Typed Faceted Values for Secure Information Flow in Haskell

Thomas H. Austin
San José State University
thomas.austin@sjsu.edu

Kenneth Knowles
University of California, Santa Cruz
kknowles@cs.ucsc.edu

Cormac Flanagan
University of California, Santa Cruz
cormac@cs.ucsc.edu

Abstract

When an application fails to ensure information flow security, it may leak sensitive data such as passwords, credit card numbers, or medical records. News stories of such failures abound. Austin and Flanagan [2012] introduce faceted values – values that present different behavior according to the privileges of the observer – as a dynamic approach to enforcing information flow policies for an untyped, imperative λ-calculus.

We implement faceted values as a Haskell library, elucidating their relationship to types and monadic imperative programming. In contrast to previous work, our approach does not require modification to the language runtime. In addition to pure faceted values, our library supports faceted mutable reference cells, secure facet-aware socket-like communication, and robust declassification.

1. Introduction

When a program deals with sensitive data, one may take precautions against that data being misused, intentionally or accidentally. *Secure information flow* refers generally to properties that embody this idea. For example, when one enters a password on a web form it is expected that it will be communicated to the site in question, but not written to disk.

Unfortunately, enforcing these policies is problematic. Developers are primarily concerned with features and correct functionality; enforcement of security properties is generally only given proper attention after an exploit has occurred.

Just as memory-safe languages relieve developers from having to reason about memory management (and the host of bugs resulting from its *mis*management), information flow analysis is a promising mechanism to enforce security properties in a systemic fashion. Information flow controls require a developer to mark sensitive information, but otherwise automatically protect any "leaks" of this data. Formally, we refer to this property as *noninterference*; that is, public outputs do not depend on private inputs[1].

[1] We refer to sensitive values as "private" and non-sensitive values as "public", as confidentiality is generally given more attention in the literature on information flow analysis. However, the same mechanism is also able to enforce integrity properties, such as that trusted outputs are not influenced by untrusted inputs.

Secure multi-execution [Devriese and Piessens 2010; Jaskelioff and Russo 2012; Rafnsson and Sabelfeld 2013] has risen in popularity as an information flow enforcement technique. A program execution is split into two versions: the "high" execution has access to all sensitive information, but may only write to private channels; the "low" execution may write to public channels, but does not have access to any sensitive information. With this elegant approach, noninterference is ensured.

Austin and Flanagan [2012] presented *faceted values* as a technique for simulating secure multi-execution with a single process. A faceted value contains both a public and private facet. During program execution, these values keep track of both the public and private views of sensitive information. With this approach, a single execution can provide many of the same guarantees that secure multi-execution provides.

This paper extends the ideas of faceted values from an untyped variant of the lambda calculus to Haskell, showing how these concepts may be adapted to a typed language. The contributions of this paper are:

- We present the first formulation of faceted values and computations in a typed context. As a consequence of this formulation, we clearly demonstrate the division between the public interface and private implementation of faceted values.
- We show how faceted values may be integrated into a language as a library, rather than requiring modifications to the language runtime environment.
- We also present a novel clarification of the relationship between explicit flows in pure calculations and implicit flows in impure computations. Specifically, we show that implicit flows between faceted *computations* manifest as explicit flows in the faceted *expressions* that determine which computation to run. These two interact via a single function that is a distributive law between the pure and impure faceted monads.

1.1 Information Flow and Faceted Values Overview

In traditional information flow systems, information might be tagged with a label to mark it as confidential to particular parties. For instance, if `pin` should be restricted to *bank*, our language might allow us to write that as:

$$\texttt{pin} = 4321^{bank}$$

In order to protect this value, we must prevent an unauthorized viewer from seeing any effect from this value. In particular, we must defend against *explicit flows* where a confidential value is directly assigned to a public variable or reference cell, and *implicit flows* where a confidential value may be deduced by reasoning about the control flow of the program. The following code shows an explicit flow from `pin` to the variable `x`.

```
pin = 4321^bank
x = pin + 1
```

Protecting against explicit flows is straightforward, and is handled by the taint mode of programming languages like Perl and Ruby; in contrast, handling implicit flows is much more complex and subtle. Continuing our example, consider the following block of code, using a mutable `IORef`:

```
do above2K <- newIORef False
   if (pin > 2000) then
     writeIORef above2K True
   else
     return ()
```

This code illustrates a simple implicit flow; the value of `above2K` reflects information about `pin`, even though the value of `pin` is never directly assigned to `above2K`. Several strategies have been proposed for handling these types of flows:

1. Disallow the update to `above2K` within the context of the sensitive conditional `pin`. When this restriction is enforced at runtime, this approach is referred to as the *no sensitive-upgrade strategy* [Zdancewic 2002; Austin and Flanagan 2009]. It is also the approach used in many information flow type systems [Volpano et al. 1996; Heintze and Riecke 1998]. This strategy is demonstrated in the *No-Sensitive-Upgrade* column of Figure 1.
2. Allow the update, but mark `above2K` as sensitive since it was changed in a sensitive context. This strategy is often used in dynamic enforcement techniques; it has proven as a useful mechanism for auditing information flows "in the wild" [Jang et al. 2010], but does not guarantee noninterference, as shown in the *Naive* column of Figure 1.
3. Ignore the update to `above2K` in a sensitive context, an approach first used by Fenton [1974]. This strategy guarantees noninterference by sacrificing correctness (the program's result may not be internally consistent). This strategy is demonstrated in the *Fenton* column of Figure 1.

Faceted values introduce a third aspect to sensitive data; in addition to the sensitive value and its label, faceted values include a default public view. Following this syntax used by Austin and Flanagan [2012], we specify '0000' as the default view of the PIN as follows:

`pin` = $\langle bank \; ? \; 4321 : 0000 \rangle$

Then, when `above2K` is updated as above, the result will be $\langle bank \; ? \; \texttt{True} : \texttt{False} \rangle$. The bank sees the correct value of `True`, but an unauthorized viewer instead sees `False`, giving a consistent picture to the unauthorized viewer.

Label-based information flow systems can reason about multiple principles by joining labels together (e.g. $3^A + 4^B = 7^{AB}$). In a similar manner, faceted evaluation can nest faceted values to represent different principals

$$\langle A \; ? \; 3 : 0 \rangle + \langle B \; ? \; 4 : 0 \rangle = \langle A \; ? \; \langle B \; ? \; 7 : 3 \rangle : \langle B \; ? \; 4 : 0 \rangle \rangle$$

essentially constructing a tree[2] matching permissions to values.

Figure 1, adapted from Austin and Flanagan [2012], demonstrates a classic code snippet first introduced by Fenton [1974]; it illustrates how the use of two conditional statements may evade some information flow controls.

The input parameter `x` is a confidential boolean value, represented as $\langle k \; ? \; \texttt{False} : \bot \rangle$ for false and $\langle k \; ? \; \texttt{True} : \bot \rangle$ for true, where $\bot$ means roughly 'undefined'. Boolean reference cells `y` and `z` are initialized to true; by default, they are public to maximize the permissiveness of these values. When `x` is set to $\langle k \; ? \; \texttt{False} : \bot \rangle$, the value for `y` remains unchanged. Similarly, since the update to `z` depends only on `y` it remains public as well. Since no private information is involved in the update to `z`, all strategies work in the same manner, shown in the *All Strategies* column of Figure 1.

The difference between these strategies lies in how they handle the update to `y` in the first conditional statement when the value of `x` is $\langle k \; ? \; \texttt{True} : \bot \rangle$. Since this update depends upon the value of `x`, we must be careful to avoid the potential implicit flow from `x` to `y`. We now compare how each approach handles this update.

In the *Naive* column of Figure 1, the influence of `x` is tracked by labeling it with k. Since `y` is false, `z` remains unchanged, and also remains public. Thus, as can be seen by comparing the *Naive* and the *All strategies* columns, one bit of information leaks, violating noninterference.

The *No-Sensitive-Upgrade* approach instead terminates execution on this update, guaranteeing termination-insensitive noninterference, but at the cost of potentially rejecting valid programs. Similarly, the *Fenton* strategy forbids this update, but allows execution to continue. This approach avoids abnormal termination, but it may return inaccurate results, as shown in Figure 1.

Faceted evaluation solves this dilemma by simulating different executions of this program. In the *Faceted Evaluation* column, we see that the update to `y` results in the creation of a new faceted value $\langle k \; ? \; \texttt{False} : \texttt{True} \rangle$. Any authorized viewer able to see k-sensitive data[3] is able to see the true value of `y`; unauthorized viewers instead see `True`, thus hiding the value of `x`. The value of `z` is updated in a similar manner in the second conditional assignment. The result of this function call therefore provides noninterference, avoids terminating execution abnormally, and provides accurate results to authorized users.

1.2 Faceted I/O

Austin and Flanagan [2012] treat both facet-aware reference cells and facet-aware file I/O, though with strikingly different mechanisms. While reference cells may contain faceted values, the assumption is that external systems might not share the same capability. Therefore, we must be able to send non-faceted, or "raw" values across channels. Consider the following code that writes a faceted character $\langle \, k \; ? \; \texttt{'a'} : \texttt{'b'} \, \rangle$ to a handle h:

`hPutChar` $h \, \langle \, k \; ? \; \texttt{'a'} : \texttt{'b'} \, \rangle$

When executing the above code, we must determine whether to send `'a'`, `'b'`, or no value at all. The solution offered by Austin and Flanagan [2012] associates a fixed set of security labels to a given channel. If k-sensitive data is permitted on the channel h, then the value `'a'` is written; otherwise `'b'` is sent instead.

Additionally, some writes are restricted altogether if they are performed in a context that should not be visible. For example, consider the following code where a write is performed in the context of a sensitive boolean value:

```
if ⟨ bank ? True : False ⟩ then
  hPutChar publicChannel 'z'
else
  return ()
```

Even though the value `'z'` is public – all constants are public – the write to `publicChannel` must not be allowed unless it is authorized to see private banking information.

[2] Alternately, a faceted value can be interpreted as a function mapping sets of labels to values, and the syntax above as merely a compact representation.

[3] That is, authorized to see data marked as sensitive to principal k.

x =	$\langle k$? False : $\bot\rangle$	$\langle k$? True : $\bot\rangle$			
do...	*All strategies*	*Naive*	*No-Sensitive-Upgrade*	*Fenton*	*Faceted Evaluation*
y <- newIORef True	y = True	y = True	y = True	y = True	y = True
z <- newIORef True	z = True	z = True	z = True	z = True	z = True
vx <- readIORef x	–	–	–	–	–
when vx	–	$pc = \{k\}$	$pc = \{k\}$	$pc = \{k\}$	$pc = \{k\}$
(writeIORef y False)	–	y = $\langle k$? False : $\bot\rangle$	*stuck*	*ignored*	y = $\langle k$? False : True$\rangle$
vy <- readIORef y	–	–		–	–
when vy	$pc = \{\}$	–		–	$pc = \{\overline{k}\}$
(writeIORef z False)	z = False	–		–	z = $\langle k$? True : False$\rangle$
readIORef z	–	–		–	–
Result:	False	True	*stuck*	False	$\langle k$? True : False$\rangle$

Figure 1. A Computation with Implicit Flows

When we add *interactive* I/O, more subtleties arise. Consider the following code that writes to handle h and then reads the result from the same channel:

```
hPutChar h ⟨ k ? 'a' : 'b' ⟩
hGetChar h
```

If the channel h may view k-sensitive data, the attacker might attempt to use the above code to break noninterference, effectively using I/O to serve as a form of declassification. To prevent this attack, reading from the channel incorporates the same sensitive influences; the result is $\langle k$? 'a' : $\bot\rangle$ in this case, hiding the k-sensitive value from unauthorized viewers.

When the channel h may *not* view k-sensitive data, the proper result is less clear. The original paper addresses this issue by incorporating the negative influences of the view; that is, the result would return $\langle k$? $\bot$: 'b'$\rangle$, where the confidential value has disappeared from the system. While this approach guarantees noninterference, it also requires the system to keep track of all possible labels, thereby leading to practical challenges for a system with an arbitrary number of labels.

In this paper, we refine the earlier approach by simply ignoring the negative influences when reading a file. In other words, if channel h cannot observe k-sensitive data, reads will return an unfaceted value of 'b'. In our experience, this appears to be a more natural approach when writing code and avoids many challenges of implementing a system supporting faceted values.

2. Library Overview

We implement faceted computation in Haskell as a library that enforces information flow security dynamically, using abstract data types to prevent circumvention of dynamic protections. In contrast, the original formulation of Austin and Flanagan [2012] added faceted values pervasively to the semantics of a dynamically-typed, imperative λ-calculus. Because of the encapsulation offered by Haskell's type system, we do not need to modify the language semantics. We are also able to implement pure faceted values entirely separately from side-effecting computations upon faceted reference cells and facet-enabled input/output channels. A preliminary version of our library is available via Hackage (`https://hackage.haskell.org/`) and our open-source code can be found at `https://github.com/haskell-faceted/haskell-faceted`.

Our library can be conceptually divided into the following components:

1. Pure faceted values of type a, represented by the type `Faceted` a. (Section 2.1)
2. Imperative faceted computations influenced by the control flow path, represented by the type `FIO` a. (Section 2.2)

```
type Label = String

data Faceted a

makePublic :: a -> Faceted a
makeFaceted :: Label -> Faceted a -> Faceted a
                     -> Faceted a

instance Functor Faceted
instance Applicative Faceted
instance Monad Faceted
```

Figure 2. Interface for pure fragment of the `Faceted` library.

3. Faceted reference cells, represented by type `FIORef` a. (Section 2.3)
4. Facet-enabled file handles / network sockets, represented by type `FHandle`. (Section 2.4)
5. Robust declassification, which selectively releases private data if the control flow has not been tainted. (Section 2.5)

2.1 Pure Faceted Values: `Faceted` a

Interface The public interface for the pure fragment of our library for faceted values is shown in Figure 2.

Information flow security is defined with respect to security labels. For simplicity and concision of presentation, we presume that security labels are strings, and set `type Label = String`, though leaving the type of labels abstract is straightforward. Our approach is compatible with other label systems, such as the decentralized label model of Myers [1999]. A security label may represent the privilege necessary to view a value or, dually, the level of integrity of a value. A *view* is a set of security labels. Intuitively, if a view L contains a label k, then that view has the privilege of viewing data protected by k or, dually, trusts data influenced by k.

A value of type `Faceted` a simultaneously represents the values, or *facets*, of type a observable to any view. The facets are not directly observable; the data type is abstract.

The function `makePublic` injects any type a into the type `Faceted` a: It accepts a value v of type a and returns a faceted value that behaves just like v for any view.

The function `makeFaceted` constructs a value of type `Faceted` a from a label k and two other faceted values $priv$ and pub, each of type `Faceted` a. To any view including k, the result behaves as $priv$. To all other views, the result behaves as pub (and so on, recursively).

From `makeFaceted` we can define a variety of derived constructors for creating faceted values with a minimum of effort. Con-

sider the following two examples, which we provide with the library, but which do not increase expressivity.

```
bottom :: Faceted a
bottom = makePublic undefined

makePrivate :: Label -> a -> Faceted a
makePrivate k v = makeFaceted k (makePublic v) bottom

makeFacets :: Label -> a -> a -> Faceted a
makeFacets k priv pub = makeFaceted k (makePublic priv)
                                      (makePublic pub)
```

The value `bottom` injection `undefined` (which we occasionally abbreviate $\bot$) into `Faceted`a for any a. The function `makePrivate` creates a faceted value from a label k and a value v of type a. This value behaves as v for any view containing k, and is otherwise undefined. The function `makeFacets` creates a faceted value from a label k and two *non*-faceted values, simply allowing the programmer to elide tedious calls to `makePublic`.

The remainder of the public interface for the pure fragment of our library consists only of a type class instances for `Functor`, `Applicative`, and `Monad`. The first two are necessarily determined by the instance for `Monad`, which is formulated to propagate security labels appropriately. Consider the following example, using Haskell's specialized `do` syntax to multiply two values of type `Faceted Int`.[4]

```
do x <- (makePrivate "k" 7)
   y <- (makePrivate "l" 6)
   return (x * y)
```

The result will behave as 42 to any view containing both `"k"` and `"l"`. To all other views, the result will appear undefined.

Implementation The private implementation of faceted values is a simple algebraic data type:

```
data Faceted a =
    Raw a
  | Faceted Label (Faceted a) (Faceted a)
```

The term `Raw` x denotes a faceted value with no visibility constraints. The term `Faceted` k $priv$ pub (written $\langle k \,?\, priv \,:\, pub \rangle$ by Austin and Flanagan [2012] and in our introduction; we now abandon that notation in favor of concrete Haskell syntax.) denotes a value that behaves as $priv$ to any view containing k and behaves as pub to any other view (and so on, recursively).

In fact, this data type may be recognized as the free monad[5] over the following non-recursive data type `Facets`:

```
data Facets a = Facets Label a a

data Free f a = Pure a | Free (f (Free f a))
type Faceted = Free Facets
```

This fact provides ready-made instances of `Monad`, `Applicative`, and `Functor` which are guaranteed to obey the appropriate equational laws. However, computing with Haskell implementation of the free monad introduces an extra layer of data constructors pervasively. So for concision and clarity, we continue our presentation using the recursive data type `Faceted`.

[4] This computation may also be written as `(*) <*> (makePrivate "k" 7) <$> (makePrivate "l" 6)`. In general, computations on faceted values may be written tersely using `Monad` and `Applicative` combinators such as `liftM2`, `liftA2`, `<*>`, `<$>`, *etc*, according to taste. Here we choose to use Haskell's `do`-notation as it is readable and familiar to even novice Haskell programmers.

[5] A Haskell implementation of free monads over functors is available on Hackage in the library `free` [Kmett 2014]

The implementations of `makePublic` and `makeFaceted` are simple wrappers on the appropriate constructors:

```
makePublic    = Raw
makeFaceted   = Faceted
```

The instances of `Functor`, `Monad`, and `Applicative` are straightforward – they are all consequences of `Faceted` being a free monad. Here are their implementations in full:

```
instance Functor Faceted where
  fmap f (Raw x)          = Raw (f x)
  fmap f (Faceted k x y) = Faceted k (f x) (f y)

instance Applicative Faceted where
  pure x                 = Raw x
  (Raw f) <*> x          = fmap f x
  (Faceted k f g) <*> x = Faceted k (f <*> x) (g <*> x)

instance Monad Faceted where
  return x               = Raw x
  (Raw x) >>= f           = f x
  (Faceted k x y) >>= f  = Faceted k (x >>= f) (y >>= f)
```

2.2 Side-effecting faceted computations: `FIO` a

Information flow security is nearly trivial for the pure language of Section 2.1 because all dependencies between values are explicit. There are no *implicit flows*. An implicit flow occurs when a value is computed based on side effects that may differ depending on the value of private data.

In the following classic example, assume that `secret` has type `Int` and the variable `x` is a reference cell of type `IORef Int` that initially contains the value `0`.

```
do if secret == 42
     then writeIORef x 1
     else return ()
   readIORef x
```

The return value will be 1 if and only if `secret == 42`. Note, however, that the value stored at `x` does not explicitly depend on `secret`.

Suppose we opt to protect the confidentiality of `secret` by setting `secret` = `makePrivate` k 42. The type of `secret` is now `Faceted Int`. Then our example can be reformulated as

```
do v <- secret
   do if v == 42
        then writeIORef x 1
        else return ()
      readIORef x
```

The outer `do` begins a computation in the `Faceted` monad, while the inner `do` begins a computation in the `IO` monad. This expression has type `Faceted (IO Int)`, so it cannot be "run" as part of a Haskell program. Using only the pure fragment of our library, implicit flows are prevented. Unfortunately, *all* implicit flows are prevented, even those that do not violate information flow security.

Guided by the types, we need a way to convert a value of type `Faceted (IO` a`)` to a value of type `IO (Faceted` a`)`. This latter should then be run to yield a value of type `Faceted` a, where the implicit flow is taken into account in the facets. In terms of our example, for any view for which `secret` is visible, the value returned should behave as `1`, while for all other views it should behave as `0`. The remainder of this section focuses on the general design of faceted `IO` computations, while Section 2.3, Section 2.4, and Section 2.5 illustrate implementations of specific side-effects.

Interface Faceted `IO` computations take place in the `FIO` monad (the name is short for "Faceted I/O"). Figure 3 shows the public

```
data FIO a

instance Monad FIO

secureRunFIO :: FIO a -> IO a
branch :: Faceted (FIO a) -> FIO (Faceted a)
```

Figure 3. Interface for `FIO`

interface for this fragment of the library. When control flow is influenced by faceted data, the result of a computation implicitly depends on the facets observed in determining the control flow path; the implementation of `FIO` manages this transparently.

The `Monad` instance for `FIO` allows for sequencing computations in the usual way, so `FIO` should act as a (limited) drop-in replacement for `IO`. If `fio1` and `fio2` each have type `FIO Int`, then the following expression also has type `FIO Int`.

```
do v1 <- fio1
   v2 <- fio2
   return v1 + v2
```

The function `secureRunFIO` converts a value of type `FIO` a to a value of type `IO` a visible to the empty view. Thus any computation depending on private data is not observable.

The novel, and vital, component of this interface is the function `branch` function that mediates between `Faceted` values and side-effecting `FIO` computations:

$$\texttt{branch} \quad :: \quad \texttt{Faceted}\ (\texttt{FIO}\ a) \rightarrow \texttt{FIO}\ (\texttt{Faceted}\ a)$$

Consider the following fragment, where `someFacetedBool` has type `Faceted Bool` and each of `fio1` and `fio2` have type `FIO` a.

```
fio :: Faceted (FIO a)
fio = do v <- someFacetedBool
         if v then fio1 else fio2

runnable :: FIO (Faceted a)
runnable = branch fio
```

The type of `fio` is `Faceted` (`FIO` a). As in the prior example, it cannot be run. To run `fio` in a sound way, we use the function `branch` to convert from `Faceted` (`FIO` a) to `FIO` (`Faceted` a). The side effects of the resulting computation accounts for the implicit flow from `someFacetedBool`.

Implementation Somewhat analogous to how a value of type `Faceted` a simultaneously represents value of type a for all views, a computation of type `FIO` a simultaneously represents many computations of type `IO` a. For each set of influences on the control flow path, the computation performed may differ. For a particular label k, the control flow path may have been influenced by data from a private facet protected by k, data from the public facet of a value faceted on k, or neither. We call the influence of a label on the current control flow a *branch* and model it directly via the data type `Branch` as either `Private` k or `Public` k (The notations of Austin and Flanagan [2012] for these are k and $\overline{k}$, respectively).

```
data Branch = Private Label | Public Label
```

The branch `Private` k denotes a control flow path that has been influenced by the private facet of a value faceted on k. The branch `Public` k denotes a control flow path that has been influence by the public facet of a value faceted on k.

A *program counter label* is a finite set of such branches describing all the influences on the current control flow path. We model a program counter label with a list here for readability.

```
type PC = [Branch]    -- finite
```

Now we may precisely state that a value of type `FIO` a simultaneously represents some underlying `IO` a for any program counter label. In other words, a faceted computation may be represented as a function of type `PC` $\rightarrow$ `IO` a.

```
data FIO a = FIO { runFIO :: PC -> IO a }
```

This data type is private to the library, and we expose only the limited ability to run an `FIO` with an empty program counter label.

```
secureRunFIO fio = runFIO fio []
```

The implementation of `branch` simulates an optimized multi-execution, adding the appropriate branches to the program counter label when determining which computations to perform. Inconsistent facets – where the program counter label contains both `Public` k and `Private` k for any k – are pruned.

```
branch (Raw fio) = FIO (\pc -> fio >>= (return . Raw))

branch (Faceted k priv pub) = FIO branchOnPC
  where branchOnPC pc
    | Private k `elem` pc = runFIO (branch priv) pc
    | Public k  `elem` pc = runFIO (branch pub) pc
    | otherwise =
        do privV <- runFIO (branch priv) (Private k : pc)
           pubV  <- runFIO (branch pub)  (Public k : pc)
           return (Faceted k privV pubV)
```

If the label on the faceted value is already visible (resp. not visible) for current program counter label `pc`, then the private (resp. public) computation is run. If the program counter label has not been influenced by `Private k` or `Public k`, then each computation is run under the appropriate added assumption about `k`. In this last case, the computations are sequenced, creating an apparent asymmetry. However, for the actual `FIO` operations we provide – discussed below in Section 2.3, Section 2.4, and Section 2.5 – the two computations operate on disjoint facets. So the result is symmetrical, and could be executed in the opposite order. This behavior is a design constraint for any `FIO` operation.

Another way to describe `branch` is as a *distributive law* – meaning it respects the monad structures of `FIO` and `Faceted` in particular ways, see Barr and Wells [1985] – between the `Faceted` and `FIO` monads. Via `branch`, the composition `FIO` $\circ$ `Faceted` forms a compatible monad, where any number of "layers" of `FIO` and `Faceted` are collapsed. Delving further into category theory is not the aim of this paper, but intuitively this monad appears similar to the computational domain of prior work, where faceted computation and faceted values are conflated. We do conjecture that, relative to the appropriate semantic interpretation of `Faceted` and `FIO`, the function `branch` forms a *strong distributive law* [Lüth and Ghani 2002] making this compatible monad equivalent to the coproduct `FIO` $\oplus$ `Faceted`.

2.3 Faceted Reference Cells: `FIORef` a

A value of type of type `FIORef` a (short for "facet-aware `IORef`") is a mutable reference cell where initialization, reading, and writing are all `FIO` computations that operate on `Faceted` values and adjust facets according to influences on the control flow path.

Interface Figure 4 presents the core of the public interface to `FIORef` a. The interface directly parallels that of conventional reference cells of type `IORef` a.[6]

[6] Haskell has many structures providing functionality of mutable references; we are gradually adding support for other types of mutable references as well as the plethora of convenience functions provided in Haskell's standard library.

```
data FIORef a

newFIORef   :: Faceted a -> FIO (FIORef a)
readFIORef  :: FIORef a -> FIO (Faceted a)
writeFIORef :: FIORef a -> Faceted a -> FIO ()
```

Figure 4. Interface for `FIORef`

When a reference cell x of type `FIORef` a is read using `readFIORef`, it yields a value v of type `Faceted` a that is faceted according to the explicit and implicit flows that led to v being written to x.

When a reference cell x is initialized via `newFIORef` or written via `writeFIORef`, the value v (of type `Faceted` a) is stored at x in a way that takes into account all the influences that led to the initialization or write being performed.

Using this interface, we can express a minimal example of an implicit flow using `FIO` and `FIORef`. In the following, `secret` has type `Faceted Int` and `x` has type `FIORef Int` and x is initially set to `makePublic` 0.

```
do branch (do v <- secret
              if v == 42
                then writeFIORef x 1
                else return ())
   readFIORef x
```

Note the necessity of `branch`. The argument to `branch` has type `Faceted (FIO ())`, and is converted to a value of type `FIO (Faceted ())` so it can be run. The resulting faceted value read from `x` will account for the influence from the facets of `secret`. What is an implicit flow in classical examples is actually quite explicit here!

This pattern is extremely common, and can be encapsulated in the helpful combinator `ifF`:

```
ifF :: Faceted Bool -> FIO a -> FIO a -> FIO Faceted a
ifF facetedBool thenBranch elseBranch =
  branch (do v <- facetedBool
             if v then thenBranch else elseBranch)
```

The above example may then be written

```
do ifF (do { v <- secret; return v == 42 })
       (writeFIORef x 1)
       (return ())
   readFIORef x
```

Implementation The private representation for a value of type `FIORef` a is simply a reference cell that stores a faceted value, of type `IORef (Faceted` a`)`.

```
data FIORef a = FIORef (IORef (Faceted a))
```

The appropriate facets are always established upon writing a value, so reading an `FIORef` requires no special logic.

```
readFIORef (FIORef x) = FIO (\pc -> readIORef x)
```

Initialization and subsequent writes to an `FIORef` incorporate the current program counter label, building facets to indicate that the value being stored may depend on any labels in the program counter. Thus process is implemented in the auxiliary function `pcF`, corresponding to $\langle\!\langle\, pc \,?\, \mathtt{x} \,:\, \mathtt{y} \,\rangle\!\rangle$ from Austin and Flanagan [2012]. The value `pcF` $pc\ x\ y$ behaves as x to any view consistent with pc and behaves as y to any other view.

```
pcF :: PC -> Faceted a -> Faceted a -> Faceted a
pcF []                 x _ = x
pcF ((Private k) : bs) x y = Faceted k (pcF bs x y) y
pcF ((Public k)  : bs) x y = Faceted k y (pcF bs x y)
```

```
data FHandle

type View = [Label]   -- finite

openFileF :: View -> FilePath -> IOMode -> FIO FHandle

hGetCharF :: FHandle -> FIO (Faceted Char)
hPutCharF :: FHandle -> Faceted Char -> FIO ()
```

Figure 5. Interface for `FHandle`

To initialize a new `FIORef` a requires an initial value `v`. The value stored is faceted according to the current program counter, with a default value of `bottom`.

```
newFIORef v = FIO $ \pc ->
  do var <- newIORef (pcF pc v bottom)
     return (FIORef var)
```

Writing a faceted reference cell updates the stored (faceted) value in such a way that it appears unchanged to any view that is inconsistent with the current program counter label, but updated for any view that is consistent.

```
writeFIORef (FIORef x) v = FIO $ \pc ->
  do old <- readIORef x
     writeIORef x (pcF pc v old)
```

Together, these definitions ensure that the value stored for any `FIORef` always contains facets that incorporate influences due to implicit flows.

2.4 File I/O and Network Communication: `FHandle`

Faceted file I/O differs from reference cells in that the network and file system, which we collectively refer to as the *environment*, lies outside the purview of our programming language. The environment has no knowledge of facets and cannot be retrofitted. In addition, it must be assumed that there are other agents/programs able to read from and write to the file. Thus we assume that the environment's access controls are used appropriately to restrict other users of the handle. We merely provide facilities within Haskell to express the same policy that the environment is presumed to uphold.

Because the contents of the file can be written or read without warning, writing to a file and sending a message on a asynchronous channel are considered equivalent, while reading from a a file and receiving a message over an asynchronous channel are likewise considered equivalent. For the purposes of demonstrating functions that closely match standard Haskell functions, we will use files but with this network send/receive semantics.

Interface Figure 5 shows the core of the public interface for facet-aware file handles, type `FHandle` (short for "facet-aware `Handle`"). Since there are a great many operations for handles, we present just those necessary to discuss how I/O interacts with faceted computation. In particular, we focus on sends and receives of a single value of type `Char` via `hGetChar` and `hPutChar`.[7]

The policies that we support are those expressed by associating to each file handle h a finite view[8] (a finite set of labels) $view_h$ of type `View` that indicates the minimum confidentiality for data written to h and, dually, the maximum integrity of data read from h. The detailed discussion below of reading and writing to facet-aware handles clarifies the meaning of these policies.

[7] Providing performant faceted wrappers for a wide variety of operations on handles remains a work in progress.

[8] Finiteness of views for file I/O is a stronger property than absolutely necessary. Our design requires at least that inclusion and non-inclusion of a label in a view be effectively computable.

The function `openFileF` accept a view $view_h$ along with a file path and mode and returns a (computation that returns a) facet-aware handle h protected by the policy expressed by $view_h$.

When writing to a handle h via `hPutCharF`, the view $view_h$ serves as an upper bound on the confidentiality assured by the external world (a lower bound on the integrity expected) for data written to h. In other words, we trust that the external world will protect the data with those labels in $view_h$. Dually, we certify that no labels beyond those in $view_h$ have influenced the values sent. Thus if the program counter label has been influenced by higher confidentiality (lower integrity) values, the write must not occur. Furthermore, the exact facet that is written is just that visible to $view_h$. Consider the following example.

```
fio :: FIO ()
fio = do h <- openFileF ["z", "w"] "/tmp/foo" WriteMode
         hPutCharF h (makeFacets "z" 'a' 'b')
         hPutCharF h (makeFacets "x" 'c' 'd')
```

First, consider running `runFIO fio ["z", "w", "x"]`. Because the label `"x"` is in the PC passed to `runFIO`, it is considered to have influenced control flow, and no writes will occur because `"x"` is not contained in $view_h$. Information confidential to `"x"` would be leaked by the mere fact of a write occurring.

Next, consider running `runFIO fio ["z"]`. Because `["z"]` is a subset of $view_{\text{h}}$, the writes are permitted. The first call to `hPutChar` will output `'a'` because `"z"` is a contained in $view_{\text{h}}$. The second call to `hPutChar` will output `'d'` because `"x"` is not contained in $view_{\text{h}}$.

When reading from a handle h via `hGetCharF`, we treat $view_h$ as a lower bound on the confidentiality expected by the external world (an upper bound on the integrity it promises) for data read from h. In other words, we trust that the external world will prevent tainting from data with labels not in $view_h$. Dually, we certify that we will protect the data with sufficient security to avoid leaking to lesser views.[9] For example, consider the following computation.

```
do h <- openFileF ["k", "l"] "/tmp/socket.0" ReadMode
   hGetCharF h
```

The character thus read from `h` is confidential, observable only to views that include labels `"k"` and `"l"`. Dually, the value is considered to be tainted by labels `"k"` and `"l"`, which may represent untrusted parties that should not be able to influence control flow.

Implementation The representation of a value of type `FHandle` is as a pair of a `Handle` and a `View`. The function `openFileF` is a simple wrapper on the standard library function `openFile`.

```
data FHandle = FHandle View Handle

openFileF :: View -> FilePath -> IOMode -> FIO FHandle
openFileF view path mode = FIO $ \pc ->
  do handle <- openFile path mode
     return (FHandle view handle)
```

The implementation of `hGetCharF` creates appropriate facets to provide information flow security for values received from a handle h. To any view that is not a superset of $view_h$, the value will appear undefined.

```
hGetCharF :: FHandle -> FIO (Faceted Char)
hGetCharF (FHandle view handle) = FIO hGetCharForPC
  where hGetCharForPC pc =
    do ch <- hGetChar handle
       return (pcF (map Private view) (Raw ch) bottom)
```

[9] We assume an asynchronous protocol, so an external writer can never observe whether a read has taken place.

The implementation of `hPutCharF` actually performs a write only when $view_h$ is visible to the current program counter label, as computed by the auxilliary function `visibleTo`. When writing a faceted character ch, the non-facted character written is the projection of ch to the view $view_h$, as computed by the auxillary function `project`.

```
visibleTo :: PC -> View -> Bool
visibleTo pc view = all consistent pc
  where consistent (Private k)  = k `elem` view
        consistent (Public k) = k `notElem` view

project :: View -> Faceted a -> a
project view (Raw v) = v
project view (Faceted k priv pub)
  | k `elem`    view = project view priv
  | k `notElem` view = project view pub

hPutCharF :: FHandle -> Faceted Char -> FIO ()
hPutCharF (FHandle view handle) ch = FIO hPutCharForPC
  where hPutCharForPC pc
    | view `visibleTo` pc =
        hPutChar handle (project view ch)
    | otherwise = return ()
```

For example, if $view_h$ is `["k"]` then the computation

$$\texttt{hPutCharF}\ h\ (\texttt{Faceted "k"}\ (\texttt{Faceted "l" 'a' 'b'})\ \texttt{'c'}))$$

will write the character `'b'` to h, because the private component of the `"k"` facet will be selected, while the public component of the `"l"` facet will be selected.

To briefly return to the fact that `Faceted` is the free monad over the `Facets` functor from Section 2.1, we note that each projection `project` $view$ can be expressed as a monad algebra for `Free Facets` generated from the following function `proj` $view$:

```
proj :: View -> Facets a -> a
proj view (Facets k priv pub)
  | k `elem`    view = priv
  | k `notElem` view = pub

foldFree :: (f a -> a) -> (Free f a -> a)
foldFree f (Pure x)  = x
foldFree f (Free fx) = f (fmap (foldFree f) fx)

project view = foldFree (proj view)
```

2.5 Robust Declassification

Faceted values support *robust declassification* [Zdancewic and Myers 2001]. This requires that labels indicate both secrecy and integrity. We still leave the type of labels as `String` to keep our presentation consistent. A label k can be interpreted by the program variously as "secret to k" or "untrusted by k", according to the string. We emphasize that this is for demonstration purposes only; a finalized design for a system with declassification should have structured types for principals and labels, interfaced with our library through appropriate type classes.

Interface Figure 6 shows the interface for declassification, including functions `untrustedBy` and `secretTo` that provide labels with the appropriate semantics.

Since the decision to declassify a value depends on the program counter label, declassification takes place "in" the `FIO` monad. If the program counter label has been influenced by untrusted labels, then the declassification operation is not performed.

Implementation The implementations of `untrustedBy` and `secretTo` need not be private nor cryptographically secure (labels in our system are forgeable in general) so we simply prepend `"untrustedBy"` or `"secretTo"` to a label.

```
type Principal = String

untrustedBy :: Principal -> Label
secretTo :: Principal -> Label

declassify :: Principal -> Faceted a -> FIO (Faceted a)
```

Figure 6. Interface for declassification

```
untrustedBy k = "untrustedBy" ++ k
secretTo k = "secretTo" ++ k
```

Due to the tight format here, we alias `untrustedBy` to `u` and `secretTo` to `s` in the following definition for `declassify`.

```
u = untrustedBy
s = secretTo

declassify :: Principal -> Faceted a -> FIO (Faceted a)
declassify p v = FIO $ \pc ->
  if (untrustedBy p) `notElem` pc
    then return downgrade p v
    else return v

downgrade :: Principal -> Faceted a -> Faceted a
downgrade p (Raw x) = Raw x

downgrade p v@(Faceted k priv pub)
  | k == s p  =  Faceted (u p) v priv

downgrade p (Faceted k priv pub)
  | k == u p  =  Faceted (u p) priv (downgrade p pub)

downgrade p (Faceted k priv pub) =
  Faceted k (downgrade p priv) (downgrade p pub)
```

If the program counter label is such that declassification is permissible, the function `downgrade` performs the declassification for principle p as follows:

For raw values and undefined values, `downgrade` behaves as the identity function.

For a value v faceted by the label `secretTo` p, the result is faceted on `untrustedBy` p. The private facet of v is moved into the public facet of the result – indicating that it is trusted by p – while the private facet of the result remains as v. Thus for any view containing the label `untrustedBy` p, the result behaves as the fully classified value v.

For a value faceted by the label `untrustedBy` p, only the public facet (trusted by p) is downgraded.

For a value faceted by any other label, the two facets are downgraded independently.

These rules ensure that declassification, when allowed, moves data from private facets of `secretTo` p to public facets, but no other facets are changed.

3. Termination-Insensitive Noninterference

Faceted computations ensure a form of security known as *termination-insensitive noninterference*. This means that for any two terminating programs, low confidentiality return values and outputs cannot be influenced by high-confidentiality values or inputs.

Faceted computations ensure termination-insensitive noninterference by simulating separate computations for each view. Any faceted value has a *projection* for every view, which is the non-faceted value visible to that view. To demonstrate termination-insensitive noninterference, we formalize the intuition that faceted computations simulate secure multi-execution, by explicitly describing the non-faceted computations being simulated for each view.

3.1 Projection Properties for Pure Faceted Values

Recall that a view is a set of labels. A view L thus defines a complete partition of the universe of labels: Those that are contained in L, for which the private facet should be considered, and those that are not, for which the public facet should be considered. A value of type `Faceted` a may be interpreted as simultaneously representing a value of type a for any view L. This is formalized by the following interpretation of a faceted term as a function from `Faceted` a to a. This interpretation appears identical to the Haskell implementation of `project` in Section 2.4, but is mathematically well-defined for all views, not only those expressible as Haskell lists.[10]

$$\begin{array}{rcl} [\![-]\!] :: \texttt{Faceted}\ a & \to & \mathcal{P}(\texttt{Label}) \to a \\ [\![\texttt{Raw}\ e]\!](L) & = & e \\ [\![\texttt{Faceted}\ k\ a\ b]\!](L) \mid k \in L & = & [\![a]\!](L) \\ \mid k \notin L & = & [\![b]\!](L) \end{array}$$

The `Functor`, `Applicative`, and `Monad` structures upon faceted values adhere to simple properties with respect to this interpretation.

Lemma 1 (View Projection for Pure Values). For any view L and faceted values of the necessary types for each equation,

$$\begin{array}{rcl} [\![\texttt{fmap}\ f\ e]\!](L) & = & f\ [\![e]\!](L) \\ [\![f\ \texttt{<*>}\ e]\!](L) & = & [\![f]\!](L)\ [\![e]\!](L) \\ [\![e\ \texttt{>>=}\ f]\!](L) & = & [\![f\ [\![e]\!](L)]\!](L) \end{array}$$

However, the above interpretation is not the same as the *projection* of the literature; the result of interpretation may still be faceted. The interpretation of a value of type `Faceted` (`Faceted` a) for a particular view L has type `Faceted` a. The same information flow guarantees apply to this value. The intuition behind projection is that it should remove *all* facets from a program, leaving the residual program that characterizes what L may observe. Also, we expect projection to fulfill the standard idempotence criterion: For a function f to be a projection, we must have $f \circ f = f$. This is clearly not true of any transformation that always removes only a single layer of facets.

Formalizing this intuition, we write $\lfloor a \rfloor$ to denote the type of the output of projection. We define this type generically by induction on the structure of a, as constructed from products, sums, and arrows. User-defined types have natural definitions derived from these.

$$\begin{array}{rcl} \lfloor - \rfloor :: * & \to & * \\ \lfloor \texttt{Faceted}\ a \rfloor & = & \lfloor a \rfloor \\ \lfloor K \rfloor & = & K \\ \lfloor a \times b \rfloor & = & \lfloor a \rfloor \times \lfloor b \rfloor \\ \lfloor a + b \rfloor & = & \lfloor a \rfloor + \lfloor b \rfloor \\ \lfloor a \to b \rfloor & = & \lfloor a \rfloor \to \lfloor b \rfloor \end{array}$$

For example, $\lfloor \texttt{Faceted}\ (\texttt{Faceted Int}) \rfloor$ is `Int`. This extends to algebraic data types such as lists in the obvious manner; for example, $\lfloor \texttt{[Faceted}\ a\texttt{]} \rfloor$ is `[`a`]`. Note that $\lfloor \lfloor a \rfloor \rfloor = \lfloor a \rfloor$; the type-level function $\lfloor - \rfloor$ is idempotent. The most important case to note is that for a function type $a \to b$, where facets in the argument type a are removed covariantly.

We now define idempotent projection for all Haskell values, which removes all possibly-nested facets recursively. We write

[10] Note that, as for `project` in Section 2.4, this interpretation is induced via the free monad structure of `Faceted` via a corresponding functor algebra over `Facets`.

$L(e)$ to denote the projection of the expression e according to the view L. We define the projection by L generically by induction over the structure of Haskell types, denoting projection for type a as L_a. For any type a, the component of the projection L_a is a map from a to $\lfloor a \rfloor$.[11] Projection is the identity on base types and extends naturally to products and sums, but to define projection across all Haskell data types, including contravariant arrow types, we provide an *embedding-projection pair*. That is, the projection by L is defined simultaneously with the injection $\tilde{L}$, where $L \circ \tilde{L} \sqsubseteq id$. Injection on $\texttt{Faceted}\ a$ is $\texttt{Raw}$ and extends to products, sums, and arrows in a manner exactly dual to projection.[12]

$$
\begin{array}{rcl}
L :: a & \to & \lfloor a \rfloor \\
L_{\texttt{Faceted}\ a}(v) & = & L_a([\![v]\!](L)) \\
L_K(v) & = & v \\
L_{a \times b}(x, y) & = & (L_a(x), L_b(y)) \\
L_{a+b}(Inl\ x) & = & Inl\ L_a(x) \\
L_{a+b}(Inr\ y) & = & Inr\ L_b(y) \\
L_{a \to b}(f) & = & L_b \circ f \circ \tilde{L}_a
\end{array}
$$

$$
\begin{array}{rcl}
\tilde{L} :: \lfloor a \rfloor & \to & a \\
\tilde{L}_{\texttt{Faceted}\ a}(v) & = & \texttt{Raw}\ v \\
\tilde{L}_K(v) & = & v \\
\tilde{L}_{a \times b}(x, y) & = & (\tilde{L}_a(x), \tilde{L}_b(y)) \\
\tilde{L}_{a+b}(Inl\ x) & = & Inl\ \tilde{L}_a(x) \\
\tilde{L}_{a+b}(Inr\ y) & = & Inr\ \tilde{L}_b(y) \\
\tilde{L}_{a \to b}(f) & = & \tilde{L}_b \circ f \circ L_a
\end{array}
$$

Taking all components of projection as together defining a function over all Haskell data, we have that for any view L and argument e, $L(L(e)) = L(e)$. (Note that at the type level, $\lfloor \lfloor a \rfloor \rfloor = \lfloor a \rfloor$).

3.2 The Projection Property for Reference Cells

In order to discuss the projection property for stateful computations, we provide a minimal semantics to a small model of just those operation of interest, following work such as Peyton Jones et al. [1996] and Swierstra and Altenkirch [2007]. We otherwise presume an informal understanding of the semantics of pure Haskell expressions.

The only $\texttt{IO}$ constructors we consider are the following, and assume all values of type $\texttt{IO}\ a$ are constructed from them, treating this as the recursive specification of a GADT.

$$
\begin{array}{rcl}
\texttt{return} & :: & a \to \texttt{IO} \\
(\texttt{>>=}) & :: & \texttt{IO}\ a \to (a \to \texttt{IO}\ b) \to \texttt{IO}\ b \\
\texttt{newIORef} & :: & a \to \texttt{IO}\ (\texttt{IORef}\ a) \\
\texttt{readIORef} & :: & \texttt{IORef}\ a \to \texttt{IO}\ a \\
\texttt{writeIORef} & :: & \texttt{IORef}\ a \to a \to \texttt{IO}\ ()
\end{array}
$$

Each of these operates on a standard store which may be modeled as a partial function from locations of type $\texttt{IORef}\ a$ to values of type a. The following use of mathematical symbols Σ and $\rightharpoonup$ is to emphasize that this is a model, not Haskell code.

$$
\Sigma \quad :: \quad \texttt{IORef}\ a \rightharpoonup a
$$

The following meta-function run maps a value of type $\texttt{IO}\ a$ to its interpretation as a function from a store to a pair of a store and a return value. These semantics are completely standard, available in Haskell as an instance of the $\texttt{State}$ monad, but reproduced here only for ease of reference.

$$
\begin{array}{rcll}
run\ (\texttt{return}\ e)\ \Sigma & = & (\Sigma, e) & \\
run\ (\texttt{newIORef}\ e)\ \Sigma & = & (\Sigma[l \mapsto e], l) & l \notin dom(\Sigma) \\
run\ (\texttt{readIORef}\ l)\ \Sigma & = & (\Sigma, \Sigma(l)) & \\
run\ (\texttt{writeIORef}\ l\ e)\ \Sigma & = & (\Sigma[l \mapsto e], ()) & \\
run\ (e\ \texttt{>>=}\ f)\ \Sigma & = & run\ (f\ v)\ \Sigma' & \\
 & \text{where} & run\ e\ \Sigma = (\Sigma', v) &
\end{array}
$$

We can similarly characterize all the constructors of interest for the type $\texttt{FIO}\ a$:

$$
\begin{array}{rcl}
\texttt{return} & :: & a \to \texttt{FIO}\ a \\
(\texttt{>>=}) & :: & \texttt{FIO}\ a \to (a \to \texttt{FIO}\ b) \to \texttt{FIO}\ b \\
\texttt{newFIORef} & :: & a \to \texttt{FIO}\ (\texttt{FIORef}\ a) \\
\texttt{readFIORef} & :: & \texttt{FIORef}\ a \to \texttt{Faceted}\ (\texttt{FIO}\ a) \\
\texttt{writeFIORef} & :: & \texttt{FIORef}\ a \to a \to \texttt{FIO}\ () \\
\texttt{branch} & :: & \texttt{Faceted}\ (\texttt{FIO}\ a) \to \texttt{FIO}\ (\texttt{Faceted}\ a)
\end{array}
$$

Any faceted computation of type $\texttt{FIO}\ a$ projects to a non-faceted computation of type $\texttt{IO}\ \lfloor a \rfloor$, in which all faceted reference cells of type $\texttt{FIORef}\ a$ have been projected to non-faceted reference cells of type $\texttt{IORef}\ \lfloor a \rfloor$. In other words,

$$
\begin{array}{rcl}
\lfloor \texttt{FIO}\ a \rfloor & = & \texttt{IO}\ \lfloor a \rfloor \\
\lfloor \texttt{FIORef}\ a \rfloor & = & \texttt{IORef}\ \lfloor a \rfloor
\end{array}
$$

Projection and injection for reference cells is a no-op that just coerces the type. The projection and corresponding injection for faceted computations are each straightforward. Since the types are necessary and inferable, we omit subscripts on L to save space and clutter.

$$
\begin{array}{rcl}
L(\texttt{FIORef}\ l) & = & l \\
\tilde{L}(l) & = & \texttt{FIORef}\ l
\end{array}
$$

[11] We exclude polymorphism from our presentation because the most natural way to discuss it requires introducing System-F style explicit instantiation. The equation for universally quantified types is $L_{\forall X.a}(e) = \Lambda X.L_X(e)$ and existential types may be derived. But this considerably complicates the presentation.

[12] Hinze and Jeuring [2003] discuss mechanisms for defining generic embedding-projection pairs in Generic Haskell, but here we define it only as a meta-function. This is also why we favor mathematical notation $a \times b$ and $a + b$ for product and sum types rather than the respective Generic Haskell syntax of $a\texttt{:*:}b$ and $a\texttt{:+:}b$.

$$
\begin{array}{rcl}
L_{\texttt{FIO}\ a}\ ::\ \texttt{FIO}\ a & \to & \texttt{IO}\ \lfloor a \rfloor \\
L(\texttt{return}\ e) & = & \texttt{return}\ L_a(e) \\
L(e\ \texttt{>>=}\ f) & = & L(e)\ \texttt{>>=}\ (L \circ f) \\
L(\texttt{newFIORef}\ e) & = & \texttt{newIORef}\ L(e) \\
L(\texttt{readFIORef}\ l)) & = & \texttt{readIORef}\ L(l)\ \texttt{>>=}\ L \\
L(\texttt{writeFIORef}\ l\ v) & = & \texttt{writeIORef}\ L(l)\ L(v) \\
L(\texttt{branch}\ e) & = & L(e) \\
\tilde{L} & = & \texttt{FIO} \circ \texttt{const}
\end{array}
$$

Projection of a store Σ is extended point-wise to each of the locations in Σ. Formally

$$L(\Sigma)(l) \quad = \quad L(\Sigma(l))$$

Projection changes the types for every location in the store; the types for each projected `IORef` in a program change accordingly.

Having defined the semantics of `IO` with reference cells and projection for `FIO` computations, we can state the key projection property for faceted computations:

Lemma 2 (Projection with Mutable Reference Cells). For any store Σ, program counter label pc, and expression e `::` `FIO` a, if

$$run\ (\texttt{runFIO}\ e\ pc)\ \Sigma \quad = \quad (\Sigma', v)$$

then for any view L which is consistent with pc,

$$run\ L(e)\ L(\Sigma) \quad = \quad (L(\Sigma'), L(v)))$$

Proof: Proceed by induction on the construction of e. Each each case follows by basic equational reasoning.□

3.3 The Projection Property for I/O

To keep the formal demonstration easy to read, we develop the semantics and projection property for I/O orthogonally to that for reference cells. The constructors we are interested in for `IO` in this case are:

$$
\begin{array}{rcl}
\texttt{hGetChar} & :: & \texttt{Handle} \to \texttt{IO Char} \\
\texttt{hPutChar} & :: & \texttt{Handle} \to \texttt{Char} \to \texttt{IO ()}
\end{array}
$$

For simplicity we omit `openFile` and simply presume some global set of handles are available.

We redefine the interpretation of I/O now to be a mapping from streams of input characters to output characters – one input stream per handle, and one output stream per handle.

$$\mathcal{R}, \mathcal{W} \quad :: \quad \texttt{Handle} \to \texttt{[Char]}$$

The semantics of a program are the following. Again, this is a straightforward elaboration of standard semantics, easily implemented with a combination of the `Reader` and `Writer` monads, fully elaborated here for ease of reference.[13] Below, list concatenation `++` on $\mathcal{W}$ is performed point-wise for each handle, and $\emptyset$ denotes a mapping from every handle to an empty stream.

$$
\begin{array}{rcl}
run\ (\texttt{return}\ e)\ \mathcal{R} & = & (\mathcal{R}, \emptyset, e) \\
run\ (\texttt{hPutChar}\ h\ c)\ \mathcal{R} & = & (\mathcal{R}, [h \mapsto c], ()) \\
run\ (\texttt{hGetChar}\ h)\ \mathcal{R} & = & (\mathcal{R}[h \mapsto es], \emptyset, e) \\
 & \text{where} & \mathcal{R}(h) = e{:}es \\
run\ (e\ \texttt{>>=}\ f)\ \mathcal{R} & = & (\mathcal{R}'', \mathcal{W}\texttt{++}\mathcal{W}', v') \\
 & \text{where} & run\ e\ \mathcal{R} = (\mathcal{R}', \mathcal{W}, v) \\
 & \text{and} & run\ (f\ v)\ \mathcal{R}' = (\mathcal{R}'', \mathcal{W}', v')
\end{array}
$$

We can characterize the `FIO` constructors of interest for I/O as well:

$$
\begin{array}{rcl}
\texttt{hGetCharF} & :: & \texttt{FHandle} \to \texttt{FIO (Faceted Char)} \\
\texttt{hPutCharF} & :: & \texttt{FHandle} \to \texttt{Faceted Char} \to \texttt{FIO ()}
\end{array}
$$

Their projections should, by now, be unsurprising. We assume that the appropriate $view_h$ for each handle is known in order to inject `Handle` into `FHandle`.

$$
\begin{array}{rcl}
L(\texttt{FHandle}\ view_h\ h) & = & h \\
\tilde{L}(h) & = & \texttt{FHandle}\ view_h\ h \\
 & & \\
L(\texttt{return}\ e) & = & \texttt{return}\ L(e) \\
L(\texttt{hGetCharF}\ h) & = & \texttt{hGetChar}\ L(h)\ \texttt{>>=}L \\
L(\texttt{hPutCharF}\ h\ v) & & \\
\quad \mid L \subseteq view_h & = & \texttt{hPutChar}\ L(h)\ L(v) \\
\quad \mid \text{otherwise} & = & \texttt{return ()} \\
\tilde{L} & = & \texttt{FIO} \circ \texttt{const}
\end{array}
$$

Projection of an input environment $\mathcal{R}$ is point-wise for each handle h. Projection for a particular input stream $\mathcal{R}(h)$ by L leaves $\mathcal{R}$ unchanged if L is permitted to read from h, in other words $L \supseteq view_h$. Otherwise the input stream is projected to an input stream containing only undefined values. Conversely, an output stream is projected to an empty stream if $L \subseteq view_h$. Thus, even if the output of a faceted computation varies, the output when projected by L will remain empty, reflecting the fact that L cannot be "held responsible" for the changes in the output to streams to which it can never write.

$$
\begin{array}{rcl}
L(\mathcal{R})(h) \mid L \supseteq view_h & = & \mathcal{R}(h) \\
\quad \mid \text{otherwise} & = & \bot, \bot, \cdots \\
 & & \\
L(\mathcal{W})(h) \mid L \subseteq view_h & = & \mathcal{W}(h) \\
\quad \mid \text{otherwise} & = & \texttt{[]}
\end{array}
$$

The projection property follows:

Lemma 3 (Projection with File I/O). For any input environment $\mathcal{R}$, program counter label pc, and expression e `::` `FIO` a, if

$$run\ (\texttt{runFIO}\ e\ pc)\ \mathcal{R} \quad = \quad (\mathcal{R}', \mathcal{W}, v)$$

then for any view L which is consistent with pc,

$$run\ L(e)\ L(\mathcal{R}) \quad = \quad (L(\mathcal{R}'), L(\mathcal{W}), L(v)))$$

Proof: Proceed by induction on the construction of e. Each case follows by basic equational reasoning.□

[13] We could avoid having the residual input streams in our output if we embraced the following nondeterministic split of the input.

$$
\begin{array}{rcl}
run\ (e\ \texttt{>>=}\ f)\ (\mathcal{R}\texttt{++}\mathcal{R}') & = & (\mathcal{W}\texttt{++}\mathcal{W}', v') \\
 & \text{where} & run\ e\ \mathcal{R} = (\mathcal{W}, v) \\
 & \text{and} & run\ (f\ v)\ \mathcal{R}' = (\mathcal{W}', v')
\end{array}
$$

This would make our semantics more like a judgment-based big-step semantics instead of a straightforward-to-implement denotation.

3.4 Termination-Insensitive Noninterference

Projection induces equivalence relations on stores, input streams, output streams, and expressions, namely:

$$\begin{array}{rcl} e_1 \sim_{pc} e_2 & \triangleq & L(e_1) = L(e_2) \\ \Sigma_1 \sim_L \Sigma_2 & \triangleq & L(\Sigma_1) = L(\Sigma_2) \\ \mathcal{R}_1 \sim_L \mathcal{R}_2 & \triangleq & L(\mathcal{R}_1) = L(\mathcal{R}_2) \\ \mathcal{W}_1 \sim_L \mathcal{W}_2 & \triangleq & L(\mathcal{W}_1) = L(\mathcal{W}_2) \end{array}$$

Combining the prior semantics (they are orthogonal), we postulate a run function that accepts an `IO` computation, an initial store, and input stream, and returns the resulting output stream and return value. Using this version of run, termination-insensitive noninterference may be stated and proved.

Theorem 1 (Termination-Insensitive Noninterference). For any expressions e_1 and e_2, initial stores Σ_1 and Σ_2, input streams $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $e_1 \sim_L e_2$ and $\Sigma_1 \sim_L \Sigma_2$ and $\mathcal{R}_1 \sim_L \mathcal{R}_2$, and

$$\begin{array}{rcl} run\ (\texttt{secureRunFIO}\ e_1)\ \Sigma_1\ \mathcal{R}_1 & = & (\Sigma_1', \mathcal{W}_1', v_1) \\ run\ (\texttt{secureRunFIO}\ e_1)\ \Sigma_2\ \mathcal{R}_2 & = & (\Sigma_2', \mathcal{W}_2', v_2) \end{array}$$

then $\Sigma_1' \sim_L \Sigma_2'$ and $\mathcal{W}_1' \sim_L \mathcal{W}_2'$ and $v_1 \sim_L v_2$

3.5 Correctness for Declassification

In the presence of declassification, the projection property as stated in other cases cannot hold. However, we can prove the following lemma, following Austin and Flanagan [2012], that indicates our `downgrade` function migrates data from the trusted secret view to the trusted public view

Lemma 4 (Projection with Declassification). For any faceted value e :: `Faceted` a and view L,

$$[\![\texttt{downgrade}\ p\ e]\!](L) = \begin{cases} [\![e]\!](L) & \text{if } (\texttt{untrustedBy}\ p) \in L \\ [\![e]\!](L \cup \{\texttt{secretTo}\ p\}) & \text{otherwise} \end{cases}$$

Proof: Proceed by induction on e.

4. Related Work

Most information flow mechanisms fall into one of three categories: run-time monitors that prevent a program execution from misbehaving; static analysis techniques that analyze the whole program and reject programs that might leak sensitive information; and finally secure multi-execution, which protects sensitive information by evaluating the same program multiple times.

Dynamic techniques dominated much of the early literature, such as Fenton's memoryless subsystems [Fenton 1974]. However, these approaches tend to deal poorly with *implicit flows*, where confidential information might leak via the control flow of the program; purely dynamic controls either ignore updates to reference cells that might result in implicit leaks of information [Fenton 1974] or terminate the program on these updates [Zdancewic 2002; Austin and Flanagan 2009]; both approaches have obvious problems, but these techniques have seen a resurgence of interest as a possible means of securing JavaScript code, where static analysis seems to be an awkward fit [Dhawan and Ganapathy 2009; Jang et al. 2010; Hedin and Sabelfeld 2012; Kerschbaumer et al. 2013].

Denning's work [Denning 1976; Denning and Denning 1977] instead uses a static analysis; her work was also instrumental in bringing information flow analysis into the scope of programming language research. Her approach has since been codified into different type systems, such as that of Volpano et al. [1996] and the SLam Calculus [Heintze and Riecke 1998]. Jif [Myers 1999] uses this strategy for a Java-like language, and has become one of the more widespread languages providing information flow guarantees. Sabelfeld and Myers [2003] provide an excellent history of information flow analysis research prior to 2003. For a more detailed comparison of the benefits of dynamic controls and static analysis for information flow guarantees, see Russo and Sabelfeld [2010]'s discussion between static and dynamic analyses.

Secure multi-execution [Devriese and Piessens 2010] executes the same program multiple times representing different "views" of the data. For a simple two-element lattice of high and low, a program could be executed twice, once where confidential (high) data is included but may only write to authorized channels, and again where high data has been replaced by default values and which writes to public channels. This approach has since been implemented in the Firefox web browser [De Groef et al. 2012] and as a Haskell library [Jaskelioff and Russo 2012]. Rafnsson and Sabelfeld [2013] show an approach to handle declassification and to guarantee transparency with secure multi-execution.

Zanarini et al. [2013] note some challenges with secure multi-execution; specifically, it alters the behavior of programs violating noninterference (potentially introducing difficult to analyze bugs), and the multiple processes might produce outputs to different channels in a different order than expected. They further address these challenges through a *multi-execution monitor*. In essence, their approach executes the original program without modification and compares its results to the results of the SME processes; if output of secure multi-execution differs from the original at any point, a warning can be raised to note that the semantics have been altered.

Faceted evaluation [Austin and Flanagan 2012] simulates secure multi-execution by the use of special faceted values, which track different views for data based on the security principals involved[14]. While faceted evaluation cannot be parallelized as easily, it avoids many redundant calculations, thereby improving efficiency [Austin and Flanagan 2012]. It also allows declassification, where private data is released to public channels. Austin et al. [2013] exploit this benefit to incorporate policy-agnostic programming techniques, allowing for the specification of more flexible policies than traditionally permitted in information flow systems.

Research on information flow analysis in Haskell has been limited, perhaps in part because idiomatic pure Haskell code avoids the complexity of implicit flows. Nonetheless, side effects are important for Haskell programming, and information flow security for Haskell in the presence of implicit flows has been studied before.

Li and Zdancewic [2006] implement an information flow system in Haskell, embedding a language for creating secure modules. Their enforcement mechanism is dynamic but relies on static enforcement techniques, effectively guaranteeing the security of the system by type checking the embedded code at runtime. Their system supports declassification, a critical requirement for specifying many real world security policies. Russo et al. [2008] provide a monadic library guaranteeing information flow properties. Their approach includes special declassification combinators, which can be used to restrict the release of data based on the what/when/who dimensions proposed by Sabelfeld and Sands [2009].

Devriese and Piessens [2011] illustrate how to enforce information flow in monadic libraries. A sequence operation e_1 `>>` e_2 is distinguished from a bind operation e_1 `>>=` e_2 in that there are no implicit flows with the `>>` operator. They demonstrate the general-

[14] Faceted values are closely related to the value pairs used by Pottier and Simonet [2003]; while intended as a proof technique rather than a dynamic enforcement mechanism, the construct is essentially identical.

ity of their approach by applying it to classic static [Volpano et al. 1996], dynamic [Sabelfeld and Russo 2010], and hybrid [Guernic et al. 2006] information flow systems.

Stefan et al. [2011] use a *labeled IO* (LIO) monad to guarantee information flow analysis. LIO tracks the current label of the execution, which serves as an upper bound on the labels of all data in lexical scope. IO is permitted only if it would not result in an implicit flow. It combines this notion with the concept of a *current clearance* that limits the maximum privileges allowed for an execution, thereby eliminating the termination channel. Buiras and Russo [2013] show how lazy evaluation may leak secrets with LIO through the use of the *internal timing covert channel*. They propose a defense against this attack by duplicating shared thunks.

References

T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*, PLAS '09. ACM Press, 2009.

T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 165–178, New York, NY, USA, 2012. ACM Press.

T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, page 15–26, New York, NY, USA, 2013. ACM Press.

M. Barr and C. Wells. *Toposes, triples and theories*, volume 278. Springer-Verlag New York, 1985.

P. Buiras and A. Russo. Lazy programs leak secrets. In H. R. Nielson and D. Gollmann, editors, *Secure IT Systems*, Lecture Notes in Computer Science, pages 116–122. Springer Berlin Heidelberg, Jan. 2013.

W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 748–759, New York, NY, USA, 2012. ACM Press.

D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

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

D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Symposium on Security and Privacy*, pages 109–124, Los Alamitos, CA, USA, 2010. IEEE.

D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '11, page 59–72, New York, NY, USA, 2011. ACM Press.

M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-Based browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, pages 382–391. IEEE, 2009.

J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *In ASIAN'06: the 11th Asian Computing Science Conference on Secure Software*, 2006.

D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Computer Security Foundations Symposium (CSF)*. IEEE, 2012.

N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM, 1998.

R. Hinze and J. Jeuring. Generic haskell: Practice and theory. In *Generic Programming*, page 1–56. Springer, 2003.

D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security*, pages 270–283, 2010.

M. Jaskelioff and A. Russo. Secure multi-execution in haskell. In *Proceedings of the 8th International Conference on Perspectives of System Informatics*, PSI'11, page 170–178, Berlin, Heidelberg, 2012. Springer-Verlag.

C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Towards precise and efficient information flow control in web browsers. In *Trust and Trustworthy Computing Conference*, pages 187–195. Springer, 2013.

E. Kmett. free: Monads for free, May 2014. `https://github.com/ekmett/free`.

P. Li and S. Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, CSFW '06, page 16–, Washington, DC, USA, 2006. IEEE Computer Society.

C. Lüth and N. Ghani. Composing monads using coproducts. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 133–144, New York, NY, USA, 2002. ACM.

A. C. Myers. JFlow: practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.

S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 295–308, New York, NY, USA, 1996. ACM.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, Jan. 2003.

W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 33–48, June 2013.

A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, page 13–24, New York, NY, USA, 2008. ACM.

A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, page 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

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

A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI'09, page 352–365, Berlin, Heidelberg, 2010. Springer-Verlag.

A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, Oct. 2009.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, page 95–106, New York, NY, USA, 2011. ACM.

W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, page 25–36. ACM, 2007.

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

D. Zanarini, M. Jaskelioff, and A. Russo. *Precise Enforcement of Confidentiality for Reactive Systems*. 2013.

S. Zdancewic and A. C. Myers. Robust declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, page 15–23. IEEE Computer Society Press, 2001.

S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.