providing correct password, the program is rejected. The following is an example of using unauthorized private keys:

  ```
  fake_bank_RSAdecrypt :: [Octet] -> Protected (SecRef Int)
                          -> (Privilege BankLabel) -> IO Int
  fake_bank_RSAdecrypt req atm_nonce priv =
   do let dereq = clientPriKey >>>
                  ... >>>
                  declassifyRef BOTTOM
      ...
  ```

  The output type of this function is in the IO monad, so declassifyRef and expose has to be used to release the result. However, the privilege priv is passing from TCB and is (PR BANK) is this case. Using unauthorized private key clientPriKey makes the security type become CLIENT and cannot be declassified with the privilege.

6.3.2 Evaluation of FlowHaskellRef

The case study shows FlowHaskellRef can be used to develop non-trivial applications. Programming interface supported by FlowHaskellRef is sufficient to build interesting programs. Combinator `lowerA` is proved to be useful and convenient to alleviate the restriction of combinator `pure`. The total size of the code is around 350 lines without counting data like public and private keys. This is relative small comparing to the Jif implementation [20] which is around 800 lines of code, although not all functions are implemented. To the best of our knowledge, this case study is the biggest secure functional program so far.

On the other hand, there are some limitations observed in the case study. In the case study, a lattice is developed to classify security levels of data. However, some redundant labels may be required to make the lattice complete. For instance, label `TOP` is only introduced to make lattice `BankLabel` have a global top element. This is due to the definition of lattice that every two elements should have a unique meet and a unique join. In real applications, some security label of data may not exhibit a join relation. For example, label `CLIENT` and label `BANK` should be the top elements in their own hierarchy.

In FlowHaskellRef, a program is certified at run-time. In the case study, several sub-programs are developed in FlowHaskellRef. If one of them is problematic, the whole program is stopped after certifying that sub-program. Therefore, to check all sub-programs are correct, every sub-program must be run at least once. This is not convenient if the application is huge or some sub-programs are executed only in some branches.

The case study contains twelve declassification statements, which is much more comparing to the four times in the Jif case study. More examination is required to make sure all declassification statements are adequate. The session keys and private keys in the Jif case study are not protected by some labels. They rely on run-time equality check of the principal that manipulates those keys. In FlowHaskellRef, no similar mechanism is supported. Those keys are protected by some static labels so declassification statements are required after using them.

The programming style looks strange for people who are not familiar with point-free programming style. The point-free programming style adopted by *arrow* interface is quite different from the pointed style commonly used in the `do`-notation of monadic programming. It takes some time to transform a piece of pointed Haskell code to FlowHaskellRef if one is not familiar with point-free style. Paterson [10] develops a syntax transformation for *arrows* to do pointed programming. However, the transformation relies on combinator `pure` heavily and cannot be used in FlowHaskellRef.

Chapter 7

Extending FlowHaskellRef with Multithreaded Information Flow

Multithreaded programs open new timing channels to leak information. As explained in Sec. 2.2.1, new information flow policies are required in multithreaded programs. In FlowHaskellRef, a scheduler-independent run-time system is developed to eliminate *internal-timing* channels and guarantee *low-view determinism* [13].

7.1 Atomic Execution of Commands

As mentioned in Sec. 2.2.2, Volpano and Simth [21] introduced primitive `protect` and a uniform scheduler to guarantee multithreaded information-flow policies. When commands are executed in a `protect` block, other threads are locked immediately until the `protect` block finishes. Thus the running time of the `protect` block is invisible to other threads. After that, next thread scheduled is chosen uniformly among all threads. However, the semantics of `protect` and uniform scheduler are hard to achieve in practice. Russo and Sabelfeld [14] explain a defective implementation based on locks. The following is program 2.2 implemented in the lock-based framework:

$$\begin{aligned} c_1 &: \texttt{lock}; (\texttt{if h} > 0\ \texttt{then}\ \texttt{skip}(120)\ \texttt{else}\ \texttt{skip}(0)); \texttt{unlock}; \\ &\quad \texttt{lock}; \texttt{l} := 1; \texttt{unlock} \\ c_2 &: \texttt{lock}; \texttt{skip}(60); \texttt{unlock}; \texttt{lock}; \texttt{l} := 0; \texttt{unlock} \end{aligned}$$

The idea is to put `lock` and `unlock` before and after a command. When a thread is executing a command, it must have obtained the lock, and other threads are blocked because there is only one such global lock. By this way, the whole if-then-else statements can be executed atomically without interleaving with other threads. However, due to fairness, a lock is usually accompanied by a wait list. Threads that try to get a lock when the lock is held by some other thread are added to the list in order. The wait list fails to obey the semantics of `protect`, since next thread which can execute depends on the order of threads in the wait list, which again depends on the running time of the thread that holds the lock. See Russo and Sabelfeld's work [14] for details. In FlowHaskellRef, we adopt similar framework and implement a cooperative scheduler in the run-time system to achieve the same goal.

In the run-time system of FlowHaskellRef, `pure` computations and reference manipulation primitives are regarded as atomic commands and executed without interleaving with other threads. Since a security label, in a lattice, which is not $\bot$ is a *high* guard w.r.t. some security label, branching computations with guards higher than $\bot$ are also executed atomically. The following figure illustrates how the execution order of two thread commands is determined in the run-time system of FlowHaskellRef.

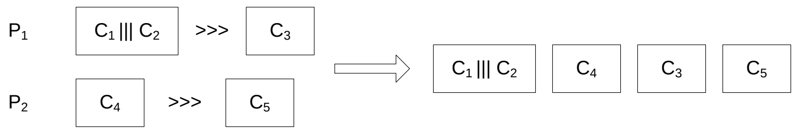

Figure 7.1: Execution order of multithreaded programs

Assume we have two threads, p_1 and p_2. Command c_1 and c_2 forms a branching computation with combinator (|||). Assume the branching computation contains a high guard and thus is executed atomically. The atomic commands of p_1 and p_2 are executed alternatively as depicted in the figure. No matter how the inputs are changed, the execution order of commands of p_1 and p_2 are still the same. Since the execution time of branching computations is invisible to other threads, no *internal-timing* channels can be exploited.

7.2 The Run-time System

This section explains the run-time system of FlowHaskellRef and shows how the system prevents *internal-timing* channels.

7.2.1 The Run-time Environment

The run-time Environment in FlowHaskellRef is implemented by data type `RRobin` as following(in FlowArrowRef.hs):

```
data RRobin a = RRobin
   { dat        :: a,
     iD         :: ThreadId,
     queue      :: TVar [ThreadId],
     blocks     :: Int
   }
```

Field `dat` stores the data used as inputs and outputs of computations. Field `iD` stores the thread identity of current thread. Field `queue` is a `TVar` pointed to a list of thread identities, while field `blocks` denotes how many nested atomic blocks a current thread is inside.

The behavior of schedulers in Haskell run-time system may differ from implementations to implementations. If the security of a program depends on certain property of a scheduler, a slightly change in the scheduler may make a program regarded as secure before fail to obey information-flow policies. Thus, a scheduler-independent run-time system is vital if we do not expect a scheduler to have certain property. In FlowHaskellRef, we implement a cooperative scheduler inside of the library instead. A thread queue, maintained by field `queue`, contains the thread identities of all running threads and is modified when a thread is created or completed. Only the thread whose identity is the same as the first element of the queue can execute commands and

$$\frac{}{\llparenthesis \texttt{arr}, m \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, m \rrparenthesis} \qquad \frac{\llparenthesis c_1, m \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, m' \rrparenthesis}{\llparenthesis c_1; c_2, m \rrparenthesis \;\rightarrow\; \llparenthesis c_2, m' \rrparenthesis}$$

$$\frac{id == (\texttt{head}\, q) \quad p == 0}{\llparenthesis \texttt{waitForYield}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, q, p) \rrparenthesis}$$

$$\frac{id \neq (\texttt{head}\, q) \quad p == 0}{\llparenthesis \texttt{waitForYield}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{sleep}, (id, q, p) \rrparenthesis}$$

$$\frac{p > 0}{\llparenthesis \texttt{waitForYield}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, q, p) \rrparenthesis}$$

$$\frac{p == 0}{\llparenthesis \texttt{yieldControl}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, \gamma(q), p) \rrparenthesis}$$

$$\frac{p > 0}{\llparenthesis \texttt{yieldControl}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, q, p) \rrparenthesis}$$

$$\frac{\llparenthesis \texttt{waitForYield}, m \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, q, p) \rrparenthesis}{\llparenthesis \texttt{beginAtomic}, m \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, (id, q, p+1) \rrparenthesis}$$

$$\frac{\llparenthesis \texttt{yieldControl}, (id, q, p-1) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, m \rrparenthesis}{\llparenthesis \texttt{endAtomic}, (id, q, p) \rrparenthesis \;\rightarrow\; \llparenthesis \texttt{stop}, m \rrparenthesis} \qquad \gamma(q) \;=\; (\texttt{tail}\, q) +\!\!+ [(\texttt{head}\, q)]$$

Figure 7.2: Semantics of `ArrowRef` Commands

the first thread identity is put to the end of the queue after executing an atomic command. In other words, the thread queue guarantees a round-robin execution order of all threads whatever the scheduler of Haskell run-time system is, and thus is scheduler independent.

7.2.2 ArrowRef

The run-time system of FlowHaskellRef is implemented in data type `ArrowRef`(in FlowArrowRef.hs).

```
data ArrowRef a b = AR ((RRobin a) -> IO (RRobin b))
```

It is a function that takes a run-time environment (`RRobin a`) and returns a run-time environment (`RRobin b`) in the IO monad. Data type `ArrowRef` is an *arrow* and has instances in type class `Arrow` and `ArrowChoice`.

7.2.3 Semantics of ArrowRef Primitives

In the lock-based framework, four new primitives in `ArrowRef` are developed. Primitive `waitForYield` and `yieldControl` are used to define an atomic command. Primitive `beginAtomic` and `endAtomic` are used to define an atomic block, which composes many atomic commands into one big atomic command. The small-step semantics of these primitives are depicted in Figure 7.2. The command syntax is defined as following.

$$\begin{aligned} \mathtt{c} ::= {} & \mathtt{stop} \mid \mathtt{sleep} \mid \mathtt{arr} \mid c; c \mid \mathtt{waitForYield} \\ & \mid \mathtt{yieldControl} \mid \mathtt{beginAtomic} \mid \mathtt{endAtomic} \end{aligned}$$

A command finishes normally becomes `stop`. Command `sleep` puts the current thread to sleep, and command `arr` represents any atomic *arrow* computation of `ArrowRef`. An environment `m` is a three-tuple (`id`,q,p), which correspond to `iD`, `queue`, `blocks` in `RRobin`. A command `c` and an environment `m` form a *command configuration*, $\langle\!|\mathtt{c},\mathtt{m}|\!\rangle$. Function `head` and γ are Haskell functions. Effects of command `arrs` in the memory are skipped here.

To avoid the wait list problem mentioned in Sec. 7.1, the implementations of those primitives are based on software transaction memory(STM) [4]. In fact, primitive `waitForYield` is implemented by an IO command `waitTurn` as following(in FlowArrowRef.hs):

```
waitTurn :: RRobin a -> IO ()
waitTurn env = if (p_count env) > 0
                 then return ()
                 else atomically (
                       do que <- readTVar (thread_que env)
                          if head que /= (ident env)
                            then retry
                            else return ())
waitForYield :: ArrowRef a a
waitForYild = AR (\env -> do waitTurn env
                             return env)
```

Command `waitTurn` first checks field `blocks` to see if the current thread is in an atomic block. If yes, it will not try to get the lock again so it returns and execute commands. Otherwise, the thread queue is read from the `TVar` `queue` of the environment and checked if current thread is the next thread which can execute commands. If it is not, a `retry` command is called and the current thread is put to sleep until other threads modify the thread queue. If it is the turn of the thread, `waitTurn` returns and the thread may executes commands that follows. Primitive `yieldControl` is implemented by a command `nextTurn` similar to `waitTurn`. We could have used more standard locking mechanism such as semaphore or `MVar`. However, the obtained code would be more complicated.

The following shows how every `pure` computation becomes an atomic command. (in FlowArrowRef.hs):

```
pure :: (b -> c) -> ArrowRef b c
pure f = waitForYield >>>
         AR (\env -> do b <- return (f (dat env))
                        return env{ dat = b } )
         >>> yieldControl
```

To define an atomic command, it is simply to enclose the computation with primitive `waitForYield` and `yieldControl`.

Similarly, to define an atomic block, a computation is enclosed by primitive `beginAtomic` and `endAtomic`. Primitive `beginAtomic` first calls `waitForYield` and then increases field `blocks` by 1. Primitive `endAtomic` first decreases field `blocks` by 1 and then calls `yieldControl`. A thread will not yield control if field `blocks` is larger than 0, which means the thread is still inside of an atomic block.

$$\textit{FORK}\frac{pc, \Theta \vdash f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l}{pc, \Theta \vdash \texttt{forkRef}\ f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ () \mid \bot}$$

$$\textit{ATOMIC}\frac{pc, \Theta \vdash f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l}{pc, \Theta \vdash \texttt{atomicA}\ f \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l}$$

Figure 7.3: Type system of `forkRef` and `atomicA`

In a lattice, except label $\bot$, other labels can be regarded as high guards w.r.t. some other labels. Therefore, an approximation is made to protect branching computations with guard higher than $\bot$. In FlowHaskellRef, combinator `right`, `left`, (+++), and (|||) create branching computations that should be protected. The following is the implementation of field `computation` of (+++) (in FlowArrowRef.hs):

```
computation = (\upd ->
                if (mext_join (upd s_in)) `label_leq` label_bottom
                  then right (c (upd.(replaceSecType sub1)))
                  else beginAtomic >>>
                       right (c (upd.(replaceSecType sub1))) >>>
                       endAtomic )
```

If the guard, `s_in`, contains any security label higher than $\bot$, the whole computation becomes a big atomic command.

Primitives of `waitForYield` and `yieldControl` as well as `beginAtomic` and `endAtomic` have to be pairwise. It is the responsibility of the library developers to maintain such property when creating new primitives in FlowHaskellRef.

7.3 New Combinators for Multithreaded Programming

Two new combinators are introduced in FlowHaskellRef. Combinator `forkRef` supports dynamic thread creation, while combinator `atomicA` provides atomic execution of a sequence of commands. The type system of the two combinators are as in Figure 7.3.

Combinator `forkRef` directs the input to the `FlowArrowRef` computation passed to it. A new thread with an exception handler is created to execute the computation. If exception `undefined` is raised in one of the running thread, all threads are killed immediately. Thus, no information is leaked. The output security type of `forkRef` is $\bot$ because only a unit is returned whatever the input is.

Combinator `atomicA` encloses a `FlowArrowRef` computation with a `beginAtomic` and `endAtomic` pair. The resulting computation is regarded as an atomic command.

Chapter 8

Case Study of Multithreaded FlowHaskellRef

The case study contains two parts and consolidates our approach towards closing *internal-timing* channels. The first example is a small multithreaded program to evaluate the effectiveness and efficiency of the run-time system. The second case study is an on-line shopping simulation. An attack program is implemented to show that FlowHaskellRef programs guarantee multithreaded information-flow security.

8.1 Experimental Example

In this example, a multithreaded program that exhibits *internal-timing* channels is implemented as follows(in MultiExp.hs):

```
t1 :: Int -> Int -> FlowArrowRef TriLabel ArrowRef () Int
t1 v dtime =
   lowerA LOW (pure (\() -> 99)) >>>
   createRefA (VSecLabel VLow) LOW >>>
   (lowerA LOW (pure id) &&& forkRef t2) >>>
   (second
     (pure (\() -> v) >>>
      pure (\i -> if i > 5 then Left dtime else Right 0) >>>
      ((skipRef >>> tagRef HIGH)
       |||
       (skipRef >>> tagRef HIGH)
      )
     )) >>>
   second (lowerA LOW (pure (\_ -> 1))) >>>
   (lowerA LOW (pure (\(x,y) -> x)) &&&
    (writeRefA >>> pure (\() -> 10000) >>> skipRef ))
   >>> lowerA LOW (pure (\(x,y) -> x)) >>>
   readRefA

t2 :: FlowArrowRef TriLabel ArrowRef (SecRef TriLabel Int) ()
t2 = (lowerA LOW (pure id) &&& lowerA LOW (pure (\_ -> 2)))
     >>> writeRefA
```

Thread `t1` creates a `LOW` reference and then dynamically creates a new thread `t2`. Thread `t2` writes integer 2 to the reference and finishes. Depending on `HIGH` value

Value of `v`	**Output** (1,2)	**Time**(**Sec.**)	**Value of** `v`	**Output** (1,2)	**Time**(**Sec.**)
10	(100,0)	43	0	(100,0)	34
10	(100,0)	44	0	(100,0)	34
10	(100,0)	43	0	(100,0)	34
10	(100,0)	44	0	(100,0)	34
10	(100,0)	43	0	(100,0)	34

Table 8.1: Result in `R1` with `dtime` $= 10000$

Value of `v`	**Output** (1,2)	**Time**(**Sec.**)	**Value of** `v`	**Output** (1,2)	**Time**(**Sec.**)
10	(83,17)	16	0	(15,85)	12
10	(90,10)	15	0	(13,87)	12
10	(86,14)	16	0	(16,84)	12
10	(90,10)	16	0	(18,82)	13
10	(89,11)	16	0	(11,89)	13

Table 8.2: Result in `R2` with `dtime` $= 10000$

`v`, thread `t1` has different executing steps in each branch. Primitive `skipRef` takes an integer `k` as the input and performs `k` empty steps. Each empty step is a `return` statement in the IO monad. In the example, if `v` > 5, then the branch executes `dtime` empty steps. Otherwise, zero empty step is executed. Then, thread `t1` writes integer 1 to the reference. After executing 10000 empty steps, the content of the reference is read out as the output.

The same program is run in two run-time systems: `R1` and `R2`. Run-time system `R1` is the one adopted in FlowHaskellRef. Run-time system `R2` is similar to `R1` but without using `waitForYield` and `yieldControl` to execute each command atomically and also `beginAtomic` and `endAtomic` to protect a branching computation. For each test case, the program is run 100 times. The results with `dtime` = 10000 are listed in Table 8.1 and Table 8.2, while the results with `dtime` = 15000 are shown in Table 8.3 and Table 8.4. If `v` = 10, the branch that runs `dtime` empty steps is chosen. Otherwise, the branch that runs zero empty step is chosen. Column **Output** records the frequency of the outputs, 1 or 2, among 100 runs. The running time of a test case is recorded in column **Time**. [1]

When the program runs in the run-time system `R1`, the output is always 1 no matter which branching computation is chosen. But when the program runs in the run-time system `R2`, the statistic results in Table 8.2 and Table 8.4 shows the *internal-timing* channels. If `v` = 10, the probability of 1 as the output is higher because the statement that assign 1 to the reference in `t1` tends to execute later than the statement that assign 2 to the reference in `t2`. On the other hand, if `v` = 0, the execution order of the two assignments tends to be reversed. Thus, the probability of 2 as the output is higher. This assumption is corroborated by comparing the results in Table 8.2 and Table 8.4. When `v` = 10, the larger `dtime` is the higher probability of 1 as the output becomes.

The running time of the program in run-time system `R1` are about three times slower than that in run-time system `R2`. The huge overhead is introduced by the internal thread queue. Most of the threads that are scheduled by the scheduler are put to sleep again because they are not the next thread which can execute commands. However, the cost

[1]The program is run on a laptop with Pentium M 1.5 GHz and 512 MB RAM.

Value of `v`	**Output** (1,2)	**Time**(**Sec.**)	**Value of** `v`	**Output** (1,2)	**Time**(**Sec.**)
10	(100,0)	47	0	(100,0)	35
10	(100,0)	48	0	(100,0)	35
10	(100,0)	47	0	(100,0)	35
10	(100,0)	47	0	(100,0)	33
10	(100,0)	47	0	(100,0)	35

Table 8.3: Result in `R1` with `dtime` $= 15000$

Value of `v`	**Output** (1,2)	**Time**(**Sec.**)	**Value of** `v`	**Output** (1,2)	**Time**(**Sec.**)
10	(96,4)	18	0	(18,82)	12
10	(98,2)	18	0	(17,83)	12
10	(98,2)	17	0	(20,80)	12
10	(93,7)	18	0	(12,88)	12
10	(95,5)	17	0	(19,81)	12

Table 8.4: Result in `R2` with `dtime` $= 15000$

is normally tolerable since only small part of a program that involves secret data is built in FlowHaskellRef. The remaining part of the program keeps the same efficiency as before.

8.2 On-line Shopping Simulation

In this case study, a program that simulates an on-line shopping application is implemented in FlowHaskellRef. The program is run in two run-time systems, called `R1` and `R2`. Run-time system `R1` is as described in Sec. 7.2, while run-time system `R2` does not use `waitForYield`, `yieldControl`, `beginAtomic`, and `endAtomic` to protect branching computations. An attack program trying to exploit *internal-timing* channels is developed to evaluate different run-time systems.

8.2.1 Simulation Program

To purchase a product, each client sends a name, a telephone number, a credit card number, and an address to a service program provided by the company that sells the product. The service program is divided into three phases. The first phase is implemented by a trusted procedure to take a list of client data and protect those data with proper security types. Names, telephone numbers, and addresses are public data and protected by security label `LOW`. Credit card numbers are regarded as a secret data and thus protected by security label `HIGH`. In the second phase, a thread executing procedure `purchase` is forked to process the protected data for each client. This procedure is not trusted. The type signature of the function is as following(in Shopping.hs):

```
type Protected b c = FlowArrowRef TriLabel ArrowRef b c

purchase :: (STSecType rs1 TriLabel,
             STSecType rs2 TriLabel)
            => Protected ((SRef rs1 [CNum],
                           SRef rs2 [PubItem]),
                          (CNum,((Name,Tel),Addr)))
                        ()
```

Function `purchase` takes two references and a client's data. The first reference stores credit card numbers and the content is protected by security type (`SecLabel (Lab HIGH)`). The second reference stores other public information and thus protected by (`SecLabel (Lab LOW)`). In the simulation, the process of data is simply to separate secret data and public data, and store them in the corresponding references. The third phase is to process the public and the secret references by a trusted procedure. In the simulation, the public data are printed on the screen.

A malicious program can be implemented in `purchase` and try to leak information about the credit card number to the public reference. Those information can be collected after the public data are printed in the third phase.

8.2.2 Malicious Program

The malicious program exploits the *internal timing* channels to infer a credit card number and store it in the public reference. Similar techniques as in Sec. 8.1 is adopted. An attack reference, which is public, is created and its final value depends on a secret data. See Sec. 8.1 for details. However, the attack only reveals one bit information each time. In the malicious program, the attack is magnified by repeating itself several times to infer all bits of a credit card number. The attack begin with the most significant bit of a credit card number. Assume the credit card number has `n` bits. The following code shows how to infer this bit(in Shopping.hs).

```
lowerA HIGH (pure (\(k,cnum) -> if cnum >= 2^k
                                  then Left 30000
                                  else Right 0)) >>>
                   ((skipRef >>> tagRef HIGH)
                    |||
                    (skipRef >>> tagRef HIGH))
```

Variable $\texttt{k} = \texttt{n} - 1$ in this case. If the credit card number, `cnum`, is larger than or equal to 2^{n-1}, the branch taking 30000 empty steps is chosen. Otherwise, the branch taking 0 empty steps is run. Based on the final value of the attack reference, the information of this bit is exploited and appended to the public address field. If $\texttt{cnum} >= 2^{n-1}$, $\texttt{cnum} - 2^{n-1}$ becomes the new `cnum`. The whole procedure is repeated to infer next bit with $\texttt{k} = \texttt{n} - 2$. By performing the attack `n` times, every bit of the credit card number is revealed in the address field and is written to the public reference later.

We assume a credit card number has 16 digits in decimal and thus at most 54 bits. The client information is as following:

```
Name  : Bob
Tel.  : 07042312323
CNum. : 9999999999999999
Addr. : Rotary 2K-1234, Gothenburg.
```

The binary representation of the credit card number is:

```
100011100001101111001001101111111000000111111111111111
```

The malicious program is run in both run-time system `R1` and `R2`.

Run-time system `R1`

When the malicious program executes in run-time system `R1`, the output is as following every time it runs.

Run No.	Result	Time(Sec.)
1	101011111001111111111011101111110000011111111111111111	27
2	110011100001101111011101101111010000001111111111111111	27
3	101011100001101111101001101111110000001111111111111111	28
4	100011100101101111001001101111110000001111111111111011	28
5	100011100001101111001001101111110100001111111111111111	29
6	100011100001101111001000101111110001011111111111111111	29
7	100011100001101111011001111111110000001111111111111111	28
8	100111100001101111001101101111110000001111111111111111	28
9	100010100001101111001001101111110000001111111111111111	27
10	101011100001111111001001101111110000001111111111111111	28
stat.	0979010999*0080000870891090000 10*9*9*80000000000000100	
cnum'	100011100001101111001001101111110000001111111111111111	

Table 8.5: Statistical results of the malicious program

```
### Client ###
Name : Bob
Tel  : 07042312323
Addr : Rotary 2K-1234, Gothenburg.
000000000000000000000000000000000000000000000000000000
```

The last line shows the bits inferred from the credit card number. They are all zeros so no information of the credit card number is revealed.

Run-time system `R2`

When the malicious program runs in run-time system `R2`, one of the output is as following.

```
### Client ###
Name : Bob
Tel  : 07042312323
Addr : Rotary 2K-1234, Gothenburg.
101011111001111111111011101111110000011111111111111111
```

There are usually 2 or 3 bits that are not correctly inferred. This is probably because the execution time of empty steps is slightly different in each run. Besides, we do not know how the scheduler of the Haskell run-time system treats a fork command. However, if the attack is repeated several times, we can still infer the credit card number with high confidence. Table 8.5 lists the results and the corresponding running times in ten runs. The row `sum` records how many times 0 appears in a certain digit. Symbol * means ten times. For instance, the second digit of row `stat.` is 9. This means that 0 appears in the second digit of the result nine times among the ten runs. The inferred credit card number, `cnum′`, is decided by row `stat.`. If the frequency of 0 appearing in a digit is higher than five times, the digit is inferred as 0. Otherwise, the digit is inferred as 1. Row `cnum′` is the final credit card number inferred, which is the same as the real credit card number. By this method, an attacker can obtain the credit card number of a client within five minutes.[2]

[2]The program is run on a laptop with Pentium M 1.5 GHz and 512 MB RAM.

Chapter 9

Discussions and Conclusions

9.1 Combinator `lowerA`

Combinator `lowerA` is created for transforming combinator `pure`. The inputs of `lowerA` first pass through an input filtering function, and the output security types are deduced from the output types. The reason that those mechanisms are not implemented in combinator `pure` is because every type used has to implement overloading functions in some Haskell type classes. To be generic, some type constraints are required in the type signature of `pure`. For instance, to deduce output security types, the *arrow* should be an instance of type class `TakeOutputType`. The type signature of `pure` is expected as following:

```
pure :: DowngradArrow SecType l a b c =>
        (b -> c) -> a b c
```

However, the type signature of `pure` is fixed and cannot be changed.

With the input filtering function, combinator `lowerA` provides more flexibility than the `pure` in the previous work. Combinator `lowerA` may takes secret inputs and produces public outputs as long as the outputs do not depend on secret inputs. This has been proved useful in the case studies. At some point of a program, programmers may like to discard all secret data and begin with a data of lower security level. For example, in the bank simulation cast study, computation `atm_nonce` takes a unit with security level `ATM_BANK` and returns a nonce. We may like to get the nonce after processing some data of security level `BANK`. Consider the following program.

```
lowerA BANK (pure (\x -> x)) >>>
lowerA ATM_BANK (pure (\i -> ())) >>> atm_nonce
```

The inputs `i` with security level `BANK` is discarded and a unit with security level `ATM_BANK` is returned in the `lowerA`. Without this flexibility, programmers have to declassify the unit. Nevertheless, more declassification requires more manual inspections.

However, the cost of the flexibility is that part of checking is postponed to the execution time of underlying computations. Value `undefined` causes an exception and aborts a running computation when it is used. To prevent such delayed checking, programmers may use combinator `equalA` instead. Combinator `equalA` has a strict constraint on the input security types but guarantees all constraints are checked when certifying a FlowHaskellRef program.

9.2 Singleton Types

As described in Sec. 4.6.2, each singleton type corresponds to a data constructor of a lattice or `SecType`. The relation is connected by type classes `STLabel` and `STSecType`. They have methods to transform a singleton type to the corresponding data constructor, but no transformation from a data constructor to its singleton type. This is because type classes can only capture the relation from singleton types to a lattice or `SecType`, but not the other way around. Since singleton types are encoded in references, constructors of singleton types are necessary to build the security types. With one-way transformation, programmers have to specify security types via singleton type constructors in `createRefA`. The constructors of singleton types can be eliminated if some form of fine-grained type is provided for constructors of security labels and security types. Ωmega [18], a programming language which supports GADTs and extensible kinds, could be used to solve the problem.

9.3 Closing Timing Channels via Cooperative Scheduler

In multithreaded programs, it is the interleaving of the threads that opens the timing channels. However, the interleaving of the threads is affected not only by the programs but also by the schedulers. Different schedulers may create timing channels in different ways.

There has been some studies showing how to apply static type system techniques to reject insecure programs [1, 21, 22]. One of the advantages of static type system is the programs have no run-time overhead. But those type systems are normally too restrictive or require non-standard semantics. Some of them may reject intuitively secure and useful programs and are not practical for real-world applications. Others retain the permissiveness to accept more secure programs but make assumptions about schedulers. Many of the assumptions are hard to achieve in practice. For instance, a non-deterministic scheduler. However, an observation is that more regulation in schedulers provides higher permissiveness to accept secure programs.

FlowHaskellRef chooses to implement an internal scheduler to eliminate internal timing channels. This approach accepts all original programs accepted in the sequential version of FlowHaskellRef. The programmers write programs as before without worrying about what kind of pattern may create timing channels. Besides, the approach is scheduler independent. Programs written in FlowHaskellRef preserve information-flow policies in all Haskell run-time system. Moreover, the implementation is simple and has potentially fewer bugs. However, the cost of the approach with high permissiveness is a non-trivial overhead. FlowArrowRef implements the internal scheduler by a thread queue in a round-robin fashion. Threads are scheduled by two schedulers. When scheduled by Haskell run-time system, if the thread is not scheduled by the internal thread queue, it is put to sleep. In worst case, every thread is scheduled once by Haskell run-time system and put to sleep before the thread scheduled by the internal scheduler is scheduled.

However, FlowHaskellRef, as an embedded language, can be used along with normal Haskell to build programs. In many cases, only small part of a system that involves secret information is written in FlowHaskellRef. Other part of the system can still be written in normal Haskell. The efficiency cost is comparatively low in exchange for no information leak.

To reduce the overhead, one way would be to give up the internal scheduler and have interactions with the scheduler of Haskell run-time system. Similar approach has been explored by Russo and Sabelfeld [14].

In FlowHaskellRef, combinator `iterateA` which might produce infinite loop is not ruled out from branching computations. This is because the design goal of FlowHaskell-Ref is to provide programmers a expressive and practical tool. Thus, we decide to give the responsibility to programmers.

9.4 Future Work

9.4.1 Decentralized Label Model

As described in Sec. 6.3.2, dummy labels are required for the integrity of the lattice in the case study. This is because a lattice has a global top element that represents a universally trusted authority. However, such authority in real-world situations are normally not existed. There are usually authorities that do not trust each other but cooperate to complete interesting tasks. Decentralized label model(DLM) [8] has been shown suitable for modelling these situations. As mentioned in the previous work [7], extending decentralized label model(DLM) in FlowHaskell is not difficult. The same idea can also be used in FlowHaskellRef.

9.4.2 Arrow Syntax Transformation

Paterson [10] proposes a `do`-notation like syntax transformation for *arrows*. It allows programmers to write pointed *arrow* code. Pointed programming style is favored by some people because it provides a similar reasoning style as in imperative languages. The transformation relies heavily on combinator `pure` to organize the structures of data. However, to build useful programs in FlowHaskellRef, combinator `pure` is normally used in conjunction with combinator `lowerA` to deduce output security types. Thus, the syntax transformation cannot be used for FlowHaskellRef.

New syntax transformation for FlowHaskellRef can be extended similarly as current syntax transformation. However, all `pure` combinators generated in the transformation are wrapped by a `lowerA` combinator. Programmers need to specifies a security label as before for `lowerA` but maybe in another way.

9.4.3 Multiple Authorities in a Secure Program

In FlowHaskellRef, only one authority can be used to certify a secure program. This is sufficient for sequential programs. Assume a program requires cooperation of two different authorities, called `A` and `B`. It can be written in several small programs that only require one of the authority. If a sub-program which belongs to authority `B` requires data from the other sub-program which belongs to authority `A`, the second program is certified by authority `A` and then pass the output to the first program.

The situation is different for multithreaded programs in FlowHaskellRef. To guarantee no *internal-timing* channels in a multithreaded program, all dynamic threads have to be created via `forkRef` in FlowHaskellRef. This means the whole secure program only belongs to one authority. Consider an auction simulation program in the following figure.

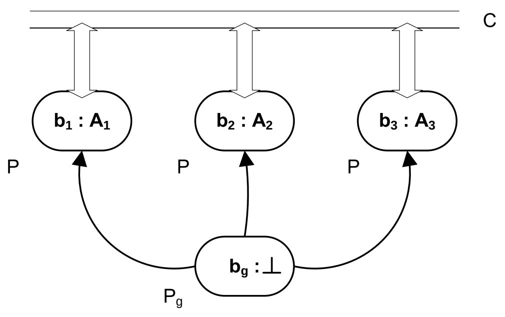

Figure 9.1: Auction simulation

There is a public global bid, b_g, which can be accessed by bidders. The global bid is maintained by program P_g. It create a client thread, P, for each bidder that has his own private bid, b_i. The snapshot is after three bidders joined the auction. There is a public channel, C, for bidders to exchange messages that are not related to their private bids. To protect program P from leaking the private bid p_i in the public channel, each client's bid is protected by a different security level, A_i. The auction proceeds with each client compares its bid b_i with the global bit b_g. If $b_i > b_g$, the client increases b_g by 1. Otherwise, the client loses and leaves the auction. The increment of b_g depends on private bid so declassification is required. Thus, client thread with bid b_i should be certified at least with authority A_i. However, the program cannot be implemented in FlowHaskellRef because only one authority can be used in the program. To allow all legal declassification, an authority higher than all A_is are required. But this allows clients to read other bidders' bids on the public channel. Mechanisms to compose programs of different authorities are necessary to build the auction simulation program.

9.5 Conclusion

FlowHaskellRef extends FlowHaskell with reference manipulation and secure multi-threaded programming. The new contributions are listed as following:

1. Complex security types permits more accurate description of data
2. Reference manipulation provides possibility of shared resources
3. Unification mechanism infers security types automatically and mitigates the responsibility of programmers
4. Scheduler-independent run-time system eliminates *internal-timing* channels in multithreaded programming
5. Two full case studies to evaluate FlowHaskellRef

The design choices of FlowHaskellRef are mainly of practical concerns. It is aimed to provide programmers with an easy-to-use and expressive secure language. As a light-weighted tool, the migrating cost is much lower comparing to full-fledged security languages like Jif [9] and FlowCaml [17]. Programmers can only write security-related part of a program in FlowHaskellRef and switch between FlowHaskellRef and Haskell seamlessly.

On the other hand, FlowHaskellRef also have some limitations. First, the certification of a program is at run-time. This makes the testing of a FlowHaskellRef program more difficult. A FlowHaskellRef program has to be run at least once to ensure all constraints are satisfied. Besides, the same checking is performed every time the program runs. Second, the debugging message is not clear. All constraints are collected at run-time so the information of where a constraint comes from is hard to provide.

Bibliography

[1] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[2] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[3] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[4] T. Harris, S. Marlow, S. Peyton Jones, M. Herlihy. Composable Memory Transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, June 2005.

[5] J. Hughes. Generalising monads to arrows. In *Science of Computer Programming*, volumn 37, pages 67–111, May 2000.

[6] J. Hughes. Programming with Arrows. In *5th International Summer School on Advanced Functional Programming*, LNCS 3622, pages 73–129, Springer, 2005.

[7] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. IEEE Computer Security Foundations Workshop*, pages 16, July 2006.

[8] A. C. Myers, B. Liskov. Protecting privacy using the decentralized label model. In *ACM Transactions on Software Engineering and Methodology*, 9(4):410-442, 2000.

[9] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. `http://www.cs.cornell.edu/jif`, July 2001–2006.

[10] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240, ACM Press, Sep. 2001.

[11] R. Paterson. Arrows and computation. In *The Fun of Programming*(Jeremy Gibbons and Oege de Moor, Eds.), pages 201–222, Palgrave, 2003.

[12] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages(POPL)*, Portland, Oregon, Jan. 2002.

[13] A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symp. on Security and Privacy*, pages 114-127 , May 1995.

[14] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Security Foundations Workshop*, July 2006.

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

[16] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[17] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[18] T. Sheard. Putting curry-howard to work. In *Proc. of the 2005 ACM SIGPLAN workshop on Haskell, page 74–85, 2005.*

[19] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[20] S. Tse and G. Washburn. Cryptographic Programming in Jif. Course project, 2003. `http://www.cis.upenn.edu/ stse/bank/main.pdf`

[21] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3):231–253, Nov. 1999.

[22] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

Appendix

A Syntactic Sugars

The section explains primitives whose functions can be achieved by `pure` and `lowerA`. However, they are concise in syntax and without pushing up output security types as in `lowerA`. These primitives provide patterns required frequently and proved convenient in case studies.

- **idRef**
  Pass the value and the security type of input.

$$\frac{}{\top, \emptyset \vdash \texttt{idRef} \ : \ \tau \mid s^l \ \rightarrow \ \tau \mid s^l}$$

- **nullRef**
  Return a unit with security type $\bot$.

$$\frac{}{\top, \emptyset \vdash \texttt{nullRef} \ : \ \tau \mid s^l \ \rightarrow \ () \mid \bot}$$

- **fstPair**
  Return the value and the security type of the first element of a pair.

$$\frac{}{\top, \emptyset \vdash \texttt{fstPair} \ : \ (\tau_1, \tau_2) \mid (s_1^l, s_2^l) \ \rightarrow \ \tau_1 \mid s_1^l}$$

- **sndPair**
  Return the value and the security type of the second element of a pair.

$$\frac{}{\top, \emptyset \vdash \texttt{sndPair} \ : \ (\tau_1, \tau_2) \mid (s_1^l, s_2^l) \ \rightarrow \ \tau_2 \mid s_2^l}$$

- **pairRight**
  Right associates nested pairs.

$$\frac{}{\top, \emptyset \vdash \texttt{pairRight} \ : \ ((\tau_1, \tau_2), \tau_3) \mid ((s_1^l, s_2^l), s_3^l) \ \rightarrow \ (\tau_1, (\tau_2, \tau_3)) \mid (s_1^l, (s_2^l, s_3^l))}$$

- **pairLeft**
  Left associates nested pairs.

$$\frac{}{\top, \emptyset \vdash \texttt{pairLeft} \ : \ (\tau_1, (\tau_2, \tau_3)) \mid (s_1^l, (s_2^l, s_3^l)) \ \rightarrow \ ((\tau_1, \tau_2), \tau_3) \mid ((s_1^l, s_2^l), s_3^l)}$$

B Complete Haskell Code

B.1 Lattice.hs

Lattice.hs

```
module Lattice where

class (Eq a, Show a) => Lattice a where
  label_top :: a
  label_bottom :: a
  label_join :: a -> a -> a
  label_meet :: a -> a -> a
  label_leq :: a -> a -> Bool

data TriLabel = LOW | MEDIUM | HIGH
                deriving (Eq, Show)

instance Lattice TriLabel where
  label_top = HIGH
  label_bottom = LOW
  label_join x y = if label_leq x y then y else x
  label_meet x y = if label_leq x y then x else y
  label_leq LOW _ = True
  label_leq MEDIUM LOW = False
  label_leq MEDIUM _ = True
  label_leq HIGH HIGH = True
  label_leq HIGH _ = False

data Label l = Lab l | LVar String
     deriving (Eq, Show)

data SecType l = SecLabel (Label l) | SecRef (SecType l) (Label l) |
                 SecPair (SecType l) (SecType l) | SecHigh |
                 SecEither (SecType l) (SecType l) (Label l) | SecVar String
                deriving (Eq, Show)

instance Lattice l => Ord (SecType l) where
-- (<=)
  (SecLabel (Lab l1)) <= (SecLabel (Lab l2)) = label_leq l1 l2
  SecHigh <= (SecLabel (Lab l2)) = label_leq label_top l2
  (SecLabel (Lab l1)) <= SecHigh = label_leq l1 label_top
  (SecRef st1 (Lab lr1)) <= (SecRef st2 (Lab lr2)) =
                                 label_leq lr1 lr2 && (st1 <= st2) && (st2 <= st1)
  SecHigh <= (SecRef st2 (Lab lr2)) = (label_leq label_top lr2) && (SecHigh <= st2)
  (SecRef st1 (Lab lr1)) <= SecHigh = (label_leq lr1 label_top) && (st1 <= SecHigh)
  (SecPair s1 t1) <= (SecPair s2 t2) = (s1 <= s2) && (t1 <= t2)
  SecHigh <= (SecPair s2 t2) = (SecHigh <= s2) && (SecHigh <= t2)
  (SecPair s1 t1) <= SecHigh = (s1 <= SecHigh) && (t1 <= SecHigh)
  (SecEither s1 t1 (Lab l1)) <= (SecEither s2 t2 (Lab l2)) =
                                   (s1 <= s2) && (t1 <= t2) && label_leq l1 l2
  (SecEither s1 t1 (Lab l1)) <= SecHigh =
                     (s1 <= SecHigh) && (t1 <= SecHigh) && label_leq l1 label_top
  SecHigh <= (SecEither s1 t1 (Lab l2)) =
                     (SecHigh <= s1) && (SecHigh <= t1) && label_leq label_top l2
  t1 <= t2 = error $ "undefined relation : " ++ (show t1) ++ " and " ++ (show t2)
-- max
  max (SecLabel (Lab l1)) (SecLabel (Lab l2)) = SecLabel (Lab (label_join l1 l2))
  max (SecRef t1 (Lab l1)) (SecRef t2 (Lab l2)) =
                           if t1 /= t2
                             then error $ "SecRef : no join defined."
                             else SecRef t1 (Lab (label_join l1 l2))
  max (SecPair s1 t1) (SecPair s2 t2) = SecPair (max s1 s2) (max t1 t2)
  max SecHigh SecHigh = SecHigh
  max (SecEither s1 t1 (Lab l1)) (SecEither s2 t2 (Lab l2)) =
                   SecEither (max s1 s2) (max t1 t2) (Lab (label_join l1 l2))
-- min
  min (SecLabel (Lab l1)) (SecLabel (Lab l2)) = SecLabel (Lab (label_meet l1 l2))
  min (SecRef t1 (Lab l1)) (SecRef t2 (Lab l2)) =
                        if t1 /= t2
                          then error $ "SecRef : no meet defined."
                          else SecRef t1 (Lab (label_meet l1 l2))
  min (SecPair s1 t1) (SecPair s2 t2) = SecPair (min s1 s2) (min t1 t2)
  min SecHigh SecHigh = SecHigh
  min (SecEither s1 t1 (Lab l1)) (SecEither s2 t2 (Lab l2)) =
              SecEither (min s1 s2) (min t1 t2) (Lab (label_meet l1 l2))

ml_label :: Lattice l => l -> SecType l -> SecType l
ml_label l (SecLabel _) = SecLabel (Lab l)
ml_label l (SecRef t _) = SecRef t (Lab l)
ml_label l (SecPair t1 t2) = SecPair (ml_label l t1) (ml_label l t2)
ml_label l SecHigh = SecHigh
ml_label l (SecEither t1 t2 _) = SecEither (ml_label l t1) (ml_label l t2) (Lab l)

mextract :: Lattice l => SecType l -> l
mextract (SecLabel (Lab l)) = l
mextract (SecRef s (Lab l)) = l
mextract (SecPair s t) = (mextract s) `label_join` (mextract t)
mextract SecHigh = label_top
mextract (SecEither s t (Lab l)) = l
mext_join s = propLeaf ext label_join s
mext_meet s = propLeaf ext label_meet s

ml_tag :: Lattice l => SecType l -> l -> SecType l
ml_tag (SecLabel _) l = SecLabel (Lab l)
ml_tag (SecRef s _) l = SecRef s (Lab l)
ml_tag (SecPair s1 s2) l = SecPair (ml_tag s1 l) (ml_tag s2 l)
ml_tag (SecEither s1 s2 _) l = SecEither (ml_tag s1 l) (ml_tag s2 l) (Lab l)
ml_tag SecHigh l = SecHigh

ml_decl :: Lattice l => SecType l -> SecType l -> l -> SecType l
ml_decl SecHigh s_out l = s_out
ml_decl (SecLabel (Lab l1)) _ l = SecLabel (Lab (if label_leq l1 l then l1 else l))
ml_decl (SecRef s (Lab l1)) (SecRef s' _) l =
                  SecRef (ml_decl s s' l) (Lab (if label_leq l1 l then l1 else l))
ml_decl (SecPair s1 s2) (SecPair s1' s2') l =
                  SecPair (ml_decl s1 s1' l) (ml_decl s2 s2' l)
ml_decl (SecEither s1 s2 (Lab l1)) (SecEither s1' s2' _) l =
              SecEither (ml_decl s1 s1' l) (ml_decl s2 s2' l)
                        (Lab (if label_leq l1 l then l1 else l))

ml_eqstruct :: Lattice l => SecType l -> SecType l -> Bool
ml_eqstruct (SecLabel _) (SecLabel _) = True
ml_eqstruct (SecRef s1 _) (SecRef s2 _) = ml_eqstruct s1 s2
ml_eqstruct (SecPair s1 t1) (SecPair s2 t2) =
                      ml_eqstruct s1 s2 && ml_eqstruct t1 t2
ml_eqstruct (SecEither s1 t1 _) (SecEither s2 t2 _) =
                      ml_eqstruct s1 s2 && ml_eqstruct t1 t2
ml_eqstruct _ _ = False

ext (SecLabel (Lab l)) = l
ext SecHigh = label_top
```

```
-- Check if a (SecType l) contains certain property at leaf
-- f   : a property function applied to leaves
-- con : a combinator for the result of f
propLeaf :: (SecType l -> a) -> (a -> a -> a) -> SecType l -> a
propLeaf f con v@(SecVar _) = f v
propLeaf f con v@(SecHigh) = f v
propLeaf f con v@(SecLabel _) = f v
propLeaf f con (SecRef t l) = (propLeaf f con t) `con` (f (SecLabel l))
propLeaf f con (SecPair t1 t2) = (propLeaf f con t1) `con` (propLeaf f con t2)
propLeaf f con (SecEither t1 t2 l) = (propLeaf f con t1) `con`
                                     (propLeaf f con t2) `con` (f (SecLabel l))

class Lattice l => Guard a l where
  label_guard :: a l -> a l -> Bool
  label_gsleql :: a l -> a l -> Bool
  label_glleqs :: a l -> a l -> Bool

instance Lattice l => Guard SecType l where
  label_guard (SecLabel (Lab l)) s = label_leq l (mextract s)
  label_gsleql s (SecLabel (Lab l)) = label_leq (mext_join s) l
  label_glleqs (SecLabel (Lab l)) s = label_leq l (mext_meet s)
```
Lattice.hs

B.2 RLattice.hs

RLattice.hs

```
module RLattice where

import Lattice

-- Singleton types for TriLabel
data SLow = VLow
  deriving Show
data SMedium = VMedium
  deriving Show
data SHigh = VHigh
  deriving Show

class STLabel rl l where
  toLabel :: rl -> l

instance STLabel SLow TriLabel where
  toLabel _ = LOW
instance STLabel SMedium TriLabel where
  toLabel _ = MEDIUM
instance STLabel SHigh TriLabel where
  toLabel _ = HIGH

-- Singleton types for SecType
data SSecLabel l = VSecLabel l
  deriving Show
data SSecRef s l = VSecRef s l
  deriving Show
data SSecPair s1 s2 = VSecPair s1 s2
  deriving Show
data SSecEither s1 s2 l = VSecEither s1 s2 l
  deriving Show

class STSecType rs l where
```

```
  toSecType :: rs -> SecType l

instance STLabel rl l => STSecType (SSecLabel rl) l where
  toSecType _ = SecLabel (Lab (toLabel (undefined::rl)))
instance (STSecType rs l, STLabel rl l)
         => STSecType (SSecRef rs rl) l where
  toSecType _ = SecRef (toSecType (undefined::rs)) (Lab (toLabel (undefined::rl)))
instance (STSecType rs1 l, STSecType rs2 l)
         => STSecType (SSecPair rs1 rs2) l where
  toSecType _ = SecPair (toSecType (undefined::rs1)) (toSecType (undefined::rs2))
instance (STSecType rs1 l, STSecType rs2 l, STLabel rl l)
         => STSecType (SSecEither rs1 rs2 rl) l where
  toSecType _ = SecEither (toSecType (undefined::rs1))
                          (toSecType (undefined::rs2))
                          (Lab (toLabel (undefined::rl)))
```

RLattice.hs

B.3 RefOp.hs

RefOp.hs

```
module RefOp where

import Data.IORef

data SRef rs a = MkSRef rs (IORef a) (a->a)

newSRef :: a -> rs -> IO (SRef rs a)
newSRef a rs = do r <- newIORef a
                  return (MkSRef rs r id)

readSRef :: SRef rs a -> IO a
readSRef (MkSRef rs r f) = do x <- readIORef r
                              return (f x)

writeSRef :: SRef rs a -> a -> IO ()
writeSRef (MkSRef rs r _) a = writeIORef r a

class RefMonad m r rs | m -> r where
  createMRef :: a -> rs -> m (r rs a)
  readMRef :: (r rs a) -> m a
  writeMRef :: (r rs a) -> a -> m ()

instance RefMonad IO SRef rs where
  createMRef = newSRef
  readMRef = readSRef
  writeMRef = writeSRef
```

RefOp.hs

B.4 SecureFlow.hs

SecureFlow.hs

```
module SecureFlow where

import Lattice
import RLattice
import Unification
import Control.Arrow
import RefOp
import Priv
import Data.IORef
import Data.LargeWord
import Codec.Utils

class (Lattice l) => BuildSecType l t where
  buildSecType :: l -> t -> SecType l

instance (Lattice l) => BuildSecType l () where
  buildSecType l _ = (SecLabel (Lab l))
instance (Lattice l) => BuildSecType l Int where
  buildSecType l _ = (SecLabel (Lab l))
instance (Lattice l) => BuildSecType l Integer where
  buildSecType l _ = (SecLabel (Lab l))
instance (Lattice l, BuildSecType l a, BuildSecType l b) =>
           BuildSecType l (a,b) where
  buildSecType l _ = (SecPair (buildSecType l (undefined::a))
                              (buildSecType l (undefined::b)))
instance (Lattice l, STSecType rs l) =>
            BuildSecType l (SRef rs a) where
  buildSecType l _ = (SecRef (toSecType (undefined::rs)) (Lab l))
instance (Lattice l,
          BuildSecType l a, BuildSecType l b)
          => BuildSecType l (Either a b) where
  buildSecType l _ = (SecEither (buildSecType l (undefined::a))
                                (buildSecType l (undefined::b)) (Lab l))
-- The following instances are for types used in case studies.
instance Lattice l => BuildSecType l [a] where
  buildSecType l _ = (SecLabel (Lab l))
instance Lattice l => BuildSecType l Word128 where
  buildSecType l _ = (SecLabel (Lab l))
instance Lattice l => BuildSecType l Bool where
  buildSecType l _ = (SecLabel (Lab l))
instance Lattice l => BuildSecType l (Privilege TriLabel) where
  buildSecType l _ = (SecLabel (Lab l))

class (Lattice l, Arrow a) => TakeOutputType l a b c where
  deriveSecType :: l -> (a b c) -> SecType l

instance (Lattice l, BuildSecType l c, Arrow a) =>
          TakeOutputType l a b c where
  deriveSecType l _ = buildSecType l (undefined::c)

class (Lattice l) => FilterData l t where
  removeData :: l -> (SecType l) -> t -> t

instance (Lattice l) => FilterData l () where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance (Lattice l) => FilterData l Int where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance (Lattice l) => FilterData l Integer where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance (Lattice l, FilterData l a, FilterData l b) =>
         FilterData l (a,b) where
  removeData l (SecPair lx ly) (x,y) = (removeData l lx x, removeData l ly y)
  --removeData l p pd = error $ (show l) ++ " :: " ++ (show p)
instance (Lattice l, STSecType rs l, FilterData l a) =>
         FilterData l (SRef rs a) where
  removeData l (SecRef t (Lab l')) (MkSRef rs c fread) =
        if label_leq l' l
          then (MkSRef rs c ((removeData l t).fread))
          else undefined
-- The following instances are for types used in case studies.
instance Lattice l => FilterData l [a] where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance Lattice l => FilterData l Word128 where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance Lattice l => FilterData l Bool where
  removeData l (SecLabel (Lab l')) t = if label_leq l' l then t else undefined
instance (Lattice l, FilterData l a, FilterData l b)
          => FilterData l (Either a b) where
  removeData l (SecEither s1 s2 (Lab l')) (Left a) =
                                  if label_leq l' l
                                    then Left (removeData l s1 a)
                                    else undefined
  removeData l (SecEither s1 s2 (Lab l')) (Right b) =
                                  if label_leq l' l
                                    then Right (removeData l s2 b)
                                    else undefined

class ResetProject t where
  resetProject :: t -> t
  resetProject = id

instance ResetProject ()
instance ResetProject Int
instance ResetProject Integer
instance (ResetProject a, ResetProject b)
         => ResetProject (a,b) where
  resetProject (a,b) = (resetProject a, resetProject b)
instance ResetProject (SRef rs a) where
  resetProject (MkSRef rs a fread) = (MkSRef rs a id)
instance ResetProject [a]
instance ResetProject Word128
instance ResetProject Bool
instance (ResetProject a, ResetProject b)
         => ResetProject (Either a b) where
  resetProject (Left a) = Left (resetProject a)
  resetProject (Right b) = Right (resetProject b)
```

SecureFlow.hs

B.5 Unification.hs

Unification.hs

```
module Unification where

import Lattice
import Control.Monad.State

-- Unification constraints
data U s = Meet (U s) (U s)
         | Join (U s) (U s)
         | Decl (U s) (U s) (U s)
         | Tag (U s) (U s)
         | LExtr (U s)
         | Fst (U s)
         | Snd (U s)
```

```
          | Id s
    deriving (Eq, Show)

-- Unification monad
type CM l a = State (UEnv l) a

-- Unification set
data UEnv l = UEnv (WorkSet l) (ResultSet l)
    deriving (Show)

type WorkSet l = [UPair (U (SecType l))]
type ResultSet l = [UPair (SecType l)]

type UPair s = (s,s)

showPair s1 s2 = "(" ++ (show s1) ++ ", " ++ (show s2) ++ ")"

unify u = evalState unifyU (UEnv u [])

-- Unification of U
unifyU :: Lattice l => CM l (ResultSet l)
unifyU =
  do p <- getNextUPair
     case p of
       Nothing -> getResultSet
       (Just n) -> case n of
                     (Id s1, Id s2) -> do
                                       us <- unifySecType (s1,s2)
                                       updateUEnv (updateSecType us)
                                       addResultSet (iterateUpdate us)
                                       (UEnv w r) <- get
                                       unifyU
                     p -> do addWorkSet p
                             unifyU

iterateUpdate [] = []
iterateUpdate ((x1,x2):xs) =
           (updateSecType xs x1,updateSecType xs x2):(iterateUpdate xs)

-- update both WorkSet and ResultSet according to new uni-pairs
updateUEnv :: Lattice l => (SecType l -> SecType l) -> CM l ()
updateUEnv upd = do (UEnv w r) <- get
                    put (UEnv (updateUs upd w) (updateS upd r))

updateS _ [] = []
updateS upd ((x1,x2):xs) = (upd x1,upd x2):(updateS upd xs)

updateUs _ [] = []
updateUs upd ((x1,x2):xs) = (updateU upd x1, updateU upd x2):(updateUs upd xs)

updateU upd x = if hasVarU x’ then x’ else Id (simplifyU x’)
  where
  x’ = updateUType upd x

updateUType upd u = mapUType upd u

mapUType :: (SecType l -> SecType l) -> U (SecType l) -> U (SecType l)
mapUType f u =
  case u of
    Meet u1 u2 -> Meet (mapUType f u1) (mapUType f u2)
    Join u1 u2 -> Join (mapUType f u1) (mapUType f u2)
    Decl u1 u2 u3 -> Decl (mapUType f u1) (mapUType f u2) (mapUType f u3)
    Tag u1 u2 -> Tag (mapUType f u1) (mapUType f u2)
    LExtr u1 -> LExtr (mapUType f u1)
    Fst u -> Fst (mapUType f u)
    Snd u -> Snd (mapUType f u)
    Id s -> Id (f s)

-- Resolve unification constraints
simplifyU :: Lattice l => (U (SecType l)) -> (SecType l)
simplifyU u =
  case u of
    Meet u1 u2 -> min (simplifyU u1) (simplifyU u2)
    Join u1 u2 -> max (simplifyU u1) (simplifyU u2)
    Decl u1 u2 u3 -> ml_decl (simplifyU u1) (simplifyU u2)
                        ((unLabel.unSecLabel) (simplifyU u3))
    Tag u1 u2 -> ml_tag (simplifyU u1) ((unLabel.unSecLabel) (simplifyU u2))
    LExtr u1 -> SecLabel $ Lab (mextract (simplifyU u1))
    Fst u1 -> case (simplifyU u1) of
                (SecPair s1 s2) -> s1
    Snd u1 -> case (simplifyU u1) of
                (SecPair s1 s2) -> s2
    Id s -> s

unSecLabel (SecLabel l) = l
unLabel (Lab l) = l

hasVarU :: (U (SecType l)) -> Bool
hasVarU = propLeafU hasVarSecType (||)

propLeafU :: (SecType l -> a) -> (a -> a -> a) -> U (SecType l) -> a
propLeafU f con (Meet u1 u2) = (propLeafU f con u1) `con` (propLeafU f con u2)
propLeafU f con (Join u1 u2) = (propLeafU f con u1) `con` (propLeafU f con u2)
propLeafU f con (Decl u1 u2 u3) = (propLeafU f con u1) `con`
                                  (propLeafU f con u2) `con` (propLeafU f con u3)
propLeafU f con (Tag u1 u2) = (propLeafU f con u1) `con` (propLeafU f con u2)
propLeafU f con (LExtr u1) = propLeafU f con u1
propLeafU f con (Fst u1) = propLeafU f con u1
propLeafU f con (Snd u1) = propLeafU f con u1
propLeafU f con (Id s) = f s

addResultSet :: ResultSet l -> CM l ()
addResultSet ps = do (UEnv w r) <- get
                     put (UEnv w (ps++r))

addWorkSet :: UPair (U (SecType l)) -> CM l ()
addWorkSet p = do (UEnv w r) <- get
                  put (UEnv (w++[p]) r)

getNextUPair :: CM l (Maybe (UPair (U (SecType l))))
getNextUPair = do (UEnv w r) <- get
                  if length w == 0
                    then return Nothing
                    else do put (UEnv (tail w) r)
                            return (Just (head w))

getResultSet :: CM l (ResultSet l)
getResultSet = do (UEnv w r) <- get
                  return r

-- Unification of security labels
unifyLabel :: Lattice l => (UPair (Label l)) -> CM l [(UPair (Label l))]
unifyLabel p = case p of
                 (Lab l1,Lab l2)  ->
```

```
                      if l1 == l2
                        then return []
                        else error $ "unifyLabel : " ++ showPair l1 l2
                (t1@(LVar _),t2) -> return [(t1,t2)]
                (t1,t2@(LVar _)) -> unifyLabel (t2,t1)

-- Unification of SecType
unifySecType :: Lattice l => (UPair (SecType l)) -> CM l [(UPair (SecType l))]
unifySecType p =
  case p of
    (SecLabel l1, SecLabel l2) -> do ul <- unifyLabel (l1,l2)
                                     return $ expandMap ul
    (SecRef s1 l1, SecRef s2 l2) -> do ul <- unifyLabel (l1,l2)
                                       us1 <- return $ expandMap ul
                                       us2 <- unifySecType $ update us1 (s1,s2)
                                       return $ us1 ++ us2
    (SecPair s1 s2, SecPair s3 s4) -> do us1 <- unifySecType (s1,s3)
                                         us2 <- unifySecType $ update us1 (s2,s4)
                                         return $ us1 ++ us2
    (SecEither s1 s2 l1, SecEither s3 s4 l2) ->
                               do ul <- unifyLabel (l1,l2)
                                  us1 <- return $ expandMap ul
                                  us2 <- unifySecType $ update us1 (s1,s3)
                                  us3 <- unifySecType $
                                       ((update us2).(update us1)) (s2,s4)
                                  return $ us1 ++ us2 ++ us3
    (SecHigh, SecHigh) -> return []
    (v@(SecVar _), s) -> if v `appearIn` s
                           then error $ "unifySecType : " ++ (show v)
                                        ++ " appears in " ++ (show s)
                           else return [(v,s)]
    (s, v@(SecVar _)) -> unifySecType (v,s)
    (SecHigh, SecLabel l) -> do ul <- unifyLabel (Lab label_top,l)
                               return $ expandMap ul
    (SecHigh, SecPair s1 s2) -> do us1 <- unifySecType (SecHigh,s1)
                                  us2 <- unifySecType $ update us1 (SecHigh,s2)
                                  return $ us1 ++ us2
    (SecHigh, SecEither s1 s2 l) -> do ul <- unifyLabel (Lab label_top, l)
                                       us1 <- return $ expandMap ul
                                       us2 <- unifySecType $ update us1 (SecHigh,s1)
                                       us3 <- unifySecType $
                                              ((update us2).(update us1))
                                              (SecHigh,s2)
                                       return $ us1 ++ us2 ++ us3
    (s, SecHigh) -> unifySecType (SecHigh, s)
    (s1, s2) -> error $ "unifySecType : " ++ showPair s1 s2
  where
  expandMap = map expand
  expand (s1,s2) = (SecLabel s1,SecLabel s2)
  update u (s1,s2) = (updateSecType u s1,updateSecType u s2)
  appearIn v1 v2@(SecVar t) = v1 == v2
  appearIn v SecHigh = False
  appearIn v (SecLabel _) = False
  appearIn v s = propLeaf (appearIn v) (||) s

-- Update a SecType according to a list of unification pairs.
updateSecType :: Lattice l => [UPair (SecType l)] -> (SecType l) -> (SecType l)
updateSecType uset t = foldl updateSec t uset

updateSec :: Lattice l => (SecType l) -> (UPair (SecType l)) -> (SecType l)
updateSec t pat@(v1@(SecVar _),new) =
  case t of
```

```
    (SecRef s l) -> SecRef (updateSec s pat) l
    (SecPair s1 s2) -> SecPair (updateSec s1 pat) (updateSec s2 pat)
    (SecEither s1 s2 l) -> SecEither (updateSec s1 pat) (updateSec s2 pat) l
    v2@(SecVar _) -> if v1 == v2 then new else v2
    s -> s
updateSec t pat@(SecLabel v1@(LVar _),SecLabel new) =
  case t of
    s@(SecLabel l) -> if v1 == l then (SecLabel new) else s
    (SecRef s l) -> if v1 == l
                      then SecRef (updateSec s pat) new
                      else SecRef (updateSec s pat) l
    (SecPair s1 s2) -> SecPair (updateSec s1 pat) (updateSec s2 pat)
    (SecEither s1 s2 l) ->
        if v1 == l
          then SecEither (updateSec s1 pat) (updateSec s2 pat) new
          else SecEither (updateSec s1 pat) (updateSec s2 pat) l
    s -> s

hasVarSecType :: (SecType l) -> Bool
hasVarSecType = propLeaf isVarSecType (||)

isVarSecType (SecVar _) = True
isVarSecType (SecLabel (LVar _)) = True
isVarSecType _ = False
```
Unification.hs

B.6 Priv.hs

Priv.hs

```
module Priv where

data Privilege l = PR l
```
Priv.hs

B.7 FlowArrowRef.hs

FlowArrowRef.hs

```
module FlowArrowRef
( FlowArrowRef
, ArrowRef
, Privilege
, tagRef
, declassifyRef
, certifyRef
, certifyRefL
, equalA
, lowerA
, iterateA
, createRefA
, readRefA
, writeRefA
, forkRef
, atomicA
, runArrowRef
, fstPair
, sndPair
, skipRef
, randomRRef
, idRef
, nullRef
, pairRight
, pairLeft
)
where

import Data.List as List
import Unification
import Lattice
import RLattice
import Control.Arrow
import Data.IORef
import SecureFlow
import RefOp
import Control.Concurrent
import Control.Concurrent.STM
import Control.Monad.Fix
import Control.Monad.State
import System.Random
import Priv
import Control.Exception

data Constraint l =
    LEQ l l | USERGEQ l | IS l l | GUARD l l | GSLEQL l l | GLLEQS l l
    deriving (Eq, Show)

data Flow l = Trans l l
       deriving (Eq,Show)

-- Flow definition of (>>>)
flow_seq::Lattice l => Flow (SecType l) -> Flow (SecType l)
                       -> ( Flow (SecType l)
                          , [Constraint (SecType l)]
                          , [(U (SecType l),U (SecType l))])
flow_seq (Trans s1 s2) (Trans s3 s4)= (Trans s1 s4, c,u)
  where
  (c,u) = seqFlow s2 s3

seqFlow s2 s3 =
  case (s2,s3) of
    (t1@(SecVar _), t2) -> if (hasVarSecType t2)
                             then ([],[(Id t1,Id t2)])
                             else ([LEQ t1 t2], [])
    (SecHigh, t2) -> if (hasVarSecType t2)
                       then ([],[(Id SecHigh,Id t2)])
                       else ([LEQ SecHigh t2], [])
    (t1, t2@(SecVar _)) -> ([],[(Id t1, Id t2)])
    (SecLabel l1, SecLabel l2) -> seqFlowLabel l1 l2
    (SecRef t1 l1, SecRef t2 l2) -> let (cl,ul) = seqFlowLabel l1 l2 in
                                    if (hasVarSecType t2)
                                      then (cl, (Id t1,Id t2):ul)
                                      else ([LEQ t1 t2,LEQ t2 t1]++cl,ul)
    (SecPair t1 r1, SecPair t2 r2) -> let (ct,ut) = seqFlow t1 t2
                                          (cr,ur) = seqFlow r1 r2
                                      in (ct++cr,ut++ur)
    (SecEither t1 r1 l1, SecEither t2 r2 l2) -> let (cl,ul) = seqFlowLabel l1 l2
                                                    (ct,ut) = seqFlow t1 t2
                                                    (cr,ur) = seqFlow r1 r2
                                                in (cl++ct++cr,ul++ut++ur)
    (t1, SecHigh) -> ([LEQ t1 SecHigh], [])

seqFlowLabel l1 l2 =
  case (l1,l2) of
    (t1, t2@(LVar _)) -> ([],[(Id (SecLabel t1),Id (SecLabel t2))])
    (t1, t2@(Lab _)) -> ([LEQ (SecLabel t1) (SecLabel t2)], [])

-- Flow definition of first, second, and (***).
flow_pair :: Lattice l => Flow (SecType l) -> Flow (SecType l) -> Flow (SecType l)
flow_pair (Trans s1 s2) (Trans s3 s4) = Trans (SecPair s1 s3) (SecPair s2 s4)

-- Flow definition of (&&&).
flow_diverge :: Lattice l => Flow (SecType l) -> Flow (SecType l)
                             -> ( Flow (SecType l)
                                , [Constraint (SecType l)]
                                , [(U (SecType l), U (SecType l))])
flow_diverge (Trans s1 s2) (Trans s1' s2') =
  let (in_flow, cons, us) = meetInFlow s1 s1' in
  (Trans in_flow out_flow, cons, us)
  where
  out_flow = (SecPair s2 s2')

meetInFlow s1 s2 =
  case (s1,s2) of
    (t1@(SecVar _), t2@(SecVar _)) -> (t1, [], [(Id t1,Id t2)])
    (t1@(SecVar _), t2) -> if hasVarSecType t2
                             then (t1, [], [(Id t1,Id t2)])
                             else (t1, [LEQ t1 t2], [])
    (t1, t2@(SecVar _)) -> if hasVarSecType t1
                             then (t2, [], [(Id t1,Id t2)])
                             else (t2, [LEQ t2 t1], [])
    (SecHigh, t2) -> (t2, [], [])
    (t1, SecHigh) -> (t1, [], [])
    ((SecLabel l1), (SecLabel l2)) -> let (l' , cons, us) = meetInFlowLabel l1 l2
                                      in (SecLabel l', cons, us)
    ((SecRef s1 l1), (SecRef s2 l2)) -> let (l', lcons, lus) = meetInFlowLabel l1 l2
                                        in (SecRef s1 l', lcons, (Id s1,Id s2):lus)
    ((SecPair s1 t1), (SecPair s2 t2)) -> let (s', scons, sus) = meetInFlow s1 s2
                                              (t', tcons, tus) = meetInFlow t1 t2
                                          in (SecPair s' t', scons++tcons, sus++tus)
    ((SecEither s1 t1 l1), (SecEither s2 t2 l2)) ->
                        let (s', scons, sus) = meetInFlow s1 s2
                            (t', tcons, tus) = meetInFlow t1 t2
                            (l', lcons, lus) = meetInFlowLabel l1 l2
                        in (SecEither s' t' l', scons++tcons++lcons, sus++tus++lus)

meetInFlowLabel k1 k2 =
  case (k1,k2) of
    (l1@(LVar _), l2@(LVar _)) -> (l1, [], [(Id (SecLabel l1),Id (SecLabel l2))])
    (l1@(LVar _), l2) -> (l1, [LEQ (SecLabel l1) (SecLabel l2)], [])
    (l1, l2@(LVar _)) -> (l2, [LEQ (SecLabel l2) (SecLabel l1)], [])
    ((Lab l1), (Lab l2)) -> (Lab (label_meet l1 l2), [], [])

-- Flow definition of left, right, and (+++).
flow_either :: Lattice l => Flow (SecType l) -> Flow (SecType l) -> Flow (SecType l)
flow_either (Trans s1 s2) (Trans s3 s4) = Trans (SecEither s1 s3 (LVar "x0"))
                                                (SecEither s2 s4 (LVar "x0"))

-- Flow definition of (|||).
flow_converge :: Lattice l => Flow (SecType l) -> Flow (SecType l)
                              -> ( Flow (SecType l)
                                 , [(U (SecType l),U (SecType l))])
```

```
flow_converge (Trans s1 s2) (Trans s1' s2') =
  let l = (LVar "x0") in
  let s_out = (SecVar "x1") in
  (Trans (SecEither s1 s1' l) s_out
  ,[(Id s_out, Tag (Join (Id s2) (Id s2')) (Id (SecLabel l)))])

-- Make substitution pair for arrow computations.
make_sub :: (Lattice l) => FlowArrowRef l a b c -> FlowArrowRef l a b' c'
              -> [SecType l]
              -> ([((SecType l),(SecType l))],
                  [((SecType l), (SecType l))],
                  [((SecType l), (SecType l))])
make_sub fa1 fa2 nvars = (r1,r2,r3)
 where
  t1 = extract_arrow fa1
  t2 = extract_arrow fa2
  len1 = length t1
  len2 = length t2
  result = zipWith renameVar (t1 ++ t2 ++ nvars) [0..]
  (r1,rt) = splitAt len1 result
  (r2,r3) = splitAt len2 rt

renameVar v n = case v of
                 (SecVar _) -> (v,(SecVar ('a':show n)))
                 (SecLabel (LVar _)) -> (v,(SecLabel (LVar ('a':show n))))

make_sub_partial fa idf nvars = (r1,r2,r3)
  where
  t1 = extract_arrow fa
  t2 = extract_flow idf
  len1 = length t1
  len2 = length t2
  result = zipWith renameVar (t1 ++ t2 ++ nvars) [0..]
  (r1,rt) = splitAt len1 result
  (r2,r3) = splitAt len2 rt

-- Gather all variables in a FlowArrowRef computation.
extract_arrow :: Lattice l => FlowArrowRef l a b c -> [SecType l]
extract_arrow (FARef _ f _ _ u ) = total
 where
  fs = extract_flow f
  us = extract_uniset u
  total = (List.union us fs)

extract_flow :: Lattice l => Flow (SecType l) -> [(SecType l)]
extract_flow (Trans s1 s2) = (extractSecVar s1) `List.union` (extractSecVar s2)

extract_uniset :: Lattice l => [(U (SecType l),U (SecType l))] -> [(SecType l)]
extract_uniset u = foldl List.union []
                         (map (\(u1,u2) -> (extractUVar u1) `List.union`
                                           (extractUVar u2)) u)

extractUVar :: Lattice l => U (SecType l) -> [SecType l]
extractUVar = propLeafU extractSecVar List.union

extractSecVar :: Lattice l => SecType l -> [SecType l]
extractSecVar v@(SecVar _) = [v]
extractSecVar (SecHigh) = []
extractSecVar (SecLabel (Lab _)) = []
extractSecVar v@(SecLabel (LVar _)) = [v]
extractSecVar s = propLeaf extractSecVar List.union s
```

```haskell
-- Variable Replacement functions.
replaceSecType :: Lattice l => [(SecType l, SecType l)] -> SecType l -> SecType l
replaceSecType s v@(SecVar _) = findReplace v s
replaceSecType s v@(SecLabel (LVar _)) = findReplace v s
replaceSecType s t@(SecLabel l) = t
replaceSecType s (SecRef t l@(LVar _)) =
                       SecRef (replaceSecType s t)
                              (unSecLabel (findReplace (SecLabel l) s))
replaceSecType s (SecRef t l) = SecRef (replaceSecType s t) l
replaceSecType s (SecPair t1 t2) = SecPair (replaceSecType s t1)
                                           (replaceSecType s t2)
replaceSecType s (SecEither t1 t2 l@(LVar _)) =
               SecEither (replaceSecType s t1) (replaceSecType s t2)
                         (unSecLabel (findReplace (SecLabel l) s))
replaceSecType s (SecEither t1 t2 l) =
               SecEither (replaceSecType s t1) (replaceSecType s t2) l
replaceSecType s (SecHigh) = SecHigh

findReplace v2 [] = v2
findReplace v2 (x:xs) = if ((fst x) == v2) then snd x else findReplace v2 xs

replace_flow sub (Trans s1 s2) =
                        Trans (replaceSecType sub s1) (replaceSecType sub s2)

replace_cons :: Lattice l => [(SecType l, SecType l)] -> [Constraint (SecType l)]
                             -> [Constraint (SecType l)]
replace_cons sub = map (replace_con sub)

replace_con sub (LEQ s1 s2)    = LEQ (replaceSecType sub s1) (replaceSecType sub s2)
replace_con sub (USERGEQ s)    = USERGEQ (replaceSecType sub s)
replace_con sub (IS s1 s2)     = IS (replaceSecType sub s1) (replaceSecType sub s2)
replace_con sub (GUARD t1 t2)  = GUARD (replaceSecType sub t1) (replaceSecType sub t2)
replace_con sub (GSLEQL t1 t2) = GSLEQL (replaceSecType sub t1) (replaceSecType sub t2)
replace_con sub (GLLEQS t1 t2) = GLLEQS (replaceSecType sub t1) (replaceSecType sub t2)

replace_uniset sub [] = []
replace_uniset sub ((u1,u2):cs) =
              (replaceUType sub u1,replaceUType sub u2):replace_uniset sub cs

replace_pc :: Lattice l => [(SecType l, SecType l)] -> (SecType l) -> (SecType l)
replace_pc = replaceSecType

replaceUType :: Lattice l => [(SecType l, SecType l)] -> U (SecType l)
                             -> U (SecType l)
replaceUType sub u = mapUType (replaceSecType sub) u

idflow = Trans (SecVar "id0") (SecVar "id0")

-- FlowArrowRef
data FlowArrowRef l a b c =
     FARef { computation  :: ((SecType l) -> (SecType l)) -> a b c
           , flow         :: Flow (SecType l)
           , constraints  :: [Constraint (SecType l)]
           , pc           :: (SecType l)
           , uniset       :: [(U (SecType l), U (SecType l))]
           }

instance (Lattice l, Arrow a) =>
          Arrow (FlowArrowRef l a) where
  pure f = FARef { computation = (\upd -> pure f)
                 , flow = Trans (SecVar "x0") SecHigh
                 , constraints = []
                 , pc = (SecLabel (Lab label_top))
                 , uniset = []
                 }
  fa1@(FARef c1 f1 t1 pc1 u1) >>> fa2@(FARef c2 f2 t2 pc2 u2) =
     let pc' = (SecLabel (LVar "x0")) in
     let (sub1,sub2,[(_,pc'')]) = make_sub fa1 fa2 [pc'] in
     let (f,c,u) = flow_seq (replace_flow sub1 f1) (replace_flow sub2 f2) in
     FARef { computation = (\upd -> c1 (upd.(replaceSecType sub1)) >>>
                               c2 (upd.(replaceSecType sub2)))
           , flow = f
           , constraints = c ++ (replace_cons sub1 t1) ++ (replace_cons sub2 t2)
           , pc = pc''
           , uniset = u ++ [(Id pc'',Meet (Id (replace_pc sub1 pc1))
                                     (Id (replace_pc sub2 pc2)))] ++
                      (replace_uniset sub1 u1) ++ (replace_uniset sub2 u2)
           }
  first fa@(FARef c f t pc u) =
      let (sub1,sub2,_) = make_sub_partial fa idflow [] in
      let f' = flow_pair (replace_flow sub1 f) (replace_flow sub2 idflow) in
      FARef { computation = (\upd -> first (c (upd.(replaceSecType sub1))))
            , flow = f'
            , constraints = (replace_cons sub1 t)
            , pc = (replace_pc sub1 pc)
            , uniset = (replace_uniset sub1 u)
            }
  second fa@(FARef c f t pc u) =
      let (sub1,sub2,_) = make_sub_partial fa idflow [] in
      let f' = flow_pair (replace_flow sub2 idflow) (replace_flow sub1 f) in
      FARef { computation = (\upd -> second (c (upd.(replaceSecType sub1))))
            , flow = f'
            , constraints = (replace_cons sub1 t)
            , pc = (replace_pc sub1 pc)
            , uniset = (replace_uniset sub1 u)
            }
  fa1@(FARef c1 f1 t1 pc1 u1) *** fa2@(FARef c2 f2 t2 pc2 u2) =
      let pc' = (SecLabel (LVar "x0")) in
      let (sub1,sub2,[(_,pc'')]) = make_sub fa1 fa2 [pc'] in
      FARef { computation = (\upd -> c1 (upd.(replaceSecType sub1)) ***
                                c2 (upd.(replaceSecType sub2)))
            , flow = flow_pair (replace_flow sub1 f1) (replace_flow sub2 f2)
            , constraints = (replace_cons sub1 t1)++(replace_cons sub2 t2)
            , pc = pc''
            , uniset = ([(Id pc'',Meet (Id (replace_pc sub1 pc1))
                                   (Id (replace_pc sub2 pc2)))] ++
                       replace_uniset sub1 u1) ++ (replace_uniset sub2 u2)
            }
  fa1@(FARef c1 f1 t1 pc1 u1) &&& fa2@(FARef c2 f2 t2 pc2 u2) =
      let pc' = (SecLabel (LVar "x0")) in
      let (sub1, sub2,[(_,pc'')]) = make_sub fa1 fa2 [pc'] in
      let (f,c,u) = flow_diverge (replace_flow sub1 f1) (replace_flow sub2 f2) in
      FARef { computation = (\upd -> c1 (upd.(replaceSecType sub1)) &&&
                                c2 (upd.(replaceSecType sub2)))
            , flow = f
            , constraints = c++(replace_cons sub1 t1)++(replace_cons sub2 t2)
            , pc = pc''
            , uniset = u ++ [(Id pc'',Meet (Id (replace_pc sub1 pc1))
                                       (Id (replace_pc sub2 pc2)))]
                       ++(replace_uniset sub1 u1) ++ (replace_uniset sub2 u2)
            }

instance (Lattice l, ArrowChoice a, ArrowAtomic a) =>
    ArrowChoice (FlowArrowRef l a) where
```

```
left fa@(FARef c f t pc u) =
    let (sub1,sub2,_) = make_sub_partial fa idflow [] in
    let f' = flow_either (replace_flow sub1 f) (replace_flow sub2 idflow) in
    let (Trans s_in s_out) = f' in
    FARef { computation = (\upd -> if (mext_join (upd s_in)) `label_leq` label_bottom
                                    then left (c (upd.(replaceSecType sub1)))
                                    else beginAtomic >>>
                                         left (c (upd.(replaceSecType sub1))) >>>
                                         endAtomic )
          , flow = f'
          , constraints = (replace_cons sub1 t)
          , pc = (replace_pc sub1 pc)
          , uniset = (replace_uniset sub1 u)
          }
right fa@(FARef c f t pc u) =
    let (sub1,sub2,_) = make_sub_partial fa idflow [] in
    let f' = flow_either (replace_flow sub2 idflow) (replace_flow sub1 f) in
    let (Trans s_in s_out) = f' in
    FARef { computation = (\upd -> if (mext_join (upd s_in)) `label_leq` label_bottom
                                    then right (c (upd.(replaceSecType sub1)))
                                    else beginAtomic >>>
                                         right (c (upd.(replaceSecType sub1))) >>>
                                         endAtomic )
          , flow = f'
          , constraints = (replace_cons sub1 t)
          , pc = (replace_pc sub1 pc)
          , uniset = (replace_uniset sub1 u)
          }
fa1@(FARef c1 f1 t1 pc1 u1) +++ fa2@(FARef c2 f2 t2 pc2 u2) =
  let pc' = (SecLabel (LVar "x0")) in
  let (sub1,sub2,[(_,pc'')]) = make_sub fa1 fa2 [pc'] in
  let f' = flow_either (replace_flow sub1 f1) (replace_flow sub2 f2) in
  let (Trans s_in s_out ) = f'  in
    FARef { computation = (\upd -> if (mext_join (upd s_in)) `label_leq` label_bottom
                                    then (c1 (upd.(replaceSecType sub1)) +++
                                          c2 (upd.(replaceSecType sub2)))
                                    else beginAtomic >>>
                                         (c1 (upd.(replaceSecType sub1)) +++
                                          c2 (upd.(replaceSecType sub2)))
                                         >>> endAtomic)
          , flow = f'
          , constraints = [GSLEQL s_in pc''] ++ (replace_cons sub1 t1)
                          ++(replace_cons sub2 t2)
          , pc = pc''
          , uniset = [(Id pc'',Meet (Id (replace_pc sub1 pc1))
                                   (Id (replace_pc sub2 pc2)))]
                    ++ (replace_uniset sub1 u1) ++ (replace_uniset sub2 u2)
          }
fa1@(FARef c1 f1 t1 pc1 u1) ||| fa2@(FARef c2 f2 t2 pc2 u2) =
  let pc' = (SecLabel (LVar "x0")) in
  let l_out' = (SecLabel (LVar "x1")) in
  let (sub1,sub2,[(_,pc''),(_,l_out'')]) = make_sub fa1 fa2 [pc',l_out'] in
  let (f',u) = flow_converge (replace_flow sub1 f1) (replace_flow sub2 f2) in
  let (Trans s_in s_out ) = f'  in
    FARef { computation = (\upd -> if (mext_join (upd s_in)) `label_leq` label_bottom
                                    then (c1 (upd.(replaceSecType sub1)) |||
                                          c2 (upd.(replaceSecType sub2)))
                                    else beginAtomic >>>
                                         (c1 (upd.(replaceSecType sub1)) |||
                                          c2 (upd.(replaceSecType sub2)))
                                         >>> endAtomic)
              , flow = f'
```

```
            , constraints = [GSLEQL s_in pc'', GSLEQL s_in l_out''] ++
                            (replace_cons sub1 t1)++(replace_cons sub2 t2)
            , pc = pc''
            , uniset = u ++ [(Id pc'',Meet (Id (replace_pc sub1 pc1))
                                           (Id (replace_pc sub2 pc2)))
                       ,(Id l_out'',LExtr (Id s_out))] ++
                       (replace_uniset sub1 u1) ++ (replace_uniset sub2 u2)
            }

-- Non-standard combinators of FlowHaskellRef

-- Push up security types to level l.
tagRef :: (Lattice l, Arrow a)
          => l -> FlowArrowRef l a b b
tagRef l =
    let s_in = (SecVar "x0") in
    let s_out = (SecVar "x1") in
    FARef { computation = (\upd -> pure id)
          , flow = Trans s_in s_out
          , constraints = [LEQ s_in s_out]
          , pc = (SecLabel (Lab label_top))
          , uniset = [(Id s_out, Tag (Id s_in) (Id (SecLabel (Lab l))))]
          }

-- Declassify a computation.
declassifyRef :: ( Lattice l, Arrow a
                 , BuildSecType l b)
                 => l -> FlowArrowRef l a b b
declassifyRef l = declassifyRef' l (pure id)

declassifyRef' :: (Lattice l, Arrow a
                  ,BuildSecType l b
                  ,TakeOutputType l (FlowArrowRef l a) b b)
                  => l -> FlowArrowRef l a b b -> FlowArrowRef l a b b
declassifyRef' l fa@(FARef c (Trans s_in _) t pc u) =
   let s_out = (deriveSecType l fa) in
   let s_out' = (SecVar "x1") in
   FARef { computation = (\upd -> c upd )
         , flow = Trans s_in s_out'
         , constraints = [USERGEQ s_in]
         , pc = (SecLabel (Lab label_top))
         , uniset = [(Id s_out', Decl (Id s_in) (Id s_out) (Id (SecLabel (Lab l))))]
         }

-- Help functions for checking constraints in certifyRef.
check_levels label_in label_out (Trans s1 s2) =
  (label_in <= s1) && (s2 <= label_out)

check_levelsL label_in label_out (Trans s1 s2) =
  (label_in <= s1) && (s2 <= (ml_label label_out s2))

check_constraint p (LEQ s1 s2)= if s1 <= s2
                                then True
                                else error $ "LEQ\n"++(show s1)++"\n"++(show s2)
check_constraint p (USERGEQ s)= if label_leq (mext_join s) p
                                then True
                                else error $ "USERGEQ\n"++(show s)
check_constraint p (IS s1 s2) = if ml_eqstruct s1 s2
                                then True
                                else error $ "IS\n"++(show s1)++"\n"++(show s2)
check_constraint p (GUARD l s) = if label_guard l s
                                 then True
```

```
                                   else error $ "GUARD "++(show l) ++ "\n "++(show s)
check_constraint p (GSLEQL s l) = if label_gsleql s l
                                   then True
                                   else error $ "GSLEQL "++(show s)++"\n "++(show l)
check_constraint p (GLLEQS l s) = if label_glleqs l s
                                   then True
                                   else error $ "GLLEQS "++(show l)++"\n "++(show s)
check_constraints p t = all (check_constraint p) t

-- Help functions for resolving variables in (SecType l) .
resolve_flow :: (Lattice l, Show l) => [(SecType l,SecType l)] ->
                              Flow (SecType l) -> Flow (SecType l)
resolve_flow = replace_flow

resolve_cons :: (Lattice l, Show l) => [(SecType l,SecType l)] ->
               [Constraint (SecType l)] -> [Constraint (SecType l)]
resolve_cons = replace_cons

resolve_pc :: Lattice l => [(SecType l, SecType l)] -> (SecType l) -> l
resolve_pc sym s = (unLabel.unSecLabel) (replace_pc sym s)

-- Check and run FlowArrowRef computation.
certifyRef :: (Lattice l, Show (Constraint l))
              => (SecType l) -> (SecType l) -> Privilege l ->
                 FlowArrowRef l a b c -> a b c
certifyRef l_in l_out priv@(PR user_label) fa@(FARef c f t pc u) =
      let u' = (varIn f)++u in
      let sym = unify u' in
      let (f', t', pc') = (resolve_flow sym f,
                           resolve_cons sym t, resolve_pc sym pc) in
      if not $ check_levels l_in l_out f' then
        error $ "security level mismatch :\n" ++"input: "++(show l_in)++"\nflow: "
               ++ (show f') ++ "\noutput: " ++ (show l_out)
      else if not $ label_leq (mext_join l_in) pc' then
        error $ "pc level mismatch : " ++ (show pc')
      else if not $ (check_constraints user_label t') then
        error $ "Ref constraints cannot be met : " ++ (show t')
      else
        c (replaceSecType sym)
  where
  varIn (Trans s1 s2) = if hasVarSecType s1 then [(Id l_in, Id s1)] else []

certifyRefL :: (Lattice l, Show (Constraint l))
              => (SecType l) -> l -> Privilege l -> FlowArrowRef l a b c -> a b c
certifyRefL l_in l_out priv@(PR user_label) fa@(FARef c f t pc u) =
      let u' = (varIn f)++u in
      let sym = unify u' in
      let (f', t', pc') = (resolve_flow sym f,
                           resolve_cons sym t, resolve_pc sym pc) in
      if not $ check_levelsL l_in l_out f' then
        error $ "security level mismatch :\n"++"input: "++(show l_in)++"\nflow: "
               ++ (show f') ++ "\noutput: " ++ (show l_out)
      else if not $ label_leq (mext_join l_in) pc' then
        error $ "pc level mismatch : " ++ (show pc')
      else if not $ (check_constraints user_label t') then
        error $ "Ref constraints cannot be met : " ++ (show t')
      else
        c (replaceSecType sym)
  where
  varIn (Trans s1 s2) = if hasVarSecType s1 then [(Id l_in, Id s1)] else []

equalA :: (Lattice l, Arrow ar,
           BuildSecType l b,
           TakeOutputType l (FlowArrowRef l ar) a b)
          =>
           l -> FlowArrowRef l ar a b -> FlowArrowRef l ar a b
equalA level fa@(FARef com’ (Trans s_in’ _) cons’ pc’ uniset’) =
   let flow_out = (deriveSecType level fa) in
   FARef { computation = (\upd -> com’ upd)
         , flow = Trans s_in’ flow_out
         , constraints = [GLLEQS (SecLabel (Lab level)) s_in’,
                          GSLEQL s_in’ (SecLabel (Lab level)),
                          LEQ (SecLabel (Lab label_top)) pc’] ++ cons’
         , pc = (SecLabel (Lab label_top))
         , uniset = uniset’
         }

lowerA :: (Lattice l, Arrow ar, BuildSecType l b,
           FilterData l a, ResetProject b,
           TakeOutputType l (FlowArrowRef l ar) a b)
          => l -> FlowArrowRef l ar a b -> FlowArrowRef l ar a b
lowerA level fa@(FARef com’ (Trans s_in’ _) cons’ pc’ uniset’) =
   let flow_out = (deriveSecType level fa) in
   let inputFilter upd = pure (\i -> (removeData level (upd s_in’) i)) in
   let removeRead = pure (\i -> resetProject i) in
   FARef { computation = (\upd -> (inputFilter upd) >>> (com’ upd) >>>
                                  removeRead)
         , flow = Trans s_in’ flow_out
         , constraints = [LEQ (SecLabel (Lab label_top)) pc’] ++ cons’
         , pc = (SecLabel (Lab label_top))
         , uniset = uniset’
         }

iterateA :: Lattice l =>
            FlowArrowRef l ArrowRef a (a,Bool) -> FlowArrowRef l ArrowRef a a
iterateA (FARef com’ (Trans fin fout) cons’ pc’ uniset’) =
    let vout = (SecVar "x0") in
    let vbool = (SecVar "x1") in
    FARef { computation = (\upd -> iterateArrowRef (com’ upd))
          , flow = Trans fin vout
          , constraints = [LEQ vout fin, GUARD vbool vout] ++ cons’
          , pc = pc’
          , uniset = (Id vbool,LExtr (Snd (Id fout))):
                     (Id vout,Fst (Id fout)):uniset’
          }

-- Reference manipulation
createRefA :: (Lattice l, STSecType st l,
              BuildSecType l b) =>
             st -> l -> FlowArrowRef l ArrowRef b (SRef st b)
createRefA = createRefA’ (pure id)

createRefA’ :: (Lattice l, STSecType st l,
               BuildSecType l b,
               TakeOutputType l (FlowArrowRef l ArrowRef) b b) =>
              FlowArrowRef l ArrowRef b b -> st -> l ->
              FlowArrowRef l ArrowRef b (SRef st b)
createRefA’ far rs lr =
   let ld = (SecVar "x0") in
   let pc’ = SecLabel (LVar "x1") in
   let s_in = deriveSecType label_bottom far in
   FARef { computation = (\upd -> waitForYield >>>
                                  AR (\env -> do ref <- createMRef (dat env) rs
                                                 return env{dat=ref}) >>>
```

```
                                 yieldControl )
          , flow = Trans ld (SecRef (toSecType rs) (Lab lr))
          , constraints = [LEQ ld (toSecType rs), IS s_in (toSecType rs)]
          , pc = pc'
          , uniset = [(Id pc',LExtr (Id ld))]
          }

readRefA :: (Lattice l, STSecType st l)
            => FlowArrowRef l ArrowRef (SRef st b) b
readRefA =
   let ld = (SecVar "x0") in
   let lr = (LVar "x1") in
   let lv = (SecVar "x2") in
   FARef { computation = (\upd -> waitForYield >>>
                                 AR (\env -> do d <- readMRef (dat env)
                                                return env{dat=d}) >>>
                                 yieldControl)
          , flow = Trans (SecRef ld lr) lv
          , constraints = []
          , pc = SecLabel (Lab label_top)
          , uniset = [(Id lv, Tag (Id ld) (Id (SecLabel lr)))]
          }

writeRefA :: (Lattice l, STSecType st l)
            => FlowArrowRef l ArrowRef (SRef st b, b) ()
writeRefA =
    let ld = (SecVar "x0") in
    let lr = (LVar "x1") in
    let pc' = (SecLabel (LVar "x2")) in
        FARef { computation = (\upd -> waitForYield >>>
                                  AR (\env -> do
                                                 writeMRef (fst (dat env))
                                                           (snd (dat env))
                                                 return env{dat=()}) >>>
                                  yieldControl )
           , flow = Trans (SecPair (SecRef ld lr) ld) (SecLabel (Lab label_bottom))
           , constraints = [GUARD (SecLabel lr) ld]
           , pc = pc'
           , uniset = [(Id pc',LExtr (Id ld))]
           }

-- The following primitives are syntactic sugers.
fstPair :: (Lattice l) => FlowArrowRef l ArrowRef (a,b) a
fstPair =
    let s1 = (SecVar "x0") in
    let s2 = (SecVar "x1") in
    FARef { computation = (\upd -> waitForYield >>>
                                  AR (\env -> do
                                                 d <- return $ fst (dat env)
                                                 return env{dat=d}) >>>
                                  yieldControl)
           , flow = Trans (SecPair s1 s2) s1
           , constraints = []
           , pc = SecLabel (Lab label_top)
           , uniset = []
           }

sndPair :: (Lattice l) => FlowArrowRef l ArrowRef (a,b) b
sndPair =
    let s1 = (SecVar "x0") in
    let s2 = (SecVar "x1") in
    FARef { computation = (\upd -> waitForYield >>>
                                   AR (\env -> do
                                                 d <- return $ snd (dat env)
                                                 return env{dat=d}) >>>
                                   yieldControl)
           , flow = Trans (SecPair s1 s2) s2
           , constraints = []
           , pc = SecLabel (Lab label_top)
           , uniset = []
           }

-- Id function to pass data.
idRef :: (Lattice l, Arrow a)
         => FlowArrowRef l a b b
idRef =
   let s_in = (SecVar "x0") in
   FARef { computation = (\upd -> pure id)
         , flow = Trans s_in s_in
         , constraints = []
         , pc = (SecLabel (Lab label_top))
         , uniset = []
         }

-- SecPair right ((a,b),c) -> (a,(b,c)).
pairRight :: (Lattice l, Arrow a)
             => FlowArrowRef l a ((b,c),d) (b,(c,d))
pairRight =
   let s1 = (SecVar "x0") in
   let s2 = (SecVar "x1") in
   let s3 = (SecVar "x2") in
   FARef { computation = (\upd -> pure (\((a,b),c) -> (a,(b,c))))
         , flow = Trans (SecPair (SecPair s1 s2) s3) (SecPair s1 (SecPair s2 s3))
         , constraints = []
         , pc = (SecLabel (Lab label_top))
         , uniset = []
         }

-- SecPair left (a,(b,c)) -> ((a,b),c).
pairLeft :: (Lattice l, Arrow a)
             => FlowArrowRef l a (b,(c,d)) ((b,c),d)
pairLeft =
   let s1 = (SecVar "x0") in
   let s2 = (SecVar "x1") in
   let s3 = (SecVar "x2") in
   FARef { computation = (\upd -> pure (\(a,(b,c)) -> ((a,b),c)))
         , flow = Trans (SecPair s1 (SecPair s2 s3)) (SecPair (SecPair s1 s2) s3)
         , constraints = []
         , pc = (SecLabel (Lab label_top))
         , uniset = []
         }

-- Return a unit.
nullRef :: (Lattice l, Arrow a)
           => FlowArrowRef l a b ()
nullRef =
   let s_in = (SecVar "x0") in
   FARef { computation = (\upd -> pure (\_ -> ()))
         , flow = Trans s_in (SecLabel (Lab label_bottom))
         , constraints = []
         , pc = (SecLabel (Lab label_top))
         , uniset = []
         }
```

```
-- The run-time system for multithreaded information flow.

data RRobin a = RRobin {
      dat       :: a,                -- real compuatation data
      blocks    :: Int,              -- nested protect count
      queue     :: TVar [ThreadId], -- round-robin thread queue
      iD        :: ThreadId          -- identify of the thread
   }

data ArrowRef a b = AR ((RRobin a) -> IO (RRobin b))

unAR (AR f) = f

runArrowRef (AR f) a = do myid <- myThreadId
                          q <- newThreadQueue myid
                          env' <- f (RRobin a 0 q myid)
                          return (dat env')

newOrder = atomically $ newTVar []

instance Arrow ArrowRef where
  pure f = waitForYield >>>
           AR (\env -> do b <- return $ f (dat env)
                          return env{ dat = b } ) >>>
           yieldControl

  (AR f) >>> (AR g) =
     AR (\env -> f env >>= g)

  first (AR f) =
     AR (\env -> let (b,d) = (dat env) in
                 f (env{dat = b}) >>=
                 (\env' -> return env'{dat = (dat env',d)}) )

instance Arrow ArrowRef => ArrowChoice ArrowRef where
  left (AR f) = AR (\env -> case (dat env) of
                              Left d  -> (f env{ dat = d }) >>=
                                         (\env' -> return env'{dat = Left (dat env')})
                              Right d -> return env{dat = (Right d)} )

iterateArrowRef (AR f) = AR (\env -> let ite = (\(env,b) ->
                                        if b
                                          then do env' <- f env
                                                  b' <- return (snd (dat env'))
                                                  ite (env'{dat = fst (dat env')},b')
                                          else return env)
                                     in
                                     ite (env,True))

class Arrow a => ArrowAtomic a where
  beginAtomic :: a b b
  endAtomic :: a b b

instance ArrowAtomic ArrowRef where
  beginAtomic = waitForYield >>>
          AR (\env -> return env{blocks = ((blocks env)+1)} )

  endAtomic = AR (\env -> if (blocks env) <= 0
                          then error "beginAtomic and endAtomic is not well paired."
                          else return env{blocks = ((blocks env)-1)} )
           >>> yieldControl
```

```
newLockRef :: IO (TVar Bool)
newLockRef = atomically $ newTVar True

newThreadQueue :: ThreadId -> IO (TVar [ThreadId])
newThreadQueue myid = atomically $ newTVar [myid]

waitForYield :: ArrowRef a a
waitForYield = AR (\env -> do waitTurn env
                              return env)

yieldControl :: ArrowRef a a
yieldControl = AR (\env -> do nextTurn env
                              return env)

waitTurn :: RRobin a -> IO ()
waitTurn env = if (blocks env) > 0
             then return ()
             else atomically $
                  do que <- readTVar (queue env)
                     if head que /= (iD env)
                       then retry
                       else return ()

nextTurn :: RRobin a -> IO ()
nextTurn env = if (blocks env) > 0
               then return ()
               else atomically $
                    do que <- readTVar (queue env)
                       if head que /= (iD env)
                         then error "nextTurn and waitTurn inconsistent."
                         else do writeTVar (queue env) ((tail que)++[head que])
                                 return ()

skip :: RRobin Int -> IO ()
skip env = if (dat env) <= 0
             then return ()
             else do waitTurn env
                     i' <- return ((dat env)-1)
                     nextTurn env
                     skip env{dat=i'}

skipRef :: (Lattice l) => FlowArrowRef l ArrowRef Int ()
skipRef =
   let s_in = (SecVar "x0") in
   FARef { computation = (\upd -> AR(\env -> do skip env
                                                return env{dat=()} ))
         , flow = Trans s_in (SecLabel (Lab label_bottom))
         , constraints = []
         , pc = SecLabel (Lab label_top)
         , uniset = []
         }

atomicA :: Lattice l =>
            FlowArrowRef l ArrowRef a b -> FlowArrowRef l ArrowRef a b
atomicA (FARef com' flow' cons' pc' uniset') =
    FARef { computation = (\upd ->  beginAtomic >>> (com' upd) >>> endAtomic)
          , flow = flow'
          , constraints = cons'
          , pc = pc'
          , uniset = uniset'
          }
```

```
forkRef :: (Lattice l) =>
           FlowArrowRef l ArrowRef a b -> FlowArrowRef l ArrowRef a ()
forkRef (FARef com' (Trans f_in f_out) cons' pc' uniset') =
   FARef { computation = (\upd -> let t = unAR (initThread >>> (com' upd) >>>
                                                    endThread ) in
                                  waitForYield >>>
                                  AR (\env ->
                                       do env' <- return env{blocks = 0}
                                          nid <-forkIO (((t env') >>= nullThread)
                                                       `Control.Exception.catch`
                                                        (finalThread env'))
                                          newThread env nid
                                          return env{dat=()}) >>>
                                  yieldControl )
         , flow = Trans f_in (SecLabel (Lab label_bottom))
         , constraints = cons'
         , pc = pc'
         , uniset = uniset'
         }
  where
  nullThread = (\_ -> return ())

newThread :: RRobin a -> ThreadId -> IO (RRobin a)
newThread env nid = do atomically $ do que <- readTVar (queue env)
                                       writeTVar (queue env) (que++[nid])
                       return env

initThread :: ArrowRef a a
initThread = AR (\env -> do myId <- myThreadId
                            return env{iD = myId} )

endThread :: ArrowRef a a
endThread = AR (\env -> do atomically $
                            do que <- readTVar (queue env)
                               writeTVar (queue env) (filter ((iD env)/=) que)
                               return que
                           return env )

finalThread :: RRobin a -> Exception -> IO ()
finalThread env = (\e -> do
                            q <- atomically $ readTVar (queue env)
                            mapM killThread q
                            return () )

-- Generate a random number in range (l,h).
randomRRef :: (Lattice l) => Int -> Int -> FlowArrowRef l ArrowRef Int Int
randomRRef l h =
   let s_in = (SecVar "x0") in
   FARef { computation = (\upd -> waitForYield >>>
                                  AR (\env -> do r <- randomRIO (l,h)
                                                 return env{dat=r}) >>>
                                  yieldControl )
         , flow = Trans s_in s_in
         , constraints = []
         , pc = SecLabel (Lab label_top)
         , uniset = []
         }
```
FlowArrowRef.hs

B.8 BankLattice.hs

BankLattice.hs

```
module BankLattice where

import Lattice
import RLattice

data BankLabel = CLIENT | BANK | ATM_BANK | BANK_ATM | TOP | BOTTOM
  deriving (Eq, Show)

instance Lattice BankLabel where
  label_top = TOP
  label_bottom = BOTTOM

  label_leq BOTTOM _ = True
  label_leq _ TOP = True
  label_leq CLIENT CLIENT = True
  label_leq ATM_BANK ATM_BANK = True
  label_leq ATM_BANK BANK = True
  label_leq BANK_ATM BANK_ATM = True
  label_leq BANK_ATM BANK = True
  label_leq BANK BANK = True
  label_leq _ _ = False

  label_join TOP _ = TOP
  label_join _ TOP = TOP
  label_join BOTTOM y = y
  label_join x BOTTOM = x
  label_join CLIENT BANK = TOP
  label_join BANK CLIENT = TOP
  label_join CLIENT ATM_BANK = TOP
  label_join ATM_BANK CLIENT = TOP
  label_join CLIENT BANK_ATM = TOP
  label_join BANK_ATM CLIENT = TOP
  label_join BANK ATM_BANK = BANK
  label_join ATM_BANK BANK = BANK
  label_join BANK BANK_ATM = BANK
  label_join BANK_ATM BANK = BANK
  label_join ATM_BANK BANK_ATM = BANK
  label_join BANK_ATM ATM_BANK = BANK
  label_join CLIENT CLIENT = CLIENT
  label_join BANK BANK = BANK
  label_join ATM_BANK ATM_BANK = ATM_BANK
  label_join BANK_ATM BANK_ATM = BANK_ATM

  label_meet BOTTOM _ = BOTTOM
  label_meet _ BOTTOM = BOTTOM
  label_meet TOP y = y
  label_meet x TOP = x
  label_meet CLIENT BANK = BOTTOM
  label_meet BANK CLIENT = BOTTOM
  label_meet CLIENT ATM_BANK = BOTTOM
  label_meet ATM_BANK CLIENT = BOTTOM
  label_meet CLIENT BANK_ATM = BOTTOM
  label_meet BANK_ATM CLIENT = BOTTOM
  label_meet BANK ATM_BANK = ATM_BANK
  label_meet ATM_BANK BANK = ATM_BANK
  label_meet BANK BANK_ATM = BANK_ATM
  label_meet BANK_ATM BANK = BANK_ATM
  label_meet ATM_BANK BANK_ATM = BOTTOM
  label_meet BANK_ATM ATM_BANK = BOTTOM
  label_meet CLIENT CLIENT = CLIENT
  label_meet BANK BANK = BANK
  label_meet ATM_BANK ATM_BANK = ATM_BANK
  label_meet BANK_ATM BANK_ATM = BANK_ATM

-- Singleton types for BankLabel
data SClient   = VClient   deriving Show
data SBank     = VBank     deriving Show
data SAtm_bank = VAtm_bank deriving Show
data SBank_atm = VBank_atm deriving Show
data STop      = VTop      deriving Show
data SBottom   = VBottom   deriving Show

instance STLabel SClient BankLabel where
  toLabel _ = CLIENT
instance STLabel SBank BankLabel where
  toLabel _ = BANK
instance STLabel SAtm_bank BankLabel where
  toLabel _ = ATM_BANK
instance STLabel SBank_atm BankLabel where
  toLabel _ = BANK_ATM
instance STLabel STop BankLabel where
  toLabel _ = TOP
instance STLabel SBottom BankLabel where
  toLabel _ = BOTTOM
```
BankLattice.hs

B.9 BankKey.hs

BankKey.hs

```
module BankKey where

import FlowArrowRef
import BankLattice
import Control.Arrow
import Codec.Utils
import Data.LargeWord

type Protected c = FlowArrowRef BankLabel ArrowRef () c

clientID = 99 :: Int

clientPriKey :: Protected [Octet]
clientPriKey = tagRef CLIENT >>> lowerA CLIENT (pure (\() -> cPriKey))
  where
    cPriKey :: [Octet]
    cPriKey =
       [0xa5, 0xda, 0xfc, 0x53, 0x41, 0xfa, 0xf2,
        0x89, 0xc4, 0xb9, 0x88, 0xdb, 0x30, 0xc1, 0xcd,
        0xf8, 0x3f, 0x31, 0x25, 0x1e, 0x06, 0x68, 0xb4,
        0x27, 0x84, 0x81, 0x38, 0x01, 0x57, 0x96, 0x41,
        0xb2, 0x94, 0x10, 0xb3, 0xc7, 0x99, 0x8d, 0x6b,
        0xc4, 0x65, 0x74, 0x5e, 0x5c, 0x39, 0x26, 0x69,
        0xd6, 0x87, 0x0d, 0xa2, 0xc0, 0x82, 0xa9, 0x39,
        0xe3, 0x7f, 0xdc, 0xb8, 0x2e, 0xc9, 0x3e, 0xda,
        0xc9, 0x7f, 0xf3, 0xad, 0x59, 0x50, 0xac, 0xcf,
        0xbc, 0x11, 0x1c, 0x76, 0xf1, 0xa9, 0x52, 0x94,
        0x44, 0xe5, 0x6a, 0xaf, 0x68, 0xc5, 0x6c, 0x09,
        0x2c, 0xd3, 0x8d, 0xc3, 0xbe, 0xf5, 0xd2, 0x0a,
        0x93, 0x99, 0x26, 0xed, 0x4f, 0x74, 0xa1, 0x3e,
        0xdd, 0xfb, 0xe1, 0xa1, 0xce, 0xcc, 0x48, 0x94,
        0xaf, 0x94, 0x28, 0xc2, 0xb7, 0xb8, 0x88, 0x3f,
        0xe4, 0x46, 0x3a, 0x4b, 0xc8, 0x5b, 0x1c, 0xb3,
        0xc1]

clientPubKey :: [Octet]
clientPubKey = [0x11]

clientModulus :: [Octet]
clientModulus =
   [0xbb, 0xf8, 0x2f, 0x09, 0x06, 0x82, 0xce, 0x9c,
    0x23, 0x38, 0xac, 0x2b, 0x9d, 0xa8, 0x71, 0xf7,
    0x36, 0x8d, 0x07, 0xee, 0xd4, 0x10, 0x43, 0xa4,
    0x40, 0xd6, 0xb6, 0xf0, 0x74, 0x54, 0xf5, 0x1f,
    0xb8, 0xdf, 0xba, 0xaf, 0x03, 0x5c, 0x02, 0xab,
    0x61, 0xea, 0x48, 0xce, 0xeb, 0x6f, 0xcd, 0x48,
    0x76, 0xed, 0x52, 0x0d, 0x60, 0xe1, 0xec, 0x46,
    0x19, 0x71, 0x9d, 0x8a, 0x5b, 0x8b, 0x80, 0x7f,
    0xaf, 0xb8, 0xe0, 0xa3, 0xdf, 0xc7, 0x37, 0x72,
    0x3e, 0xe6, 0xb4, 0xb7, 0xd9, 0x3a, 0x25, 0x84,
    0xee, 0x6a, 0x64, 0x9d, 0x06, 0x09, 0x53, 0x74,
    0x88, 0x34, 0xb2, 0x45, 0x45, 0x98, 0x39, 0x4e,
    0xe0, 0xaa, 0xb1, 0x2d, 0x7b, 0x61, 0xa5, 0x1f,
    0x52, 0x7a, 0x9a, 0x41, 0xf6, 0xc1, 0x68, 0x7f,
    0xe2, 0x53, 0x72, 0x98, 0xca, 0x2a, 0x8f, 0x59,
    0x46, 0xf8, 0xe5, 0xfd, 0x09, 0x1d, 0xbd, 0xcb]

bankPriKey :: Protected [Octet]
bankPriKey = tagRef BANK >>> lowerA BANK (pure (\() -> cPriKey))
  where
    cPriKey :: [Octet]
    cPriKey =
       [0xa5, 0xda, 0xfc, 0x53, 0x41, 0xfa, 0xf2,
        0x89, 0xc4, 0xb9, 0x88, 0xdb, 0x30, 0xc1, 0xcd,
        0xf8, 0x3f, 0x31, 0x25, 0x1e, 0x06, 0x68, 0xb4,
        0x27, 0x84, 0x81, 0x38, 0x01, 0x57, 0x96, 0x41,
        0xb2, 0x94, 0x10, 0xb3, 0xc7, 0x99, 0x8d, 0x6b,
        0xc4, 0x65, 0x74, 0x5e, 0x5c, 0x39, 0x26, 0x69,
        0xd6, 0x87, 0x0d, 0xa2, 0xc0, 0x82, 0xa9, 0x39,
        0xe3, 0x7f, 0xdc, 0xb8, 0x2e, 0xc9, 0x3e, 0xda,
        0xc9, 0x7f, 0xf3, 0xad, 0x59, 0x50, 0xac, 0xcf,
        0xbc, 0x11, 0x1c, 0x76, 0xf1, 0xa9, 0x52, 0x94,
        0x44, 0xe5, 0x6a, 0xaf, 0x68, 0xc5, 0x6c, 0x09,
        0x2c, 0xd3, 0x8d, 0xc3, 0xbe, 0xf5, 0xd2, 0x0a,
        0x93, 0x99, 0x26, 0xed, 0x4f, 0x74, 0xa1, 0x3e,
        0xdd, 0xfb, 0xe1, 0xa1, 0xce, 0xcc, 0x48, 0x94,
        0xaf, 0x94, 0x28, 0xc2, 0xb7, 0xb8, 0x88, 0x3f,
        0xe4, 0x46, 0x3a, 0x4b, 0xc8, 0x5b, 0x1c, 0xb3,
        0xc1]

bankPubKey :: [Octet]
bankPubKey = [0x11]

bankModulus :: [Octet]
bankModulus =
   [0xbb, 0xf8, 0x2f, 0x09, 0x06, 0x82, 0xce, 0x9c,
    0x23, 0x38, 0xac, 0x2b, 0x9d, 0xa8, 0x71, 0xf7,
    0x36, 0x8d, 0x07, 0xee, 0xd4, 0x10, 0x43, 0xa4,
    0x40, 0xd6, 0xb6, 0xf0, 0x74, 0x54, 0xf5, 0x1f,
    0xb8, 0xdf, 0xba, 0xaf, 0x03, 0x5c, 0x02, 0xab,
    0x61, 0xea, 0x48, 0xce, 0xeb, 0x6f, 0xcd, 0x48,
    0x76, 0xed, 0x52, 0x0d, 0x60, 0xe1, 0xec, 0x46,
    0x19, 0x71, 0x9d, 0x8a, 0x5b, 0x8b, 0x80, 0x7f,
    0xaf, 0xb8, 0xe0, 0xa3, 0xdf, 0xc7, 0x37, 0x72,
    0x3e, 0xe6, 0xb4, 0xb7, 0xd9, 0x3a, 0x25, 0x84,
    0xee, 0x6a, 0x64, 0x9d, 0x06, 0x09, 0x53, 0x74,
    0x88, 0x34, 0xb2, 0x45, 0x45, 0x98, 0x39, 0x4e,
    0xe0, 0xaa, 0xb1, 0x2d, 0x7b, 0x61, 0xa5, 0x1f,
    0x52, 0x7a, 0x9a, 0x41, 0xf6, 0xc1, 0x68, 0x7f,
    0xe2, 0x53, 0x72, 0x98, 0xca, 0x2a, 0x8f, 0x59,
    0x46, 0xf8, 0xe5, 0xfd, 0x09, 0x1d, 0xbd, 0xcb]

-- Initial vector used in cbc scheme
sessionIV  = 0x3dafba429d9eb430b422da802c9fac41 :: Word128
```

BankKey.hs

B.10 BankSystem.hs

BankSystem.hs

```
module BankSystem where

import BankLattice
import BankKey
import FlowArrowRef
import Lattice
import RLattice
import Control.Arrow
import SecureFlow
import RefOp
import System.Random
import Codec.Utils
import Data.LargeWord
import Codec.Encryption.RSA as RSA
import Codec.Encryption.AES as AES
import Codec.Encryption.Modes
import Codec.Encryption.Padding

-- Certify function
expose :: (Privilege BankLabel) -> SecType BankLabel -> Protected c -> IO c
expose p r t = runArrowRef (certifyRef (SecLabel (Lab label_bottom))
                                       (ml_label label_bottom r)
                                       p t) ()

exposeL :: (Privilege BankLabel) -> Protected c -> IO c
exposeL p t = runArrowRef (certifyRefL (SecLabel (Lab label_bottom))
                                       label_bottom p t) ()

-----------------
-- Client Card --
-----------------
-- Get account id
client_getAccountId :: IO Int
client_getAccountId = return clientID

---------
-- ATM --
---------
-- Generate authentication request
atm_authRequest :: Protected (SRef (SSecLabel SAtm_bank) Int) ->
                   (Privilege BankLabel) -> IO [Octet]
atm_authRequest atm_nonce priv =
  do id <- client_getAccountId
```

```
     atm_getNewNonce priv atm_nonce
     req <- atm_buildAuthRequest id atm_nonce
     atm_RSAencrypt priv req (bankModulus,bankPubKey)

-- Generate transaction request
atm_tranRequest :: [Octet] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                   -> Protected (SRef (SSecLabel SBank_atm) Int)
                   -> Protected (SRef (SSecLabel SBank) Word128)
                   -> (String -> IO (Maybe (Protected ([Octet],[Octet]))))
                   -> (Privilege BankLabel)
                   -> IO [Word128]
atm_tranRequest res atm_nonce bank_nonce sessionKey client_prikey priv =
  do (ok,pw) <- atm_AuthResDecrypt res atm_nonce bank_nonce sessionKey
                                   client_prikey priv
     if not ok
       then error "ATM nonce is not consistent."
       else do putStrLn "Authentication response received and correct."
               atm_getNewNonce priv atm_nonce
               tran <- atm_buildTransaction atm_nonce
               sig <- atm_getSignature tran client_prikey pw priv
               atm_buildTranRequest tran sig atm_nonce bank_nonce sessionKey priv

-- End of transaction
atm_resultProcess :: [Word128] -> IO ()
atm_resultProcess r = putStrLn "Transaction response received.\nThank you."

-- Generate new atm nonce
atm_getNewNonce priv = getNewNonce ATM_BANK priv

atm_buildAuthRequest :: Int -> Protected (SRef (SSecLabel SAtm_bank) Int)->
                        IO (Protected [Octet])
atm_buildAuthRequest id atm_nonce =
  do let req = atm_nonce >>> readRefA >>>
               lowerA ATM_BANK (pure (\r -> (toOctets 256 id)++(toOctets 256 r)))
     return req

-- Use RSA and public key of bank to encrypt request and release.
atm_RSAencrypt = encryptRSA ATM_BANK

-- Decrypt authentication response with private key of client card.
atm_AuthResDecrypt :: [Octet] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                      -> Protected (SRef (SSecLabel SBank_atm) Int)
                      -> Protected (SRef (SSecLabel SBank) Word128)
                      -> (String -> IO (Maybe (Protected ([Octet],[Octet]))))
                      -> (Privilege BankLabel)
                      -> IO (Bool, Protected String)
atm_AuthResDecrypt res atm_nonce bank_nonce sessionKey client_prikey priv=
  do putStr "Please enter password : "
     pw <- getLine
     cpri <- client_prikey pw
     case cpri of
       Nothing   -> do putStrLn "Password is not correct."
                       atm_AuthResDecrypt res atm_nonce bank_nonce sessionKey
                                          client_prikey priv
       (Just pk) ->
         do
         let dres = pk >>>
                    lowerA BANK (pure (\key -> RSA.decrypt key res)) >>>
                    declassifyRef BANK_ATM >>>
                    lowerA BANK_ATM (pure (\p -> drop (128-18) p)) >>>
                    lowerA BANK_ATM (pure (\p ->
                           (fromOctets 256 (take 16 p),(drop 16 p)))) >>>
                    ( (atm_StoreRef BANK sessionKey >>> tagRef BANK)
                      ***
                      ( lowerA BANK_ATM (pure (\p -> (fromOctets 256 (take 1 p),
                                                      fromOctets 256 (drop 1 p)))) >>>
                        ( ( ((lowerA BOTTOM (pure (\_ -> () )) >>>
                              atm_nonce >>> readRefA  )
                             &&& idRef ) >>>
                            -- The following code guarantee no mix of
                            -- local and remote atm nonce.
                            ((tagRef ATM_BANK) *** (tagRef BANK_ATM)) >>>
                            lowerA BANK (pure (\(a,b) -> if a == b
                                                           then True else False))
                          )
                          *** (atm_StoreRef BANK_ATM bank_nonce >>> tagRef BANK)
                        ) >>> fstPair
                      )
                    ) >>> sndPair >>>
                    declassifyRef BOTTOM
         v <- expose priv (SecLabel (Lab BOTTOM)) dres
         let ppw = lowerA BANK (pure (\() -> pw))
         return (v,ppw)

atm_StoreRef :: forall a rs.
                (FilterData BankLabel a,
                 STSecType rs BankLabel,
                 ResetProject a,
                 BuildSecType BankLabel a) =>
                 BankLabel -> Protected (SRef rs a) ->
                 FlowArrowRef BankLabel ArrowRef a ()
atm_StoreRef l ref = ((lowerA l (pure (\_ -> ())) >>> ref) &&&
                      (lowerA l (pure id)) ) >>>
                     writeRefA

atm_buildTransaction :: Protected (SRef (SSecLabel SAtm_bank) Int) ->
                        IO (Protected [Octet])
atm_buildTransaction atm_nonce =
  do id <- client_getAccountId
     putStr "Please choose action([1] deposite [2] withdraw) : "
     ac <- getLine
     action <- return (read ac)
     putStr "Amount(0~255) : "
     am <- getLine
     amount <- return (read am)
     let ptran = atm_nonce >>> readRefA >>>
                 tagRef BANK >>>
                 lowerA BANK (pure (\n -> (toOctets 256 id) ++ (toOctets 256 action) ++
                                         (toOctets 256 amount) ++ (toOctets 256 n) ))
     return ptran

atm_getSignature :: Protected [Octet]
                    -> (String -> IO (Maybe (Protected ([Octet],[Octet]))))
                    -> Protected String -> Privilege BankLabel
                    -> IO (Protected [Octet])
atm_getSignature tran client_prikey ppw priv =
  do let password = ppw >>> declassifyRef BOTTOM
     pw <- expose priv (SecLabel (Lab BOTTOM)) password
     cpri <- client_prikey pw
     case cpri of
       Nothing   -> error "password is wrong!"
       (Just cp) -> do
                    let sig = (cp &&& tran) >>>
                              lowerA BANK (pure (\(key,t) -> RSA.encrypt key t))
                    return sig

atm_buildTranRequest :: Protected [Octet] -> Protected [Octet]
                        -> Protected (SRef (SSecLabel SAtm_bank) Int)
                        -> Protected (SRef (SSecLabel SBank_atm) Int)
                        -> Protected (SRef (SSecLabel SBank) Word128)
                        -> (Privilege BankLabel)
                        -> IO [Word128]
atm_buildTranRequest ptran psig atm_nonce bank_nonce sessionKey priv =
  do let tran = ( (ptran &&& psig)
                   &&&
                  ((atm_nonce >>> readRefA >>>
                    tagRef BANK)
                   &&&
                   (bank_nonce >>> readRefA >>>
                    tagRef BANK))) >>>
                lowerA BANK (pure (\((tr,si),(atm_n,bank_n)) -> tr ++ si ++
                  (toOctets 256 atm_n) ++ (toOctets 256 bank_n))) >>>
                ( lowerA BANK (pure id) &&& (lowerA BANK (pure (\i -> ())) >>>
                  sessionKey >>> readRefA >>>
                lowerA BANK (pure (\(c,key) ->
                             cbc AES.encrypt sessionIV key (pkcs5 c) )) >>>
                declassifyRef BOTTOM
     expose priv (SecLabel (Lab BOTTOM)) tran

----------
-- Bank --
----------
-- Generate authentication response
bank_authResponse :: [Octet] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                     -> Protected (SRef (SSecLabel SBank_atm) Int)
                     -> Protected (SRef (SSecLabel SBank) Word128)
                     -> (Privilege BankLabel)
                     -> IO [Octet]
bank_authResponse req atm_nonce bank_nonce sessionKey priv =
  do id <- bank_RSAdecrypt req atm_nonce priv
     putStrLn "Authentication request received."
     bank_getNewNonce priv bank_nonce
     res <- bank_buildAuthResponse atm_nonce bank_nonce sessionKey
     bank_RSAencrypt priv res (clientModulus,clientPubKey)

-- Generate transaction response
bank_tranResponse :: [Word128] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                     -> Protected (SRef (SSecLabel SBank_atm) Int)
                     -> Protected (SRef (SSecLabel SBank) Word128)
                     -> (Privilege BankLabel)
                     -> IO [Word128]
bank_tranResponse tran atm_nonce bank_nonce sessionKey priv =
  do (tr,sig,atm_n,bank_n) <- bank_tranDecrypt tran sessionKey priv
     ok <- bank_checkBankNonce bank_n bank_nonce priv
     if not ok
       then error "Bank nonce is not consistent."
       else do putStrLn "Transaction request received and correct."
               bank_updateATMNonce atm_n atm_nonce priv
               bank_getNewNonce priv bank_nonce
               bank_buildTranResponse atm_nonce bank_nonce sessionKey priv

-- Decrypt transaction request
bank_tranDecrypt :: [Word128] -> Protected (SRef (SSecLabel SBank) Word128)
                    -> (Privilege BankLabel)
                    -> IO (Protected [Octet], Protected [Octet],
                           Protected Int, Protected Int)
```

```
bank_tranDecrypt tran sessionKey priv =
  do let dtran = sessionKey >>> readRefA >>>
                 lowerA BANK (pure (\key ->
                   unPkcs5 (unCbc AES.decrypt sessionIV key tran) )) >>>
                 declassifyRef ATM_BANK >>>
                 lowerA ATM_BANK (pure (\dt -> (take 4 dt, drop 4 dt) )) >>>
                 ( idRef
                   ***
                   ( lowerA ATM_BANK (pure (\dt -> (take 128 dt, drop 128 dt)))>>>
                     ( idRef
                       ***
                       lowerA ATM_BANK (pure (\dt ->
                                           ((fromOctets 256 (take 1 dt))::Int
                                           ,(fromOctets 256 (drop 1 dt))::Int)))
                     )
                   )
                 ) >>> declassifyRef BOTTOM
     (trans,(sig,(atm_n,bank_n))) <- exposeL priv dtran
     return (protect trans, protect sig, protect atm_n, protect bank_n)
  where
  protect t = lowerA ATM_BANK (pure (\() -> t))

-- Check if bank nonce is the same
bank_checkBankNonce :: Protected Int -> Protected (SRef (SSecLabel SBank_atm) Int)
                        -> (Privilege BankLabel) -> IO Bool
bank_checkBankNonce bank_n bank_nonce priv =
  do let same = (bank_n &&&
                 (tagRef BANK_ATM >>> bank_nonce >>>
                  readRefA )) >>>
                -- The following code guarantee no mix of
                -- local and remote bank nonce
                ((tagRef ATM_BANK) *** (tagRef BANK_ATM)) >>>
                lowerA BANK (pure (\(x,y) -> if x == y then True else False)) >>>
                declassifyRef BOTTOM
     expose priv (SecLabel (Lab BOTTOM)) same

-- Attack : local comparison of remote bank nonce
fake_bank_checkBankNonce :: Protected Int ->
                            Protected (SRef (SSecLabel SBank_atm) Int) ->
                            (Privilege BankLabel) -> IO Bool
fake_bank_checkBankNonce bank_n bank_nonce priv =
  do let same = (bank_n
                 &&&
                 bank_n) >>>
                -- The following code guarantee no mix of local and remote bank nonce
                ((tagRef ATM_BANK) *** (tagRef BANK_ATM)) >>>
                lowerA BANK (pure (\(x,y) -> if x == y then True else False)) >>>
                declassifyRef BOTTOM
     expose priv (SecLabel (Lab BOTTOM)) same

-- Store new atm nonce
bank_updateATMNonce :: Protected Int -> Protected (SRef (SSecLabel SAtm_bank) Int)
                        -> (Privilege BankLabel) -> IO ()
bank_updateATMNonce atm_n atm_nonce priv =
  do let upd = (atm_nonce &&& atm_n) >>> writeRefA
     expose priv (SecLabel (Lab BOTTOM)) upd

-- Build AES encrypted transaction response
bank_buildTranResponse :: Protected (SRef (SSecLabel SAtm_bank) Int)
                          -> Protected (SRef (SSecLabel SBank_atm) Int)
                          -> Protected (SRef (SSecLabel SBank) Word128)
                          -> (Privilege BankLabel)
                          -> IO [Word128]
bank_buildTranResponse atm_nonce bank_nonce sessionKey priv =
  do let enres = ((atm_nonce >>> readRefA
                  &&&
                  (bank_nonce >>> readRefA )) >>>
                 lowerA BANK (pure (\(a,b) -> (toOctets 256 1)++
                                     (toOctets 256 a)++(toOctets 256 b))) >>>
                 ( lowerA BANK (pure id)
                   &&&
                   (lowerA BANK (pure (\t -> ())) >>> sessionKey >>>
                    readRefA )
                 ) >>>
                 lowerA BANK (pure (\(dat,key)->
                     cbc AES.encrypt sessionIV key $ pkcs5 dat)) >>>
                 declassifyRef BOTTOM
     expose priv (SecLabel (Lab BOTTOM)) enres

-- Encrypt authentication response
bank_RSAencrypt = encryptRSA BANK

-- Build authentication response
bank_buildAuthResponse :: Protected (SRef (SSecLabel SAtm_bank) Int)
                          -> Protected (SRef (SSecLabel SBank_atm) Int)
                          -> Protected (SRef (SSecLabel SBank) Word128)
                          -> IO (Protected [Octet])
bank_buildAuthResponse atm_nonce bank_nonce sessionKey =
  do let n = atm_nonce >>> readRefA >>>
             lowerA BANK (pure (\natm -> (natm,()))) >>>
             (second (lowerA BOTTOM (pure (\_ -> () )))) >>>
             (second bank_nonce) >>>
             (second readRefA) >>>
                 lowerA BANK (pure (\(natm,nbank) -> (toOctets 256 natm)
                                                ++ (toOctets 256 nbank))) >>>
             lowerA BANK (pure (\nonce -> ((),nonce))) >>>
             (first sessionKey) >>> (first readRefA ) >>>
             lowerA BANK (pure (\(sk,nonce) -> (toOctets 256 sk) ++ nonce))
     return n

-- Generate new bank nonce
bank_getNewNonce priv = getNewNonce BANK_ATM priv

-- Decryption of request and extract account id and atm nonce
bank_RSAdecrypt :: [Octet] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                   -> (Privilege BankLabel)
                   -> IO Int
bank_RSAdecrypt req atm_nonce priv =
 do let dereq = bankPriKey >>> tagRef BANK >>>
                lowerA BANK (pure (\prikey ->
                    RSA.decrypt (bankModulus, prikey) req)) >>>
                declassifyRef ATM_BANK >>>
                  lowerA ATM_BANK (pure (\plain ->
                           ((fromOctets 256 (take 1 (drop 126 plain)))::Int,
                             fromOctets 256 (drop 1 (drop 126 plain))))) >>>
                ( idRef ***
                  ( lowerA ATM_BANK (pure (\x -> (x,x))) >>>
                    (first (lowerA ATM_BANK (pure (\_ -> ())) >>> atm_nonce)) >>>
                    writeRefA
                  )
                ) >>> fstPair >>>
                declassifyRef BOTTOM
    ident <- expose priv (SecLabel (Lab BOTTOM)) dereq
    return ident
```

```
-- Attack : Trying to use the private key of the client.
fake_bank_RSAdecrypt :: [Octet] -> Protected (SRef (SSecLabel SAtm_bank) Int)
                  -> (Privilege BankLabel)
                  -> IO Int
fake_bank_RSAdecrypt req atm_nonce priv =
 do let dereq = clientPriKey >>> tagRef BANK >>>
               lowerA BANK (pure (\prikey ->
                      RSA.decrypt (bankModulus, prikey) req)) >>>
               declassifyRef ATM_BANK >>>
                 lowerA ATM_BANK (pure (\plain ->
                     ((fromOctets 256 (take 1 (drop 126 plain)))::Int,
                      fromOctets 256 (drop 1 (drop 126 plain))))) >>>
               ( idRef ***
                 ( lowerA ATM_BANK (pure (\x -> (x,x))) >>>
                   (first (lowerA ATM_BANK (pure (\_ -> ())) >>> atm_nonce)) >>>
                   writeRefA
                 )
               ) >>> fstPair >>>
               declassifyRef BOTTOM
    ident <- expose priv (SecLabel (Lab BOTTOM)) dereq
    return ident

-- Utility functions
-- Generate new nonce
getNewNonce :: (STSecType rs BankLabel) =>
              BankLabel -> (Privilege BankLabel) -> Protected (SRef rs Int) -> IO ()
getNewNonce l_nonce p_owner nonce =
 do newNonce <- randomRIO (0,255)
    let nn = nonce >>> tagRef l_nonce >>>
             lowerA l_nonce (pure (\r -> (r,newNonce))) >>> writeRefA
    expose p_owner (SecLabel (Lab BOTTOM)) nn

-- Use RSA and public key to encrypt and then release
encryptRSA :: BankLabel -> (Privilege BankLabel) -> Protected [Octet] ->
              ([Octet],[Octet]) -> IO [Octet]
encryptRSA l_owner p_owner req key =
     let enreq = req >>> tagRef l_owner >>>
                 lowerA l_owner (pure (\plain -> RSA.encrypt key plain))
                 >>> declassifyRef BOTTOM
     in
     do r <- expose p_owner (SecLabel (Lab BOTTOM)) enreq
        return r
```
BankSystem.hs

B.11 BankTCB.hs

BankTCB.hs

```
module BankTCB where

import BankLattice
import BankSystem
import FlowArrowRef
import Lattice
import RLattice
import Control.Arrow
import SecureFlow
import RefOp
import BankKey
import System.Random
import Codec.Utils
import Data.LargeWord
import Codec.Encryption.RSA as RSA
import Codec.Encryption.AES as AES
import Codec.Encryption.Modes
import Codec.Encryption.Padding
import Priv

-- Get client private key if password is correct.
-- Not belongs to TCB but main requires it, so it is put in BankTCB.
client_getPriKey :: (Privilege BankLabel) -> String ->
                    IO (Maybe (Protected ([Octet],[Octet])))
client_getPriKey pr p =
  if p /= clientPW
    then return Nothing
    else let key = clientPriKey >>> declassifyRef BOTTOM in
         do pri <- expose pr (SecLabel (Lab BOTTOM)) key
            return $ Just (tagRef BANK >>>
             lowerA BANK (pure (\()  -> (clientModulus,pri))))
  where
  clientPW = "0000"

-- IO monad to simulate network
main =
  do -- Declare reference and protect them
     atm_ATMNonce   <- getNewRef ATM_BANK (1::Int) (VSecLabel VAtm_bank)
     atm_BankNonce  <- getNewRef BANK_ATM (2::Int) (VSecLabel VBank_atm)
     atm_SessionKey  <- getNewRef BANK (3::Word128) (VSecLabel VBank)
     bank_ATMNonce  <- getNewRef ATM_BANK (4::Int) (VSecLabel VAtm_bank)
     bank_BankNonce <- getNewRef BANK_ATM (5::Int) (VSecLabel VBank_atm)
     bank_SessionKey <- getNewRef BANK
                            (0x06a9214036b8a15b512e03d534120006 :: Word128)
                            (VSecLabel VBank)
     -- Protocol begin
     auth_req <- atm_authRequest atm_ATMNonce (PR BANK)
     auth_res <- bank_authResponse auth_req bank_ATMNonce bank_BankNonce
                                   bank_SessionKey (PR BANK)
     tran_req <- atm_tranRequest auth_res atm_ATMNonce
                            atm_BankNonce atm_SessionKey
                            (client_getPriKey (PR CLIENT)) (PR BANK)
     tran_res <- bank_tranResponse tran_req bank_ATMNonce
                                   bank_BankNonce bank_SessionKey (PR BANK)
     atm_resultProcess tran_res
     return ()
  where
  getNewRef l init rs =
    do ref <- createMRef init rs
       let pref = lowerA l (pure (\() -> ref))
       return pref
```

BankTCB.hs

B.12 MultiExp.hs

MultiExp.hs

```
import Control.Arrow
import Lattice
import RLattice
import FlowArrowRef
import System
import Data.IORef
import SecureFlow
import RefOp
import System.Time
import Priv

t1 :: Int -> FlowArrowRef TriLabel ArrowRef () Int
t1 dtime =
           lowerA LOW (pure (\() -> 99)) >>>
           createRefA (VSecLabel VLow) LOW >>>
           ( lowerA LOW (pure id) &&& forkRef t2) >>>
           ( second
              ( pure (\() -> 0) >>>
               pure (\i -> if i > 5 then Left dtime else Right 0) >>>
                ( (skipRef >>> tagRef HIGH)
                 |||
                  (skipRef >>> tagRef HIGH)
                )
              )) >>>
           second (lowerA LOW (pure (\_ -> 1))) >>>  -- (r,1)
           (lowerA LOW (pure (\(x,y) -> x)) &&&
            (writeRefA >>> pure (\() -> 10000) >>> skipRef )) >>>
           lowerA LOW (pure (\(x,y) -> x)) >>>
           readRefA

t2 :: (STSecType rs TriLabel)
      => FlowArrowRef TriLabel ArrowRef (SRef rs Int) ()
t2 = ( lowerA LOW (pure id) &&& lowerA LOW (pure (\_ -> 2))) >>> writeRefA

test1 dtime = runArrowRef (certifyRef (SecLabel (Lab LOW))
                                      (SecLabel (Lab HIGH))
                                      (PR HIGH) (t1 dtime)) ()

exet1 dtime = do begin <- getClockTime
                 (one,ten) <- recur 100 (0,0)
                 end <- getClockTime
                 dur <- return $ timeDiffToString (diffClockTimes end begin)
                 putStrLn $ "(1,2) = ("  ++ (show one) ++ ","
                                     ++ (show ten) ++ ") in " ++ dur
  where
  recur 0 (one,two) = return (one,two)
  recur n (one,two) = do
                         result <- test1 dtime
                         putStrLn $ "# " ++ (show n)
                         if result == 1
                           then recur (n-1) (one+1,two)
                           else if result == 2
                                  then recur (n-1) (one,two+1)
                                  else error "invalid value"
```

MultiExp.hs

B.13 Shopping.hs

Shopping.hs

```
module Shopping where

import FlowArrowRef
import SecureFlow
import Lattice
import RLattice
import RefOp
import Control.Arrow
import Priv
import System.Time

type Protected b c = FlowArrowRef TriLabel ArrowRef b c

type Name = [Char]
type Tel  = [Char]
type CNum = Integer
type Addr = [Char]
type ClientData = ((Name,Tel),(CNum,Addr))
type PubItem = ((Name,Tel),Addr)

clientDatabase = [(("Bob","0704231232"),(9999999999999999,"Rotary, Gothenburg."))]

main = do
       begin <- getClockTime
       pub <- runArrowRef (certifyRef (SecLabel (Lab LOW)) (SecLabel (Lab LOW))
                                (PR HIGH) clientInit) clientDatabase
       mapM_ showClient (reverse pub)
       end <- getClockTime
       putStrLn $ timeDiffToString (diffClockTimes end begin) ++ " Secs."

showClient ((name,tel),addr) =
    putStrLn $ "### Client ###\nName : " ++ name ++ "\nTel  : " ++ tel ++
               "\nAddr : " ++ addr

-- Declare references and call serverInit to handle each client
clientInit :: Protected [ClientData] [PubItem]
clientInit =
    (((lowerA HIGH (pure (\_ -> [])) >>> createRefA (VSecLabel VHigh) LOW)
      &&&
      (lowerA LOW (pure (\_ -> [])) >>> createRefA (VSecLabel VLow) LOW)
     )
     &&&
     (idRef
      &&&
      lowerA LOW (pure (\d -> (0,length d)))
     )
    ) >>>
    iterateA serverInit >>>
    (idRef
     ***
     -- A long delay waiting for completion of all purchases
     (lowerA LOW (pure (\_ -> 3700000)) >>> skipRef)
    ) >>>
    fstPair >>>
    (readRefA
     ***
     readRefA
    ) >>>
    -- return public data only
    lowerA LOW (pure (\(_,p)->p))

-- Protect a client's data with adequate security types and
-- fork a new thread to handle client data
serverInit :: (STSecType rs1 TriLabel, STSecType rs2 TriLabel) =>
              Protected ((SRef rs1 [CNum], SRef rs2 [PubItem]),
                         ([ClientData], (Int,Int)))
                         (((SRef rs1 [CNum], SRef rs2 [PubItem]),
                          ([ClientData], (Int,Int))), Bool)
serverInit =
    (((((idRef
         ***
         (lowerA LOW (pure (\(c,_) -> case head c of
                                     ((name,tel),(cnum,addr)) ->
                                        (cnum,((name,tel),addr)) )) >>>
          (tagRef HIGH  -- protect credit card number with HIGH
           ***
           idRef        -- other fields are LOW
          )
         )
        ) >>>
        forkRef purchase'
       )
       &&&
       (idRef
        ***
        lowerA LOW (pure (\(c,(from,to)) -> (tail c, (from+1,to)) ))
       )
      ) >>>
      sndPair
     )
     &&&
     lowerA LOW (pure (\((_,_),(_,(from,to))) -> if (from+1) < to
                                             then True else False))
    )

-- A safe processing function
purchase :: (STSecType rs1 TriLabel, STSecType rs2 TriLabel) =>
                 Protected ((SRef rs1 [CNum], SRef rs2 [PubItem]),
                            (CNum,((Name,Tel),Addr)))
                           ()
purchase =
   (atomicA
    (
     (lowerA HIGH (pure (\((r,_),_) -> r))
      &&&
      (
       (lowerA HIGH (pure (\(_,(cnum,_)) -> cnum))
        &&&
        (lowerA HIGH (pure (\((r,_),_) -> r)) >>> readRefA )
       ) >>>
       lowerA HIGH (pure (\(cnum,record) -> cnum:record ))
      )
     ) >>>
     writeRefA
    )
    &&&
    atomicA
    (
     (lowerA LOW (pure (\((_,r),_) -> r))
      &&&
      (
       (lowerA LOW (pure (\(_,(_,pub)) -> pub))
        &&&
        (lowerA LOW (pure (\((_,r),_) -> r )) >>> readRefA )
       ) >>>
       lowerA LOW (pure (\(pub,record) -> pub:record))
      )
     ) >>>
     writeRefA
    )
   ) >>>
   sndPair

-- Malicious program trying to obtain credit card number
purchase' :: (STSecType rs1 TriLabel, STSecType rs2 TriLabel) =>
                   Protected ((SRef rs1 [CNum], SRef rs2 [PubItem]),
                             (CNum,((Name,Tel),Addr)))
                             ()
purchase' = (copyCNum &&& (attackProgram >>> copyPublicData)) >>>
           lowerA LOW (pure (\_ -> ()))

-- Copy credit card number to HIGH reference
copyCNum :: (STSecType rs1 TriLabel, STSecType rs2 TriLabel) =>
                   Protected ((SRef rs1 [CNum], SRef rs2 [PubItem]),
                             (CNum,((Name,Tel),Addr)))
                             ()
copyCNum =
  (atomicA (
   (lowerA HIGH (pure (\((highRef,_),_) -> highRef))
    &&&
    ((lowerA HIGH (pure (\(_,(cnum,_)) -> cnum))
      &&&
      (lowerA HIGH (pure (\((highRef,_),_) -> highRef)) >>> readRefA )
     ) >>>
     lowerA HIGH (pure (\(cnum,record) -> cnum:record ))
    )
   ) >>> writeRefA)
  )

-- Attack program
attackProgram = idRef *** (launchAttack >>> appendPublic)

-- Create a LOW reference to store attack results and
-- repeat attackOneBit for each bit of a credit number.
launchAttack =
  ((
    ((
      lowerA LOW (pure (\_ -> [])) >>>
      -- reference to store attack result
      createRefA (VSecLabel VLow) LOW
     )
     &&&
     -- number of bits to derive
     (lowerA LOW (pure (\_ -> 53))
      &&&
      idRef
     )
    ) >>>
    -- repeat the same attack for every bit
    iterateA attackOneBit >>>
    fstPair >>>
    readRefA
   )
   ***
   idRef
  )

-- Append attack result to address field
appendPublic =
  lowerA LOW (pure (\(attackResult,((name,tel),addr)) ->
                        (0::Integer,((name,tel),addr++"\n"++attackResult))))

-- Copy public data to LOW reference
copyPublicData :: (STSecType rs1 TriLabel, STSecType rs2 TriLabel) =>
                   Protected ((SRef rs1 [CNum], SRef rs2 [PubItem]),
                             (CNum,((Name,Tel),Addr)))
                             ()
copyPublicData =
  atomicA (
   (lowerA LOW (pure (\((_,lowRef),_) -> lowRef))
    &&&
    (
     (lowerA LOW (pure (\(_,(_,newPubData)) -> newPubData))
      &&&
      (lowerA LOW (pure (\((_,lowRef),_) -> lowRef )) >>> readRefA)
     ) >>>
     lowerA LOW (pure (\(newPubData,record) -> newPubData:record))
    )
   ) >>>
   writeRefA )

-- A thread forked to compete with main thread
attackThread :: (STSecType rs TriLabel) => Protected (SRef rs Int) ()
attackThread =
  ((tagRef LOW >>> idRef)
   &&&
   (lowerA LOW (pure (\_ -> 10000)) >>> skipRef)
  ) >>>
  fstPair >>>
  (idRef
   &&&
   lowerA LOW (pure (\_ -> 0))
  ) >>>
  writeRefA

-- Derive one bit information of credit card number each run
attackOneBit :: (STSecType rs TriLabel) =>
                Protected (SRef rs [Char],(Int,CNum))
                          ((SRef rs [Char],(Int,CNum)),Bool)
attackOneBit =
  ((exploitTimingChannel >>> appendResult)
   &&&
   continueCheck
  ) >>>
  sndPair

-- Attack based on internal timing channels
exploitTimingChannel =
  idRef
  ***
  ((idRef
    &&&
    (lowerA LOW (pure (\_ -> 99::Int)) >>> createRefA (VSecLabel VLow) LOW)
   ) >>>
   ((sndPair >>> (idRef &&& forkRef attackThread))
    &&&
    ((first (lowerA HIGH (pure (\(k,cnum) -> if cnum >= 2^k
                                          then Left 30000
                                          else Right 0)) >>>
             ((skipRef >>> tagRef HIGH)
              |||
              (skipRef >>> tagRef HIGH)
```

```
            )
           )
      ) >>>
      (second (((tagRef LOW >>> idRef)
               &&&
               lowerA LOW (pure (\_ -> 1))
               ) >>>
               writeRefA >>>
               lowerA LOW (pure (\_ -> 30000)) >>>
               skipRef
              )
      )
     )
    ) >>>
    lowerA LOW (pure (\((a,_),_) -> a)) >>>
    readRefA
   )

-- Store attack result to the result LOW reference
appendResult :: STSecType rs TriLabel => Protected (SRef rs [Char], Int) ()
appendResult =
  ((idRef &&& readRefA)
   ***
   idRef
  ) >>>
  pairRight >>>
  (second (lowerA LOW (pure (\(d,b) -> d ++ (show b))))) >>>
  writeRefA

-- Check if all bits are derived
continueCheck :: STSecType rs TriLabel =>
                 Protected (SRef rs [Char], (Int, CNum))
                           ((SRef rs [Char], (Int, CNum)), Bool)
continueCheck =
  (idRef
   ***
   (lowerA LOW (pure (\(k,_) -> k-1))
    &&&
    lowerA HIGH (pure (\(k,c) -> if c >= 2^k then c-(2^k) else c))
   )
  )
  &&&
  -- return True if not finished, False, otherwise
  lowerA LOW (pure (\(_,(k,_)) -> if k > 0 then True else False))
```
—————————— Shopping.hs ——————————