CHALMERS | UNIVERSITY OF GOTHENBURG

SWAP-IFC

Secure Web Applications with Information Flow Control

Master of Science Thesis in Computer Science

ALEXANDER SJÖSTEN

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, February 2015

SWAP-IFC
Secure Web Applications with Information Flow Control

ALEXANDER SJÖSTEN

Examiner: ANDREI SABELFELD

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden February 2015

Abstract

This thesis explores the possibility to create a library in Haskell which enables a static analysis with regards to information flow control. This library should then be compiled with Haste and produce secure JavaScript code with regards to information flow control. In doing so, the compiled code should be able to be run through JSFlow information flow control-enforcing JavaScript interpreter with no halted execution due to information leakage.

In order to create the library, three different prototypes were developed. From these prototypes, the most promising was selected. Once a proper library implementation, which involves integration with Haste and code generated towards JSFlow, had been created, thorough testing was performed to verify correctness.

Creating a secure web application with regard to information flow control poses a big challenge and there has been a lot of research in the area of information flow control. When creating a web application, a language like JavaScript is usually used. Since JavaScript is deployed in the browser and can gain access to sensitive information, securing JavaScript application with regards to information flow control is crucial and to help with this, a dynamic interpreter called JSFlow has been developed at Chalmers University of Technology.

However, it is not enough to secure JavaScript with regards to information flow control. Research has been made to help strengthen the weak type system of JavaScript. The research includes creating new languages and creating compilers. The compiler Haste generates JavaScript code from the high-level, strict statically typed language Haskell.

Acknowledgements

I would like to give thanks to Andrei Sabelfeld for acting as my examiner for this thesis and providing valuable input. I would also like to thank Anton Ekblad for his supervision and sharing his knowledge when giving advice and valuable feedback on the draft of the report; Daniel Hedin for helping out with questions about JSFlow. I would also like to thank my opponents Anders Nordin, Hannes Sandahl and Daniel Eddeland for their feedback during my presentation. Finally, a big thank goes to all computer scientists in our lunch room Monaden for great laughs and good discussions about everything and nothing.

Alexander Sjösten, February 26, 2015, Gothenburg

Contents

1

Introduction

When developing a web application, there are numerous issues the application writer must consider. One of the most used languages when designing the front-end of a web application is JavaScript [1], a dynamically typed, multi-paradigm programming language [2]. Unfortunately, JavaScript suffers from a couple of drawbacks and attacks towards the language through Cross-Site Scripting (XSS) were on the third place on the OWASP Top 10 list of most common attacks in 2013 [3]. In an XSS attack, the attacker manages to inject JavaScript code into websites that are considered secure [4, 5]. These injections can do anything from harmless pranks (e.g. showing an alert box) to redirecting the non-suspecting user to a fake website to steal valuable information. When an attack like XSS succeeds, the culprit is usually non-escaped input from the users. If a website includes the user input verbatim, an attacker can insert input that will be treated as code by the victim [5]. Examples of valuable information retained by the browser for an attacker are cookies and session tokens which can be used to gain access to e.g. personal information and passwords.

1.1 Problems with JavaScript

From a security standpoint with regards to information flow control, JavaScript suffers from two big problems, namely

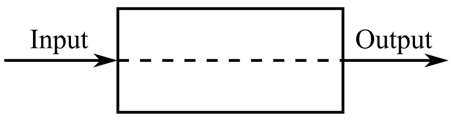
- It uses weak typing, i.e. it can do both implicit and explicit type conversions depending on the operation, often violating the principle of least surprise (e.g. `+` does not always mean addition of two numbers) and leading to hard to find bugs. This is explained more in depth in Chapter 2.1.1.

- It can gain access to sensitive information from the browser.

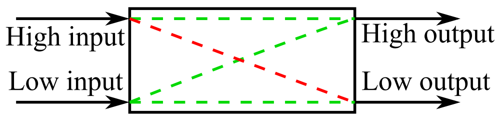
There are more problems with JavaScript in general, e.g. bad scoping semantics, poor support for the functional paradigm and a lack of modularity. Since those problems are not that important from a information flow security standpoint, it is not in the scope of this report. The curious reader can read more about those problems in [6].

1.2 Information Flow Control

Confidentiality in any system is important. Unfortunately, there are few built in mechanisms for ensuring end-to-end confidentiality policies in programming languages like JavaScript [7]. The general idea behind information flow control is to tag the data with one of two security levels; *high* or *low*. These levels can be seen as either private data (high security level) or public data (low security level). Figure 1.1a shows the flow in an application developed with the core language of JavaScript. In JavaScript, there is no way of controlling the flow by dividing the different values of the application into either a high or low context. It is more of a straight line from the input to the output. However, if an implementation of a system that enforces information flow control were in place, the flow could be seen as in Figure 1.1b. The only flow that is not allowed is a flow from a high context to a low context. All the other flows, i.e. low to low, low to high and high to high, are valid. A system that enforces information flow control will help ensuring end-to-end confidentiality.



(a) Normal information flow



(b) Controlled information flow

Figure 1.1: Different kinds of flow in a web application

2

Background

As mentioned in Chapter 1, JavaScript has several shortcomings. This chapter will explain those shortcomings in more detail. It will also introduce two of the tools that will be used in this project; Haste [6] and JSFlow [8, 9, 10]. Finally, some terminology and related work will be presented.

2.1 Problems with JavaScript - part deux

From a security standpoint JavaScript suffers from weak dynamic typing and having access to sensitive information in the browser.

2.1.1 Weak dynamic typing

There are several odd features one can use in JavaScript due to the weak dynamic typing. Where a statically typed language (e.g. Java) would give a compile error, the JavaScript code will be run and perhaps succeed but with unexpected results. Figure 2.1 shows a logical error that is caught at compile time in Java. If the code in Figure 2.1 would be allowed to run, it would be up to the runtime environment to determine how the code would be interpreted. The same error in JavaScript is shown in Figure 2.2. Since JavaScript does not have any static type checking the decision on how to interpret the code will be made at runtime. Instead of failing and raising a runtime error, JavaScript will convert the **true** value into a number which in this case corresponds to the number **1**. Hence the addition will be evaluated to **3**. Situations like that, where a clear type error is allowed to propagate and potentially change the state of the application should be considered dangerous and prone to producing bugs.

Another issue with the weak type system in JavaScript can be seen in Figure 2.3. It is perfectly legal to compare a function with an array in JavaScript. The code in Figure 2.3 will evaluate to **true**. This can in turn make it very difficult to find potential bugs since JavaScript will

```
int a = 2;
boolean b = true;
a + b;  // This should not be allowed to be run
```

Figure 2.1: Logical error in Java

```
var a = 2;
var b = true;
a + b;  // This will evaluate to 3
```

Figure 2.2: Logical error in JavaScript

try to convert values of different types to the same types and then make the comparison. In e.g. Java, the comparison in Figure 2.3 would not go through the type checker.

```
(function(x) { return x * x; }) > [1,2,3];
```

Figure 2.3: Weird comparison in JavaScript

In principle, weak typing is when a programmer can mix different types. In some cases it can make sense to allow it, e.g. adding an integer and a float value. It is not only in JavaScript that the examples in Figure 2.2 and Figure 2.3 would go through a "compilation phase". Other dynamic languages such as Erlang, Python and Ruby would allow those examples through the compiler. However, Erlang, Python and Ruby all have type checking at runtime and those examples would generate an error at runtime.

One could argue that it is rather silly examples. Who would ever compare a function with an array? The example in Figure 2.4 shows exactly why weak typing is a bad thing. Assume that a user inputs **Alice** as the name and **42** as the age. The example in Figure 2.4 would gladly write

"Alice is now 42 years old. In 20 years Alice will be 4220 years old!"

to the console. This is because `+` is not only addition, it is also concatenation and if one of the operands is a string, `+` will always convert the second operand to a string and do a concatenation.

2.1.1.1 Attempts to solve the type problem

There have been several attempts of providing a more secure type system to JavaScript, everything from creating a statically typed language that

```
var name = prompt("What is your name?", "");
var age = prompt("What is your age?", "");
console.log(
    name + " is now " + age + " years old. In 20 years " +
    name + " will be " + (age + 20) + " years old!"
);
```

Figure 2.4: Weak types in JavaScript

compiles to JavaScript to a compiler from an already existing programming language and compile it to JavaScript to a static type checker. Examples of attempted soultions are

- **TypeScript** [11], a typed superset of JavaScript that compiles to plain JavaScript.
- **TeJaS** [12, 13], which allows you to annotate type signatures as comments in the JavaScript code and then type checks the code.
- **GHCJS** [14] and **Haste** [6, 15], which compiles from Haskell, a statically typed, high-level functional programming language [16], to JavaScript.

2.1.2 Sensitive information in the browser

The browser has access to several different types of sensitive information and can be used by an attacker to get the sensitive information. JavaScript can gain access to e.g. cookies, send HTTP requests and make arbitrary DOM (*Document Object Model*) modifications. If untrusted JavaScript is executed in a victim's browser, the attacker can, among other things, perform the following attacks:

- **Cookie theft**, where the attacker can gain access to the victim's cookies that are associated with the current website.
- **Keylogging**, where the attacker can create and register a keyboard event listener and send the keystrokes to the attacker's own server in order to potentially record passwords, credit card information etc.
- **Phishing**, where the attacker can insert fake forms by manipulating the DOM and fool the user to submit sensitive information which will be redirected to the attacker.

2.1.2.1 Attempts to solve the information problem

The current solution to secure the sensitive information JavaScript has access to as of now is to sandbox the script and run it in a secure environment. There are mainly three ways of doing this.

- Wrap the script inside a call to `with` and pass a faked `window` object and execute the code with `eval`.
- Use an iframe and set the sandbox attribute to either not allow scripts to run or allow the scripts to run within that iframe only. Unfortunately, as explained in [17], iframes have some issues as well.
- Use existing tools to help sandbox third party code, such as:
  - **JavaScript in JavaScript** [17], an interpreter that allows an application to execute third-party scripts in a completely isolated, sandboxed environment.
  - **Caja** [18], a compiler to make third party code safe to embed within a website.

2.2 Haste

Haste (*HASkell To Ecmascript compiler*) is a compiler that compiles the high-level language Haskell to JavaScript. The Haste compiler is plugged into the compilation process from GHC, the *Glasgow Haskell Compiler*. As can be seen in Figure 2.5, Haste starts its compilation process after GHC has done some code optimization. From the optimized code from GHC, Haste will create an AST, *Abstract Syntax Tree*, for JavaScript. The AST will then be optimized and after the optimization process in Haste is done the actual JavaScript code will be generated.

When compiling the Haskell source code with Haste, the compilation process will result in a JavaScript file. If *Haste.App*, a client-server communication framework for Haste, is used, the compilation process will create the source code for the client in JavaScript and a server binary [15].

2.3 JSFlow

JSFlow is an interpreter written in JavaScript that dynamically checks the JavaScript code at runtime to ensure information flow security. Currently, JSFlow supports full information flow control for *ECMA-262 v5*, the standard which JavaScript is built upon, apart from *strict mode* and JSON (*JavaScript Object Notation*).

Within information flow security, there are two types of flow that must be checked - *explicit flow* and *implicit flow*. Even though there are several

Step	Operation	GHC/Haste
1	Parse	GHC
2	Type check	GHC
3	Desugar	GHC
4	Intermediate code generation	GHC
5	Optimization	GHC
6	Intermediate code generation (JS AST)	Haste
7	Optimization	Haste
8	Code generation to JavaScript	Haste

Figure 2.5: The compilation process for the Haste compiler

different ways an attacker can gain information about a system (e.g. via timing attacks where the attacker analyzes the computation time to gain information about the system), only explicit and implicit flows for computations are considered. JSFlow does not provide security for e.g. timing attacks and due to this, handling timing attacks and other side-channel attacks will be outside of the scope for this thesis.

2.3.1 Explicit flow

With explicit flow, one means when a data in a *high* context leaks information to a *low* context explicitly. An example can be seen in Figure 2.6a where the value of the high variable `h` is leaked to the low variable `l`. Obviously this should be illegal when information flow security is applied and explicit flows are not difficult to find when dynamically checking the code. When data is written to a variable, one simply must keep track of the context of the variable and the context of the data. If the variable is in low context and the data is in high context an error should be produced and execution of the JavaScript code should be stopped. All other scenarios (high variable with high data, high variable with low data and low variable with low data) are allowed.

2.3.2 Implicit flow

Implicit flows occurs when e.g. a language's control structure is used in combination with side effects to leak information. Figure 2.6b shows an example of an implicit flow. The variable `l` will get the value **1** if and only if the variable `h` is an odd number. Otherwise `l` will have the value **0**. A dynamic system that will handle implicit flows must associate a security context with the control flow [9]. In Figure 2.6b, the body of the if statement should be executed in a secure context and therefore the variable `l` must be a secure variable in order for the flow to be valid.

```
l := h
```

(a) Explicit flow

```
h := h mod 2;
l := 0;
if h = 1 then l := 1;
else skip;
```

(b) Implicit flow

Figure 2.6: Implicit and explicit flow

2.3.3 Example of coding for JSFlow

When creating a web application in JavaScript that JSFlow should be able to check, there is only one function that the programmer must know about - the `upg` function. The function `upg` is used when lifting a computation into a high context. In Figure 2.7, the variable `h` is lifted into a high context by calling `upg` on the data to be stored to `h`. Due to the call to `upg`, the variable `h` will be a high variable containing the number **42**. As can be seen with the variables `l`, which is a low variable, and `t`, which is a high variable, the default level for a compuation is low unless JSFlow infers that a variable **must** be high due to part of the compuation being high.

```
// Variable l is a low variable of value 2
var l = 2;

// Variable h is a high variable of value 42
var h = upg(42);

/* Variable t must be a high variable due to h
   being high */
var t = l + h;
```

Figure 2.7: Creating high and low variables in JavaScript with JSFlow

The `upg` function will take a computation (in Figure 2.7 the compuation is simply the number 42) and bind that value to the assigned variable (in Figure 2.7 the variable `h`) and put the variable in a high context.

2.3.4 Flows in a pure functional language

Even though there are two different flows in JavaScript (implicit and explicit flows), there is only one type of flow in a pure functional programming language like Haskell, an explicit flow [19]. As described in Chapter 2.3.2, an implicit flow depends on control structures in combination with side effects in order to leak information. However, even though a pure functional programming language like Haskell contains control structures, a pure function does not contain side effects. A control structure like an if-statement can be interpreted as a regular function returning a constant value. This means that a function like

```
f :: HInt -> LInt
f x = if x ‘mod‘ 2 == 0
          then 42
        else -42
```

will look like an implicit flow but is in fact an explicit flow. Note that, in this case `HInt` stands for a *High Int* and `LInt` stands for a *Low Int*. An if-then-else can be rewritten as a regular function [20]

```
myIf :: Bool -> a -> a -> a
myIf True  b1 _ = b1
myIf False _ b2 = b2
```

where `b1` and `b2` are the different branches. The different branches are of type `a`, which in this case can be any arbitrary expression. Rewriting the function `f` using `myIf` can be done as follows:

```
f :: HInt -> LInt
f x = myIf (x ‘mod‘ 2 == 0) 42 (-42)
```

where a constant value, either **42** or **-42**, is returned. Since there are no side effects in a pure function like `f`, there can be no implicit flows. Every flow will be explicit, which in turn makes it easier to create a structure in the type system that keeps track of the flow.

2.4 Non-interference

The principles of non-interference were introduced by Goguen and Meseguer in 1982 [21] who defined non-interference to be

*”one group of users, using a certain
set of commands, is noninterfering
with another group of users if what*

the first group does with those
commands has no effect on what the
second group of users can see."

When talking about non-interference with regards to information flow control, one means a property that states that the public outcome does not depend on any private input. An attacker should not be able to distinguish between two computations from their outputs if the computations only vary in the secret input.

As an example of non-interference, assume the following function where something of type `Char` is of high value and something of type `Int` is of low value:

```
fOk :: (Char, Int) -> (Char, Int)
fOk (c, i) = (chr (ord c + i), i + 42)
```

The function `fOk` preserves confidentiality since it does not leak any valuable information about the value `c`. In this case, preserving the confidentiality of `c` means that no information of `c` is leaked. It is said to be *non-interfered* since the public result (the `Int` value) is independent of the value of `c`. If the function instead was defined as

```
fBad1 :: (Char, Int) -> (Char, Int)
fBad1 (c, i) = (c, ord c)
```

it would not be non-interfered because the confidentiality is broken. Information about `c` is leaked through the low `Int` and the corresponding decimal value of the ASCII number of `c` is returned as the second value of the tuple.

Unfortunately, information leakage is seldom as explicit as in the `fBad1` function. An attacker might be clever and attempt

```
fBad2 :: (Char, Int) -> (Char, Int)
fBad2 (c, i) = (c, if ord c > 31 then 1 else 0)
```

which would give information about whether or not the variable `c` is a printable character (the ASCII values of printable characters start at 32 [22]) [19]. Just as with `fBad1`, `fBad2` does not satisfy the non-interference property.

2.5 Declassification

A system that satisfies the non-interference policy is a very strict system. A system usually needs some kind of controlled release of confidential data. As an example, imagine a login system. A user's password should be handled as confidential data when the user attempts to login. If the login succeeded the user should be authenticated and redirected whereas if the login failed the user should be prompted with a message saying the username/password combination was incorrect. If the system was non-interfered, there could be no message explaining to the user that the username/password was incorrect since that output would rely on confidential data. Another example could be when a credit card is being used for online payment. In the order receipt it is not uncommon to include the last four digits of the credit card number. Again, this is something that can not be done if the system is non-interfered since a credit card number should be considered confidential.

Unfortunately, there is no way for the non-interference policy to distinguish between intended release of information and release that occurs due to an attack or programming error. In order to allow controlled information leak one can use declassification policies. Taking the example of the credit card explained above and assuming a function `getLastFour` that takes a secret credit card number `h` as an argument and returns a secret value containing the last four digits of the credit card number, producing an intended information release to the variable `l` can be achieved by calling a `declassify` function:

```
l := declassify(getLastFour(h))
```

However, just allowing declassification everywhere can be dangerous. In theory, an attacker could compromise the declassification and extract more information than intended. Due to this, work on classifying the declassification into four different dimensions has been presented in [23]. The proposed dimensions are

- **What** information is released.
- **Who** controls the information release.
- **Where** in the system the information is released from.
- **When** the information is released.

and these should be seen as recommendations for how to build the declassification policies in systems.

2.6 Related Work

There has been research within information flow control and libraries have been created in order to help enforce information flow policies and secure both confidentiality and integrity of the information. Some of the most relevant findings for this project will be described below.

2.6.1 Labeled IO

Labeled IO (*LIO*) is a library created in Haskell for dynamic information flow control [24]. Compared to the library created for this thesis, LIO keeps track of a *current label*, which is an upper bound of the labels of all data that can be observed or modified in the evaluation context. One can also bound the current label with a *current clearance*, where the clearance of a region of code can be set in advance to provide an upper bound of the current label within that region. LIO attempts to close the gap with static analysis, which even though it has its advantages (fewer run-time failures, reduced run-time overhead etc.) has a problem when new kinds of data (e.g. user input) can be encountered at runtime, and dynamic systems.

2.6.2 Seclib

Seclib is a light-weight library for information flow control in Haskell [19, 25]. Just as with the library created in this thesis, Seclib is based on monads and all private data lives within the created monad. However, compared to the library in this thesis, Seclib creates two different monads, *Sec* and *SecIO*, where SecIO is an extended IO monad (an IO monad wrapped in a Sec monad). Seclib was designed to be a small, lightweight library and consists, as of Jan. 29 2015, of only 342 lines of code.

2.6.3 LMonad

LMonad [26] is a library tailored to provide information flow control for web applications written in Yesod, a framework for creating web applications in Haskell [27]. It is a generalization of LIO and provides a Monad Transformer [28], which in theory means it can be "wrapped around" any monad. Programmers can define their own information flow control policies and after defining them, LMonad guarantees that database interactions follow the policies.

2.6.4 RDR

RDR is a monad for Restricted Delegation and Revocation which builds on the DLM (*Decentralized Label Model*) to provide information flow control. DLM allows declassification in a decentralized way [29] and consists of four important parts [30]:

- *Principles*, which is an entity with power to change and observe certain aspects of the system. A principle *p* can delegate authority to a principle *q* and the actions taken by *q* is implicitly assumed to be authorized by *p*. The principals can have different security policies which makes the system modular.

- *Reader policy*, which allows the owner of the policy to specify which principals who are allowed to read a given piece of information. This is a *confidentiality policy*.

- *Writer policy*, which allows the owner of the policy to specify which principals who may have influenced the value of a given piece of information. This is an *integrity policy.*
- *Labels*, which is a pair of a confidentiality policy and an integrity policy. It is the labels that are used to anotate programs.

The purpose of RDR is to extend restricted delegation and revocation to information flow control. The main principle is to allow information flow to a predefined chain of principals, but keeping a right to revoke it at any time. The RDR monad is built by using a Reader monad over the IO monad. One of the ideas of RDR is to see restricted delegation as declassification and protected values should only be allowed to be read by using a valid principal and due to the DLM, mutually-distrusting principals should be allowed to express individual confidentiality and integrity policies [31].

2.6.5 Why another library?

As explained above, there are already working libraries for information flow control so why would another library be needed? Even though the current libraries are good, they lack one part - integration with a dynamic system. The library will help close the gap between static control for information flow control and already created tools for dynamic check of information flow control (JSFlow) and created tools for securing JavaScript from the viewpoint of a type system (Haste).

3

Goal and limitations

This chapter will describe the goals and limitations of the thesis.

3.1 Goal

The goal of the thesis is to create a library that will help close the gap between static and dynamic checks for information flow control. In particular, this will be done by extending Haste to produce JavaScript code targeting JSFlow.

3.2 Limitations

The following areas will not be considered in this thesis:

- DOM support.
- Support for Haste.App, which will include HTML5 with Websockets.
- Dynamic information flow control.
- Side-channel attacks towards a system (e.g. timing attack).
- Securing any communication outside of the JavaScript that is to be generated by Haste.
- More than two security levels.

4

Prototypes

For this project, three different prototypes were developed. There are not many differences between the non-monadic and the monadic version and for the final implementation the monadic version was chosen.

4.1 Embedded Domain Specific Language

The first prototype developed was a deeply embedded domain specific language (*EDSL*) [32]. In theory, an EDSL is a embedded language within another language. In this case, the library would be a EDSL within Haskell. In a deep EDSL, an abstract syntax tree is created to represent a program while in a shallow EDSL, language constructs are mapped directly to their semantics [33].

In the prototype, the different operations allowed were translated to constructors of a data type called `Expr`, where `Expr` represented expressions. Even though an EDSL is very powerful it suffers from a major drawback. It is difficult to gain access to features from the main language. An example is recursion, something that exists in Haskell, that needs to be implemented again within the EDSL (usually as a constructor which needs to be interpreted). Due to the constraint of not being able to use language-specific features without implementing them again and the increased overhead of defining every action that should be allowed within a datatype, this prototype was quickly disposed of.

4.2 Non-monadic

In the non-monadic prototype, the first thing that needed to be tested was different implementations for a type that defines a flow. The first version can be seen in Figure 4.1 and contained a *State monad* from a tuple containing a *tag* and a value to an *internal state*. The internal state were to keep track of the program counter (`pc`) and all active scopes (`activeScopes`). By keeping track of an internal state and the program counter, one could check if a computation leaks information or not. This has several drawbacks. First, it is complex; a program counter and keeping track of the internal state are not needed for a statical analysis - it can be done within the type system of Haskell. Second, keeping track of every variable is very tedious from the library that is to be created since Haste will generate variable names when the application is compiled. Having a dynamic representation of variable names within the created library will therefore not be necessary.

```
type Ident = String
type Scope a = Map Ident (Tag, Term a)

data Tag = High | Low

data InternalState a = InState { pc :: Tag
                               , activeScopes :: [Scope a]
                               }

newtype Flow level a = Flow (State (level, a) (InternalState a))

data LType a where
  LInt    :: Tag -> Int    -> LType (Tag, Int)
  LString :: Tag -> String -> LType (Tag, String)
  LBool   :: Tag -> Bool   -> LType (Tag, Bool)

data Term a where
  TType :: Term (LType a)
  TApp  :: Term a -> Term a
```

Figure 4.1: First attempt at a Flow type

The prototype shown in Figure 4.1 was revised to what is shown in Figure 4.2. As in Figure 4.1, it contained a State monad but this time it was from a tuple containing a tag and a value to a list of tuples of tags and values. In principle, a state is used to keep track of the different data values (i.e. computations) and their respective tags. This would then be used to ensure that no information was leaked. Since Haste makes a lot of computations that is encapsulated within the IO monad, it was later decided to have the Flow type use the IO monad instead of the State monad. How Flow was implemented can be seen in Chapter 5.1.

In order to use the Flow type properly, several help functions were imple-

```
newtype Flow tag a = Flow (State (tag, a) [(tag, a)])
```

Figure 4.2: Second attempt at a Flow type

mented. It turned out that one of the help functions was similar to the *bind operator* for monads while other help functions were similar to the functions in the *Applicative* and *Functor* instances. With that in mind, the logical step was to attempt to write a monadic version of the Flow type.

4.3 Monadic

The monadic prototype was a continuation of the non-monadic version explained in the previous section. The only difference was that a *Monad instance* was implemented for the Flow type, making it possible to combine an action (i.e. something of the Monad type) and a reaction (i.e. a function from a computation of the action to another action). In short, this is the bind operator, `(>>=)`. It also makes it easy to encapsulate a value into a monadic value. The type signatures of these actions can be seen in Figure 4.3. After implementing the monadic operations and deciding on the monadic version, the work was continued by implementing Applicative and Functor instances. The continuation of this work is described in Chapter 5.

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

Figure 4.3: Type signatures for the mandatory functions for Monadic instance

5

Implementation

The implementation of the library (hereby referred to as *SwapIFC*) consists of several modules which a user is allowed to use in order to create programs which are secure with regards to information flow control. The standard implementation of SwapIFC consists of a type *Flow*, which is a flow in the program (i.e. data with a high or low tag). It also provides the user with standard operations, e.g. adding two flows if the flows are of such a type where addition is valid.

Currently, SwapIFC has a representation of all the standard operations within the following Haskell types:

- Num
- Frac
- Bool
- Eq
- Ord

The Monad, Functor and Applicative instances for the high and low flow can be seen in Appendix B and an example of how the standard operations for the Num type in Haskell can be found in Appendix C. An overview of the structure of SwapIFC can be seen in Appendix A.

5.1 The Flow type

The Flow type is defined as a *newtype* in Haskell, i.e. a definition that can only contain one constructor. The Flow type became

```
newtype Flow tag a = Flow (IO a)
```

and even though it has a tag in the definition (the variable `tag` in the left hand side), it does not contain a tag on the right hand side. Instead, the tag is implemented as a *phantom type* [34]: a type which does not appear on the right hand side of a type definition and so has no representation on the value level. A phantom type can be checked by the type checker but it can not be used by the user via e.g. pattern-matching. This means that from a value standpoint, the Flow type is simply a container for IO actions and the rules for the information flow control are implemented in the type checker alone. However, from a user standpoint, the Flow type is a tagged computation. An example of how to add five to a Flow of high value is

```
addFiveFlow :: Flow High Int -> Flow High Int
addFiveFlow f = do
  let b = mkHigh 5
  f .+. b
```

where a new Flow (the variable `b`) is created and added with the given Flow `f`.

5.1.1 Monad instance

Two separate monad instances were created in order to be able to handle potential code generation towards Haste, one for high and one for low flow. If Haste was the compiler, then the high flow needed to be compiled with a call to the `upg` function in JSFlow.

Every Monad created in Haskell must also be a Functor and an Applicative by the Functor-Applicative-Monad proposal. Starting from GHC 7.10, every Monad that is not a Functor or Applicative will generate a compile error [35]. Due to the Functor-Applicative-Monad proposal, the created Monad for the Flow type is a Functor and an Applicative as well.

5.1.1.1 Monad laws and Flow

If the Flow type is implemented correctly as a monad, it should satisfy the following three laws [36]:

```
1. return a >>= k  ==  k a
2. m >>= return  ==  m
3. m >>= (λx -> k x >>= h)  ==  (m >>= k) >>= h
```

The first monadic law is easy to prove. It can be done in two steps:

```
1. return a >>= k = Flow (IO a) >>= k
2.                = k a
```

where the transformation on line 1 is due to the definition of `return` in SwapIFC and the transformation on line 2 is due to the definition of (`>>=`) in SwapIFC.

The second law is just as easily shown.

```
1. m >>= return = Flow (IO a) >>= return
2.              = return a
3.              = Flow (IO a)
```

The transformation between lines 1 and 2 is due to the definition of (`>>=`) and the transformation on line 3 is due to the definition of `return`.

To show that the third law is satisfied, it is enough to show that the left hand side and the right hand side evaluate to the same value. The left hand side is evaluated as:

```
1. m >>= (λx -> k x >>= h) = Flow (IO a) >>= (λx -> k x >>= h)
2.                         = (λx -> k x >>= h) a
3.                         = k a >>= h
```

where the transformation between lines 1 and 2 is due to the definition of (`>>=`). It is simple lambda-calculus to apply the value `a` to the function `λx -> k x >>= h` on line two and it is through this the result on line 3 is derived.

Similarly, deriving the right hand side will yield

```
1. (m >>= k) >>= h = (Flow (IO a) >>= k) >>= h
2.                 = k a >>= h
```

where the transformation between lines 1 and 2 is due to the definition of (`>>=`). As can be seen, the left hand side yields the same result as the right hand side and the two sides are therefore equal.

At first glance, the third law can be non-intuitive. However, due to the type signature of (`>>=`), `m` must be of type `Monad m => m a`, `k x` must be a monadic value (which indicates that the function `k` must have the type signature `(Monad m => a -> m b)` and `h` must have type `(Monad m => b -> m c)`. Due to this, it should be allowed to have `m >>= k` since that would mean `(Monad m => m a >>= (a -> m b))` which would type check. The result of `m >>= k` is of type `Monad m => m b` and since `h` has type signature `(Monad m => b -> m c)`, it is valid to have `(m >>= k) >>= h`.

5.1.2 Functor instance

In order for the Flow type to be an instance of Functor, the function `fmap` must be implemented. When defining a functor, one can see it as transforming a pure function into a monadic function (if the functor is also a monad). If a type is an instance of both the Monad and Functor classes, then the implementation of `fmap` must obey the following law:

```
fmap :: Functor f => (a -> b) -> f a -> f b
fmap f ioa = ioa >>= return . f
```

which uses the (`>>=`) and the `return` functions of the monad instance [36]. Since the Flow type should not be allowed to be transformed into something that is not a monad, this is the implementation of `fmap` SwapIFC will use.

5.1.2.1 Functor laws and Flow

In order for the Flow type to have a proper implementation of `fmap`, there are two laws that must be obeyed [37], namely

```
1.  fmap id  ==  id
2.  fmap (f . g)  ==  fmap f . fmap g
```

In order to see that the functor implementation satisfies the first law, it can be simplified:

```
1. fmap id = λx -> x >>= return . id
2.         = λx -> x >>= (return . id)
3.         = λx -> x >>= return
4.         = λx -> x
5.         = id
```

where the transformations on line 2 is due to precedence of function composition, line 3 is due to the fact that (`return . id`) has the type (`Monad m => a -> m a`). Line 4 uses the second monadic law and line 5 uses the fact that a function that takes one argument and does nothing but returning that argument is by definition the `id` function.

For the second law, expanding the left hand side and the right hand side will show that the results are equal. The left hand side is expanded as

```
1. fmap (f . g) = λx -> x >>= return . (f . g)
2.              = λx -> x >>= return . f . g
```

where the first transformation on line 1 is due to the definition of `fmap` and the transformation on line 2 is due to the fact that `(.)` is right-associative.

The right hand side expansion is more interesting:

```
1. fmap f . fmap g = λx -> fmap f (fmap g x)
2.                 = λx -> (fmap g x) >>= return . f
3.                 = λx -> (x >>= return . g) >>= return . f
4.                 = λx -> x >>= (λy -> return (g y) >>= return . f)
5.                 = λx -> x >>= (λy -> (return . f) (g y))
6.                 = λx -> x >>= (λy -> return f (g y))
7.                 = λx -> x >>= (return . f . g)
8.                 = λx -> x >>= return . f . g
```

where the transformations on lines 1-3 are simply re-writing using the definition of `fmap`. The transformation between lines 3 and 4 is due to the monad associativity law (the third law shown in Chapter 5.1.1.1) and transformation on line 5 is due to the left associativity law (the first law shown in Chapter 5.1.1.1). On line 6, a simple application of the argument `g y` is performed and lines 7-8 are due to the fact that λ`x -> f (g x) == f . g`.

As can be seen, the functor instance implementation does satisfy the functor laws.

5.1.3 Applicative instance

The Applicative instance is implemented simply by using primitives from the Monad class by having

```
pure  = return
(<*>) = ap
```

In order for a instance of Applicative to be correctly implemented, the following laws must be satisfied:

```
1.  pure id <*> v = v
2.  pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
3.  pure f <*> pure x = pure (f x)
4.  u <*> pure y = pure ($ y) <*> u
```

The first law:

```
1. pure id <*> v = return id `ap` v
2.               = Flow id `ap` v
3.               = v
```

The first law is very straight-forward. It is basically the same as applying a value to the `id` function, which of course will yield the same value.

When proving the second law, the best course of action is to evaluate the two sides one by one and show they evaluate to the same expression. The left hand side of the second law can be evaluated as

```
1. pure (.) <*> u <*> v <*> w = return (.) `ap` u `ap` v `ap` w
2.                            = Flow (.) `ap` u `ap` v `ap` w
3.                            = Flow ((.) u) `ap` v `ap` w
4.                            = Flow ((.) u v)`ap` w
5.                            = Flow (u . v) `ap` w
6.                            = Flow (u (v w))
```

where the transformation on lines 1 and 2 are due to the definitions of `pure` and `<*>`. On lines 3 and 4, the values of `u` and `v` are applied to the `Flow`

`(.)`. Line 5 is a transformation that can be done since the function `(.)` can be written infixed. Finally, the value in `w` is applied.

The right hand side of the second law is evaluated as

```
1. u <*> (v <*> w) = u ‘ap‘ (v ‘ap‘ w)
2.                 = u ‘ap‘ Flow (v w)
3.                 = Flow (u (v w))
```

where the transformation on line 1 is due to the definition of `<*>` and the transformations on lines 2 and 3 are due to evaluation of the `ap` expressions.

The third law is not tricky but is it not straight-forward.

```
1. pure f <*> pure x = return f ‘ap‘ return x
2.                   = (Flow f) ‘ap‘ (Flow x)
3.                   = Flow (f x)
4.                   = return (f x)
```

The key is to note that `f` must be a function of type `(a -> b)` and `x` must be of type `a`. Due to this, it is trivial to make the transformations on line 2 (”wrapping” the values in the Flow monad) and applying `Flow x` to `Flow f` on line 3. On line 4, we go back from `Flow (f x)` to `return (f x)` since these are equivalent.

The fourth law is very straight-foward. Once again, in order to prove the fourth law the right hand side and the left hand side will be evaluated individually.

```
1. u <*> pure y = u ‘ap‘ return y
2.              = u ‘ap‘ (Flow y)
3.              = Flow (u y)
```

The first transformation on line 1 is due to the definitions of `<*>` and `pure` respectively. On line 2, we use the fact that `return` will ”wrap” the value of `y` into the Flow monad. Finally, on line 3 the function `ap` is evaluated.

Evaluating the right hand side for the fourth law is just as easilly done.

```
1. pure ($ y) <*> u = return ($ y) ‘ap‘ u
2.                  = Flow ($ y) ‘ap‘ u
3.                  = Flow (u y)
```

Again, the transformations are rather straight-forward. Line 1 is due to the definitions of `pure` and `<*>` and lines 2-3 are due to how the evaluation order is. Note that `($ y)` is a partially applied function where `($)` has type `(a -> b) -> a -> b`. Due to this, the partially applied function `($ y)` will have the type `(a -> b) -> b`.

There are two more laws for applicatives that state that if the applicative is also a monad, then the following properties must hold:

```
pure  = return
(<*>) = ap
```

and it is trivial to see that these properties do hold since it is the exact implementation of the Applicative instance for Flow.

5.2 Controlling the flow

Even though there are only two types of tags for the flow, they need to be thoroughly controlled. In order to model this, the language extensions *MultiParamTypeClasses* [38] and *FunctionalDependencies* [39] were used. Normally, a class instance in Haskell can only take one argument, but with the extension MultiParamTypeClasses, a created class can take several arguments. In the case for the class *FlowBool*, i.e. a class that contains all operations for the Haskell type Bool wrapped in the Flow type, it is created as

```
class FlowBool t1 t2 t3 | t1 t2 -> t3 where
  -- Function implementations of boolean operators
```

where `t1`, `t2` and `t3` are the arguments. As can be seen, given two tags, `t1` and `t2`, it is possible to derive the resulting tag `t3`. This functionality is due to the extension FunctionalDependencies. In order for Haskell to know how the different type variables relate to eachother, instance definitions must be added:

```
instance FlowBool High High High
instance FlowBool High Low High
instance FlowBool Low High High
instance FlowBool Low Low Low
```

The instance definitions give the rules for the Haskell type system which types the type variables can have and how to derive the resulting type. As can be seen, the only way for a `FlowBool` to produce something of low value is when the two type variables `t1` and `t2` are low. As soon as either `t1` or `t2` or both are high, the result will be high. This holds for all classes in SwapIFC.

5.2.1 Declassification of flows

When looking at the declassification dimensions mentioned in Chapter 2.5, SwapIFC implements the *who* and the *what* dimension. The only person who can declassify information is a person who has access to the *trusted* code base (see Appendix A) and the only data that can be declassified is data that is tagged as high value. This is due to the fact that declassifying a low value makes no sense and if it is done then the programmer might not know exactly what he/she is doing. Due to this, a compile error will be given if declassification of low data occur. However, there is no limit as to *when* a person can declassify data or *where* a person can declassify data from.

There is also no way to declassify data without using the declassification function in the trusted code base. Without the primitives `declassify` and `upgrade`, there should not be possible to use any operations within SwapIFC to accidently declassify or upgrade data. One could imagine a programmer that does not know exactly how to handle the data and accidently creates a bug. Could it be possible to downgrade a flow from high to low using `(>>=)`? The `(>>=)` operator has the following type signature:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Assume the programmer has a `Flow High` called `fHigh`, i.e.

```
fHigh :: Flow High
```

There exists a function `mkLow` that, given a value, creates a `Flow Low` instance with the given value, i.e.

```
mkLow :: a -> Flow Low a
```

Now, assume the programmer accidently tries to use `(>>=)` and `mkLow` to downgrade a `Flow High a` to a `Flow Low a`. He/she could try to create the following malicious function:

```
blooperFunction :: a -> Flow Low b
blooperFunction x = mkLow x
```

and attempt to combine `fHigh` with `blooperFunction`:

```
fHigh >>= blooperFunction
```

with the hopes of getting something of type `Flow Low a`. However, this will not be possible as the compiler will produce a compile error. This is understandable if one looks at how the type signature is for the innocent yet malicious expression:

```
(fHigh >>= blooperFunction) :: Flow High a
                               -> (a -> Flow Low b)
                               -> Flow ??? b
```

There are two different monad instances for the Flow type, one for high and one for low values and each has its own implementation of `(>>=)`. Since the first argument to `(>>=)` is of type `Flow High a`, the compiler can deduce that it must use the implementation of `(>>=)` that exists in the `Flow High` monad instance. That means that the second argument, the function, must be of type `(a -> Flow High b)`. However, the second argument in this accidental attempt is of type `a -> Flow Low b` and due to this, accidental declassification can not be done by using `(>>=)`. Note that this holds the other way around as well, i.e. there is no way to upgrade a value of type `Flow Low a` to `Flow High a` by using `(>>=)`. Instead the function `upgrade` must be used.

5.2.2 Non-interference of flows

Due to the functional dependencies and the instance definitions, using only the primitives and not having access to the trusted code base will lead to non-interference. However, non-interference can not be guaranteed within the trusted code base due to the declassification primitive and the unsafe operations (`unsafeShow` to show the value of a flow and `unwrapValue` that will strip the flow and IO monad and return the pure value). The tests described in Chapter 6 shows that the confidentiality level of a flow is indeed preserved when using the primitives in SwapIFC.

5.3 Handling side effects

Handling side effects in any language is a challenge from a security point of view. Potential shared states can be altered, data can be printed or exceptions could be thrown indicating an error has occurred. From the viewpoint of information flow control, all of these side effects must be monitored closely.

5.3.1 JSFlow and side effects

In order to handle side effects, JSFlow keeps track of a *program counter label (pc)*. The program counter reflects the confidentiality level for guard expressions controlling branches (e.g. an if-statement) in the program and prevents the modification of less confidential values. This is exactly how the implicit flows are handled in JSFlow as well. More formal, it must hold that:

$$\forall ge \in GuardExpressions : ge_c \geq pc$$

where *GuardExpressions* is the set of expressions within the specified guard and ge_c is the confidentiality level for the expression *ge*. If any violations occur, the execution will halt. In short, given two tags, the pc's tag *t* and the computation's tag *t'*, the following holds:

```
t == High, t’ == High ==> Valid
t == High, t’ == Low  ==> Not Valid
t == Low,  t’ == High ==> Valid
t == Low,  t’ == Low  ==> Valid
```

5.3.2 SwapIFC and side effects

Side effects in SwapIFC is implemented by creating a newtype called *FlowRef*, which is a wrapper around *IORef*. The FlowRef takes a tag and a value and, just as the Flow type, wraps the value around a standard Haskell structure, in this case an IORef. The tag for the FlowRef is a phantom type, just as for the Flow type.

The FlowRef is implemented as

```
-- newtype for IORefs within Flow
newtype FlowRef tag a = FlowRef (IORef a)

newFlowRef :: a -> Flow t (FlowRef t a)
readFlowRef :: FlowRef t a -> Flow t a
writeFlowRef :: FlowRef t a -> a -> Flow t ()
modifyFlowRef :: FlowRef t a -> (a -> a) -> Flow t ()
```

and follows the exact same pattern as IORef does in the standard Haskell library. When creating a new FlowRef, the function `newFlowRef` must be called. Since `newFlowRef` returns a FlowRef within a Flow computation and the tag `t` is the same, then it is guaranteed that the side effect contained in the FlowRef can only be modified in the same context as the Flow it is wrapped inside. The function `readFlowRef` will read the given FlowRef and wraps the value in a Flow computation and the function `writeFlowRef` will, given a FlowRef and a value, write the given to the given FlowRef and wrap the result in a Flow computation. Since the base of FlowRef is IORef, the standard function `writeIORef` will be used internally and due to this, a unit-type is used in the Flow computation. Finally, `modifyFlowRef` will given a FlowRef and a function apply the function to the value inside the FlowRef. Again, the return type will be a Flow computation of unit-type.

Comparing the strictness of the handling of side effects between JSFlow and SwapIFC, one can see that SwapIFC is indeed stricter. Where JSFlow demands the computations to be in **at least** as high a context as the pc, SwapIFC demands the computations to be in **exactly** the same context. Given two tags, the current flow's tag *t* and the FlowRef's tag *t'*, the following holds for SwapIFC:

```
t == High, t' == High ==> Valid
t == High, t' == Low  ==> Not Valid
t == Low,  t' == High ==> Not valid
t == Low,  t' == Low  ==> Valid
```

The function that will run the flow has the following implementation:

```
runFlow :: Flow t () -> IO ()
runFlow (Flow ioa) = do
```

```
   res <- try ioa
   case res of
     Left err -> let e = err :: SomeException
                 in return ()
     Right () -> return ()
```

When running the Flow computation, the IO computation will be evaluated and as can be seen, the implementation uses Haskell's unit-type. Due to this fact, there is no way to extract any actual value. The only thing that can be done is to run the IO computation. Information will not leak in the current implementation even when exceptions occur. This means that a person with malicious intent will not notice any difference when executing a program that throws an exception and a program which does not throw an exception.

5.4 Integrating with Haste

In order to integrate SwapIFC with Haste, a communication between SwapIFC and Haste had to be created. This is done by using a preprocessing directive, the same preprocessing directive that Haste will set if the compilation process is done with Haste. The key difference for how SwapIFC should behave between GHC and Haste is the `return` of the high flow within the Monad instance. The main difference is highlighted in the following code snippet:

```
#ifdef __HASTE__
  return = Flow . upg
#else
  return = Flow . return
#endif
```

If the program is compiled with Haste, then the computation given to return will be executed within the function `upg`

```
upg :: a -> IO a
upg = fmap fromOpaque . ffi "upg" . toOpaque
```

which makes a FFI (*Foreign Function Interface*) call in order to wrap the compiled JavaScript code with a call to the JSFlow function `upg`. It is made opaque in order to avoid any conversions: the value is sent to JavaScript as it is and is returned as it was rather than being converted to something more JavaScript friendly. This process does not have to be done with a low flow type since it should be executed as normal code by JSFlow. The full implementation of the Monad instance can be seen in Appendix B.

The same principle as with `upg` was used to implement `lprint`, a function that prints the value and the corresponding tag in JSFlow. The function in SwapIFC which communicates with JSFlow was implemented as

```
lprintHaste :: Show a => a -> IO ()
lprintHaste = ffi "lprint" . show
```

Unfortunately, this solution combined with Haste will produce a corner case in JavaScript. Haste has some static runtime functions. One of these functions is a function which applies arguments to a function (call it `A` for apply). This in turn "builds" functions by function application. It takes a function (call it `f`) and its arguments (call them `args`) as argument and applies `f` to `args`. Note that `A` is only called if `f` is either over-saturated (i.e. applied to too many arguments), under-saturated (i.e. applied to too few arguments) or if the arity of `f` can not be defined. If `f` is under-saturated, `A` will return a closure waiting for the rest of the arguments. If `f` is over-saturated it will be applied with the expected arguments and the result will be applied to the rest of the arguments. Otherwise `f` will be fully applied to `args`. However, if the arity can not be defined, it will get the value of the `length` field. Arguments to a function in JavaScript are optional to define [40] and even though they are not defined they can still be reached. In the following example

```
function g() {
    lprint(arguments[0]);
}
```

the function `g` takes an argument but it is not defined in the parameter list. But every function in JavaScript have an argument object assiciated to it [41], so calling `g(42);` will indeed work since the value **42** will be at `arguments[0]`. But checking the length field by calling `g.length` will yield **0**. This will cause a problem since the code generated from Haste will check

```
if(f.arity === undefined) {
    f.arity = f.length;
}
```

but if the function `f` does not have variables in the arguments list, `f.length` will be **0**.

The problem with printing using `lprint` is solved by supplying an anonymous function through the FFI like below:

```
lprintHaste :: Show a => a -> IO ()
lprintHaste = ffi "(function(x) { lprint(x); })" . show
```

where the anonymous function has a variable in the arguments list. This means that the length of the function will be **1** and the problem explained above will not occur.

5.4.1 Modifications done to Haste

Since JSFlow only supports ECMA-262 v5 in non-strict mode and does not support JSON [8], some modifications to Haste were needed in order to produce JavaScript source code which followed the ECMA-262 v5 standard [42]. The code generation of Haste needed to be altered in order to not produce code for the JSON library and ECMA features that are introduced in ECMA-262 v6. Haste allows for ArrayBuffers to be produced, but since the concept of ArrayBuffer is in draft mode for ECMA-262 v6 [43, 44] they can not be supported by the current version of JSFlow. However, since crippling every user of Haste by removing features is not a good idea, a flag was added in case a user wants to compile towards JSFlow. This means that Haste will produce JavaScript source code with features that are introduced in ECMA-262 v6 unless the user tells the compiler to produce source code that can be dynamically checked by JSFlow.

6

Testing of the implementation

The testing part of the project was done in two different ways, namely with unit tests and randomized tests. While the unit tests are manually written tests, the random tests for SwapIFC were generated by using QuickCheck [45, 46].

6.1 "Naïve" testing of the primitives

One does not need to write unit tests or randomized tests in order to test the primitives in SwapIFC to validate that no information leak can occur. All that is needed is to write functions that captures every potential scenario. For the `(.+.)` primitive of two flows, the valid flows can be seen in Figure 6.1. However, if one attempted any of the following flows that is present in Figure 6.2, it would yield a compile time error due to the typeclass `FlowNum` not being defined for those cases. `FlowNum` is only defined to allow information flow like the ones in Figure 6.1. This holds for every primitive in SwapIFC. This is of course not a feasible solution and because of this, unit testing and random testing with QuickCheck was done.

6.2 Unit testing

A slightly more clever way of testing compared to the naïve testing described above is unit testing. The main principle of the unit testing is to isolate parts of a program or library and show its correctness by conducting tests on the specific part. For SwapIFC, the unit testing consisted of 202 tests which tested every primitive of SwapIFC. Given that all 202 tests yielded a correct result, the probability that the primitives behave correctly is considered high. However, unit testing is manual testing and it can be difficult to find

```
testFNumLL :: Num a => Flow Low a -> Flow Low a -> Flow Low a
testFNumLL = (.+.)

testFNumLH :: Num a => Flow Low a -> Flow High a -> Flow High a
testFNumLH = (.+.)

testFNumHL :: Num a => Flow High a -> Flow Low a -> Flow High a
testFNumHL = (.+.)

testFNumHH :: Num a => Flow High a -> Flow High a -> Flow High a
testFNumHH = (.+.)
```

Figure 6.1: Valid flows for add primitive

```
testFNumLL' :: Num a => Flow Low a -> Flow Low a -> Flow High a
testFNumLL' = (.+.)

testFNumLH' :: Num a => Flow Low a -> Flow High a -> Flow Low a
testFNumLH' = (.+.)

testFNumHL' :: Num a => Flow High a -> Flow Low a -> Flow Low a
testFNumHL' = (.+.)

testFNumHH' :: Num a => Flow High a -> Flow High a -> Flow Low a
testFNumHH' = (.+.)
```

Figure 6.2: Invalid flows for add primitive

edge cases and because of this, random tests using QuickCheck were also conducted.

6.3 Testing SwapIFC with QuickCheck

In order to test SwapIFC with QuickCheck, every operator for the different Flow instances needed to be modeled as a constructor of a datatype. The probability distribution for each operator within an instance is uniformed. The general algorithm for the `FlowNum` instance tests can be seen in Figure 6.3. Note that the *oneof* function is a function that will pick one of the elements in the list at a uniform distribution and the *randomInteger* function simply generates a random integer. The *validate* function will return true if and only if the result of the operand *op* applied to the flows *flow1, flow2* will generate the same value as the normal operand corresponding to *op* applied to the normal values corresponding to the flows, *val1, val2* and if the tag is being preserved. The preservation of the tag means that if any of the operands have a high tag, then the result must have a high tag. An example of this is the following:

```
let a = mkHigh 42
let b = mkLow 10
let c = a .+. b
return (tag of c == High && value of c == 42 + 10)

let d = mkLow 42
let e = mkLow 10
let f = a .<. b
return (tag of f == Low && value of f == 42 < 10)
```

The tags are only part of the type system and because of this some of the unsafe functions must be used in order to validate the tags. The unsafe functions used in the tests are `unwrapValue` in order to check the computed flow value and `unsafeShow` in order to get a string representation of the flow value. The string representation will be the value followed by "_<T>", where *T* is either *H* or *L* for high values and low values respectively. In the example above, `unsafeShow a` will yield the string "42_<H>" and `unsafeShow d` will yield the string "42_<L>".

Each test function is repeated 100000 times with random values and the source code for the `FlowNum` random tests can be found in Appendix D.

Algorithm 1 Testing FlowNum in SwapIFC algorithm

```
function TestNumBothLow
    $op \leftarrow oneof([Add, Sub, Mul, Neg, Abs, Sig])$
    $(val1, val2) \leftarrow (randomInteger, randomInteger)$
    $flow1 \leftarrow mkLow(val1)$
    $flow2 \leftarrow mkLow(val2)$
    return Validate flow1 flow2 op val1 val2 Low
end function
function TestNumBothHigh
    $op \leftarrow oneof([Add, Sub, Mul, Neg, Abs, Sig])$
    $(val1, val2) \leftarrow (randomInteger, randomInteger)$
    $flow1 \leftarrow mkHigh(val1)$
    $flow2 \leftarrow mkHigh(val2)$
    return Validate flow1 flow2 op val1 val2 High
end function
function TestNumMixedHighLow
    $op \leftarrow oneof([Add, Sub, Mul, Neg, Abs, Sig])$
    $(val1, val2) \leftarrow (randomInteger, randomInteger)$
    $flow1 \leftarrow mkHigh(val1)$
    $flow2 \leftarrow mkLow(val2)$
    return Validate flow1 flow2 op val1 val2 High
end function
```

Figure 6.3: Pseudo code for the testing of FlowNum in SwapIFC

7

Future work

No matter how satisfied one is, there is always more work that can be done. In this section, several different suggestions for how to proceed in the future will be presented. The suggestions are not only about how to make SwapIFC better, but also other potential work within the area of information flow control.

7.1 Full communication for SwapIFC to JSFlow

At the moment, code that is generated for high flows will not work with JSFlow. The reason for this is due to the runtime function `A` along with what the actual code generation of the Haskell code generates. A simple program for a high flow is

```
main = runFlow $ lprint (mkHigh 42)
```

which, when being executed in JSFlow should yield the output

```
(<>):42_<T>
```

However, the generated code will create an error in JSFlow, stating that *"write context <T> not below return context <>"*. This error is comming from the line `return f;` in the `A` function. The error is due to the following code

```
B(A(new T(function(){
    return _n/* Haste.Foreign.$wunsafeEval */("upg");
}),[[0, 42], _]))
```

which is the code that calls the `A` function with the arguments. Here, the value 42 will be upgraded to a secure value. However, the return context, i.e. where the call to `A` is made, is in a low context. Since it is not allowed to return a high value to a low context, this will produce an error.

Due to this, the most critical future work will be to get a full communication with high values working for JSFlow.

7.2 More primitives for SwapIFC

A library for a language is never completed until it easily supports the entire language. For SwapIFC, an easy yet time consuming extension would be to add classes and primitives for more standard classes of Haskell. Adding support for handling strings and lists within the Flow monad is an easy next step to do. However, adding a class for every standard Haskell class would be tedious, so looking more on if it is possible to have a more general class within the Flow monad and easily lift values into it could also be a good next step.

As of now, if a programmer want to use primitives for lists within a Flow instance he or she must write the entire function him/herself. This is something one should be able to assume a library should handle. As of right now, it can be handled using the Applicative instance. A very simple example of adding the primitive `(++)`, which given two lists concatenates them, is

```
flowConcat :: Applicative f => f [a] -> f [a] -> f [a]
flowConcat f1 f2 = (++) <$> f1 <*> f2
```

However, it has a major drawback. Since it uses Applicative and there are two different Applicative implementations (one for `High` and one for `Low`), both flows (`f1` and `f2`) must have the same tag. So as of right now, a programmer who uses SwapIFC must either implemement the missing primitives him/herself or accept that all operands must be of the same type (in this case the same tag).

7.3 Add support for Haste.App

When using Haste one can use Haste.App to perform program slicing [15]. The point of the program slicing is to help the programmer to split the application between the server side and the client side. Haste.App introduces three monads, *App*, *Client* and *Server* where Client is the monad for the client side and Server is the monad for the server side. App is the monad that ties an application together [47]. When using Haste.App and compiling with Haste, the compiler will produce client side code (i.e. JavaScript code) for everything in the Client monad while producing a binary file for the server side for everything within the Server monad. This allows a programmer to write code as one application while during the compile phase have it split. Currently, SwapIFC does not support Haste.App. Even though it could work, no work has been done to support Haste.App and therefore a good idea in the future would be to look at Haste.App and ensure that SwapIFC does indeed support Haste.App.

7.4 Remove the apply function in generated code

The function that is designed to handle function application in the generated JavaScript code from Haste has a flaw. As described in Chapter 5.4, Haste currently has a problem with an edge case in JavaScript. Whenever there is a function that should be applied to an argument list, but the function does not have an explicit argument list, Haste will run into problem, i.e. any function that has the property

```
function f() {
    // Do stuff here..
}
```

applied to an argument will yield a problem for Haste. The apply function `A` is part of the staticly generated runtime code that Haste generates. The function `A` takes a function and its argument list as arguments:

```
function A(f, args) {
    // ....
}
```

Since the code that is generated using FFI with `lprint` from SwapIFC was translated as

```
_s = B(A(new T(function(){
    return _n/* Haste.Foreign.$wunsafeEval */("lprint");
}), [_r = E(_r), _]))
```

where the argument `f` to `A` can be seen as

```
function lprint() {
    // ....
}
```

and the argument `args` to `A` is `[_r = E(_r), _]`. The argument `f` will have a length of 0 and due to this, the arguments in `args` will not be applied to the function `f`.

7.5 More features for JSFlow

As JSFlow only supports ECMA-262 v5 in non-strict mode and does not support JSON, a natural continuation would be to add and implement support for strict mode and JSON communication. Information flow control for

JSON objects imposes a big challenge. JSON objects are used to simplify data transfer over a network and is both easy to read/write for humans and easy to parse for computers [48]. It consists of *key, value* pairs, where a key (a string) is mapped to a value. An example of a simple JSON object is

```
{
  "firstName" : "Sherlock",
  "lastName" : "Holmes",
  "alive" : false,
  "children" : [],
  "address" : {
    "streetName" : "Baker Street",
    "streetNbr" : "221 B",
    "city" : "London",
    "country" : "England"
  },
  "spouse" : null
}
```

where the key `"firstName"` is mapped to the string `"Sherlock"` and the key `"address"` is mapped to a JSON object containing the information about the address. The challenges from an information flow control viewpoint is *how to effectively guarantee that no private information is leaked through JSON objects?*. It is easy to see it is not trivial to guarantee information flow security through JSON, but should be considered as a natural next step for a dynamic system designed to guarantee information flow control in JavaScript.

Another feature for JSFlow would be to start add what is new in ECMA-262 v6, e.g. adding support for *ArrayBuffer* which represents a generic, fixed-length binary data buffer [43]. Added support for *HTML5* with *Websockets* and make it compatible to run in a browser should of course also be considered as important features to add.

7.6 End-to-End information flow control

JSFlow can help check the JavaScript code of an application. However, currently there is no system that ensures a cross-origin end-to-end information flow control. As mentioned in Chapter 2.6, work has been done to create good information flow control libraries in Haskell. An example is LMonad, which provided information flow control for Yesod, including database interactions. There has also been work on securing database communication in LINQ using F# with SeLINQ [49], which would help track information flow across the boundaries of applications and databases. As a proof of concept, a type checker for a subset of F# was implemented. A good continuation would be to generalize the end-to-end communication. There are several ways this can be done.

- Build a dynamic system for information flow control in back-end communication (e.g. database communication).
- Create a compiler which compiles from a high-level language to e.g. the database query language to guarantee type safety and use the aforementioned tool to guarantee run-time information flow control.
- Model the system in e.g. Haskell and enable the rich type system to create a static analysis of the model to ensure no information leakage occurs.

Combining communication towards an information flow controlled database (as SeLINQ) with the extension mentioned in Chapter 7.3 could in theory give a static guarantee that the communication within the system will be free from information leakage. However, in order to do this, an adequate interface for database communication must be added as well as support for Haste.App. If one assumes a dynamic system (as SeLINQ), one could have a dynamic check for the front-end (using JSFlow) and the database communication (using SeLINQ). The only communication that will not have a dynamic information flow control would be the binary file that is the server. However, if one could validate that the static guarantee is indeed correct, a dynamic guarantee might not be needed.

8

Conclusions

This thesis attempted to explore how to effectively combine already created tools for securing JavaScript with focus on information flow control. While some success has been made with the implementation of a library and code generation towards JSFlow for all low flows, as explained in Chapter 7, there are still work and improvements to be done.

As explained in Chapter 4, there are numerous ways one could go about to create a library for information flow control. The fact that a monadic version was chosen had to do a lot with how easy it would be to add more features. Once the foundation is set, adding more primitives is rather trivial.

SwapIFC has a large set of primitives. It has an implementation of several standard classes in Haskell. One can also use SwapIFC and make standard computations without using the primitives but it comes at a cost. As explained in Chapter 7.2, using the Applicative instance is a good option if one needs primitives that are not implemented in SwapIFC. However, the cost is that one is stuck with one label - one can not combine flows with different tags using the Applicative instance.

When compiling with Haste, SwapIFC does fully support JSFlow with regards to low flows. Due to how the generated code is handled, support for high flows are currently not supported. However, in order for SwapIFC to be considered "ready", full support for JSFlow must be implemented.

Due to the extensive testing of SwapIFC, one can make an argument that it does indeed behave correctly. However, since no support for high flows are present in the compiled code, full testing of the code generation could not be done. Due to this, it is impossible to say if the code generation works or not. The primitives do not allow for "downgrading" of a flow, this is a fact due to the tests. But more tests must be conducted on the code generation, especially after full support for high flows has been implemented.

I believe that the focus for future work should be the following (with rank):

1. Add support for side effects using `IORef`.
2. Add full support for high flows (this might include removing the `A` function in the generated code).
3. Add support for Haste.App.
4. End-to-end communication, including database communication.
5. More features for JSFlow (in particular add support for JSON and have JSFlow run in a browser).
6. More primitives for SwapIFC.

Even though SwapIFC solves a subset of the problem set out in this thesis, it might be worth to look at potential restructuring. In theory, it could be possible to remove all primitives and simply use the IO monad.

The source code for SwapIFC can be downloaded and looked at from `https://github.com/alexandersjosten/swap-ifc`.

Bibliography

[1] I. Spectrum, S. Cass, N. Diakopoulus, Top 10 Programming Languages, [Accessed Oct. 24, 2014] (2014).
URL `http://spectrum.ieee.org/computing/software/top-10-programming-languages`

[2] Mozilla, About JavaScript, [Accessed Oct. 24, 2014] (2014).
URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript`

[3] OWASP, OWASP Top 10 2013, [Accessed Oct. 21, 2014] (2014).
URL `https://www.owasp.org/index.php/Top_10_2013-Top_10`

[4] OWASP, Cross-site Scripting (XSS), [Accessed Oct. 21, 2014] (2014).
URL `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`

[5] J. Kallin, I. L. Valbuena, Excess XSS - a comprehensive tutorial on cross-site scripting, [Accessed Oct. 21, 2014] (2013).
URL `http://excess-xss.com/`

[6] A. Ekblad, Haste, [Accessed Oct. 24, 2014] (2014).
URL `http://haste-lang.org/`

[7] A. Sabelfeld, A. C. Myers, Language-Based Information-Flow Security.
URL `http://www.cse.chalmers.se/~andrei/jsac.pdf`

[8] D. Hedin, L. Bello, A. Sabelfeld, A. Birgisson, Jsflow, [Accessed Oct. 30, 2014] (2014).
URL `http://chalmerslbs.bitbucket.org/jsflow/`

[9] D. Hedin, A. Sabelfeld, Information-flow security for a core of javascript.
URL `http://www.cse.chalmers.se/~andrei/jsflow-csf12.pdf`

[10] D. Hedin, A. Birgisson, L. Bello, A. Sabelfeld, Jsflow: Tracking information flow in javascript and its apis.
URL `http://www.cse.chalmers.se/~andrei/sac14.pdf`

[11] Microsoft, Typescript, [Accessed Oct. 21, 2014] (2012).
URL `http://www.typescriptlang.org/`

[12] B. S. Lerner, J. G. Politz, A. Guha, S. Krishnamurthi, TeJaS: Retrofitting Type Systems for JavaScript.
URL `http://cs.brown.edu/~blerner/papers/dls2013_tejas.html`

[13] TeJaS Git repository, [Accessed Oct. 24, 2014].
URL `https://github.com/brownplt/TeJaS`

[14] GHCJS Git repository, [Accessed Oct. 24, 2014].
URL `https://github.com/ghcjs/ghcjs`

[15] A. Ekblad, K. Claessen, A Seamless, Client-Centric Programming Model for Type Safe Web Applications.
URL `http://haste-lang.org/haskell14.pdf`

[16] Haskellwiki, What is Haskell?, [Accessed Oct. 24, 2014] (2013).
URL `http://www.haskell.org/haskellwiki/Introduction`

[17] J. Terrace, S. R. Beard, N. P. K. Katta, JavaScript in JavaScript (js.js): Sandboxing third-party scripts.
URL `http://jeffterrace.com/docs/jsjs.pdf`

[18] M. S. Miller, M. Samuel, B. Laurie, I. Awad, M. Stay, Caja - Safe active content in sanitized JavaScript.
URL `https://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`

[19] A. Russo, K. Claessen, J. Hughes, A library for light-weight information-flow security in haskell.
URL `http://www.cse.chalmers.se/~russo/publications_files/haskell22Ext-russo.pdf`

[20] HaskellWiki, If-then-else, [Accessed Jan. 3, 2015] (2008).
URL `https://www.haskell.org/haskellwiki/If-then-else`

[21] J. Goguen, J. Meseguer, Security policies and security models.
URL `https://www.cs.purdue.edu/homes/ninghui/readings/AccessControl/goguen_meseguer_82.pdf`

[22] ASCII Table and Desription, [Accessed Jan. 4, 2015] (2010).
URL `http://www.asciitable.com/`

[23] A. Sabelfeld, D. Sands, Dimensions and Principles of Declassification.
URL `http://www.cse.chalmers.se/~andrei/csfw05.ps`

[24] D. Stefan, A. Russo, J. C. Mitchell, D. Mazières, Flexible dynamic information flow control in haskell.
URL `http://www.scs.stanford.edu/~deian/pubs/stefan:2011:flexible.pdf`

[25] A. Russo, Seclib GIT, [Accessed Jan. 4, 2015] (2014).
URL `https://bitbucket.org/russo/seclib/src/`

[26] J. Parker, LMonad: Information Flow Control for Haskell Web Applications.
URL `http://jamesparker.me/doc/parker-thesis.pdf`

[27] Yesod Web Framework, [Accessed Feb. 5, 2015] (2012).
URL `http://www.yesodweb.com/`

[28] B. O'Sullivan, D. Stewart, J. Goerzen, Real World Haskell, O'Reilly Media, Inc, USA, 2008, [Chapter 18].
URL `http://book.realworldhaskell.org/read/monad-transformers.html`

[29] A. C. Myers, B. Liskov, Protecting Privacy using the Decentralized Label Model.
URL `http://www.cs.cornell.edu/andru/papers/iflow-tosem.pdf`

[30] Cornell University, Decentralized label model, [Accessed Feb. 13, 2015].
URL `https://www.cs.cornell.edu/jif/doc/jif-3.2.0/dlm.html`

[31] D. Hassan, A. Sabry, Encoding Secure Information Flow with Restricted Delegation and Revocation in Haskell.
URL `https://www.cs.indiana.edu/~sabry/papers/rdr.pdf`

[32] HaskellWiki, Embedded domain specific language, [Accessed Oct. 30, 2014] (2014).
URL `https://www.haskell.org/haskellwiki/Embedded_domain_specific_language`

[33] J. Svenningsson, E. Axelsson, Combining Deep and Shallow Embedding for EDSL.
URL `http://www.cse.chalmers.se/~emax/documents/svenningsson2013combining.pdf`

[34] HaskellWiki, Phantom type, [Accessed Dec. 8, 2014] (2013).
URL `https://www.haskell.org/haskellwiki/Phantom_type`

[35] HaskellWiki, Functor-Applicative-Monad Proposal, [Accessed Jan. 19, 2015] (2014).
URL https://www.haskell.org/haskellwiki/Functor-Applicative-Monad_Proposal

[36] Hackage, [Accessed Feb. 6, 2015].
URL http://hackage.haskell.org/package/base-4.7.0.2/docs/Prelude.html#t:Monad

[37] Hackage, [Accessed Feb. 6, 2015].
URL http://hackage.haskell.org/package/base-4.7.0.2/docs/Prelude.html#t:Functor

[38] HaskellWiki, Multi-parameter type class, [Accessed Feb. 13, 2015].
URL https://wiki.haskell.org/Multi-parameter_type_class

[39] HaskellWiki, Functional dependencies, [Accessed Feb. 13, 2015].
URL https://wiki.haskell.org/Functional_dependencies

[40] Mozilla, function, [Accessed Feb. 5, 2015] (2015).
URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function

[41] Mozilla, Arguments object, [Accessed Feb. 5, 2015] (2014).
URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments

[42] ECMA International, ECMAScript Language Specification (2011).
URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[43] Mozilla, Arraybuffer, [Accessed Jan. 28, 2015] (2014).
URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer

[44] ECMA International, ECMAScript Language Specification ECMA-262 6th Edition - DRAFT, [Accessed Jan. 28, 2015] (2014).
URL https://people.mozilla.org/~jorendorff/es6-draft.html#sec-arraybuffer-constructor

[45] K. Claessen, J. Hughes, QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.
URL http://www.cse.chalmers.se/edu/course/TDA342_Advanced_Functional_Programming/Papers/QuickCheck-claessen.ps

[46] HaskellWiki, Introduction to QuickCheck, [Accessed Feb. 2, 2015] (2013).
URL https://wiki.haskell.org/Introduction_to_QuickCheck1

[47] A. Ekblad, Haste.app, [Accessed Jan. 29, 2015] (2014).
URL `https://hackage.haskell.org/package/haste-compiler-0.4.4.1/docs/Haste-App.html`

[48] Introducing JSON, [Accessed Feb. 7, 2015].
URL `http://www.json.org/`

[49] D. Schoepe, D. Hedin, A. Sabelfeld, SeLINQ: Tracking Information across Application-Database Boundaries.
URL `http://www.cse.chalmers.se/~schoepe/selinq/selinq_long.pdf`

A

Structure of SwapIFC library

The SwapIFC library is divided into two different code bases - the trusted code base and the non-trusted code base. Within the trusted code base, all the unsafe operations and declassification and upgrading of values are permitted and exposed. In the non-trusted code base, the only exposed modules are the module defining `Flow` (*SwapIFC.Types*) and parts of the module defining `mkHigh, mkLow` (*SwapIFC.Auxiliary*).

The structure for the trusted code base is

```
SwapIFC.Trusted:
    exposed modules:
        SwapIFC.Unsafe
        SwapIFC.Auxiliary
        SwapIFC.Types
```

whereas the structure for the non-trusted code base is

```
SwapIFC:
    exposed modules:
        SwapIFC.Auxiliary (mkHigh, mkLow, lprint)
        SwapIFC.Types
```

B

Instance implementation

The implementation for Monad, Functor and Applicative for the Flow type became:

```
-- High Flow starts here!
instance Monad (Flow High) where
#ifdef __HASTE__
  return = Flow . upg
#else
  return = Flow . return
#endif

  (Flow ioa) >>= f = Flow $ do
    a <- ioa
    case (f a) of
      Flow iob -> iob

instance Functor (Flow High) where
  fmap f (Flow ioa) = Flow $ ioa >>= return . f

instance Applicative (Flow High) where
  pure = return
  (<*>) = ap

-- Low Flow starts here!
instance Monad (Flow Low) where
  return = Flow . return

  (Flow ioa) >>= f = Flow $ do
    a <- ioa
    case (f a) of
```

```
      Flow iob -> iob

instance Functor (Flow Low) where
  fmap f (Flow ioa) = Flow $ ioa >>= return . f

instance Applicative (Flow Low) where
  pure = return
  (<*>) = ap
```

C

Num instance in Flow type

```
-- Num instance for Flow
infixl 7 .*.
infixl 6 .+., .-.
class FlowNum t1 t2 t3 | t1 t2 -> t3 where
  -- | Binary addition operator for Flow
  (.+.) :: Num a => Flow t1 a -> Flow t2 a -> Flow t3 a
  (.+.) = calcNumFlow (+)

  -- | Binary multiplication operator for Flow
  (.*.) :: Num a => Flow t1 a -> Flow t2 a -> Flow t3 a
  (.*.) = calcNumFlow (*)

  -- | Binary subtraction operator for Flow
  (.-.) :: Num a => Flow t1 a -> Flow t2 a -> Flow t3 a
  (.-.) = calcNumFlow (-)

  -- | Unary negation for Flow
  fNeg :: (Num a, t1~t2, t1~t3) => Flow t1 a -> Flow t3 a
  fNeg = appNumFlow negate

  -- | Unary absolute value for Flow
  fAbs :: (Num a, t1~t2, t1~t3) => Flow t1 a -> Flow t3 a
  fAbs = appNumFlow abs

  -- | Unary signum for Flow
  fSig :: (Num a, t1~t2, t1~t3) => Flow t1 a -> Flow t3 a
  fSig = appNumFlow signum

instance FlowNum High High High
instance FlowNum High Low High
instance FlowNum Low High High
instance FlowNum Low Low Low

calcNumFlow :: Num a => (a -> a -> a)
                     -> Flow t1 a
                     -> Flow t2 a
                     -> Flow t3 a
calcNumFlow op (Flow ioa1) (Flow ioa2) = Flow $ do
  a1 <- ioa1
  a2 <- ioa2
  return $ op a1 a2

appNumFlow :: Num a => (a -> a) -> Flow t1 a -> Flow t1 a
appNumFlow f (Flow ioa) = Flow $ do
  a <- ioa
  return $ f a
```

D

Tests

```haskell
-- The FlowNum tests
prop_lowNumExpr :: (NumOp, Int, Int) -> Bool
prop_lowNumExpr (op, val1, val2) =
  case op of
    Add -> (show (flow1 .+. flow2) == show (flow2 .+. flow1)) &&
           show (flow1 .+. flow2) == show (val1 + val2) ++ "_<L>"
    Sub -> show (flow1 .-. flow2) == show (val1 - val2) ++ "_<L>" &&
           show (flow2 .-. flow1) == show (val2 - val1) ++ "_<L>"
    Mul -> (show (flow1 .*. flow2) == show (flow2 .*. flow1)) &&
           show (flow1 .*. flow2) == show (val1 * val2) ++ "_<L>"
    Neg -> show (fNeg flow1) == show (negate val1) ++ "_<L>"
    Abs -> show (fAbs flow1) == show (abs val1) ++ "_<L>"
    Sig -> show (fSig flow1) == show (signum val1) ++ "_<L>"
  where flow1 = mkLow val1
        flow2 = mkLow val2

prop_highNumExpr :: (NumOp, Int, Int) -> Bool
prop_highNumExpr (op, val1, val2) =
  case op of
    Add -> (show (flow1 .+. flow2) == show (flow2 .+. flow1)) &&
           show (flow1 .+. flow2) == show (val1 + val2) ++ "_<H>"
    Sub -> show (flow1 .-. flow2) == show (val1 - val2) ++ "_<H>" &&
           show (flow2 .-. flow1) == show (val2 - val1) ++ "_<H>"
    Mul -> (show (flow1 .*. flow2) == show (flow2 .*. flow1)) &&
           show (flow1 .*. flow2) == show (val1 * val2) ++ "_<H>"
    Neg -> show (fNeg flow1) == show (negate val1) ++ "_<H>"
    Abs -> show (fAbs flow1) == show (abs val1) ++ "_<H>"
    Sig -> show (fSig flow1) == show (signum val1) ++ "_<H>"
  where flow1 = mkHigh val1
```

```
        flow2 = mkHigh val2

prop_mixedNumExpr :: (NumOp, Int, Int) -> Bool
prop_mixedNumExpr (op, val1, val2) =
  case op of
    Add -> (show (flow1 .+. flow2) == show (flow2 .+. flow1)) &&
           show (flow1 .+. flow2) == show (val1 + val2) ++ "_<H>"
    Sub -> show (flow1 .-. flow2) == show (val1 - val2) ++ "_<H>" &&
           show (flow2 .-. flow1) == show (val2 - val1) ++ "_<H>"
    Mul -> (show (flow1 .*. flow2) == show (flow2 .*. flow1)) &&
           show (flow1 .*. flow2) == show (val1 * val2) ++ "_<H>"
    Neg -> show (fNeg flow1) == show (negate val1) ++ "_<H>"
    Abs -> show (fAbs flow1) == show (abs val1) ++ "_<H>"
    Sig -> show (fSig flow1) == show (signum val1) ++ "_<H>"
  where flow1 = mkHigh val1
        flow2 = mkLow val2
```