Encoding Multithreaded Information Flow in Haskell

Ta-chung Tsai

December 27, 2006

Contents

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 requited in order to protect credit card clients from fraudsters.

The problems above arise from 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 [13], 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 [13], in particular, *non-interference* policies [3]. A non-interference policy basically allows manipulation of secret data as long as the public output of programs are 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 [8], as a variant of Java, and FlowCaml [15], as an extension of Caml. They both adopt programming language techniques to specify and enforce information-flow policies.

Li and Zdancewic [6] 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 the *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 to 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, 12, 14, 18], 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 the programmers to write multithreaded programs that respect information-flow policies. We extend Li and Zdancewic's work to support complex security types. This allows the programmers to keep track of security types for each component of data precisely rather than make approximation by one lattice label. Reference manipulation is added in the sub-language and described by *arrows*. A unification mechanism is implemented to passing required information and alleviates the responsibility of the programmers. 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 report is organized as following. Chapter 2 describes background knowledge and existing approaches for multithreaded information flow. 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 evaluate 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 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 login program reveals if the password stores in the system(secret data) matches with password entered by the user(public data). Declassification is a mechanism to decrease security level of data in a controlled way. Jif [8] has a declassification mechanism based on *decentralized label model*(DLM) [7]. 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

Share 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:

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

where h and l are high and low variables, respectively. Assume $\texttt{skip}(n)$ executes n consecutive $skip$ commands. Notice that both b_1 and b_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 ran 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 on its definition. For instance, *Possibilistic noninterference*, explored by Smith and Volpano [16], 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* [18]. It requires that the probability distribution of *low* outputs is independent of *high* inputs. Apart from the non-interferences described above, Roscoe [11] 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 [18] 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 can not diverge, otherwise all the program will diverge as well. The following is an example of using `protect`:

$$\begin{array}{l} c_1 : \texttt{protect}(\texttt{if}\ \texttt{h} > 0\ \texttt{then}\ \texttt{skip}(120)\ \texttt{else}\ \texttt{skip}(0));\ \texttt{l} := 1 \\ c_2 : \texttt{skip}(60);\ \texttt{l} := 0 \end{array} \tag{2.2}$$

Since the if-statement of c_1 is executed atomically, c_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 primitive. Second, a strictly uniform scheduler 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 every low assignment is not dependent on running time of high value, low outputs will not change as high inputs vary. However, the program 2.1 is rejected. This shows the type system potentially reject secure and useful programs.

Russo and Sabelfeld [12] 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 low outputs.

Sabelfeld and Sands [14] 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 timing property of each command. Padding techniques may also change the efficiency of a program.

Zdancewic and Myers [19] 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 program like ($l := 1 \mid\mid l := 0$) and has 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.

Chapter 3

Encoding Information Flow in Haskell

This chapter begins with the explanation of the previous work that our project is based on. Then we define the terms and the problems of this project.

3.1 FlowHaskell

Instead of designing a new language from the scratch, Li and Zdancewic [6] showed how to develop an embedded security language in Haskell.[1] The approach used by FlowHaskell is considerably light-weighted comparing to Jif [8] and FlowCaml [15], 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 *arrows* combinators. Those constraints, when satisfied, guarantee that no data of a higher level in lattice flows to a place where data of a lower level is expected, unless that 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. The rest of the section introduces FlowHaskell in more detail.

3.1.1 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
```

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

```
  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. The programmers can thus defined their own security labels according to their needs.

3.1.2 FlowArrow

`FlowArrow` is an abstract data type implementing *arrows* 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
```

Field `computation` stores an arrow that takes an input of type `b`, and returns a value of type `c`. Type variable `l` represents a security label belonging to type class `Lattice`. 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 related 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 or equal than `l`.

`FlowArrow` requires underlying computations to be arrows. All 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.

3.1.3 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 the 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 label is `LOW` and gives `LOW` as its output label. This eliminates the case where the output label of the `pure` is higher than `LOW`.

3.1.4 Declassification

The declassification mechanism provided by FlowHaskell is similar to the *decentralized label model*(DLM) [7]. 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] }
```

The programmers specify original security label, $\mathtt{l}_1$, and also declassified security label, $\mathtt{l}_2$. A constraint `USERGEQ` is generated to ensure the authority of the code has a greater or equal privilege to $\mathtt{l}_1$.

3.1.5 Policy Enforcement

The programmers construct a `FlowArrow` program using all the combinators described in previous part of the section. Meanwhile, an underlying computation, which performs computation to solve problems of the 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.

3.2 Definition of Terms

FlowHaskell The previous work done by Li and Zdancewic [6].

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*.

3.3 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 [10] , 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.

Multi-threaded programs with shared resources open new timing channels. This has been understood for a long time. However, there is still no practical tools so far. Jif [8], FlowCaml [15], and FlowHaskell [6] all have no support for multithreaded information flow. In Chapter7, we propose a run-time system for FlowHaskellRef to eliminate internal timing channels.

Chapter 4

Extending FlowHaskell with Reference Manipulation

As a first step towards having multithreaded information-flow secure programs, we need share 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 Extension 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 to them. The types are elements of the language denoted by the following grammar.

$$\tau \ ::= \mathbf{int} \mid \tau \ \mathbf{ref} \mid (\tau, \tau) \mid \mathbf{either}\ \tau\ \tau \mid \ \ldots \tag{4.1}$$

In FlowHaskell, every type has only associated one security label as its security type. According to Pottier and Simonet [10], 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.

$$s^l \ ::= \ \ell \mid s^l \ \mathbf{ref}^\ell \mid (s^l, s^l) \mid (\mathbf{either}\ s^l\ s^l\ )^\ell \mid \mathbf{high} \tag{4.2}$$

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 Grammar 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 4.1: Subtyping

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

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

Figure 4.2: Function e

```
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 Grammar 4.2. For security labels ℓ in s^l, the sub-typing relationship is the same as in the lattice l. Type class `MultiLattice` provides methods to implement such sub-typing relation and other actions which will be useful later on (in Lattice.hs).

```
class Lattice l => MultiLattice s l where
  ml_top :: s l -> s l
  ml_bottom :: s l -> s l
  ml_label :: l -> s l -> s l
  ml_join :: s l -> s l -> s l
  ml_meet :: s l -> s l -> s l
  ml_leq :: s l -> s l -> Bool
  mextract :: s l -> l
  mext_join :: s l -> l
  mext_meet :: s l -> l
  ml_tag :: s l -> l -> s l
  ml_decl :: s l -> s l -> l -> s l
```

Type variable `s` is an abstract data type that represents a multiple lattice structure based on a lattice `l`. For instance, type variable `s` can be instantiated with `SecType` described previously. In the rest of the section, we assume type variable `s` is always instantiated to `SecType` and explained each method under this assumption. Class method `ml_top` and `ml_bottom` take a security type and produce top and bottom elements in the same lattice respectively. For example, method `ml_top` that applies to security type $(\bot, \bot)$

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

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

$$\frac{\uparrow\ s^l_1\ \ell\ \rightarrow\ s^l_3 \quad \uparrow\ s^l_2\ \ell\ \rightarrow\ s^l_4}{\uparrow\ (s^l_1, s^l_2)\ \ell\ \rightarrow\ (s^l_3, s^l_4)} \qquad \frac{\ell_1 \sqsubseteq \ell_2 \quad \uparrow\ s^l_1\ \ell_2\ \rightarrow\ s^l_3 \quad \uparrow\ s^l_2\ \ell_2\ \rightarrow\ s^l_4}{\uparrow\ (\mathbf{either}\ s^l_1\ s^l_2)^{\ell_1}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s^l_3\ s^l_4)^{\ell_2}}$$

$$\frac{\ell_2 \sqsubset \ell_1 \quad \uparrow\ s^l_1\ \ell_2\ \rightarrow\ s^l_3 \quad \uparrow\ s^l_2\ \ell_2\ \rightarrow\ s^l_4}{\uparrow\ (\mathbf{either}\ s^l_1\ s^l_2)^{\ell_1}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s^l_3\ s^l_4)^{\ell_1}} \qquad \frac{}{\uparrow\ \mathbf{high}\ \ell\ \rightarrow\ \mathbf{high}}$$

Figure 4.3: Function ↑

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

$$\frac{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_3 \quad \downarrow\ s^l_1\ s^l_2\ \ell_2\ \rightarrow\ s^l_3}{\downarrow\ (s^l_1\ \mathbf{ref}^{\ell_1})\ (s^l_2\ \mathbf{ref}^{\ell_2})\ \ell_2\ \rightarrow\ s^l_3\ \mathbf{ref}^{\ell_3}} \qquad \frac{\downarrow\ s^l_1\ s^l_3\ \ell\ \rightarrow\ s^l_5 \quad \downarrow\ s^l_2\ s^l_4\ \ell\ \rightarrow\ s^l_6}{\downarrow\ (s^l_1, s^l_2)\ (s^l_3, s^l_4)\ \ell\ \rightarrow\ (s^l_5, s^l_6)}$$

$$\frac{\downarrow\ \ell_1\ \ell_2\ \ell_2\ \rightarrow\ \ell_3 \quad \downarrow\ s^l_1\ s^l_3\ \ell_2\ \rightarrow\ s^l_5 \quad \downarrow\ s^l_2\ s^l_4\ \ell_2\ \rightarrow\ s^l_6}{\downarrow\ (\mathbf{either}\ s^l_1\ s^l_2)^{\ell_1}\ (\mathbf{either}\ s^l_3\ s^l_4)^{\ell_2}\ \ell_2\ \rightarrow\ (\mathbf{either}\ s^l_5\ s^l_6)^{\ell_3}} \qquad \frac{}{\downarrow\ \mathbf{high}\ s^l\ \ell\ \rightarrow\ s^l}$$

Figure 4.4: Function ↓

returns $(\top, \top)$. They are special cases of method `ml_label` which produces an element in the same lattice as the second parameter, a security type, but with all security labels being the first parameter, a lattice label. Method `ml_join` and `ml_meet` return the join and the meet of two elements in the same lattice respectively. Method `ml_leq` implements the sub-typing relation showed in Figure 4.1. This sub-typing relation corresponds to the partial ordering relation in lattices of different constructor. Method `mextract` implements function e in Figure 4.2. It returns a lattice label that represents the security level of a information flow and is used to eliminate *implicit* flow. Class method `mext_join`(`mext_meet`, resp.) returns a lattice label which is the join(the meet, resp.) of labels inside of a security type. Method `ml_tag` implements function ↑ in Figure 4.3 and method `ml_decl` implements function ↓ in Figure 4.4. The second security type of `ml_decl` has identical constructors to the first security type except when the first one is **high**. All security labels inside of the second security type are the same as the third lattice label.

Programmers only need to define a lattice type to achieve their goals as in FlowHaskell-Ref. 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 `MultiLattice`.

4.2 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 *arrows* computations in `FlowArrowRef`. It replaces `FlowArrow` and implements *arrows* interface.

4.2.1 Arrows Computation

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

```
data FlowArrowRef l a b c =
    FARef { computation :: a b c
          , flow        :: Flow (SecType l)
          , constraints :: [Constraint (SecType l)]
          , gconstraints:: [GConstraint (SecType l)]
          , pc          :: 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. Fields `computation`, `flow`, and `constraints` are the same as in `FlowArrow`, see in Sec. 3.1.2. Field `gconstraints` is a list of constraints which relate security types s^l with lattice labels ℓ. It differs from `Constraint` which involves two security types. Field `pc` keeps track of the lowest label of visible side effects produced in a computation.

4.2.2 Type System

The type system conceptually implemented by `FlowArrowRef` is depicted in Figure 4.5, Figure 4.6, and Figure 4.7. The type judgement has following form:

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

Environment variable pc is the lower bound of side effects produced by computation `f`. Variable Δ denotes constraints of the form $\ell \lhd s^l$, $\ell \blacktriangleleft s^l$, and $s^l \preceq \ell$, which are constructed by using the rules in Figure 4.8, Figure 4.9, and Figure 4.10. Variable Θ represents constraints of the form $s^l \sqsubseteq s^l$ and $s^l \sqsubseteq$ `user`. The input and output types of computation `f` are τ_1 and τ_2 respectively. The input and output security types of `f` are s_1^l and s_2^l, respectively.

$$(PURE)\,\frac{f\ :\ \tau_1\ \rightarrow\ \tau_2}{\top,\emptyset,\emptyset\vdash \texttt{pure}\,f\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid \mathbf{high}}$$

$$(SEQ)\,\frac{pc_1,\Delta_1,\Theta_1\vdash f_1\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid s_2^l\quad pc_2,\Delta_2,\Theta_2\vdash f_2\ :\ \tau_2\mid s_3^l\ \rightarrow\ \tau_4\mid s_4^l}{pc_1\sqcap pc_2,\Delta_1\cup\Delta_2,\Theta_1\cup\Theta_2\cup\{s_2^l\sqsubseteq s_3^l\}\vdash f_1\ \ggg\ f_2\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_4\mid s_4^l}$$

$$(FIRST)\,\frac{pc,\Delta,\Theta\vdash f\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid s_2^l}{pc,\Delta,\Theta\vdash \texttt{first}\,f\ :\ (\tau_1,\tau_3)\mid(s_1^l,s_3^l)\ \rightarrow\ (\tau_2,\tau_3)\mid(s_2^l,s_3^l)}$$

$$(SECOND)\,\frac{pc,\Delta,\Theta\vdash f\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid s_2^l}{pc,\Delta,\Theta\vdash \texttt{second}\,f\ :\ (\tau_3,\tau_1)\mid(s_3^l,s_1^l)\ \rightarrow\ (\tau_3,\tau_2)\mid(s_3^l,s_2^l)}$$

$$(PAR1)\,\frac{pc_1,\Delta_1,\Theta_1\vdash f_1\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid s_2^l\quad pc_2,\Delta_2,\Theta_2\vdash f_2\ :\ \tau_3\mid s_3^l\ \rightarrow\ \tau_3\mid s_4^l}{pc_1\sqcap pc_2,\Delta_1\cup\Delta_2,\Theta_1\cup\Theta_2\vdash f_1\ {*}{*}{*}\ f_2\ :\ (\tau_1,\tau_3)\mid(s_1^l,s_3^l)\ \rightarrow\ (\tau_2,\tau_4)\mid(s_2^l,s_4^l)}$$

$$(PAR2)\,\frac{pc_1,\Delta_1,\Theta_1\vdash f_1\ :\ \tau_1\mid s_1^l\ \rightarrow\ \tau_2\mid s_2^l\quad pc_2,\Delta_2,\Theta_2\vdash f_2\ :\ \tau_1\mid s_3^l\ \rightarrow\ \tau_4\mid s_4^l}{pc_1\sqcap pc_2,\Delta_1\cup\Delta_2,\Theta_1\cup\Theta_2\vdash f_1\,\&\&\&\,f_2\ :\ \tau_1\mid(s_1^l\sqcap s_3^l)\ \rightarrow\ (\tau_2,\tau_4)\mid(s_2^l,s_4^l)}$$

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

Combinator `pure` lifts any Haskell function `f` into `FlowArrowRef`. Function `f` takes the input security types s_1^l. 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 $\mathbf{high}$. 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 lowest level 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 *arrows* computations should have the same input type. The new input security label becomes the meet of the two input security types.

Rules for branching computations are shown in Figure 4.6. Rule (`LEFT`) and (`RIGHT`) lift the input and output types and security types in $either$ and $\mathbf{either}$ types respectively. In rule (`CHOICE1`), the constraint $(\mathbf{either} s_1^l\ s_3^l)^\ell \blacktriangleleft (pc_1 \sqcap pc_2)$ 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 $(\mathbf{either}\ s_1^l\ s_3^l)^\ell$. This prevents from leaking some 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 $\mathbf{either}$ constructor. 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

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

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

$$(\mathit{CHOICE1})\frac{pc_1, \Delta_1, \Theta_1 \vdash f_1 \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l \quad pc_2, \Delta_2, \Theta_2 \vdash f_2 \ : \ \tau_3 \mid s_3^l \ \rightarrow \ \tau_4 \mid s_4^l}{pc_1 \sqcap pc_2, \Delta_1 \cup \Delta_2 \cup \mathit{constraint}1, \Theta_1 \cup \Theta_2 \vdash f_1 +\!\!+\!\!+ f_2 \ : \ \mathit{flow}1}$$

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

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

$$(\mathit{CHOICE2})\frac{pc_1, \Delta_1, \Theta_1 \vdash f_1 \ : \ \tau_1 \mid s_1^l \ \rightarrow \ \tau_2 \mid s_2^l \quad pc_2, \Delta_2, \Theta_2 \vdash f_2 \ : \ \tau_3 \mid s_4^l \ \rightarrow \ \tau_2 \mid s_4^l}{pc_1 \sqcap pc_2, \Delta_1 \cup \Delta_2 \cup \mathit{constraint}2, \Theta_1 \cup \Theta_2 \vdash f_1 \mid\mid\mid f_2 \ : \ \mathit{flow}2}$$

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

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

Figure 4.6: Type system of methods in type class `ArrowChoice`

$e(\uparrow (s_2^l \sqcup s_4^l, \ell))$. This constraint is necessary in the case s_1^l or s_3^l has security types higher than the identity security label ℓ.

Figure 4.7 shows the type system of new combinators that are not in standard *arrow* interface. 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. This is implemented by function $\uparrow$, as defined in Figure 4.3. Combinator `declassifyRef` takes a lattice label, ℓ, and declassifies the input security type. Each security label in the input security type which is higher than ℓ is downgraded to ℓ. This is done by function $\downarrow$ in Figure 4.4. Constraint $s^l \sqsubseteq \mathtt{user}$ guarantees the authority of the code, denoted by `user`, is equal or higher than the input security type. Combinator `lowerA` takes a lattice label and a `FlowArrowRef`, and transform the output security type of the computation based on the output type. Function ρ, defined in Figure 4.12, generates a security type with every security label being ℓ. 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 when certifying a program. Combinator `iterateA` takes a `FlowArrowRef` and transforms the computation of the *arrow*. If the second element of the output is `True`, the whole computation is repeated again by taking the first element

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

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

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

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

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

Figure 4.7: Type system of new arrow combinators

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

Figure 4.8: Constraint $\lhd$

of the output as input. Thus, constraint $s^l_2 \sqsubseteq s^l_1$ is required. The other constraint is necessary because the outputs depends on the boolean value.

The type system for `certifyRef` is described in Figure 4.11. 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 is well-typed if all the constraints in the premise are satisfied.

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

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

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 $\preceq$

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

Figure 4.11: Type system of certify

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

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

Figure 4.12: Function ρ

$$(CREATE)\frac{}{e(s^l), \emptyset, \emptyset \vdash \texttt{createRef}\ \ell\ :\ \tau \mid s^l\ \rightarrow\ \tau\ ref \mid s^l\ \mathbf{ref}^\ell}$$

$$(READ)\frac{}{\top, \{\ell \lhd s_2^l\}, \{s_1^l \sqsubseteq s_2^l\} \vdash \texttt{readRef}\ s_2^l\ :\ \tau\ ref \mid s_1^l\ \mathbf{ref}^\ell\ \rightarrow\ \tau \mid s_2^l}$$

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

Figure 4.13: Type system of reference primitives

4.3 Reference Manipulation

FlowHaskellRef adopts mutable references in the IO monad for experiment. An abstract data type `SRef` is introduced as following(in RefOp.hs):

```
data SRef l a = MkSRef (IORef a) (a -> a) ((SecType l) -> a -> a))
```

Data type `SRef` parameterises on a lattice and the type of a reference's content. The constructor `MkSRef` takes an `IORef`, a reading projection function, and a writing protection function. The reading projection function is applied to the result of reading the reference and returns a value of the same type. The writing protection function takes a security type and a value and produce a value that is going to be written to the reference. It may raise an error if the security type is higher than expected. The purpose of these two functions are explained in Sec. 4.5.

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

```
newSRef :: (Lattice l) => a -> IO (SRef l a)
newSRef a = do r <- newIORef a
               return (MkSRef r id (outFilter SecHigh))

readSRef :: (Lattice l) => SRef l a -> IO a
readSRef (MkSRef r fi fo) = do a <- readIORef r
                               return (fi a)

writeSRef :: (Lattice l) => SRef l a -> a
                          -> (SecType l) -> IO ()
writeSRef (MkSRef r fi fo) a s = writeIORef r (fo s a)

outFilter :: Lattice l => (SecType l) -> (SecType l) -> a -> a
outFilter s1 s2 a = if s2 ‘ml_leq‘ s1
                      then a
                      else error $ "..."
```

The projection function is set to `id` when new references are created. The writing protection function is (`outFilter SecHigh`) so data of any security type that is going to be written to the reference can pass the function. This is because the content of a reference is protected by its security type. In `readSecRef`, the projection function `f` is applied to the result of reading internal IORef, `r`. Function `writeSecRef` applies the writing protection function to the value that is going to be written to the reference and writes the result to the reference.

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 Lattice l => RefMonad m r l | m -> r where
  createMRef :: a -> m (r l a)
  readMRef :: (r l a) -> m a
  writeMRef :: (r l a) -> a -> (SecType l) -> m ()
```

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

Three standard operations for reference manipulation, called `createRef`, `readRef`, and `writeRef`, are implemented in FlowHaskellRef. They lift methods of `RefMonad` into `FlowArrowRef`. The implementation follows the type system in Figure 4.13 directly and can be found in code FlowArrowRef.hs in the appendix. The type system is based on Pottier and Simonet's work [10].

In declassification, the write protection function of a reference is updated because the security types of the reference's content is also declassified. This is done by a type class `DeclOp`(in SecureFlow.hs).

```
class DeclOp t where
  declop :: t -> t

instance (Lattice l) => DeclOp (SRef l a) where
  declop (MkSRef r fin fout) = MkSRef r fin (outFilter SecHigh)
```

Method `declop` is a run-time operation applied to the declassified data. For a reference, the write protection function is updated to be the one as a reference is created. The reason is that the declassified security types of the reference's content already protect the reference from illegal manipulations violating information-flow policies. New write protection functions are added when the reference passes through combinator `lowerA`. For other types, method `declop` is implemented as function `id`.

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 the programmers with the ability to define new *arrow* computations based on Haskell functions they need. It is an essential building block when programming in FlowHaskellRef.

However, to derive the output security label 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. The other problem is the dependency of inputs and outputs of a