Arrows for Secure Information Flow

Peng Li Steve Zdancewic

University of Pennsylvania, Department of Computer and Information Science, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, USA

Abstract

This paper presents an embedded security sublanguage for enforcing information-flow policies in the standard Haskell programming language. The sublanguage provides useful information-flow control mechanisms including dynamic security lattices, run-time code privileges and declassification all without modifying the base language. This design avoids the redundant work of producing new languages, lowers the threshold for adopting security-typed languages, and also provides great flexibility and modularity for using security-policy frameworks.

The embedded security sublanguage is designed using a standard combinator interface called *arrows*. Computations constructed in the sublanguage have static and explicit control-flow components, making it possible to implement information-flow control using static-analysis techniques at *run time*, while providing strong security guarantees. This paper presents a formal proof that our embedded sublanguage provides noninterference, a concrete Haskell implementation and an example application demonstrating the proposed techniques. [1]

Key words: Information flow, security, Haskell, arrows, type systems, combinators

1 Introduction

Language-based information-flow security (Sabelfeld and Myers, 2003) has a long, rich history with many (mostly theoretical) results. This prior work has focused mainly on the problems of static program analysis for a wide variety of computation models and policy features. Often these analyses are presented as type systems whose soundness is justified by some form of *noninterference*

[1] This paper is an expanded version of an earlier paper that appeared in IEEE CSFW (Li and Zdancewic, 2006).

Email addresses: `lipeng@cis.upenn.edu` (Peng Li), `stevez@cis.upenn.edu` (Steve Zdancewic).

result. The approach is compelling because programming-language techniques can be used to specify and enforce security policies that cannot be achieved by conventional mechanisms such as access control and encryption. Two full-scale language implementations have been developed: Jif (Myers, 1999; Myers et al., 2001) is a variant of Java, and Flow Caml (Simonet, 2003; Pottier and Simonet, 2002) is an extension of Caml.

However, despite this rather large (and growing!) body of work on language-based information-flow security, there has been relatively little adoption of the proposed techniques—two success stories are the "taint-checking mode" available in the Perl language and the use of information-flow analysis to separate lower integrity components from higher integrity components built in the SparkAda (Chapman and Hilton, 2004) language.

One important reason why these domain-specific security-typed languages have not been widely applied is that, because the information-flow policies are intended to apply in an end-to-end fashion, the whole system has to be written in the new language. However, it is expensive to build large software systems in a new language. Doing so can be justified only if the benefit of using the new language outweighs the cost of migrating to the new language—including the costs of retraining programmers and the time and expense necessary to port existing libraries and other code bases.

Moreover, in practice, it may well be the case that only a small part of the system (maybe only a few variables in a large program) has information-flow security requirements. Although the system may be large and complex, the secret information flow in the system may not be completely unmanageable. In such cases, it is probably more convenient to use the programming language that best fits the primary functionality of the system rather than its security requirements, and manage security issues by traditional means such as code auditing and careful software engineering practices. Such practical solutions are often a compromise, because there is no *provable* security indeed. In reality, there is a language adoption threshold based on the ratio of *security requirements* to *functionality requirements*, and this threshold is very high.

1.1 Background on security-typed languages

In security-typed languages like Jif (Myers, 1999; Myers et al., 2001) and Flow Caml (Simonet, 2003; Pottier and Simonet, 2002), variables can have security annotations in their type declarations. Each variable has a security level, represented syntactically by a *label* in the program. The security levels are usually partially ordered and form a *lattice*. For example, the simplest security lattice $\{\mathsf{L}, \mathsf{H}\}, \mathsf{L} \sqsubseteq \mathsf{H}$ defines two labels L and H and specifies an or-

dering between the two security levels represented by these labels. At compile time, security-typed programs are checked to guarantee that there is no information flow from higher security levels to lower security levels; violations of this information-flow policy result in type errors.

Security-typed languages often favor *static* type checking, because information-flow analysis requires the view of the entire control-flow graph in order to examine *implicit* information flows caused by conditional branches. For example, the C program "`if (h==0) l=0; else l=1;`" has an implicit flow from `h` to `l` and, although there is no *explicit* assignment "`l=(h==0);`", these code fragments are effectively equivalent. In a dynamically-typed language, such implicit flows are difficult to capture. For example, the taint-checking mode of Perl does not capture the implicit flow in the above code: even if `h` is tainted and there is information flow from `h` to `l`, `l` is still not tainted. This is fine because the taint-checking mode is used only to provide a modest level of *integrity* guarantees. However, for *confidentiality* purposes, implicit flows are often unacceptable, especially when the program is not trusted.

To make security-typed languages practical, a feature called *declassification* is necessary: sometimes, we *do* need information flow from higher levels to lower levels, but only in permitted ways. For example, secret information can be sent to public places after it is encrypted. One popular solution is to make *declassification* an explicit (and unsafe) type cast; it is thus the programmers' responsibility to use declassification safely. When the code is not trusted, declassification can be dangerous. The *decentralized label model* (Myers and Liskov, 2000) solves this problem by assigning code privileges to program modules, so each module can only declassify information of certain security levels that are determined by the code privilege.

1.2 Embedded security-typed sublanguages

This paper presents a different approach to enforcing information-flow security policies. Rather than producing a new language from scratch, we show how to encode traditional information-flow type systems using general features of an existing, modern programming language. In particular, we show how the *abstract data type* and the *type class* features found in Haskell (Peyton Jones et al., 2002) can be used to build a module that effectively provides a security-typed sublanguage embedded in Haskell itself. This sublanguage can interoperate smoothly with existing Haskell code while still providing strong information-flow security guarantees. Importantly, we do not need to modify the design or implementation of Haskell itself—we use features in its standard (but advanced) type system.

Our approach reduces the adoption threshold for systems implemented in Haskell: such systems can be made more secure without completely rewriting them in a new language. The implementation can be a fine-grained mixture of normal code and security-hardened code (variables, data and computations over secure data). The programmer needs to protect only sensitive data and computation using a software library, which enforces the information-flow policies throughout the entire system and provides end-to-end security goals like noninterference.

Another benefit of our approach is flexibility. A specialized language like Jif must pick a fixed policy framework in which the security policies are expressed. Considering the plethora of features present in the literature expressing the label lattice, declassification options (Sabelfeld and Sands, 2005), dynamic policy information (Tse and Zdancewic, 2004a; Zheng and Myers, 2004), etc., it is unlikely that any particular choice of policy language will be suitable for all programs with security concerns. By contrast, since it is much easier to build a library module than to build a new language, it is conceivable that different programs would choose to implement entirely different policy frameworks. Our embedded sublanguage approach is *modular* in the sense that it provides an interface through which the programmer can choose which policy framework and type system to use for specific security goals. In this paper we sketch one possible policy framework that illustrates one particular choice of label lattice, declassification mechanism, and support for dynamic policies, but others could readily be implemented instead.

Although we use Haskell's advanced type system and helpful features like the ability to overload syntax, studying how to encode information-flow policies in the context of Haskell can point to how similar efforts might be undertaken in more mainstream languages like Java. Also, since the features we use are intended to be "general purpose," they are more likely to find a home in a mainstream language than the less widely applicable security types. Evidence of this can be found, for example, in Sun's recent addition of parametric polymorphism, a key component of our approach, to Java.

1.3 Overview of technical development

There are two key technical challenges in embedding a useful security-typed sublanguage in Haskell. The first problem is that enforcing information-flow policies requires static analysis of the control flow graph of the embedded programs—purely dynamic enforcement mechanisms are generally too conservative in practice. The second problem is representing the policy information itself—depending on the desired model, the policy information might be quite complex, perhaps depending on information available only at run time.

Our solution to the first problem is to use *arrows* (Hughes, 2000). Intuitively, arrows provide an abstract interface for defining embedded sublanguages that support standard programming constructs familiar to programmers: sequential composition, conditional branches, and loops. Haskell provides convenient syntactic sugar for writing programs whose semantics are given by an arrow implementation.

To address the second problem, we use Haskell's *type class* mechanism to give an interface for security lattices. Programs written in the embedded language can be parameterized with respect to this interface. Moreover, the embedded language can easily be given security-specific features such as a declassification operation or run-time representation of privileges for access-control checks.

In both cases, we make use of Haskell's strong type system to guarantee that the abstractions enforcing the security policies are not violated. This encapsulation means that it is not possible to use the full power of the Haskell language to circumvent the information-flow checks performed by the embedded language, for example.

The rest of the paper is organized as follows. Section 2 presents a brief tutorial of the arrow interface. Section 3 presents the detailed implementation of an arrow-based sublanguage for information-flow control. Section 4 gives some example programs that illustrate how the secure embedded language and the Haskell program can be smoothly integrated and considers the issues with enforcing the desired security properties. Section 5 formalizes and proves the security guarantee of our security sublanguage. Section 6 discusses some limitations and caveats with this approach and describes some future work. Section 7 concludes.

2 The arrows interface

The concept of *arrows* was proposed by Hughes (2000) as a generalization of *monads* (Wadler, 1992). Both monads and arrows are generic interfaces for constructing programs using combinators. This section presents an informal and brief tutorial of arrows [2] and shows how arrows can be used to construct computations with explicit control flow structures.

[2] The related references (Hughes, 2000; Paterson, 2003, 2001) in the bibliography can be used for more detailed studies of arrows.

The following code specifies the simplest `Arrow` type class. Here, `a` is an abstract type of arrow with input type `b` and output type `c`. This arrow type class supports only three operations: `pure`, `(>>>)`, and `first`.

```
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)
```

Basic blocks and compositions

The `pure` operation lifts a Haskell function of type `b->c` into the arrow; such lifted functions serve as the "basic blocks" of the control-flow graphs constructed via arrow combinators. The infix operation `(>>>)` provides sequential (horizontal) composition of computations, and `first` provides parallel (vertical) composition of computations.

An instance of the `Arrow` type class is required to satisfy a set of axioms that specify coherence properties between the operations. For example, `(>>>)` is required to be associative. We omit the complete description of the arrow axioms here; for our purposes, there is only one interesting case to consider, and it is discussed in Section 6. The simplest instance of `Arrow` is Haskell's function arrow constructor `(->)` itself: every function of type `b -> c` is also an arrow computation `(->) b c`.

Using the concept of *monad transformers* (Liang et al., 1995), one can also build *arrow transformers*, which correspond closely to the categorical notion of *functors*. Monad and arrow transformers allow program functionality be composed in a modular, layered fashion.

Representing conditionals and loops

The basic `Arrow` interface does not provide the ability to construct conditional computations—it can construct only control-flow graphs that represent straight-line code with no branches. Two other type classes refine arrows by permitting conditional branches and loops.

The `ArrowChoice` type class provides an operation called `left` that extends the `Arrow` interface with the ability to perform a one-sided branch computation depending on the arrow's input value. In Haskell, the type `Either b d` describes a value that is a *tagged union* (or *option*) type that carries either a value of type `b` or one of type `d`.

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

Using `left` and the other arrow primitives, the following operations can be implemented to construct different kinds of conditional computations:

```
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
```

`ArrowLoop` provides the embedded language with a loop construct sufficient for encoding `while` and `for` loops. Intuitively, the `loop` operator feeds the `d` output of the arrow back into the `d` input of the arrow, introducing a cycle in the control-flow graph:

```
class Arrow a => ArrowLoop a where
 loop :: a (b, d) (c, d) -> a b c
```

The benefit of having this operation is that recursive computations can be constructed as a finite combination of arrow components. We will use this property in Section 3.6.

Translating the do-syntax

Programming directly with the arrow operations is sometimes cumbersome, because arrows require a point-free programming style. The `do`-syntax for arrows (Paterson, 2001) provides syntactic sugar for arrow programming, such as arrow abstraction, arrow application, sequential composition, conditional branching and recursion. Internally, the Haskell compiler[3] translates the `do`-syntax used in the embedded sublanguage into the basic arrow operations. For example, conditional statements are translated to `pure, >>>` and `|||` operators, using the following rule:

```
⟦ proc p -> if e then c1 else c2 ⟧ =
  pure (\p -> if e then Left p else Right p)
     >>> ⟦ proc p -> c1 ⟧ ||| ⟦ proc p -> c2 ⟧
```

This rule translates the `if` command in the sublanguage. The `if` construct in the translated code is the conditional expression in the base Haskell language. The sublanguage syntax "`proc p->`" provides an arrow abstraction that binds the arrow input to the variable `p`.

The compiler is able to resolve this syntax overloading by using type information. For example, the type `Protected` mentioned in our implementation described below informs the compiler to use our definition for `|||`, which is

[3] We use the Glasgow Haskell Compiler (`http://www.haskell.org/GHC`).

given by our `FlowArrow` instance of the `ArrowChoice` type class. This feature allows programmer-defined embedded sublanguages to have convenient syntax.

2.2 Control flow in arrow sublanguages

The Haskell programming language itself provides branching, looping, and other control-flow constructs, so one might wonder why it is necessary to re-implement all of these features in the embedded sublanguage. Compared to Haskell's full control-flow mechanisms (which also include function calls and exceptions, for example), the arrow type classes are actually quite impoverished. The arrow interfaces isolate the base language (Haskell) and the sublanguage (arrows): by design, the control flow constructs in the base language cannot be directly used to represent the control flow of the sublanguage.

This separation property is crucial for the security analysis of arrow-based sublanguages. If an arrow implements only the operations in the `ArrowChoice` type class, conditional branches on arrow computations can only be implemented using the given arrow operations `left`, `right`, `(+++)`, and `(|||)`. By keeping the arrow implementation abstract, the programmer is forced to use these arrow operations for writing conditional branches, because there is no other way to manipulate the interface.

Therefore, by designing the arrow interface with limited control-flow primitives, the control-flow graph of an arrow computation is determined by the composition of primitive arrow operations. In other words, arrows can force computations to be constructed with static and explicit control-flow structures. This makes it possible to completely analyze the information flow in an arrow-based sublanguage before running the computation. A more permissive interface to the sublanguage (such as provided by *monads* for example) would allow base language branches to leak information about supposedly protected data.

3 An embedded security-typed language

This section presents the design of our secure embedded sublanguage using the arrows interface. Our design uses the structure of *arrow transformers*, which allows arrow sublanguages to be composed in a modular, layered fashion.

3.1 Encoding the security lattice

In our embedded language, the security labels are encoded using term-level values. We start by defining a generic Haskell interface for security labels and lattices: the type class `Lattice` provides a set of operations common for all security lattices.

```
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
```

The programmer has the freedom to choose the implementation of the actual security lattice. Because we are encoding labels using terms, there is no limitation on the expressiveness of security policies: any security lattice can be encoded as long as its labels and the operations on them can be represented using Haskell. For simplicity and ease of presentation, we use the following three-point lattice throughout the rest of the paper.

```
data TriLabel = LOW | MEDIUM | HIGH deriving (Eq, Show)
instance Lattice TriLabel 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 LOW _ = True
  label_leq MEDIUM LOW = False
  label_leq MEDIUM _ = True
  label_leq HIGH HIGH = True
  label_leq HIGH _ = False
```

3.2 Encoding flow types and constraints

This paper employs a simple information-flow type system for purely-functional arrow computations. An arrow computation has an input security label l_1 and an output security label l_2. Each typing judgment has the form

$$\Phi \vdash c : l_1 \rightarrow l_2$$

where c is a purely functional computation, $l_1 \rightarrow l_2$ is the *flow type* assigned to c, and Φ is a list of label constraints. The type system is presented in Figure 1. There is also a certification judgement in the type system. We defer its discussion until Section 3.5.

The sublanguage types appearing in the typing judgments are encoded using the Haskell data type:

```
data Flow l = Trans l l | Flat
```

(1) `Trans` l_1 l_2 specifies a security type $l_1 \rightarrow l_2$.
(2) `Flat` means the input and output can be given the same arbitrary label. It specifies a security type $l \rightarrow l$, where the label l can be determined by constraints in the context.

The label constraints are encoded using the `Constraint` data type:

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

(1) `LEQ` l_1 l_2 represents a direct ordering between two labels: $l_1 \sqsubseteq l_2$.
(2) `USERGEQ` l represents the constraint $l \sqsubseteq$ `user`. It requires that the run-time code privilege, which is represented as a label, be at least l. This will be used in Section 3.5 when we implement declassification.

The purpose of the constraint set Φ is to implement late binding of the security lattice. The type system collects the label constraints when secure computations are constructed from individual components. Such constraints are checked when the secure computation is accessed through the policy enforcement mechanism, namely, the `cert` operation. This design makes it possible to use dynamic security lattices and also helps when implementing declassification.

3.3 Encoding typing judgments and rules

The abstract datatype `FlowArrow` defines our secure embedded language by implementing the arrow interfaces described above:

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

A value of type `FlowArrow l a b c` is a record with three fields. The `computation` field encapsulates an arrow of type `a b c` that is the underlying computation to be protected. The `flow` field specifies the security levels for the input and output of the computation. The `constraints` field stores the list of flow constraints Φ when the arrow computation is constructed from smaller components.

`FlowArrow` encodes an information-flow typing judgment for a purely functional arrow computation, using the encoding of flow types and constraints we just

$\boxed{\Phi \vdash c : l_1 \to l_2}$

$$\frac{}{\emptyset \vdash \texttt{pure}\ f : l \to l}\ \text{PURE}$$

$$\frac{\Phi_1 \vdash c_1 : l_1 \to l_2 \quad \Phi_2 \vdash c_2 : l_3 \to l_4}{\Phi_1 \cup \Phi_2 \cup \{l_2 \sqsubseteq l_3\} \vdash c_1 \texttt{ >>> } c_2 : l_1 \to l_4}\ \text{SEQ}$$

$$\frac{\Phi \vdash c : l_1 \to l_2}{\Phi \vdash \texttt{op}\ c : l_1 \to l_2}\ \text{ONE}$$

$$\frac{\Phi_1 \vdash c_1 : l_1 \to l_2 \quad \Phi_2 \vdash c_2 : l_3 \to l_4}{\Phi_1 \cup \Phi_2 \vdash c_1\ \texttt{op}\ c_2 : l_1 \sqcap l_3 \to l_2 \sqcup l_4}\ \text{PAR}$$

$$\frac{\Phi \vdash c : l_1 \to l_2}{\Phi \cup \{l_2 \sqsubseteq l_1\} \vdash \texttt{loop}\ c : l_1 \to l_2}\ \text{LOOP}$$

$$\frac{}{\emptyset \vdash \texttt{tag}\ l : l \to l}\ \text{TAG}$$

$$\frac{}{\{l_1 \sqsubseteq \texttt{user}\} \vdash \texttt{declassify}\ l_1\ l_2 : l_1 \to l_2}\ \text{DECL}$$

$\boxed{\mathbb{L}, l_{user} \succ c : l_{in} \to l_{out}}$

$$\frac{\Phi \vdash c : l_1 \to l_2 \qquad l_{in} \sqsubseteq l_1 \quad l_2 \sqsubseteq l_{out} \qquad \mathbb{L} \vdash \Phi[l_{user}/\texttt{user}]}{\mathbb{L}, l_{user} \succ c : l_{in} \to l_{out}}\ \text{CERT}$$

Fig. 1. Information-flow type system implemented by the sublanguage

defined. The typing judgment $\Phi \vdash c : l_1 \to l_2$ is represented by the value:

```
FA c (Trans l1 l2)Φ
```

`FlowArrow` is implemented using a generic design of arrow transformers and it is parameterized by several types:

(1) The type l of security labels. `(FlowArrow l)` is an arrow transformer.
(2) The purely functional arrow `a` we are transforming from. The simplest and most common case of `a` is the function arrow `(->)`. The result `(FlowArrow l a)` is also an arrow.
(3) The input type `b` and output type `c`.

```
instance (Lattice l,Arrow a)=>Arrow (FlowArrow l a) where
  pure f = FA { computation = pure f   -- PURE --
              , flow = Flat
              , constraints = [] }
  (FA c1 f1 t1) >>> (FA c2 f2 t2) =    -- SEQ --
      let (f,c) = flow_seq f1 f2 in
           FA { computation = c1 >>> c2
              , flow = f
              , constraints = t1 ++ t2 ++ c }
  first (FA c f t) =                   -- ONE --
      FA { computation = first c
         , flow = f
         , constraints = t }
  (FA c1 f1 t1) &&& (FA c2 f2 t2) =    -- PAR --
      FA { computation = c1 &&& c2
         , flow = flow_par f1 f2
         , constraints = t1++t2 }
  (FA c1 f1 t1) *** (FA c2 f2 t2) =    -- PAR --
      FA { computation = c1 *** c2
         , flow = flow_par f1 f2
         , constraints = t1++t2 }
instance (Lattice l, ArrowChoice a) =>
              ArrowChoice (FlowArrow l a) where
  left (FA c f t) =                    -- ONE --
     FA { computation = left c
        , flow = f
        , constraints = t }
  (FA c1 f1 t1) +++ (FA c2 f2 t2) =    -- PAR --
     FA { computation = c1 +++ c2
        , flow = flow_par f1 f2
        , constraints = t1++t2 }
  (FA c1 f1 t1) ||| (FA c2 f2 t2) =    -- PAR --
     FA { computation = c1 ||| c2
        , flow = flow_par f1 f2
        , constraints = t1++t2 }
instance (Lattice l, ArrowLoop a) =>
              ArrowLoop (FlowArrow l a) where
 loop (FA c f t) =                     -- LOOP --
   let t' = constraint_loop f in
     FA { computation = loop c
        , flow = f
        , constraints = t ++ t' }
   where
   constraint_loop Flat = []
   constraint_loop (Trans l1 l2) = [LEQ l2 l1]
```

Fig. 2. Implementation of arrow operations

```
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
```

Fig. 3. Implementation of arrow operations (continued)

The arrow `a` must be purely functional and must have no side effects. Although `FlowArrow` is a generic arrow transformer, we do require that the arrow `a` represent a purely functional computation where information flows from one end to the other, so the information-flow types in the form of $l_1 \to l_2$ makes sense. In the rest of the paper, the reader can assume `a` is the function arrow `(->)` for ease of understanding.

`(FlowArrow l a)` is an arrow, so we can use `FlowArrow` as a sublanguage to represent computation. At the same time, `(FlowArrow l a)` also encodes a typing judgment, so we can verify the information-flow policies for the computation. Essentially, the implementation of `(FlowArrow l a)` is a type checker: each arrow operation implements a typing rule for that operation. For standard arrow operations on `a`, `FlowArrow` lifts them by (1) running the original operation on the `computation` fields of arguments, and (2) computing the flow types and constraints using such information from the arguments.

The implementation of `FlowArrow` is given in Figures 2 and 3. In the definition of each arrow operation, the operation in `FlowArrow` (on the left hand side) is implemented using operations in the arrow `a` (on the right hand side).

The `pure` operation returns a `Flat` flow type and no constraints, because such computations have no information-flow policies on them. It implements the PURE typing rule. `Flat` represents a flow type $l \to l$, but the label l never appears in the implementation—it can always be inferred from context.

The `(>>>)` operation sequentially composes two arrow computations. The flow types and constraints are computed using the `flow_seq` function, which implements the SEQ typing rule.

The `first` and `left` operations implement the ONE typing rule. The `(&&&)`, `(***)`, `(|||)` and `(+++)` operations are parallel compositions and they implement the PAR rule. The `flow_par` function is used to compute parallel composition of flow types.

The `loop` operation is slightly more interesting. It implements the LOOP typing rule. Since `loop` connects the output of a computation back to its input, we generate a constraint to capture this information flow. This requires that the input and the output have the same security level, unless there is declassification inside the computation.

As described in Section 2, the `do`-syntax of the secure sublanguage is translated to these standard arrow operations. Therefore, the typing judgment for code written in the `do`-syntax can be derived by combining the typing rules implemented in these standard arrow operations. For the translation of conditional commands shown in the end of Section 2, the combination of typing rules yields essentially the same constraints as the following COND rule found in conventional information-flow type systems:

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : l_1 & \Gamma \vdash e_2 : l_2 \\ \Gamma \vdash e_3 : l_3 & l_1 \sqsubseteq l_2 \sqcup l_3\end{array}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : l_2 \sqcup l_3}\ \text{COND}$$

3.4 Policy specification

So far, the typing rules implemented in `FlowArrow` permit the construction of computations from smaller components while composing their information-flow policies. Pure computations are given the $l \rightarrow l$ flow type, but we need a way to introduce more interesting flow types. The `tag` operation annotates a computation with a security label. It implements the TAG rule in Figure 1.

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

When `tag` is applied to a label l, it creates an arrow that represents an empty step of computation, with the flow type $l \rightarrow l$. Intuitively, `tag` inserts a "pipe" in the middle of the computation, with explicit flow types specified on both ends. For example, to annotate the confidential value `secret` with label `HIGH`, we can use the following code which has a flow type `HIGH` $\rightarrow$ `HIGH`:

```
pure (\_->secret) >>> tag HIGH
```

To assert that the output of a computation `c` has no confidential information, we can simply use the following code to connect `c` to a "low" pipe:

```
c >>> tag LOW
```

For confidentiality policies, we care about the future of secret computation, i.e. where information will flow to. Therefore, a good pattern for protecting confidentiality is to append `tag` to the output of the computation we want to protect. The design of `tag` and the arrow types are completely symmetrical. If we have labels for integrity policies, we can connect the output of `tag` to computations that require trustworthy data as inputs. This is a bi-directional design that works for both confidentiality and integrity policies.

3.5 Declassification

Declassification is a practical requirement for language-based information-flow control. We need a mechanism to allow information flow from high levels to low levels, but only in controlled ways. The *decentralized label model* (DLM) (Myers and Liskov, 2000) solves this problem by assigning code with *authority*. Each declassification statement can only weaken the security policy that belongs to the authority of code. In our arrows framework, there is no difficulty of encoding the labels in DLM, but we need a declassification mechanism that takes code authority into account.

For simplicity of presentation, we designed the declassification construct and its corresponding flow constraints for simple lattices such as the `TriLabel` lattice implemented earlier in this section. It is simpler than the declassification mechanism in Jif, but it implements the essence of code authority checking. This design can be generalized to any finite lattice.

```
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 `declassify` operation implements the DECL rule in Figure 1. It is similar to the `tag` operation except that it constructs a "pipe" where the security level of the output is lower than the level of input. Similar to other operations, `declassify` does not check the policies directly, but it creates a constraint which can be checked later. When applied to two label values, `declassify` l_1 l_2 creates an arrow with flow type $l_1 \rightarrow l_2$ and a flow constraint `USERGEQ` l_1 stating that the code privilege must be at least l_1. For example, if the code privilege is `HIGH`, it can declassify information from `MEDIUM` to `LOW`. But if the code privilege is `MEDIUM`, it cannot declassify from `HIGH` to `LOW`.

Finally, we need to check the flow types and constraints that we have accumulated during the construction of a secure computation. Since we have a declassification mechanism which takes code privilege into account, the code privilege must be provided to check the constraints.

```
data Privilege 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
```

The judgment $\mathbb{L}, l_{user} \succ c : l_{in} \rightarrow l_{out}$ states the security property to be checked: given a security lattice $\mathbb{L}$ and the label l_{user} representing the code privilege, does the arrow computation c satisfy the information-flow policy $l_{in} \rightarrow l_{out}$? The CERT rule in Figure 1 checks this security property and the `certify` function implements this rule.

The `certify` operation takes a few arguments:

(1) The information-flow types l_{in} and l_{out} that we expect the computation to have. Suppose the flow type of the secure computation is $l_1 \rightarrow l_2$, `certify` calls another function to verify that $l_{in} \sqsubseteq l_1$ and $l_2 \sqsubseteq l_{out}$.
(2) The code privilege l_{user}, under which the computation is performed. The `certify` operation checks all the constraints that come with the secure computation. For any constraints of the form `USERGEQ` l, a check is performed to make sure that $l \sqsubseteq l_{user}$. If any constraint is not satisfied, a run-time error is generated.
(3) A `FlowArrow` value that includes the secure computation to be checked. If all the above checks are successful, the embedded secure computation is returned. Note that we stacked the arrow transformer `FlowArrow` on another arrow `a`, this `certify` operation strips `FlowArrow` off and gives back computations in arrow `a`.

Although `certify` is a dynamic enforcement mechanism (it executes as part of the Haskell program), it provides strong security guarantees. When an embedded computation is certified, its whole control structure is examined using an information-flow analysis before any part of embedded computation is performed. This process is like type checking the embedded sublanguage.

Branches over arrow computations can only be constructed using the operators provided in `FlowArrow`.[4] The control structure of a secure computation is independent of the values generated in the computation. Therefore, if the run-time check fails, the failure does not leak information about secrets inside the computation.

A minor caveat is that recursive arrow computations should be constructed using the `loop` operation rather than using standard Haskell recursion. The `certify` function checks the flow types and constraints of the whole computation, so it forces evaluation of all arrow operations used to construct the computation. If the computation is recursively constructed using standard Haskell recursion, `certify` will essentially try to check an infinite control flow graph with an infinite typing derivation, which will exhaust Haskell's stack space and eventually abort the program. In such cases secret information is not leaked, but the secure computation should be re-written using the `loop` operation.

The `certify` interface seems verbose, but it is very flexible to use. For protecting confidentiality, we only care about the security level of the output, so the argument l_{in} can always be `label_bottom`. We also require all secrets be declassified to the lowest security level before reaching the output channel, so we let the argument l_{out} always be `label_bottom`. Thus, we hide the definition of `certify` and define a simpler operation `cert`:

```
cert = certify label_bottom label_bottom
```

3.7 Code privileges

When using `certify`, it is important that the code privilege is correctly specified: untrusted code cannot call `certify` using a code privilege that it does not have. Our solution is to define an abstract data type `Privilege` that internally stores a label as code privilege. The `certify` operation takes values of the abstract type `Privilege` as its input.

```
data Privilege l = PR l
```

The key point is to make the constructor `PR` only available in trusted modules. The program must be organized such that untrusted code can only treat the `Privilege` type abstractly. Privileges can only be created in trusted code and passed to untrusted code.

[4] Importantly, `FlowArrow` does not implement a richer interface such as the `ArrowApply` type class that would make it impossible to analyze the control structure.

Developing appropriate design patterns for structuring privileged code is an important task that we leave to future work. An interesting question, as with all capability-based authorization mechanisms, is how to revoke the privileges passed to untrusted code. If the untrusted code has state and runs under several privileges in different places, it can steal privileges by storing and reusing them. One solution is to encode version numbers in such privileges and have a global state to indicate valid privileges, doing so would require the top level code be inside a monad.

4 Example use of the embedded language

This section presents the features of our secure embedded sublanguage using code examples. All examples use the three-point security lattice encoding in Section 3. First, we present some simple program fragments to show how information-flow policies can be specified in programs. Then, we use a larger application to demonstrate declassification and policy enforcement.

4.1 Programming with information-flow policies

The `FlowArrow` interface is designed to be generic, but it is fairly verbose to use. To make programs look more concise, we define a type abbreviation for the common uses of `FlowArrow`. The type `Protected a` represents a secure computation that takes no input and produces the output of type `a`.

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

By default, protected computations constructed by `pure` have no information-flow constraints. Information-flow policies can be specified by using the `tag` operation. The function `tag_val` takes an arbitrary computation `x` and a security label `l` as inputs, converts `x` to a protected closure using `pure`, and composes it with an information-flow annotation using `>>>` and `tag`. The output is a protected computation with an output label `l`.

```
tag_val :: a -> TriLabel -> Protected a
tag_val x l = pure (\_ -> x) >>> tag l
cH = tag_val 3 HIGH
cM = tag_val 4 MEDIUM
cL = tag_val 5 LOW
```

Using `tag_val`, we can define `cH, cM, cL` as protected values with different information-flow policies. The following shows some computation using these protected values:

```
t1 = liftA2 (+) cL cM
t2 = liftA2 (*) cH cM
t3 = proc () -> do
      h <- cH -< ()
      if h>3 then do x <- cM -< ()
                     returnA -< x
             else do x <- t1 -< ()
                     returnA -< x
```

The `liftA2` function is a generic arrow operation that can be used in our sublanguage to convert any standard binary operator to an operator on protected types: `t1` is the sum of `cL` and `cM`, `t2` is the product of `cH` and `cM`. The definition of `t3` uses the `do`-syntax of arrows. Informally speaking, it represents the computation `if cH>3 then cM else t1`.

The security sublanguage rigorously captures both explicit and implicit information flows in protected computations. When the above code is executed, `t1` will have label `MEDIUM`, while `t2` and `t3` will have label `HIGH`. The control flow of the protected computation is represented using operations provided by the sublanguage, and these operations keep track of the information flow policies and constraints incrementally during the construction of protected computations.

Any protected computation can be used together with the `tag` operation to restrict the information flow. The function `expects_medium` takes a protected computation `c` as argument and requires the output of `c` to be no higher than `MEDIUM`:

```
expects_medium :: Protected a -> Protected a
expects_medium c = c >>> tag MEDIUM
```

Now, we can use this function with protected computations:

```
success1 = expects_medium t1
failure2 = expects_medium t2
failure3 = expects_medium t3
```

The first computation `success1` is fine, because `t1` has label `MEDIUM`. The second and the third both violate the information-flow policies, because `t2` and `t3` both have label `HIGH` while `MEDIUM` is expected: information flows from `HIGH` to `MEDIUM`. If we try to certify these three compuatations, the first will pass the certification (and hence be executable) but the latter two fill fail certification (and hence not be executable).

This subsection uses an interactive application to demonstrate a more realistic use of the security sublanguage. It simulates an online network service in which users can log in to access information. There are only two kinds of users: guests and administrators. Guests have security level `LOW` while administrators have security level `HIGH`. we use the type abbreviation `Priv` for code privileges:

```
type Priv = Privilege TriLabel
```

In this application, guests can enter numbers as price bids, while the administrator can log in to see the highest bid. The information-flow policy is that guests are not allowed to know what the highest bid is. To implement this policy, we maintain the highest bid as a global state `stat` with security level `HIGH`. The following code shows a simple session for guest services.

```
guest_service::Priv->(Protected Int)->IO(Protected Int)
guest_service priv stat = do
{  putStrLn "Enter␣a␣number:";
   i <- getNumber;
   let stat' = proc () -> do
     {  x <- stat -< ();
        if i>x then returnA -< i
               else returnA -< x;
     }
   return stat';
}
```

The function `guest_service` takes two arguments: `priv` has type `Priv` and it represents a code privilege passed to this function; `stat` has type `Protected Int` and it is the secret global state. The function returns a new state, which also has type `Protected Int`. The main body of the guest service is written in Haskell's standard IO monad; when it runs, it reads a number from input and updates the state if the input `i` is larger than the protected state `stat`.

The body of the `let` expression is the computation written in our embedded sublanguage. It uses the `do`-syntax for arrow operations, and the Haskell compiler translates code in the `do`-syntax to the standard arrow operations, which are overloaded by our sublanguage using Haskell type classes.

In the `do` block, there are two commands. The first command "`x <- stat -< ();`" binds the value of the secret computation `stat` to a local variable `x`, where `x` has type `Int`. Now, `x` can be freely used in any computation, but it cannot escape the scope of the `do`-block. The next command "`if..then..else..`" performs a conditional branching in the sublanguage. The body of the branch "`returnA -< i`" generates the output for the `do`-block. Finally, the computa-

tion represented by the "`proc ...do`" block is bound to the variable `stat'` and it is returned as the result.

The function `admin_service` implements a similar session for administrator services. It has the same type signature as `guest_service`.

```
admin_service: Priv->(Protected Int)->IO(Protected Int)
admin_service priv stat = do
{   let low = stat >>> (declassify HIGH LOW) in
    let summary = cert priv low () in
    putStrLn (show summary);

    let stat_new = (pure (\_->0) >>> tag HIGH) in
    return stat_new;
}
```

In contrast to the previous function, `admin_service` uses the combinators provided by the sublanguage directly, without using the `do`-syntax. In the first `let` expression, it declassifies the protected state to `LOW` level using the `declassify` operation and binds the result to the variable `low`. The variable `low` also has type `Protected Int`, but the security level associated with it is `LOW` after the declassification. Then, it uses the `cert` operation, together with its code privilege `priv`, to access the computation protected in `low`. The result `summary` has an unrestricted `Int` type and thus can be used as any other common Haskell value. The next line calls the standard Haskell printing function `putStrLn` to send this value to the program output. Finally, it creates a new protected value `stat_new` initialized to 0, sets the security policy of this protected value to be `HIGH`, and returns it as the new global state.

The `service_loop` function is part of the trusted computing base. It authenticates users and dispatches to the appropriate service.

```
service_loop::(Protected AuthDB)->(Protected Int)->IO()
service_loop auth_db stat = do
{
  putStrLn "Enter␣username␣and␣password:";
  u <- getLine; p <- getLine;
  let (ident,priv) = authenticate auth_db u p
  ;
  stat_new <- case ident of
    "admin" -> admin_service priv stat;
    "guest" -> guest_service priv stat;
     _ -> do {
               putStrLn "login␣error";
               return stat;
             };
  service_loop auth_db stat_new;
```

```
}
```

On every loop, it reads a user name and a password from the input, and authenticates the user. The variable `auth_db` contains the authentication database, which is a list of user names, passwords and code privileges. The `authenticate` function searches for the current user in the authentication database and retrieves the corresponding code privilege. Once a code privilege is available, it is used to execute the corresponding service function.

The top-level `main` function is also part of the trusted computing base. It creates the authentication database and the initial global state, both are tagged by the security label `HIGH`.

```
main = do
{
  let auth_db :: (Protected AuthDB) =
       (
         pure (\_-> [("admin","admin",HIGH),
                     ("guest","guest",LOW)] ) >>>
         tag HIGH
       )
      secret_val :: (Protected Int) =
       (
          pure (\_->0) >>>
          tag HIGH
       )
  in
  service_loop auth_db secret_val;
}
```

Security guarantees:

To see how this code enforces our information-flow policies, suppose an untrusted programmer adds the following code in `guest_service` to declassify the secret state:

```
let s = cert priv (stat >>> declassify HIGH LOW) ()
```

Or, suppose the services are incorrectly dispatched:

```
  stat_new <- case ident of
   "admin" -> guest_service priv stat   — should be admin_service
   "guest" -> admin_service priv stat   — should be guest_service
```

In such cases, when the program tries to declassify the data and certify the result using the guest privilege, a run-time error will be generated. The global state is tagged with the label `HIGH`, but guests can only acquire `LOW` privileges during the authentication process. The declassification operation requires that

the user privilege must be higher than the security level of the data to be declassified. Therefore, a guest cannot declassify the global state to `LOW` and use `cert` to steal the secret state. This provides a security mechanism similar to that of using *run-time principals* (Tse and Zdancewic, 2004a).

Aside from the authentication process and initial setup of confidential state, the information-flow policies are automatically enforced throughout the system. The `guest_service` and the `admin_service` are very simple in this example, but they can be scaled to more complex services, system states and security policies.

The application program is a fine-grained mixture of normal components written in standard Haskell and secure components constructed using a few special operations from the sublanguage. At a glance, it may be hard to distinguish where the "embedded" language ends and the "base" language begins, but that is part of the point—the programmer has easy access to both the strong security guarantees of the embedded language *and* the full power of Haskell at the same time: all the Haskell language features and software libraries are still available.

5 Formalizing the security guarantee

This section studies the formal security guarantees of the embedded `FlowArrow` sublanguage. Because our sublanguage is used in a purely functional setting, the type system shown in Figure 1 is fairly simple. Instead of reasoning about all the information flow channels in a program, it only checks the information flow of an individual channel, which is represented by a purely functional computation.

The traditional notion of *noninterference* is also simplified in this setting. Noninterference requires that the low outputs of a program do not depend on high inputs. In the purely functional programming style, this is simply saying that high computations cannot be certified for use at low output channels. The addition of declassification makes the security guarantee more interesting. Informally, the security guarantee can be stated as: "code running at privilege l_p cannot observe information of label l if $\neg(l \sqsubseteq l_p)$."

Another question concerns the soundness of the run-time checking mechanism, i.e. the implementation of `cert`. We need to formally prove that the checking mechanism itself is not a source of information leakage.

$$
\begin{array}{rrl}
\text{term} & e ::= & () \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{fix}\ e \\
& & \mid \mathtt{c}_i\ e \mid \mathtt{case}\ e\ e\ e \mid (e,e) \mid \pi_i\ e \qquad (i \in \{1,2\}) \\
& & \mid \mathtt{pure}\ e\ l \mid e \mathbin{\texttt{>>>}} e \mid e \mathbin{\texttt{\&\&\&}} e \mid e \mathbin{\texttt{|||}} e \mid \mathtt{loop}\ e \\
& & \mid \mathtt{tag}\ l \mid \mathtt{decl}\ l{\rightarrow}l \mid \mathtt{cert}\ e \\
& & \mid \mathtt{FA}(e, l{\rightarrow}l, \Phi) \mid \bullet \\
\text{value} & v ::= & () \mid x \mid \lambda x.e \mid \mathtt{c}_i\ e \mid (e,e) \mid \mathtt{FA}(e, l{\rightarrow}l, \Phi)
\end{array}
$$

Fig. 4. Formalizing the sublanguage

$$
\begin{array}{l}
E ::= [] \mid E\ e \mid \mathtt{case}\ E\ e\ e \mid \pi_i\ E \mid \mathtt{cert}\ E \mid E\ \mathtt{op}\ e \mid v\ \mathtt{op}\ E \mid \mathtt{loop}\ E \\
\qquad\qquad (\mathtt{op} \in \{\texttt{>>>}, \texttt{|||}, \texttt{\&\&\&}\}) \\
E[(\lambda x.e_1)\ e_2] \longrightarrow_p E[[e_2/x]e_1] \\
E[\mathsf{fix}\ e] \longrightarrow_p E[e\ (\mathsf{fix}\ e)] \\
E[\mathtt{case}\ (\mathtt{c}_i\ e)\ e_1\ e_2] \longrightarrow_p E[e_i\ e] \\
E[\pi_i\ (e_1, e_2)] \longrightarrow_p E[e_i] \\
E[\mathtt{pure}\ e\ l] \longrightarrow_p E[\mathtt{FA}(e, l{\rightarrow}l, \emptyset)] \\
E[\mathtt{tag}\ l] \longrightarrow_p E[\mathtt{FA}(\lambda x.x, l{\rightarrow}l, \emptyset)] \\
E[\mathtt{decl}\ l_1{\rightarrow}l_2] \longrightarrow_p E[\mathtt{FA}(\lambda x.x, l_1{\rightarrow}l_2, \{\mathsf{usergeq}(l_1)\})] \\
E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1) \mathbin{\texttt{>>>}} \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)] \\
\longrightarrow_p E[\mathtt{FA}(\lambda x.e_2\ (e_1\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})] \\
E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1) \mathbin{\texttt{|||}} \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)] \\
\longrightarrow_p E[\mathtt{FA}(\lambda x.\mathtt{case}\ x\ (\lambda y.c_1\ (e_1\ x))\ (\lambda y.c_2\ (e_2\ x)), l_1 \sqcap l_2{\rightarrow}l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)] \\
E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1) \mathbin{\texttt{\&\&\&}} \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)] \\
\longrightarrow_p E[\mathtt{FA}(\lambda x.(e_1\ (\pi_1\ x), e_2\ (\pi_2\ x)), l_1 \sqcap l_2{\rightarrow}l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)] \\
E[\mathtt{loop}\ \mathtt{FA}(e, l_1{\rightarrow}l_2, \Phi)] \\
\longrightarrow_p E[\mathtt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.f\ (a, \pi_2\ (e\ a)))))\ x, l_1{\rightarrow}l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})]
\end{array}
$$

$$
\frac{l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi)}{E[\mathtt{cert}\ \mathtt{FA}(e, l_1{\rightarrow}l_2, \Phi)] \longrightarrow_p E[\mathtt{c}_1\ e]} \qquad \frac{\neg(l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi))}{E[\mathtt{cert}\ \mathtt{FA}(e, l_1{\rightarrow}l_2, \Phi)] \longrightarrow_p E[\mathtt{c}_2\ ()]}
$$

Fig. 5. Operational semantics

5.1 Language syntax and semantics

We formalize the security sublanguage as a simple call-by-need λ-calculus extended with sums, products and the primitive arrow operations we defined in `FlowArrow`, as shown in Figure 4. Two special syntax nodes are added to the language. $\mathtt{FA}(e, l_1 \rightarrow l_2, \Phi)$ represents the run-time representation of the `FlowArrow` data type and it does not appear in source programs. The special syntax node $\bullet$ is used to represent term erasure, which is explained below. The operational semantics of this language is formalized in Figure 5 using evaluation contexts. Note that the evaluation relation ($\longrightarrow_p$) is parameterized

$$
\begin{array}{l}
\mathsf{erase}_p(\bullet) = \bullet \\
\mathsf{erase}_p(\texttt{()}) = \texttt{()} \\
\mathsf{erase}_p(x) = x \\
\mathsf{erase}_p(\lambda x.\, e) = \lambda x.\, \mathsf{erase}_p(e) \\
\mathsf{erase}_p(e_1\ e_2) = \mathsf{erase}_p(e_1)\ \mathsf{erase}_p(e_2) \\
\mathsf{erase}_p(\mathsf{fix}\ e) = \mathsf{fix}\ (\mathsf{erase}_p(e)) \\
\mathsf{erase}_p(\mathsf{c}_i\ e) = \mathsf{c}_i\ \mathsf{erase}_p(e) \\
\mathsf{erase}_p(\texttt{case}\ e_1\ e_2\ e_3) = \texttt{case}\ \mathsf{erase}_p(e_1)\ \mathsf{erase}_p(e_2)\ \mathsf{erase}_p(e_3) \\
\mathsf{erase}_p(\texttt{(}e_1\texttt{,}e_2\texttt{)}) = \texttt{(}\mathsf{erase}_p(e_1)\texttt{,}\mathsf{erase}_p(e_2)\texttt{)} \\
\mathsf{erase}_p(e_1\ \texttt{op}\ e_2) = \mathsf{erase}_p(e_1)\ \texttt{op}\ \mathsf{erase}_p(e_2) \\
\mathsf{erase}_p(\texttt{(loop}\ e\texttt{)}) = \texttt{loop}\ (\mathsf{erase}_p(e)) \\
\mathsf{erase}_p(\texttt{tag}\ l) = \texttt{tag}\ l \\
\mathsf{erase}_p(\texttt{decl}\ l_1 \rightarrow l_2) = \texttt{decl}\ l_1 \rightarrow l_2 \\
\mathsf{erase}_p(\texttt{pure}\ e\ l) = \begin{cases} \texttt{pure}\ (\mathsf{erase}_p(e))\ l & : \quad l \sqsubseteq l_p \\ \texttt{FA}(\bullet, l \rightarrow l, \emptyset) & : \quad \text{otherwise} \end{cases} \\
\mathsf{erase}_p(\texttt{FA}(e, l_1 \rightarrow l_2, \Phi)) = \begin{cases} \texttt{FA}(\mathsf{erase}_p(e), l_1 \rightarrow l_2, \Phi) & : \quad l_2 \sqsubseteq l_p \wedge \mathsf{valid}(l_p, \Phi) \\ \texttt{FA}(\bullet, l_1 \rightarrow l_2, \Phi) & : \quad \text{otherwise} \end{cases} \\
\mathsf{erase}_p([]) = [] \\
\mathsf{erase}_p(E\ e) = \mathsf{erase}_p(E)\ \mathsf{erase}_p(e) \\
\mathsf{erase}_p(\texttt{case}\ E\ e_1\ e_2) = \texttt{case}\ \mathsf{erase}_p(E)\ \mathsf{erase}_p(e_1)\ \mathsf{erase}_p(e_2) \\
\mathsf{erase}_p(\pi_i\ E) = \pi_i\ \mathsf{erase}_p(E) \\
\mathsf{erase}_p(\texttt{cert}\ E) = \texttt{cert}\ \mathsf{erase}_p(E) \\
\mathsf{erase}_p(E\ \texttt{op}\ e) = \mathsf{erase}_p(E)\ \texttt{op}\ \mathsf{erase}_p(e) \\
\mathsf{erase}_p(v\ \texttt{op}\ E) = \mathsf{erase}_p(v)\ \texttt{op}\ \mathsf{erase}_p(E) \\
\mathsf{erase}_p(\texttt{(loop}\ E\texttt{)}) = \texttt{loop}\ (\mathsf{erase}_p(E))
\end{array}
$$

Fig. 6. Term erasure and the modified semantics

by the code privilege p. In the operational semantics, the predicate $\mathsf{valid}(l_p, \Phi)$ means that all the label constraints in Φ are valid under code privilege p: it checks that the expected ordering of security labels are all correct and that the user has enough privilege to run the code.

$$\mathsf{valid}(l_p, \Phi) \triangleq (\forall \mathsf{leq}(l_1, l_2) \in \Phi.\ l_1 \sqsubseteq l_2) \wedge (\forall \mathsf{usergeq}(l) \in \Phi.\ l \sqsubseteq l_p)$$

For ease of presentation, this formal language is made simpler than the actual `FlowArrow` implementation. The major differences are:

- Label inference. In the actual `FlowArrow` implementation, the `pure` operation creates a flow type $l \rightarrow l$ where the label l is polymorphic and it can always be inferred from the context when `FlowArrow` values are combined. Our formal

$$\mathsf{good}_p(\texttt{()}) = \mathbb{T}$$
$$\mathsf{good}_p(x) = \mathbb{T}$$
$$\mathsf{good}_p(\lambda x.\, e) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(e_1\ e_2) = \mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2)$$
$$\mathsf{good}_p(\mathsf{fix}\ e) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(\mathsf{c}_i\ e) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(\texttt{case}\ e_1\ e_2\ e_3) = \mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2) \wedge \mathsf{good}_p(e_3)$$
$$\mathsf{good}_p(\texttt{(}e_1\texttt{,}e_2\texttt{)}) = \mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2)$$
$$\mathsf{good}_p(e_1\ \texttt{op}\ e_2) = \mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2)$$
$$\mathsf{good}_p(\texttt{loop}\ e) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(\texttt{tag}\ l) = \mathbb{T}$$
$$\mathsf{good}_p(\texttt{decl}\ l_1 \rightarrow l_2) = \mathbb{T}$$
$$\mathsf{good}_p(\texttt{cert}\ e) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(\texttt{pure}\ e\ l) = \mathsf{good}_p(e)$$
$$\mathsf{good}_p(\texttt{FA}(e, l_1 \rightarrow l_2, \Phi)) = \mathsf{good}_p(e) \wedge [(\neg(l_1 \sqsubseteq l_p) \wedge (l_2 \sqsubseteq l_p)) \Rightarrow \neg\mathsf{valid}(l_p, \Phi)]$$

Fig. 7. Invariant under evaluation

language in Figure 4 requires the label l be explicited annotated in the `pure` operation. This explicit annotation makes programming less convenient, but it makes the semantics less verbose and it does not affect the expressiveness and the security guarantee of the language.

- Label computations. In the actual implementation, labels are first-class terms and the program can manipulate labels in arbitrary ways. Our formal language does not permit interesting computations on labels, but the programmer still has the ability to manipulate labels in interesting ways. For example, instead of writing `tag` (`case` $e\ \lambda x.l_1\ \lambda x.l_2$), the programmer can write `case` e ($\lambda x.$`tag` l_1) ($\lambda x.$`tag` l_2). For any finite security lattice, this simplification indeed does not make a difference in terms of expressiveness.

5.2 Erasure and the modified semantics

To help formalize the security guarantee of this language, we use the technique of term erasing: a "useless" part of a term is rewritten to a special syntax node "•". Figure 6 shows the definition of erasure for terms and evaluation contexts. The idea is that any protected computation with output labels higher than the current code privilege can be safely erased, because information from such a computation cannot be observed during code execution. Also, any protected computation with unsatisfiable label constraints can also be erased because the `cert` operation will eventually eliminate them. These ideas are made precise in Figure 6, where the interesting cases are for `pure` and `FA`—the remaining cases simply extend the $\mathsf{erase}_p(-)$ function homomorphically over terms. It is worth noting that the pure rule is redundant: it could

be given as $\mathsf{erase}_p(\mathtt{pure}\ e\ l) = \mathtt{pure}\ (\mathsf{erase}_p(e))\ l$. We state the rule in the form in Figure 6 to simplify the proof of Theorem 5.3.1.

We also need to define an alternate evaluation relation ($\Longrightarrow_p$) for erased terms. The erased terms evaluate in the same way as normal terms, except that after every step of evaluation, the result is erased again, as shown in the following definition:

$$\frac{e_1 \longrightarrow_p e_2}{e_1 \Longrightarrow_p \mathsf{erase}_p(e_2)}$$

The semantics of ($\Longrightarrow_p$) guarantees that "useless" `FA` nodes are erased as soon as they are created.

5.3 The security guarantee

Our strategy is to formalize the security guarantee by establishing a simulation between normal term evaluation ($\longrightarrow_p$) and erased term evaluation ($\Longrightarrow_p$). The intuition is that a program runs as if its high-security components are erased. Such components are never used, so high-security information is not leaked. Lemma 5.3.1 establishes this simulation, Lemma 5.3.2 generalizes the simulation to multiple steps and Theorem 5.3.1 use the simulation to prove a noninterference-like result.

The crucial part of this strategy is to connect the meaning of the information-flow type system we implemented in Figure 1 with the operational semantics: how can we know that the type system guarantees that all high-security computations are erased during evaluation? To do so, we need to formally define what terms are "well-formed" and prove that the evaluation relation ($\longrightarrow_p$) preserves this predicate as an invariant.

Figure 7 defines an invariant predicate $\mathsf{good}_p()$ that is preserved by the evaluation relation ($\longrightarrow_p$), as shown in Lemma 5.3.1. It states that for all the records $\mathtt{FA}(e, l_1 \rightarrow l_2, \Phi)$ created at run time, if the flow $l_1 \rightarrow l_2$ is not permitted using the current code privilege, then the constraint Φ must be unsatisfiable. Intuitively, this invariant specifies the semantics of the simple information-flow type system we implemented. If $l \sqsubseteq l_p$ then l is considered as a low security label; otherwise it is considered as a high security label. The invariant $\mathsf{good}_p()$ permits the following kinds of information flow: from low to low, from high to high, from low to high, but it disallows the information flow from high to low (by generating unsatisfiable constraints).

Using this definition, Lemma 5.3.1 and Lemma 5.3.2 are restricted to only well-formed terms that satisfy the invariant $\mathsf{good}_p()$. As shown later in the

proof of Lemma 5.3.1, this restriction is necessary to establish the simulation between $(\longrightarrow_p)$ and $(\Longrightarrow_p)$.

We begin with propositions about the validity of constraint sets.

Proposition 5.3.1 (Properties of label constraints)

(1) $\mathsf{valid}(l_p, \Phi \cup \Phi') \Rightarrow \mathsf{valid}(l_p, \Phi)$
(2) $\neg(\mathsf{valid}(l_p, \Phi)) \Rightarrow \neg(\mathsf{valid}(l_p, \Phi \cup \Phi'))$

Proof: By definition of valid.

We next establish some simple propositions about the compositional behavior of $\mathsf{erase}_p(-)$.

Proposition 5.3.2 (Properties of term erasure)

(1) $\mathsf{erase}_p(E[e]) = \mathsf{erase}_p(E)[\mathsf{erase}_p(e)]$
(2) $\mathsf{erase}_p([e_2/x]e_1) = [\mathsf{erase}_p(e_2)/x]\mathsf{erase}_p(e_1)$
(3) $\mathsf{erase}_p(\mathsf{erase}_p(e)) = \mathsf{erase}_p(e)$
(4) $\mathsf{erase}_p(\mathsf{erase}_p(E)) = \mathsf{erase}_p(E)$
(5) $$\frac{\mathsf{erase}_p(E)[e_1] \longrightarrow_p \mathsf{erase}_p(E)[e_2]}{\mathsf{erase}_p(E)[e_1] \Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(e_2)]}$$

Proof: By induction on terms and evaluation contexts.

The following standard propositions express that the operational semantics of our simple language is deterministic and that the erased version of the evaluation relation is also deterministic.

Proposition 5.3.3 (Deterministic evaluation)

(1) For any term e, there is a unique E and a unique e' such that $e = E[e']$.
(2) The evaluation relation $(\longrightarrow_p)$ is deterministic.
(3) The evaluation relation $(\Longrightarrow_p)$ is determnistic.

Proof: By induction on terms and evaluation contexts.

Finally, we need some inversion principles related to the $\mathsf{good}_p(-)$ predicate.

Proposition 5.3.4 (Properties of the invariant)

(1) Inversion holds for each rule in the definition of $\mathsf{good}_p()$*:*
  (a) $\mathsf{good}_p(\lambda x.e) \Rightarrow \mathsf{good}_p(e)$
  (b) $\mathsf{good}_p(e_1\ e_2) \Rightarrow \mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2)$
  (c) ...
(2) $\mathsf{good}_p(E[e]) \Rightarrow \mathsf{good}_p(e)$.

(3) $\mathsf{good}_p(E[e_1]) \wedge \mathsf{good}_p(e_2) \Rightarrow \mathsf{good}_p(E[e_2])$.
(4) $\mathsf{good}_p(e_1) \wedge \mathsf{good}_p(e_2) \iff \mathsf{good}_p([e_2/x]e_1)$.

Proof: By definition of $\mathsf{good}_p(-)$ and induction on terms and evaluation contexts.

Putting together the propositions above, we can establish the following simulation relation between the language and its erased version. The lemma simultaneously proves that $\mathsf{good}_p(-)$ actually is an invariant.

Lemma 5.3.1 (Single-step simulation) *If* $\mathsf{good}_p(e_1)$ *and* $e_1 \longrightarrow_p e_2$, *then* $\mathsf{good}_p(e_2)$ *and* $\mathsf{erase}_p(e_1) \Longrightarrow_p^* \mathsf{erase}_p(e_2)$.

Proof sketch: Case analysis on the evaluation rule used in $e_1 \longrightarrow_p e_2$.

(1) $E[(\lambda x.\, e_1)\ e_2] \longrightarrow_p E[[e_2/x]e_1]$.
Given $\mathsf{good}_p(E[(\lambda x.\, e_1)\ e_2])$, we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(e_1)$, $\mathsf{good}_p(e_2)$, $\mathsf{good}_p([e_2/x]e_1])$ and $\mathsf{good}_p(E[[e_2/x]e_1])$.
Now we need to prove that $\mathsf{erase}_p(E[(\lambda x.\, e_1)\ e_2]) \Longrightarrow_p^* \mathsf{erase}_p(E[[e_2/x]e_1])$:
$\mathsf{erase}_p(E[(\lambda x.\, e_1)\ e_2])$
$= \mathsf{erase}_p(E)[(\lambda x.\, \mathsf{erase}_p(e_1))\ \mathsf{erase}_p(e_2)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p([\mathsf{erase}_p(e_2)/x]\mathsf{erase}_p(e_1))]$
$= \mathsf{erase}_p(E)[[\mathsf{erase}_p(e_2)/x]\mathsf{erase}_p(e_1)]$
$= \mathsf{erase}_p(E[[e_2/x]e_1])$ (by Prop. 5.3.2)
Each step in the above derivation can be justified by Prop. 5.3.2.
(2) $E[\mathsf{fix}\ e] \longrightarrow_p E[e\ (\mathsf{fix}\ e)]$. Similar.
(3) $E[\mathtt{case}\ (\mathsf{c}_i\ e)\ e_1\ e_2] \longrightarrow_p E[e_i\ e]$. Similar.
(4) $E[\pi_i\ (e_1, e_2)] \longrightarrow_p E[e_i]$. Similar.
(5) $E[\mathtt{pure}\ e\ l] \longrightarrow_p E[\mathtt{FA}(e, l \rightarrow l, \emptyset)]$
Given $\mathsf{good}_p(E[\mathtt{pure}\ e\ l])$, we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(e)$. By definition we know that $\mathsf{good}_p(\mathtt{FA}(e, l \rightarrow l, \emptyset))$ always holds because l and l are the same label. Therefore we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(E[\mathtt{FA}(e, l \rightarrow l, \emptyset)])$.
Now we need to prove $\mathsf{erase}_p(E[\mathtt{pure}\ e\ l]) \Longrightarrow_p^* \mathsf{erase}_p(E[\mathtt{FA}(e, l \rightarrow l, \emptyset)])$.
(a) If $l \sqsubseteq l_p$:
$\mathsf{erase}_p(E[\mathtt{pure}\ e\ l])$
$= \mathsf{erase}_p(E)[\mathtt{pure}\ \mathsf{erase}_p(e)\ l]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(\mathsf{erase}_p(e), l \rightarrow l, \emptyset))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\mathsf{erase}_p(e), l \rightarrow l, \emptyset)]$
$= \mathsf{erase}_p(E[\mathtt{FA}(e, l \rightarrow l, \emptyset)])$

(b) If $\neg(l \sqsubseteq l_p)$:
$\mathsf{erase}_p(E[\mathtt{pure}\ e\ l])$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\bullet, l \rightarrow l, \emptyset)]$
$\Longrightarrow_p^* \mathsf{erase}_p(E)[\mathtt{FA}(\bullet, l \rightarrow l, \emptyset)]$ (0 steps)

$= \mathsf{erase}_p(E[\mathtt{FA}(e, l{\rightarrow}l, \emptyset)])$

(6) $E[\mathtt{tag}\ l] \longrightarrow_p E[\mathtt{FA}(\lambda x.x, l{\rightarrow}l, \emptyset)]$. Similar to $E[\mathtt{pure}\ e\ l]$.

(7) $E[\mathtt{decl}\ l_1{\rightarrow}l_2] \longrightarrow_p E[\mathtt{FA}(\lambda x.x, l_1{\rightarrow}l_2, \{\mathsf{usergeq}(l_1)\})]$. Similar to previous cases except we need to verify that this rule indeed preserves $\mathsf{good}_p()$.

(8) $E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)]$
$\longrightarrow_p E[\mathtt{FA}(\lambda x.e_2\ (e_1\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})]$.

This is the most interesting rule. Let us start from proving the invariant. Given $\mathsf{good}_p(E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)])$, we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1))$, $\mathsf{good}_p(\mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2))$, $\mathsf{good}_p(e_1)$,$\mathsf{good}_p(e_2)$ and $\mathsf{good}_p(\lambda x.e_2\ (e_1\ x))$. Then we use a case analysis to prove that the invariant holds:

(a) If $\neg(\neg(l_1 \sqsubseteq l_p) \wedge (l_4 \sqsubseteq l_p))$: by definition, we have $\mathsf{good}_p(\mathtt{FA}(\lambda x.e_2\ (e_1\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\}))$, so we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(E[\mathtt{FA}(\lambda x.e_2\ (e_1\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})])$.

(b) If $\neg(l_1 \sqsubseteq l_p) \wedge (l_4 \sqsubseteq l_p)$: we use a case analysis to prove that $\neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})$, thus the invariant is preserved.

(i) If $l_2 \sqsubseteq l_p$:

(A) If $l_3 \sqsubseteq l_2$: by transitivity, $l_3 \sqsubseteq l_p$. Now we have $\mathsf{good}_p(\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1))$, $\neg(l_1 \sqsubseteq l_p)$ and $(l_3 \sqsubseteq l_p)$, we can derive $\neg\mathsf{valid}(l_p, \Phi_1)$. By Prop.5.3.1, $\neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})$.

(B) If $\neg(l_3 \sqsubseteq l_2)$: by definition, $\neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})$.

(ii) If $\neg(l_2 \sqsubseteq l_p)$: we have $\mathsf{good}_p(\mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2))$, $\neg(l_2 \sqsubseteq l_p)$ and $(l_4 \sqsubseteq l_p)$, we can derive $\neg\mathsf{valid}(l_p, \Phi_2)$. By Prop.5.3.1, $\neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})$.

Now we proceed to prove the $\Longrightarrow_p$ part of the lemma.

(a) If $\neg(l_4 \sqsubseteq l_p \wedge \mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\}))$: this is the easy case because the erasure of the result must be in the erased form $\mathtt{FA}(\bullet, \ldots, \ldots)$.

$\mathsf{erase}_p(E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)])$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\ldots, l_1{\rightarrow}l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(\ldots, l_2{\rightarrow}l_4, \Phi_2)]]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(\ldots, l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\}))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\bullet, l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})]$
$= \mathsf{erase}_p(E[\mathtt{FA}(\lambda x.e_2\ (e_1\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})])$

(b) If $l_4 \sqsubseteq l_p \wedge \mathsf{valid}(l_p, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})$: by Prop.5.3.1, $\mathsf{valid}(l_p, \Phi_1)$, $\mathsf{valid}(l_p, \Phi_2)$ and $l_3 \sqsubseteq l_2$.

(i) If $l_2 \sqsubseteq l_p$: by transitivity, $l_3 \sqsubseteq l_p$. The two sub-terms on the left-hand side are not erased:

$\mathsf{erase}_p(E[\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2)])$
$= \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(e_1, l_1{\rightarrow}l_3, \Phi_1))\ \mathtt{>>>}\ \mathsf{erase}_p(\mathtt{FA}(e_2, l_2{\rightarrow}l_4, \Phi_2))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\mathsf{erase}_p(e_1), l_1 \rightarrow l_3, \Phi_1)\ \mathtt{>>>}\ \mathtt{FA}(\mathsf{erase}_p(e_2), l_2 \rightarrow l_4, \Phi_2)]]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(\lambda x.\mathsf{erase}_p(e_2)\ (\mathsf{erase}_p(e_1)\ x), l_1{\rightarrow}l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\}))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\lambda x.\mathsf{erase}_p(e_2)\ (\mathsf{erase}_p(e_1)\ x), l_1 \rightarrow l_4, \Phi_1 \cup \Phi_2 \cup$

$\{\mathsf{leq}(l_3, l_2)\})]$
$= \mathsf{erase}_p(E[\mathtt{FA}(\lambda x.\, e_2\ (e_1\ x), l_1 \rightarrow l_4, \Phi_1 \cup \Phi_2 \cup \{\mathsf{leq}(l_3, l_2)\})])$

(ii) If $\neg(l_2 \sqsubseteq l_p)$: we have $\neg(l_2 \sqsubseteq l_p)$, $(l_4 \sqsubseteq l_p)$ and $\mathsf{valid}(l_p, \Phi_2)$, which contradicts with our assumption that $\mathsf{good}_p(\mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2))$. So this case cannot happen.

(9) $E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)]$
$\longrightarrow_p E[\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (e_1\ x))\ (\lambda y.\, c_2\ (e_2\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)]$.

Suppose $\mathsf{good}_p(E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)])$, we can use Prop. 5.3.4 to prove $\mathsf{good}_p(\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1))$, $\mathsf{good}_p(\mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2))$, $\mathsf{good}_p(e_1)$ and $\mathsf{good}_p(e_2)$. Now we only need to prove that $\mathsf{good}_p(\mathtt{FA}(\ldots, l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))$.

(a) If $l_1 \sqsubseteq l_p$ and $l_2 \sqsubseteq l_p$: then $l_1 \sqcap l_2 \sqsubseteq l_p$, so by definition, $\mathsf{good}_p(\mathtt{FA}(\ldots, l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))$.

(b) If $\neg(l_1 \sqsubseteq l_p)$:

(i) If $l_3 \sqsubseteq l_p$: by $\mathsf{good}_p(\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1))$ we know that $\neg\mathsf{valid}(l_p, \Phi_1)$. By Prop. 5.3.1, $\neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2)$. Therefore, $\mathsf{good}_p(\mathtt{FA}(\ldots, l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))$.

(ii) If $\neg(l_3 \sqsubseteq l_p)$: because $l_3 \sqsubseteq l_3 \sqcup l_4$, it must be the case that $\neg(l_3 \sqcup l_4 \sqsubseteq l_p)$. Therefore $\mathsf{good}_p(\mathtt{FA}(\ldots, l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))$.

(c) If $\neg(l_2 \sqsubseteq l_p)$: the dual case.

Now we proceed to prove the $\Longrightarrow_p$ part of the lemma.

(a) If $l_3 \sqsubseteq l_p \wedge l_4 \sqsubseteq l_p \wedge \mathsf{valid}(l_p, \Phi_1) \wedge \mathsf{valid}(l_p, \Phi_2)$: then $l_3 \sqcup l_4 \sqsubseteq l_p$ and $\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2)$.
$\mathsf{erase}_p(E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)])$
$= \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1))\ \mid\mid\mid\ \mathsf{erase}_p(\mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\mathsf{erase}_p(e_1), l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(\mathsf{erase}_p(e_2), l_2 \rightarrow l_4, \Phi_2)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (\mathsf{erase}_p(e_1)\ x))\ (\lambda y.\, c_2\ (\mathsf{erase}_p(e_2)\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (\mathsf{erase}_p(e_1)\ x))\ (\lambda y.\, c_2\ (\mathsf{erase}_p(e_2)\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)]$
$= \mathsf{erase}_p(E[\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (e_1\ x))\ (\lambda y.\, c_2\ (e_2\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)])$

(b) If $\neg(l_3 \sqsubseteq l_p) \vee \neg\mathsf{valid}(l_p, \Phi_1)$: then $\neg(l_3 \sqcup l_4 \sqsubseteq l_p) \vee \neg\mathsf{valid}(l_p, \Phi_1 \cup \Phi_2)$.
$\mathsf{erase}_p(E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)])$
$= \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1))\ \mid\mid\mid\ \mathsf{erase}_p(\mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\bullet, l_1 \rightarrow l_3, \Phi_1)\ \mid\mid\mid\ \mathtt{FA}(\ldots, l_2 \rightarrow l_4, \Phi_2)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (\bullet\ x))\ (\lambda y.\, c_2\ (\ldots\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2))]$
$= \mathsf{erase}_p(E)[\mathtt{FA}(\bullet, l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)]$
$= \mathsf{erase}_p(E[\mathtt{FA}(\lambda x.\, \mathtt{case}\ x\ (\lambda y.\, c_1\ (e_1\ x))\ (\lambda y.\, c_2\ (e_2\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)])$

(c) If $\neg(l_4 \sqsubseteq l_p) \vee \neg\mathsf{valid}(l_p, \Phi_2)$: the dual case.

(10) $E[\mathtt{FA}(e_1, l_1 \rightarrow l_3, \Phi_1)\ \mathtt{\&\&\&}\ \mathtt{FA}(e_2, l_2 \rightarrow l_4, \Phi_2)]$
$\longrightarrow_p E[\mathtt{FA}(\lambda x.\, (e_1\ (\pi_1\ x), e_2\ (\pi_2\ x)), l_1 \sqcap l_2 \rightarrow l_3 \sqcup l_4, \Phi_1 \cup \Phi_2)]$.
Similar to the previous case.

(11) $E[\texttt{loop FA}(e, l_1 \rightarrow l_2, \Phi)] \longrightarrow_p E[\texttt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.\, f\ (a, \pi_2\ (e\ a))))\ x, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})]$

It is easy to verify that the invariant holds. For the $(\longrightarrow_p)$ relation there are two cases:

(a) If $\neg(l_2 \sqsubseteq l_1) \vee \mathsf{erase}_p(\texttt{FA}(e, l_1 \rightarrow l_2, \Phi)) = \texttt{FA}(\bullet, l_1 \rightarrow l_2, \Phi)$: then $\mathsf{erase}_p(\texttt{FA}(\ldots, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})) = \texttt{FA}(\bullet, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})$. So we can prove:

$\mathsf{erase}_p(E[\texttt{loop FA}(e, l_1 \rightarrow l_2, \Phi)])$
$= \mathsf{erase}_p(E)[\texttt{loop FA}(\bullet, l_1 \rightarrow l_2, \Phi)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\texttt{FA}(\ldots, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\}))]$
$= \mathsf{erase}_p(E)[\texttt{FA}(\bullet, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})]$
$= \mathsf{erase}_p(E[\texttt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.\, f\ (a, \pi_2\ (e\ a))))\ x, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})])$

(b) If $l_2 \sqsubseteq l_1 \wedge \mathsf{erase}_p(\texttt{FA}(e, l_1 \rightarrow l_2, \Phi)) = \texttt{FA}(\mathsf{erase}_p(e), l_1 \rightarrow l_2, \Phi)$:

$\mathsf{erase}_p(E[\texttt{loop FA}(e, l_1 \rightarrow l_2, \Phi)])$
$= \mathsf{erase}_p(E)[\texttt{loop FA}(\mathsf{erase}_p(e), l_1 \rightarrow l_2, \Phi)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\texttt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.\, f\ (a, \pi_2\ (\mathsf{erase}_p(e)\ a))))\ x, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\}))]$
$= \mathsf{erase}_p(E)[\texttt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.\, f\ (a, \pi_2\ (\mathsf{erase}_p(e)\ a))))\ x, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})]$
$= \mathsf{erase}_p(E[\texttt{FA}(\lambda x.(\pi_1\ (\mathsf{fix}\ \lambda p.\lambda a.\, f\ (a, \pi_2\ (e\ a))))\ x, l_1 \rightarrow l_2, \Phi \cup \{\mathsf{leq}(l_2, l_1)\})])$

(12) $$\frac{l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi)}{E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)] \longrightarrow_p E[\mathsf{c}_1\ e]}.$$

Suppose $\mathsf{good}_p(E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)])$, we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(e)$ and $\mathsf{good}_p(E[\mathsf{c}_1\ e])$. Given the condition $l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi)$, we can prove:

$\mathsf{erase}_p(E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)])$
$= \mathsf{erase}_p(E)[\texttt{cert FA}(\mathsf{erase}_p(e), l_1 \rightarrow l_2, \Phi)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathsf{c}_1\ \mathsf{erase}_p(e))]$
$= \mathsf{erase}_p(E)[\mathsf{c}_1\ \mathsf{erase}_p(e)]$
$= \mathsf{erase}_p(E[\mathsf{c}_1\ e])$

(13) $$\frac{\neg(l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi))}{E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)] \longrightarrow_p E[\mathsf{c}_2\ ()]}$$

Suppose $\mathsf{good}_p(E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)])$, we can use Prop. 5.3.4 to prove that $\mathsf{good}_p(E[\mathsf{c}_2\ ()])$. Given the condition $\neg(l_2 \sqsubseteq \bot \wedge \mathsf{valid}(l_p, \Phi))$, we can prove:

$\mathsf{erase}_p(E[\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi)])$
$= \mathsf{erase}_p(E)[\mathsf{erase}_p(\texttt{cert FA}(e, l_1 \rightarrow l_2, \Phi))]$
$= \mathsf{erase}_p(E)[\texttt{cert FA}(\ldots, l_1 \rightarrow l_2, \Phi)]$
$\Longrightarrow_p \mathsf{erase}_p(E)[\mathsf{erase}_p(\mathsf{c}_2\ ())]$
$= \mathsf{erase}_p(E[\mathsf{c}_2\ ()])$

□

Lemma 5.3.2 (Multiple-step simulation) *If* $\mathsf{good}_p(e_1)$ *and* $e_1 \longrightarrow_p^* e_2$, *then* $\mathsf{good}_p(e_2)$ *and* $\mathsf{erase}_p(e_1) \Longrightarrow_p^* \mathsf{erase}_p(e_2)$.

Proof: Straightforward induction using Lemma 5.3.1. □

Now we have established a simulation between term evaluation ($\longrightarrow_p$) and erased term evaluation ($\Longrightarrow_p$), we can proceed to formalize the security guarantee. Intuitively, secret information cannot be leaked because the program runs as if such information is erased. Theorem 5.3.1 gives a noninterference-like security guarantee:

Theorem 5.3.1 (Security guarantee) *If* $\neg(l \sqsubseteq l_p)$, e *has no FA or* $\bullet$ *syntax nodes and* $i, j \in \{1, 2\}$, *then*

$$\forall e_1.\forall e_2.\ (e\ (\texttt{pure}\ e_1\ l) \longrightarrow_p^* \mathsf{c}_i\ ()) \wedge (e\ (\texttt{pure}\ e_2\ l) \longrightarrow_p^* \mathsf{c}_j\ ()) \Rightarrow i = j$$

This theorem states that, if an input of the program e has a security label l higher than the current code privilege, then there is no information flow from that input to the output of the program execution using the current code privilege. Note that the operation `cert` can be used freely in the program, this theorem also guarantees that the result of `cert` does not leak information about the secret values.

Proof: Because e has no FA or $\bullet$ syntax nodes, it is easy to verify that $\mathsf{good}_p(e\ (\texttt{pure}\ e_1\ l))$ and $\mathsf{good}_p(e\ (\texttt{pure}\ e_2\ l))$. Using Lemma 5.3.2, we know that $\mathsf{erase}_p(e\ (\texttt{pure}\ e_1\ l)) \Longrightarrow_p^* \mathsf{erase}_p(\mathsf{c}_i\ ())$ and $\mathsf{erase}_p(e\ (\texttt{pure}\ e_2\ l)) \Longrightarrow_p^* \mathsf{erase}_p(\mathsf{c}_j\ ())$. However, $\mathsf{erase}_p(e\ (\texttt{pure}\ e_1\ l)) = \mathsf{erase}_p(e)\ \texttt{FA}(\bullet, l \rightarrow l, \emptyset)$ and $\mathsf{erase}_p(e\ (\texttt{pure}\ e_2\ l)) = \mathsf{erase}_p(e)\texttt{FA}(\bullet, l \rightarrow l, \emptyset)$, so we have $\mathsf{erase}_p(e)\texttt{FA}(\bullet, l \rightarrow l, \emptyset) \Longrightarrow_p^* \mathsf{c}_i\ ()$ and $\mathsf{erase}_p(e)\texttt{FA}(\bullet, l \rightarrow l, \emptyset) \Longrightarrow_p^* \mathsf{c}_j\ ()$. By Prop. 5.3.3, the evaluation relation $\Longrightarrow_p$ is deterministic, the term $\mathsf{erase}_p(e)\ \texttt{FA}(\bullet, l \rightarrow l, \emptyset)$ can only be normalized to one value, so $i = j$. □

6 Discussion and future work

6.1 Compile time vs. run time

For applications written using the embedded language, there are two stages of type-checking. At compile time, the base language (in this case, Haskell) is type checked and compiled. At run time, the embedded language is type checked before embedded secure computations are executed. Therefore, the information-flow policy violations are not detected until the application is

launched. Although the sublanguage uses static analysis techniques and provides similar strong security guarantees, this two-stage mechanism is sometimes not as convenient as specialized languages such as Jif. Each run of the application may only use part of the secure computations, so debugging can be more difficult. Therefore, it is appealing to have a one-stage, compile-time enforcement mechanism for the sublanguage. Such a mechanism can be possible if it is built entirely in the static type system of the host language.

Abadi et. al. developed the *dependency core calculus* (DCC) (Abadi et al., 1999), which uses a hierarchy of monads to model information flow. Tse and Zdancewic (Tse and Zdancewic, 2004b) translated DCC to System F and showed that noninterference can be translated to a more generic property called *parametricity*, which states that polymorphic programs behave uniformly for all their instantiations. An intuitive demonstration of this idea is that abstract data types can be used as a protection mechanism to hide high-security information. They presented a Haskell implementation where each security level is encoded using an abstract data type and binding operators are defined to compose computations with permitted information flows. This approach works well for simple lattices, but encoding the security lattice of n points would require $O(n^2)$ definitions for binding operators. This makes it difficult to implement more complex security lattices such as the *decentralized label model*.

The problem with this approach is policy *expressiveness*. The type system of the base language must be expressive enough to encode the syntax and the semantics of security policies. Although Haskell has an expressive type system, it is not clear how to encode more expressive policies directly in the type language — we leave that as a open question to investigate in the future.

6.2 Parallel composition and arrow axioms

We used the arrow interface to build the embedded language, but it remains to show that `FlowArrow` satisfies the appropriate arrow axioms. A quick check of the arrow axioms (Paterson, 2003) shows that the *exchange* axiom does not seem to hold. Let f have the flow type $l_1 \rightarrow l_2$ and g have the flow type $l_3 \rightarrow l_4$. There are two canonical ways to compose f and g in parallel using the `first` combinator. They should be equivalent:

`first` f `>>> pure (id` $\times$ g`) = pure (id` $\times$ g`) >>> first` f

However, our `FlowArrow` implementation yields the flow type $l_1 \rightarrow l_4$ with constraints $\{l_2 \sqsubseteq l_3\}$ on the left side and $l_3 \rightarrow l_2$ with $\{l_4 \sqsubseteq l_1\}$ on the right. This seems to violate the arrow axioms! Does our implementation make sense?

If we compose f and g naturally using the `(***)` operator, we get $l_1 \sqcap l_3 \rightarrow l_2 \sqcup l_4$ with no constraints, which is the least restrictive flow type. The types we get from using `first` are both more restrictive than this one. The problem with using `first` is that our analysis technique is not fine-grained enough—it reasons about information flow in a syntax-directed, end-to-end fashion that yields imprecise flow types for `first` f `>>> pure (id` $\times$ g`)`. This coarse analysis does not compromise the security guarantee because it always is conservative.

To justify that our arrow implementation satisfies the arrow axioms, we would need to give finer semantic interpretations to the flow types and constraints. Intuitively, although both sides of the *exchange* axiom are over-restrictive and have different types, they can be considered equivalent in the sense that they are both *sound*: using such flow types will not lead to acceptance of insecure programs.

The practical ramification of this imprecision is that although soundness is not affected by using `first`, programmers are encouraged to use the `(&&&), (***), (|||), (+++)` operations directly so that safe programs are more likely to be accepted. The use of `first, second, left, right` should be avoided whenever possible. Compared to `first`, `(&&&)` is a more intuitive operation for parallel composition because it resembles the product morphism in category theory. The prior work on arrows (Hughes, 2000) uses `first` as the primitive operation because it is simpler and it gives a definite evaluation order.

Another consequence of this imprecision is that the security analysis can be too restrictive for arrow computations written in the `do`-syntax, because Haskell implements some translation rules using `first` instead of `(&&&)`. Fortunately, conditional branches are translated using `(|||)`, so programmers can still write conditional branches in the natural way. In general, we need a more precise type system to avoid depending on particular implementations of the translation rules of the `do`-syntax.

6.3 DLM and practical applications

The declassification mechanism in this paper can be adapted to work with the *decentralized label model* (DLM) (Myers and Liskov, 2000), where the constraints on code authority are expressed using the act-for relation of principals. We are currently working on the encoding and integration of DLM in the arrows framework. Unlike Jif, where information-flow control are mostly-static, the arrows framework is a run-time mechanism, so the principals, the act-for hierarchy and the security lattice can all be dynamic. Such dynamic policies have long been sought in language-based information-flow security because they address practical requirements.

Once the dynamic DLM is implemented, it will be interesting to see how it works in real applications. An important benefit of our approach is that existing Haskell applications can be enhanced with information-flow control without complete rewriting. The programmers may proceed gradually by changing the representation of secure components while leaving most normal components untouched. It would be ideal if the security-sensitive computation only takes a small portion of the whole program, so information-flow policies can be globally enforced by a few local modifications to the program.

As mentioned in Section 3.7, there are still interesting open questions about the protection and revocation of code privileges. Moreover, the dynamic checking in our approach makes debugging more difficult because run-time errors are hard to observe, reproduce and locate. All of these problems need to be explored in the context of more concrete applications.

6.4 Implementing other type systems

Although `FlowArrow` is a generic arrow transformer, the type system implemented in `FlowArrow` only works with arrows that have no side-effects, because we assign a simple information-flow type $l_1 \rightarrow l_2$ to such arrow computations. This raises two questions. First, what arrows can be used besides the function arrow `(->)`? The stream processor arrow (Hughes, 2000) is one example: we can use `FlowArrow` to track information flow for stream processors, which map input streams to output streams. But in general, we need to formally state the properties of arrows that can be used with `FlowArrow`. Another question is how to modify the type system in `FlowArrow` so that it works for mutable state and other side-effects. We conjecture that for a specific effect such as mutable state, we can extend the type system implemented in `FlowArrow` and lift the arrow operations such as `get` and `put` to `FlowArrow` while implementing appropriate typing rules. The feasibility of this approach has yet to be studied, but one intriguing possibilitity is the potential to use multiple security type systems in one application at the same time.

7 Conclusion

Using an embedded sublanguage of arrows, end-to-end information-flow policies can be directly encoded and enforced in Haskell using modular library extensions, with a modest overhead of run-time checking. There is no need to modify the Haskell language, and this embedded sublanguage approach permits information-flow technology to be adopted gradually. The security mechanism is designed to be generic with respect to computation types and security

lattices. There is great flexibility in the choice of security policy frameworks and multiple policy frameworks can co-exist in the same program. Dynamic information-flow policies can be expressed, yet the security guarantee is as strong as that of static analysis. This paper has demonstrated one example of this embedded security-typed language approach, and established its proof of noninterference.

Acknowledgments

We thank Benjamin Pierce, Geoffrey Washburn, Stephanie Weirich, the other members of the PL Club at the University of Pennsylvania and the anonymous reviewers for their valuable suggestions and feedback about our work. Funding for this research was provided in part by NSF grants CNS-0346939 and CCF-0524035, and grant N00014-04-1-0725 provided by the ONR.

References

Abadi, M., Banerjee, A., Heintze, N., Riecke, J., Jan. 1999. A core calculus of dependency. In: Proc. 26th ACM Symp. on Principles of Programming Languages (POPL). San Antonio, TX, pp. 147–160.

Chapman, R., Hilton, A., 2004. Enforcing security and safety models with an information flow analysis tool. ACM SIGAda Ada Letters XXIV (4), 39–46.

Hughes, J., May 2000. Generalising monads to arrows. Science of Computer Programming 37, 67–111.

Li, P., Zdancewic, S., 2006. Encoding information flow in Haskell. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW'06).

Liang, S., Hudak, P., Jones, M., 1995. Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 333–343.

Myers, A. C., Jan. 1999. JFlow: Practical mostly-static information flow control. In: Proc. 26th ACM Symp. on Principles of Programming Languages (POPL). San Antonio, TX, pp. 228–241.

Myers, A. C., Liskov, B., 2000. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology 9 (4), 410–442.

Myers, A. C., Nystrom, N., Zheng, L., Zdancewic, S., Jul. 2001. Jif: Java information flow, software release. Located at http://www.cs.cornell.edu/jif.

Paterson, R., Sep. 2001. A new notation for arrows. In: International Conference on Functional Programming. ACM Press, pp. 229–240.

Paterson, R., 2003. Arrows and computation. In: Gibbons, J., de Moor, O. (Eds.), The Fun of Programming. Palgrave, pp. 201–222.

Peyton Jones, S., Augustsson, L., Barton, D., 2002. Haskell 98 language and libraries (the revised report). http://www.haskell.org/report.

Pottier, F., Simonet, V., Jan. 2002. Information flow inference for ML. In: Proc. 29th ACM Symp. on Principles of Programming Languages (POPL). Portland, Oregon.

Sabelfeld, A., Myers, A. C., Jan. 2003. Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21 (1), 5–19.

Sabelfeld, A., Sands, D., 2005. Dimensions and principles of declassification. In: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05). pp. 255–269.

Simonet, V., Mar. 2003. Flow Caml in a nutshell. In: Hutton, G. (Ed.), Proceedings of the first APPSEM-II workshop. Nottingham, United Kingdom, pp. 152–165.

Tse, S., Zdancewic, S., 2004a. Run-time Principals in Information-flow Type Systems. In: Proc. IEEE Symposium on Security and Privacy.

Tse, S., Zdancewic, S., 2004b. Translating Dependency into Parametricity. In: ACM International Conference on Functional Programming.

Wadler, P., August 1992. Monads for functional programming. In: Proceedings of the Marktoberdorf Summer School on Program Design Calculi.

Zheng, L., Myers, A. C., 2004. Dynamic security labels and noninterference. In: Proceedings of the Second Workshop on Formal Aspects in Security and Trust (FAST2004).