Thesis for the degree of Licentiate of Engineering

Programming Language Design

Issues in Web Programming and Security

Niklas Broberg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2005

Programming Language Design
Issues in Web Programming and Security
Niklas Broberg

Technical report 24L
ISSN 1652-876X
Department of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, Sweden, 2006

Programming Language Design

Issues in Web Programming and Security

Niklas Broberg

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

Abstract

This thesis consists of two separate parts. Both concern programming language design, the first in the domain of web programming and the other for security.

The first part consists of two papers, both discussing various aspects of how to extend the general purpose programming language Haskell to make it serve as a specialized scripting language for writing dynamic web pages. The first paper in this area concerns one specific theoretic aspect of this extension, namely how to extend Haskell with regular expression pattern matching. We discuss syntax, typing and semantics for regular expression patterns, and show an implementation of the system in Haskell. In the second paper we give an overview of Haskell Server Pages, an extension of Haskell for writing dynamic web pages. Then we go on to discuss how to implement the runtime system of this language by using on-request compilation and dynamic loading of pages into a running server application.

The second part of the thesis concerns security, and in particular language-based information flow security. We show a calculus, based on λ-calculus with references, that allows dynamic changes to the flow policies of a program during execution. We also give a type system for the calculus that tracks valid flows, and a semantics. To prove that our type system is sound, we define a non-interference-like semantic security property and prove that it is implied by the type system using a bisimulation-style proof. Our aim with the calculus is to provide a core calculus that can be used to explain properties of other systems. To establish it as such, we also show how to encode various other similar systems in our calculus.

Keywords: Functional programming, web programming, dynamic loading, regular expressions, security, non-interference, calculus, core calculus, bisimulation.

Acknowledgements

With such diverse research interests, I have plenty of people to give thanks.

First of all, to my supervisor David Sands who introduced me to the all the fun of language-based security. He is truly a great supervisor, a constant source of helpful suggestions and good advise, and with an exceptional ability to inspire and enthuse. Not to mention he's a fun fellow too.

To Josef Svenningsson, my first supervisor during my undergraduate years, now a very good friend and coworker. Without him pushing and guiding me back then, I would not have been where I am today.

To Ulf Norell, my second supervisor, for all the interesting discussions, and for our fun-filled ICFPC sessions. Next year...

To Rogardt Heldal for the inspiring cooperation and our interesting discussions on a variety of topics, and for always supporting me.

To all my fellow PhD students of helping create the nice working environment we all share, and for constantly distracting me when I should be doing less fun stuff.

To Andrei Sabelfeld, Daniel Hedin and Ulf Norell for insightful comments on this introductory text.

And most of all, to my wife, for understanding and supporting my long and crazy hours. For coping with my frustration when things don't work as I want them to, and for multiplying my happiness when they do. And for reminding me of what is most important in life.

Table of Contents

Introduction

Imagine that you want to split a log into smaller pieces to use as firewood. To your disposal you have an axe and a knife. Modulo inexperience, it will probably not be very hard for you to chop the log into smaller pieces with the axe, and then if needed use the knife to split those pieces into even smaller ones.

Now imagine instead that you want to cut a wooden board in half, to use on the wall of a house, and you get to use the same axe and knife. Clearly this time the task won't be quite as easy, since the tools are not well suited. It will be doable, hacking and whittling away at the proper place on the board, possibly trying to mimic a saw with the knife, but it will take a lot of time and the end result may be somewhat lacking. In particular when compared to doing the same thing with a real saw.

Even worse, what if you wanted to build that wall, and actually had a saw at your disposal, but only firewood logs to build the wall from?

Whatever the craft, the key to success is having the right tools and materials. Skill is important, but it can never fully compensate for the lack of good materials, or not having the right tools at hand.

Programming language design is the art of supplying the proper tools and materials to programmers so that they can properly do their craft, writing programs. And just like with any other craft, what tools you would prefer to use depends very much on the task at hand. If you want to write a program that does mathematical computations, a language that provides floating point numbers and operations on these would be well suited. If your program is supposed to communicate with a user via text commands, it would be preferable to use a language that provides text strings and operations on these. You could probably manage the input and output of strings by encoding them as numbers and converting these to text when printing them on the screen, and historically languages have done just that, but in a sense that would be analogous to building your wall from logs.

R. D. Tennent formulated this general idea in his book Principles of Programming Languages [14] as the principle of abstraction, that "values of a syntactically relevant domain can be given a name". In other words, if we want to use text strings, we should call them text strings, and distinguish them within the language from simple sequences of numbers.

This may all sound completely obvious, but unfortunately not all languages adhere to this principle of abstraction. Either the abstractions they supply the programmer with are not particularly well suited for the tasks within the domains they are intended for, leaving you to build your house with tools really intended for other tasks. Or they don't supply anything at

all for the domain, leaving you like stranded like Robinson Crusoe, trying to build your house from whatever you can find.

There are really two ways that languages could adhere to the principle of abstraction. One is quite naturally to equip a language with data types and operations for the relevant domain as primitives, analogous to putting a toolbox together that contains just the tools you need for a particular task. A language specialized in this way to work in a particular niche domain is often referred to as a domain-specific language. The other way is to equip the language with abstraction mechanisms powerful enough to let the programmer define the proper operations and types himself. By stretching the analogy a bit, we could see this as having a set of tool parts that could be pieced together in various ways to form different tools depending on what is needed at the time. Such languages are referred to general-purpose languages. However, it may be tedious to piece the parts together to form the proper tools for a particular complex task. Thus if a general-purpose language is to be used to write programs within a particular domain, language design can also be to define the proper types and operations within that general-purpose language as a library, to relieve the programmer of that burden. A language created in this way is called an *embedded* domain-specific language, since the special-purpose language is embedded inside the general-purpose one.

In particular within the areas of web programming and security, the languages commonly used all have serious design short-comings. We will discuss each of these areas in turn, showing what the flaws of the common languages are and what is being done to address them, by us and by others.

Web programming Long gone are the days when the World Wide Web consisted mostly of static HTML pages, shown to web users when requested. Today most web sites consist not of pages, but of programs that generate pages when they are requested. This means that the same location may show a different page depending on for instance client input or the contents of a database. Such programs that generate pages on request are often referred to as dynamic web pages, and web programming is the task of writing such pages.

As the use of dynamic web pages has increased, so too has the need for better tools to use when creating them. The Common Gateway Interface (CGI) [2] is used by web servers to allow dynamic web pages written in any language, as long as they produce output on a particular format. This means that any general-purpose programming language can be used for web programming, and for many such languages there exist libraries to help web programmers do this in an easier way. Some of the languages most com-

monly used like this are Perl, Python and C. However, far more common than using CGI for dynamic web pages is to use a domain-specific scripting language such as PHP, ASP or ASP.NET [3, 6, 1]. These are all designed with web programming in mind, and come with built-in support for many of the specifics that dynamic web pages need.

However, most if not all of these commonly used languages, general-purpose and domain-specific alike, share a fundamental flaw – they model the page output as raw text! This severely violates the principle of abstraction, since we cannot distinguish the HTML or XML data that makes up a document from a simple string of characters. This means, among other things, that we can never be quite sure that the output of such a page is really HTML data that we can read from a web browser. It is fully possible for a web programmer to write a program that outputs an ill-formed result page that will look like nothing at all to a web browser.

It is widely recognized [4, 7, 9, 10, 16] that the functional programming idiom is particularly well suited for creating and manipulating XML and HTML documents. This is because functional languages support algebraic datatypes, an abstraction mechanism that can be used to represent HTML or XML data in an easy, non-text-based way that can give well-formedness guarantees for the resulting pages. A good deal of libraries exist [5, 8, 15, 16] that assist in writing CGI programs in functional languages, forming embedded domain-specific languages that do not suffer from the text-based output representation.

Unfortunately CGI programs suffer from some drawbacks. They are inherently stateless since one request of a CGI page causes one execution of the corresponding CGI program. Also, writing CGI programs requires at least some non-trivial knowledge of the host language, even when adding very simple dynamic contents, e.g. an access counter. Suddenly a web page is a program, and coming from a background of writing HTML web sites this could be quite a leap. Such a steep initial learning curve means many aspiring web programmers with no previous programming experience, but with experience in creating static HTML pages, will instead choose one of the specialized scripting languages that allow a much simpler transition from static XHTML to dynamic pages. Indeed, this is probably one of the most prominent reasons why specialized scripting languages, and PHP in particular, are so popular today.

In our paper "Haskell Server Pages through Dynamic Loading", starting on page 49, we discuss how to get the best of both worlds. We extend the functional general-purpose language Haskell with syntactic and runtime support for it to act as a specialized scripting language, called Haskell Server Pages (HSP), that models XML not as text but as an algebraic datatype. In

particular we focus in the paper on the runtime aspects of the language, i.e. how to run a dynamic web page written in HSP. We do this by compiling pages on the first request, and dynamically loading the compiled code into a running server application that provides the pages with a useful environment.

This paper was first published in the *Proceedings of the ACM SIGPLAN 2005 Haskell Workshop, Haskell '05, ACM Press, 2005.*

Our other paper in the web area, "Regular Expression Patterns", starting on page 15, written together with Andreas Farre and Josef Svenningsson, discusses one particular aspect of extending Haskell towards a specialized scripting language. When pattern matching on the tree structure of XML, ordinary pattern matching is not powerful enough, you need the full expressional power of regular expressions. In the paper we show how to extend the normal pattern matching facility of Haskell with regular expression matching. We discuss the design of syntax and semantics, and show a lightweight implementation as a preprocessor to vanilla Haskell.

This paper was first published in the *Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming, ICFP '04, Volume 39 Issue 9, ACM Press, 2004.*

Security We can talk about web programming, but security programming is a rather odd term – there really is no such thing as a security program, the way we would talk about a web program. There are programs where security is an important aspect, but the main functionality lies almost certainly within some other domain.

For this reason it is a bit tricky to design a language that takes security into account. It is hard to do it as a domain-specific language, since security is not a domain per se. Rather it needs to be a complement to any other language in which we intend to write security critical applications. If we are designing a specialized language for a domain, we need to ensure that any security aspects of that domain are taken into account. If we want to write security critical applications in a general-purpose language, we would need that language to support some notion of security.

When we talk about security we mean security of data. Some aspects of data security are information flow, i.e. tracking where data flows in a program to ensure that it doesn't end up where it shouldn't be, and data integrity, i.e. making sure that no untrusted data ends up in places where it could affect the behavior of a program.

For any program with security constraints, one could imagine a solution based on runtime checks that works for any domain. For every security critical operation in the program, we insert a check to see whether that

operation is allowed, and if it isn't we halt the program with an error, throw an exception, or something similar. There are problems with this simple approach however. One is that it might not be obvious to the programmer where the checks should be inserted, which could lead to unintended leaks due to missing checks. Another is that using such a check can in itself leak information, since knowing whether or not the program halted can tell us something about the values it was trying to operate on.

However, the most crucial problem with this approach is that runtime checks can only ever detect the presence of a flow, never the absence of one. This can give rise to so called *indirect* leaks, where observing the value of some public data can give away what the value of some secret is. In fact, it is impossible to discover indirect leaks dynamically, it must be done statically e.g. through static analysis of the code.

The term normally used for when a program does not leak anything is *end-to-end* security. Somewhat simplified, we think of a program as having various public and secret inputs, and producing various public and secret outputs. The program is end-to-end secure if the values of secret inputs do not influence the values of public outputs in any way. The secrecy of those values is preserved from one end of the program to the other.

Today, no mainstream programming language directly supports end-to-end security. Programmers are left on their own, and any security guarantees that they want to place on programs must be asserted through other means, e.g. program analysis. Many languages may seem to support security at a first glance, e.g. through mechanisms for access control, but as we have argued, dynamic solutions can never provide full end-to-end security.

There is however a lot of ongoing research in this area, and one approach that many find interesting is the idea of security-typed languages, i.e. languages where the security aspects of programs are tracked by the type system at compile time. Doing the checks at compile time instead of using runtime checks means that there will be no leaks through the checks themselves, and that indirect leaks can be caught. Also the security typing would stop a programmer from writing programs that contain unintentional leaks, by pointing out any leaks that could arise.

One particularly tricky problem within this area, that has attracted a lot of attention lately, is how to design a security type system that allows the programmer to *intentionally* leak information. This is generally referred to as declassification, i.e. making data non-classified, and is very often needed in real applications that deal with security, e.g. releasing some information to a customer after payment has been made. Sabelfeld and Sands have written a survey paper that gives an overview of the research in this particular area [12].

Two full-scale security-typed languages exist today as research products – FlowCaml [13], an extension of OCaml that deals with security typing, and JIF (Java Information Flow) [11], a similar extension for Java. FlowCaml has a very rigorous theoretical background, but lacks mechanisms for declassification. JIF on the other hand supports declassification, but the theory behind it is less well investigated.

In our paper "Flow Locks: Towards a Core Calculus for Dynamic Dependencies", written together with David Sands, we define a calculus with a security type system that allows information flow policies in a program to vary during execution, thereby enabling (among other things) declassification. We intend this as a core calculus, i.e. a calculus that can be used as a foundation for other systems, and that can thus help explain how those other systems work. We give the calculus, its semantics and type system, and prove that the type system gives the expected security guarantees. We also give a brief overview of other systems and how these could be encoded in, and thus explained by, our calculus.

This paper is an extended version of the paper that was published in the *Proceedings of the 15th European Symposium on Programming, ESOP 2006, LNCS 3924, Springer-Verlag, 2006.*

Bibliography

[1] ASP.NET home. `http://msdn.microsoft.com/asp.net/`.

[2] Common Gateway Interface. `http://www.w3.org/CGI/`.

[3] PHP. `http://php.net/`.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[5] A. Gill. The Haskell HTML Library, version 0.3. `http://www.cse.ogi.edu/~andy/html/intro.htm`.

[6] S. Hillier and D. Mezick. *Programming Active Server Pages*. Microsoft Press, 1997.

[7] H. Hosoya and B. C. Peirce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 2(3):117–148, 2003.

[8] E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 1(1), 1998.

[9] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[10] E. Meijer and D. van Velzen. Haskell Server Pages: Functional programming and the battle for the middle tier. 2001.

[11] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2004.

[12] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. 2005.

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

[14] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[15] P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, 2002.

[16] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation. In *Proceedings of the ACM International Conference on Functional Programming*. ACM Press, 1999.

Regular Expression Patterns

Niklas Broberg Andreas Farre Josef Svenningsson

Abstract

We extend Haskell with *regular expression patterns*. Regular expression patterns provide means for matching and extracting data which goes well beyond ordinary pattern matching as found in Haskell. It has proven useful for string manipulation and for processing structured data such as XML. Regular expression patterns can be used with arbitrary lists, and work seamlessly together with ordinary pattern matching in Haskell.

Our extension is lightweight, it is little more than syntactic sugar. We present a semantics and a type system, and show how to implement it as a preprocessor to Haskell.

1 Introduction

Pattern matching as found in many functional languages is a nice feature. It allows for clear and succint definitions of functions by cases and works very naturally together with algebraic data types. But sometimes ordinary pattern matching is not enough. A distinct feature of this form of pattern matching is that it only examines the outermost constructors of a data type. While this allows for efficient implementations it is also a rather limited construct for analysing and retrieving data.

A well-known example of a construct that provides deeper and more complex retrievals are regular expressions for strings. While this is not a very common feature among programming languages it is one of the key constructs that have made Perl so popular. Regular expressions are ideal for various forms of string manipulation, text extraction etc, however, they remain a very domain specific and ad-hoc construct targeted only for one particular data structure, namely strings.

On another axis we find the recent trend in XML centric languages. The first attempts at such languages used the ordinary pattern matching facility of functional languages to analyze XML fragments [13]. This was found to be

too restrictive, so in order to be able to express more sophisticated patterns and transformations on XML fragments the notion of *regular expression patterns* were invented. Examples of languages including this feature are XDuce [7] and CDuce [2]. While this is a great boost for the XML programmer, in the case of XDuce it only works for XML data and not for any other data. Furthermore those pattern matching constructs are closely tied to rather sophisticated type systems which makes them somewhat heavyweight.

In this paper we extend Haskell with regular expression patterns. Our extension has the following advantages:

- Our proposal is *lightweight*. It is hardly more than syntactic sugar. Most notably it does not require any complex additions to the type system.

- It works for *arbitrary lists*. It is a general construct and not tied to a specific data type for elements. But it should be noted that it works in particular for strings since strings are just lists of characters in Haskell.

- It fits seamlessly with the ordinary pattern matching facility found in Haskell.

In this paper we give a detailed semantics and type system of regular expression patterns. The extension has been implemented as a preprocessor to Haskell, and we sketch the implementation

While we have chosen to focus on Haskell in this paper there are very little Haskell specific details. We are quite confident that our proposal could be adapted to any similar functional language.

In recent years a number of papers have been devoted to developing efficient pattern matching and efficient regular matching [5, 6, 9]. This is not the concern of this paper. Although efficiency is an important consideration we focus only on language design.

Another issue that we do not address is the question of overlapping and exhaustive patterns. We are confident that the existing techniques developed for XML centric languages will do the job nicely [8]. Note also that in general it is undecidable to check whether patterns are overlapping or non-exhaustive in Haskell because of guards, so in our setting it is something of a non-issue.

2 Regular expression patterns by example

2.1 Ordinary pattern matching

Assume that we have the following datatype representing an entry in an address book.

```
data Contact = Person Name [ContactMode]
data ContactMode = Tel TelNr
```

We can assume that the types `Name` and `TelNr` are type synonyms for `String`. The reason for not inlining `TelNr` in `Contact` is because we will later want to add other means of contact, e.g. email addresses, to our address book.

Now consider two different functions that extract information from a contact; `firstTel` will return the first `TelNr` in the list of contact modes associated with a contact. `lastTel` will analogously return the last associated `TelNr`. The first is easy to write using simple pattern matching on a contact:

```
firstTel :: Contact -> TelNr
firstTel (Person _ (Tel nr : _)) = nr
firstTel (Person _ []) = error "No Tel"
```

The second function, although its functionality is very similar to `firstTel`, cannot be written in the same simple way. We must instead resort to recursion and an auxiliary function to step through the list until we reach the end.

```
lastTel :: Contact -> TelNr
lastTel (Person _ nrs) = aux nrs
  where aux [] = error "No Tel"
        aux [Tel nr] = nr
        aux (_:nrs) = aux nrs
```

Although the two functions have very similar functionality, only one of them can be written using direct pattern matching. Why is this so? The answer lies, of course, in the list datatype. A (non-empty) list has a head and a tail, so extracting the first element is easy. To get to the last element however, we must recursively look at the tail for its last element. In other words, we must first match on the structure of the list, before being able to look at the elements.

Haskell has a construct for matching directly on the elements of a list, but only for fixed-size lists. If we know that a contact never has more than three phone numbers, we could write `lastTel` as (we will ignore the erroneous case from now on)

```
lastTel (Person _ [Tel nr]) = nr
lastTel (Person _ [_, Tel nr] = nr
lastTel (Person _ [_, _, Tel nr] = nr
```

Clearly this is not a very good solution. Even for this very small task we must write far more than we are comfortable with, and for larger lists or more complex datatypes this approach quickly becomes infeasible. What we need is a way of saying "match a list containing a `Tel`, preceded by any number of other elements". This is where regular expression patterns enter the picture.

2.2 Regular expression patterns

Mathematically a regular expression defines a regular language, where language in this context means a (possibly infinite) set of words, and each word is a sequence of elements taken from some alphabet. We can use a regular expression as a validator and try to match an arbitrary word against it to find out if the word belongs to that regular language or not. The basic regular expression operators are repetition, concatenation and choice. Concatenation is straight-forward, ab means a followed by b. Choice $(a|b)$ means either a or b. Repetition $a*$ means zero or more occurrences of a. Repetition can be defined using choice and recursion as $a* = \epsilon|aa*$ where ϵ denotes the empty sequence. As an example, consider the regular expression $e = a*|b*$. The language defined by e, denoted $L(e)$, is the set of all words consisting of only a's or only b's, including the empty word. We have that $aa \in L(e)$, $bbb \in L(e)$, but $ab \notin L(e)$. In other words, aa and bbb both match the regular expression e, but ab doesn't.

This notion of treating a regular expression as a validator is very similar to the concept of pattern matching in Haskell. We take a Haskell value (a word) and a pattern (a regular expression) and try to match them, getting a yes or no as the result. Combining these two concepts is straight-forward, yielding what we call regular expression patterns. As noted, a regular expression can be matched against a sequence of elements from some alphabet. Lifting this idea into Haskell, a regular expression pattern can be matched against a list of elements of some datatype. When we speak of a sequence, we mean a sequence of elements in the abstract sense. In contrast, when we speak of a list, we mean the list datatype that is used to encode sequences in Haskell.

Returning to our `lastTel` function, we can now easily write it with a single pattern match by using a repetition regular expression pattern:

```
lastTel (Person _ [_*, Tel nr]) = nr
```

We write concatenation using commas as with ordinary Haskell lists, and we denote repetition with *. As we can see from the example, regular expression

patterns are actually more flexible than bare regular expressions. A regular expression is built from elements of some alphabet, the same alphabet that the words it may match are built from. A regular expression pattern on the other hand is built from *patterns over* elements of some datatype, allowing us to use constructs like wildcards and pattern variables. We use the term regular expression pattern both for the subpatterns (repetition, choice etc) and for a top-level list pattern that contains the former. It should be clear from the context which we are referring to.

2.3 Repetition and Ambiguities

Let us see what else we can do with regular expression patterns. First, as promised, we extend our datatype with email addresses.

```
data Contact = Person Name [ContactMode]
data ContactMode = Tel TelNr | Email EAddr
```

If we only have ordinary pattern matching we cannot even write `firstTel` without resorting to recursion and auxiliary functions.

```
firstTel (Person _ cmodes) = aux cmodes
 where aux (Tel nr : _) = nr
       aux (_ : cmodes) = aux cmodes
```

Using a regular expression pattern, we can write it in one go:

```
firstTel (Person _ [(Email _)*, Tel nr, _*]) = nr
```

The straight-forward intuition of the pattern above is that the first Tel in the list is preceded by zero or more `Email`s (but no `Tel`s), and any number of other elements may follow it. We can easily write `lastTel` in a similar way as

```
lastTel (Person _ [_*, Tel nr, (Email _)*]) = nr
```

But seeing these two definitions leads to an interesting question: What happens if we write the function

```
someTel :: Contact -> TelNr
someTel (Person _ [_*, Tel nr, _*]) = nr
```

i.e. where the `Tel` in question may both be preceded and succeeded by other `Tel`s? Clearly this pattern is ambiguous, since if we match it to e.g. `Person "Niklas" [Tel 12345, Tel 23456, Tel 34567]` we can derive a match for

either of the three `TelNr`s to be bound to `nr`, by letting the first `_*` match either 0, 1 or 2 `Tel`s. To disambiguate such issues, we adopt the policy that a repetition pattern will always match as few elements as possible while still letting the whole pattern match the given list. In standard terminology, our repetition regular patterns are non-greedy. This policy means that `someTel` above will be exactly the same as our `firstTel` function, since the first `_*` will now try to match as few elements as possible.

In some cases though, such as `lastTel`, we want the greedy behavior. To this end we let the programmer specify if a repetition pattern should be greedy by adding an exclamation mark (`!`) to it, e.g. in the following definition of `lastTel`:

```
lastTel (Person  [_*!, Tel nr, _*]) = nr
```

2.4 Choice patterns

Now that we've seen the power of repetition patterns, we turn our attention to choice patterns. Assume that we want a function `allTels` that returns a list of all telephone numbers associated with a contact. Without regular expression patterns we must once more resort to recursion and auxiliary functions.

```
allTels :: Contact -> [TelNr]
allTels (Person _ cmodes) = aux cmodes
 where aux [] = []
       aux (Tel nr : cmodes)
           = nr : aux cmodes
       aux (_ : cmodes) = aux cmodes
```

Using a combination of repetition and choice, we can write it as

```
allTels (Person _ [ (Tel nr | _)* ]) = nr
```

The intuition here is that each element in the list of contact modes is either a `Tel` or something else (`_`). Every time that we encounter a `Tel`, we should include the associated `TelNr` in the result. As the example shows we can achieve this accumulation of `TelNr`s with a single pattern variable. Since the intuition of a repetition pattern is that its subpattern, i.e. the pattern it encloses, should be matched zero or more times, the same must be true for any pattern variables inside such a pattern. For each repetition, such a variable will match a new value. Clearly the only sensible thing to do is to let that variable bind to a list of all those matched values.

This treatment of variables breaks one aspect of Haskell's linearity property – that the occurrence of a variable in a pattern will bind that variable to exactly one value of the type that it matches. We will therefore call such a variable non-linear. A non-linear variable will be bound to a list of values that it matches, in the order that they were matched (i.e. the order in which they appeared in the matched list). When we speak of a non-linear binding, we mean a binding of a non-linear variable to a list of values. We will also use the terms *non-linear context* to mean a context in which linear variables cannot appear, and *non-linear patterns*, by which we mean patterns whose subpatterns will always be matched in a non-linear context.

By the example above we see that a repetition pattern is a non-linear pattern, and consequently that the variable `nr` appears in a non-linear context. Similarly a choice pattern is also non-linear. If we remove the repetition from the regular expression pattern in `allTels` we get the pattern `[Tel nr|_]` for matching a list of exactly one element. If that element is a `Tel` we will have a value to bind to `nr`, but if it is an `Email` we have none! Thus we still cannot guarantee that a variable gets one value; in this case `nr` will be bound to a list with zero or one element.

The function `allTels` shows how regular expression patterns can be used for filtering a list based on pattern matching. We can go one step further and do partitioning, e.g.

```
allTelsAndEmails :: Contact -> ([TelNr],[EAddr])
allTelsAndEmails (Person _
    [(Tel nr | Email eaddr)* ]) = (nr, eaddr)
```

A choice pattern can also be ambiguous if any of its subpatterns overlap, as in

```
sillyAllTels :: Contact -> ([TelNr],[TelNr])
sillyAllTels (Person _ [ (Tel nr | Tel mr | _)* ])
 = (nr, mr)
```

To disambiguate this we adopt a first-match policy, much like that of Haskell pattern matching. Thus we first check if the first subpattern matches, and consider the k:th subpattern only if no pattern $i < k$ matches. Note that we allow choice patterns to contain more than two subpatterns. Choice patterns are right associative so for example `[ (Tel nr | Tel mr | _)* ]` is parenthesised like `[ (Tel nr | (Tel mr | _))* ]`. Another interesting thing about choice patterns is that we allow a variable to appear in both subpatterns assuming that it binds to values of the same type. For instance, if our datatype for modes of contact was defined as

```
data ContactMode = Home TelNr | Work TelNr
```

we could define `allTels` as

```
allTels (Person _ [(Home nr | Work nr)*]) = nr
```

Variables in choice patterns are still non-linear even if they appear in all subpatterns, so the function

```
singleTel (Person _ [(Home nr | Work nr)]) = nr
```

will have the type `Contact -> [TelNr]`.

2.5 Subsequences and option patterns

Regular expressions allow grouping of elements and subexpressions using parentheses. For example, the regular expression $e = (ba)*$ will match the words *ba*, *baba* etc. To add this feature to our regular expression patterns we need to introduce some new syntax, since using ordinary parentheses in Haskell will denote tuples, as in

```
wrongEveryOther [(_,b)*] = b
```

We (somewhat arbitrarily) choose to denote subsequences with `(/` and `/)`, so a correct function that picks out every other element from a list can be written as

```
everyOther :: [a] -> [a]
everyOther [(/_, b/)*] = b
```

There's a problem with the above definition though; it works for lists of even length only. Surely we want `everyOther` to work for any list. To achieve this we could add another declaration to the one above like

```
everyOther [(/_, b/)*, _] = b
```

to catch the cases where the list is of odd length too. But couldn't we write these two cases as a single pattern? Indeed we can, using a choice pattern

```
everyOther [(/_, b/)*, ((/ /) | _)]
```

where `(/ /)` denotes the empty subsequence, ϵ. However, this pattern is so common that regular expressions define a separate operator, ?, to denote optional regular expressions. The definition of ? is $e? = e|\epsilon$, and by lifting this to regular expression patterns we can write `everyOther` more compactly as

```
everyOther [(/_, b/)*, _?] = b
```

Obviously, optional patterns are non-linear since they can be defined in terms of choice patterns which are non-linear. Just as for a repetition pattern, an optional pattern is non-greedy by default. We also define greedy optional patterns by `?!` in analogy with greedy repetition patterns.

2.6 Non-empty repetition patterns

There is one more operator to discuss, namely `+` that is used to denote non-empty repetition. For instance we might require all contacts to have at least one mode of contact registered, either a telephone number or an email, otherwise it is an error. To enforce this we may want to define `allTelsAndEmails` from above as

```
allTelsAndEmails
 (Person _ [(Tel nr  | Email eaddr)
           ,(Tel nrs | Email eaddrs)*])
   = (nr ++ nrs, eaddr ++ eaddrs)
```

Using `+` we can define this more compactly as

```
allTelsAndEmails (Person _ [(Tel nr | Email eaddr)+])
 = (nr, eaddr)
```

Modulo variables bound, $p+ \equiv pp*$. It is non-linear and non-greedy just like `*`, and there is a greedy counterpart `+!`.

2.7 Variable bindings and their types

Since we can use any Haskell pattern inside regular expression patterns, we can in particular use pattern variables to extract values from the list that we match against, as we have seen in various examples already. Haskell also defines a way to explicitly bind values to a variable using the `@` operator. E.g. in the declaration

```
allCModes :: Contact -> [ContactMode]
allCModes (Person _ all@[(Tel _ | Email _)+]) = all
```

the variable `all` will be bound to the (non-empty) list of `ContactMode`s associated with a contact. This is a very useful feature to have for regular expression patterns as well, for instance we may want to write a function that picks the first two elements from a list as

```
twoFirst :: [a] -> [a]
twoFirst [a@(/_, _/), _*] = a
```

However, adding this feature raises some interesting questions. Firstly, what will the type of a variable bound to a regular expression pattern be? For a subsequence it seems fairly obvious that it will have a list type, but what about repetitions, choices and optional patterns? To this issue there is no obvious right answer, one way might be to let a variable be bound to all elements matched by the subpattern in analogy with implicitly bound variables. We have chosen a slightly different approach in which we assign different types to patterns to mirror the intuition behind them.

Subsequences and repetition patterns will both have list types since they represent sequences. There's a difference between them though; a subsequence is just what the name implies, a subsection of the original sequence. Thus a variable bound to it will always have the same type as the input list, i.e. a list of elements. A repetition pattern on the other hand is a repetition of some subpattern, and so it will have the type of a list of that subpattern. For choice patterns we make use of Haskell's built-in `Either` type defined as

```
data Either a b = Left a | Right b
```

By using this type we can allow the left and right subpatterns of a choice pattern to have different types, for instance

```
singleCMode :: [ContactMode]
             -> Either ContactMode ContactMode
singleCMode [a@(Tel _ | Email _)] = a

maybeSingleTel :: [ContactMode]
               -> Either ContactMode [ContactMode]
maybeSingleTel [a@(Tel _ | _*)] = a
```

Similarly for optional patterns we use another built-in Haskell type:

```
data Maybe a = Nothing | Just a
```

so if we write a function

```
singleOrNoTel [(Email _)*,a@(Tel _)?,(Email _)*] = a
```

it will have the type `[ContactMode] -> Maybe ContactMode`.

One way to think about this is to see the regular expression pattern operators as special data constructors. In an analogy with ordinary Haskell, we don't expect `a` to have the same type in the two uses `a@(Just _)` and

`(Just a@_)`. Nor do we expect the `a` in `a@(_?)` to have the same type as the `a` in `(a@_)?`.

The second issue concerns linear vs. non-linear binding. We have already seen that implicit bindings, i.e bindings that arise from the use of ordinary pattern variables, are context dependent; in linear context they get the ordinary types, whereas in non-linear context they get list types. This context dependence unfortunately makes it easy for the programmer to make mistakes, since it isn't clear just by looking at a variable in the pattern what type it will have. We cannot do anything about implicit bindings, but we can avoid the same problem for explicit binding. Therefore we let the ordinary `@` operator signify linear explicit binding, the only kind available in ordinary Haskell. For non-linear explicit binding we introduce a new operator `@:` (read "as cons" or "accumulating as"). The former may not appear in non-linear context, whereas the latter may appear anywhere inside a regular expression pattern. Their differences are shown by the following examples:

```
[a@(Tel _)   , _*]  => a :: ContactMode
[a@(Tel _)* , _*]  => a :: [ContactMode]
[(Tel a@_)   , _*]  => a :: TelNr
[(Tel a@_)* , _*]  => Not allowed!
[(Tel a@:_)*, _*]  => a :: [TelNr]
```

We can define the semantics of implicit bindings in terms of explicit bindings. In linear context we have that a pattern variable `a` is equivalent to the pattern `a@_`. This can be seen in the example `[(Tel a), _*]` which is clearly equivalent to `[(Tel a@_), _*]`. In non-linear context, `a` is equivalent to `a@:_`, as in the examples `[(Tel a)*, _*]` and `[(Tel a@:_)*, _*]`.

2.8 Further examples

Now that we've seen all the basic building blocks that our regular expression patterns consist of, let us put them to some real use.

Traditionally regular expressions have been used in programming languages for text matching purposes, and certainly our regular expression patterns are well suited for this task. As an example, assume we have a specification of a simple options file. An option has a name and a value, written on a single row, where name and value are separated with a colon and a whitespace. Different options are written on different lines. Here are the contents of a sample options file:

```
author: Niklas Broberg
author: Andreas Farre
author: Josef Svenningsson
title: Regular Expression Patterns
submitted: ICFP 2004
```

A simple parser for such option files can be written using a regular expression pattern as

```
parseOptionFile :: String -> [(String,String)]
parseOptionFile
 [(/ names@:_*, ':', ' ', vals@:_*, '\n' /)*]
   = zip names vals
```

where `zip` is a function that takes two lists and groups the elements pair-wise.

XML processing is another area that greatly benefits from regular expressions, since "proper pattern matching on XML fragments requires ... matching of regular expressions" [14]. Indeed several recent XML-centric languages (XDuce, CDuce) include regular expressions as part of their pattern matching facilities.

As an example we encode XML in Haskell using a simple datatype

```
data XML = Tag String [XML]
         | PCDATA String
```

An XML fragment is either a `Tag`, e.g. `<P> ... </P>`, which has a name (a `String`) and a list of `XML` children, or it is `PCDATA` (XML lingo for a string inside tags). This model is of course extremely simplified, we've left out anything that will not directly add anything to our example, most notably XML attributes. Now assume that we have a simple XML email format, where a sample email message in this format might look like:

```
<MSG>
 <FROM>d00nibro@dtek.chalmers.se</FROM>
 <RCPTS>
  <TO>d00farre@dtek.chalmers.se</TO>
  <TO>josefs@cs.chalmers.se</TO>
 </RCPTS>
 <SUBJECT>Regular Expression Patterns</SUBJECT>
 <BODY>
  <P>Regular expression patterns are useful</P>
 </BODY>
</MSG>
```

which would be encoded in our `XML` datatype as

```
Tag "MSG" [
 Tag "FROM" [PCDATA "d00nibro@dtek.chalmers.se"],
 Tag "RCPTS" [
  Tag "TO" [PCDATA "d00farre@dtek.chalmers.se"],
  Tag "TO" [PCDATA "josefs@cs.chalmers.se"]
 ],
 Tag "SUBJECT"
  [PCDATA "Regular Expression Patterns"],
 Tag "BODY" [
  Tag "P"
   [PCDATA "Regular expression patterns are useful"]
 ]
]
```

We can write a function to convert messages from this XML format into the standard RFC822 format using regular expression patterns:

```
xmlToRfc822 :: XML -> String
xmlToRfc822
  (Tag "MSG" [
     Tag "FROM" [PCDATA from],
     Tag "RCPTS" [
       (Tag "TO" [PCDATA tos])+
     ],
     Tag "SUBJECT" [PCDATA subject],
     Tag "BODY" [
       (Tag "P" [PCDATA paras])*
     ]
   ]) = concat
          ["From: ", from, crlf,
           "To: ", concat (intersperse ", " tos),
           crlf,
           "Subject: ", subject, crlf, crlf,
           concat (intersperse crlf paras), crlf]
  where crlf = "\r\n"
```

3 Syntax

The previous section has gone over all of regular expression patterns by example. This section starts the formal treatment by giving a grammar for

the syntax, which can be seen in figure 1. We refer to the nonterminal for Haskell's ordinary patterns as *pattern* and extend it with a new production for regular expression patterns.

$$
\begin{array}{lll}
\mathit{pattern} & \rightarrow & \ldots \\
 & | & \text{'['}\ \mathit{regpat}_1\ \ldots\ \mathit{regpat}_n\ \text{']'} \\
 & & \\
\mathit{regpat} & \rightarrow & \mathit{pattern} \\
 & | & \mathit{regpat}\ \text{'*'}[\text{'!'}] \\
 & | & \mathit{regpat}\ \text{'+'}[\text{'!'}] \\
 & | & \mathit{regpat}\ \text{'?'}[\text{'!'}] \\
 & | & \mathit{regpat}\ \text{`|`}\ \mathit{regpat} \\
 & | & \text{'(/'}\ \mathit{regpat}_1 \ldots \mathit{regpat}_n\ \text{'/)'} \\
 & | & \text{'('}\ \mathit{regpat}\ \text{')'} \\
 & | & \mathit{var}\ \text{'@'}\ \mathit{regpat} \\
 & | & \mathit{var}\ \text{'@:'}\ \mathit{regpat}
\end{array}
$$

Figure 1: Regular expression pattern syntax

The concrete syntax is quite close to that of e.g. Perl [15] or CDuce [2] with the notable exception that we have non-greedy patterns as default. An extra exclamation mark indicates greediness.

Ordinary Haskell patterns are regular expressions patterns. The operators are repetition (*), non-empty repetition (+) and option (?). Furthermore there are choice patterns indicated by a vertical bar and subsequences are enclosed in subsequence brackets. Regular expression patterns can be enclosed in parenthesis. The last two productions are for linear and non-linear variable bindings. Precendence of the operators is as follows: *, +, ?, *!, +! and ?! binds strongest. They are followed by choice patterns which are also right associative. Lastly we have @ and @: which bind weakest. All constructs in regular expression patterns bind stronger than constructor application.

4 Semantics

In this section we turn to the formal semantics for regular expression patterns. Our semantics divides natually into two parts; one for linear and one for non-linear patterns. The reason for this division is that variable bindings are treated differently.

4.1 Structure of semantics

We give the semantics as an all-match semantics. This leads to possibly ambiguous matches, the same list can be matched in many different ways. Since this may affect how variables are bound to their values we need to disambiguate our rules. We follow the approach taken by Hosoya and Pierce [7] and introduce an ordering on the rules indicating which rule will have precedence when several rules can match. The order is given by numbers in the name of the rules, where lower numbers have higher precedence. Intuitively this means that when building the derivation tree for a match, one must always try to use the rule with the highest precedence first, and choose the other rule only if choosing the first rule cannot lead to a match.

Before we begin with the semantics we will define some concepts which will be used in our explanation of the semantics. We will use sets of *variable bindings* to map variables to values. A variable binding is denoted $x \mapsto v$. In repetition patterns we will need to merge sets of variable bindings with overlapping domains. We use $\uplus$ to this end and define it as follows:

$$\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \uplus \{x_1 \mapsto vs_1, \ldots, x_n \mapsto vs_n\} = \{x_1 \mapsto v_1 +\!\!+ vs_1, \ldots, x_n \mapsto v_n +\!\!+ vs_n\}$$

When giving a semantics for subsequence patterns we will use a type indexed function *flatten* to merge lists of values. It is defined as follows:

$$flatten_T(v) = [v]$$
$$flatten_{[\tau]}([]) = []$$
$$flatten_{[\tau]}(v, vs) = flatten_\tau(v) +\!\!+ flatten_{[\tau]}(vs)$$
$$flatten_{Maybe\,\tau}(Nothing) = []$$
$$flatten_{Maybe\,\tau}(Just\,v) = flatten_\tau(v)$$
$$flatten_{Either\,\tau_1\tau_2}(Left\,v) = flatten_{\tau_1}(v)$$
$$flatten_{Either\,\tau_1\tau_2}(Right\,v) = flatten_{\tau_2}(v)$$

We will refer to the set of bound variables in a pattern p as $vars(p)$.

4.2 Semantics for linear patterns

The semantics for linear regular expression patterns can be found in figure 2. Due to space reasons we only give a few of the rules as we explain below.

The judgement for matching linear patterns is denoted $l \in_l p \to v; \beta; l'$. It should read as "$l$ is matched by a pattern p yielding a value v, a set of variable bindings β, and a remainder list l' ". l and l' range over Haskell lists, where l is the list we wish to match and l' is a (possibly empty) suffix of l that wasn't matched.

First of all we have a rule HM-RegPat that extends Haskell's pattern matching semantics, denoted $\in_h$, with regular expression patterns. It does

$$\text{LM-Base}\frac{e \in_h \pi \to \beta}{e : l \in_l \pi \to e; \beta; l}$$

$$\text{LM-As}\frac{l_1 \in_l p \to v_1; \beta_1; l_2}{l_1 \in_l x@p \to v_1; \{x \mapsto v_1\} \cup \beta_1; l_2}$$

$$\text{LM-AccAs}\frac{l_1 \in_l p \to v_1; \beta_1; l_2}{l_1 \in_l x@ : p \to v_1; \{x \mapsto [v_1]\} \cup \beta_1; l_2}$$

$$\text{LM-Seq}\frac{l_1 \in_l p_1 \to v_1; \beta_1; l_2 \quad \ldots \quad l_n \in_l p_n \to v_n; \beta_n; l_f}{l_1 \in_l (/p_1 \ldots p_n/) \to \gamma_1 +\!\!+ \ldots +\!\!+ \gamma_n; \beta_1 \cup \cdots \cup \beta_n; l_f}$$
$$where\ \gamma_i = flatten_\tau(v_i), p_i : \tau$$

$$\text{LM-Star}\frac{l_1 \in p^* \to v, \beta, l_2}{l_1 \in_l p^* \to v; \beta, l_2}$$

$$\text{HM-RegPat}\frac{l \in_l (/p_1 \ldots p_n/) \to l; \beta; []}{l \in_h [p_1 \ldots p_n] \to \beta}$$

Figure 2: Semantics for linear regular expression patterns

so by performing a linear match.

$$\frac{l \in_l (/p_1 \ldots p_n/) \to l; \beta; []}{l \in_h [p_1 \ldots p_n] \to \beta}$$

Here we require that the remainder list is empty i.e. that the whole input list is succesfully matched. This requirement together with the ordering on the rules determines which derivation must be chosen.

The base rule, LM-Base, is that where the pattern to match is a normal Haskell pattern. In this case we piggy-back on Haskell's normal mechanism for binding variables from patterns.

$$\frac{e \in_h \pi \to \beta}{e : l \in_l \pi \to e; \beta; l}$$

Apart from ordinary Haskell patterns there are two ways that we can bind variables to values at toplevel, given by the rules LM-As and LM-AccAs. The `@` operator simply binds the variable to a value, whereas the `@:` operator binds the variable to a list containing the value. The behavior of `@:` clearly

makes more sense in a non-linear context, where the number of bound values may vary, but since it is harmless to do so we have chosen to allow it to appear in linear contexts as well.

For subsequences we simply match each pattern in the sequence in order, as stated by the rule LM-Seq. The values produced after matching are concatenated and the resulting disjoint sets of variable bindings are merged. The value yielded by matching a subsequence should always be a list of elements, so before we can concatenate the values of the sub-matches we need to flatten these values to simple lists. Here we need to use the typing relation on patterns defined in section 5. The typing relation is defined relative to some base type T that during the actual matching will be instantiated to the type of the elements in the matching list.

Matching a non-linear pattern in a linear context is identical to matching it in a non-linear context. This is exemplified by the rule LM-Star. The rules for the rest of the operators are similar and are left out due to space restrictions.

4.3 Semantics for non-linear patterns

The relation for matching in a non-linear context, denoted $l \in p \to v; \beta; l'$ (the only difference in syntax is that we drop the subscript on $\in$), is similar to the relation for linear contexts. It differs in two crucial aspects, namely variable bindings and that we handle non-linear patterns. The rules can be found in figures 3 and 4, split into two for space reasons.

The base rule M-Base is once again that where the pattern to match is an ordinary Haskell pattern. Since the matching now takes place in a non-linear context, the values of variables being bound while matching this pattern are put into lists instead of just being bound outright. Binding variables explicitly in a non-linear context can only be done using the `@:` (accumulating as) operator that binds its variable argument to a list of the value matching its pattern argument, as shown in the rule M-AccumAs.

The rule for matching a subsequence, M-Seq, is identical to LM-Seq except that subpatterns in the sequence are also matched in a non-linear context.

The rules for a repetition pattern, M-Star1 and M-Star2, give a non-greedy semantics to the operator by giving the rule for not matching higher precedence than the rule for actually matching the subpattern. The first rule simply doesn't try to match anything, whereas the second rule matches the given subpattern p once and then recurses to obtain more matches. The value obtained from matching p is then prepended to the result values of the recursive second premise. Similarly the values of bound values are prepended

$$\text{M-Base}\frac{e \in_h \pi \to \beta}{e : l \in \pi \to e; \sigma; l}$$
$$where\ \sigma = \{x \mapsto [v] |\ x \mapsto v \in \beta\}$$

$$\text{M-AccAs}\frac{l_1 \in p \to v_1; \beta_1; l_2}{l_1 \in x@{:}p \to v_1; \{x \mapsto [v_1]\} \cup \beta_1; l_2}$$

$$\text{M-Seq}\frac{l_1 \in p_1 \to v_1; \beta_1; l_2 \quad \ldots \quad l_n \in p_n \to v_n; \beta_n; l_f}{l_1 \in (/p_1 \ldots p_n/) \to \gamma_1 +\!\!+ \ldots +\!\!+ \gamma_n; \beta_1 \cup \cdots \cup \beta_n; l_f}$$
$$where\ \gamma_i = flatten_\tau(v_i), v_i :: \tau$$

$$\text{M-Star1}\frac{}{l \in p^* \to []; \beta; l}$$
$$where\ \beta = \{x \mapsto [] |\ x \in vars(p)\}$$

$$\text{M-Star2}\frac{l_1 \in p \to v_1; \beta_1; l_2 \quad l_2 \in p^* \to v_2; \beta_2; l_3}{l_1 \in p^* \to v_1 : v_2; \beta_1 \uplus \beta_2; l_3}$$

$$\text{M-GStar1}\frac{l_1 \in p \to v_1; \beta_1; l_2 \quad l_2 \in p^*! \to v_2; \beta_2; l_3}{l_1 \in p^*! \to v_1 : v_2; \beta_1 \uplus \beta_2; l_3}$$

$$\text{M-GStar2}\frac{}{l \in p^*! \to []; \beta; l}$$
$$where\ \beta = \{x \mapsto [] |\ x \in vars(p)\}$$

Figure 3: Semantics for non-linear regular expression patterns (i)

to the bindings from the recursive call. To get a greedy semantics in the rules M-GStar1 and M-GStar2 we simply swap the order of the rules to give precedence to preforming a match.

The non-empty repetition pattern operator p^+ is defined as $p^+ \equiv pp^*$, similarly its greedy counterpart $p^+! \equiv pp^*!$, and the rules M-Plus and M-GPlus can easily be derived from these facts.

The rules M-Opt1 and M-Opt2 for optional patterns are very similar to the rules for repeating patterns, only that no recursion to obtain more matches is done. The values returned by an optional pattern are of the Haskell `Maybe` type for optional values.

For choice regular expression patterns we return values of the Haskell

$$\text{M-Plus}\frac{l_1 \in p \to v_1, \beta_1, l_2 \quad l_2 \in p^* \to v_2, \beta_2, l_3}{l_1 \in p^+ \to v_1 : v_2, \beta_1 \uplus \beta_2, l_3}$$

$$\text{M-GPlus}\frac{l_1 \in p \to v_1, \beta_1, l_2 \quad l_2 \in p^*! \to v_2; \beta_2; l_3}{l_1 \in p^+! \to v_1 : v_2, \beta_1 \uplus \beta_2, l_3}$$

$$\text{M-Opt1}\frac{}{l \in p? \to Nothing, \beta, l} \; where\; \beta = \{x \mapsto [] | \; x \in vars(p)\}$$

$$\text{M-Opt2}\frac{l_1 \in p \to v_1, \beta_1, l_2}{l_1 \in p? \to (Just\, v_1), \beta_1, l_2}$$

$$\text{M-GOpt1}\frac{l_1 \in p \to v_1, \beta_1, l_2}{l_1 \in p?! \to (Just\, v_1), \beta_1, l_2}$$

$$\text{M-GOpt2}\frac{}{l \in p?! \to Nothing, \beta, l} \; where\; \beta = \{x \mapsto [] | \; x \in vars(p)\}$$

$$\text{M-Choice1}\frac{l_1 \in p_1 \to v_1; \beta; l_2}{l_1 \in (p_1|p_2) \to (Left\, v_1); \sigma; l_2}$$
$$where\; \sigma = \beta \cup \{x \mapsto [] | \; x \in vars(p_2)\} \setminus vars(p_1))$$

$$\text{M-Choice2}\frac{l_1 \in p_2 \to v_1; \beta_1; l_2}{l_1 \in (p_1|p_2) \to (Right\, v_1); \beta; l_2}$$
$$where\; \beta = \beta_1 \cup \{x \mapsto [] | \; x \in vars(p_1)\} \setminus vars(p_2))$$

Figure 4: Semantics for non-linear regular expression patterns (ii)

`Either` type to indicate which choice was taken. In the rules M-Choice1 and M-Choice2 we give precedence for matching the left pattern. Furthermore all variables occuring only in the branch not taken are assigned empty lists.

5 Well-formed regular expression patterns

We now turn our attention to the static semantics of regular expression patterns. We will refer to the static semantics as well-formedness of regular expression patterns.

There are two reasons why we need a static semantics. The first reason

concerns where and how a variable is bound in a pattern. In ordinary patterns a variable may appear only once, with the notable exception for *or*-patterns found in Ocaml and SML/NJ. In these languages all alternatives must bind exactly the same set of variables. We have similar yet more liberal restrictions on variable bindings. Bound variables must not necessarily be bound in all alternatives in a choice pattern.

The second reason is that we need to ensure that the types of the bound variables are correct. The same variable should in particular have the same type for all its occurrences in a choice pattern.

To express the well-formedness of a regular expression pattern we use the judgment $\Delta \vdash_l p$ which says that a (linear) regular expression pattern p is well-formed in the typing context Δ. The typing context Δ gives types to the variables bound in the pattern. When checking the validity of patterns in a non-linear context we use the judgment $\Delta \vdash p$ which is similar to the judgment for linear patterns. We will also refer to the well-formedness of patterns in Haskell, using the judgment $\Delta \vdash_h p$. We refer to Faxén's paper for a static semantics of Haskell patterns [4]. We require that $\Delta \vdash_h p$ can only be derived if p binds exactly the variables in the typing context Δ. Finally we will need a notion of types for regular expression patterns. We use the judgment $p :: \tau$ to say that the pattern p has the type τ.

Checking the well-formedness of a regular expression pattern as an ordinary pattern in the host language is done using the following rule. Is is noteworthy that we split the typing context. All the typing contexts Δ_i must bind different names. We use this to enforce that a variable may only be bound once.

$$\frac{\Delta_1 \vdash_l p_1 \ \ldots \ \Delta_n \vdash_l p_n}{\Delta_1 \ldots \Delta_n \vdash_h [p_1 \ldots p_n]} \ \Delta_i \cap \Delta_j = \emptyset \ \forall ij.i \neq j$$

The rules for establishing well-formedness of linear patterns can be found in figure 5. In this section we only present the rules for non-greedy operators as the rules for greedy counterparts are exactly the same. The only interesting thing to note about the rules for $*$, $+$ and ? is the fact that when checking their subpatterns we are in a non-linear context and therefore use the corresponding judgment for the premises. The rule for sequences is reminiscent of that for regular expression patterns in the context of ordinary patterns explained above.

The variable binding rules are interesting to contrast against each others. "As"-patterns are well-formed if the variable is bound to a pattern with the same type as the variable. "Accumulating as"-patterns on the other hand may match several times so the type of the variable must be a list.

$$\frac{\Delta \vdash p}{\Delta \vdash_l p^*} \quad \frac{\Delta \vdash p}{\Delta \vdash_l p^+} \quad \frac{\Delta_1 \vdash p \quad \Delta_2 \vdash q}{\Delta \vdash_l p|q} \Delta = \Delta_1 \cup \Delta_2 \quad \frac{\Delta \vdash p}{\Delta \vdash_l p?}$$

$$\frac{\Delta_1 \vdash p_1 \ \ldots \ \Delta_n \vdash p_n}{\Delta_1 \ldots \Delta_n \vdash_l (/p_1 \ldots p_n/)} \Delta_i \cap \Delta_j = \emptyset \ \forall ij.i \neq j$$

$$\frac{p :: \tau \quad \Delta \vdash_l p}{\Delta, x :: \tau \vdash_l x@p} \quad \frac{p :: \tau \quad \Delta \vdash_l p}{\Delta, x :: [\tau] \vdash_l x@:p} \quad \frac{\Delta \vdash_h hpat}{\Delta \vdash_l hpat}$$

Figure 5: Wellformed linear regular expression patterns

$$\frac{\Delta \vdash p}{\Delta \vdash p*} \quad \frac{\Delta \vdash p}{\Delta \vdash p+} \quad \frac{\Delta_1 \vdash p \quad \Delta_2 \vdash q}{\Delta \vdash p|q} \Delta = \Delta_1 \cup \Delta_2$$

$$\frac{\Delta \vdash p}{\Delta \vdash p?} \quad \frac{\Delta_1 \vdash p_1 \ \ldots \ \Delta_n \vdash p_n}{\Delta_1 \ldots \Delta_n \vdash (/p_1 \ldots p_n/)} \Delta_i \cap \Delta_j = \emptyset \ \forall ij.i \neq j$$

$$\frac{p :: \tau \quad \Delta \vdash p}{\Delta, x :: [\tau] \vdash x@:p} \quad \frac{\Delta' \vdash_h hpat}{\Delta \vdash hpat}$$

$$where \ \Delta = \{x :: [\tau] | x :: \tau \in \Delta'\}$$

Figure 6: Wellformed regular expression patterns

In figure 6 we present the rules for establishing the well-formedness of non-linear patterns. Most of the rules carry over straightforwardly from those for linear patterns. It should be noted though that the rule for ordinary patterns rebuilds the typing context so that all variables have list types.

Figure 7 gives the typing rules for regular expression patterns. The intuition behind these rules is that a pattern has a type which reflects the ways it can match. For example a pattern which can match many times has a list type, hence variables bound to $*$ and $+$ patterns get list types. Choice patterns can match one of two things which is captured by the `Either` type of Haskell. A sequence pattern matches yields a sequence and hence it also has a list type. Variable binding patterns don't affect the typing. The last typing rule for ordinary patterns in the underlying language is more surprising, since it refers to a specific type `T`. This means that the typing rules should

$$\frac{p :: \tau}{p^* :: [\tau]} \quad \frac{p :: \tau}{p^+ :: [\tau]} \quad \frac{p :: \tau \quad q :: \tau'}{p|q :: Either\ \tau\ \tau'} \quad \frac{p :: \tau}{p? :: Maybe\ \tau}$$

$$\frac{p_1 :: \tau_1 \ \ldots \ p_n :: \tau_n}{(/p_1 \ldots p_n/) :: [T]} \quad \frac{p :: \tau}{x@p :: \tau} \quad \frac{p :: \tau}{x@{:}p :: \tau} \quad \frac{}{hpat :: T}$$

Figure 7: Typing rules for regular expression patterns

be interpreted in a context where we are matching on a list of type `[T]`, i.e. `T` is the type of the elements of the list.

6 Implementation

We currently have an implementation of our regular expression pattern system that works as a preprocessor for GHC. It takes a source code file possibly containing regular expression patterns and translates it into semantically equivalent vanilla Haskell code. It also comes with a matching engine, which we implement as a simple parser monad. The preprocessor does not check any types, instead we rely on GHC's type checker to catch type errors.

6.1 Matching engine

The datatype for a matching parser, which we from now on will refer to as a matcher, looks like

```
data Matcher e a = Matcher ([e] -> [(a,[e])])
```

It is essentially a function that takes an input list, conducts a match, and returns a list of results. Each result will consist of a value, a set of values for bound variables, and a remainder list. All of this is read directly from our semantic rules.

Since different variables will be bound to values of different types, we need to model the set of bindings as a tuple, with each entry corresponding to the value(s) for one specific variable. As is customary, we let the remainder list be the state of the matcher monad, so that it is implicitly threaded through a series of matches. The individual matcher functions then need to return a value for future bindings, and a tuple with values for variables.

To account for our all-match semantics the parser generates a list of results at each step. At places where we need to branch we can use the `+++`

operator which lets us proceed with two different matchers. We define `+++` as

```
(+++) :: Matcher e a -> Matcher e a -> Matcher e a
(Matcher f) +++ (Matcher g) =
  Matcher (\es -> let aes1 = f es
                      aes2 = g es
                  in aes1 ++ aes2)
```

As we can see from the definition `+++` is left-biased, i.e. any results from its left operand will end up before any results from its right operand in the list of results. This allows us to define a function that conducts the full matching by, as defined by our first-match policy, selecting the first result in this list of results for which the matcher has reached the end of the input list (i.e. the remainder list is empty). This function, called `runMatch`, corresponds to the rule HM-RegPat from figure 2, and is defined as

```
runMatch :: Match e a -> [e] -> Maybe a
runMatch (Matcher f) es =
   let allps = f es
       allMatches = filter (null . snd) allps
    in case allMatches of
        []                -> Nothing
        (((_, vars),_):_) -> Just vars
```

6.2 Translation

The basic idea behind translating a regular expression pattern into vanilla Haskell is to generate a matcher for each subpattern, all the way down to ordinary Haskell patterns, and then combine these to form a top-level matcher corresponding to the whole of the pattern.

6.2.1 Base patterns

The base case is when the pattern in question is an ordinary Haskell pattern. First we must generate a function that actually takes an element from the input list and tries to match it to the given pattern. For example, if the pattern in question is `Tel nr`, the corresponding function would look like

```
match0 :: CMode -> Maybe TelNr
match0 e = case e of
            Tel nr -> Just (nr)
            _ -> Nothing
```

No type signatures are actually generated, we just supply them here to simplify understanding. To avoid overly long signatures we abbreviate `ContactMode` with `CMode` in our examples.

What the function returns if the match succeeds is a tuple containing the values of bound variables. The function above works in linear context since we return the bound variable as is. If we instead wanted a function to work in non-linear context, we would wrap the values in lists, like

```
match0 :: CMode -> Maybe [TelNr]
match0 e = case e of
            Tel nr -> Just ([nr])
            _ -> Nothing
```

We also need to lift a generated matching function into the matcher monad. This lifting works identically regardless of what the pattern is, so we have a function in the matcher engine that does this, defined as

```
baseMatch :: (e -> Maybe a) -> Matcher e (e,a)
baseMatch matcher = do
   e <- getElement
   case matcher e of
    Nothing -> mfail
    Just b -> do discard
                 return (e, b)
```

The functions used by `baseMatch` are inherent to our matcher monad. `getElement` retreives the head of the input list, `discard` drops the head of the input list, and `mfail` is a matcher that always returns an empty list of results. We now need to generate a matcher by applying `baseMatch` to our generated function, i.e.

```
match1 :: Matcher CMode (CMode, TelNr)
match1 = baseMatch match0
```

The type states that `match1` is a matcher for a list of `CMode`s. The value matched is a `CMode`, and the only variable bound is of type `TelNr`. The numbers 0 and 1 in the names of these functions signify that each name is fresh, i.e. these numbers could be any positive integers, but no two functions share the same integer.

For Haskell patterns that are guaranteed to always match, i.e. pattern variables and wildcards (_), we can simplify these steps. For a wildcard, what we need to generate is the matcher

```
match0 :: Matcher e (e, ())
match0 = baseMatch (\_ -> Just ())
```

meaning we will always match, and no variables are bound. The only difference for a pattern variable is that the variable in question is also bound, e.g. for the pattern `a` we get

```
match0 :: Matcher e (e, e)
match0 = baseMatch (\a -> Just (a))
```

Once again the shown function works in linear context, in non-linear context we would wrap the returned `a` in a list.

6.2.2 Repetition

All regular expression patterns have one or more subpatterns, and the first step when translating a regular expression pattern will be to translate these subpatterns. For a repetition pattern, $p*$, we would first translate the subpattern p into some matcher function `matchX`. According to the rules M-Star1 and M-Star2, a matcher for a repetition pattern should if possible continue without trying to match anything, otherwise it should match one element and then recursively match the repetition pattern again. This behavior is common to all repetition patterns so we define it as a function in the matching engine:

```
manyMatch :: Matcher e a -> Matcher e [a]
manyMatch matcher = (return []) +++
                    (do a <- matcher
                        as <- manyMatch matcher
                        return (a:as))
```

The problem with this definition is that `manyMatch` returns a list in which each element is the result of one step of the recursion. We need to unpack this list so that we instead return a tuple, in which each entry is a list of results for a specific variable binding. We cannot do this generically since the number of bound variables, and thus the size of the tuple, will vary. Therefore we must supply an appropriate unzipping function that works for the correct number of variables. The exact function to use can be determined by the preprocessor, that has the necessary meta-information on what variables are bound. Note that all variables inside the repetition will be non-linear, so the result of matching a variable in each step of the recursion will be a list of values. If we only unzip to get a list of such results for each variable, what we would really get is a list of lists of values. Thus to get a list of values

we should also let the unzipping function concatenate the results for each variable in the resulting tuple.

Inside `manyMatch` the unpacking will be done in two steps. The first is to simply unzip the list into two lists, one containing all values (v_i from the rules), the other containing all values of bound variables. In the second step we need to apply the supplied unzipping-and-concatenating function to the latter list to get the variable values proper. This new improved `manyMatch` will thus look like

```
manyMatch :: Matcher e (a,b) -> ([b] -> c)
               -> Matcher e ([a], c)
manyMatch matcher unzipper = do
   res <- mMatch matcher
   let (vals, vars) = unzip res
       vs = unzipper vars
   return (vals, vs)
```

where `mMatch` is our old definition of `manyMatch`.

As an example, we show the translation of the pattern `(Tel nr)*`. The first step is to translate the subpattern `Tel a`, which we have already seen how to do. The new function that we generate will then look like

```
match2 :: Matcher CMode ([CMode],[TelNr])
match2 = manyMatch match1 unzip1
```

assuming the matcher for the subpattern is called `match1`. The function `unzip1` here is simply the `concat` function, since there is only one variable bound. To account for the greedy version of a repetition pattern, `*!`, we simply flip the arguments to `+++` in `manyMatch`, which will give a higher priority to the case when we actually match an element.

Non-empty repetition patterns, `+`, are very similar to ordinary repetition patterns, the only difference is of course that we make an initial match before starting the recursion, as shown in

```
neManyMatch :: Matcher e (a,b) -> ([b] -> c)
                 -> Matcher e ([a], c)
neManyMatch matcher unzipper = do
   res1 <- matcher
   res <- mMatch matcher
   let (vals, vars) = unzip (res1:res)
       vs = unzipper vars
   return (vals, vs)
```

6.2.3 Choice and Optional patterns

Choice patterns are slightly trickier to handle because of the way variables are bound. As we saw in the rules M-Choice1 and M-Choice2, any variables appearing in the other branch than the one being matched should be bound to empty lists. This is very difficult to handle generically since we need access to the meta-information of variable names. Thus we instead generate the full code for the choice pattern during translation. As an example we translate the pattern `(Tel nr | Email eaddr)`. We start by translating the subpatterns, resulting in two functions that we assume are named `match1` and `match2`. The code generated for the choice pattern will be

```
match3 :: Matcher CMode
           (Either CMode CMode, ([TelNr],[EAddr]))
match3 = (do (val, (a)) <- match1
             return (Left val, (a, [])))
     +++ (do (val, (b)) <- match2
             return (Right val, ([], b)))
```

where we have tagged the result value of the pattern match with the respective constructors from the `Either` type. The story is very similar for optional patterns, but this time all variables should be bound to empty lists if no match is done. For the pattern `(Tel nr)?` we get

```
match4 :: Matcher CMode (Maybe CMode, [TelNr])
match4 = (return (Nothing, [])) +++
         (do (val, (a)) <- match1
             return (Just val, a))
```

For a greedy optional pattern we would simply switch the arguments to `+++`, just as for repetition patterns.

6.2.4 Subsequences

The trickiest pattern to implement is subsequence, due to the need for flattening. As we saw in section 5, flattening is done based on the type of a subpattern (with respect to some base type for elements in the input list), which means that the preprocessor must keep track of these types in order to insert the proper flattening functions. For a pattern `(/ (Tel nr)?, (Email eaddr)* /)` we get the following translation, assuming the two subpatterns are translated into matcher functions `match1` and `match2` respectively:

```
match5 :: Matcher CMode ([CMode], ([TelNr],[EAddr]))
match5 = do (v1, (a)) <- match1
            (v2, (b)) <- match2
            let v1f = maybe [] (\v -> [v]) v1
                v2f = concatMap (\v -> [v]) v2
            return (v1f ++ v2f, (a,b))
```

The value `v1` is the result of `match1`, i.e. the matcher for `(Tel nr)?`, so it will have type `Maybe CMode`. To flatten it we use the built-in Haskell function `maybe` that takes two arguments, one that is a default value to return if it encounters a `Nothing` (in this case `[]`), the other a function to apply to a value held by a `Just` (in this case the flattening function for a value of the base type). Similarly `v2` comes from `match2`, so its type will be `[CMode]`. We flatten it using the built-in function `concatMap` that takes a function, applies is to all elements of a list, and then concatenates the results.

6.2.5 Variable bindings

Finally we turn to the explicit binding operators. Binding a variable to a value in our matcher means to add that value to the result tuple. Since an explicitly bound variable syntactically appears to the left of any variables in its subpattern, we add the value in the leftmost position in the tuple, i.e. before those bound in the subpattern. Thus we know that the values in the result of the top-level matcher should be bound to variables from left to right in the order they appear in the pattern. As an example consider the pattern `a@(Tel nr | Email eaddr)`. We first translate the subpattern `(Tel nr | Email eaddr)` into a matcher `match1`. The matcher generated for the variable binding will then be

```
match2 :: Matcher CMode (Either CMode CMode,
             (Either CMode CMode,[TelNr],[Eaddr]))
match2 = do (val, (nr, eaddr)) <- match1
            return (val, (val, nr, eaddr))
```

If we had instead used non-linear binding, i.e. `a@:(Tel nr | Email eaddr)`, we would get a list for the returned value, i.e.

```
match2 :: Matcher CMode
           (Either CMode CMode,
             ([Either CMode CMode],[TelNr],[Eaddr]))
match2 = do (val, (nr, eaddr)) <- match1
            return (val, ([val], nr, eaddr))
```

6.3 Matching

Now we know how to translate a regular expression pattern into a top-level matcher function, what is left is to insert and invoke the generated matcher at the right place to preserve the pattern matching semantics. To this end we use Haskell pattern guards [3] that allow us to evaluate a function and pattern match on the result as part of the original pattern match. The function that we so wish to evaluate is `runMatch` applied to our generated top-level matcher and the input list that we wish to match. For our matcher functions to be in scope we add them to the `where` clause of the declaration that the regular expression pattern appears in. To show a complete example of the translation of a function declaration we revisit our function `allTels` defined as

```
allTels (Person _ [(Tel nr | _)*]) = nr
```

since it contains several different features of regular expression patterns. The translated version of this function will look like

```
allTels (Person _ arg0)
    | Just (nr) <- runMatch match5 arg0 = nr
 where match0 e = case e of
                    Tel nr -> Just ([nr])
                    _ -> Nothing
       match1 = baseMatch match0
       match2 = baseMatch (\_ -> Just ())
       match3 = (do (val, (nr)) <- match1
                    return (Left val, (nr)))
                +++
                (do (val, ()) <- match2
                    return (Right val, ([])))
       match4 = manyMatch match3 unzip1
       match5 = do (v1, (nr)) <- match4
                   let v1f = concatMap
                             (either (\v -> [v])
                                     (\v -> [v]))
                             v1
                   return (v1f, (nr))
```

The functions `match0` and `match1` together correspond to the pattern `(Tel nr)`. Note the list around the returned variable `nr` signaling that the pattern is matched in a non-linear context. `match2` corresponds to the pattern `_`. Combining these two into a choice patterns yields `(Tel nr | _)`, which

is translated to `match3`. On top of that we add a repetition, which gives us `match4` when translated. Finally since the top-level pattern should be matched as a subsequence, as seen in the rule HM-RegPat, we translate it into `match5`. The actual matching is done in the pattern guard that applies `runMatch` to the matcher and the input list. The latter is held by an automatically generated fresh variable, in this case `arg0`. It is also interesting to note that the actual binding of variables to values does not happen until `runMatch` is evaluated. Any mention of variable names in the matcher functions, e.g. `nr` in `match0`, are only there as mnemonic aids to a human reader. We could change all such names to freshly generated variable names without changing any semantics.

In Haskell, patterns can appear in numerous places such as function declarations, case expressions, let expressions, statements etc. Translating regular expression patterns into vanilla Haskell is slightly different depending on just where the pattern appears. The generated matchers will be identical in all cases, but the placement of them and of the evaluation may differ. We will not go through these differences in detail, but our implementation handles all cases correctly. Irrefutable (lazy) patterns also require special care, and we have yet to implement support for them in full.

7 Related Work

Pattern matching is a well-known and much studied feature of functional languages [1, 17, 11, 12]. It provides the startingpoint for the work presented in this paper.

Regular expressions have been used in programming for a long time, mostly for text matching purposes. Perl's support for regular expressions is probably one of the most well-known [15], but most mainstream languages, including Haskell, have some library support for regular expression text matching. Regular expressions in such libraries are themselves encoded as strings. Matching them means taking two strings, where one encodes a regular expression, and match them to each other. This is in some sense very low-level when compared to our regular expression patterns since there are no guarantees that regular expressions encoded as strings are well-formed, and there is no direct way to bind variables to values during a match. Yet another drawback is of course that such regular expressions work on strings only, whereas our regular expression patterns work over lists of any datatype.

The recent trend in XML-centric languages has led to several new languages with support for regular expression pattern matching such as XMLambda [13], XDuce [7] and CDuce [2]. Most similar to ours is probably

CDuce, a general purpose XML-centric programming language. The main focus in this language is its regular expression types which are used to validate XML documents. Borrowing from XDuce they also have regular expression patterns which are tightly coupled with the type system. This allows for very precise type information to be propagated in the right hand side of a pattern. The main difference with our work is the close connection with the type system. Our extension is little more than just syntactic sugar which makes it very easy to implement.

Another recently developed language that features regular expression patterns is Scala [16]. Scala is a multi paradigm language supporting both object oriented and functional programming. Its regular expression facility is rather similar to ours but differs at the following points. Firstly, there is only one variable binding construct which has a context dependent behaviour. Secondly, Scala has non-greedy operators just as we do but have no greedy counterparts. This can make some patterns awkward to express. Scala's regular expression patterns work for arbitrary sequences.

There has been some work in extending Haskell with the full power of XDuce, called XHaskell [10]. This work focuses on fitting the type system of XDuce into Haskell and encoding it using Haskell's class system. They also have regular expression patterns but these are intimately coupled with regular expression types and do not work together with ordinary pattern matching.

8 Future Work

There are several areas where our regular expression patterns extension can be improved. It is not obvious that our implementation using a monadic parser is the most efficient approach, on the contrary. There has been lots of work on efficient matching of regular expressions and it is likely that some of these techniques could be used with our system to make it more efficient.

We will need to devise and implement a type checking algorithm for our regular expression patterns on top of Haskell´s type checking mechanism. Being able to type check our regular expression patterns before translating them into vanilla Haskell, as opposed to our current implementation that first translates and then lets a Haskell type checker do the work, would, if nothing else, lead to much improved error messages.

9 Acknowledgement

We would like to thank our shepherd Erik Meijer for his many suggestions which improved the paper enormously. Thanks also to Karol Ostrovsky and David Sands who gave valueable feedback on draft versions of this paper. The participants of the Multi Meeting provided insightful comments when we presented the material in this paper. Lastly thanks to the anonymous referees for their comments.

This work was partially funded by the Swedish Foundation for Strategic Research.

References

[1] L. Augustsson. Compiling pattern matching. In *Functional Programming and Computer Architecture*, 1985.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[3] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Haskell Workshop*, 2000.

[4] K.-F. Faxén. A static semantics for haskell. *Journal of Functional Programming*, 12(4–5), 2002.

[5] A. Frisch. Regular tree language recognition with static information. In *3rd IFIP International Conference on Theoretical Computer Science*, 2004.

[6] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003.

[7] H. Hosoya and B. C. Peirce. Xduce: A typed xml processing language. *ACM Transactions on Internet Technology*, 2(3):117–148, 2003.

[8] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. In *Proceedings of the ACM International Conference on Functional Programming*, 2000.

[9] M. Y. Levin. Compiling regular patterns. In *Proceedings of the ACM International Conference on Functional Programming*, pages 65–78, 2003.

[10] K. Z. M. Lu and M. Sulzmann. XHaskell: Regular expression types for haskell. http://www.comp.nus.edu.sg/ sulzmann/, 2004.

[11] L. Maranget. Compiling lazy pattern matching. In *Proc. of the 1992 conference on Lisp and Functional Programming*. ACM Press, 1992.

[12] L. Maranget. Two techniques for compiling lazy pattern matching. Research report 2385, INRIA, 1994.

[13] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[14] E. Meijer and D. van Velzen. Haskell server pages. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier, 2001.

[15] Perl. www.perl.org.

[16] The Scala Programming Language. http://scala.epfl.ch/.

[17] P. Wadler. *The Implementation of Functional Programming Languages*, chapter Efficient Compilation of Pattern Matching. Prentice Hall, 1987.

Haskell Server Pages through Dynamic Loading

Niklas Broberg

Abstract

Haskell Server Pages (HSP) is a domain specific language, based on Haskell, for writing dynamic web pages. Its main features are concrete XML expressions as first class values, pattern-matching on XML, and a runtime system for evaluating dynamic web pages.

The first design of HSP was made by Erik Meijer and Danny van Velzen in 2000, but it was never fully designed nor implemented. In this paper we refine, extend and improve their design of the language and describe how to implement HSP using dynamic loading of pages.

1 Introduction

Long gone are the days when the world wide web consisted mostly of static HTML pages. Today, dynamic web pages, i.e. programs that generate page contents on demand, are used for a multitude of purposes. They range from simple access counters to complete business applications built entirely on the web.

As the use of dynamic web pages has increased, so too has the need for better tools to use when creating them. To create dynamic web pages, programmers can use either specialized scripting languages that allow mixing of XML and code, e.g. PHP [5] or ASP [11], or they can use CGI [2] programs written in basically any programming or scripting language, but more often than not in Perl, Python or C.

However, most if not all of the commonly used languages share a common flaw, and a severe one at that — they model HTML data as raw text. This violates one of the most fundamental principles of language design, Tennent's principle of abstraction [21, 19], that says that values of a syntactically relevant domain can be given a name. Clearly, in a language targeted at writing programs that create HTML documents there should be the notion of an

HTML type, and built-in support for creating and manipulating HTML values.

It is widely recognized [6, 12, 16, 17, 25] that the functional programming idiom is particularly well suited for creating and manipulating XML and HTML documents, and a good deal of libraries exist [9, 15, 23, 25] that assist in writing CGI programs in functional languages. Unfortunately CGI programs suffer from some drawbacks. They are inherently stateless since one request of a CGI page causes one execution of the corresponding CGI program. Also, writing CGI programs requires at least some non-trivial knowledge of the host language, even when adding very simple dynamic contents like an access counter. Such a steep initial learning curve means many aspiring web programmers with no previous programming experience will instead choose one of the specialized scripting languages that allow a simpler transition from static HTML to dynamic pages.

What we would like is a functional language that supports a stateful programming model and the ease of use of specialized scripting languages, while still retaining its nice XML processing capabilities. Enter Haskell Server Pages.

In 2000, Erik Meijer and Danny van Velzen presented what they called Haskell Server Pages (HSP) [17], a domain-specific web programming language based on the functional general-purpose programming language Haskell. It improved over its peers by introducing a central XML data datatype, which guarantees well-formedness of produced pages and leads to a better integration between XML data and other code. Indeed, XML fragments were just another form of expressions. Their HSP was intended to work on top of the Haskell Execution Platform (HEP) [20]. Unfortunately their intended implementation was stalled together with HEP and was never resumed. In this paper we pick up the thread left by the original authors. Our contributions to HSP are threefold:

1. We redesign the implementation of the HSP runtime system, replacing the dependency on HEP with dynamic loading of object files based on hs-plugins [18].

2. We refine, improve and extend the original HSP programming model.

3. We provide a few central low-level libraries to support common web programming tasks, and a framework to make it easy to create more specialized higher-level libraries on top of this core.

This paper covers 1 and 2, while 3 is covered in the thesis that is the basis for this paper [7]. The thesis also contains a more thorough explanation of 2 than what we give here.

The rest of this paper is organized as follows. Section 2 starts by giving examples of the HSP language itself. Section 3 covers the extensions, refinements and improvements that we have made to the HSP language as presented by Meijer and van Velzen. In section 4 we give an overview of our implementation, and in section 5 we discuss the current status of that implementation. Sections 6 and 7 cover related and future work respectively, and section 8 concludes.

2 Examples

In this section we give an overview of the HSP language and how it differs from ordinary Haskell, to give the reader a feel for the issues we tackle in the subsequent sections.

2.1 XML meets Haskell

At the core of HSP, that which makes us consider it a language of its own, is the feature that XML fragments are first class values, both syntactically and semantically.

As a first example, we can write a simple paragraph containing the customary compulsory text as

```
helloWorld = <p>Hello World!</p>
```

There are two interesting things to note with the above declaration. First, we do not need to use any escape mechanism to leave code context when we want to write the XML fragment, as we would have had to in PHP or ASP. This comes for free in HSP since an XML fragment is itself an expression.

The other thing to note is that in HSP all XML fragments are guaranteed to be well-formed at all times. This means that opening tags must have corresponding closing tags. The ill-formed expression in the following definition is thus a syntactic error and will be statically rejected by the HSP parser:

```
helloWorld = <p>Hello World!</q>
```

A similar error in PHP or ASP would be accepted without further ado, where an XML fragment is simply a string.

As we saw in the above example, text inside tags (PCDATA) can be written just like in XML, without escape notation like that of Haskell string literals. The tags themselves function as escapes, so anything that comes between them is assumed to be either text, or another tag as in

```
boldHelloWorld = <p><b>Hello World!</b></p>
```

Instead we must use escapes whenever we want inner content to be computed by an expression, as in the function

```
hello name = <p>Hello <% name %>!</p>
```

where `hello` takes a name and produces an XML fragment. In summary, in HSP we need to use escapes to write code inside XML, but not the other way around. In string-based languages like PHP, we need to do both. This leads to a nested code structure in HSP reflecting the hierarchical nature of XML, i.e. we write code inside XML inside code inside XML ..., as opposed to the flat structure of PHP.

Not only strings can be embedded inside XML fragments in HSP, but rather anything that can be represented as part of the XML tree being built. The most obvious example of such a value is of course another XML fragment, as an example we can define the page body for our initial example as

```
helloBody = <body><% helloWorld %></body>
```

Other examples of embeddable values include numbers, optional (`Maybe`) values and lists of embeddable values.

We also want our XML elements to be able to have attributes, so for example we can define

```
redHelloWorld :: HSP XML
redHelloWorld =
    <p style="color:red">Hello World!</p>
```

as a somewhat more colorful greeting. All attributes come as name-value pairs just as in ordinary XML. Like children, attribute values can also be computed from embedded expressions as in

```
hwColor :: String -> HSP XML
hwColor c =
    <p style=("color:" ++ c)>Hello World!</p>
```

There is no need to escape embedded expressions for attributes, since in the static case, e.g. style="color:red", the value can simply be treated as a Haskell string expression. Again just like for children, we allow a wide range of types for embedded attribute expressions.

We can also construct and assign attributes programmatically, in which case we need to use a slightly different syntax. Using the `set` function that sets an attribute on an element, we can define

```
redHelloWorld =
    <p>Hello World!</p>
        `set` "style":="color:red"
```

where the operator := associates an attribute name with a value.

It is also often convenient to be able to pass in complete attributes, i.e. both name and value, to tags directly. We allow an extra expression directly inside a (self-contained or opening) tag, denoting a list of additional attributes for the element in question. As an example, we can define a more general function for the above as

```
hwWithAttrs attrs = <p attrs>Hello World!</p>
```

and define e.g.

```
hwColor color =
    hwWithAttrs
        ["style":= "color:" ++ color]
```

All HSP pages must define and export a function `page` defining the contents of the page. It is this function that is called when the page is requested. To complete our example we can thus define

```
page = <html>
         <head><title>Hello World!</title></head>
         <% helloBody %>
       </html>
```

to produce a complete (X)HTML page.

2.2 Pattern matching

Now that we know how to build XML values using concrete XML syntax, we will have a look at how to take them apart again using pattern matching. This area was covered only briefly in the original HSP design, so most of it is of our own making.

2.2.1 Elements

First of all we can match directly on elements, as in

```
isImg <img/> = True
isImg _ = False
```

Our intuitive interpretation of the above is simply that isImg will return True if it is given an img element, and False otherwise.

2.2.2 Attributes

For pattern matching on attributes, we first need to consider how we want to use them. First of all, in XML the order of attributes is irrelevant, so for instance the two elements

```
<img src="img/myImg.jpg" alt="My image" />
```

and

```
<img alt="My image" src="img/myImg.jpg" />
```

should be equivalent. Second, the far most common reason for pattern matching on attributes is when you want to know the value of a certain attribute, regardless of (the existence of) other attributes. Therefore we want to model our system so that these two things are easy to express. In this example

```
imgSrcAlt <img src=s alt=a/> = Just (s,a)
imgSrcAlt <img/> = Nothing
imgSrcAlt _ = error "Not an image"
```

we let the first case match any *img* element with the *src* and *alt* attributes set, regardless of their internal order, or whether any other attributes are also set. The second case will match any *img* element whatsoever. In effect we treat the attributes of an element as a set, and matches on specific attribute values as lookups into this set.

In some cases we need to know about the full set of attributes, so analogous to expressions we allow an extra pattern at the end, denoting the remaining set of attributes. For instance we can write the pattern

```
<img src=s [] />
```

that will match any img element with only the src attribute given, while

```
<img as/>
```

will bind the whole set (list) of attributes of an img element to the variable as for subsequent lookups.

2.2.3 Children

Pattern matching on children follows just as easily, as in

```
getPText <p><% PCDATA t %></p> = t
```

where the pattern inside code escape tags matches exactly one child, in this case a PCDATA child. We use the word PCDATA here as a marker, denoting that the matched child should be text and not another element. In truth PCDATA is a data constructor in the XML datatype that we cover in section 3.1, but the programmer doesn't need to know that in order to use it as a marker.

Matching a single child is simple enough, but for more complicated examples we run into trouble. When matching XML, we often want to be able to say things like "match an arbitrary number of <p> elements", or "start by matching a <h1> element, followed by one or more occurrences of a <h2> element and a <p> element in sequence". Clearly, basic Haskell pattern matching will not be powerful enough for these purposes. More formally, "proper patterns matching on XML fragments requires [...] matching of regular expressions" [17].

To this end we have developed HaRP (Haskell Regular Patterns), the system for regular expression patterns in Haskell [8]. Using these regular expression patterns we can express more powerful patterns, for example to get all the text of all paragraphs in a page body, we can say

```
getText :: XML -> [String]
getText <body>[
          <p><% PCDATA t %></p>*
         ]</body> = t
```

where the * denotes that they may be zero or more p elements enclosed inside the body element.

2.3 Formal Syntax

The formal syntax of the XML extension has been shown by Meijer and van Velzen already. Ours is only slightly different, but we show it here for reference.

We extend the Haskell grammar with new productions for XML expressions, which we add to the language as a possible atomic expression:

```
aexp ::= var
       | lit
       | ( exp )
       ...
       | xml
```

The new form of expression, `xml`, can be either an enclosing element with children, or an empty self-contained tag:

```
xml ::= <name attrs>child...child</name>
      | <name attrs/>
```

Attributes are name-value pairs, optionally followed by an extra expression:

```
attrs ::= attrs1 aexp
        | attrs1

attrs1 ::= name = aexp ... name = aexp
```

A child can be a nested element, PCDATA or an embedded Haskell expression

```
child ::= xml
        | PCDATA
        | <% exp %>
```

PCDATA should basically match anything written between tags that is not another tag or an embedded expression.

A name in XML may optionally be qualified by a namespace to which the element belongs. Our `name` production thus looks like

```
name ::= string : string
       | string
```

where the former means a namespace is specified.

2.4 Environment

HSP pages have access to a special environment that contains information regarding the context in which they are evaluated. The different components that together make up the environment are inspired by similar functionality in other languages, ASP and PHP in particular. The main components are:

Request	contains information on the HTTP request that initiated the call to the page.
Response	contains information on the HTTP response being generated, that will be sent with the results of the call to the page.
Application	contains data with application scope lifetime, i.e. data that persists between page transactions.
Session	contains data that pertains to a specific client.

In ASP these four components are modelled as objects that can be referred to statically, e.g. `Request("hello")` reads the value of the parameter "hello" from the incoming request. In PHP the components don't exist per se, but the functionality still exists through standalone globally available functions and collections, e.g. `$_REQUEST["hello"]` which is equivalent to the ASP expression above. HSP takes a middle road, where we model the components similarly to ASP but access them using ordinary functions. The above expression in HSP would be `getParameter "hello"`. Below we look more in detail at the different components and what they contain.

2.4.1 Request

Probably the most important information in the Request component, from the point of view of the programmer, is query string data. The query string is a set of name-value pairs, written $param_1 = value_1 \& ... \& param_n = value_n$, that contains any parameters supplied by the client, such as XHTML form data. Parameters in the request can be accessed using the functions

```
getParameter :: String -> HSP (Maybe String)
readParameter :: Read a => String -> HSP (Maybe a)
```

The HSP monad that these functions reside in will be introduced in section 3.2. Apart from parameter data, the Request component also contains HTTP information such as the HTTP method used (GET or POST), as well as any headers set in the incoming request. All this information can be accessed using special purpose functions such as

```
getHTTPMethod :: HSP Method
getHTTPUserAgent :: HSP String
```

2.4.2 Response

The Response component stores headers that will be sent back to the receiving client along with the page contents. These are set using specialized functions like

```
setNoCache :: HSP ()
```

Notable is the functionality not present, namely writing any content output to the response. In ASP the Response object has a method `write` (and PHP has the equivalent function `echo`) used as in

```
<p>Hello <% Response.write(name) %></p>
```

In HSP no output may be added to the response manually, all content is generated by the `page` function, guaranteeing well-formedness of the generated XML.

2.4.3 Application

A web application rarely consists of a single page, more likely it is spread over many different pages that work together to provide some functionality. For HSP, we define an application as all pages on a particular hierarchical level, i.e. pages in the same directory. The Application component contains data that is shared between all the pages of this application, and that persists between single requests to pages. With our definition, an application cannot spread into sub-directories, which is of course the case in real web applications. We are looking at suitable ways to extend our application model to allow this.

In ASP, as well as in the original design of HSP, the Application component is a simple data repository of string variables. For many applications this is not general enough, some forms of data cannot be represented as string values. Common examples are an open connection to a database, or a channel for communicating with some external application. We have chosen a more general approach, in which we allow the data in the Application component to assume any form, and we leave it up to the runtime system to keep the data alive between calls. The entire contents of the Application component is user-definable, in a file called Application.hsp within the domain of the application. This module should contain a definition of a function `initApplication :: IO Application` that should yield the initial contents. This function will be called by the runtime system before the first request of a page within the application. The `Application` type itself is abstract, so the only way to create a value of that type is using the function

```
toApplication :: (Typeable a) => a -> Application
```

The Typeable constraint arises from the fact that Application is actually a wrapper type for `Dynamic`, which is used to enable an Application component to be of any type.

```
newtype Application =
   MkApp {appContents :: Dynamic}

toApplication = MkApp . toDynamic
```

To access the application data from within a page, a programmer can use the function

```
getApplication :: (Typeable a) => HSP a
```

that returns a value of their particular Application type.

Using values of type Dynamic can be rather error-prone, since in effect we are suppressing any static type information that we have on those values, relying instead on dynamic typing. In the following example we will show how to reclaim some of those lost properties by clever structuring of modules.

"Counter Example" Assume that we want to add a simple access counter to a page. We can use the Application component to store the value of the counter, since the data therein is kept alive between calls to pages. We start by declaring the type of our application data. Since we want to update the value of the counter, we need to store it in a mutable reference:

```
type MyApplication = Mvar Int
```

We put this declaration in a module we call MyApplication.hs, so that we can import it into our pages. Next, we add the following declarations:

```
toMyApplication :: MyApplication -> Application
toMyApplication = toApplication
getMyApplication :: HSP MyApplication
getMyApplication = getApplication
```

With these two functions, we now have all the functionality we need in order to work with our application data. Also, since we have specialized the types, we can be sure that as long as we only use these functions and not the original polymorphic versions, everything will be typechecked statically.

Note that this is an idiom that works well in all cases. The only thing that is specific to our access counter example is the mention of the type `Mvar Int`, and that can be replaced by whatever type is needed.

In this particular example we can go on and define the functions that we expect to use on our counter. First we want to be able to increment the counter, so we define

```
incrCounter :: HSP ()
incrCounter = do
   ctr <- getMyApplication
   doIO $ modifyMVar ctr (+1)
```

Another thing we may want to do is to read the current value of the counter:

```
readCounter :: HSP Int
readCounter = do
   ctr <- myGetApllication
   doIO $ readMVar ctr
```

The last thing we need to do to make our counter work is to supply an initial value. In the module *Application.hsp* we first import *MyApplication.hs*, and the define

```
initApplication = do
   ctr <- newMVar 0
   return $ toMyApplication ctr
```

This function will be called by the runtime system before the first call to a page in this application, so the value of the counter is initially 0.

Now we can write a small example page using our access counter:

```
import MyApplication

page = do
   incrCounter
   <html>
    <head>
     <title>Hello visitor nr <%
             readCounter %></title>
    </head>
    <body>
     <h1>Hello!</h1>
     <p>You are visitor nr <%
         readCounter %> to this page.</p>
    </body>
    </html>
```

2.4.4 Session

A session is a series of page transactions between the server and a specific client within a certain time frame. The Session component is a data repository that lets pages maintain a state between transactions. The repository

is a simple dictionary of name-value pairs, that can be accessed or updated using the functions

```
getSessionVar :: String -> HSP (Maybe String)
setSessionVar :: String -> String -> HSP ()
```

The programmer may also affect the lifetime of the session using

```
setSessionExpires :: ClockTime -> HSP ()
```

or forcibly terminate it using

```
abandon :: HSP ()
```

Between invocations of pages the session data could be stored client-side using cookies, or server-side in a database. Our current implementation uses the latter, though this may be subject to change. In either case it will be stored as string data, which is why values are restricted to the `String` type.

It would not be feasible to allow Session components to hold arbitrary data the way the Application component can. The reason is sheer numbers — while there will be a very limited number of applications running on the same server, the number of sessions active at any given time could be huge. For this reason, Session data must be stored outside the server itself, which means we must restrict the data to a storable type. `String` seems the most natural choice.

3 The HSP Programming Model

The design of the HSP language presented by Meijer and van Velzen [17] was mostly proof-of-concept, and they left several areas sparsely detailed, or not addressed at all. To get a fully functioning language we have made several refinements and improvements to the original design. We use this section to discuss these changes and the reasons behind them.

We do not cover all parts that we have updated. There are many smaller issues, for instance how to lex PCDATA literals, that are simply not interesting enough to be included in this paper.

3.1 The XML datatype

Structurally XML fragments are trees, and as such they can be represented very naturally in Haskell using an algebraic datatype. This approach is common to well nigh every XML or HTML manipulating Haskell library

[25, 15, 9]. Just reading the productions in the syntax straight off we would get a datatype in two levels, one each for the `xml` and `child` productions respectively, as

```
data XML = Element Name Attributes Children
type Children = [Child]
data Child = ChildXML XML
           | PCDATA String
type Name = (Maybe String, String)
```

There is no need for a separate constructor to represent self-contained elements, as these are simply elements with no (an empty list of) children. Neither do we need a constructor for embedded expressions since these will result in something that can fit into the tree on its own.

The original design of HSP used a datatype in two levels like the one above. Unfortunately using such a two-level data type leads to problems in the presence of pattern matching with concrete XML syntax. In short, the problems arise because it is impossible to syntactically distinguish between XML patterns meant to match top-level elements, of type `XML`, from those meant to match child elements, of type `Child`. One could imagine various fixes to the problem, for instance using explicit annotations on patterns, but doing so is not very pretty as it breaks the abstraction of the `XML` datatype.

To avoid these problems altogether, we instead merge the two levels of the datatype into one single level, i.e.

```
data XML = Element Name Attributes Children
         | PCDATA String
type Children = [XML]
type Name = (Maybe String, String)
```

Now there is no distinction between top-level elements and child elements, and translation of pattern matching is straight-forward.

Things are seldom perfect however, this single-level datatype comes with problems of its own. Since PCDATA now belongs to the `XML` type directly, any function operating on values of type `XML` should now also consider the case where that value is actually PCDATA. This can become quite awkward and cumbersome, but at this point we can see no satisfactory solutions.

For the `Attributes` type we use a simple list of name-value pairs:

```
type Attribute = (Name, AttrValue)
type Attributes = [Attribute]
newtype AttrValue = Value String
```

The only mildly surprising part ought to be the newtype for attribute values, isomorphic to `String` yet separate. The reason is that we want to control how such values are created, so we make the `AttrValue` type abstract.

3.2 The HSP Monad

In the original HSP, the `XML` type was actually even more complex than the two-level type given above. Apart from the standard constructors for the syntactic productions, the `Child` type also had two extra constructors, `ChildIO` holding values of type `IO Child`, and `ChildList` holding values of type `[Child]`. The reason was to allow embedding of expressions that were non-pure in the case of `ChildIO`, and expressions that returned a list of children in one go in the case of `ChildList`. These constructors would then be removed during the actual evaluation of the XML tree, and replaced by what their values really represented.

We find this approach less suitable for several reasons. First it gives the impression that functions returning XML values are pure, when actually they may contain unevaluated I/O computations. Second and perhaps more important, it means that there is no way to distinguish between XML values that are actually pure and those that contain suspended computations, leading to a number of problems with pattern matching, rendering, filtering etc.

We have instead chosen to make this potential impurity explicit whenever the concrete XML syntax is used. We introduce a monad `HSP` that encapsulates any side effects, and let all XML expressions be evaluated in this monad. Further we have removed the two offending constructors and replaced them with a more general mechanism for embedding expressions. In the original HSP design, embedded expressions had to be of a type that instantiated the type class `IsChild`, with the single member function `toChild :: a -> XML`. In our version, the type class is called `IsXMLs` with the member function `toXMLs :: a -> HSP [XML]`. This allows both computations in the `HSP` monad, as well as expressions returning lists, to be embedded without cluttering the `XML` data type.

This solves the problem of the old approach, but instead introduces another problem: we can no longer use concrete XML syntax to construct pure values of type `XML`. For instance the expression `<p>Hello World</p>` is of type `HSP XML`, even though no side effects take place. Our approach is thus not perfect, but far preferable to the to the alternative, and we consider it a small price to pay.

One possible suggestion is to use a type class for the result of a concrete XML expression, so that an expression like the one above could have either

type `XML` or `HSP XML` depending on the context in which it appears. The problem with this approach is that it would lead to many situations where the type inference engine of Haskell would not be able to infer the type properly, which would force the programmer to add type annotations in situations like

```
let hw = <p>Hello World!</p>
 in <body><% hw %></body>
```

To properly infer the type of `hw` to either `XML` or `HSP XML`, we would need a mechanism for defining default instances, i.e. a way to tell the inference engine to try to infer one type whenever in doubt, and only use the other if the first didn't work out.

Note that with our approach there is a discrepancy between the type of XML expressions using the concrete syntax, and the type of expressions matched by similarly built patterns. The patterns expect values of the `XML` data type, whereas expressions produce values of type `HSP XML`. Thus the following is type correct:

```
do <img src=s /> <- <img src= img/myImg.jpg  />
   ...
```

while this is not:

```
let <img src=s /> = <img src= img/myImg.jpg  />
 in ...
```

Apart from encapsulating I/O computations, our `HSP` monad also supplies pages with the special HSP runtime environment discussed in section 2.4. The original design used implicit parameters to distribute the various components to pages. The benefit of that approach is that you get more precise types, i.e. an expression that only uses one of the components will only show that particular component in its type. With our approach the type will be in the `HSP` monad, regardless of how much of the environment is used. The downside of using implicit parameters is that it is possible for programmers to rebind the parameters to hold new values of the same type. Meijer and van Velzen solved this by making all the component types abstract, but it is still possible to rebind a parameter to `undefined`. Hiding the components in the monad means the programmer can never touch the components at all, except through the specific functions that we provide, and thus no rebinding can occur.

On the top level, we require the `page` function to have type `HSP XML`, analogous to `main` having type `IO ()` for ordinary Haskell executables.

3.3 HSP Pages

So far we have shown HSP from a programmer's perspective, using a series of function definitions. As we argued in the introduction, we also want to attract those that have no previous programming experience, but know how to write static web pages using XHTML. To accomplish this we adopt the convention that a valid XML (and thus XHTML) document is also a meaningful HSP program. Expressions can then be embedded directly into the tree, making it truly simple to add small pieces of dynamic content to an otherwise static page. We call such pages XML pages, as opposed to standard HSP pages. The following XML page is mostly static, but uses a clock to show when the page was requested:

```
<html>
 <head><title>XML page</title></head>
 <body>
  <h1>Hello World!</h1>
  <p>Page requested at <% getSystemTime %></p>
 </body>
</html>
```

To connect the two perspectives, we simply note that an XML page is a standard HSP page where its XML contents implicitly make up the body of the page function. The standard HSP page equivalent to the above is thus

```
page =
  <html>
   <head><title>XML page</title></head>
   <body>
    <h1>Hello World!</h1>
    <p>Page requested at <% getSystemTime %></p>
   </body>
  </html>
```

and it will be compiled as such.

There is a slight problem with this page though. The function `getSystemTime` resides in the module `System.Time`, which needs to be imported for the function to be used. With a standard HSP page we can simply add the import in the proper place, but there is no proper place to add an import in an XML page.

To solve this problem we introduce hybrid pages, which combine XML pages with top-level declarations. To get the import that we want, we can write

```
<% import System.Time %>
<html>
 <head><title>XML page</title></head>
 <body>
  <h1>Hello World!</h1>
  <p>Page requested at <% getSystemTime %></p>
 </body>
</html>
```

where the escape tags show that we are leaving the otherwise prevalent XML context, just as when we embed expressions. The top-level code section, allowed only at the top of a hybrid page, can contain anything that an ordinary HSP page can, i.e. both imports and declarations.

3.4 Pattern Matching

As we noted in section 2.2, the original design of HSP only covered pattern matching using XML syntax very briefly, and we have extended that design on a number of points. Pattern matching on attributes now works as lookups into the set of attributes of an element. We also allow an extra pattern to match the remainder set of elements, as a list. This gives some increased expressiveness since it is now possible to dispatch on wether a particular attribute is *not* set at all.

3.5 Regular Expression Patterns

As we also noted in section 2.2, when pattern matching on XML, ordinary pattern matching as found in Haskell is simply not powerful enough. Meijer and van Velzen note that "proper pattern matching on XML fragments requires [...] matching of regular expressions" [17]. They consider this to be one of the things that make Haskell less suited for XML processing. We instead extend Haskell pattern matching with the full power of regular expressions over lists whose elements can be of arbitrary type. This extension was presented in full in an earlier paper [8]. Examples of the use of regular expression patterns are

```
last :: [a] -> a
last [_*, x] = x

concatMaybe :: [Maybe a] -> [a]
concatMaybe [(Just x | Nothing)*] = x
```

Since we model the children of an XML element as a simple list, it is straight-forward to use regular expression patterns when matching on them. We extend the syntax to allow regular expression patterns to be mixed with concrete XML patterns, as in the example we gave in section 2.2.

3.6 The Application Component

As noted in section 2.4.3, our Application component is more general than in the original design. Instead of treating it as a simple data repository, we allow the programmer to define the contents freely. We cope with this generality by using the `Dynamic` type, which allows us to handle all possible user-defined Application components with the same code.

4 Implementation

When you want to view a certain web page, all you need to do is request that page using a browser. You expect, without having to do anything further, to receive the page to your browser within a reasonable time. In a sense it is very similar to an ordinary file request in a file system, indeed it has been argued that a web server can be viewed as an operation system [10].

From a web author's point of view, this is equally true. An XHTML page is just a file, and all that is required to give others the possibility of viewing it is to put it in the correct folder in the virtual file system of a web server. Moving a page from one server to another is simple enough, different server brands or operating systems pose no problems whatsoever.

To make the transition from static to dynamic pages smooth for a web programmer, as much as possible of this simplicity should be retained for dynamic pages as well. Many traditionally strong CGI languages such as Perl and Python, as well as the special purpose language PHP, are all interpreted. This makes it easy enough to share or deploy programs since the only necessary documents are the program sources.

For Haskell CGI programmers there are two choices, neither really satisfactory. Interpreting pages using e.g. runHugs or runGHC retains the simplicity of deployment, but interpretation of Haskell code is too slow to be suited for larger applications in commercial use. On the other hand, compiling pages makes them faster, but complicates sharing and deployment of pages.

For us it is imperative to give programmers a smooth transition from static XHTML pages, so relying on the programmer to compile pages before deployment is not really an option. To compete with the simplicity of similar

languages and systems, we must ensure that a programmer only ever needs to deal with source files. But interpretation is too slow, so instead we build our system around what we call on-request compilation:
When a page is requested, our runtime system is responsible for checking whether that page needs to be compiled, or if that has been done already. If the page is not previously compiled, the runtime system compiles it and puts the resulting object file(s) in a cache. On the other hand, if the page has already been compiled, the runtime system can use the cached object file directly, without having to recompile anything. This approach obviously leads to substantial waiting times on the first request of each page, but much faster responses on any subsequent calls. In effect, the programmer is actually still compiling pages, but the difficulty of doing so has been reduced to simply requesting them through a browser.

Our approach is very similar to that taken by the ASP.NET framework [1] that allows pages to be written using a variety of different languages. The source code documents are distributed, and pages are upon being requested compiled to .NET bytecode. The bytecode can then be efficiently interpreted for subsequent requests of the same page.

4.1 Designing the runtime system

There are a lot of decisions to make when designing our runtime system, but perhaps the most crucial and determining fact is that data with application scope should be kept alive between page transactions. Since there is no way in Haskell to store and restore arbitrary values to and from secondary storage, the only viable option is to let the runtime system itself be a running application, i.e. a server. Application data can then be kept alive in the main memory of the server, and be supplied directly to any incoming page requests.

Another important issue is how the runtime system should communicate with its surrounding environment. Our goal is to make it easy to integrate the runtime system into just about any existing web server, and to accomplish this we need a simple interface between our own server and the outside world.

The solution is not very dramatic. We want to build a server that, in the presence of an HTTP request, generates an HTTP response. In other words, our runtime system is an HTTP server for HSP pages only, and as such, communication is conducted over sockets using the HTTP protocol. This choice means that to integrate our runtime system into a general purpose web server, that server should simply forward incoming requests for HSP pages to the HSP runtime system on a different port and then wait for a response to become available.

4.2 HSP(r)

Our HSP runtime server, HSP(r), is greatly influenced by the Haskell Web Server (HWS) [14], and like HWS we use Concurrent Haskell [13] to ensure high throughput and stability, and exceptions for handling timeouts and other server malfunctions.

HSP(r) consists of one main server thread that listens for incoming requests on a port. When it receives one it forks off a dedicated light-weight request handler thread that will perform all necessary operations. Handling each request in a different thread means the server thread is free to receive new requests without waiting for the handling of previous requests to finish. This gives us concurrent execution, leading to a presumably higher throughput. To stop pages that won't terminate, the main thread will fork off a second thread for each request handler thread. This second thread will sleep for a set amount of time, and then send a timeout exception asyncronously to its associated request handler, forcibly terminating the execution.

Request handling is accomplished by a sequence of operation steps applied to the incoming request in a pipeline-like fashion. The stages of the pipeline are, in order;

1 Parse request

2 Find page

3 Load page

4 Set up environment

5 Evaluate page

6 Render results

7 Generate response

8 Send response

Stages 1,2,6,7 and 8 are all straight-forward to implement. The interesting and slightly innovative parts are the stages that load the requested page, set up the proper environment, and then evaluate the page in that environment.

4.2.1 Loading pages dynamically

When an incoming request has been successfully read and the existance of the requested HSP page has been verified, the server should load that page

into its own execution space in order to evaluate it. To load files dynamically in this fashion we rely on hs-plugins [18], which lets Haskell applications load and access content in external object files. Apart from the actual loading, hs-plugins also provides us with functionality to compile pages, and to add pieces of static content to them. As we will show, this extra functionality comes in very handy.

On the first request of a particular page, the server must go through several steps in order to get something that it can evaluate to a response. The first step is to merge the file with a stub file containing exactly two things, namely

```
import HSP
page :: HSP XML
```

Adding the import means that all HSP pages get access to the standard functionality that HSP provides - the HSP Prelude of a sorts. The import is added in a safe fashion, so if the page already explicitly imports the HSP module then nothing will be added to it. The type declaration on the other hand will always be added to the source file, overwriting any existing type given for the `page` function in the module. This guarantees both that a successfully compiled HSP page will indeed have a `page` function, and that it will be of the required type `HSP XML`.

This is all handled nicely by hs-plugins through the function `mergeToDir`:

```
tmpSrcFile <- mergeToDir srcFile stubFile cacheDir
```

We have simplified this and the following code slightly. In particular we do not consider the erroneous case here.

The second step is to compile the intermediate source file that is the result of the merging. Once again hs-plugins gives us the required functionality:

```
objFile <- makeAll tmpSrcFile hspFlags
```

The `makeAll` function works just like the GHC compiler when given the `--make` flag (and the `-no-link` flag as well; we just get the object files). Not surprising considering that this function really calls GHC behind the scenes, with said flag(s) given. We also supply a number of extra flags, among them the flag that tells GHC to use our syntactic preprocessor on the source files. This preprocessor will go through all files, search for XML expressions and patterns, and replace them with equivalent vanilla Haskell code.

The object file resulting from the compilation can then be loaded into the server application itself. More specifically we load the `page` function from the object file, using the core functionality of hs-plugins:

```
page <- load "page" objFile
```

We know for certain that such a function exists due to the stub file declarations, so we need none of the possible extra safeguards that hs-plugins could provide.

Of these three steps, the first two will only be done on the very first request of any given page. On subsequent requests the server can use the already compiled object file for that page from the cache. The loading step is also only done once per page, but needs to be redone whenever the server is restarted.

4.2.2 Runtime environment

To set the scene for the page evaluation, we need to set up the proper runtime environment. HSP defines four runtime components: Request, Response, Application and Session as discussed in section 2.4.

The Request component is an interface to the data contained in the HTTP request that triggered the call to this page. It is more or less already set up in the *Read request* stage, what remains to be done is to parse the various components of the request into values of suitable data types.

The Response component is even easier to set up. It is intended as an interface to the outgoing HTTP response that will carry the results of this page back to the caller, and it will naturally be empty initially. We model it as a `MVar` holding a list of headers, to which code in the HSP monad can add.

Setting up the Application component is a lot trickier. The contents of this component is user definable, accessible to all pages in the application, and should be persistent during the lifespan of the application. To enable this we use much the same techniques as for pages. The user defines the initial contents of the component in a file called *Application.hsp*. On the first request of a page in the application, that file is merged with a stub file to guarantee that it contains a function `initApplication :: IO Application`. The intermediate file is compiled in the cache, and the resulting object file is loaded into the server.

The main difference from how pages are treated is that the loaded function `initApplication` is only called once, not once for each request, and the result of that call is stored to be given to all subsequent calls to pages in the same application. Thus the first thing the server does when it tries to set up the Application component is to check among the stored components, and only if it finds nothing will it try to load the Application from its object file. If no *Application.hsp* file exists for the application, the server inserts

`(throw NoApplicationDefined)` in its place, so as long a page doesn't try to access the Application component everything will work fine.

The Application component is modelled in the server as a value of type `Dynamic`, which means it can be (almost) whatever type the programmer wishes.

The Session component contains data that should be persistent across transactions, i.e. a series of requests to (possibly) different pages on the server from the same client. To enable this persistency we rely on the HTTP mechanism for persistent state - cookies. There are clearly problems associated with cookies as has been discussed by Thiemann [23], but at this point we see no better solution.

The actual data contained in each Session component is stored in a server-side database, indexed by unique session IDs, and it is the IDs that are stored in cookies on the client. To set up the Session component for a particular page, the handler looks up the appropriate cookie among the headers of the request, and asks the database for the corresponding data. We wrap the Session component in a `Maybe` type, so if no session cookie is set in the request (i.e. no session is active) the handler uses `Nothing` for the Session field. A session can be started at any time during the execution of a page, and using a `Maybe` value instead of an exception as for Application means we can replace the `Nothing` with something if that should occur.

4.2.3 Page evaluation

Evaluating the page now simply amounts to running the page function in the environment. The result will, if nothing goes wrong, be an XML tree that can be straight-forwardly rendered. If something does go wrong, i.e. an uncaught exception occurs, the handler catches it and generates a *500 Internal Server Error* response, optionally with details regarding the error that occurred.

The XML tree itself is not the only result of the execution however. Both the Session and Response components could hold new or updated data that should be handled. Any headers in the Response component will simply be added to the outgoing HTTP response. If a session is active, the handler will extract the session ID and generate a *Set-Cookie* header for it. Also any updates to session data during the evaluation should be pushed down to the database for future reference.

5 Status

We have a working implementation of HSP and HSP(r), available through darcs [3] at

```
http://www.cs.chalmers.se/~d00nibro/hsp
```

HSP(r) works as specified as a stand-alone server, however, to be really useful we also need to write code that binds HSP(r) to general-purpose web servers (e.g. Apache).

We also supply a module RunCGI that allows programmers to run their HSP programs in CGI mode. This is somewhat less efficient than using the server, and neither the Application component nor the Response component will be accessible.

There are a number of implementation details left to address, such as the handling of HTTP POST requests, file uploads, etc. These are not conceptually difficult in any way, and they have been left out due to time constraints only.

Our preprocessor is built as an extension of the standard libraries package haskell-src, which means we get the parsing of ordinary Haskell code for free.

6 Related Work

The work most related to ours is of course the original HSP by Erik Meijer and Danny van Velzen. We have revised, refined and extended the original design and redesigned the runtime system, so while their work might be seen as a prototype, ours is a full-blown implementation.

The closest relative of HSP is probably WASH [23], a Haskell domain specific language for writing CGI programs. It supports concrete XML syntax in expressions, and elegantly handles session state over page transitions. Using typed combinators for HTML generation, WASH supports a very high-level programming model for writing dynamic web pages.

WASH uses type classes to statically enforce that the HTML trees generated are not only well-formed, but also (almost) type correct with respect to the XHTML DTD [22]. By doing this they can still use a very simple datatype for XML trees in the background, since the type enforcement takes place outside the datatype itself. The XML datatype is then hidden from the programmer, so no further validations are needed.

Unfortunately such an approach would not work for HSP since we allow pattern matching on XML trees, and without type information in the XML

trees there is no way to give a correct "type" to XML fragments obtained this way.

Many other CGI libraries exist for Haskell [15, 9], all based on combinators for HTML. HSP sports a few low-level combinator libraries for web page creation, comparable in design to many of these other libraries. Still, the strength of HSP is as a framework for higher-level libraries, and it should be possible to modify at least some of these other libraries to work with HSP as well. The runtime system of HSP enables application state which is not possible using CGI only.

The languages that HSP aspires to compete with are instead languages like PHP [5] and ASP [11] that have programming models very similar to that of HSP. These languages are very easy to pick up and use for small applications, indeed the ease with which one can add dynamic content to an otherwise static page using PHP is probably the number one reason for its popularity today. The problem with these languages is that they are not really suited for processing HTML or XML, in fact they treat HTML fragments as strings. Thus they cannot make any guarantees of wellformedness of the generated output, which in turn means they do not scale very well for larger applications.

The new improved version of ASP, ASP.NET [1], is basically ASP with a large set of customizable and highly useful so called server tags for writing HTML. These are in some sense analogous to combinators in functional languages. As long as the programmer only uses these server tags the output is guaranteed to be well-formed, improving conditions for scalability, but it is still possible to work directly on the string level as in the old ASP.

Several functional languages exist that are designed solely for the purpose of XML processing, e.g. XMLambda [16], XDuce [12], CDuce [6]. These so-called XML-centric languages are clearly better suited for XML processing than HSP in some aspects since they can not only guarantee wellformedness of the generated XML, but also statically validate the XML against e.g. a DTD. This is accomplished by using more sophisticated type systems than that of Haskell.

HSP's advantage is once again the runtime system that gives specialized support for dynamic web pages. Also since HSP builds on the general-purpose programming language Haskell, there exists a large codebase already available, which cannot be said about these XML-centric languages.

FastCGI [4] is a new open standard intended by its creators to subsume and replace CGI. It improves CGI by allowing programs that generate pages to live across several requests to the same page, thereby allowing them to maintain a state between different requests.

7 Future Work

HSP is continously evolving, and there are many areas that could be greatly improved. The programming model is fairly complete, but needs lots of cosmetics to be really attractive to work with. It is also very low-level in many ways, and the code one would write in the base programming model is essentially imperative. Many functional programmers will find this a bit tedious and there are most certainly more functional approaches to web page construction. However, this could very well simply be a matter of providing the right libraries. We would like to see HSP as a platform on which library constructors can create abstractions that provide more functional interfaces, while getting much of the tedious, low-level parts such as server integration for free. It would be really nice to see an implementation of e.g. WASH built on the HSP platform.

7.1 Continuations

Our HSP programming model lacks one major component — a model to smoothly handle continuations. The ability to work with first-order continuations is one of the greater advantages that functional languages have when it comes to web programming, for instance it is one of the main motivations for the creation of Links [24].

As it is, we can define a library that restricts continuations to be defined on top-level. Real continuations are so much more than that however, clearly we would like to be able to compute continuations programmatically within our pages. To handle real continuations we would need a way to store these computed continuations on the server between transactions, or use a trick like that of WASH to reconstruct continuations from data stored in data sent to the client. The former approach can lead to huge space problems since there is really no way to know if and when a particular continuation will be resumed by a call from a client. Also in the presence of laziness it is not always obvious just what code will be executed when the continuation is invoked and what code will be executed when the continuation is created. The WASH trick alleviates the server from the burden of storing continuations, but means more data will be sent between the server and the client during each transaction.

We leave the implementation of continuations an open issue for now and hope to return to tackle it in not too long.

7.2 HSP(r)

The implementation provides a fully functional HSP system, but there is room for plenty of improvements. Perhaps the greatest deterrent at this point for prospective users is the horrible error messages that arise from the fact that we use a purely syntactic preprocessor. This means that all code is translated into Haskell before it is type checked, so any type errors will be reported on the post-processed code. The conceptually best, possibly the only, solution to the problem is daunting, namely to extend the preprocessor with a type checker that can handle our XML syntax as well as ordinary Haskell code, including all extensions that HSP uses.

To be able to use HSP(r) at all we also need to provide bindings to it for general purpose web servers like Apache. At the moment we provide a server-independent version of HSP that uses CGI and manual compilation, but due to the limitations of CGI it cannot use the Application component, nor is it as efficient as the server.

8 Conclusions

All in all, HSP is a really cool language that at the very least is as good as other specialized scripting languages like PHP and ASP. The area where we need to put the most effort from now on is library construction, where these other languages definitely have the edge so far. Luckily much functionality is available through ordinary Haskell libraries, something that has been a major motivation to build a web language around Haskell in the first place.

Many things presented in this paper simply reiterate what Meijer and van Velzen said five years ago. Perhaps the most important conclusion of this paper, and the project as a whole is thus that HSP is working, and it is every bit as good as we had hoped for. Our initial design goals were a language that has the expressive power of Haskell to appeal to the hardcore functional programmer, while at the same time allowing fledgling web programmers to find it easy enough to begin with after having written static pages only. It remains to see if this will be proven true, but we believe HSP has every chance to succeed if given a chance.

It is our hope that our presentation in this paper will convince Haskell programmers to write their web applications in HSP. To really take off, HSP will need an active community to help with extensions and improvements, libraries and example programs.

We also believe that our use of dynamic loading and on-request compilation are interesting in themselves, as nice code examples for others. The same could be said about the way we use the `Dynamic` type to smoothly handle values of varying types.

9 Acknowledgements

To Andreas Farre, who is the co-author of this paper, in spirit if not in fact, and has been working with me on HSP in different stages of its development. To Josef Svenningsson and Ulf Norell for being insightful and inspiring supervisors.
To the anonymous referees, and numerous others, who gave helpful and insightful comments on the paper. Finally a large thank you to Erik Meijer and Danny van Velzen for doing the original design of HSP. Without their pioneering work this project would never have gotten as far as it has today.

References

[1] ASP.NET home. `http://msdn.microsoft.com/asp.net/`.

[2] Common Gateway Interface. `http://www.w3.org/CGI/`.

[3] darcs. `http://abridgegame.org/darcs/`.

[4] FastCGI. `http://www.fastcgi.com`.

[5] PHP. `http://php.net/`.

[6] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, 2003.

[7] N. Broberg. Haskell Server Pages, 2005. Master's thesis.

[8] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proceedings of the ACM International Conference on Functional Programming*, 2004.

[9] A. Gill. The Haskell HTML Library, version 0.3. `http://www.cse.ogi.edu/~andy/html/intro.htm`.

[10] P. Graunke, S. Krishnamurthi, S. V. der Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *Proceedings of the European Symposium On Programming*, 2001.

[11] S. Hillier and D. Mezick. *Programming Active Server Pages*. Microsoft Press, 1997.

[12] H. Hosoya and B. C. Peirce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 2(3):117–148, 2003.

[13] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, 1996.

[14] S. Marlow. Writing high-performance server applications in Haskell, case study: A Haskell web server. In *Haskell Workshop*, 2000.

[15] E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 1(1), 1998.

[16] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[17] E. Meijer and D. van Velzen. Haskell Server Pages: Functional programming and the battle for the middle tier. 2001.

[18] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.

[19] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, 1986.

[20] J. Seward, S. Marlow, A. Gill, S. Finne, and S. P. Jones. Architecture of the Haskell Execution Platform (HEP) version 6. `http://www.haskell.org/ghc/docs/papers/hep.ps.gz`.

[21] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[22] P. Thiemann. A typed representation for HTML and XML documents in haskell. 2001.

[23] P. Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, 2002.

[24] P. Wadler. Links. `http://homepages.inf.ed.ac.uk/wadler/`.

[25] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation. In *Proceedings of the ACM International Conference on Functional Programming*. ACM Press, 1999.

Flow Locks

Towards a Core Calculus for Dynamic Flow Policies

Niklas Broberg David Sands

Abstract

Security is rarely a static notion. What is considered to be confidential or untrusted data varies over time according to changing events and states. The static verification of secure information flow has been a popular theme in recent programming language research, but information flow policies considered are based on multilevel security which presents a static view of security levels. In this paper we introduce a very simple mechanism for specifying dynamic information flow policies, flow locks, which specify conditions under which data may be read by a certain actor. The interface between the policy and the code is via instructions which open and close flow locks. We present a type and effect system for an ML-like language with references which permits the completely static verification of flow lock policies, and prove that the system satisfies a semantic security property generalising noninterference. We show that this simple mechanism can represent a number of recently proposed information flow paradigms for declassification.

1 Introduction

Unlike access control policies, enforcing an information flow policy at run time is difficult because information flow is not a runtime property; we cannot in general characterise when an information leak is about to take place by simply observing the actions of a running system. From this perspective, statically determining the information-flow properties of a program is an appealing approach to ensuring secure information flow. However, security *policies*, in practice, are rarely static: a piece of data might only be untrusted until its signature has been verified; an activation key might be secret only until it has been paid for.

This paper introduces a simple policy specification mechanism based on the idea that the reading of storage location ℓ by certain actors (principals, levels) is guarded by boolean flags, which we call *flow locks*. For example, the policy $\ell_{\{High;paid \Rightarrow Low\}}$ says that ℓ can always be read by an actor with a high clearance level, and also by an actor with a low clearance level providing the "paid" lock is open.

The interface between the flow lock policies and the security relevant parts of the program is provided by simple instructions for opening and closing locks. The program itself does not depend on the lock state, and the intention is that by statically verifying that the dynamic flow policy will not be violated, the lock state does not need to be computed at run time.[1]

In addition to the introduction of flow locks, the main contributions of this paper are:

- The definition of a type system for an ML-like language with references which permits the completely static verification of flow lock policies
- A formulation of the semantics of secure information flow for flow locks, and a proof that well typed programs are flow-lock secure (the reader is referred to the extended version of this article for the details).
- The demonstration that flow lock policies can represent a number of recently proposed information flow paradigms.

Regarding the last point, the work presented here can be viewed as a study of *declassification* mechanisms. In a recent study by Sabelfeld and Sands [18], declassification mechanisms are classified along four dimensions: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released. One of the key challenges stated in that work is to *combine* these dimensions. In fact, combination is perhaps not difficult; the real challenge is to combine these dimensions without simply amassing the combined complexities of the contributing approaches. Later in this paper we argue that flow locks can encode a number of recently proposed "declassification" paradigms, including the lexically scoped flow policies introduced by Almeida Matos and Boudol [2], Chong and Myers' notion of *noninterference until declassification* [5], and Zdancewic and Myers *robust declassification* [22, 13]. These examples, represent the "where", "when" and "who" dimensions of declassification, respectively, suggesting that flow locks have the potential to provide a core calculus of dynamic information flow policies.

[1] The term *dynamic* flow policy could have different interpretations. We use it in the sense that the flow policies vary over time, but they are still statically known at compile time.

The remainder of the paper is organised as follows. Section 2 gives an informal introduction to flow locks by showing a few motivating examples. In Section 3 we then present the system formally, and outline a semantic security condition in Section 4. Section 5 discusses related systems, with an emphasis on how we can use flow locks to encode them. Finally Section 6 concludes.

2 Motivating Examples

First let us assume we have a simple imperative language without any security control mechanisms of any kind. Borrowing an example from Chong and Myers [5], suppose we want to implement a system for online auctions with hidden bids in this language. We could write part of this system as the code on the right.

```
1 int aBid = getABid();
2 int bBid = getBBid();
3 makePublic(aBid);
4 makePublic(bBid);
5  ... decide winner + sell item
```

This surely works, but there is nothing in the language that prevents us from committing a serious security error. We could for instance accidently switch the lines 2 and 3, resulting in A's bid being made public before B places her bid, giving B the chance to tailor her bid after A's.

Flow locks are a mechanism to ensure that these and other kinds of programming errors are caught and reported in a static check of the code.

The basic idea is very similar to what many other systems offer. To deny the flow of data to places where it was not meant to go, we annotate variables with policies that govern how the data held by those variables may be used. Looking back on our example, a proper policy annotation on the variable `aBid` could be $\{A; \texttt{BBid} \Rightarrow B\}$. The intuitive interpretation of this policy is that the data held by variable `aBid` may always be accessed by A, and may also be accessed by B whenever the condition `BBid`, that B has placed a bid, is fulfilled. `BBid` here is a *flow lock* — only if the lock is *open* can the data held by this variable flow to B. To know whether the lock is open or not we must look at how the functions for getting the bids could be implemented.

```
function getABid(){
  int {A;BBid⇒B} x
    = bidChanFromA;
  open ABid;
  return x;
}
```

The function shown on the right first fetches the bid sent by A. We model the incoming channel as a global variable that can be read from, one with the same policy as `aBid`. When the bid has been read, the function signals this by opening the `ABid` lock—A has now placed a bid and the program can act accordingly. The im-

plementation of `getBBid` follows the same pattern, and will result in `BBid` being open. Now both bids have been placed and can thus be released. The `makePublic` function would be implemented as shown on the right. The outgoing `publicChannel` is also modelled as a global variable that can be written to. This one has the policy $\{A; B\}$ attached to it, denoting that both A and B will be able to access any data written into it. At the points in the program where `makePublic` is applied, both A and B will have placed their bids, the locks `ABid` and `BBid` will both be open, and the flows to the public channel will both be allowed. However, if the lines 2 and 3 were now accidently switched, it would be a different story. Then we would attempt to release A's bid, guarded by the policy $\{A; \texttt{BBid} \Rightarrow B\}$, onto the public channel with policy $\{A; B\}$. Since the flow lock `BBid` will then not yet be opened, this flow is illegal and the program can be rejected.

```
function makePublic(bid){
  publicChannel = bid;
}
```

Taking the example one step further, assume that we have two items up for auction, one after the other. We can implement this rather naively as the program to the right. The locks `ABid` and `BBid` will both be opened on the first calls to the `getXBid` functions. But unless we have some means to reset them, there is again nothing to stop us from accidently switching lines to make our program insecure, this time lines 9 and 10. The same problem could also be seen from a different angle: what if the locks were already open when we got to this part of the program? Clearly we need a closing mechanism to go with the open. The function `auctionItem` could then be implemented as shown here. By closing the locks when an auction is initiated, we can rest assured that both A and B must place new bids for the new item before either bid is made public.

```
1  auctionItem(firstItem);
2  aBid = getABid();
3  bBid = getBBid();
4  makePublic(aBid);
5  makePublic(bBid);
6   ... decide winner + sell item
7  auctionItem(secondItem);
8  aBid = getABid();
9  bBid = getBBid();
10 makePublic(aBid);
11 makePublic(bBid);
12  ... decide winner + sell item
```

```
function auctionItem(item){
  close ABid, BBid;
  ... present item ...    }
```

It should be fairly easy to see that what we have here is a kind of state machine. The state at any program point is the set of locks that are open at that point, and the open and close statements form the state

transitions. A clause $\sigma \Rightarrow A$ in a policy means that A may access any data guarded by that policy in any state where σ is open.

Our lock-based policies also give us an easy way to separate truly secret data from data that is currently secret, but that may be released to other actors under certain circumstances. Assume for instance that payment for auctioned items is done by credit card, and that the server stores credit card numbers in memory locations `aCCNum` and `bCCNum` respectively. Assume further that the line `aBid := aCCnum;` is inserted, either by sheer mistake or through malicious injection, just before where `aBid` is made public. This would release A's credit card number to B, however, the natural policy on `aCCNum` would be $\{A\}$, meaning only A may view this data, ever. Thus when we attempt the assignment above, it will be statically rejected since the policy on `aBid` is too permissive.

All the above are examples of policies to track confidentiality. The dual of confidentiality is integrity, i.e. deciding to what extent data can be trusted, and it should come as no surprise that flow locks can handle both kinds.

Returning to the example with the credit card, we assume that when A gives her credit card number, it must be validated (in some unspecified way) before we can trust it. To this end we introduce a "pseudo" actor T (for "trusted") who should only be allowed to read data that is fully trusted. We then use an intermediate location `tmpACCNum` to hold the credit card number when it is submitted by A. This location is given the policy $\{A; \texttt{ACCVal} \Rightarrow T\}$, stating that this data is trusted only if the lock `ACCVal` is open, which is done when the submitted number has been validated. Once validated we can transfer the value to `aCCNum`, which now has the policy $\{A; T\}$ stating that this data is trusted.[2]

3 A Secure Type and Effect System

In the previous section we used a simple imperative language to give an easy introduction to the concept of flow locks. In this section we define the type system for flow locks in the more general context of an ML-like language with recursion and references (but without polymorphism).

3.1 The language λ_{FL}

The terms and types of our language, dubbed λ_{FL}, are listed in Figure 1.

[2] In order to prevent overwriting this data with a new number that hasn't been validated, we should also be sure to close the lock `ACCVal` once the assignment is done.

$$
\begin{array}{lll}
\textbf{Policies:} & p ::= \{\, c_1; \ldots; c_n \} & c ::= \{\, \sigma_1, \ldots, \sigma_k \} \Rightarrow A \\
\textbf{Values and types:} & v ::= n \mid b \mid () \mid \lambda x.M \mid \ell_{p,\tau} & \\
& \tau ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid (\tau, p) \xrightarrow{\Sigma,p,p,\Sigma} \tau \mid \mathit{ref}_{\,p}\, \tau & \\
\textbf{Terms:} & M ::= v \mid x \mid MM \mid \textbf{if } M \textbf{ then } M \textbf{ else } M \mid \textbf{rec } x.M & \\
& \quad \mid \textbf{ref}_{p,\tau}\ M \mid !M \mid M := M \mid \textbf{open } \sigma \mid \textbf{close } \sigma & \\
\textbf{Derived forms:} & \textbf{let } x = M_1 \textbf{ in } M_2 \equiv (\lambda x.M_2)M_1 & \\
& M_1; M_2 \equiv (\lambda \_.M_2)M_1 &
\end{array}
$$

Figure 1: The λ_{FL} language

The policy language is worth some extra attention. The flow lock policies with which we work assumes a set of *actors* (or *levels*, *principals*) ranged over by A, B, and a set of flow locks ranged over by σ, with Σ for sets of locks. Both actors and flow locks are global in a program. A *policy* is a set of *clauses*, where each clause of the form $\Sigma \Rightarrow A$ states the circumstances (Σ) under which A may view the data governed by this policy. Σ is a set of locks which we name the *guard* of the clause, and interpret it as a conjunction. Thus for the guard to be fulfilled, all the locks in Σ must be open. We can however have more than one clause for the same A, in which case the separate clauses also form a conjunction — A may read the data if either of the guards are fulfilled. In the special case where the guard contains no locks, signifying that the corresponding actor A may always view the data, we write the clause as only A instead of $\{\} \Rightarrow A$. From a logical perspective a policy is just a conjunction of definite Horn clauses, i.e. $\bigwedge_i \{\sigma_{i1} \wedge \cdots \wedge \sigma_{in} \Rightarrow A_i\}$. We implicitly identify policies up to logical equivalence.[3]

Now we can continue with the language itself. Apart from the terms from standard λ calculus with recursion, λ_{FL} has constructs for creating (ref), dereferencing (!) and assigning to (:=) memory locations ($\ell_{p,\tau}$) through references. In addition to the core terms, we can also derive a few useful language constructs as is also shown in Figure 1.

The reference creation construct takes an extra parameter p which is the policy that the contents should be governed by. The same parameter also shows up on the memory locations themselves, together with the base type τ of the contents. In many cases this τ is irrelevant, or clear from the context,

[3]It is worth noting that we do not allow negative flow policies. Our policy language is monotonic, i.e. the more locks that are open, the more flows are allowed.

and in those cases we omit it and just write ℓ_p. Function types are annotated with read and write policies, and start and end states, and arguments are annotated with a reading policy. We discuss the meaning of these when we define the type system. There are also the open and close terms for manipulation flow locks, thereby changing the state of the program.

The semantics of the language is standard, but apart from the term M and a memory μ, the configurations include the current state Σ. This state is the set of currently open locks, which are effected by the execution of **open** and **close** expressions. The small-step semantics of these are simply:

$$\langle \Sigma, \textbf{open}\ \sigma, \mu \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, (), \mu \rangle \qquad \langle \Sigma, \textbf{close}\ \sigma, \mu \rangle \rightarrow \langle \Sigma \setminus \{\sigma\}, (), \mu \rangle$$

It is important to note that the only interaction between a program and the lock state is via the open and close instructions. This is because we are aiming for a completely static verification — we include the lock state in the semantics only to be able to prove properties about flows, but the state is not actually represented at runtime. For this reason we also do not need to consider potential covert channels introduced by the flow lock state.

3.2 Some intuitions about flow-lock security

Before we define our type system, it is useful to get some intuitions about which programs we deem secure/insecure. At this point we only concern ourselves with information leaks arising from direct or indirect data flows. In particular we will not consider timing or termination sensitivity.

A few small example programs are presented on the right. All of these contain insecure direct data flows, except (3). In (1) the contents of $m_{\{B\}}$ may only be read by B, but we are attempting to leak them into a location readable by A. Same thing goes for (2) — even though B can read the contents of the target location, we are still leaking the contents of $m_{\{B\}}$ to A. The simple pattern is that we may not write data to a memory location if that location may be read by someone who cannot already access the data. What's more, this should hold for future time as well. Thus if a reader could access the data from the location we are writing to in some future state, that reader must also have access to the data that is being written, in that same state. Thus the example $m_{\{\sigma \Rightarrow A\}} := !\ell_{\{\sigma \Rightarrow A\}}$ is secure while program (4) is not. In program (5) we attempt to take data not yet readable by A, and put it in a location where A could read it right away. This should clearly not be allowed for the same reasons as for (4).

(1) $\ell_{\{A\}} := !m_{\{B\}}$
(2) $\ell_{\{A;B\}} := !m_{\{B\}}$
(3) $\ell_{\{A\}} := !m_{\{A;B\}}$
(4) $\ell_{\{\sigma \Rightarrow A;B\}} := !m_{\{B\}}$
(5) $\ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$

The lock state in effect at the point of the assignment determines its validity, so the programs (6) and (7) are secure.

(6) $\quad \mathbf{open}\ \sigma; \ell_{\{A\}} := !m_{\{\sigma \Rightarrow A\}}$

(7) $\quad \ell_{\{A\}} := (\mathbf{open}\ \sigma; !m_{\{\sigma \Rightarrow A\}})$

However, we also want a program like (8) below to be considered secure, so we should take the policy of data read from some memory location to be the policy on the location, but taking into account the current state.

(8) $$\ell_{\{A\}} := \mathbf{let}\ x = (\mathbf{open}\ \sigma; !m_{\{\sigma \Rightarrow A\}})\ \mathbf{in}\ (\mathbf{close}\ \sigma; x)$$

In program (8) above, the data read from the reference will thus have the policy $\{A\}$ and not $\{\sigma \Rightarrow A\}$, since it is read in a state where σ is open.

Putting all this slightly more formally, data may be written to a memory location if and only if the policy on the location is at least as restrictive as the one on the data, with respect to the state in effect at the point of the assignment. We give a formal definition of this in the next section.

We must also handle indirect flows that arise from various branching situations. A very simple example program containing an invalid indirect flow is

(9) $$\mathbf{if}\ !\ell_{\{A\}}\ \mathbf{then}\ m_{\{B\}} := \mathbf{true}\ \mathbf{else}\ m_{\{B\}} := \mathbf{false}$$

This program is obviously insecure since it will leak the value of $\ell_{\{A\}}$ into $m_{\{B\}}$, but for some programs it is not so easy to tell. Consider the three programs

(10) $\quad \mathbf{if}\ !\ell_{\{\sigma \Rightarrow A\}}\ \mathbf{then}\ (\mathbf{open}\ \sigma; m_{\{A\}} :=\ \mathbf{true})\ \mathbf{else}\ (\mathbf{open}\ \sigma; m_{\{A\}} :=\ \mathbf{false})$

(11) $\quad \mathbf{if}\ !\ell_{\{\sigma \Rightarrow A\}}\ \mathbf{then}\ (\mathbf{open}\ \sigma; m_{\{A\}} :=\ \mathbf{true}; \mathbf{close}\ \sigma)\ \mathbf{else}\ ()$

(12) $\quad \mathbf{if}\ (\mathbf{open}\ \sigma; !\ell_{\{\sigma \Rightarrow A\}})\ \mathbf{then}\ (\mathbf{close}\ \sigma; m_{\{A\}} :=\ \mathbf{true})\ \mathbf{else}\ ()$

Program (10) could be argued correct since at the points where we leak the information to A, i.e. the assignments, the state allows A to access the result of the branching conditional directly, and hence the leak is secure.

However, as program (11) shows it is not that simple. If the second branch in (11) is chosen, the value of the condition is still leaked to A by the absence of a write, but at no point does the state allow the flow. The leaks come from knowing which of the two branches is taken, which suggests that the leak actually occurs at the branch point. Thus it is the policy of the condition, taken in the state in effect at the branch point, that decides what writes the branches may perform. This means that (9), (10) and (11) are all insecure, while (12) is secure even though the lock is closed again before the write.

Another possible source of indirect leaks is function application. If the function itself is secret, an attacker could still get information about what that function is by observing its effects, just like he could know which branch was taken by observing the effects of a conditional expression. Thus in a sense we can view function application as a kind of branching.

(13) $(!\ell_{\{A\}})\ ()$
(14) $(!\ell_{\{\sigma \Rightarrow A\}})\ ()$
(15) $(!\ell_{\{\sigma \Rightarrow A\}})\ (\mathbf{open}\ \sigma; ())$
(16) $(!\ell_{\{A\}}) := 0$
(17) $(!\ell_{\{\sigma \Rightarrow A\}}) := (\mathbf{open}\ \sigma; 0)$
(18) $(\lambda x.\ell_{\{B\}} := x)\ (!m_{\{A\}})$
(19) $(\lambda x.\ell_{\{B\}} := 0)\ (!m_{\{A\}})$

Consider the programs (13) – (19). In the program (13) we must ensure that the function read from the reference does not write to locations visible by anyone other than A, otherwise we could leak information about which function that was used. As an example, if the function read from $\ell_{\{A\}}$ in (13) is $(\lambda x.m_{\{B\}} := 1)$ or $(\lambda x.m_{\{B\}} := 2)$, B can determine which of the two that was used by reading $m_{\{B\}}$. We treat the application point in the same way as the branch point of a conditional, so in program (14) the body of the function must not write to a location directly visible to A, even if it first opens σ. However, since we have a call-by-value semantics, in program (15) the function body may perform writes to locations directly visible to A, even if it first closes σ, since σ will be open at the application point.

A similar situation is assignment to a reference that in turn has been read from a reference, as illustrated in program (16) which should be disallowed if the reference read from $\ell_{\{A\}}$ is visible to anyone other than A. In particular, the contents of $\ell_{\{A\}}$ could be $m_{\{B\}}$ or $n_{\{B\}}$, in which case B can determine the contents of $\ell_{\{A\}}$ by checking which of the two latter locations that contain the value 0. However, just as for application, program (17) is secure if the reference assigned to has policy $\{A\}$, or any policy that is more restrictive than $\{A\}$, since σ is opened before the assignment takes place.

We also need to look at how functions handle the values passed to them as arguments. Clearly we want to rule out a direct leak in the function body, as the one in example (18). One solution attempt could be to rule out all functions that write to "low" memory, i.e. locations with less restrictive policies that the one placed on the argument. But this also rules out perfectly secure programs such as (19) which in particular would mean that we could not derive a sequential composition form as in figure 1 without placing too heavy restrictions on the writing capabilities of the second sub-program. Thus we want our type system to treat these two programs differently — (18) should be deemed insecure, but not (19).

Other issues such as whether our system is termination sensitive or timing sensitive (see [16] for an overview of these concepts) are orthogonal to the

above discussion. We choose to develop a type system and semantics for termination and timing insensitive security. Termination insensitivity makes the type system simpler but the semantics more complex.

3.3 The Type System

Now we have all the intuition needed to construct the type system. We choose to model our system as a type and effect system in the style of Almeida Matos and Boudol [2]. This means in particular that all expressions will be given a *reading effect* and a *writing effect*. In our system the reading effect of an expression is a policy which states who may read the result of that expression, and in what lock states they may do so. The writing effect is also a policy, which records which actors and in what lock states they can see the memory effect of the expression's execution. Type judgments then have the form

$$\Gamma; \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$$

- Γ is a typing environment for variables giving a type and policy for each variable.
- Σ is the state, i.e. the set of locks currently open.
- τ is the type of the term
- (r, w) are the reading and writing effects of the term, both on the form of policies
- Σ' is the state the program will be in after evaluating the term

First we need to define a few operators on policies that we will use in the typing rules. The aforementioned ordering of how restrictive policies are is defined as

$$p_1 \preceq p_2 \equiv \forall(\Sigma_2 \Rightarrow A) \in p_2.\exists(\Sigma_1 \Rightarrow A) \in p_1.\Sigma_1 \subseteq \Sigma_2$$

Read out, we say that p_1 is less restrictive than p_2 if and only if every clause in p_2 is matched by a clause in p_1 for the same A with a less restrictive guard (one with no additional locks). From the logical perspective, this ordering corresponds directly to implication. The most restrictive policy is $\{\}$, also written $\top$, and data with this policy can never be accessed by anyone. On the other end of the spectrum is $\bot$, defined as the set of all actors in the system. In other words, data marked with $\bot$ can be read by everyone at all times.

To join two policies means combining their respective clauses, thereby forming the logical disjunction. We define

$$p_1 \sqcup p_2 \equiv \{\Sigma_1 \cup \Sigma_2 \Rightarrow A \mid \Sigma_1 \Rightarrow A \in p_1,\ \Sigma_2 \Rightarrow A \in p_2\}$$

It should be intuitively clear that the join of two policies is at least as restrictive as each of the two operands, i.e. $p \preceq p \sqcup p'$ for all p, p'. In contrast, forming the union of two policies, i.e. the meet, corresponding to $\sqcap$ or logical conjunction, makes the result less restrictive, so we have $p \sqcap p' \preceq p$ for all p, p'. Both $\sqcap$ and $\sqcup$ are clearly commutative and associative.

Finally we need to define using a policy with respect to a particular state, or normalising to a state. We say that policy p normalised at state Σ is

$$p(\Sigma) \equiv \{\Sigma' \setminus \Sigma \Rightarrow A \mid \Sigma' \Rightarrow A \in p\}$$

Informally, we remove all open locks from all guards in p, since these no longer restrict data governed by p. This function is antimonotonic, so $\Sigma \subseteq \Sigma' \implies p(\Sigma') \preceq p(\Sigma)$, and in particular $p(\Sigma) \preceq p$ for all Σ. Logically this operation is a partial evaluation, where all variables (locks) that appear in Σ are set to *true* in p.

The type and effect system is presented in Figure 2. The rules for literal values are straight-forward, giving all such values the reading effect bottom. However, from the variable rule we see that variables are given a reading policy. This is used to keep track of the reading policies of function arguments, as can be seen from the rules for abstraction and application, and the purpose is to disallow programs like (18) while still allowing (19). It is important to note that we do *not* check that $r_2(\Sigma_2) \preceq w_f$ in the application rule, since doing so would invalidate program (19). Instead we rely on the type checking of the body of the function to find any leaks inside it, with the help of the annotation on its parameter.

In the rule for abstractions, we annotate the function arrow with the latent read and write effects that will be accurate for the function body once it is applied. We also annotate the arrow with the state that the program will be in at the application point, and the state the program will be in after evaluating the body. The interpretation of a function with type $(\tau, r_\alpha) \xrightarrow{\Delta, r, w, \Delta'} \tau'$ is thus that when applied in state Δ on an argument of type τ and with reading policy r_α, it will produce a result of type τ' with reading policy r. The writing policy w states who could see that the function has been applied, and the whole program will be in state Δ' afterwards. This is all mirrored by the appropriate states in the application rule.

$$\frac{}{\Gamma;\Sigma \vdash n : int, (\bot,\top) \Rightarrow \Sigma} \qquad \frac{}{\Gamma;\Sigma \vdash b : bool, (\bot,\top) \Rightarrow \Sigma}$$

$$\frac{}{\Gamma;\Sigma \vdash \ell_{p,\tau} : \mathbf{ref}_p\ \tau), (\bot,\top) \Rightarrow \Sigma} \qquad \frac{}{\Gamma;\Sigma \vdash () : unit, (\bot,\top) \Rightarrow \Sigma}$$

$$\frac{\Gamma, x : (\tau, r_\alpha); \Delta \vdash M : \tau', (r, w) \Rightarrow \Delta'}{\Gamma;\Sigma \vdash \lambda x.M : (\tau, r_\alpha) \xrightarrow{\Delta, r, w, \Delta'} \tau', (\bot,\top) \Rightarrow \Sigma} \qquad \frac{x : (\tau, r) \in \Gamma}{\Gamma;\Sigma \vdash x : \tau, (r(\Sigma),\top) \Rightarrow \Sigma}$$

$$\frac{}{\Gamma;\Sigma \vdash \mathbf{open}\ \sigma : unit, (\bot,\top) \Rightarrow \Sigma \cup \{\sigma\}} \qquad \frac{}{\Gamma;\Sigma \vdash \mathbf{close}\ \sigma : unit, (\bot,\top) \Rightarrow \Sigma \setminus \{\sigma\}}$$

$$\frac{\Gamma, x : (\tau, r); \Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma}{\Gamma;\Sigma \vdash \mathbf{rec}\ x.M : \tau, (r, w) \Rightarrow \Sigma}$$

$$\frac{\Gamma;\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'}{\Gamma;\Sigma \vdash \mathbf{ref}_p\ M : \mathbf{ref}_p\ \tau, (\bot, w \sqcap p) \Rightarrow \Sigma'} \qquad \frac{\Gamma;\Sigma \vdash M : \mathbf{ref}_p\ \tau, (r, w) \Rightarrow \Sigma'}{\Gamma;\Sigma \vdash !M : \tau, (r \sqcup p(\Sigma'), w) \Rightarrow \Sigma'}$$

$$\frac{\Gamma;\Sigma \vdash M_1 : \mathbf{ref}_p\ \tau, (r_1, w_1) \Rightarrow \Sigma' \quad \Gamma;\Sigma' \vdash M_2 : \tau, (r_2, w_2) \Rightarrow \Sigma''}{\Gamma;\Sigma \vdash M_1 := M_2 : unit, (\bot, w_1 \sqcap w_2 \sqcap p) \Rightarrow \Sigma''}\ r_1(\Sigma'') \sqcup r_2(\Sigma'') \preceq p$$

$$\frac{\Gamma;\Sigma \vdash M_0 : bool, (r_0, w_0) \Rightarrow \Sigma' \quad \Gamma;\Sigma' \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma_i \quad r_0(\Sigma') \preceq w_1 \sqcap w_2}{\Gamma;\Sigma \vdash \mathbf{if}\ M_0\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \tau, (r_0 \sqcup r_1 \sqcup r_2, w_0 \sqcap w_1 \sqcap w_2) \Rightarrow \Sigma_1 \cap \Sigma_2}$$

$$\frac{\begin{array}{c} r_1(\Sigma_2) \preceq w_f \\ \Gamma;\Sigma \vdash M_1 : (\tau, r_2) \xrightarrow{\Sigma_2, r_f, w_f, \Sigma_3} \tau', (r_1, w_1) \Rightarrow \Sigma_1 \quad \Gamma;\Sigma_1 \vdash M_2 : \tau, (r_2, w_2) \Rightarrow \Sigma_2 \end{array}}{\Gamma;\Sigma \vdash M_1\ M_2 : \tau', (r_1 \sqcup r_f, w_1 \sqcap w_2 \sqcap w_f) \Rightarrow \Sigma_3}$$

Figure 2: Type and Effect system

Direct leaks, like the ones in programs (1), (2), (4) and (5), are handled by the check $r_2(\Sigma'') \preceq p$ in the rule for assignment. Since we normalise the policy r_2 of the assignee to the state in effect at the point of the assignment, program (5) would be secure if run in a state where σ is open, which is exactly what happens in programs (6) and (7). Also the normalisation to the current state in the dereferencing rule, i.e. $p(\Sigma')$ in the reading effect of the conclusion, means that program (8) will be deemed secure. The same kind of normalisation also appears in the variable rule.

The check $r_0(\Sigma') \preceq w_1 \sqcap w_2$ in the conditional rule will ensure that an indirect leak like the one in (9) will not be allowed. The normalisation of r_0 to Σ' means that it is the state at the branch point that is important, which disallows (10) and (11) but lets (12) through. The branches may open and close different locks, so the end states can differ. Since policies are monotonic, we can use the intersection of the end states as a safe approximation for the following program.

The checks $r_1(\Sigma'') \preceq p$ in the assignment rule, and the corresponding $r_1(\Sigma_2) \preceq w_f$ in the application rule handle indirect flows like in (13), (14) and (16), but allow (15) and (17).

In the assignment rule, the reading effect in the conclusion is $\bot$. The reason is that the result of an assignment is always (), independent of the result values of the two expressions M_1 and M_2, so no information is leaked by making the () result public. For similar reasons, r_2 does not show up in the reading effect in the conclusion of the application rule. Since function arguments are annotated with their reading effects, if the result of M_2 has any effect on the result of the whole application expression, this fact will be seen through r_f.[4]

4 Semantic Security Properties

In this section we define the semantic security property appropriate for flow locks, and outline the proof that the flow lock type system does indeed satisfy this property.

4.1 CORE$_{FL}$

The first observation we make, which we will explain in more depth in section 4.5, is that the λ_{FL} language and the given substitution semantics are not well suited when defining the semantic security property. In order to assert the properties we require, we need to be able to reason about values resulting from evaluating each subterm, and λ_{FL} does not give us the means to do this.

To this end we define a monadic core language, CORE$_{FL}$, defined in figure 3. The main difference from λ_{FL} is that we have made sequential computation explicit in the language by the introduction of a bind construct. All other terms in the language have been syntactically restricted to contain no subterms other than variables in positions suited for reduction. Another difference is that variables are now annotated with a policy and a type, just like locations. This means that references need not be typed since their type is given by the annotion on the variable argument. We use boldface metavariables $\boldsymbol{x}$, $\boldsymbol{y}$ etc., to range over policy- and type- annotated variables of the form $x_{p,\tau}$, $y_{p',\tau'}$.

[4] The rules involving functions are fairly restrictive as they are formulated here. One could easily imagine various forms of subsumption, both for lock states and argument policies, that would make the system less restrictive. However, adding subsumption would complicate the overall formulation of the type system, so we leave it for now.

$$
\begin{array}{lll}
\textbf{Annotated variables} & \boldsymbol{x} & ::= x_{p,\tau} \\
\textbf{Values and types:} & v & ::= n \mid b \mid () \mid \lambda x.M \mid \ell_{p,\tau} \\
 & \tau & ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit} \mid (\tau, p) \xrightarrow{\Sigma,p,p,\Sigma} \tau \mid \mathit{ref}_p\, \tau \\
 & & \\
\textbf{Terms:} & M & ::= v \mid \boldsymbol{x} \mid \boldsymbol{x}\, \boldsymbol{y} \mid \textbf{if } \boldsymbol{x} \textbf{ then } M \textbf{ else } M \mid \textbf{rec } \boldsymbol{x}.v \\
 & & \quad \mid \textbf{ref}_p\, \boldsymbol{x} \mid !\boldsymbol{x} \mid \boldsymbol{x} := \boldsymbol{y} \mid \textbf{open } \sigma \mid \textbf{close } \sigma \\
 & & \quad \mid \textbf{bind } \boldsymbol{x} = M \textbf{ in } M
\end{array}
$$

Figure 3: The CORE_{FL} language

4.2 Semantics

The semantics for CORE_{FL}, presented in figure 4, and is given by single-step labelled transitions of the form

$$\langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', N, S' \rangle$$

where

- Σ is the set of flow locks currently open,
- M is the term being computed,
- S is the *store*: a finite mapping from annotated values and locations to CORE_{FL} values.
- p records the policy relating to any store access that that takes place during that step (and is simply $\top$ if there is no memory access in that step).

We assume the usual well-formedness conditions for configurations $\langle \Sigma, M, S \rangle$, namely that the free variables and the locations in M and in the range of S are in the domain of S.

We will write $\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', N, S' \rangle$ to mean $\exists p. \langle \Sigma, M, S \rangle \xrightarrow{p} \langle \Sigma', N, S' \rangle$, and $\langle \Sigma, M, S \rangle \Uparrow$ to mean that the configuration diverges – i.e. can be reduced indefinitely

$$\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma_0, N_0, S_0 \rangle \rightarrow \cdots \rightarrow \langle \Sigma_i, N_i, S_i \rangle \rightarrow \cdots$$

$$\langle \Sigma, x_{p,\tau}, S\rangle \xrightarrow{p} \langle \Sigma, S(x_{p,\tau}), S\rangle$$

$$\langle \Sigma, \mathbf{ref}\ x_{p,\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, \ell_{p,\tau}, S[\ell_{p,\tau} \mapsto S(x_{p,\tau})]\rangle \quad \ell_{p,\tau} \notin dom(S)$$

$$\langle \Sigma, !x_{p,\tau}, S\rangle \xrightarrow{p \sqcap p'} \langle \Sigma, S(S(x_{p,\tau})), S\rangle \quad \text{where } S(x_{p,\tau}) = \ell_{p',\tau'}$$

$$\langle \Sigma, \boldsymbol{x} := \boldsymbol{y}, S\rangle \xrightarrow{\top} \langle \Sigma, (), S[S(\boldsymbol{x}) \mapsto S(\boldsymbol{y})]\rangle$$

$$\langle \Sigma, \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_0, S\rangle \quad \text{if } S(x_{p,\tau}) = \mathbf{true}$$

$$\langle \Sigma, \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_1, S\rangle \quad \text{if } S(x_{p,\tau}) = \mathbf{false}$$

$$\langle \Sigma, x_{p,\tau}\ \boldsymbol{y}, S\rangle \xrightarrow{p} \langle \Sigma, M[\boldsymbol{y}/\boldsymbol{z}], S\rangle \quad \text{where } S(x_{p,\tau}) = \lambda \boldsymbol{z}.M,\ \boldsymbol{z} \text{ fresh}$$

$$\langle \Sigma, \mathbf{open}\ \sigma, S\rangle \xrightarrow{\top} \langle \Sigma \cup \{\sigma\}, (), S\rangle$$

$$\langle \Sigma, \mathbf{close}\ \sigma, S\rangle \xrightarrow{\top} \langle \Sigma \setminus \{\sigma\}, (), S\rangle$$

$$\langle \Sigma, \mathbf{rec}\ \boldsymbol{x}.v, S\rangle \xrightarrow{\top} \langle \Sigma, v, S[\boldsymbol{x} \mapsto v]\rangle$$

$$\langle \Sigma, \mathbf{bind}\ \boldsymbol{x} = v\ \mathbf{in}\ M, S\rangle \xrightarrow{\top} \langle \Sigma, M, S[\boldsymbol{x} \mapsto v]\rangle \quad \boldsymbol{x} \notin dom(S)$$

$$\frac{\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle}{\langle \Sigma, \mathbf{bind}\ \boldsymbol{x} = M\ \mathbf{in}\ N, S\rangle \xrightarrow{p} \langle \Sigma', \mathbf{bind}\ \boldsymbol{x} = M'\ \mathbf{in}\ N, S'\rangle}$$

Figure 4: Store-based semantics for CORE_{FL}

$$\frac{}{\Sigma \vdash n : int, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash b : bool, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{}{\Sigma \vdash \ell_{p,\tau} : ref_{\,p}\,\tau, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash () : unit, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{\Delta \vdash M : \tau, (r_f, w_f) \Rightarrow \Delta'}{\Sigma \vdash \lambda x_{p',\tau'}.M : (\tau', p') \xrightarrow{\Delta, r_f, w_f, \Delta'} \tau, (\bot, \top) \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash x_{p,\tau} : \tau, (p(\Sigma), \top) \Rightarrow \Sigma}$$

$$\frac{}{\Sigma \vdash \textbf{open}\ \sigma : unit, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}} \qquad \frac{}{\Sigma \vdash \textbf{close}\ \sigma : unit, (\bot, \top) \Rightarrow \Sigma \setminus \{\sigma\}}$$

$$\frac{\Sigma \vdash v : \tau, (\bot, \top) \Rightarrow \Sigma}{\Sigma \vdash \textbf{rec}\ x_{\bot,\tau}.v : \tau, (\bot, \top) \Rightarrow \Sigma}$$

$$\frac{p(\Sigma) \preceq p'}{\Sigma \vdash \textbf{ref}_{p'}\ x_{p,\tau} : \textbf{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma} \qquad \frac{}{\Sigma \vdash\ !x_{p,ref_{\,p'}\,\tau} : \tau, (p(\Sigma) \sqcup p'(\Sigma), \top) \Rightarrow \Sigma}$$

$$\frac{p(\Sigma'') \sqcup p'(\Sigma'') \preceq p''}{\Sigma \vdash x_{p,ref_{\,p''}\,\tau} := y_{p',\tau'} : unit, (\bot, p'') \Rightarrow \Sigma}$$

$$\frac{\Sigma' \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma' \qquad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \textbf{if}\ x_{p,bool}\ \textbf{then}\ M_0\ \textbf{else}\ M_1 : \tau, (r_0 \sqcup r_1 \sqcup p(\Sigma), w_0 \sqcap w_1) \Rightarrow \Sigma'}$$

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}\ \textit{where}\ \tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$$

$$\frac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau', (r_1, w_1) \Rightarrow \Sigma'' \quad r_0(\Sigma') \preceq p}{\Sigma \vdash \textbf{bind}\ x_{p,\tau} = M_0\ \textbf{in}\ M_1 : \tau', (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$$

Figure 5: Specialized Type and Effect system for $CORE_{FL}$

4.3 Type System for CORE$_{FL}$

The type system for λ_{FL} is valid also for CORE$_{FL}$ terms with the addition of a typing rule for the bind construct. However, since CORE$_{FL}$ terms are simpler than their λ_{FL} counterparts, we can specialise the type rules for CORE$_{FL}$ terms, and use the simpler formulations to good effect in our proofs. The result of this specialisation can be found in figure 5. Note that the type environment is now redundant since each variable carries its type.

We can establish some standard properties relating well-typed programs and reduction: *progress*, which says that well-typed programs do not get "stuck", and *preservation* (subject reduction), which says roughly that well-typed terms reduce to well-typed terms. We simply state these properties as

lemmas here while the proofs are given in appendix A.1.

Lemma 1 (Progress). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *then either*

- $M \in \mathrm{Val}$, *or*
- *for all* S *such that* $\mathrm{dom}(S) \supseteq \mathrm{fv}(M) \cup \mathrm{loc}(M)$ *and* $\vdash S$ *then* $\exists \Sigma', M', S'. \langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', M', S' \rangle$.

Lemma 2 (Preservation). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *and* $\vdash S$ *and* $\mathrm{dom}(S) \supseteq \mathrm{fv}(M) \cup \mathrm{loc}(M)$ *and* $\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', M', S' \rangle$ *then* $\vdash S'$ *and* $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ *where* $r' \preceq r$ *and* $w \preceq w'$.

4.4 Semantic security property

To prove standard noninterference one needs to show that the observable behaviour of a program, from the perspective of a given actor, does not change when the values of secrets (things not readable by that actor) are changed. At the top level we may settle for a notion of "observable behaviour" to mean the results of computations — the final state or values.

In the next section we will show that our notion of flow lock security does indeed imply a standard noninterference property. However, since we have dynamic policies we are forced to consider the intermediate states of a computation, because it is at such state that the policy may change.

Visibility An actor α can directly observe the contents of a memory location $\ell_{p,\tau}$ in lock state Σ, when there is a clause $\Sigma' \Rightarrow \alpha \in p$ such that $\Sigma' \subseteq \Sigma$, or equivalently, when $\{\} \Rightarrow \alpha \in p(\Sigma)$. In this case we sometimes say that α *can see* p *at* Σ. This kind of property is used often, so we introduce some specific notations:

Definition 1 (Visibility).

$$\alpha \lhd p \stackrel{\text{def}}{=} (\{\} \Rightarrow \alpha) \in p \qquad (\alpha \text{ can see } p)$$

$$\alpha \not\lhd p \stackrel{\text{def}}{=} \neg(\alpha \lhd p) \qquad (\alpha \text{ can't see } p)$$

$$\alpha \not\lhd^{\Omega} p \stackrel{\text{def}}{=} \forall \Theta. \alpha \not\lhd p(\Theta \setminus \Omega) \qquad (\alpha \text{ can't see } p \text{ without } \Omega)$$

$$guards_{\alpha}(p) \stackrel{\text{def}}{=} \begin{cases} \{\{\}\} & \text{if } \alpha \lhd p \\ \{\Phi \mid \Phi \Rightarrow \alpha \in p\} & \text{otherwise} \end{cases} \qquad (\text{The guards of } \alpha \text{ in } p)$$

The last of these definitions, the guards of an actor α in policy p, definies the sets of locks which have an influence on the visibility of the policy to α. We can connect the guards of a policy and its visibility through the following lemma:

Lemma 3 (Guard lemma). *If $\alpha \not\lhd p$, then $\alpha \not\lhd^{\Omega} p$ where $\Omega = \bigcup guards_{\alpha}(p)$.*

The proof of this lemma can be found in appendix A.2

For the visibility operators we note that the $\alpha \lhd p$ relation is anti-monotonic in its policy argument, i.e.

$$\alpha \lhd p \;\&\; p' \preceq p \implies \alpha \lhd p'$$

Clearly $\alpha \not\lhd p$ and $\alpha \not\lhd^{\Omega} p$ are then monotonic. Also $\alpha \not\lhd^{\Omega} p$ is monotonic in its lock-set argument, i.e.

$$\alpha \not\lhd^{\Omega} p \;\&\; \Omega' \supseteq \Omega \implies \alpha \not\lhd^{\Omega'} p$$

Actor indistinguishable stores In order to charactersise when information has leaked we first need to characterise when two stores are indistinguishable for a given actor. In order to do this we need to take into account which locks are open. Once we know wich locks are open we can compute which parts of the store are visible to the actor.

Definition 2 (α-indistinguishable stores $=_{\alpha}^{\Theta}$). Define two stores S and T to be indistinguishable by α at lock state Θ, written $S =_{\alpha}^{\Theta} T$, if the location domains of S and T are the same, and for all policies p such that $\alpha \lhd p(\Theta)$,

1. for all locations $\ell_{p,\tau}$ in S and T we have $S(\ell_{p,\tau}) = T(\ell_{p,\tau})$, and
2. for all variables $x_{p,\tau} \in \mathrm{dom}(S) \cap \mathrm{dom}(T)$ we have $S(x_{p,\tau}) = T(x_{p,\tau})$.

The definition asserts the equality, in S and T respectively, of locations $\ell_{p,\tau}$ and variables $x_{p,\tau}$ which are visible to actor α at lock state Θ. The stronger requirement on locations – that S and T have the same locations – is due to the fact that locations are first class values that can be passed around and inspected, and their values can be updated, so an actor can potentially observe the presence or absence of a given memory location in a store. Variables on the other hand can never be observed directly.

The relation $=_{\alpha}^{\Theta}$ is not transitive in general since the domains may vary freely in the parts that deal with variables. As an example of this we could have $\{x_{\perp,\tau} \mapsto v\} =_{\alpha}^{\Theta} \{\}$ and $\{x_{\perp,\tau} \mapsto v'\} =_{\alpha}^{\Theta} \{\}$, but clearly not $\{x_{\perp,\tau} \mapsto v\} =_{\alpha}^{\Theta} \{x_{\perp,\tau} \mapsto v'\}$.

However, we are going to need to argue about transitivity in our proofs, so we need to assert that transitivity holds for a certain domain of memories. In particular we can show that $S =_{\alpha}^{\Theta} S'$ and $S =_{\alpha}^{\Theta} T$ gives $S' =_{\alpha}^{\Theta} T$, assuming that $\mathrm{dom}(S') \backslash \mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$. We would then have that $\mathrm{dom}(S') \cap \mathrm{dom}(T) \subseteq \mathrm{dom}(S) \cap \mathrm{dom}(T)$, and thus for all variables $x_{p,\tau} \in \mathrm{dom}(S') \cap \mathrm{dom}(T)$ we have $S'(x_{p,\tau}) = T(x_{p,\tau})$ as required.

Whenever we argue transitivity in our proofs, we implicitly mean this restricted form, but the condition on domains will always be true in the contexts where we use it.

Flow lock security Our definition of flow-lock security follows the "self-bisimulation" approach from [17], whereby security is characterised by a more general property of two programs being bisimilar with respect the the observable parts of memory. One particular feature of the definition from [17] is that the bisimulation is defined over programs and not configurations (program-memory pairs). The idea is that at each step of the bisimulation the pair of programs under comparison are inspected in all pairs of memory states which are indistinguishable to the attacker. This very strong requirement was needed to make the definition of security compositional with respect to parallel composition. But this approach of "resetting" the store at each step has another very useful property: it enables us to reset the state in the event of a policy change. For example, one particular difficulty is that when the current policy becomes *more* restrictive — in our case when locks are closed — then we need a way to reestablish a stronger security requirement at that point in the execution. It is notable that two previous semantic accounts of temporary policy weakening mechanisms, Mantel and Sands's language based intransitive noninterference condition [8], and Almeida Matos and Boudol's *nondisclosure* policy [2], both rely on such a "resetting" bisimulation not only to deal with threads, but more importantly to provide a semantics to local policy change mechanisms. Our definition is close in spirit to Almeida Matos and Boudol's definition, although our less structured (more general) policy-change mechanism creates additional problems.

Without further ado, we now provide the definition of bisimulation upon which our notion of security is based.

Definition 3 ($\sim_\alpha^\Omega$). For any actor α let $\{\sim_\alpha^\Omega\}$ be the lock-set indexed family (i.e. Ω is a set of locks) of relations defined to be the largest symmetric relations on *preconfigurations* (lockstate-term pairs) such that if

$$\langle \Sigma, M\rangle \sim_\alpha^\Omega \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =_\alpha^\Theta T \;\&\; \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$$

then there exists Δ', N', T' such that

$$\text{either } \langle \Delta, N, T\rangle \rightarrow^* \langle \Delta', N', T'\rangle \;\&\; S' =_\alpha^{\Theta\backslash\Omega} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha^{\Omega'} \langle \Delta', N'\rangle,$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

Now we can state that a program is secure if and only if it is bisimilar to itself:

Definition 4 (Flow-lock security). We say that a term M is flow-lock secure, written $M \in FL$, if and only if $\langle \{\}, M \rangle \sim_\alpha^{\{\}} \langle \{\}, M \rangle$

4.5 The bisimulation definition explained

We will try to explain our definition using a sequence of "attempts", each of which introduces parts of the final solution. These are:

1. Bisimulation up to nontermination – adding termination insensitivity to a configuration-level bisimulation-based noninterference condition.
2. A location-resetting bisimulation – adding lock states to the bisimulation, and motivating the "resetting" style of bisimulation.
3. A store-resetting bisimulation – motivating why we have to reset not only the locations but also the variables
4. Future-sensitive bisimulation – why we have to quantify over all lock states which include the current lock state;
5. Past-sensitive bisimulation – why we have to add the lockset Ω.

Let us begin with a view of an attacker (an actor) who can observe intermediate states of computation, but not the speed of computation. Let us further suppose a simple semantics without lockstate, and in which the state is just a mapping for locations (ranged over by μ and ν), and that there are no free variables in the term (i.e. we have a suubstitution semantics). Intuitively, any program when run with two inputs which are indistinguishable to an actor should produce intermediate states indistinguishable to that actor. With no flow locks and only static policies, a possible bisimulation formulation could be of the form:

Attempt 1 (Bisimulation up to nontermination). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation such that if $\langle M, \mu \rangle \sim_\alpha \langle N, \nu \rangle$ then $\mu =_\alpha \nu$, and if $\langle M, \mu \rangle \rightarrow \langle M', \mu' \rangle$ then there exists N', ν' such that either $\langle N, \nu \rangle \rightarrow^* \langle N', \nu' \rangle$ and $\langle M', \mu' \rangle \sim_\alpha \langle N', \nu' \rangle$, or $\langle N, \nu \rangle \Uparrow$.

We use here the obvious notion of low-equivalence of stores, $=_\alpha$, which ensures that we start with inputs that do not differ in the public parts, i.e. locations visible to α. To match a single computation step from the first configuration we can take zero or more steps. This makes the definition

insensitive to timing issues. The divergence clause is added simply to make the definition termination insensitive, so that we cannot (by choice) detect leaks which are encoded in the termination behaviour alone.

This definition is clearly inadequate in the presence of locks. Our next step is to observe that we need to define the bisimulation relation over $\langle \Sigma, M \rangle$ pairs, which we call *preconfigurations*. This is because in order to characterise which states are indistinguishable to a given actor α we need to know the lock state. With dynamic policies we need to take into account the fact that when the policy changes, memory locations that were previously considered secret could now be public, and vice versa. We handle this, as mentioned previously, by resetting the memory at each computation step. This brings us to our second attempt:

Attempt 2 (Memory-resetting bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M \rangle \sim_\alpha \langle \Delta, N \rangle \;\&\; \langle \Sigma, M, \mu \rangle \xrightarrow{p} \langle \Sigma', M', \mu' \rangle \;\&\; \mu =^{\Sigma}_{\alpha} \nu$$

then there exists Δ', N', ν' such that

$$\text{either } \langle \Delta, N, \nu \rangle \to^* \langle \Delta', N', \nu' \rangle \;\&\; \mu' =^{\Sigma}_{\alpha} \nu' \;\&\; \langle \Sigma', M' \rangle \sim_\alpha \langle \Delta', N' \rangle,$$

$$\text{or } \langle \Delta, N, \nu \rangle \Uparrow,$$

This definition is somewhat similar in spirit to non-disclosure [2]. For the moment we still view stores as containing memory locations only, and thus assume a semantics which avoids free variables altogether. This attempt takes into account that the effective secrecy status of memory locations can change during program execution, but this is not enough. In this rich language it is also possible to do the same for values that have been computed in the term, as shown by program (16) in section 3:

$$(!\ell_{\{\sigma \Rightarrow A\}}) := (\mathbf{open}\ \sigma; ())$$

In this example, we first compute a value on the left-hand side, which will be given the reading policy $\{\sigma \Rightarrow A\}$). From the point of view of A, this is a secret value, and could thus be different values in different runs of the program. However, when we compute the right-hand side, the value on the left-hand side is declassified, though it can still be different values in different runs, which means we could have an α-observable difference in the output of the two programs. This difference is fine though, since we explicitly changed the state to allow the flow to α, but we must still ensure that there are no other observable differences that do not arise from the newly opened lock.

To check this, we want to continue the bisimulation but assume that we in fact had the same value on the left-hand side, and continue as before. This is the same thing that we do when "resetting" the memories, but we need to do the same thing for values in the term. In order to do this for values, we need a handle on those values, which is the motivation behind using the monadic CORE_{FL} language and the store-based operational semantics. Thus we arrive at our third attempt:

Attempt 3 (Store-resetting bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M\rangle \sim_\alpha \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \\ \&\; S =^{\Sigma}_{\alpha} T \textit{ where } \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$$

then there exists Δ', N', T' such that

$$\text{either } \langle \Delta, N, T\rangle \rightarrow^* \langle \Delta', N', T'\rangle \;\&\; S' =^{\Sigma}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha \langle \Delta', N'\rangle,$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

The main difference from the previous attempt is not in the formulation itself, but rather in the use of stores S and T ranging over both variables and locations instead of memories μ and ν, and the corresponding different formulation of the $=^{\Sigma}_{\alpha}$ relation.

The condition $\mathrm{dom}(S')\backslash\mathrm{dom}(S)\cap\mathrm{dom}(T) = \{\}$ is just a hygiene condition that states that new variables introduced in S' are chosen to be distinct from the variables already present in T. Since the operational semantics is free to choose any locations this is not a restriction *per se*. In the "attempts" that follow we will tacitly elide this hygiene condition, but it is needed in all cases.

This definition of bisimulation is still not strong enough. It is not enough to require only that memories should be α-indistinguishable in the current state. A program such as $\ell_{\{\sigma \Rightarrow A\}} := !m_{\{\sigma' \Rightarrow A\}}$ is not secure (unless σ' is open), but with the above definition both locations would be considered unobservable by α, and hence no α-observable differences could be observed. The problem is that this insecure flow might only be revealed at some *future* time. To capture this problem we need to check the α-indistinguishability of the two memories in a state where σ is open but σ' is not. More generally, we must take into account all possible (more permissive) future lock states. Thus our fourth attempt at a definition is:

Attempt 4 (Future-sensitive bisimulation). For any actor α, define $\sim_\alpha$ to be the largest symmetric relation on preconfigurations such that if

$$\langle \Sigma, M\rangle \sim_\alpha \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =^{\Theta}_{\alpha} T$$

then there exists Δ', N', T' such that

$$\text{either } \langle \Delta, N, T\rangle \to^* \langle \Delta', N', T'\rangle \;\&\; S' =^{\Theta}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha \langle \Delta', N'\rangle,$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

Now we will rule out programs like the one above, but we're still not quite there. The final problem is that the definition is now actually too strong – it rules out some (well-typed) programs that should be considered secure, such as (somewhat simplified)

$$\textbf{if } x_{\{\sigma \Rightarrow \alpha\},\tau} \textbf{ then } \ell_{\{\sigma \Rightarrow \alpha\}} := 0 \textbf{ else } ().$$

The indirect flow from x to ℓ should be fine since they have the same policy, but since x is considered secret to α, the above definition requires us to show (after one computation step) that $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := 0\rangle \sim_\alpha \langle \{\}, ()\rangle$, which clearly does not hold $\forall \Theta \supseteq \Sigma$; in particular it will not hold when $\sigma \in \Theta$.

The problem is that opening σ means that the condition that we branched on becomes visible to α as well, but we've passed that point in the program and don't have access to the condition any more. To be sure we don't rule out programs such as these we must remember what branches we have taken, and in particular what possible future states that could make any of the branches visible to α, and make sure that we ignore leaks in those states. Thus our fifth and final attempt is formulated by parameterising the bisimulation relation by the set of locks that were closed at earlier branching points, to ensure that we are not future-sensitive to these locks.

Attempt 5 (Past-aware bisimulation). For any actor α let $\{\sim^{\Omega}_{\alpha}\}$ be the lock-set indexed family of relations defined to be the largest symmetric relations on preconfigurarions such that if

$$\langle \Sigma, M\rangle \sim^{\Omega}_{\alpha} \langle \Delta, N\rangle \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =^{\Theta}_{\alpha} T$$

then there exists Δ', N', T' such that

$$\text{either } \langle \Delta, N, T\rangle \to^* \langle \Delta', N', T'\rangle \;\&\; S' =^{\Theta\setminus\Omega}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle \sim^{\Omega'}_{\alpha} \langle \Delta', N'\rangle,$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

The difference to the previous attempt is that we allow stores to differ after computation as long as those differences are only visible in certain states — in particular those states in which a previous branching point would not have led to a branch at all.

This definition is less restrictive than the former in order to not rule out programs with indirect flows like the one presented above. There might be some concern as to whether this definition is now too weak, since we allow stores to differ in certain states. In particular, what of a direct leak observable only in such a state, like in the program **if** $x_{\{\sigma \Rightarrow \alpha\},\tau}$ **then** $\ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'}$ **else** ()? This leak will indeed not be caught when we are working with $\Omega = \{\sigma\}$, so we have $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle \sim_\alpha^{\{\sigma\}} \langle \{\}, () \rangle$. But recall that we still quantify over all $\Theta \supseteq \{\}$ when considering the conditional expression. Then for any $\Theta \supseteq \{\sigma\}$ the variable whose value we branch on will be considered public, and we will continue with the same branch in both cases. Also since the variable was public, there will be no states in which we allow future memories to differ in what α can see, and we must have $\langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle \sim_\alpha^{\{\}} \langle \{\}, \ell_{\{\sigma \Rightarrow \alpha\}} := y_{\top,\tau'} \rangle$ which cannot hold.

This fifth attempt is our actual definition of a bisimulation.

4.6 Non-circular reasoning for bisimulation

As the sharp-eyed reader may well have noticed, our notion of a bisimulation implicitly constrains the stores used to be well-typed, i.e. if a location or variable is said to hold an integer value, it does indeed hold an integer value. This is not an unreasonable assumption to make in general, and since we reset the stores before each computation step and require the bisimulation properties to be fulfilled for *any* stores that are equal, it is an assumption that is crucial for this to work at all. It would be impossible for all but the simplest programs to be considered secure otherwise.

But unfortunately this assumption leads to a circular reasoning when we want to prove that our type system guarantees flow lock security. We allow the store to contain not only simple values like ints, but also functions with arbitrary terms as their bodies. In order to show that such a value is well-typed we need to use the full power of the type system.

Thus we end up in a situation where we want to show that well-typed terms are bisimilar to themselves, but the notion of bisimilarity already depends on the type system. To break this loop we can give a more general definition of a bisimulation where we parametrise the relation on some well-formedness predicate on stores:

Attempt 6 (Parametrised bisimulation). For any actor α, let $\sim_\alpha^\Omega$ be the lock-set indexed family of relations defined to be the largest symmetric relations on preconfigurations such that if

$$\langle \Sigma, M\rangle \sim_\alpha^\Omega \langle \Delta, N\rangle \;\&\; \mathcal{P}(S) \;\&\; \langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \mathcal{P}(S') \;\&\; \Theta \supseteq \Sigma \;\&\; S =_\alpha^\Theta T \;\&\; \mathcal{P}(T)$$

then there exits Δ', N', T' such that

$$\text{either } \langle \Delta, N, T\rangle \to^* \langle \Delta', N', T'\rangle \;\&\; \mathcal{P}(T') \;\&\; S' =_\alpha^{\Theta\setminus\Omega} T' \;\&\; \langle \Sigma', M'\rangle \sim_\alpha^{\Omega'} \langle \Delta', N'\rangle,$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

We would then prove that well-typed terms are bisimilar to themselves, assuming they start off in well-typed, well-formed stores, i.e. $\mathcal{P}(S) = \vdash S$. Subject reduction gives us that the typeability of stores is retained, so this would not complicate the proofs the least.

This is the complete, most general definition of a bisimulation. The previous definition, which we actually use, can be seen as an instantiation of this definition for the proper $\mathcal{P}$, and we will use that one for simplicity.

4.7 Well-typed Programs are Flow-Lock Secure

We now want to prove that all programs typeable with our type system are indeed secure. The proof follows a similar structure to the corresponding proof from Almeida Matos and Boudol [2].

The basic approach is to utilise the coinductive nature of the bisimulation definition. We show that for well-typed closed M, $\langle \emptyset, M\rangle \sim_\alpha \langle \emptyset, M\rangle$ by construction of a candidate relation R_α^α, that in particular contains the pair $(\langle \emptyset, M\rangle, \langle \emptyset, M\rangle)$, and which can be shown to be an α-bisimulation. This gives us that $(\langle \emptyset, M\rangle, \langle \emptyset, M\rangle) \in R_\alpha^\alpha \subseteq \sim_\alpha$.

To be able to define the candidate relation R_α^Ω we need the notion of programs that are *high* with respect to some actor α. We say that a program is α-Ω-high if it does not modify any locations that α could see while all the locks in Ω remains closed. However, this operational notion of being high is a bit akward to work with, so instead we use a stronger, syntactic notion stating that a program is *syntactically* α-Ω-high if it does not *write* to any locations that α could see while the locks in Ω remain closed.

Definition 5 (Syntactically α-Ω-high programs: H_α^Ω). Let H_α^Ω be the set of all terms M such that $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\alpha \not\preccurlyeq^\Omega w$.

Now we can define our candidate relation:

Definition 6 (Candidate relation R_α^Ω). Let R_α^Ω be a symmetric relation on well-typed preconfigurations, inductively defined as follows:

$$1\frac{}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, M\rangle} \qquad 2\frac{M, N \in H_\alpha^\Omega}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle}$$

$$3\frac{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Sigma, N\rangle \quad \alpha \not\preccurlyeq^\Omega p}{\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = M \textbf{ in } M']\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N \textbf{ in } M']\rangle}$$

where $\mathbb{E}[\cdot]$ are the evaluation contexts for CORE_{FL}, given by

$$\mathbb{E}[\cdot] ::= [\cdot] \mid \textbf{bind } \boldsymbol{x} = \mathbb{E}[\cdot] \textbf{ in } M$$

In words, two well-typed preconfigurations are related by R_α^Ω if the programs in them are either equal, both are high, or they are two sub-programs related by R_α^Ω inside nested (equal) bind constructs, where the results of those sub-computations are secret to α. The lock-state components constrain what preconfigurations are in the relation only through the typeability requirement.

The final piece of the puzzle is now to show that this candidate relation is indeed a bisimulation.

Lemma 4 ($\bigcup_\Omega R_\alpha^\Omega$ is a bisimulation). *If $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ and $\vdash S$ and*

$$\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =_\alpha^\Theta T \;\&\; \vdash T$$

then $\vdash S'$, and there exists Δ', N', T' such that

either $\langle \Delta, N, T\rangle \rightarrow^* \langle \Delta', N', T'\rangle \;\&\; \vdash T'$
$\;\&\; S' =_\alpha^{\Theta\setminus\Omega} T' \;\&\; \langle \Sigma', M'\rangle R_\alpha^{\Omega'} \langle \Delta', N'\rangle,$

or $\langle \Delta, N, T\rangle \Uparrow,$

where $\Omega' = \Omega \cup \bigcup guards_\alpha(p(\Theta))$

We prove this by induction on the size of the typing derivation of $\langle \Sigma, M\rangle$. The details of this proof can be found in appendix A.2.

5 Relating to Other Systems and Idioms

Standard Noninterference As a first example of the expressiveness of our system, consider a standard termination insensitive noninterference property for a lattice-based security model in the standard Denning style [6].

In this setting we have a lattice of security levels $\langle \mathcal{L}, \sqsubseteq, \sqcup \rangle$, and a policy level : $\mathsf{Loc} \to \mathcal{L}$ that fixes the intended security level of the storage locations in the program (and of variables). Given such a policy we can define noninterference. To do this let us first assume that all policies are made up of sets of clauses of the form $\{\} \Rightarrow \alpha$, and that programs do not use lock open/close operations. Furthermore, for simplicity we consider programs of unit type which do not perform any allocation of new references (locations). In what follows let metavariables P and Q range over such programs.

Definition 7 (Noninterference). Given two stores S and T, and a level $k \in \mathcal{L}$, define S and T to be *location indistinguishable at level* k, written $S =_k T$, iff the location domains of S and T are the same, and for all $\ell \in \mathrm{dom}(S)$ such that $\mathrm{level}(\ell) \sqsubseteq k$ we have $S(\ell) = T(\ell)$.

Then we say that variable-free program P is *noninterfering* if for all k, whenever $\langle P, S \rangle \to^* \langle (), S' \rangle$, and $\langle P, T \rangle \to^* \langle (), T' \rangle$, then $S =_k T$ implies $S' =_k T'$.

To represent a lattice policy we do not need any locks; we represent the reading level of a variable by the set of levels at which it may be read. Thus the policy for a storage location ℓ is the upwards closure of its lattice level, written $\uparrow \mathrm{level}(\ell)$, where $\uparrow k = \{\{\} \Rightarrow j \mid j \sqsupseteq k\}$.

In what follows we will *implicitly* identify lattice levels k with the corresponding flow lock policy $\uparrow k$

Given this, we have the following:

Theorem 1. *If P is flow lock secure then P is noninterfering.*

The details of the proof are given in Appendix A.3.

But it is perhaps not too surprising that our security specification is stronger than standard noninterference. A reasonable concern might be that the definition, or indeed the type system, is too strong to be useful. Here we show that despite being stronger, we are still able to type just as much as "typical" systems for regular noninterference.

Figure 6 presents a simple type system for a while language which can be seen as a straightforward reformulation of the typing system presented by Volpano, Irvine and Smith [21].

$$\frac{p = \bigsqcup_{\ell \in E} \mathrm{level}(\ell).}{\vdash_{NI} E : p} \qquad \frac{\vdash_{NI} E : q \qquad p \sqcup q \sqsubseteq \mathrm{level}(\ell)}{p \vdash_{NI} u := E} \qquad \frac{p \vdash_{NI} C_1 \quad p \vdash_{NI} C_2}{p \vdash_{NI} C_1; C_2}$$

$$\frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C_i \quad i = 1, 2}{p \vdash_{NI} \textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2} \qquad \frac{\vdash_{NI} E : q \quad p \sqcup q \vdash_{NI} C}{p \vdash_{NI} \textbf{while } (E)\ C}$$

Figure 6: Standard Noninterference Type System

Define the following translation $\lceil \cdot \rceil$ from terms in the while language to λ_{FL}:

$$\begin{aligned}
\lceil \textbf{while } (E)\ C \rceil &= \textbf{rec } x.\textbf{if } \lceil E \rceil \textbf{ then } \lceil C \rceil; x \textbf{ else } () \\
\lceil \textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \rceil &= \textbf{if } \lceil E \rceil \textbf{ then } \lceil C_1 \rceil \textbf{ else } \lceil C_2 \rceil \\
\lceil C_1; C_2 \rceil &= \lceil C_1 \rceil; \lceil C_2 \rceil \\
\lceil \ell := E \rceil &= \ell_p := \lceil E \rceil \quad \text{where } p = \uparrow \mathrm{level}(\ell) \\
\lceil E \rceil &= E' \quad \text{where } E' \text{ is the result of replacing} \\
&\qquad \text{each location } \ell \text{ in } E \text{ with } \ell_{\uparrow \mathrm{level}(\ell)}.
\end{aligned}$$

To make our formulations easier, let us restrict the language of expressions to booleans (so we do not have to consider typing issues). Now we can state that whenever something is typeable in the simple noninterference system, a corresponding derivation holds for the flow locks system:

Theorem 2. *Let Γ_0 be the type environment that maps every storage location to bool. Then*

1. *If $\vdash_{NI} E : k$ then $\Gamma_0; \emptyset \vdash \lceil E \rceil : bool, (r, \top) \Rightarrow \emptyset$ where $r = \uparrow k$*
2. *If $pc \vdash_{NI} C$ then $\Gamma_0; \emptyset \vdash \lceil C \rceil : unit, (r, w) \Rightarrow \emptyset$ where $w \subseteq \uparrow pc$*

We also expect that a similar theorem holds for some suitable termination-insensitive version of DCC [1], although we have not attempted to show this formally.

Simple Declassification We can encode a simple declassification mechanism in the same Denning-style setting as used in the previous example. The needed extra step is to extend all policies with clauses to allow declassification. For each level j not in the policy already, we introduce a flow lock σ_j representing a declassification to that level. The new policies then look like

$$\{k \mid k \sqsupseteq \mathrm{level}(\ell)\} \cup \{\sigma_j \Rightarrow k \mid j \not\sqsupseteq \mathrm{level}(\ell), k \sqsupseteq j\}$$

We can now define a declassification operator to level j as

$$\mathit{declassify}_j \equiv (\lambda v.\mathbf{let}\ x = (\mathbf{open}\ \sigma_j; v)\ \mathbf{in}\ (\mathbf{close}\ \sigma_j; x))$$

It is easy to verify from the type system that the only effect of applying this function to some value is that the value will then be readable also at level j, as was our intention.

Lexically Scoped Flows In the setting of a multilevel security model, Almeida Matos and Boudol describe how to build a system with lexically scoped dynamic flow policies [2]. They start from a λ-calculus with recursion and references like we do, and introduce a construct *"flow $\alpha \prec \beta$ in M"* that extends the current global flow policy to also allow flows from level α to β in the scope of M. These flows are transitive, so if the current policy already allows flows from say β to γ, flows from α to γ would also be allowed in M.

Modelling scoped flows using flow locks is easy, but the global nature of policies in Almeida Matos and Boudol's system, as opposed to our local policies on memory locations, needs special treatment. We introduce a lock $\sigma_{\alpha \prec \beta}$ for each pair of levels α and β that data could flow between. Each policy on some data must record the fact that a future flow declaration could allow that data to flow to many new locations due to the transitive nature of flows. Thus if a location in Almeida Matos and Boudol's system would have level A, we could represent that as

$$A \cup \left\{ \sigma_{\alpha \prec \beta_0}, \sigma_{\beta_0 \prec \beta_1}, \ldots, \sigma_{\beta_{k-1} \prec \beta_k} \Rightarrow \beta_k \mid \alpha \in A, \beta_i \notin A \right\}$$

where the $\notin$ is taken with respect to some universal set of levels. In effect, each location records all possible future transitive flows from it. We then derive our representation of the "flow" construct that opens a lock in the scope of some subprogram:

$$\mathbf{flow}\ \sigma\ \mathbf{in}\ M \equiv \mathbf{let}\ x = (\mathbf{open}\ \sigma; M)\ \mathbf{in}\ (\mathbf{close}\ \sigma; x)$$

Almeida Matos and Boudol also include parallel execution in their system, and as a consequence make their type system and semantic security definition, called *non-disclosure*, sensitive to possible non-termination. Our system has no parallel execution so we cannot model their full system, only the sequential subset.

Intransitive Noninterference Flow locks represent a lower level abstraction than lattice-based information flow models in the sense that the lattice ordering is not "built in" but must be represented explicitly. One advantage

of such a lower level view is that it can also represent *intransitive noninterference* policies [15, 14] — i.e. ones in which the flow relation is intentionally not transitive. Since intransitive policies are the default case for flow locks, it is straightforward to represent simple language-based intransitive policies such as the one described by Mantel and Sands [8].

Noninterference Until Declassification Chong and Myers' [5] introduce a class of temporal declassification policies. This is achieved by annotating variables with types of the form $k_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_n}{\rightsquigarrow} \underline{k_n}$, which intuitively means that a variable with such an annotation may be successively declassified to the levels $k_1, \ldots, k_n$, and that the conditions $c_1, \ldots, c_n$ will hold at the execution of the corresponding declassification points. The exact nature of the conditions are left unspecified, and it is assumed in the type system that these conditions are verified at certain key program points by some external tool.

We can achieve a similar effect fairly naturally using flow locks, where we would use a distinct lock C_i for each condition c_i. One should then insert **open** C_i constructs in the program at points where the intended declassification takes place, and verify (with an external tool) that the corresponding condition c_i does indeed hold at these points, and that lock C_{i-1} has been opened (we assume that locks are never closed in this encoding). The policy above could then be represented as

$$\{k_0; \{C_1\} \Rightarrow k_1; \cdots; \{C_1, \ldots, C_n\} \Rightarrow k_n\}.$$

Robust Declassification Information flow may be used to verify integrity properties, to ensure that untrusted (low integrity) data does not influence the values of trusted (high integrity) data. Since flow lock policies are neutral with respect to whether we are dealing with confidentiality or integrity properties it is no problem to add such integrity policies to data, and we can easily have clauses for integrity and confidentiality in the same policy. The interesting case, however, is the interaction between confidentiality and integrity in the presence of dynamic policies.

Zdancewic and Myers [22] introduced the concept of *robust declassification* to characterise the property that an attacker (who controls low integrity data) cannot influence what is declassified. This guarantees that the attacker cannot manipulate the amount of information which is released through declassification.

In the setting of flow lock policies, "declassification" can be thought of as the process of opening locks, since whenever a lock is opened more flows

are enabled. Thus we can interpret robust declassification as the question of whether low integrity data can influence the decision to open locks. [5]

One possible way of enforcing robust declassification using flow locks is to observe the following: since we cannot perform any computation with locks, the only way that an open operation can be influenced by low integrity data is via indirect information flow from low integrity data. Suppose that our policies use an indexed set of locks $\sigma_i, i \in I$ to control confidentiality. These are unguarded (i.e. we ignore *endorsement*). Let us assume that in addition to the actors of the system we have the pseudo-actor *trusted* used to track integrity information, just as we did in Section 2.

In order to prevent indirect flow from low integrity data to the opening of locks, we will log each use of an open operation by writing to a variable *log*. An obvious way to enforce this is to define a "robust" version of open:

$$\textbf{ropen}\ \sigma_i \equiv \textbf{open}\ \sigma_i; \mathit{log} := i$$

Now we give *log* the policy $\{\mathit{trusted}\}$. This ensures that the assignment is always safe from a confidentiality perspective (since normal actors can never read it anyway), and that the open operation can never have taken place in a low integrity context (since otherwise the assignment would cause information to flow from untrusted to trusted data). Finally, to additionally prevent the declassification of low integrity data we can syntactically enforce that lock-guarded policies are only used on high integrity data.

The Decentralized Label Model In the Decentralized Label Model (DLM) [10, 11, 12], data is said to be *owned* by a set of principals. These principals may allow other principals to read the data, and the effective reader set is those principals that all owners agree may read the data. Allowing a new reader roughly corresponds to declassification, and we can model it similarly. The DLM also defines a global principal hierarchy, where one principal may allow another principal to *act for* it, which means it may read all the same things. This is very similar in spirit to introducing a new flow in the system by Almeida Matos and Boudol, including transitivity, and we can model it in the same way. Apart from clauses for declassification and hierarchic flows, the policies must also include clauses for the combination of the two, e.g. A can read the data if B owns it, has declassified it for C to read it, and A acts for C.

A common extension of the DLM [22, 20, 19] deals with integrity and trust. The interesting part for us is the integration with the principal hierarchy, where if A trusts some data and A acts for B, then B also trusts that

[5] If we also take the view from [13], then we extend this concept with the requirement that we should not be able to declassify low integrity data

data. This can be modelled as the reverse of the normal clauses for transitive flows, and the clauses will be very similar to those for forward flows.

The complete general policy for a DLM variable encoded with flow locks would be fairly large and awkward, so we do not show it here.

Other Related Work The JFlow language [9], as well as several recent papers [19, 23, 7], supports runtime mechanisms to enforce security in situations where this cannot be determined statically, e.g. permissions on a file that cannot be known at compile time. Our flow locks is a static, compile-time mechanism only, and thus cannot handle these issues.

Banerjee and Naumann [4] describe a combination of stack-based access control and information flow types to allow the static checking of policies such as "the method returns a result at level L unless the caller has permission p". It may be possible to encode these kinds of policies in a straightforward way using flow locks, but this remains a topic for future work.

6 Conclusions and Future Work

Flow locks are a very simple mechanism that generalises many existing systems and idioms for dynamic information flow policies. We have only just started looking at flow locks however, and much remains to be done.

To really establish flow locks as a core calculus, we need to show more formally how to embed other systems and idioms, and prove that our semantic condition is sufficiently strong compared to the semantic conditions of these other systems. It would also be worthwhile to look at extensions of our core system, in order to handle systems that we definitely cannot model at this point. Examples of such systems include the parallel execution of Almeida Matos and Boudol [2], and also systems that use various runtime mechanisms [19, 23, 7].

Furthermore, we would need to investigate how to implement the flow locks system as a programming language, and to determine what kinds of inference would be needed for policies and locks. Also, flow locks are fairly low-level in nature, being a raw mechanism for controlling data flows in a program. As such it is nontrivial to write and maintain correct flow lock programs. It would therefore be useful to look at what higher-level abstractions and design patterns that could be used together with flow locks. There exists some work specifically targeting the question of patterns, for instance the *seal* pattern by Askarov and Sabelfeld [3].

Acknowledgements Thanks to Ulf Norell and our colleagues in the ProSec group for helpful comments, and to the anonymous referees (ESOP'06) for numerous helpful comments and suggestions. This work was partly supported by the Swedish research agencies SSF, VR and Vinnova, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, Jan. 1999.

[2] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, June 2005.

[3] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, 2005.

[4] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[5] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, Oct. 2004.

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

[7] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proc. Foundations of Computer Security Workshop*, 2005.

[8] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, Nov. 2004.

[9] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[10] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, Oct. 1997.

[11] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

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

[13] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004.

[14] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995.

[15] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992.

[16] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[17] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Security Foundations Workshop*, pages 200–214, July 2000.

[18] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, 2005.

[19] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. Symposium on Security and Privacy*, 2004.

[20] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, Apr. 2005.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[22] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

[23] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Proc. Workshop on Formal Aspects in Security and Trust*, 2004.

A Appendix

A.1 Proofs that the type system guarantees semantic soundness

Lemma 1 (Progress). If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ then either

- $M \in \text{Val}$, or
- for all S such that $\text{dom}(S) \supseteq \text{fv}(M) \cup \text{loc}(M)$ and $\vdash S$ then $\exists \Sigma', M', S'. \langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', M', S' \rangle$.

Proof. In $CORE_{FL}$ the syntax restricts the terms of the language to be in a reductive form, except the *bind* construct. This is thus the only case for which the lemma does not trivially hold. By induction on the size of the typing derivation for M we get for the *bind* case from the induction hypothesis that it holds for the bound expression, and thus it holds for M. □

Lemma 2 (Preservation). If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ and $\vdash S$ and $\text{dom}(S) \supseteq \text{fv}(M) \cup \text{loc}(M)$ and $\langle \Sigma, M, S \rangle \rightarrow \langle \Sigma', M', S' \rangle$, then $\vdash S'$ and $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ where $r' \preceq r$ and $w \preceq w'$.

Proof. We prove this by induction on the typing derivation for M, and by cases according to the structure of M.

Case: $M \in \text{Val}$. The statement is vacuously true.

Case: $M = x_{p,\tau}$. The reduction has the form $\langle \Sigma, x_{p,\tau}, S \rangle \rightarrow \langle \Sigma, S(x_{p,\tau}), S \rangle$, and the typing derivation is of the form $\Sigma \vdash x_{p,\tau} : \tau, (p(\Sigma), \top) \Rightarrow \Sigma$. Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,\tau}) : \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq p(\Sigma)$ and $\top \preceq \top$ as required.

Case: $M = \mathbf{ref}_{p'}\ x_{p,\tau}$. The reduction has the form

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S \rangle \rightarrow \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] \rangle$$

and the typing derivation is of the form

$$\frac{p \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : \mathbf{ref}_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma}.$$

Since $\vdash S$ we have that $\Sigma \vdash S(x_{p,\tau}) : \tau, (\bot, \top) \Rightarrow \Sigma$, so we have $\vdash S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]$ and $\Sigma \vdash \ell_{p',\tau} : \mathbf{ref}_{p'}\ \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq \bot$ and $p' \preceq \top$ as required.

Case: $M = !x_{p,\mathbf{ref}_{p'}\ \tau}.$ The reduction has the form

$$\langle \Sigma, !x_{p,\mathbf{ref}_{p'}\ \tau}, S \rangle \rightarrow \langle \Sigma, S(S(x_{p,\tau})), S \rangle$$

and the typing derivation is of the form

$$\Sigma \vdash !x_{p,\mathbf{ref}_{p'}\ \tau} : \tau, (p(\Sigma) \sqcup p'(\Sigma), \top) \Rightarrow \Sigma$$

. Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,\tau}) : \mathbf{ref}_{p'}\ \tau, (\bot, \top) \Rightarrow \Sigma$, and thus that $\Sigma \vdash S(S(x_{p,\tau})) : \tau, (\bot, \top) \Rightarrow \Sigma$. We have $\bot \preceq p(\Sigma) \sqcup p'(\Sigma)$ and $\top \preceq \top$ as required.

Case: $M = x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'}.$ The reduction has the form

$$\langle \Sigma, x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'}, S \rangle \rightarrow \langle \Sigma, (), S[S(x_{p,\mathbf{ref}_{p''}\ \tau} \mapsto S(y_{p',\tau'})] \rangle$$

and the typing derivation has the form

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,\mathbf{ref}_{p''}\ \tau} := y_{p',\tau'} : unit, (\bot, p'') \Rightarrow \Sigma}.$$

Since $\vdash S$ this means that $\Sigma \vdash S(x_{p,ref_{p''}\ \tau}) : \mathbf{ref}_{p''}\ \tau, (\bot, \top) \Rightarrow \Sigma$, and that $\Sigma \vdash S(y_{p',\tau'}) : \tau, (\bot, \top) \Rightarrow \Sigma$, and thus we have that $\vdash S[S(x_{p,\mathbf{ref}_{p''}\ \tau} \mapsto S(y_{p',\tau'})]$. We have $\Sigma \vdash () : unit, (\bot, \top) \Rightarrow \Sigma$, and $\bot \preceq \bot$ and $p'' \preceq \top$ as required.

Case: $M = \mathbf{if}\ x_{p,bool}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1.$ The reduction has the form

$$\langle \Sigma, \mathbf{if}\ x_{p,bool}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1, S \rangle \rightarrow \langle \Sigma, M_i, S \rangle$$

and the typing derivation is of the form

$$\frac{\Sigma \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Delta \quad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \mathbf{if}\ x_{p,\tau}\ \mathbf{then}\ M_0\ \mathbf{else}\ M_1 : \tau, (p(\Sigma) \sqcup r_0 \sqcup r_1, w_0 \sqcap w_1) \Rightarrow \Delta}.$$

We have $r_i \preceq p(\Sigma) \sqcup r_0 \sqcup r_1$ and $w_0 \sqcap w_1 \preceq w_i$ as required.

Case: $M = x_{p,\tau_f}\ y_{p',\tau'}$ **where** $\tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$. The reduction has the form

$$\langle \Sigma, x_{p,\tau_f}\ y_{p',\tau'}, S\rangle \to \langle \Sigma, M, S[z_{p',\tau'} \mapsto S(y_{p',\tau'})]\rangle,$$

where $S(x_{p,\tau_f}) = \lambda z_{p',\tau'}.M$. The typing derivation is of the form

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}.$$

Since $\vdash S$ this means that

$$\frac{\Sigma \vdash M : \tau, (r_f, w_f) \Rightarrow \Sigma'}{\Sigma \vdash \lambda z_{p',\tau'}.M : (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau, (\bot, \top) \Rightarrow \Sigma}$$

and that $\Sigma \vdash S(y_{p',\tau'}) : \tau', (\bot, \top) \Rightarrow \Sigma$, so we can show that $\vdash S[z_{p',\tau'} \mapsto S(y_{p',\tau'})]$. We have $r_f \preceq p(\Sigma) \sqcup r_f$ and $w_f \preceq w_f$ as required.

Case: $M =$ **open** σ. The reduction has the form

$$\langle \Sigma, \textbf{open}\ \sigma, S\rangle \to \langle \Sigma \cup \{\sigma\}, (), S\rangle$$

and by typing we know $\Sigma \vdash$ **open** $\sigma : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}$, and we can show $\Sigma \cup \{\sigma\} \vdash () : \mathit{unit}, (\bot, \top) \Rightarrow \Sigma \cup \{\sigma\}$. We have $\bot \preceq \bot$ and $\top \preceq \top$ as required.

Case: $M =$ **close** σ. Similar to the previous case.

Case: $M =$ **bind** $x_{p,\tau} = M'$ **in** N. Here we have two cases: either $M' \in$ Val, or we can do a reduction in M'.

Subcase: $M' = v$. The reduction and type derivations respectively have the form

$$\langle \Sigma, \textbf{bind}\ x_{p,\tau} = v\ \textbf{in}\ N, S\rangle \to \langle \Sigma, N, S[x_{p,\tau} \mapsto v]\rangle$$

$$\frac{\Sigma \vdash v : \tau, (\bot, \top) \Rightarrow \Sigma \quad \Sigma \vdash N : \tau', (r, w) \Rightarrow \Delta}{\Sigma \vdash \textbf{bind}\ x_{p,\tau} = v\ \textbf{in}\ N : \tau', (r, w) \Rightarrow \Delta}.$$

We thus have $\vdash S[x_{p,\tau} \mapsto v]$, and $r \preceq r$ and $w \preceq w$ as required.

Subcase: $M' \notin$ Val. The type derivation for M must have the form

$$\frac{\Sigma \vdash M' : \tau, (r', w') \Rightarrow \Sigma'' \quad \Sigma'' \vdash N : \tau', (r_N, w_N) \Rightarrow \Delta \quad r'(\Sigma'') \preceq p}{\Sigma \vdash \textbf{bind}\ x_{p,\tau} = M'\ \textbf{in}\ N : \tau', (r_N, w' \sqcap w_N) \Rightarrow \Delta}.$$

and by progress that we can reduce M'. We can thus perform the reduction $\langle \Sigma, M', S\rangle \rightarrow \langle \Sigma', M'', S'\rangle$, and by the induction hypothesis we know $\Sigma' \vdash M'' : \tau, (r'', w'') \Rightarrow \Sigma''$ where $r'' \preceq r'$, $w' \preceq w''$ and $\vdash S'$. We can thus show that

$$\frac{\Sigma' \vdash M'' : \tau, (r'', w'') \Rightarrow \Sigma'' \quad \Sigma'' \vdash N : \tau', (r_N, w_N) \Rightarrow \Delta \quad r''(\Sigma'') \preceq p}{\Sigma \vdash \mathbf{bind}\ x_{p,\tau} = M'' \ \mathbf{in}\ N : \tau', (r_N, w'' \sqcap w_N) \Rightarrow \Delta},$$

and we have $r_N \preceq r_N$ and $w' \sqcap w_N \preceq w'' \sqcap w_N$ as required.

□

A.2 Proof that Well-typed Programs are Flow-Lock Secure

In this section we prove our claim that all programs typeable with our type system are indeed secure.

The basic approach is to utilise the coinductive nature of the bisimulation definition. We show that for well-typed closed M, $\langle \emptyset, M\rangle \sim_\alpha \langle \emptyset, M\rangle$ by construction of a candidate relation R^Ω_α, that in particular contains the pair $(\langle \emptyset, M\rangle, \langle \emptyset, M\rangle)$, and which can be shown to be an α-bisimulation. This gives us that $(\langle \emptyset, M\rangle, \langle \emptyset, M\rangle) \in R^\emptyset_\alpha \subseteq \sim_\alpha$.

A.2.1 The candidate relation R^Ω_α

To be able to define the candidate relation R^Ω_α we need the notion of programs that are *high* with respect to some actor α. (A similar concept is introduced by Almeida Matos and Boudol [2]). We say that a program is α-Ω-high if it does not modify any locations that α could see while all the locks in Ω remain closed. However, this operational notion of being high is a bit awkward to work with, so instead we use a stronger, syntactic notion stating that a program is *syntactically* α-Ω-high if it does not *write* to any locations that α could see while the locks in Ω remain closed.

Definition 5 (Syntactically α-Ω-high programs: H^Ω_α). Let H^Ω_α be the set of all terms M such that $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\alpha \not\preceq^\Omega w$.

Now we can define our candidate relation as follows:

Definition 6 (Candidate relation R^Ω_α). Let R^Ω_α be a symmetric relation

on well-typed preconfigurations, inductively defined as follows:

$$1\frac{}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, M\rangle} \qquad 2\frac{M, N \in H_\alpha^\Omega}{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle}$$

$$3\frac{\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Sigma, N\rangle \quad \alpha \not\in^\Omega p}{\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = M \textbf{ in } M']\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N \textbf{ in } M']\rangle}$$

where $\mathbb{E}[\cdot]$ are the evaluation contexts for CORE_{FL}, given by

$$\mathbb{E}[\cdot] ::= [\cdot] \mid \textbf{bind } \boldsymbol{x} = \mathbb{E}[\cdot] \textbf{ in } M$$

We can identify a useful property of this set, namely that if two preconfigurations are related and one of the programs is high, then so is the other.

Lemma 5 (R_α^Ω relates high terms to other high terms). *If $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ and $M \in H_\alpha^\Omega$, then $N \in H_\alpha^\Omega$.*

Proof. By induction on the size of the typing derivation of M.

If $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ by rule 1, then $M = N$ and we have $N \in H_\alpha^\Omega$.

If $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ by rule 2, then $N \in H_\alpha^\Omega$ by construction.

If $\langle \Sigma, M\rangle R_\alpha^\Omega \langle \Delta, N\rangle$ by rule 3, then we have

$$M = \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1] \text{ and } N = \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1],$$

where $\langle \Sigma, M_0\rangle R_\alpha^\Omega \langle \Delta, N_0\rangle$.

By typing we have $\dfrac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau', (r_1, w_1) \Rightarrow \Sigma''}{\Sigma \vdash \textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1 : \tau', (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$.

Since $M \in H_\alpha^\Omega$ we have $\alpha \not\in^\Omega w_0 \sqcap w_1$, which means that $M_0, M_1 \in H_\alpha^\Omega$. We apply the induction hypothesis on M_0 to get that $N_0 \in H_\alpha^\Omega$, and thus that $\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1 \in H_\alpha^\Omega$. Continuing the same argument for all binds in $\mathbb{E}[]$, we get that $N \in H_\alpha^\Omega$ as required. □

A.2.2 Proof that R_α^Ω is a bisimulation

Now that we have our candidate relation, the final step is to prove that it is indeed a bisimulation. In order to do this, we first need to state a number of helper lemmas.

We begin by proving that syntactically high programs are also operationally high, i.e. that they never produce any α-observable changes to the store. We do this in three separate steps. First we prove that syntactically

high terms reduce to syntactically high terms. Second, we prove that reducing a syntactically high term will not result in any α-observable changes to the store. Finally we put these two together to form a notion of uninterrupted high computation.

Lemma 6 (H^Ω_α is closed under reduction). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *and* $\alpha \not\downarrow^\Omega w$ *and* $\vdash S$ *and* $\langle \Sigma, M, S\rangle \to \langle \Sigma', M', S'\rangle$ *then* $\vdash S'$ *and* $\Sigma' \vdash M' : \tau, (r', w') \Rightarrow \Delta$ *and* $\alpha \not\downarrow^\Omega w'$.

Proof. Preservation gives us $w \preceq w'$, so if $\alpha \not\downarrow^\Omega w$ then $\alpha \not\downarrow^\Omega w'$. □

Lemma 7 (H^Ω_α is α-Ω-high).
If $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Delta$ *and* $\alpha \not\downarrow^\Omega w$ *and* $\vdash S$ *and* $\langle \Sigma, M, S\rangle \to \langle \Sigma', M', S'\rangle$ *then* $\forall \Theta. S =_\alpha^{\Theta \setminus \Omega} S'$.

Proof. By induction on the size of the typing derivation. For terms that do not update or create a location in the store when reduced, the above is trivially true. The remaining cases are reference creation, assignment and the recursive case of bind:

Case: $M = \mathbf{ref}_{p'}\ x_{p,\tau}$. In this case the reduction and typing derivation are of the following form:

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S\rangle \to \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]\rangle$$

$$\frac{p \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : ref_{\,p'}\ \tau, (\bot, p') \Rightarrow \Delta}$$

and we know $\alpha \not\downarrow^\Omega p'$. Thus the newly created location is secret to α, and we have $\forall \Theta. S =_\alpha^{\Theta \setminus \Omega} S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]$.

Case: $M = x_{p,ref_{\,p''}\,\tau} := y_{p',\tau}$. The reduction and typing derivation have the form

$$\langle \Sigma, x_{p,ref_{\,p''}\,\tau} := y_{p',\tau}, S\rangle \to \langle \Sigma, (), S[S(x_{p,ref_{\,p''}\,\tau}) \mapsto S(y_{p',\tau})]\rangle$$

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,ref_{\,p''}\,\tau} := y_{p',\tau} : unit, (\bot, p'') \Rightarrow \Sigma}$$

and we know $\alpha \not\downarrow^\Omega p''$. Since $\vdash S$ we know that

$$\Sigma \vdash S(x_{p,ref_{\,p''}\,\tau}) : ref_{\,p''}\ \tau, (\bot, \top) \Rightarrow \Sigma$$

and thus $S =_\alpha^{\Theta \setminus \Omega} S[S(x_{p,ref_{\,p''}\,\tau}) \mapsto S(y_{p',\tau})]$.

Case: $M = \textbf{bind}\ x_{p,\tau} = M_0\ \textbf{in}\ N.$ If $M_0 \in \text{Val}$ the reduction step will not change the value of any memory location so the conclusion trivially holds. Otherwise we can apply the induction hypothesis on M_0 to get that if $\langle \Sigma, M_0, S\rangle \rightarrow \langle \Sigma', M_0', S'\rangle$ then $\forall \Theta. S =_\alpha^{\Theta\backslash\Omega} S'$. From this we can conclude that if $\langle \Sigma, \textbf{bind}\ x_{p,\tau} = M_0\ \textbf{in}\ N, S\rangle \rightarrow \langle \Sigma', \textbf{bind}\ x_{p,\tau} = M_0'\ \textbf{in}\ N, S'\rangle$ then $\forall \Theta. S =_\alpha^{\Theta\backslash\Omega} S'$.

□

Lemma 8 (Uninterrupted high evaluation). *If* $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ *and* $\alpha \not\lhd^\Omega w$ *and* $\vdash S$ *and* $\langle \Sigma, M, S\rangle \rightarrow^* \langle \Sigma', v, S'\rangle$ *then* $\forall \Theta. S =_\alpha^{\Theta\backslash\Omega} S'$.

Proof. We prove this by induction on the length of the derivation. We have two cases: Either (1) M is a value, or (2) we can reduce M.

Case: 1. If M is a value, then by typing we have $\Sigma' = \Sigma$, so we can take 0 steps to get $S = S'$ and the conclusion holds.

Case: 2. M is not a value, so by progress we can reduce it further. By preservation and α-Ω-high, since $\Sigma \vdash M : \tau, (r, w) \Rightarrow \Sigma'$ and $\vdash S$ and $\langle \Sigma, M, S\rangle \rightarrow \langle \Sigma'', M', S''\rangle$ then $\forall \Theta. S =_\alpha^{\Theta\backslash\Omega} S''$ and $M' \in H_\alpha^\Omega$. We can then do $\langle \Sigma'', M', S''\rangle \rightarrow^* \langle \Sigma', v, S'\rangle$ and the induction hypothesis gives us $\forall \Theta. S'' =_\alpha^{\Theta\backslash\Omega} S'$. By transitivity we can conclude that $\forall \Theta. S =_\alpha^{\Theta\backslash\Omega} S'$. □

Apart from these lemmas pertaining to high programs, we need to prove the lemma from section 4 that connects the visibility of a policy to its guards. We first give a helper lemma that gives an alternative interpretation of visibility:

Lemma 9. $\alpha \lhd p(\Theta) \Leftrightarrow \exists (\Phi \Rightarrow \alpha) \in p.\Phi \subseteq \Theta$

Proof. If $\alpha \lhd p(\Theta)$ then by definition we have $\{\} \Rightarrow \alpha \in p(\Theta)$, which in turn means that we have $\{\} \Rightarrow \alpha \in \{\Phi \setminus \Theta \Rightarrow \beta \mid \Phi \Rightarrow \beta \in p\}$. For this to be true we must have $\exists \Phi \Rightarrow \alpha. \Phi \setminus \Theta = \{\}$, which means $\Phi \subseteq \Theta$. □

Now we can prove the guard lemma:

Lemma 3 (Guard lemma). If $\alpha \not\lhd p$, then $\alpha \not\lhd^\Omega p$ where $\Omega = \bigcup guards_\alpha(p)$.

Proof. By contradiction. Assume that $\exists \Theta. \alpha \lhd p(\Theta \setminus \Omega)$. This means that $\exists \Phi \Rightarrow \alpha \in p.\Phi \neq \{\}$ and $\Phi \subseteq \Theta \setminus \Omega$. But if $\Phi \Rightarrow \alpha \in p$ then $\Phi \subseteq \Omega$, so we have a contradiction. □

With these lemmas in hand, we can finally move on to prove the main lemma, that our candidate relation is a bisimulation.

Lemma 4 ($\bigcup_{\Omega} R^{\Omega}_{\alpha}$ is a bisimulation). If $\langle \Sigma, M\rangle R^{\Omega}_{\alpha}\langle \Delta, N\rangle$ and $\vdash S$ and

$$\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle \;\&\; \Theta \supseteq \Sigma \;\&\; S =^{\Theta}_{\alpha} T \;\&\; \vdash T$$

then $\vdash S'$, and there exists Δ', N', T' such that

$$\begin{aligned}&\text{either } \langle \Delta, N, T\rangle \to^{*} \langle \Delta', N', T'\rangle \;\&\; \vdash T' \\ &\qquad\qquad \&\; S' =^{\Theta\setminus\Omega}_{\alpha} T' \;\&\; \langle \Sigma', M'\rangle R^{\Omega'}_{\alpha}\langle \Delta', N'\rangle,\end{aligned}$$

$$\text{or } \langle \Delta, N, T\rangle \Uparrow,$$

where $\Omega' = \Omega \cup \bigcup guards_{\alpha}(p(\Theta))$

Proof. We will conduct the proof by induction on the size of the typing derivation of $\langle \Sigma, M\rangle$.

By preservation, we already know that if we reduce a well-typed term in the presence of a well-typed store, the resulting term and store are going to be well-typed as well, so we will not bother about controlling either of those facts in the rest of this proof.

In many cases we will implicitly make use of properties of the relations $=^{\Theta}_{\alpha}$ and $\preceq$, such as transitivity and monotonicity, which we discussed in the main body of the paper.

Case: $\langle \Sigma, M\rangle R^{\Omega}_{\alpha}\langle \Delta, N\rangle$ by rule 1. We have that $M = N$. Without loss of generality we will assume that $M, N \notin H^{\Omega}_{\alpha}$, since we will cover that when considering rule 2. This means that M cannot be a variable, a value, a dereferencing, a recursion, an open or a close. Remains a reference creation, an assignment, a conditional, an application or a bind.

Subcase: $M \equiv \mathbf{ref}_{p'}\ x_{p,\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbf{ref}_{p'}\ x_{p,\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, \ell_{p',\tau}, S[\ell_{p',\tau} \mapsto S(x_{p,\tau})]\rangle \quad \text{and}$$
$$\langle \Delta, \mathbf{ref}_{p'}\ x_{p,\tau}, T\rangle \xrightarrow{\top} \langle \Delta, \ell_{p',\tau}, T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]\rangle$$

By typing we have

$$\frac{p(\Sigma) \preceq p'}{\Sigma \vdash \mathbf{ref}_{p'}\ x_{p,\tau} : ref_{p'}\ \tau, (\bot, p') \Rightarrow \Sigma}$$

Suppose that $\alpha \lhd p(\Theta)$. Then $S(x_{p,\tau}) = T(x_{p,\tau})$ and we have $S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] =^{\Theta\setminus\Omega}_{\alpha} T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]$. If on the other hand we have that $\alpha \not\lhd p(\Theta)$, then $S(x_{p,\tau}) = T(x_{p,\tau})$ is not guaranteed, so to assure that

$S[\ell_{p',\tau} \mapsto S(x_{p,\tau})] =_\alpha^{\Theta\setminus\Omega} T[\ell_{p',\tau} \mapsto T(x_{p,\tau})]$ we require that $\alpha \not\triangleleft p'(\Theta \setminus \Omega)$. This follows from $p(\Sigma) \preceq p'$. We can finally conclude $\langle \Sigma, \ell_{p',\tau}\rangle R_\alpha^\Omega \langle \Delta, \ell_{p',\tau}\rangle$ by rule 1.

Subcase: $M \equiv x_{p,ref_{p''}\,\tau} := y_{p',\tau}.$ For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, x_{p,ref_{p''}\,\tau} := y_{p',\tau}, S\rangle \xrightarrow{\top} \langle \Sigma, (), S[S(x_{p,ref_{p''}\,\tau}) \mapsto S(y_{p',\tau})]\rangle \quad \text{and}$$
$$\langle \Delta, x_{p,ref_{p''}\,\tau} := y_{p',\tau}, T\rangle \xrightarrow{\top} \langle \Delta, (), T[T(x_{p,ref_{p''}\,\tau}) \mapsto T(y_{p',\tau})]\rangle$$

By typing we have

$$\frac{p(\Sigma) \sqcup p'(\Sigma) \preceq p''}{\Sigma \vdash x_{p,ref_{p''}\,\tau} := y_{p',\tau} : unit, (\bot, p'') \Rightarrow \Sigma}$$

Now we reason by cases according to the following three exhaustive conditions: (i) $\alpha \lhd p(\Theta)$ and $\alpha \lhd p'(\Theta)$, (ii) $\alpha \not\triangleleft p(\Theta)$, and (iii) $\alpha \not\triangleleft p'(\Theta)$.
In case (i) we have that $S(x_{p,ref_{p''}\,\tau}) = T(x_{p,ref_{p''}\,\tau})$ and $S(y_{p',\tau}) = T(y_{p',\tau})$ and thus we have

$$S[S(x_{p,ref_{p''}\,\tau}) \mapsto S(y_{p',\tau})] =_\alpha^{\Theta\setminus\Omega} T[T(x_{p,ref_{p''}\,\tau}) \mapsto T(y_{p',\tau})]$$

as required.
In case (ii) then $S(x_{p,ref_{p''}\,\tau}) = T(x_{p,ref_{p''}\,\tau})$ is not guaranteed, so to assure that

$$S[S(x_{p,ref_{p''}\,\tau}) \mapsto S(y_{p',\tau})] =_\alpha^{\Theta\setminus\Omega} T[T(x_{p,ref_{p''}\,\tau}) \mapsto T(y_{p',\tau})]$$

we require that $\alpha \not\triangleleft p''(\Theta \setminus \Omega)$. This follows from $p(\Sigma) \preceq p''$.
In case (iii) then $S(y_{p',\tau}) = T(y_{p',\tau})$ is not guaranteed, so to assure that

$$S[S(x_{p,ref_{p''}\,\tau}) \mapsto S(y_{p',\tau})] =_\alpha^{\Theta\setminus\Omega} T[T(x_{p,ref_{p''}\,\tau}) \mapsto T(y_{p',\tau})]$$

we again require that $\alpha \not\triangleleft p''(\Theta \setminus \Omega)$. This follows from $p'(\Sigma) \preceq p''$.

We can finally conclude $\langle \Sigma, ()\rangle R_\alpha^\Omega \langle \Delta, ()\rangle$ by rule 1.

Subcase: $M \equiv \textbf{if } x_{p,bool} \textbf{ then } M_0 \textbf{ else } M_1.$ For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \textbf{if } x_{p,bool} \textbf{ then } M_0 \textbf{ else } M_1, S\rangle \xrightarrow{p} \langle \Sigma, M_i, S\rangle \quad \text{and}$$
$$\langle \Delta, \textbf{if } x_{p,bool} \textbf{ then } M_0 \textbf{ else } M_1, T\rangle \xrightarrow{p} \langle \Delta, M_j, T\rangle$$

where $i, j \in \{0, 1\}$. By typing we have

$$\frac{\Sigma \vdash M_i : \tau, (r_i, w_i) \Rightarrow \Sigma' \quad p(\Sigma) \preceq w_0 \sqcap w_1}{\Sigma \vdash \textbf{if } x_{p,bool} \textbf{ then } M_0 \textbf{ else } M_1 : \tau, (p(\Sigma) \sqcup r_0 \sqcup r_1, w_0 \sqcap w_1) \Rightarrow \Sigma'}$$

Assume that $\alpha \lhd p(\Theta)$. Then $S(x_{p,bool}) = T(x_{p,bool})$ which means that $i = j$ and $\bigcup guards_\alpha(p(\Theta)) = \{\}$. We have $M_i = M_j$ and we can conclude $\langle \Sigma, M_i \rangle R^\Omega_\alpha \langle \Delta, M_j \rangle$ by rule 1.

Assume now instead that $\alpha \not\lhd p(\Theta)$. Then $S(x_{p,bool}) = T(x_{p,bool})$ is not guaranteed, and thus possibly $M_i \neq M_j$. But since $\alpha \not\lhd p(\Theta)$, by the guard lemma we have that $\alpha \not\lhd^{\Omega'} p(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p(\Theta))$, and further since $p(\Sigma) \preceq w_i$ and $\Theta \supseteq \Sigma$ we have that $\alpha \not\lhd^{\Omega'} (w_i)$. This means that $M_i, M_j \in H^{\Omega \cup \Omega'}_\alpha$ and we can conclude $\langle \Sigma, M_i \rangle R^{\Omega \cup \Omega'}_\alpha \langle \Delta, M_j \rangle$ by rule 2.

Subcase: $M \equiv x_{p,\tau_f}\ y_{p',\tau'}$ **where** $\tau_f = (\tau', p') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$. For $\Theta \supseteq \Sigma$ we have $S =^\Theta_\alpha T$ and

$$\langle \Sigma, x_{p,\tau_f}\ y_{p',\tau'}, S \rangle \xrightarrow{p} \langle \Sigma, M_0, S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] \rangle, \quad \text{and}$$
$$\langle \Delta, x_{p,\tau_f}\ y_{p',\tau'}, T \rangle \xrightarrow{p} \langle \Delta, M_1, T[w_{p',\tau'} \mapsto S(y_{p',\tau'})] \rangle$$

where $S(x_{p,\tau_f}) = \lambda z_{p',\tau'}.M_0$ and $T(x_{p,\tau_f}) = \lambda w_{p',\tau'}.M_1$. By typing we have

$$\frac{p(\Sigma) \preceq w_f}{\Sigma \vdash x_{p,\tau_f}\ y_{p',\tau'} : \tau, (p(\Sigma) \sqcup r_f, w_f) \Rightarrow \Sigma'}$$

Assume that $\alpha \lhd p(\Theta)$. Then $S(x_{p,\tau_f}) = T(x_{p,\tau_f})$, which in turn means that $z_{p',\tau'} = w_{p',\tau'}$ and $M_0 = M_1$. It also means that $\bigcup guards_\alpha(p(\Theta)) = \{\}$, and we can conclude $\langle \Sigma, M_0 \rangle R^\Omega_\alpha \langle \Delta, M_1 \rangle$ by rule 1.

We have $S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] =^{\Theta \setminus \Omega}_\alpha T[z_{p',\tau'} \mapsto T(y_{p',\tau'})]$ since if $\alpha \lhd p'(\Theta)$ then $S(y_{p',\tau'}) = T(y_{p',\tau'})$.

Assume now instead that $\alpha \not\lhd p(\Theta)$. Then $S(x_{p,\tau_f}) = T(x_{p,\tau_f})$ is not guaranteed, and thus possibly $M_0 \neq M_1$. But since $\alpha \not\lhd p(\Theta)$, by the guard lemma we have that $\alpha \not\lhd^{\Omega'} p(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p(\Theta))$, and further since $p(\Sigma) \preceq w_f$ and $\Theta \supseteq \Sigma$ we have that $\alpha \not\lhd^{\Omega'} (w_f)$. This means that $M_0, M_1 \in H^{\Omega \cup \Omega'}_\alpha$ and we can conclude $\langle \Sigma, M_0 \rangle R^{\Omega \cup \Omega'}_\alpha \langle \Delta, M_1 \rangle$ by rule 2. We have $S[z_{p',\tau'} \mapsto S(y_{p',\tau'})] =^{\Theta \setminus \Omega}_\alpha T[w_{p',\tau'} \mapsto S(y_{p',\tau'})]$ since the equivalence relation doesn't care about variables not in the intersection of the domains of the two stores.

Subcase: $M \equiv \mathbb{E}[\textbf{bind}\ x_{p,\tau} = M_0\ \textbf{in}\ M_1]$. Here we must proceed by inspection of M_0. For all terms we have by typing of the inner term that

$$\frac{\Sigma \vdash M_0 : \tau, (r_0, w_0) \Rightarrow \Sigma' \quad \Sigma' \vdash M_1 : \tau_1, (r_1, w_1) \Rightarrow \Sigma'' \quad r_0(\Sigma') \preceq p}{\Sigma \vdash \textbf{bind}\ x_{p,\tau} = M_0\ \textbf{in}\ M_1 : \tau_1, (r_1, w_0 \sqcap w_1) \Rightarrow \Sigma''}$$

Subsubcase: $M_0 \equiv v$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], S\rangle \xrightarrow{\top} \langle \Sigma, \mathbb{E}[M_1], S[x_{p,\tau} \mapsto v]\rangle \quad \text{and}$$
$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], T\rangle \xrightarrow{\top} \langle \Delta, \mathbb{E}[M_1], T[x_{p,\tau} \mapsto v]\rangle$$

We have that $S[x_{p,\tau} \mapsto v] =_\alpha^{\Theta\setminus\Omega} T[x_{p,\tau} \mapsto v]$ and we can conclude $\langle \Sigma, \mathbb{E}[M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[M_1]\rangle$ by rule 1.

Subsubcase: $M_0 \equiv y_{p',\tau}$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = y_{p',\tau} \textbf{ in } M_1], S\rangle \xrightarrow{p'} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1], S\rangle \quad \text{and}$$
$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = y_{p',\tau} \textbf{ in } M_1], T\rangle \xrightarrow{p'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1], T\rangle$$

Assume $\alpha \lhd p'(\Theta)$. Then $S(y_{p',\tau}) = T(y_{p',\tau})$ and we can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1]\rangle.$$

Assume now instead that $\alpha \not\lhd p'(\Theta)$. Then $S(y_{p',\tau}) = T(y_{p',\tau})$ is not guaranteed, so we could end up with two different values. By the guard lemma we know that $\alpha \not\lhd^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p'(\Theta))$. By typing we know that $p'(\Sigma) \preceq p$, so this means that $\alpha \not\lhd^{\Omega'} p(\Theta)$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(y_{p',\tau}) \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(y_{p',\tau}) \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv \textbf{ref}_{p''}\ y_{p',\tau'}$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \textbf{ref}_{p''}\ y_{p',\tau'} \textbf{ in } M_1], S\rangle \xrightarrow{p'} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \ell_{p'',\tau'} \textbf{ in } M_1], S[\ell_{p'',\tau'} \mapsto S(y_{p',\tau'})]\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \textbf{ref}_{p''}\ y_{p',\tau'} \textbf{ in } M_1], T\rangle \xrightarrow{p'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \ell_{p'',\tau'} \textbf{ in } M_1], T[\ell_{p'',\tau'} \mapsto T(y_{p',\tau'})]\rangle$$

We reason that we have the required equality on the memories like we did for the reference creation at top level. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \ell_{p'',\tau'} \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,ref_{p''}\,\tau'} = \ell_{p'',\tau'} \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv !y_{p',ref_{p''}\,\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = !y_{p',ref_{p''}\,\tau} \textbf{ in } M_1], S\rangle \overset{p'\sqcap p''}{\rightarrow} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(S(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1], S\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = !y_{p',ref_{p''}\,\tau} \textbf{ in } M_1], T\rangle \overset{p'\sqcap p''}{\rightarrow} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(T(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1], T\rangle$$

By typing we know that

$$\frac{}{\Sigma \vdash !y_{p',ref_{p''}\,\tau} : \tau, (p'(\Sigma) \sqcup p''(\Sigma), \top) \Rightarrow \Sigma}$$

Assume $\alpha \lhd p'(\Theta)$ and $\alpha \lhd p''(\Theta)$. Then $S(S(y_{p',ref_{p''}\,\tau})) = T(T(y_{p',ref_{p''}\,\tau}))$ and we can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(S(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1]\rangle R^{\Omega}_{\alpha} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(T(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1]\rangle.$$

Assume now instead that $\alpha \not\lhd p'(\Theta)$ or $\alpha \not\lhd p''(\Theta)$. Then we cannot guarantee $S(S(y_{p',ref_{p''}\,\tau})) = T(T(y_{p',ref_{p''}\,\tau}))$, so we could end up with two different values. By the guard lemma we know that $\alpha \not\lhd^{\Omega'} p'(\Theta)$ and $\alpha \not\lhd^{\Omega''} p''(\Theta)$ where $\Omega' = guards_{\alpha}(p'(\Theta))$ and $\Omega'' = guards_{\alpha}(p''(\Theta))$. But since $p'(\Sigma) \sqcup p''(\Sigma) \preceq p$ and $\Theta \supset \Sigma$ we know that $\alpha \not\lhd^{\Omega' \cup \Omega''} p$ and thus we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = S(S(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1]\rangle R^{\Omega \cup \Omega' \cup \Omega''}_{\alpha} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = T(T(y_{p',ref_{p''}\,\tau})) \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv y_{p',ref_{p''}\,\tau} := z_{p',\tau}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,unit} = y_{p',ref_{p''}\,\tau} := z_{p',\tau} \textbf{ in } M_1], S\rangle \overset{\top}{\rightarrow} \langle \Sigma, \textbf{bind } x_{p,unit} = () \textbf{ in } M_1, S[S(y_{p',ref_{p''}\,\tau}) \mapsto S(z_{p',\tau})]\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,unit} = y_{p',ref_{p''}\,\tau} := z_{p',\tau} \textbf{ in } M_1], T\rangle \overset{\top}{\rightarrow} \langle \Delta, \textbf{bind } x_{p,unit} = () \textbf{ in } M_1, T[T(x_{p',ref_{p''}\,\tau}) \mapsto T(z_{p',\tau})]\rangle$$

We reason that we have the required equality on the memories like we did for the assignment at top level. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,unit} = () \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,unit} = () \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv \textbf{if } y_{p',bool} \textbf{ then } N_0 \textbf{ else } N_1$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{if } y_{p',bool} \textbf{ then } N_0 \textbf{ else } N_1 \textbf{ in } M_1], S\rangle \xrightarrow{p'} \langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_i \textbf{ in } M_1], S\rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{if } y_{p',bool} \textbf{ then } N_0 \textbf{ else } N_1 \textbf{ in } M_1], T\rangle \xrightarrow{p'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_j \textbf{ in } M_1], T\rangle$$

Assume $\alpha \lhd p'(\Theta)$. Then $N_i = N_j$ and we conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_i \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_j \textbf{ in } M_1]\rangle$$

Assume now instead that $\alpha \not\lhd p'(\Theta)$, then possibly $N_i \neq N_j$. By the guard lemma we know $\alpha \not\lhd^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p'(\Theta))$. By the same reasoning as for a conditional at top level we know that $\langle \Sigma, N_i\rangle R_\alpha^{\Omega\cup\Omega'} \langle \Delta, N_j\rangle$.

By $p'(\Sigma) \preceq p$ and $\Theta \supseteq \Sigma$ we know $\alpha \not\lhd^{\Omega'} p$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_i \textbf{ in } M_1]\rangle R_\alpha^{\Omega\cup\Omega'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_j \textbf{ in } M_1]\rangle.$$

Subsubcase: $M_0 \equiv y_{p',\tau_f}\ z_{p'',\tau'} \textbf{ where } \tau_f = (\tau', p'') \xrightarrow{\Sigma, r_f, w_f, \Sigma'} \tau$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, y_{p',\tau_f}\ z_{p'',\tau'}, S\rangle \xrightarrow{p'} \langle \Sigma, N_0, S[w_{p'',\tau'} \mapsto S(z_{p'',\tau'})]\rangle \quad \text{and}$$
$$\langle \Delta, y_{p',\tau_f}\ z_{p'',\tau'}, T\rangle \xrightarrow{p} \langle \Delta, N_1, T[w'_{p'',\tau'} \mapsto T(z_{p'',\tau'})]\rangle$$

where $S(y_{p',\tau_f}) = \lambda w_{p'',\tau'}.N_0$ and $T(y_{p',\tau_f}) = \lambda w'_{p'',\tau'}.N_1$.

We reason that we have the required equality on memories like we did for application at top level.

Assume $\alpha \lhd p'(\Theta)$. Then $N_0 = N_1$ and we conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1]\rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_1 \textbf{ in } M_1]\rangle.$$

Assume now instead that $\alpha \not\in p'(\Theta)$, then possibly $N_0 \neq N_1$. By the guard lemma we know $\alpha \not\in^{\Omega'} p'(\Theta)$ where $\Omega' = \bigcup guards_\alpha(p'(\Theta))$. By the same reasoning as for an assignment at top level we know that $\langle \Sigma, N_0 \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, N_1 \rangle$.

By $p'(\Sigma) \preceq p$ and $\Theta \supseteq \Sigma$ we know $\alpha \not\in^{\Omega'} p$, and we can conclude (by rule 3) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1] \rangle R_\alpha^{\Omega \cup \Omega'} \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_1 \textbf{ in } M_1] \rangle.$$

Subsubcase: $M_0 \equiv \textbf{open } \sigma$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{open } \sigma \textbf{ in } M_1], S \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, \textbf{bind } x_{p,\tau} = () \textbf{ in } M_1, S \rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{open } \sigma \textbf{ in } M_1], T \rangle \rightarrow \langle \Delta \cup \{\sigma\}, \textbf{bind } x_{p,\tau} = () \textbf{ in } M_1, T \rangle$$

We can conclude (by rule 1) that

$$\langle \Sigma \cup \{\sigma\}, \mathbb{E}[\textbf{bind } x_{p,\tau} = () \textbf{ in } M_1] \rangle R_\alpha^\Omega \langle \Delta \cup \{\sigma\}, \mathbb{E}[\textbf{bind } x_{p,\tau} = () \textbf{ in } M_1] \rangle.$$

Subsubcase: $M_0 \equiv \textbf{close } \sigma$. Similar to the previous case.

Subsubcase: $M_0 \equiv \textbf{rec } y_{\perp,\tau}.v$. For $\Theta \supseteq \Sigma$ we have $S =_\alpha^\Theta T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{rec } y_{\perp,\tau}.v \textbf{ in } M_1], S \rangle \rightarrow \langle \Sigma \cup \{\sigma\}, \textbf{bind } x_{p,\tau} = v \textbf{ in } M_1, S[y_{\perp,\tau} \mapsto v] \rangle$$

and

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = \textbf{rec } y_{\perp,\tau}.v \textbf{ in } M_1], T \rangle \rightarrow \langle \Delta \cup \{\sigma\}, \textbf{bind } x_{p,\tau} = v \textbf{ in } M_1, T[y_{\perp,\tau} \mapsto v] \rangle$$

By typing of M_0 we know

$$\frac{\Sigma \vdash v : \tau, (\perp, \top) \Rightarrow \Sigma}{\Sigma \vdash \textbf{rec } y_{\perp,\tau}.v : \tau, (\perp, \top) \Rightarrow \Sigma}$$

and we have $S[y_{\perp,\tau} \mapsto v] =_\alpha^{\Theta \setminus \Omega} T[y_{\perp,\tau} \mapsto v]$. We can conclude (by rule 1) that

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1] \rangle R_\alpha^\Omega \langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1] \rangle.$$

Case: $\langle \Sigma, M\rangle R^{\Omega}_{\alpha}\langle \Delta, N\rangle$ **by rule 2.** We have that $M, N \in H^{\Omega}_{\alpha}$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and $\langle \Sigma, M, S\rangle \xrightarrow{p} \langle \Sigma', M', S'\rangle$. By the highness lemma we know that $S' =^{\Theta\setminus\Omega}_{\alpha} S$, and so we can choose to match this by taking 0 steps for N, i.e. $\langle \Delta, N, T\rangle \to^0 \langle \Delta, N, T\rangle$, and since $S =^{\Theta\setminus\Omega}_{\alpha} T$, by transitivity we have $S' =^{\Theta\setminus\Omega}_{\alpha} T$ as required. By lemma 6 we know that subject reduction preserves the highness property, so we have $M' \in H^{\Omega}_{\alpha}$, and thus $M', N \in H^{\Omega'}_{\alpha}$ where $\Omega' = \Omega \cup \bigcup guards_{\alpha}(p(\Theta))$ and we conclude $\langle \Sigma', M'\rangle R^{\Omega'}_{\alpha}\langle \Delta, N\rangle$ by rule 2.

Case: $\langle \Sigma, M\rangle R^{\Omega}_{\alpha}\langle \Delta, N\rangle$ **by rule 3.** We have that $M = \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0 \textbf{ in } M_1]$ and $N = \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1]$ and that $\langle \Sigma, M_0\rangle R^{\Omega}_{\alpha}\langle \Delta, N_0\rangle$ and $\alpha \not\in^{\Omega} p$. Here we can separate two cases — either M_0 is a value, or we can reduce in M_0.

Subcase: $M_0 \equiv v$. For $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and

$$\langle \Sigma, \mathbb{E}[\textbf{bind } x_{p,\tau} = v \textbf{ in } M_1], S\rangle \xrightarrow{\top} \langle \Sigma, \mathbb{E}[M_1], S[x_{p,\tau} \mapsto v]\rangle$$

If M_0 is a value, then by lemma 5 we have N_0 is high. We have that either $\langle \Delta, N_0, T\rangle \Uparrow$, in which case we have $\langle \Delta, N, T\rangle \Uparrow$, or $\langle \Delta, N_0, T\rangle \to^* \langle \Delta', v', T'\rangle$. In the latter case we have $\forall\Theta.T =^{\Theta\setminus\Omega}_{\alpha} T'$ by lemma bigstep high and by transitivity that $S =^{\Theta\setminus\Omega}_{\alpha} T'$. This means we can match the step in M by

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T\rangle \to^* \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = v' \textbf{ in } M_1], T'\rangle \to \langle \Delta', \mathbb{E}[M_1], T'[x_{p,\tau} \mapsto v']\rangle$$

Since we know $\alpha \not\in^{\Omega} p$ we have $S[x_{p,\tau} \mapsto v] =^{\Theta\setminus\Omega}_{\alpha} T'[x_{p,\tau} \mapsto v']$. We can conclude (by rule 1) that $\langle \Sigma, \mathbb{E}[M_1]\rangle R^{\Omega}_{\alpha}\langle \Delta', \mathbb{E}[M_1]\rangle$.

Subcase: $M_0 \notin \text{Val}$. By the progress lemma this means we can reduce M_0, so for $\Theta \supseteq \Sigma$ we have $S =^{\Theta}_{\alpha} T$ and $\langle \Sigma, M_0, S\rangle \xrightarrow{p'} \langle \Sigma', M_0', S'\rangle$. By the induction hypothesis we have that

either $\exists \Delta', N_0', T'.\langle \Delta, N_0, T\rangle \to^* \langle \Delta', N_0', T'\rangle$ and $S' =^{\Theta\setminus\Omega}_{\alpha} T'$
and $\langle \Sigma', M_0'\rangle R^{\Omega'}_{\alpha}\langle \Delta', N_0'\rangle$ where $\Omega' = \Omega \cup guards_{\alpha}(p'(\Theta))$,

or $\langle \Delta, N_0, T\rangle \Uparrow$.

In the latter case we have

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T\rangle \Uparrow$$

For the former case we can choose to match the reduction in M by

$$\langle \Delta, \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0 \textbf{ in } M_1], T\rangle \rightarrow^* \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0' \textbf{ in } M_1], T'\rangle$$

and we conclude

$$\langle \Sigma', \mathbb{E}[\textbf{bind } x_{p,\tau} = M_0' \textbf{ in } M_1]\rangle R_\alpha^{\Omega'} \langle \Delta', \mathbb{E}[\textbf{bind } x_{p,\tau} = N_0' \textbf{ in } M_1]\rangle$$

by rule 3.

□

A.3 Proofs that Flow Lock Security Implies Noninterference

In this appendix we provide details of the proof that flow lock security implies noninterference. The strategy is to

- Strengthen the definition of location indistinguishability at a given level to include variables;
- Generalise the definition of noninterference to a binary relation between pairs of programs, and strengthen to include variables in the store;
- Specialise the definition of α indistinguishability to lock-free policies.
- Specialise the definition of flow lock bisimulation to lock-free programs and stores.

Recall that we consider a lattice of security levels $\langle \mathcal{L}, \sqsubseteq, \sqcup\rangle$, and a policy level : $\mathsf{Loc} \rightarrow \mathcal{L}$ that fixes the intended security level of the storage locations in the program.

We assume that these locations are typed, but we will elide typing issues in the following discussion. Programs P and Q operate over these locations, and are assumed to be of unit type, and are assumed not to perform any location allocation.

To be precise we need to define the lock-free semantics for configurations of the form $\langle P, S\rangle$. But it is easy to see that if P is lock free then the lock part of the state can simply be ignored since it neither influences computation nor does it change, so transitions for $\langle P, S\rangle$ re derived by simply projecting out the lock state in the transition system.

Definition 8 (Noninterference (Generalised)). Given two stores S and T, and a level $k \in \mathcal{L}$, define S and T to be *indistinguishable at level* k, written $S \equiv_k T$, iff the location domains of S and T are the same, and for

all $\ell \in \mathrm{dom}(S)$ and for all $x \in \mathrm{dom}S \cap \mathrm{dom}T$ such that $\mathrm{level}(\ell) \sqsubseteq k$ we have $S(\ell) = T(\ell)$. and $S(x) = T(x)$.

Now define, for each level k, the binary relation $\sim_k^{\mathrm{NI}}$ on lock-free programs as follows: $P \sim_k^{\mathrm{NI}} Q$ if for all S and T such that $S \equiv_k T$, whenever $\langle P, S\rangle$ and $\langle Q, T\rangle$ are terminating configurations, $\langle P, S\rangle \rightarrow^* \langle (), S'\rangle$ and $\mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$ and $S \equiv_k T$, then there exists a T' such that $\langle Q, T\rangle \rightarrow^* \langle (), T'\rangle$, and $S' \equiv_k T'$.

The following lemma states that these definitions are indeed generalisations, and can be seen by inspection of the definitions:

Lemma 10. *1. For all lock free stores S and T, $S \equiv_k T$ implies $S =_k T$.*

2. For all closed (i.e. variable free) lock free programs P, if $P \sim_k^{\mathrm{NI}} P$ for all k, then P is noninterfering.

Now we build a bridge from the opposite side, by specialising the definition of flow lock security to lock-free programs. Firstly we note that for lock free stores, level indistinguishability $\equiv_k$ given above coincides with the indistinguishability relation $=_k^\Theta$ for any Θ, i.e.

Lemma 11. $S =_k^\Theta T \iff S \equiv_k T$

The proof is again just by inspection of the definition, so we omit a detailed argument. Now we turn to the definition of bisimulation for lock-free programs.

Lemma 12. *Define the largest symetric relation between lock-free programs, $\approx_k$, such that whenever*

$$P \approx_k Q \;\&\; S \equiv_k T \;\&\; \mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\} \;\&\; \langle P, S\rangle \rightarrow \langle P', S'\rangle$$

then there exits Q', T' such that

either $\langle Q, T\rangle \rightarrow \langle Q', T'\rangle$ & $S \equiv_k T$ & $P' \approx_k Q'$,

or $\langle Q, T\rangle \Uparrow$,

Then we have that $\langle \Sigma, P\rangle \sim_\alpha^\Omega \langle \Delta, Q\rangle$ implies $P \approx_k Q$.

The proof is again straightforward by specialisation of the bisimulation definition, and using the preceeding lemma.

Now we can provide the proof that if P is flow lock secure then P is noninterfering.

Proof. (Theorem 1) Suppose that closed program P is flow lock secure. I.e. for all levels k $\langle\{\}, P\rangle \sim_k \langle\{\}, P\rangle$. By lemma 12 this implies that $P \approx_k P$. We will prove that $P \approx_k P$ implies $P \equiv_k P$, from which it follows that P is noninterferring by lemma 10.

In order to prove that $P \approx_k P$ implies $P \equiv_k P$ we will prove the more general statement, namely that

$$\forall P, Q. P \approx_k Q \Longrightarrow P \equiv_k Q$$

i.e. we prove this for open P and Q.

Assume that $P \approx_k Q$ and that $\langle P, S\rangle \rightarrow^n \langle(), S'\rangle$, $\langle Q, T\rangle$ is terminating, $S \equiv_k T$ and
$\mathrm{dom}(S')\backslash\mathrm{dom}(S)\cap\mathrm{dom}(T) = \{\}$. We are then required to show that $\langle Q, T\rangle \rightarrow^n \langle(), T'\rangle$ for some T' such that $S' \equiv_k T'$, and we do so by induction by induction on n

Base case: $n = 0.$ In this case $P = ()$ and hence $S = S'$. By the convergence assumption we know that $\langle Q, T\rangle \rightarrow \cdots \rightarrow \langle Q_i, T_i\rangle \rightarrow \cdots \rightarrow \langle(), T_m\rangle$ for some stores T_i, and since we are free to choose store variable names, we can assume that $\mathrm{dom}(T_i) \setminus \mathrm{dom}(T) \cap \mathrm{dom}(S) = \{\}$. By symmetry $Q \approx_k P$, and thus from the definition of $\approx_k$, each of these computation steps from can only be matched by taking zero steps from $\langle P, S\rangle$, and hence $S \equiv_k T_m$ as required.

Inductive case: $\langle P, S\rangle \rightarrow \langle P_1, S_1\rangle \rightarrow^* \langle(), S'\rangle.$ Since $\mathrm{dom}(S')\backslash\mathrm{dom}(S)\cap \mathrm{dom}(T) = \{\}$, and since computation only increases the domain of the store, $\mathrm{dom}(S') \subseteq \mathrm{dom}(S_1)$, and hence $\mathrm{dom}(S_1)\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$. Given this, by assumption that $P \approx_k Q$ and from the fact that $\langle Q, T\rangle$ does not diverge, we know that $\langle Q, T\rangle \rightarrow^* \langle Q_1, T_1\rangle$ for some T_1 such that $S_1 \equiv_k T_1$. Since $\mathrm{dom}(S')\backslash\mathrm{dom}(S) \cap \mathrm{dom}(T) = \{\}$ and $\mathrm{dom}(S_1) \supseteq \mathrm{dom}(S)$ it follows that $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1) \cap \mathrm{dom}(T) = \{\}$. Now we know that $\mathrm{dom}(T_1) = \mathrm{dom}(T)\cup X$ for some set of variables X. We can assume that X is chosen to be disjoint from $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1)$, and hence we have that $\mathrm{dom}(S')\backslash\mathrm{dom}(S_1)\cap \mathrm{dom}(T_1) = \{\}$. Now we can apply the induction hypothesis to obtain the existance of a T' such that $\langle Q_1, T_1\rangle \rightarrow^* \langle(), T'\rangle$ $S_1 \equiv_k T_1$ as required. □