HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell

Pablo Buiras
Chalmers University of Technology, Sweden
buiras@chalmers.se

Dimitrios Vytiniotis
Microsoft Research, United Kingdom
dimitris@microsoft.com

Alejandro Russo [*]
Chalmers University of Technology, Sweden
russo@chalmers.se

Abstract

Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without disclosing it. IFC is typically enforced via type systems and static analyses or via dynamic execution monitors. The LIO Haskell library, originating in operating systems research, implements a purely dynamic monitor of the sensitivity level of a computation, particularly suitable when data sensitivity levels are only known at runtime. In this paper, we show how to give programmers the flexibility of deferring IFC checks to runtime (as in LIO), while also providing static guarantees—and the absence of runtime checks—for parts of their programs that can be statically verified (unlike LIO). We present the design and implementation of our approach, HLIO (Hybrid LIO), as an embedding in Haskell that uses a novel technique for deferring IFC checks based on singleton types and constraint polymorphism. We formalize HLIO, prove non-interference, and show how interesting IFC examples can be programmed. Although our motivation is IFC, our technique for deferring constraints goes well beyond and offers a methodology for programmer-controlled hybrid type checking in Haskell.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Security and Protection*]: Information flow controls

Keywords Information-flow control, hybrid typing, gradual typing, dynamic typing, data kinds, constraint kinds, singleton types

1. Introduction

Preserving confidentiality of data has become of extreme importance, particularly in complex systems where *untrusted* components require access to sensitive information (e.g. text messages, contact lists, pictures, etc.) in order to provide their functionality. Information-Flow Control (IFC) is a well-established approach for allowing untrusted code to manipulate sensitive data without *disclosing* it (Sabelfeld and Myers 2003). IFC essentially scrutinizes source code to track how data of different sensitivity levels flows within a program, where security alarms are raised when confidentiality might be at stake. IFC research has produced three mature compilers for secure programs: *Jif* (Myers and Liskov 2000) (based on Java), *FlowCaml* (Simonet 2003) (based on Caml and not developed any more), and *Paragon* (Broberg et al. 2013) (based on Java). Alternatively, IFC can be provided via simple libraries in Haskell where concepts like arrows and monads are repurposed to protect confidentiality (Li and Zdancewic 2006; Russo et al. 2008).

There exists a broad spectrum of enforcement mechanisms for IFC, ranging from fully dynamic ones, e.g., in the form of execution monitors (Austin and Flanagan 2009; Askarov and Sabelfeld 2009), to static ones, e.g., in the form of type systems (Volpano et al. 1996). Although dynamic and static techniques provide similar security guarantees (Sabelfeld and Russo 2009), there are many arguments for choosing dynamic over static approaches and vice versa. Several of these arguments have their roots in the long-term dispute between dynamic and static analyses, e.g., overhead vs. performance, enforcing properties for a program once and for all vs. monitoring properties in every run of a program, etc.

From the security point of view, specifically, there are good reasons to prefer dynamic over static approaches. Code statically verified to preserve confidentiality clearly adheres to data sensitivity levels and policies valid *at compile* time. However, data sensitivity levels may be entirely dynamic (e.g. we may read data from a trusted or a non-trusted domain at runtime) and even policies may change at runtime (e.g. principals (users) can change the set of principals they share data with by—for instance—altering their list of friends). In situations like this, the statically verified code has to be restructured to perform runtime checks in ways that the static analysis or the type system can understand and exploit to verify the program (we will see an example of that in Section 3.1). Alternatively, programs have to be written in a way that can statically deal with *all possible* sensitivity levels or policies that they could potentially encounter at runtime; this in turn may limit the set of useful side-effects programs can perform.

The LIO library (Stefan et al. 2011b) for Haskell offers a way of tackling this problem by providing a monad that dynamically enforces IFC. Borrowing ideas from operating systems research (VanDeBogart et al. 2007; Zeldovich et al. 2006), the LIO monad implements an execution monitor that keeps track of a *current* label to indicate the sensitivity level of the computation. The current label may get raised, or *tainted*, when the computation depends on sensitive data. Furthermore, sensitive computations are prevented from writing into public channels. In practice, LIO has proven suitable for building production secure web systems (Giffin et al. 2012).

There are plenty of opportunities to optimize away LIO runtime security checks. For example, it is enough to perform a *single check*

[*] Work done while visiting Stanford University

for computations that, within a long loop, attempt to write to the same channel without affecting the current label. Ideally, runtime checks should only be applied to those parts of the program where sensitive labels are unknown at compile time or susceptible to changes at runtime. Although a state-of-the-art tool, LIO does not support mixing static and dynamic IFC. In this work, we address this shortcoming.

We present HLIO, a Hybrid IFC library which combines the best of both approaches. HLIO statically protects confidentiality while allowing the programmers to *defer* selected checks to be done at runtime. In that manner, security checks involving statically-unknown or prone-to-change labels can be performed at runtime, while providing static guarantees for the rest of the code. Existing LIO code can easily be embedded in HLIO. Furthermore, HLIO provides a very similar interface to LIO. As a result, existing LIO code can also be incrementally refactored to work in HLIO so that programmers can obtain static guarantees where possible. The main purpose of HLIO is making a LIO-like IFC analysis hybrid rather than making LIO better in the kind of leaks it prevents. Specifically, our contributions with this paper are:

- We design and implement HLIO, a hybrid approach to IFC that allows programmers to defer IFC constraints to runtime. (Section 4)
- We present a novel technique for embedding HLIO as a library in Haskell. Our technique makes essential use of advanced features of the GHC type system and type inference, namely (a) singleton types (Eisenberg and Weirich 2012),(b) data promotion (Yorgey et al. 2012), and (c) constraint polymorphism[1], i.e., data types that can be parameterized over type class constraints, to enable deferring IFC checks to runtime. We remark that it is not necessary to understand these advanced type system features in order to use our library. (Sections 5 and 6)
- We formalize the core features of HLIO in a calculus that allows us to establish a simulation with LIO, thereby showing that HLIO cannot leak secrets, i.e., that it satisfies termination-insensitive noninterference. (Section 7)
- As an overall contribution, we describe a general-purpose mechanism for deferring static constraints without any compiler or language modifications. Those constraints can go well beyond IFC, and can even include ordinary type equalities emitted by GHC's type inference engine (see Section 8). We thus make it easier for programmers to move across the static/dynamic boundary, following the mantra of Meijer and Drayton "Static typing where possible, dynamic typing when needed!" (Meijer and Drayton 2005).

2. LIO: Flexible Dynamic IFC for Haskell

In this section, we briefly review LIO and its mechanism for dynamically protecting confidentiality of data.

Security lattices In an IFC system, data gets classified according to its sensitivity degree, which is often denoted by a security label (from now on, just labels). Formally, labels form a lattice *Label* to indicate the allowed flows of information within a program. Data associated with label ℓ_1 can flow into entities labeled as ℓ_2 provided that they respect the order relationship of the lattice, i.e., $\ell_1 \sqsubseteq \ell_2$. The encoding of security lattices can be given as a type class, providing join ($\sqcup$), and the order relationship ($\sqsubseteq$)—see Figure 1. In LIO, this type class also includes a meet ($\sqcap$) operation, but we exclude it from our definition since it is not important for our purposes.[2] Our running example is the classical two-point security lattice, *Label*, that introduces labels L (low) and H (high) to classify data as public and secret, respectively.

class $Lattice\ \alpha$ **where**
  $\sqcup :: \alpha \to \alpha \to \alpha$
  $\sqsubseteq :: \alpha \to \alpha \to Bool$

Figure 1. Security lattice

data $Label = L \mid H$
instance $Lattice\ Label$

The *Label* lattice implementation is what one expects; public data can flow into secret entities, i.e., $L \sqsubseteq H$, but not vice versa, i.e., $H \not\sqsubseteq L$.

The *LIO* ***monad*** LIO provides the *LIO* monad to guarantee that computations manipulate data according to the security lattice. In (Stefan et al. 2012b), this monad is parametric on the security lattice being considered, but we consider this lattice to be fixed to type *Label* to simplify exposition.

data $LIO\ a$
instance $Monad\ LIO$
$getLabel :: LIO\ Label$
$runLIO :: Label \to LIO\ a \to IO\ a$

Figure 2. *LIO* interface

It is expected that untrusted code is written using this monad (and not directly in the IO monad) in order to have some guarantees about its behavior—this can be enforced using other mechanisms (Terei et al. 2012). *LIO* encapsulates IO actions so that they are only executed when confidentiality is not compromised. To achieve that, the monad keeps track of a label $\ell_{\text{cur}} :: Label$, called the *current floating label* (or current label for short), which can be retrieved at any time by the function *getLabel*. The role of the current label is two-fold. Firstly, it implicitly labels all the data in scope. Secondly, it only allows computations to write to channels that are labeled with $\ell :: Label$ such that $\ell_{\text{cur}} \sqsubseteq \ell$; otherwise, *LIO* aborts execution. For instance, a computation $m :: LIO\ a$ with $\ell_{\text{cur}} = H$ indicates that a secret has already been observed by m—thus, m cannot subsequently write to public channels.

LIO computations have the flexibility to read sensitive data above the current label, but at the cost of raising the current label and thus being more restrictive in subsequent computations. More specifically, when reading data with sensitivity $\ell :: Label$, the current label ℓ_{cur} is raised to $\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell$—in the LIO terminology, the new current label *floats above the observed data*. Consequently, the current label protects *all the data* that have been observed.

Labeled expressions As in many other IFC systems, *LIO* provides abstractions to label data with different sensitivity degrees in a fine-grained manner—see Figure 3. Data type $Labeled\ a$ associates an expression of type a with a label in *Label*. The pure function *labelOf* can retrieve the label associated with a labeled expression. The functions *label* and *unlabel* are used to respectively create and destroy elements of this data type. Term $label\ \ell\ x$ creates a labeled expression which associates label ℓ with expression x, only if $\ell_{\text{cur}} \sqsubseteq \ell$. This constraint ensures that *LIO* computations do not allocate data below

data $Labeled\ a$
$labelOf :: Labeled\ a \to Label$
$label :: Label \to a$
  $\to LIO\ (Labeled\ a)$
$unlabel :: Labeled\ a \to LIO\ a$
$toLabeled :: Label \to LIO\ a$
  $\to LIO\ (Labeled\ a)$

Figure 3. Labeled expressions

[1] GHC 7.8.1 manual, Section 7.12

[2] Meet is normally used for tracking integrity, e.g. for checking that data has not been corrupted by untrusted parties.

the current label, which could potentially be returned and read by lower-labeled computations.

Term $\mathit{unlabel}\ x$ never fails; it extracts the data inside a labeled expression x but taints (as a side-effect) the current label by joining it ($\sqcup$) with the label of the expression. From a security point of view, creating a labeled expression with label ℓ can be regarded as writing into a channel at security level ℓ. Similarly, observing (i.e., unlabeling) a labeled expression is analogous to reading from a channel with the same security label. For simplicity, we only consider labeled expressions in this paper—they are the simplest examples of labeled entities. Nevertheless, LIO does support labeled mutable references (Stefan et al. 2011b), exceptions (Stefan et al. 2012b), and synchronization variables (Stefan et al. 2012a), which could be orthogonally added.

EXAMPLE 1. *(Tainting ℓ_{cur})* An LIO computation only raises its current label when observing (unlabeling) labeled expressions, as the secure string concatenation example below shows:

```
lconcat :: Labeled String → Labeled String → LIO String
lconcat lstr₁ lstr₂ = do    -- Initial current label $\ell_{cur}$
  str₁ ← unlabel lstr₁      -- $\ell'_{cur} = \ell_{cur} \sqcup (labelOf\ lstr_1)$
  str₂ ← unlabel lstr₂      -- $\ell''_{cur} = \ell'_{cur} \sqcup (labelOf\ lstr_2)$
  return (str₁ ++ str₂)     -- Final current label $\ell''_{cur}$
```

Label creep *Label creep* is the problem of raising the current label to a point where computations are no longer capable of performing useful side-effects (Sabelfeld and Myers 2003), i.e., the current label becomes "too high, too soon." To address this problem, LIO provides the primitive $\mathit{toLabeled}$ (Figure 3) to allow computations to only temporarily raise their current label. Specifically, $\mathit{toLabeled}\ \ell\ m$ executes m with the current label ℓ_{cur} at the time of executing this action. It first ensures that $\ell_{\text{cur}} \sqsubseteq \ell$ since it would attach ℓ to the result of m—after all, it is creating a labeled value. Computation m can in turn raise the current label during its execution, to a new ℓ'_{cur}. After m terminates, $\mathit{toLabeled}$ checks that $\ell'_{\text{cur}} \sqsubseteq \ell$, and if that is the case, label ℓ is used to protect the sensitivity of the result (in the return value of type $\mathit{Labeled}\ a$).

In $\mathit{toLabeled}\ \ell\ m$, ℓ is an upper bound on the final current label of m. The reason for that is to avoid leaks by manipulating the current label inside m (Stefan et al. 2011b). Imagine that the labeled value is instead wrapped with the final current label of m, and that the current label before executing $\mathit{toLabeled}$ is set to L. It could happen that in one run, the current label in m is ℓ_1, where $L \sqsubset \ell_1$, and depending on information at that level, it decides to unlabel a piece of data which takes the current label to ℓ_2 ($\ell_2 \not\equiv \ell_1$). After $\mathit{toLabeled}$ gets executed, the next instruction simply reads the label of the returned value ($\mathit{labelOf}$), which returns either ℓ_1 or ℓ_2 without raising the current label. In that manner, code with current label L can learn from data at level ℓ_1—an information leak!

EXAMPLE 2. *(Avoiding label creep)* With $\mathit{toLabeled}$ in place, we can provide a more flexible version of $\mathit{lconcat}$ as follows.

```
lconcat' :: Labeled String → Labeled String
              → LIO (Labeled String)
lconcat' lstr₁ lstr₂ = do     -- Initial current label $\ell_{cur}$
  let lab = labelOf lstr₁ ⊔ labelOf lstr₂
  lresult ← toLabeled lab (lconcat lstr₁ lstr₂)
  return lresult   -- Final current label $\ell_{cur}$
```

Observe that $\mathit{lconcat}'$, in contrast with $\mathit{lconcat}$, can concatenate secret strings without raising the current label.

Running LIO actions without leaking secrets Function runLIO uses its first argument to initialize the current label and executes the LIO action given as its second argument. It returns an IO action which is IFC-compliant, i.e., where side-effects do not leak sensitive information with respect to that label.

EXAMPLE 3. *(Preventing secret leaking)* We describe below a function which runs untrusted code and publishes a returned string value in a public web site.

```
publish :: LIO String → IO String
publish m = do {r ← runLIO L action; report r}
  where
    action = do
      x ← m
      lx ← label L x   -- succeeds if $\ell_{cur} \sqsubseteq L$
      unlabel lx   -- $\ell_{cur}$ is not modified
    report s = wget ("http://reports/str=" ++ s) [] []
```

Function wget sends an HTTP request to the URL given as argument. The action computation runs the untrusted code m but guards the result x with L by calling $\mathit{label}\ L$. This call only succeeds when the final label of m is less than or equal to L.

Dynamically labeled values As mentioned in the introduction, runtime IFC enforcement is particularly useful in systems where values get classified based on runtime information. For instance one can assume (or implement) a primitive that reads a remote labeled value from the network:

```
readRemote :: URI → LIO (Labeled String)
```

The primitive does not necessarily increase the current label as sensitive data can be encapsulated in the labeled value we return. A more realistic example of such a primitive can be found in the extended version of the paper (Buiras et al. 2015).

Untrusted scripts can freely call $\mathit{readRemote}$ without compromising confidentiality since, in order to observe the returned value, they would have to have their current label tainted and thus would be restricted from performing unsafe side-effects. While not a problem for dynamic LIO, we will see in the next section how dynamically labeled data complicates the programming model in a statically-typed IFC discipline.

3. SLIO: Static IFC for Haskell

LIO performs information-flow checks at runtime, and hence the ability to discharge those statically is certainly appealing.

Security labels at the type level The first step towards a statically typed version of LIO in Haskell is to transport labels and lattice operations over labels to the type level. We illustrate how this can be done in Haskell for the familiar 2-point lattice:

```
data Label = L | H
class Flows (ℓ₁ :: Label) (ℓ₂ :: Label)
instance Flows L L
instance Flows L H
instance Flows H H
type family Join (ℓ₁ :: Label) (ℓ₂ :: Label) :: Label where
  Join L L = L
  Join L H = H
  Join H L = H
  Join H H = H
```

The Label datatype constructors will be used at the type-level. In Haskell terminology, Label will be a *promoted* datatype (Yorgey et al. 2012). Moreover, we can represent $\sqsubseteq$-constraints at the type level using the type class $\mathit{Flows}\ (\ell_1 :: \mathit{Label})\ (\ell_2 :: \mathit{Label})$ over labels. The instances of the type class encode specific cases of the

$\sqsubseteq$-relationship. [3] We also use a *closed type family* (Eisenberg et al. 2014) *Join* to express the $\sqcup$ computation at the type level.

Ordinary term-level labels can now be indexed by type-level labels, i.e., they can be defined as *singleton types* in the dependent type theory jargon—see Figure 4.

$$
\begin{array}{l}
\textbf{data}\ \mathit{SLabel}\ (\ell :: \mathit{Label})\ \textbf{where} \\
\quad L :: \mathit{SLabel}\ L \\
\quad H :: \mathit{SLabel}\ H
\end{array}
$$

Figure 4. Singleton labels

In a proper dependently typed language, such as Agda or F*, there would be no need for duplication of labels and lattice functionality at the type level and, in fact, our formal treatment (Section 7) does away with the duplication.

Although our running example is the 2-point lattice, we have successfully applied similar techniques to implement a more complicated type-level lattice, namely DC-labels (Stefan et al. 2011a), a decentralized security label model for IFC that can express security concerns from different actors in a mutual distrust environment. As far as we know, this is the first implementation of DC-labels at the type level.

An LIO Hoare state monad Once the type-level machinery is in place, we replace our dynamic *LIO* monad with a *Hoare state monad* (Nanevski et al. 2006), indexed by the initial label of a computation (analogous to a pre-condition) and the final label of a computation (analogous to a post-condition):

$$
\begin{array}{l}
\textbf{data}\ \mathit{SLIO}\ (\ell_i :: \mathit{Label})\ (\ell_o :: \mathit{Label})\ a \\
\mathit{runSLIO} :: \mathit{SLabel}\ \ell_i \rightarrow \mathit{SLIO}\ \ell_i\ \ell_o\ a \rightarrow \mathit{IO}\ a
\end{array}
$$

SLIO is just an intermediate step towards our final solution, but readers can assume a very similar implementation as that of *LIO*: a state monad over the current label.

$$\textbf{type}\ \mathit{SLIO}\ \ell_i\ \ell_o\ a = \mathit{SLabel}\ \ell_i \rightarrow \mathit{IO}\ (a, \mathit{SLabel}\ \ell_o)$$

Due to its more expressive type, *SLIO* is not a Haskell monad. Nevertheless, it is a monad in the sense that it is possible to define meaningful $(\gg\!=)$ and *return* operators that satisfy the usual monad laws:

$$
\begin{array}{l}
(\gg\!=) :: \mathit{SLIO}\ \ell_1\ \ell_2\ a \rightarrow (a \rightarrow \mathit{SLIO}\ \ell_2\ \ell_3\ b) \rightarrow \mathit{SLIO}\ \ell_1\ \ell_3\ b \\
\mathit{return} :: a \rightarrow \mathit{SLIO}\ \ell\ \ell\ a
\end{array}
$$

It is easy to see how these functions are implemented.

A statically typed API for IFC *SLIO* so far seems like a more precise typing of *LIO*. However, the ability to express labels and their operations at the type level immediately opens up the possibility for converting the *dynamic checks* of LIO to *static proof requirements*. We do this below by simply rewriting the dynamic API to use static constraints instead:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Labeled}\ (\ell :: \mathit{Label})\ a = \mathit{Labeled}\ (\mathit{SLabel}\ \ell)\ a \\
\mathit{getLabel} :: \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{SLabel}\ \ell_i) \\
\mathit{labelOf} :: \mathit{Labeled}\ (\ell :: \mathit{Label})\ a \rightarrow \mathit{SLabel}\ \ell \\
\mathit{label} :: \mathit{Flows}\ \ell_i\ \ell \Rightarrow \mathit{SLabel}\ \ell \rightarrow a \\
\quad \rightarrow \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{Labeled}\ \ell\ a) \\
\mathit{unlabel} :: \mathit{Labeled}\ \ell\ a \rightarrow \mathit{SLIO}\ \ell_i\ (\mathit{Join}\ \ell_i\ \ell)\ a \\
\mathit{toLabeled} :: \mathit{SLIO}\ \ell_i\ \ell_o\ a \rightarrow \mathit{SLIO}\ \ell_i\ \ell_i\ (\mathit{Labeled}\ \ell_o\ a)
\end{array}
$$

Function *getLabel* returns the current label without affecting it. Function *labelOf* returns the singleton type corresponding to the initial label of the computation. Function *label* creates a labeled value with label ℓ without modifying the current label ℓ_i, provided that $\ell_i \sqsubseteq \ell$, expressed this time as a static proof obligation *Flows* $\ell_i\ \ell$. Function *unlabel*, on the other hand, *taints* the current label with the value of the labeled expression. Function *toLabeled* has a very simple type: just encapsulate the output label in the labeled value that we return. The careful reader may observe a small disconnect between the static and dynamic versions of *toLabeled*—this is due to a significant simplification that the static world enables, a point we discuss in detail in Section 8.

Finally, in order to give a valid type to primitives such as *readRemote*, it is often convenient to hide the label of a labeled value with an existential type, so that it no longer appears in the type. Haskell does not support first-class existential types, so we encode this with a datatype definition:

$$
\begin{array}{l}
\textbf{data}\ \mathit{LabeledX}\ a\ \textbf{where} \\
\quad \mathit{LabeledX} :: (\mathit{Labeled}\ (\ell :: \mathit{Label})\ a) \rightarrow \mathit{LabeledX}\ a
\end{array}
$$

3.1 Problems when programming in SLIO

Let us consider how one can program using the *SLIO* primitives. Suppose that we have a function *report* with type

$$\mathit{report} :: \mathit{Flows}\ \ell_i\ L \Rightarrow \mathit{String} \rightarrow \mathit{SLIO}\ \ell_i\ \ell_i\ ()$$

that sends a given *String* to a public server and publishes it on the Internet. This function has a *Flows* type class constraint which specifies that the current label at the time when *report* is run should not exceed L, i.e., the public label. For the simple lattice that we consider in this paper, *report* can effectively be called only when ℓ_i is L. One could imagine more complex situations with a richer label hierarchy, where more than one label is allowed to report or when the label associated with the public server is not fixed to L in advance but is rather dynamically obtained. Such situations would amplify our arguments in the rest of this section, but the simpler *report* above is sufficient for our presentation.

Figure 5 considers the secure string concatenation example *lconcat* (from the previous section), except that we instead use the statically typed counterparts to the LIO operations, and we incorporate a call to *report* in order to publish the result of the concatenation. This function, called *lReport2*, is a perfectly well-typed program with type

$$
\begin{array}{l}
\mathit{lReport2}\ \mathit{lstr}_1\ \mathit{lstr}_2 = \\
\textbf{do}\ v_1 \leftarrow \mathit{unlabel}\ \mathit{lstr}_1 \\
\quad\ v_2 \leftarrow \mathit{unlabel}\ \mathit{lstr}_2 \\
\quad\ \textbf{let}\ \mathit{result} = v_1 +\!\!+\ v_2 \\
\quad\ \mathit{report}\ \mathit{result} \\
\quad\ \mathit{return}\ \mathit{result}
\end{array}
$$

Figure 5. Static *lReport2*

$$
\begin{array}{l}
\mathit{lReport2} :: \mathit{Flows}\ (\mathit{Join}\ (\mathit{Join}\ \ell_i\ \ell_1)\ \ell_2)\ L \Rightarrow \\
\quad \mathit{Labeled}\ \ell_1\ \mathit{String} \rightarrow \mathit{Labeled}\ \ell_2\ \mathit{String} \\
\quad \rightarrow \mathit{SLIO}\ \ell_i\ (\mathit{Join}\ (\mathit{Join}\ \ell_i\ \ell_1)\ \ell_2)\ \mathit{String}
\end{array}
$$

Client scripts can call *lReport2* provided that they can satisfy the constraint, which enforces that both strings should be public, i.e., labeled with L. For instance, assume that we have $\mathit{lv}_1 :: \mathit{Labeled}\ L\ \mathit{String}$, $\mathit{lv}_2 :: \mathit{Labeled}\ L\ \mathit{String}$, and code

$$
\begin{array}{l}
\mathit{foo} :: \mathit{SLIO}\ L\ L\ \mathit{String} \\
\mathit{foo} = \mathit{lReport2}\ \mathit{lv}_1\ \mathit{lv}_2
\end{array}
$$

All labels are statically resolved, and *foo* can typecheck as all constraints can be discharged by the type class and type family instances.

Consider now the case where some of the labeled values are dynamically loaded from the network with *readRemote* from the previous section, and we furthermore address the label creep issue by packing the result in a labeled value:

[3] Although type classes in Haskell are *open*, we can prevent malicious users from introducing bogus instances by employing superclasses and Haskell's export mechanism.

```
readRemote :: URI → SLIO ℓ_i ℓ_i (LabeledX String)
foo = do
  LabeledX (lv_1 :: Labeled ℓ_1 String) ← readRemote host_1
  LabeledX (lv_2 :: Labeled ℓ_2 String) ← readRemote host_2
  toLabeled (lReport2 lv_1 lv_2)
```

The program is ill-typed for two reasons. First, the existential label variables ℓ_1 and ℓ_2, arising from unpacking the existentials that we read with *readRemote*, escape in the return type, i.e., *Labeled* (*Join* (*Join* ℓ_i ℓ_1) ℓ_2) *String*. To address this problem we could pack the return type in an existential (using *LabeledX*) to prevent the existential label from escaping. The modification is shown in Figure 6. However, even if we prevent the escape of existential variables in the return type of *foo*, there is another problem: the existential variables also escape *in the constraint*, i.e., *Flows* (*Join* (*Join* ℓ_i ℓ_1) ℓ_2) L, which makes *foo* ill-typed.

```
foo = do
  ...
  r ← toLabeled (lReport2 lv_1 lv_2)
    -- pack result in existential
  return (LabeledX r)
```

Figure 6. Hiding existential types

Since we do not statically know the remote labels, one may wonder if there is a way to rewrite the program to "assume the worst" (that they are both H) and that the current label after unlabelling them is always—conservatively—H. This option is a non-starter: first, *lReport2* would always be returning high-labeled values, but much more worryingly, we would not be in a position to call *report* any more, even in the case where the actually read remote labels were both L.

A more appealing way to implement *foo* is to restructure the code to incorporate a *runtime test* that inspects the remote labels:

```
foo = do
  LabeledX lv_1 ← readRemote host_1
  LabeledX lv_2 ← readRemote host_2
  case (labelOf lv_1, labelOf lv_2) of
    (L, L) → do
      lv ← toLabeled (lReport2 lv_1 lv_2) :: SLIO L L String
      return (LabeledX lv)
    _ → error "Both strings should be public!"
```

The GADT branch on the labels tests for a specific combination of remote labels, which allows the type checker to refine the corresponding type-level labels and discharge all generated constraints. We have also introduced annotations in each branch to fix the SLIO pre- and post-conditions and guide the type inference engine. In the case of the 2-point lattice, the above restructuring is not terrible (only one combination of 4 is a non-error), but more complicated lattices can quickly introduce lots of GADT pattern matches in potentially multiple places inside the user code.

The example illustrates one awkward aspect of the static approach: every time we have to move dynamic data into a statically typed piece of code, programs have to be restructured to introduce runtime tests. While the runtime tests in this situation are *unavoidable*, in this paper we show how to do this *without restructuring* the implementation.

4. HLIO: Mixing Static and Dynamic Typing

In HLIO, users can instead take the "natural" way to write *foo* and make the program typeable by using our primitive *defer* (underlined below):[4]

[4] We do not yet give type signatures since types slightly differ from the types of the corresponding primitives in SLIO.

```
foo host_1 host_2 = do
  LabeledX lv_1 ← readRemote host_1
  LabeledX lv_2 ← readRemote host_2
  lv ← defer (toLabeled (lReport2 lv_1 lv_2))
  return (LabeledX lv)
```

The role of *defer* is to defer static constraints to runtime; in this case, the one which arises from *toLabeled* (*lReport2* lv_1 lv_2). This constraint will be $\ell_i \sqcup \ell_1 \sqcup \ell_2 \sqsubseteq L$, where ℓ_i is the initial label and ℓ_1 and ℓ_2 are the labels of the returned labeled values from the two *readRemote* calls.

To demonstrate how this works, assume that *readRemote* returns a high-labeled value from host "secure.org", but a low-labeled value from "public.org". The following sequence of calls (using *runHLIO*, the *HLIO* analogue of *runSLIO*) shows that indeed our primitive performs the check at runtime:

```
ghci> runHLIO L (foo "secure.org" "public.org")
*** Exception: IFC violation!
ghci> runHLIO L (foo "public.org" "public.org")
Success
ghci> runHLIO H (foo "public.org" "public.org")
*** Exception: IFC violation!
```

In the first case, the first labeled value will contain a high label that taints the current label and results eventually in an IFC exception. In the second case, we only *readRemote* from public domains and hence no exception is thrown. In the final case, although we read from two public sites, we start from an already high label.

The *defer* primitive can be used at every point in the assembly of a computation to selectively defer to runtime the constraints arising from a subcomputation, *at the programmer's will*. For example, the following variations are all well-typed:

```
lvL :: Labeled L String    -- a statically known public value
bar x = do
  LabeledX lv ← readRemote host
  s_1 ← defer (toLabeled (lReport2 x lv))
  s_2 ← lReport2 x lvL
  return s_2
baz x = do
  LabeledX lv ← readRemote host
  s_1 ← defer (toLabeled (lReport2 x lv))
  s_2 ← defer (lReport2 x lvL)
  return s_2
```

The difference between *bar* and *baz* lies in the set of constraints they dynamically check; in *bar*, we have to statically discharge the constraints that arise from the computation of s_2, but we will dynamically check the constraints arising from *lReport2* x lv when computing s_1. In *baz*, we will convert the constraints from s_2 to be runtime checks. In both cases, we *must* defer the constraints that arise from the computation of s_1 as the label of lv would otherwise escape in the returned constraint.

The mechanism of *defer* has also the benefit of addressing the incompleteness of type inference engines or type-level lattice specifications—any time we are faced with a constraint that we cannot statically discharge, *defer* will convert it to a runtime check.

4.1 The HLIO API

Having described the functionality we are aiming for, we now present the HLIO API without yet diving into the internals of its implementation.

Label expressions Whenever a *getLabel* operation runs, we must produce a runtime representation of the current label, i.e., a singleton. Consider the case where the current label is of the form *Join* ℓ_1 ℓ_2. When ℓ_1 and ℓ_2 are known statically, we can just apply the type family and compute the resulting label. However, if ℓ_1 and ℓ_2 are existentially quantified, we need a way of computing a singleton for the *Join* by combining the singletons for ℓ_1 and ℓ_2. Therefore, it will be convenient to introduce another promoted datatype that captures unevaluated label *expressions* as well as a type family to reduce them to *Label* types. As we will see in Section 6, this additional level of indirection allows us to compute singletons for *Join* and also to defer constraints involving existentials.

```
data LExpr a = LVal a | LJoin (LExpr a) (LExpr a)
type family ℰ (ℓ :: LExpr Label) :: Label where
  ℰ (LVal x) = x
  ℰ (LJoin ℓ₁ ℓ₂) = Join (ℰ ℓ₁) (ℰ ℓ₂)
class Flows (ℰ ℓ₁) (ℰ ℓ₂) ⇒
  FlowsE (ℓ₁ :: LExpr Label) (ℓ₂ :: LExpr Label)
instance Flows (ℰ ℓ₁) (ℰ ℓ₂) ⇒ FlowsE ℓ₁ ℓ₂
data Labeled (ℓ :: LExpr Label) a =
  Labeled (SLabel (ℰ ℓ)) a
```

Data type *LExpr Label* captures unevaluated label expressions at the type level, and $\mathcal{E}$ reduces them to *Label* values. The type class *FlowsE* is isomorphic to *Flows*, with the exception that it ranges over *LExpr Label* instead of *Label*. Note that we also redefine the *Labeled* data type to include arbitrary labeled expressions. Type family *LJoin* encodes $\sqcup$ at the level of types.

***HLIO* monad** GHC introduces a kind *Constraint* to classify constraints and allows constraint polymorphism (Orchard and Schrijvers 2010). This means that ADTs can be parameterized over constraints. HLIO exploits this feature to provide a monad *HLIO* below:

```
data HLIO (c :: Constraint)
          (ℓᵢ :: LExpr Label) (ℓₒ :: LExpr Label) a
```

The *HLIO* datatype is very similar to *SLIO* except that it also records a constraint c :: *Constraint* (we motivate this design choice in Section 6). The rest of the HLIO API provides mechanisms to discharge these constraints statically or dynamically. A computation *HLIO* $c\ \ell_i\ \ell_o\ a$ should be read as a computation that, under constraint c and from initial label ℓ_i produces a value a and raises the current label to ℓ_o. The types of $(\gg\!=)$ and *return* show how constraints are collected:

```
(>>=) :: HLIO c₁ ℓ₁ ℓ₂ a
  → (a → HLIO c₂ ℓ₂ ℓ₃ b) → HLIO (c₁, c₂) ℓ₁ ℓ₃ b
return :: a → HLIO () ℓᵢ ℓᵢ a
```

Note that the type of $(\gg\!=)$ creates a *tuple* of constraints (c_1, c_2) by collecting constraints c_1 and c_2 from the sub-computations. The type of *return* collects a trivial constraint ().

IFC functionality HLIO provides the same API as *SLIO*:

```
labelOf :: Labeled ℓ a → SLabel (ℰ ℓ)
getLabel :: HLIO () ℓᵢ ℓᵢ (SLabel (ℰ ℓᵢ))
unlabel :: Labeled ℓ a → HLIO () ℓᵢ (LJoin ℓᵢ ℓ) a
label :: SLabel ℓ → a
          → HLIO (FlowsE ℓᵢ (LVal ℓ))
                  ℓᵢ ℓᵢ (Labeled (LVal ℓ) a)
toLabeled :: HLIO c ℓᵢ ℓₒ a
          → HLIO c ℓᵢ ℓᵢ (Labeled ℓₒ a)
```

Unlike in SLIO, *label* just records constraint *FlowsE* ℓ_i (*LVal* ℓ) in its result type—instead of actually constraining the whole type of the function. This is the only *HLIO* primitive that generates a constraint.

Deferring and simplifying constraints In addition to the core IFC functionality, *HLIO* adds the ability to *defer* collected constraints, or explicitly *simplify* them in one go:

```
defer :: Deferrable c ⇒ HLIO c ℓᵢ ℓₒ a → HLIO () ℓᵢ ℓₒ a
simplify :: c ⇒ HLIO c ℓᵢ ℓₒ a → HLIO () ℓᵢ ℓₒ a
```

The function *defer* accepts an *HLIO* computation that would be typeable under constraint c, and returns a computation that is typeable under *no constraint!* Indeed, the purpose of this combinator is to discharge the constraint by a runtime test. The puzzled reader may wonder how it is even possible to have a sound implementation of *defer*. The magic is in the *Deferrable* type class, which we describe in Section 5.

Dually to deferring constraints to runtime, we may require them to be *statically discharged*—function *simplify* allows us to do that. Like *defer*, *simplify* accepts an *HLIO* computation that is typeable under constraint c, and returns a computation that is typeable under the empty constraint provided that we can discharge c statically (hence the quantification $c \Rightarrow \ldots$).

Running* HLIO *computations Finally, the function that runs *HLIO* computations is analogous to *runSLIO*, except that we require the collected constraints to be provable.[5]

```
runHLIO :: c ⇒ SLabel ℓ → HLIO c (LVal ℓ) ℓₒ a → IO a
```

The rest of the paper In the rest of the paper, we describe the *Deferrable* class which enables us to implement the *defer* combinator (Section 5), and we present the design decisions and the implementation of the HLIO API (Section 6). We formalize the core features of HLIO as a calculus and prove non-interference by elaboration to (ordinary) *LIO* (Section 7). We discuss other applications of *Deferrable* beyond IFC (Section 8).

5. Deferrable Constraints

To understand the implementation of HLIO, we first dive into the internals of *Deferrable*. For a given constraint c, an instance of *Deferrable* c defines a single function *deferC*:

```
class Deferrable (c :: Constraint) where
  deferC :: forall a.Proxy c → (c ⇒ a) → a
```

The *Proxy* c argument is a commonly used technique to get around the lack of explicit type applications in the Haskell source language—instead, we provide a never-evaluated *Proxy* c argument that we can provide an annotation for, e.g., *deferC* $(\perp$ :: *Proxy* (*C Int*)) m.

The second argument, $c \Rightarrow a$, represents a computation that can only be executed if we can statically satisfy the constraint c. The return type of defer is plainly the result of that computation.

It should (rightly so) seem impossible to implement an instance of *Deferrable* for *every* possible constraint c. However, we can provide instances for specific constraints, provided we have enough runtime information around. In what follows, we show how to provide an instance for *FlowsE*. We start by creating a type-class capturing a singleton label:

```
class ToSLabel (ℓ :: LExpr Label) where
  slabel :: LProxy ℓ → SLabel (ℰ ℓ)
instance ToSLabel (LVal H) where slabel _ = H
instance ToSLabel (LVal L) where slabel _ = L
instance (ToSLabel ℓ₁, ToSLabel ℓ₂)
```

[5] Alternatively, we could equally require that c be simply *Deferrable*, or that c be () and make use of the appropriate *defer* or *simplify* combinators when constructing an HLIO computation.

```
  ⇒ $\mathit{ToSLabel}\ (\mathit{LJoin}\ \ell_1\ \ell_2)$ where
  slabel _ = case (slabel $p_1$, slabel $p_2$) of
    $(H, H) \to H$
    $(H, L) \to H$
    $(L, L) \to L$
    $(L, H) \to H$
    where $p_1 = \bot :: \mathit{LProxy}\ \ell_1; p_2 = \bot :: \mathit{LProxy}\ \ell_2$
```

Note that we have given instances for the full range of label expressions $\mathit{LExpr\ Label}$.

If we have instances for $\mathit{ToSLabel}\ \ell_1$ and $\mathit{ToSLabel}\ \ell_2$ around, then we effectively have runtime witnesses for the corresponding singleton labels, and, in that case, it is very simple to provide an instance for $\mathit{Deferrable}\ (\mathit{FlowsE}\ \ell_1\ \ell_2)$ [6]:

```
instance ($\mathit{ToSLabel}\ \ell_1$, $\mathit{ToSLabel}\ \ell_2$) ⇒
  Deferrable ($\mathit{FlowsE}\ \ell_1\ \ell_2$) where
  deferC p m = case (slabel $p_1$, slabel $p_2$) of
    $(L, L)$  → m
    $(L, H)$  → m
    $(H, H)$  → m
    $(H, L)$  → error "IFC violation!"
      where $p_1 = \bot :: \mathit{LProxy}\ \ell_1; p_2 = \bot :: \mathit{LProxy}\ \ell_2$
```

The implementation of deferC pattern matches against the runtime representations of the labels ℓ_1 and ℓ_2. In each corresponding case, the GADT pattern match (e.g. (L, L) in the first case) allows the type system to refine ℓ_1 and ℓ_2 (e.g. $\ell_1 := \mathit{LVal}\ L$ and $\ell_2 := \mathit{LVal}\ L$ in the first case). Thus, *every constraint* $\mathit{FlowsE}\ \ell_1\ \ell_2$ required by m can be refined (e.g. to $\mathit{FlowsE}\ (\mathit{LVal}\ L)\ (\mathit{LVal}\ L)$ in the first case) and can be readily discharged by top-level instances for FlowsE. It is still possible to forget to include some of the cases, but this will only make the test more conservative.

Note that in the fourth case above (for which no instance exists!), we have no way of calling m, i.e., deferC would be ill-typed if we tried. This case corresponds to a genuine runtime error, and we return an error indicating a violation of the IFC policy.

Constraints will be collected together in tuples through uses of $(\gg\!=)$ and hence we also provide an instance for pairs of constraints, whose definition we omit, i.e., $\mathit{Deferrable}\ (c_1, c_2)$.

Finally, we also revisit our definition of $\mathit{LabeledX}$ to include a dictionary for $\mathit{ToSLabel}$ to produce a singleton for the existentially-quantified label.

```
data LabeledX a where
  LabeledX :: ToSLabel ℓ ⇒ Labeled (ℓ :: Label) a
                                   → LabeledX a
```

This is necessary for applying defer to computations involving labeled expressions that have been unpacked from a $\mathit{LabeledX}$.

The $\mathit{Deferrable}$ class is an extremely powerful abstraction for transforming static errors to dynamic checks, and we later show that even type checker equalities generated by the compiler inference mechanism can be defered (Section 8). We proceed to show how $\mathit{Deferrable}$ can be used to implement the defer primitive.

6. HLIO Design and Implementation

In Haskell, we embed HLIO as a GADT where the constructors correspond to the primitives described in Section 4. More specifically, data type HLIO has constructors Return, Bind, $\mathit{Unlabel}$, Label, $\mathit{ToLabeled}$, $\mathit{GetLabel}$, Defer, and $\mathit{Simplify}$, which represent uninterpreted commands return, bind, $\mathit{unlabel}$, label, $\mathit{toLabeled}$, $\mathit{getLabel}$, defer, and $\mathit{simplify}$, respectively. The types for these constructors match the types given for the commands they represent. In order to give semantics to HLIO terms, we provide an interpretation function go with the type

```
go :: forall c ℓ_i ℓ_o a.HLIO c ℓ_i ℓ_o a
  → (c ⇒ SLabel (ℰ ℓ_i) → IO (a, SLabel (ℰ ℓ_o)))
```

The interpretation of HLIO is in an IO monad combined with a state to represent the current label (in the style of LIO). Although it might be tempting to get rid of the runtime representation of the current label, this is not possible since code is allowed to inspect it at any time (as a runtime value) using $\mathit{getLabel}$.

```
go (Return x) ℓ_i = return (x, ℓ_i)
go (Bind m f) ℓ_i = do (a, ℓ'_i) ← go m ℓ_i; go (f a) ℓ'_i
go (GetLabel ℓ_i) = return (ℓ_i, ℓ_i)
go (Unlabel (Labeled ℓ v)) ℓ_i = return (v, ℓ_i ‘ljoin‘ ℓ)
go (Label ℓ a) ℓ_i = return (Labeled ℓ a, ℓ_i)
go (ToLabeled (m :: HLIO c ℓ_i ℓ'_o a')) ℓ_i = do
  (x, ℓ_o) ← go m ℓ_i; return (Labeled ℓ_o x, ℓ_i)
go (Defer slio) ℓ_i = deferC (setProxy slio) (go slio ℓ_i)
  where setProxy :: HLIO c ℓ_i ℓ_o a → Proxy c
    setProxy = error "Proxy!"
go (Simplify m) ℓ_i = go m ℓ_i
```

The interesting cases are the definitions for $\mathit{Unlabel}$, Defer, and $\mathit{Simplify}$. For $\mathit{Unlabel}$, go performs an ordinary term-level ljoin:

```
ljoin :: SLabel ℓ_1 → SLabel ℓ_2 → SLabel (Join ℓ_1 ℓ_2)
```

but we never get to *inspect* the return label unless we explicitly perform a $\mathit{getLabel}$ and subsequently strictly use the label, or unless we perform some form of runtime check. For Defer, go applies the technique from Section 5 with the appropriate proxy. $\mathit{Simplify}$ executes m, but exposing its constraints to GHC in order to statically discharge them.

6.1 Design decisions

We briefly motivate some of the design choices made in HLIO.

(Singleton classes) We have seen in the previous section that the motivation for a type-class $\mathit{ToSLabel}\ \ell$ containing a singleton $\mathit{SLabel}\ (\mathcal{E}\ \ell)$ comes from the need for deferring FlowsE constraints.

(*$\mathit{LExpr\ Label}$ *and* $\mathit{Deferrable}$*) When describing SLIO, we used the Label datatype and the Flows type class. However, HLIO shifted to datatype $\mathit{LExpr\ \ Label}$ and the FlowE type class to be able to defer constraints. To illustrate the reason behind that, consider an alternative $\mathit{Deferrable}$ instance, without all the LExpr complications, and where $\mathit{ToSLabel}$ was indexed by Label:

```
instance (ToSLabel (ℓ_1 :: Label), ToSLabel (ℓ_2 :: Label)) ⇒
  Deferrable (Flows ℓ_1 ℓ_2) where
```

With this definition, we may find ourselves in need of deferring constraints of the form $\mathit{Flows}\ (\mathit{Join}\ \ell_1\ L)\ L$, where Join is the $\sqcup$-operation type family implementation directly on Labels. But type class axioms do not match on type families! (They only match on rigid type constructors.) Consequently, it is *impossible* to discharge that constraint either statically or dynamically. In contrast, by exposing a rigid constructor LJoin, we were able to give instances for the join of two labels; with our approach, *it is* true that the constraint $\mathit{ToSLabel}\ (\mathit{LJoin}\ \ell_1\ L)$ is automatically discharged from $\mathit{ToSLabel}\ \ell_1$.

(Embedding constraints in HLIO) The introduction of constraint c as part of the HLIO definition achieves a purely syntactic manipulation of constraints, and excludes *any possible simplification* by GHC—except when the programmer explicitly requires so with

[6] Readers can ignore the proxy arguments p, p_1 and p_2.

$simplify$. This aspect is beneficial for two reasons: Firstly, this allows us to prevent eager simplification of certain constraints into a form that cannot be deferred or even discharged. For instance, imagine that a constraint $FlowsE\ \ \ell_1\ \ \ell_2$ floats outside of the $HLIO$ type. In this case, GHC tries to discharge it by proving $Flows\ (\mathcal{E}\ \ell_1)\ (\mathcal{E}\ \ell_2)$. However, as we discussed before, type class axioms do not match on type families. Moreover, even if that were possible, deferring such a constraint would require instances of $ToSLabel\ (LVal\ (\mathcal{E}\ \ell_1))$ and $ToSLabel\ (LVal\ (\mathcal{E}\ \ell_2))$, which cannot be constructed from instances of $ToSLabel\ \ell_1$ or $ToSLabel\ \ell_2$. Secondly, when evaluating a $defer$ expression, the constraint c in $HLIO$ makes it possible for the go function to automatically supply a proxy to instantiate c (by *unification*) for a particular constraint in the type of $deferC$, thus allowing the type checker to select the right instance of $Deferrable$ without any help from the programmer. If we were not collecting the constraint c in $HLIO$, the programmer would have to supply these proxies explicitly, making HLIO much more cumbersome to use.

In summary, we have chosen to keep the constraints in their unsimplified form as much as possible, and give the programmer the freedom to decide whether they are to be checked statically or dynamically via explicit annotations ($simplify$ and $defer$).

7. Formal Semantics and Non-interference

In this section, we formalize HLIO and provide security guarantees for our approach by interpreting HLIO in LIO and showing an equivalence in the security checks performed by both systems.

Figure 7 presents a type system for HLIO. For the sake of brevity, the figure only shows the security-relevant rules; the remaining rules are standard and can be found in the extended version of this paper. The terms of HLIO are the same as in LIO, with the addition of the $defer$ construct. A lattice expression ℓ is either a primitive label $Label$, a *join* operation ($\sqcup$), or a *meet* operation ($\sqcap$). A constraint c is either the empty constraint $()$, a pair of two constraints ((c, c)), or a flow constraint among label expressions ($\ell\ \sqsubseteq\ \ell$). The type $HLIO$ is a Hoare state monad in the style of statically-typed LIO, as presented in Section 3, except that it also includes a constraint c. A computation with type $HLIO\ c\ \ell_i\ \ell_o\ \tau$ is subject to constraints c, and takes the current label from ℓ_i to ℓ_o, and produces a value of type τ. The type $Labeled\ \ \ell\ \tau$ represents expressions with label ℓ and type τ, and the type $Label\ \ell$ is a *singleton* type for label ℓ, i.e., a type with a single total inhabitant, which can be identified with ℓ.

The typing rule for $return$ simply states that the current label is not changed and no constraints need to be checked. Rule (BIND) looks like the usual typing rule for ($\gg\!=$), but it additionally combines the constraints generated by m and f (c and c') into one and also expresses that the final label of computation m should match the initial label of the computation produced by f. Rule (LABEL) generates a security check as a constraint ($\ell_i\ \ \sqsubseteq\ \ \ell$), and also expresses that the current label does not change. Note that in this rule we also check the connection between term-level s and type-level ℓ, by using the singleton type. Rule (UNLABEL) reflects the fact that unlabeling an expression labeled ℓ raises the current label ℓ_i to the join $\ell_i\ \ \sqcup\ \ \ell$. Rule (TOLABELED) checks that the subcomputation m has a valid HLIO type, and expresses that the $toLabeled$ computation will not change the current label (or rather, that it will be restored after m finishes), and also that the resulting value of type a is protected by label ℓ_o, i.e., the maximum (and final) label attained by m. Rule (DEFER) checks that the subcomputation m has a valid type and hides the constraints produced by m, so that the expression $defer\ \ m$ is subject to no static checks.

$$
\begin{array}{llcl}
\textit{Values} & v & ::= & \textit{True} \mid \textit{False} \mid () \mid \lambda x.t \mid \textit{Label} \\
& & & \mid \textit{LIO}^{\mathsf{TCB}}\ t \mid \textit{Labeled}^{\mathsf{TCB}}\ \ell\ t \\
\textit{Terms} & t & ::= & v \mid x \mid t\ t \mid \textit{fix}\ t \mid \mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t \\
& & & \mid t\ \otimes\ t \mid \textit{return}\ t \mid t \gg\!= t \mid \textit{getLabel} \\
& & & \mid \textit{label}\ t\ t \mid \textit{unlabel}\ t \mid \textit{labelOf}\ t \\
& & & \mid \textit{toLabeled}\ t\ t \mid \textit{defer}\ t \\
\textit{LOps} & \otimes & ::= & \sqcup \mid \sqcap \mid \sqsubseteq \\
\textit{Lattice} & \ell & ::= & \textit{Label} \mid \ell\ \sqcup\ \ell \mid \ell\ \sqcap\ \ell \\
\textit{Constraints} & c & ::= & () \mid (c, c) \mid \ell\ \sqsubseteq\ \ell \\
\textit{Types} & \tau & ::= & \textit{Bool} \mid () \mid \tau \to \tau \mid \textit{HLIO}\ c\ \ell_i\ \ell_o\ \tau \\
& & & \mid \textit{Labeled}\ \ell\ \tau \mid \textit{Label}\ \ell
\end{array}
$$

$$
\textsc{return}\quad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \textit{return}\ x : \textit{HLIO}\ ()\ \ell_i\ \ell_i\ \tau}
$$

$$
\textsc{bind}\quad \frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell\ a \qquad \Gamma \vdash f : a \to \textit{HLIO}\ c'\ \ell\ \ell_o\ b}{\Gamma \vdash m \gg\!= f : \textit{HLIO}\ (c, c')\ \ell_i\ \ell_o\ b}
$$

$$
\textsc{label}\quad \frac{\Gamma \vdash t : a \qquad \Gamma \vdash s : \textit{Label}\ \ell}{\Gamma \vdash \textit{label}\ s\ t : \textit{HLIO}\ (\ell_i\ \sqsubseteq\ \ell)\ \ell_i\ \ell_i\ (\textit{Labeled}\ \ell\ a)}
$$

$$
\textsc{unlabel}\quad \frac{\Gamma \vdash v : \textit{Labeled}\ \ell\ a}{\Gamma \vdash \textit{unlabel}\ v : \textit{HLIO}\ ()\ \ell_i\ (\ell_i\ \sqcup\ \ell)\ a}
$$

$$
\textsc{toLabeled}\quad \frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell_o\ a}{\Gamma \vdash \textit{toLabeled}\ m : \textit{HLIO}\ c\ \ell_i\ \ell_i\ (\textit{Labeled}\ \ell_o\ a)}
$$

$$
\textsc{defer}\quad \frac{\Gamma \vdash m : \textit{HLIO}\ c\ \ell_i\ \ell_o\ a}{\Gamma \vdash \textit{defer}\ m : \textit{HLIO}\ ()\ \ell_i\ \ell_o\ a}
$$

Figure 7. Type system for HLIO.

7.1 Semantics for LIO

Figure 8 shows the semantics of LIO, which we will use to interpret HLIO. The semantics closely follows previous work on LIO (Stefan et al. 2011b), given as a small-step operational semantics based on a transition relation $\longrightarrow$ between configurations of the form $\langle \ell_{\text{cur}}\ \mid\ t\rangle$, where ℓ_{cur} is the current label and t is the term being evaluated. As before, we only show the rules for computations with security-relevant effects. The full presentation also includes a relation for pure computation ($\rightsquigarrow$), which is used in the rule for $labelOf$, but we elide the details since they are not relevant for our purposes. The semantics uses Felleisen-style evaluation contexts to specify evaluation order, where Ep stands for contexts for pure computations and E stands for contexts for effectful ones. As usual, we define $\longrightarrow^*$ to be the reflexive and transitive closure of $\longrightarrow$. Additionally, our transitions are labeled by the information-flow constraints that are being checked at runtime, as can be seen in rule (LABEL). We write $A\ \xrightarrow{c}{}^{*}\ B$ if $A\ \longrightarrow^*\ B$ while performing the set of security checks c. For technical reasons, we also include a nonstandard primitive $eval$ which is used to force pure computations. Despite not being a part of LIO, we remark that it acts on pure values and its evaluation involves no security-relevant effects, so it is easy to prove that the calculus is still sound after adding it. Essentially, LIO already includes a way to force evaluation for booleans (if statements), so $eval$ is merely a generalization of this construct.

$$\begin{aligned} Ep &::= Ep\ t \mid \mathit{fix}\ Ep \mid \textbf{if}\ Ep\ \textbf{then}\ t\ \textbf{else}\ t \mid Ep\ \otimes\ t \mid v\ \otimes\ Ep \\ &\quad \mid \mathit{label}\ Ep\ t \mid \mathit{unlabel}\ Ep \mid \mathit{labelOf}\ Ep \mid \mathit{toLabeled}\ Ep\ t \\ E &::= [\,] \mid Ep \mid E \gg\!= t \end{aligned}$$

getLabel
$$\frac{}{\langle \ell_{\text{cur}} \mid E\ [\mathit{getLabel}]\rangle \longrightarrow \langle \ell_{\text{cur}} \mid E\ [\mathit{return}\ \ell_{\text{cur}}]\rangle}$$

toLabeled
$$\frac{\ell_{\text{cur}} \sqsubseteq \ell \qquad \langle \ell_{\text{cur}} \mid t\rangle \xrightarrow{c}{}^{*} \langle \ell'_{\text{cur}} \mid \mathit{LIO}^{\mathsf{TCB}}\ t'\rangle \qquad \ell'_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{toLabeled}\ \ell\ t]\rangle \xrightarrow{c} \langle \ell_{\text{cur}} \mid E\ [\mathit{label}\ \ell\ t']\rangle}$$

label
$$\frac{\ell_{\text{cur}} \sqsubseteq \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{label}\ \ell\ t]\rangle \xrightarrow{\ell_{\text{cur}} \sqsubseteq \ell} \langle \ell_{\text{cur}} \mid E\ [\mathit{return}\ (\mathit{Labeled}^{\mathsf{TCB}}\ \ell\ t)]\rangle}$$

unlabel
$$\frac{\ell'_{\text{cur}} = \ell_{\text{cur}} \sqcup \ell}{\langle \ell_{\text{cur}} \mid E\ [\mathit{unlabel}\ (\mathit{Labeled}^{\mathsf{TCB}}\ \ell\ t)]\rangle \longrightarrow \langle \ell'_{\text{cur}} \mid E\ [\mathit{return}\ t]\rangle}$$

Eval
$$\frac{t \rightsquigarrow^{*} v}{Ep\ [\mathit{eval}\ t] \rightsquigarrow Ep\ [v]}$$

labelOf
$$\frac{}{Ep\ [\mathit{labelOf}\ (\mathit{Labeled}^{\mathsf{TCB}}\ \ell\ t)] \rightsquigarrow Ep\ [\ell]}$$

Figure 8. Evaluation contexts and reduction rules.

$$\begin{aligned} &\mathit{interp}\ (\mathit{label}\ t\ t') = \mathit{Labeled}^{\mathsf{TCB}}\ (\mathit{eval}\ t)\ (\mathit{interp}\ t') \\ &\mathit{interp}\ (\mathit{unlabel}\ t) = \mathit{unlabel}\ (\mathit{interp}\ t) \\ &\mathit{interp}\ (\mathit{Labeled}^{\mathsf{TCB}}\ t : \mathit{Labeled}\ \ell\ \tau) = \mathit{Labeled}^{\mathsf{TCB}}\ \ell\ (\mathit{interp}\ t) \\ &\mathit{interp}\ (\mathit{defer}\ (m : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ \tau)) = \mathit{guards}\ c \gg \mathit{interp}\ m \\ &\mathit{interp}\ (\mathit{toLabeled}\ (m : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ \tau)) = \\ &\quad \mathit{toLabeled}\ \ell_o\ (\mathit{interp}\ m) \\ &\mathit{interp}\ (m \gg\!= f) = \mathit{interp}\ m \gg\!= \mathit{interp}.f \\ &\ldots \\ &\mathit{toLIO}\ (\mathit{label}\ t\ t') = \mathit{label}\ t\ (\mathit{toLIO}\ t') \\ &\mathit{toLIO}\ (\mathit{unlabel}\ t) = \mathit{unlabel}\ (\mathit{toLIO}\ t) \\ &\mathit{toLIO}\ (\mathit{Labeled}^{\mathsf{TCB}}\ t : \mathit{Labeled}\ \ell\ a) = \mathit{Labeled}^{\mathsf{TCB}}\ \ell\ (\mathit{toLIO}\ t) \\ &\mathit{toLIO}\ (\mathit{defer}\ m) = \mathit{toLIO}\ m \\ &\mathit{toLIO}\ (\mathit{toLabeled}\ (m : \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a)) = \\ &\quad \mathit{toLabeled}\ \ell_o\ (\mathit{toLIO}\ m) \\ &\mathit{toLIO}\ (m \gg\!= f) = \mathit{toLIO}\ m \gg\!= \mathit{toLIO}.f \\ &\ldots \end{aligned}$$

Figure 9. The functions interp and toLIO. The missing equations just behave homomorphically.

7.2 Semantics for HLIO

Figure 9 introduces the functions interp and toLIO, which we use to interpret HLIO and relate this interpretation with the corresponding standard LIO semantics. These two functions are defined as term-to-term transformations. The returned term, however, only utilizes LIO primitives. (Function interp closely follows the definition of function go described in Section 6.)

The function interp provides an interpretation of HLIO in dynamic LIO (Stefan et al. 2012b). Given a well-typed HLIO computation m, $\mathit{interp}\ \ m$ runs m without performing any security checks, except for those in defer. This fact can be seen in the definition for label—the case where security side-effects are triggered. This case simply synthesizes a labeled term ($\mathit{Labeled}^{\mathsf{TCB}}\ \ell\ t$), thus skipping any security check. The $\mathit{unlabel}$ operation performs no security checks, so its interpretation is exactly the same as in LIO. The interpretation of labeled terms are simply cast into dynamic labeled terms in LIO, where the dynamic label is determined by static information (i.e., $\mathit{Labeled}^{\mathsf{TCB}}\ t : \mathit{Labeled}\ \ell\ a$). In the interpretation of defer, we use the guards command, which takes a set of constraints and checks all of them at runtime, aborting the program if any of them fails. These constraints are checked in one go, before running the subcomputation itself. The static version of $\mathit{toLabeled}$ is translated into its dynamic counterpart, where the final current label (after executing m) is predicted to be ℓ_o. The interpretation of ($\gg\!=$) simply applies interp to its arguments.

Different from interp, the function toLIO directly translates an HLIO computation into a dynamic LIO computation where *all* the security checks occur dynamically. The translation for labeled terms, $\mathit{toLabeled}$, and ($\gg\!=$) are defined similarly as in interp. Label and unlabel, however, simply reformulate the command in LIO, where the corresponding security side-effects might be triggered.

7.3 Non-interference

We define the simulation relation $\sim$, which expresses that two terminating programs perform the same information flow checks and compute the same values.

Definition 1. *(Simulation between LIO terms) Let A and B be LIO configurations, then $A \sim B$ iff $A \xrightarrow{c}{}^{*} X$ and $B \xrightarrow{c}{}^{*} X$, where X is A's weak head normal form. Note that we only consider terminating programs due to the fact that LIO only provides security guarantees for terminating runs.*

We define a big-step evaluation relation $\Downarrow$ for HLIO terms.

Definition 2. *(Big-step semantics for HLIO) Given an HLIO term, $(t : \mathit{HLIO}\ \ c\ \ \ell_i\ \ \ell_o\ \ \tau) \Downarrow v$ if and only if $\langle \ell_i \mid \mathit{interp}\ \ t\rangle \xrightarrow{c'}{}^{*} \langle \ell_o \mid \mathit{toLIO}\ v\rangle$.*

The definition leverages the LIO semantics. It applies interp to the term being reduced as well as toLIO to the result. Observe that toLIO is needed for cases where v still contains HLIO terms, e.g., when v is composed of nested labeled terms.

The next lemma (see details in the extended version of the paper (Buiras et al. 2015)) introduces a relationship between the security checks done by HLIO and LIO.

Lemma 1 (Simulation between HLIO and LIO terms).
Given that $(t : \mathit{HLIO}\ \ c\ \ \ell_i\ \ \ell_o\ \ \tau) \Downarrow v$, then $\langle \ell_i \mid \mathit{guards}\ \ c \gg \mathit{interp}\ t\rangle \sim \langle \ell_i \mid \mathit{toLIO}\ t\rangle$.

The lemma states that if we take the statically-determined constraints c for a well-typed term t into account, we can prove that the programs $\mathit{guards}\ c \gg \mathit{interp}\ t$ and $\mathit{toLIO}\ t$ are in simulation with respect to their security checks and final values. The former performs all statically-determined security checks in the beginning, and then runs the program with the deferred checks. The latter is obtained by viewing the original program as an LIO program, where all defer operations are removed.

The semantic correspondence from Lemma 1 guarantees that if an HLIO program is well-typed and terminates successfully, then the equivalent LIO program would also terminate successfully. Conversely, if the LIO program fails with a security error, the HLIO program will either not have a type or fail during a defer computation. Since the HLIO and LIO enforcement mechanisms are equivalent in this sense, and LIO enforces noninterference (Stefan et al. 2011b), we can show that HLIO enforces the same property.

For our security guarantees, we consider an attacker at sensitivity level l, who can only observe values at a security level at most l. LIO defines two terms t_1 and t_2 to be l-equivalent (written

$t_1 \approx_l t_2$) if the attacker is unable to distinguish between them, e.g. $\mathit{Labeled}^{\mathsf{TCB}}\ L\ 3 \approx_l \mathit{Labeled}^{\mathsf{TCB}}\ L\ 3$ and $\mathit{Labeled}^{\mathsf{TCB}}\ H\ 1 \approx_l \mathit{Labeled}^{\mathsf{TCB}}\ H\ 5$, but $\mathit{Labeled}^{\mathsf{TCB}}\ L\ 2 \not\approx_l \mathit{Labeled}^{\mathsf{TCB}}\ L\ 1$—LIO also extends this notion to configurations. We leverage LIO definitions to express our non-interference theorem—after all, HLIO gets interpreted in LIO!

Noninterference expresses the notion that a program cannot leak secrets. Intuitively, a program is noninterfering if, considering two independent runs with l-equivalent inputs, their final values are also l-equivalent. In other words, attackers cannot distinguish the values of secret inputs by observing the outputs.

THEOREM 1 (Termination-insensitive noninterference). *Given HLIO terms t_1 and t_2 with no constructors $\cdot^{\mathsf{TCB}}$ such that constraints c_1 and c_2 hold, $(t_1 : \mathit{HLIO}\ c_1\ \ell_1\ \ell_2\ \tau) \Downarrow v_1$, $(t_2 : \mathit{HLIO}\ c_2\ \ell_i\ \ell_o\ \tau') \Downarrow v_2$, and $\langle \ell_i \mid \mathit{toLIO}\ t_1\rangle \approx_l \langle \ell_i \mid \mathit{toLIO}\ t_2\rangle$, then it holds that $\langle \ell_o \mid \mathit{toLIO}\ v_1\rangle \approx_l \langle \ell_o \mid \mathit{toLIO}\ v_2\rangle$.*

PROOF SKETCH 1. *The proof uses Lemma 1 to relate the reductions of $\mathit{interp}\ t_1$ and $\mathit{interp}\ t_2$ with $\mathit{toLIO}\ t_1$ and $\mathit{toLIO}\ t_2$, respectively. Once that is done, the result follows by applying the LIO non-interference theorem in (Stefan et al. 2012b). This theorem requires that t_1 and t_2 do not include constructors of the form $\cdot^{\mathsf{TCB}}$. Consequently, observe that it is not possible to directly consider l-equivalence between interpreted terms, i.e., $\mathit{interp}\ t_1$ and $\mathit{interp}\ t_2$—they introduce constructors $\mathit{Labeled}^{\mathsf{TCB}}$ to avoid security checks. The proof is given in the extended version of the paper (Buiras et al. 2015).*

The theorem indicates that l-equivalent (fully) dynamic interpretations of HLIO terms (i.e., $\langle \ell_i \mid \mathit{toLIO}\ t_1\rangle \approx_l \langle \ell_i \mid \mathit{toLIO}\ t_2\rangle$), where the static checks hold, produce l-equivalent results (in LIO) (i.e., $\langle \ell_o \mid \mathit{toLIO}\ v_1\rangle \approx_l \langle \ell_o \mid \mathit{toLIO}\ v_2\rangle$). Observe that if any HLIO terms leaked secrets, l-equivalence involving v_1 and v_2 would not hold.

8. Discussion

This section explains some design choices, while exploring others.

The $\mathit{toLabeled}$ function The HLIO type for $\mathit{toLabeled}$ deserves some attention. From Section 2, we know that $\mathit{toLabeled}\ \ell\ m$ in LIO performs two security checks: $\ell_{\text{cur}} \sqsubseteq \ell$ at the begining of $\mathit{toLabeled}$, and $\ell'_{\text{cur}} \sqsubseteq \ell$ where ℓ'_{cur} is the current label obtained by evaluating m. A directly corresponding static version of $\mathit{toLabeled}$ (and its dynamic checks) might be:

$$\begin{aligned}&\mathit{toLabeled} :: \mathit{Label}\ \ell \to \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a\\ &\quad \to \mathit{HLIO}\ (c, \mathit{FlowsE}\ \ell_i\ \ell, \mathit{FlowsE}\ \ell_o\ \ell)\ \ell_i\ \ell_i\ (\mathit{Labeled}\ \ell\ a)\end{aligned}$$

Recall that, for security reasons, the role of the first argument (of type $\mathit{Label}\ \ell$) is to statically predict an upper bound of the current label obtained by running m. The constraints in the return type of $\mathit{toLabeled}$ express this fact. In HLIO, however, that prediction is already given! Observe that type $m :: \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a$ says "after running m, the final current label is ℓ_o." We can use ℓ_o as the upper bound, i.e., $\ell \equiv \ell_o$, and remove the static check $\mathit{FlowsE}\ \ell_o\ \ell$. Moreover, we know that $\ell_i \sqsubseteq \ell_o$ by construction, which allows the removal of $\mathit{FlowE}\ \ell_i\ \ell$. By taking all these facts together, we can dismiss all the extra constraints.

$$\mathit{toLabeled} :: \mathit{HLIO}\ c\ \ell_i\ \ell_o\ a \to \mathit{HLIO}\ c\ \ell_i\ \ell_i\ (\mathit{Labeled}\ \ell_o)\ a$$

In Section 7, we have formally proved that this primitive is secure by establishing a simple relationship with its counterpart in LIO.

Conditionals The monad HLIO is embedded in Haskell as a GADT, so it is possible to use Haskell's **if** statements to express conditional branching. However, the Haskell type system requires that the types of both branches be the same. In particular, if the branches are HLIO computations, their types must also completely agree, *including* constraints and initial and final labels. Unfortunately, this means that it is not possible to have **if** statements where one branch produces a constraint and the other one does not or, more generally, where the branches produce different sets of constraints. For example, the following expression, where $x :: \mathit{Int}$, is ill-typed:

$$\textbf{if}\ x > 0\ \textbf{then}\ (\mathit{label}\ H\ x \gg \mathit{return}\ x)\ \textbf{else}\ \mathit{return}\ (x+1)$$

The reason for the type error is that one branch has type $\mathit{HLIO}\ (\mathit{Flows}\ \ell_i\ H)\ \ell_i\ \ell_i\ \mathit{Int}$, while the other one has type $\mathit{HLIO}\ ()\ \ell_i\ \ell_i\ \mathit{Int}$. When it comes to disparities in the constraints, it is possible to work around this restriction by means of defer operations. The programmer can use defer to check one or both of the branches dynamically, which causes the constraints in the HLIO type to be $()$, thus keeping the Haskell type checker happy. However, if the current label is not updated in exactly the same way in both branches, the **if** statement will also be ill-typed. Note that this cannot be solved with defer.

An alternative solution that addresses the problem with both constraints and the current label involves adding another primitive for **if** statements, i.e., a constructor If for the HLIO GADT. The type of this constructor would accurately express the connection between constraints and current labels in both branches, as follows:

$$\begin{aligned}&\mathit{If} :: \mathit{Bool} \to \mathit{HLIO}\ c_1\ \ell_i\ \ell_o\ a \to \mathit{HLIO}\ c_2\ \ell_i\ \ell'_o\ a\\ &\quad \to \mathit{HLIO}\ (c_1, c_2)\ \ell_i\ (\mathit{LJoin}\ \ell_o\ \ell'_o)\ a\end{aligned}$$

Essentially, the primitive would over-approximate the constraints and the final label, as can be expected from a static analysis. This solution would not only introduce notational overhead but also complicate the formal treatment of HLIO significantly, as we would no longer have a one-to-one correspondence between static and dynamic checks. Instead, we could prove that the dynamic checks are a *subset* of the statically-determined constraints. In order to simplify our exposition, we chose to avoid this solution, but we believe it would be a reasonably straightforward extension.

Deferring constraints beyond non-interference The $\mathit{Deferrable}$ type class enables programmers to give instances for deferring the check for a constraint to runtime. In this section, we show how to push this idea to the extreme, by deferring the check for type equalities that are generated by GHC's type inference. We iterate that the code in this section (and everywhere in this paper) requires no modifications to GHC.

We wish to defer a type equality between two types ta and tb, which in GHC type system would be expressed as $\mathit{ta}{\sim}\mathit{tb}$ of kind $\mathit{Constraint}$. Of course, in order to perform such a test at runtime, we need to have *runtime type information* around about the shape of types ta and tb. GHC provides the $\mathit{Typeable}$ type class that captures runtime type representations. This enables the following instance definition:

```
instance (Typeable a, Typeable b) ⇒
  Deferrable (a∼b) where
  defer _p m = case eqT :: Maybe (a :∼: b) of
    Nothing → error "type error!"
    Just Refl → m
```

Function eqT is a standard library function, providing a runtime witness of the equality of two types that are instances of $\mathit{Typeable}$:

$$\mathit{eqT} :: (\mathit{Typeable}\ b, \mathit{Typeable}\ a) \Rightarrow \mathit{Maybe}\ (a :\sim: b)$$

and $a :\sim: b$ is a GADT expressing with its only constructor Refl the fact that a and b are in fact equal:

$$\textbf{data}\ (a :\sim: b)\ \textbf{where}\ \mathit{Refl} :: (a{\sim}b) \Rightarrow (a :\sim: b)$$

If programmers write a program that contains a type error:

$foo :: forall\ a.a \rightarrow a \rightarrow a$
$foo\ x\ y = \mathbf{if}\ x\ \mathbf{then}\ False\ \mathbf{else}\ y$

GHC will report: `Couldn't match expected type 'Bool' with actual type 'a'`. As we may, in fact, apply *foo* to two boolean values at runtime, programmers may want to make this program typeable by deferring the constraint:

$foo :: forall\ a. Typeable\ a \Rightarrow a \rightarrow a \rightarrow a$
$foo\ x\ y = defer\ p\ (\mathbf{if}\ x\ \mathbf{then}\ False\ \mathbf{else}\ y)$
$\quad \mathbf{where}\ p :: Proxy\ (a{\sim}Bool) = \bot$

In this case, $foo\ True\ False$ returns $False$, while $foo\ 3\ 4$ produces `*** Exception: type error`. Note that this behaviour differs from related work (Vytiniotis et al. 2012), which defers unsatisfiable constraints as errors to runtime. Instead, we do genuinely defer the check at the (unavoidable) cost of having the type representation around.

9. Related work

Hybrid IFC There is considerable literature on static analyses aiding IFC execution monitors for different purposes. To boost permissiveness, Le Guernic et al. provide monitors which statically analyze non-taken branches of secret conditionals (Le Guernic et al. 2007; Le Guernic 2007). Similarly, Shroff et al. design a monitor which leverages variable dependencies (provided by a type system) when programs branch on secrets (Shroff et al. 2007). Besides permissiveness, hybrid analyses are used to avoid leaks in dynamic flow-sensitive IFC monitors, where variables change their security levels at runtime based on what data they store (Russo and Sabelfeld 2010). Moore and Chong utilizes static analysis to avoid tracking variables which do not impose security violations, thus improving performance on dynamic monitors (Moore and Chong 2011). Jif, an IFC-aware compiler for Java programs, supports dynamic labels to classify data based on runtime observations (Zheng and Myers 2007). Similar to our work, operations on labels are modeled at the level of types. In the dynamic part, however, they only allow for runtime checks based on the $\sqsubseteq$ relationship. As in this work, there is some literature which connects dynamic and static analysis at the programming-language level. Disney and Flanagan describe an IFC type-system for a pure λ-calculus which defers cast checks to runtime when they cannot be determined statically (Disney and Flanagan 2011). Luminous and Thiemann extend that work to consider references (Fennell and Thiemann 2013).

Security libraries Li and Zdancewic's seminal work (Li and Zdancewic 2006) shows how arrows (Hughes 2000) can provide IFC without runtime checks as a library in Haskell. Tsai et al. (Tsai et al. 2007) extend Li and Zdancewic's work to support concurrency and data with multiple security labels. Rather than using arrows, Russo et al. (Russo et al. 2008) shows that monads are capable of providing a library which statically enforces IFC. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC. Their technique is applied to dynamic, static, and hybrid IFC techniques. Devriese and Piessens' work requires a deep embedding of the target language in order to perform static analysis. In contrast, our approach leverages the type-system features found in Haskell. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) (Jaskelioff and Russo 2011)—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. The series of work on LIO can be referred to as the state-of-the-art in dynamic IFC in Haskell (Stefan et al. 2011b, 2012b,a; Buiras et al. 2013; Buiras and Russo 2013; Buiras et al. 2014).

Programming languages Combining dynamic and static analysis is not exclusive to IFC research. It has been extensively studied by the programming languages community. We briefly mention some highlights and their relation to this work. Flanagan (Flanagan 2006) develops the concept of *hybrid type checking* for type systems capable of delaying subtyping checks until runtime. Siek and Taha (Siek and Taha 2006) coined the term *gradual typing*, which applies when programmers can control the combination of static and dynamic approaches at the programming language level—simultaneously, Hochstadt and Felleisen (Hochstadt and Felleisen 2006) introduce similar ideas. Due to the *defer* primitive, HLIO can be considered as a simple gradual typing system. Wadler and Findler (Wadler and Findler 2009) presents the idea of *blame* to explain failure of dynamic type casts (specially for languages with higher-order functions). HLIO is a system which only produces positive blame. Recently, the idea of gradual typing has gained popularity among several programming languages. Typed Scheme (Hochstadt and Felleisen 2006) and Racket (Takikawa et al. 2012) allow Scheme programmers to decorate their code with type annotations. Reticulated Python (Vitousek et al. 2014) implements gradual typing, where a type checker is provided in combination with a code-to-code transformation into Python 3. JavaScript has been also a recent target of this kind of systems (Swamy et al. 2014; Rastogi et al. 2015). Different from these approaches, HLIO does not provide a fully-fledged gradual typing system. On the other hand, it avoids any compiler modification by leveraging Haskell's powerful type system.

10. Conclusions and Future Work

We have presented HLIO, a new hybrid IFC enforcement in Haskell that allows programmers to defer static constraints to runtime. This feature is particularly useful, for instance, in production systems—where it is often the case that security labels are not available (or even known) at compile time. Different from other programming languages, GHC's powerful type-system and features allowed us to build HLIO as a simple library, where no runtime or compiler modifications were needed. On formal aspects, we showed that the library satisfies termination-insensitive non-interference for an arbitrary security lattice.

As part of developing HLIO, we have identified an independently useful technique for deferring other forms of static constraints, including ordinary type equalities. In future work, we aim to explore the use of these techniques in languages with similarly expressive type systems, such as dependently typed languages. In addition, we plan to further explore the design and application space of these techniques, and explore their usability in embedded domain-specific languages and code generators.

Acknowledgments This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation. We thank the anonymous reviewers and Bart van Delft for useful comments and suggestions.

References

A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2009.

T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.

P. Buiras and A. Russo. Lazy programs leak secrets. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Springer Verlag, 2013.

P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013*, 2013.

P. Buiras, D. Stefan, and A. Russo. On flow-sensitive floating-label systems. In *Proc. of 27th IEEE Computer Security Foundations Symp.*, July 2014.

P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell (Extended version), 2015. URL `http://www.cse.chalmers.se/~buiras/hlio/`.

T. Disney and C. Flanagan. Gradual information flow typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. . URL `http://doi.acm.org/10.1145/2364506.2364522`.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

L. Fennell and P. Thiemann. Gradual security typing with references. In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*, CSF '13. IEEE Computer Society, 2013.

C. Flanagan. Hybrid type checking. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06. ACM, 2006.

D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

S. T. Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI*, 2011.

G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*. IEEE Computer Society, 2007.

G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues*, ASIAN'06. Springer-Verlag, 2007.

P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW'06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

E. Meijer and P. Drayton. Static Typing Where Possible, Dynamic Typing When Needed. *Revival of Dynamic Languages*, 2005. URL `\url{http://research.microsoft.com/\~emeijer/Papers/RDL04Meijer.pdf}`.

S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proc. of the 24th IEEE Computer Security Foundations Symposium*. IEEE Press, June 2011.

A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. on Computer Systems*, 9(4):410–442, October 2000.

A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. . URL `http://doi.acm.org/10.1145/1159803.1159812`.

D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *Lecture Notes in Computer Science,*, pages 56–71. Springer, 2010. . URL `https://lirias.kuleuven.be/handle/123456789/259608`.

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe and efficient gradual typing for typescript. In *In Proc. of the ACM Conference on Principles of Programming Languages (POPL) 2015*, Jan. 2015.

A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. of the 2010 23rd IEEE Computer Security Foundations Symp.*, CSF '10, pages 186–199. IEEE Computer Society, 2010.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proc. of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, 2008.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, Lecture Notes in Computer Science (LNCS). Springer Verlag, June 2009.

P. Shroff, S. Smith, and M. Thober. Dynamic Dependency Monitoring to Secure Information Flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proc. of Scheme and functional programming workshop*. Technical Report. University of Chicago, 2006.

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

D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec 2011*, LNCS. Springer, October 2011a.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011b.

D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of 17th ACM SIGPLAN International Conference on Functional Programming*, Sep. 2012a.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012b.

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual Typing Embedded Securely in JavaScript. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. ACM, 2012.

D. Terei, S. Marlow, S. P. Jones, , and D. Mazières. Safe Haskell. In *Proceedings of the 5th Symposium on Haskell*, September 2012.

T. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Computer Security Foundations Symp., 2007. CSF '07. 20th IEEE*, pages 187–202, July 2007.

S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. on Computer Systems*, 25(4):11:1–43, December 2007. A version appeared in *Proc. of the 20th ACM Symp. on Operating System Principles*, 2005.

M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. of the 10th ACM Symposium on Dynamic Languages*, DLS '14. ACM, 2014.

D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. . URL `http://doi.acm.org/10.1145/2364527.2364554`.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09. Springer-Verlag, 2009.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. . URL `http://doi.acm.org/10.1145/2103786.2103795`.

N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, November 2006.

L. Zheng and A. C. Myers. Dynamic security labels and static information flow. *International Journal of Information Security*, 6(2–3), 2007.