SEMANTICS-DRIVEN DESIGN AND IMPLEMENTATION OF HIGH-ASSURANCE HARDWARE

A Dissertation presented to
the Faculty of the Graduate School
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

---

by

ADAM PROCTER

Dr. William L. Harrison, Dissertation Supervisor

DEC 2014

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

SEMANTICS-DRIVEN DESIGN AND IMPLEMENTATION
OF HIGH-ASSURANCE HARDWARE

presented by Adam Procter, a candidate for the degree of Doctor of Philosophy, and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. William L. Harrison

Dr. Gerard Allwein

Dr. Michela Becchi

Dr. Prasad Calyam

Dr. Rohit Chadha

ACKNOWLEDGMENTS

Without a doubt I owe the greatest debt of gratitude to my advisor, Professor William L. Harrison. It has been an immense pleasure to have played a small part in his research program's growth from a tiny group of researchers laboring in a lead-lined room in the basement of the old student health building into the Center for High Assurance Computing (CHACO), which now counts three tenure-track faculty members, two postdoctoral researchers, four Ph.D. students, and one standard poodle named Pearl (*not* Perl) among its members and associates.

I also wish to thank Ian Graves, Benjamin Schulz, Chris Hathhorn, Dr. Soumya Deepta Sanyal, Andrew Lukefahr, and all the members of CHACO past and present for their friendship, good humor, collaborative spirit, and countless illuminating discussions both technical and non-technical. Chris Hathhorn in particular has provided an enormous amount of feedback on this dissertation, including but not limited to the discovery of numerous typos. His help has improved the presentation of this work substantially. (As seems to be standard practice, however, I must state for the record that any remaining errors are my own.)

Of course I also extend my thanks to the members of my dissertation committee: Dr. Gerard Allwein of the U.S. Naval Research Laboratory, Professor Michela Becchi, Professor Prasad Calyam, and Professor Rohit Chadha. Professor Becchi has been an enormously generous collaborator and role model as I have fumbled towards establishing myself as a researcher. Dr. Allwein has been a great friend to the lab, and a great contributor the development of my research over the years. Professor Chadha's logical insight is a constant source of amazement for me, and

I greatly enjoyed the opportunity to provide assistance to him in organizing the 2014 Midwest Verification Day this past October—though not, unfortunately, as much assistance as I had originally intended to provide, due to the time pressures of finalizing this dissertation; here again Chris Hathhorn saved the day! I look forward to future collaboration with everyone here at Mizzou as I continue on as a postdoctoral researcher at CHACO.

Professor David Andrews of the University of Arkansas and his former student Dr. Jason Agron of Intel Corporation, as well as Professor Andy Gill of the University of Kansas, have been enormously generous with their time and expertise in the areas of reconfigurable computing, functional programming, and hardware design languages. Professor Aaron Stump of the University of Iowa was gracious enough to host me for a summer visit in 2010, which ultimately resulted in my first publication as first author. The summer I spent there, and the many discussions I had with with Professor Stump and his students at the time, especially Professor Harley D. Eades III (now of Georgia Regents University), opened my mind to the vast possibilities and enormous depth of type theory, an area which I am excited to explore further in the future.

During the latter part of my graduate career I was supported by a U.S. Department of Education Graduate Assistance in Areas of National Need (GAANN) Fellowship (grant number P200A100053). Professor Chi-Ren Shyu was the principal investigator for this grant, and I am most grateful both for the financial support it provided and for Professor Shyu's guidance during this time. Prior to this I was supported for a time by a fellowship funded by the Gilliom Cyber Security Gift Fund.

I thank Hiromi Miyagi-Lusthaus, now of Boston University, for teaching me both the finer points of the Japanese language and how to face a classroom full of bored undergraduates with confidence; Dr. Zachary Ernst of Narrative Science Inc. for his tutelage in matters logico-philosophical; Fadi Muheidat of the Department of Electrical and Computer Engineering for his excellent instruction in the VHDL course that turned out to be crucial to my dissertation work; and my father Dr. Michael Procter for teaching me how to program in the first place some twenty-three years ago. (I remember coming to him once with a source code printout for a buggy program I had written. He identified that I was generating a random number via the built-in BASIC function RND, which was a nullary function but, due to BASIC's syntax, looked exactly like a variable. I was then attempting to re-use the same random number I had previously generated by reading the "variable" RND again at a later program point and, of course, this did not produce the expected result. This anecdote shows that programming with effects has been a thorn in my side for a very long time. It was not until I was introduced to the idea of referential transparency around fifteen years later that I finally found satisfaction.)

Ian Graves and William Lyle "William Leonard Moore" Moore, thanks for being cool dudes and awesome friends. Guess what? There are no length restrictions on these acknowledgments, so you might say: *it's free real estate*. Also, *I'm a doctor too*. Great Job!

Finally, none of this work would have been possible without the support of my family. With love and thanks from the bottom of my heart, I dedicate this work to my parents Professor Brenda and Dr. Michael Procter, my sister Erin, my cousins Meagan Hill and Sarah Moss, my aunt Lisa and my uncle Professor

Michael Goldschmidt, and my grandmother Edith Hamilton; and to the memory of my uncle Howard Hill and my grandfathers Roe Hamilton and Dan Procter.

Columbia, Missouri

December, 2014

Contents

List of Tables

List of Figures

ABSTRACT

Modularity, that is the division of complex systems into less complex and more easily understood parts, is a pervasive concern in computer science, and hardware design is no exception. Existing hardware design languages such as Verilog and VHDL support modular design by enabling hardware designers to decompose designs into *structural* features that may be developed independently and connected together to form more complex devices. In the realm of high assurance for security, however, this sort of modularity is often of limited utility. Security properties are notoriously non-compositional, i.e. subsystems that independently satisfy some security property cannot necessary be relied upon to maintain that property when operating in tandem.

The aim of this research is to establish *semantically modular* techniques for hardware design and implementation, in contrast to the conventional structural notion of modularity. A semantically modular design is constructed by adding "layers" of semantic features, such as state and reactivity, one at a time. From the high assurance aspect, semantic modularity enables different layers of semantic features to be reasoned about independently, greatly simplifying the structure of correctness proofs and improving their reusability. The major contribution of this work is a prototype compiler called ReWire which translates semantically modular hardware specifications to efficient implementations on FPGAs. In this dissertation I present the design and implementation of the ReWire compiler, along with a number of case studies illustrating both the practicality of the ReWire compiler and the elegance of the semantically modular approach to hardware verification.

Chapter 1

Introduction

In this dissertation, I advocate a novel approach to confronting the complexities and interlocking concerns of hardware design and verification. The key contributions of this work are threefold.

1. A **novel, semantically modular style of hardware specification**. In contrast with traditional design techniques typified by mainstream hardware design languages like VHDL, semantically modular designs may easily be extended with new semantic features without the need to rearchitect large portions of the design.

2. A **semantics-guided approach to hardware verification**, where separate semantic features may be reasoned about independently, thus reducing the complexity of formal verification both for new designs and for existing designs extended with new features.

3. The **development and implementation of novel compilation techniques en-**

abling circuit generation directly from high-level semantic specifications.

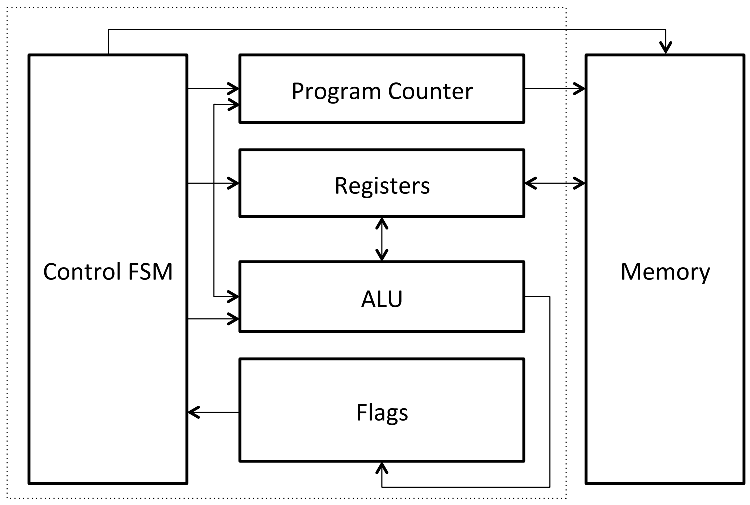
The first contribution is achieved by applying modular monadic semantics [**?**, **?**] to hardware design. I will demonstrate via several case studies that modular monadic hardware designs possess a high degree of semantic extensibility. The second contribution is also supported by the choice of modular monadic semantics. I will demonstrate that existing reasoning techniques grounded in modular monadic semantics result in a style of deductive hardware verification that scales as the semantic complexity of designs increases. The third contribution takes the form of a newly developed compiler called ReWire, which translates modular monadic specifications written in a subset of the pure functional programming language Haskell into efficient FPGA-based implementations. ReWire provides built-in support for a monadic construct called *reactive resumptions*, which enable the specification of systems combining reactivity and other sorts of effects in a modular monadic style. Taking reactive resumptions as the core abstraction means that the formal semantics of hardware specifications codifies precisely the expected timing properties of the implementation.

This chapter introduces the challenges that my doctoral research addresses, and gives a high-level overview of the tools and techniques that underpin that research. Section 1.1 contrasts two notions of modularity—structural modularity and semantic modularity—as they pertain to hardware design, and argues that (1) semantic modularity is often more important than structural modularity, and (2) existing hardware design languages and tools do not provide sufficient support for semantic modularity. Section 1.3 outlines the use of modular monadic semantics (MMS) as a vehicle for semantically modular hardware design. Section 1.4 gives an overview of existing work on modular monadic semantics as a technique for structuring security proofs. Section 1.5 discusses the challenges of synthesizing efficient circuits from monadic specifications.

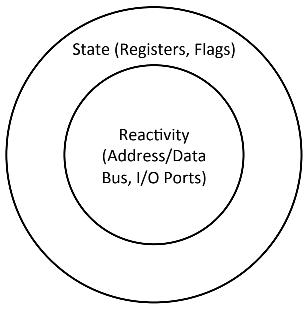
1.1 Structural Modularity vs. Semantic Modularity

Modularity, that is the division of complex systems into less complex and more easily understood subsystems, is a pervasive concern in computer science. Hardware design is no exception. While hardware designers are ultimately concerned with the fabrication of working devices constructed of basic components such as logic gates and flip-flops, modern hardware designs are so complex that high level design abstractions are absolutely essential. This need for high level abstractions leads directly to a need for high level design languages. It is reasonable to ask, therefore, whether existing languages actually offer the *right* high level abstractions: do the languages we are using support the kind of abstractions we need to construct hardware that is both efficient *and* easy to reason about?

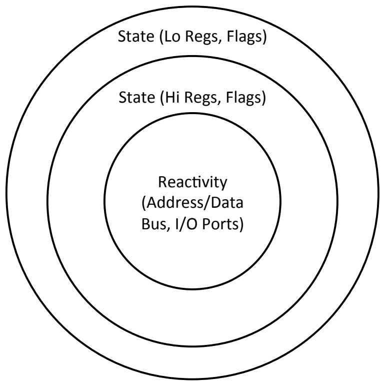
In one sense, conventional hardware design languages such as VHDL and Verilog *do* support modular design. For example, a CPU design in VHDL might be broken down into one module for the ALU, one module for the register file, one module for the microcode logic, and so on (Figure 1.1a). These subcomponents may then be connected together to form a working CPU. This paradigm—let us call it *structural modularity*—serves designers well when the expected behavior of the device as determined by, for example, the semantics of an processor's instruction set, is fixed, and all one needs to do is construct a device conforming to that fixed



(a) Structurally Modular CPU Design



(b) Semantically Modular CPU Design



(c) Semantically Modular CPU w/Separation

Figure 1.1: Structural vs. Semantic Modularity

semantics.

When a hardware designer wishes to explore semantically novel ideas, however, structural modularity may be of limited utility. Suppose, for example, that we wish to augment an existing CPU design with support for separation among security domains [**?**] at the hardware level, enabling the safe interleaving of processes handling both classified and unclassified data. Even if the existing design is structurally modular, we are inevitably faced with the fact that separation is a cross-cutting concern, touching all facets of the design's structure. Put another way, there is no obvious place in the structural diagram of Figure 1.1a to insert a "separation module". Does this new feature go between the control FSM and the memory? Between the registers and memory? Between the ALU and flags register? The answer is likely to be some or all of the above. This leaves us two choices:

1. restructure our design to support separation, or
2. retrofit an existing structure to support separation.

Each of these choices, however, comes with a major drawback. Choice (1) may require us to discard a substantial amount of already-expended engineering effort. Choice (2) may add substantially to the complexity of the design, and we are still left with the question of whether the newly restructured design implements separation correctly.

From a formal methods standpoint, the problem is even more vexing. Effective and scalable formal methods *require* the existence of a concise, abstract, and mathematically elegant semantics. Structural modularity offers no such thing. Nor does it offer any assurances that our design is faithful to the intended semantics. If the need for rigor is not taken into account from the very beginning of the design process, the resulting design will be far more difficult to verify.

At the core of this dissertation is a novel hardware design process that is driven not by structural modularity, but by *semantic modularity*. Figure 1.1 illustrates the difference. In contrast to the structurally modular block diagram of Figure 1.1a, a semantically modular specification is constructed by adding "layers" of semantic features. The layered semantic universe of our simple CPU is illustrated in Figure 1.1b. At the core, we have the semantic realm of *reactivity*, that is, responding to input and output signals. Layered on top of this we have a semantic notion of *state*, i.e. registers and flags. Semantic modularity makes the transition from the non-separating CPU of Figure 1.1b to the separating CPU of Figure 1.1c very simple. We need only add one more layer of state to the semantics, representing the extra layer of state corresponding to the privileged (high-security) domain. Critically, parts of the design pertaining only to the pre-existing functionality are largely unchanged. The advantages of semantic modularity extend to the domain of verification, as well. A semantically modular design along the lines of Figure 1.1c allows the different layers of semantic features to be reasoned about more or less independently. If, for example, we have a correctness proof for our implementation of one of the CPU's instructions, this correctness proof will be reusable for the separating CPU as well.

1.2 Making Semantics-Driven Design a Reality

The particular approach to semantics-directed hardware design advocated in this work has three main ingredients. The first is an idea borrowed from programming language semantics known as *modular monadic semantics* (MMS). MMS arose from the observation that denotational semantics of programming languages are often difficult to construct in a modular way. The basic idea of MMS is to structure an interpreter or compiler for a programming language in terms of semantic building blocks called *monad transformers*. Beginning with a core type of pure, effect-free computations, one may construct an enriched semantic universe with features such as updatable state, concurrency, reactivity, non-determinism, and I/O one "layer" at a time, by applying a monad transformer for each semantic feature. The present work uses MMS to separate the various semantic concerns pertaining to hardware designs: if a design requires a mutable store, this will be reflected in the use of a state monad transformer. If a design requires separate state domains, this will be reflected in the use of a layered state monad [**?**].

The second ingredient is a somewhat less well known construction called a *reactive resumption monad*. The reactive resumption monad is the essence of synchronous, reactive computation. It forms a semantic domain of computational processes in which a process's state is transformed in response to each value that arrives on a synchronous input channel. This transformation of state is assumed to happen "instantaneously", much like state transitions in a finite automaton. (In hardware implementation, "instantaneously" may simply mean "fast enough that all the work is done by the time the next clock pulse arrives.") The corresponding reactive resumption monad *transformer* will allow us to combine reactivity with semantic features such as state in an *à la carte* fashion.

The third and final ingredient is a prototype compiler called ReWire, which produces synthesizable VHDL code from hardware designs written in a monadic calculus. This calculus, called ReWire Core (RWC), provides support for stateful and reactive computation. It borrows a concrete syntax from Haskell—and in fact, can be interpreted as an embedded domain-specific language in Haskell—but restricts recursion in various ways that ensure realizability in hardware.

1.3 Background: Modular Monadic Semantics

In the realm of programming languages, the distinction between structural modularity (e.g., the partitioning of a compiler into distinct phases of lexing, parsing, static analysis, code generation, optimization, and instruction selection) and semantic modularity (e.g., the construction of an interpreter whose object language may be extended with new semantic features in a modular way) has been the subject of a great deal of research. The fruits of this research [**?**, **?**, **?**, **?**, **?**] are a paradigm known as *modular monadic semantics* or MMS. This section gives background information on MMS.

1.3.1 Language of Discourse: Haskell

Haskell [**?**] is a strongly-typed, purely functional programming language with a non-strict semantics. “Purely functional” means that functions in Haskell really *are* functions in the mathematical sense. That is, a Haskell function of type $Int \to Int$ will always map any given integer to the same result value; if $f(x) = 3$ right now,

$f(x) = 3$ tomorrow and the day after as well. Furthermore, evaluation of the function produces no side effects. It is not possible that $f : Int \rightarrow Int$ will, say, mutate some state variable or overwrite a file on its way to computing its final result. "Non-strict", for our purposes, means that evaluation of expressions is delayed until the value of that expression is actually needed. Non-strictness is an essential feature of Haskell, closely tied to functional purity.

Haskell's purely functional, non-strict semantics make it a favorite tool of mathematically-minded computer scientists and programmers. Due to the absence of side effects, Haskell exhibits a very useful property called *referential transparency*; roughly speaking, this means that one may substitute "equals for equals" without changing the meaning of a program. This property does not hold for effectful languages like C or Java. But the purely functional nature of Haskell comes at a cost. Many real world programs actually *need* I/O, and other classes of effect like mutable state are often necessary to implement a program efficiently. How can one possibly hope to implement these features in a language that, by its very design, shuns side effects?

The answer is that one may use a *monad* [**?**, **?**]. From a programming point of view, a monad is a construction that allows us to embed effectful programming features inside of a programming language that does not directly support them. At the type level, monads provide a separation between ordinary values (which have types like *Int*) and computations (which have types like *M Int*, where *M* is a monad). Monads have proven to be a perfect fit for Haskell; they are manifested both in an opaque abstraction for interfacing with the outside world called the *IO monad* [**?**], and in a large class of programmer-defined abstractions that enable support for everything from parser construction [**?**] to concurrent programming [**?**].

Due to the popularity of monads in the Haskell community, Haskell is also the language that is most widely used by researchers to discuss, express, and explore the subject of monadic computation. For this dissertation, I have made the same choice. Apart from the very formal semantics given in Chapter 3, where monads are expressed in a mathematical notation more commonly seen in denotational semantics (replete with Greek letters and oddly shaped brackets), most high-level discussion of monads and monadic programming will use Haskell as a surface language. Indeed, the main contribution of this work is a monadic programming language that itself borrows (a proper subset of) Haskell's concrete syntax.

1.3.2 Monads

Modular monadic semantics is founded on algebraic structures called *monads*, originally discovered in the context of category theory. Before delving into the mathematical particulars, it is useful to consider the motivation behind monads. To a computer scientist, a monad may be thought of as a way of assigning a denotational semantics to languages with effects. By "effects" we mean any computational notion that brings us outside the domain of pure, mathematical functions. For example, in a language like C that features mutable global state, a "function" `int f(int x,int y)` does not necessarily correspond semantically to any mathematical function $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, for the simple reason that evaluating `f(a,b)` might, as a side effect, read and/or alter some global variable, or even make a series of system calls that results in personal information being transmitted over the Internet. Monads allow us to deal with this fact in a mathematically precise yet well-structured fashion—in this case, our C function *does* correspond to a mathematical function $f : \mathbb{Z} \times \mathbb{Z} \to M(\mathbb{Z})$, where M is some monad (let us not worry about which!) encapsulating the effect of updatable state and any other "impure" semantic features that C offers.

Outside of programming language semantics, another area where monads have found application is in functional programming. In a purely functional programming language like Haskell, monads allow us to implement effectful computation—stateful computation, computation with I/O, and so on—without compromising the purity of the underlying language. The advantage of this approach lies partly in *type discipline*: if a function has type $Int \to Int \to M\ Int$ for some monad M, then it is clear that the function can have any of the side effects offered by the monad M, but no others.

Rather than present monads in categorical language, the introductory explanations of this chapter are written in terms of Haskell's concrete syntax, as this notation is considerably more convenient and accessible to computer scientists (the author included). We may define the notion of a monad according to Haskell's type class system as follows.

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

In other words, a monad is a type constructor (m) with two associated operators (**return** and $\gg\!=$). The operator **return** takes a value of any type a and returns a value of type $m\ a$. The operator $\gg\!=$ takes a value of type $m\ a$, and takes a value of type $a \to m\ b$ (that is, a function from a to $m\ b$), and returns a value of type $m\ b$.

Intuitively speaking:

- The type $m\ a$ represents the type of *computations* producing a value of type a. The computation may have side effects, but those side effects are restricted to those provided by the monad m.
- The expression **return** v represents a computation that has no side effects, but returns the value v.
- The expression $m \gg\!= f$ represents a computation that first "does" m, then feeds the result value of m to f, and executes the computation that results. Essentially, $\gg\!=$ is an operator of *sequencing*, except that the return value of the first computation is available to the one that follows it.

Moreover, the monad operators are expected to satisfy the following laws, though Haskell's type system contains no means of enforcing that requirement.

$$\begin{array}{lcll} m \gg\!= \mathbf{return} & = & m & \langle\text{right-unit}\rangle \\ \mathbf{return}\ x \gg\!= f & = & f\ x & \langle\text{left-unit}\rangle \\ (m \gg\!= f) \gg\!= g & = & m \gg\!= (\lambda x \to f\ x \gg\!= g) & \langle\text{associativity}\rangle \end{array}$$

The simplest example of a monad is the *identity* monad.

```
newtype Identity a = Identity a
```

This Haskell declaration has the effect of declaring a new type constructor called *Identity*, whose only data constructor is also called *Identity* and carries a value of type a. That is, the type *Identity* a merely encapsulates a, with no further structure. Formally, for every type a, the type *Identity* a is *isomorphic* to a.

We declare *Identity* to be a monad—or in Haskell terms, we declare it to be an *instance* of the type class *Monad*—as follows.

```
instance Monad Identity where
  return v          = Identity v
  Identity v >>= f = f v
```

That is, **return** simply boxes up its argument value, and $\gg=$ merely unboxes the value in the computation and feeds it to the function on the right hand side. For the sake of convenience, we can also define a function to project result values out of the monad.

```
runIdentity :: Identity a -> a
runIdentity (Identity v) = v
```

So what notion of effects is represented by this monad? The answer is: nothing! The identity monad is, in fact, a trivial monad of effect-free computations. The only thing one can do with it is return a value, and feed the results of one computation into another. When we are dealing later with monad *transformers*, however, it will serve as a useful base on which to build more complex monads.

Let us now consider a less trivial monad, called the *state* monad. The state monad allows us to express computations with state that may be read and written as the computation progresses. Its underlying type constructor is as follows.

```
newtype State s a = State (s -> (a,s))
```

In other words, the type *State s a* merely wraps a function from type s to pairs of type (a, s). This type may be understood as a state transformer that also produces a return value of type a "on the side". If we want to connect one computation to another using $\gg=$, we will have to pass not just its return value but also its post state. The *Monad* instance is then:

```
instance Monad (State s) where
  return v       = State (\ s -> (v,s))
  State f >>= g = State (\ s -> let (v,s') = f s
                                in  deState (g v) s')
```

where:

```
deState :: State s -> (s -> (a,s))
deState (State f) = f
```

Unlike the identity monad, there are other useful operations besides **return** and $\gg=$ that we can define and take as primitive. The "get" operation $g :: State\ s\ s$ which returns the current state value, and the "put" operation $p :: s \rightarrow State\ s\ ()$ which overwrites the current state with a new value and returns a value of unit ("void") type, are defined as follows.

```
g   = State (\ s -> (s,s))
p v = State (\ s -> ((),v))
```

Operations like g and p, which (unlike **return** and $\gg=$) only apply to a particular monad, are sometimes referred to as *non-proper morphisms*.

With the state monad, we can define the semantics of an imperative language in a concise and straightforward way. Consider a simple language containing only three sorts of statements: *Reset*, which resets a global counter to zero, *Incr*, which increments the global counter, and *Seq*, which sequentially composes two computations. In Haskell, the abstract syntax for such a language may be written as follows.

```
data Stmt = Reset | Incr | Seq Stmt Stmt
```

The semantics of *Stmt* is given by a function from *Stmt* to computations in the *State* monad. Specifically:

```
exec :: Stmt -> State Int ()
exec Reset         = p 0
exec Incr          = g >>= \ ctr ->
                     p (ctr+1)
exec (Seq s1 s2) = exec s1 >>= \ _ ->
                     exec s2
```

This code exhibits a pattern that is very common in monadic programs, of mimicking sequentiality by placing λ-abstractions on the right hand side of $\gg=$, which (as with the variable *ctr* in the case for *Incr* in the above code) has the effect of assigning a name to the result of the subcomputation on the left so that it can be used later. In fact, this pattern is so common that Haskell provides a spoonful of syntactic sugar called *do*-notation to encapsulate it. The above code written in *do*-notation reads as follows:

```
exec :: Stmt -> State Int ()
exec Reset         = p 0
exec Incr          = do ctr <- g
                        p (ctr+1)
exec (Seq s1 s2) = do exec s1
                        exec s2
```

As a final example of a monad, let us consider the monad of potentially failing computations, called *Maybe*. The standard Haskell libraries define a data type called *Maybe* as follows.

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

This type (which is used very frequently in Haskell, even in code that is not written in monadic style) can be used to represent the possible absence of a value. Expressions of the form *Just v* represent the presence of a value *v*. The expression *Nothing*, on the other hand, reflects the absence of any value. For example, a function *safediv* that takes two integers and returns their quotient, but should return some sort of failure value when division by zero is requested, could be written as follows.

```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv x y = Just (x `div` y)
```

The *Maybe* type forms a monad where **return** is simply *Just*, and ≫= returns a failed computation (*Nothing*) if the input computation is failed, and otherwise passes the return value forward.

```haskell
instance Monad Maybe where
  return v       = Just v
  Nothing >>= f = Nothing
  Just v  >>= f = f v
```

Using *Maybe* as a monad is particularly handy in cases where multiple points of failure exist in a function. Consider a somewhat contrived example that takes two lists of integers, either of which may be empty, and returns the sum of their initial elements, but fails if either list is empty. Without the *Maybe* monad this code may involve an unwieldy nesting of **case** expressions, with a characteristic cascade of mappings from *Nothing* to *Nothing*.

```
addFirst :: [Int] -> [Int] -> Maybe Int
addFirst l1 l2 = case head l1 of
                   Just x  -> case head l2 of
                                Just y  -> Just (x+y)
                                Nothing -> Nothing
                   Nothing -> Nothing
  where head (x:xs) = Just x
        head []     = Nothing
```

With the *Maybe* monad we obtain a much more elegant and readable formulation.

```
addFirsts :: Int -> Int -> Maybe Int
addFirsts l1 l2 = do x <- head l1
                     y <- head l2
                     return (x+y)
  where head (x:xs) = Just x
        head []     = Nothing
```

1.3.3 Monad Transformers

We have seen how monads may be used to express particular notions of computational effect. But what if one wants to *mix* notions of effect? For example, what if one requires updatable state *and* the possibility of failure? We have seen one monad for state, and one monad for failure, but there is no obvious way to combine their notions of effect. We could define from scratch an entirely new monad of state and failure, whose type constructor would be isomorphic either to $s \to \mathit{Maybe}\,(a, s)$ or to $s \to (\mathit{Maybe}\,a, s)$, but this is a tedious and error prone process, requiring us to define the new monad's type constructor, and its **return** and $\gg=$ operations. One might wonder if simply composing the desired monads will do the trick, but sadly this is not the case in general.

As it turns out, a construct called a *monad transformer* will do the trick. A monad transformer T is a mechanism for extending a pre-existing monad M with "more" kinds of effects, producing a new monad $T\,M$. For example, if we start with a monad of non-deterministic computation, and apply the state monad transformer to that monad, this will produce a new monad of non-deterministic *and* stateful computation.

In Haskell, we define the class of monad transformers as follows.

```
class MonadTrans t where
  lift :: Monad m => m a -> (t m) a
```

Implicit in this definition is the requirement that a monad transformer must itself operate on a type constructor (because t is applied to m, which is in turn applied to a in the signature for **lift**); and that the result of applying a monad transformer (e.g., $t\,m$ in the signature for **lift**) is itself a type constructor. It is also expected, though this requirement is not enforced by the type system, that if m is a monad, so is $t\,m$; and that **lift** follows certain laws [**?**]:

$$\begin{aligned}
\mathbf{lift}\,(\mathbf{return}_m\; v) &= \mathbf{return}_{tm}\; v \\
\mathbf{lift}\,(m \gg\!=_m f) &= \mathbf{lift}\, m \gg\!=_{tm} \lambda x \rightarrow \mathbf{lift}\,(f\; x)
\end{aligned}$$

(The subscripts here are intended to disambiguate between the operations of the "base" monad m and the "lifted" monad $t\,m$.) In other words, lifting an effect-free (**return**) computation produces an effect-free computation; and sequencing two computations then lifting the sequence, is the same as lifting two computations and then sequencing them.

One useful example of a monad transformer is the *state monad transformer*, which augments an existing monad with stateful effects, akin to those provided by the

State monad seen earlier.

```
newtype StateT s m a = StateT (s -> m (a,s))

deStateT :: StateT s m a -> (s -> m (a,s))
deStateT (StateT f) = f

instance Monad m => Monad (StateT s m) where
  return v       = StateT (\ s -> return (v,s))
  StateT f >>= g = StateT (\ s -> f s >>= \ (v,s') ->
                                  deStateT (g v) s')
```

Parenthetically, it is interesting to note that *StateT Identity* is isomorphic to *State* from the previous section.

Just as with *State*, the non-proper morphisms to "get" and "put" the state can be defined for *StateT*.

```
g :: Monad m => StateT s m s
g = StateT (\ s -> return (s,s))

p :: Monad m => s -> StateT s m ()
p v = StateT (\ _ -> return ((),v))
```

Since these definitions are parametric in the base monad *m*, they can be used with any monad that has *StateT* on the top of the transformer stack. If *StateT* is not on top, **lift** may be applied as many times as needed to promote the operations into the transformed monad.

With monad transformers in hand, we now have the means to achieve semantic modularity. Let us illustrate this by extending the statement language of the preceding section with a new statement form *Fail*, representing an abnormal end to the program.

```
data Stmt = Reset | Incr | Seq Stmt Stmt | Fail
```

In order to represent the possibility of failure on the semantic side, we will change our monad from *State* to one that layers *StateT* over *Maybe*. For convenience, let us define this and give it the name *M*.

```
type M = StateT Int Maybe
```

Now the semantics of *Stmt* is defined in terms of the new monad.

```
exec :: Stmt -> M ()
exec Reset         = p 0
exec Incr          = do ctr <- g
                        p (ctr+1)
exec (Seq s1 s2) = do exec s1
                        exec s2
exec Fail          = lift Nothing
```

Pleasingly, we find that the new semantics looks exactly like the old semantics where the pre-existing commands (*Reset*, *Incr*, and *Seq*) are concerned. The only new case is that for *Fail*, which we handle by lifting the failed computation, *Nothing*, into *M*.

All of the above comes with an important caveat. While a monad transformer, by definition, always produces a monad, understanding exactly *which* monad it produces can be slightly tricky. The transformers *StateT* and *MaybeT* (which is the monad transformer analogue of the monad *Maybe*) provide a classic example of this. One ordering of the transformers:

$$StateT\ s\ (MaybeT\ Identity)\ a \quad \cong \quad s \rightarrow Maybe\ (a, s)$$

produces a monad that will not retain its state on failure. The other:

$$\mathit{MaybeT}\ (\mathit{StateT}\ s\ \mathit{Identity})\ a \quad \cong \quad s \rightarrow (\mathit{Maybe}\ a, s)$$

produces a monad that *does* retain state on failure. The sad fact in general is that monad transformers do not commute; thus some care must be taken with their use. A more fundamental annoyance is that lifted operations do not always retain useful equational properties that they have in the base monad. In spite of these caveats, however, monad transformers have proven to be an excellent basis for semantic modularity in the world of programming languages.

1.3.4 Hardware as Reactive Computation, Reactive Computation as a Monad

We have seen a few examples of monads and monad transformers reflecting modular notions of computation. Will this toolkit be sufficient for representing hardware? The answer is no. There is a critical aspect of hardware computation that cannot be represented by *StateT*, *MaybeT*, nor any of the standard menagerie of monad transformers described by Liang et al [**?**]: namely, external I/O channels. A hardware circuit may have semantic features like state on the inside, but fundamentally it must also possess the means communicating with the outside world during the course of its execution.

If we aim to apply MMS to hardware design, then, we need some way of representing I/O monadically. Let us begin by approaching the underlying type structure. A hardware circuit, viewed from the outside, has input ports, output ports, and some kind of logic on the inside that produces outputs based on inputs. This sounds very much like a *function* mapping inputs to outputs. At first blush, then, we might try representing hardware simply as such a function.

```
type Hardware i o = i -> o
```

(Note that multiple inputs or outputs could simply be represented with tuple types for *i* and *o*.)

It quickly becomes clear, however, that this type is too restrictive. Suppose we wanted to design a three-bit up-counter with a reset signal. The type of the input to this circuit is *Bit*, and the type of its output is (Bit, Bit, Bit). So we have the type *Hardware Bit* (Bit, Bit, Bit), which is synonymous with the function type $Bit \rightarrow (Bit, Bit, Bit)$. But the circuit we are trying to specify is not a simple function, in the sense that the same input may be mapped to different outputs at different times. If at time t it maps an input (reset signal) of 0 to an output of $(0, 1, 0)$, then at time $t + 1$ it should map 0 to $(0, 1, 1)$. Extrapolating from this example, the problem is that hardware is allowed to have *memory* in a general sense of the word: i.e., it is allowed to vary its behavior *over time*, based on what has happened in the past.

Evidently, we will need a richer type structure to represent sequential hardware. One way we can represent this is as a function that takes an input and returns an output, but *also* returns a new function representing the behavior expected on the *next* input.

```
newtype Hardware i o =
        Hardware (i -> (o,Hardware i o))
```

In fact, this very type structure has been used in the context of hardware design centered on an alternative structure called an *arrow*. Specifically, it is known as the automaton arrow [**?**]. Using *Hardware* we can define an up-counter with reset as follows.

```
type Input  = Bit
type Output = Bit

count :: Int -> Hardware Input Output
count x = Hardware (\ rst -> if rst == 1 then
                               (x,count 0)
                             else
                               (x,count (x+1)))

main :: Hardware Input Output
main = count 0
```

The only problem with this type structure—if our aim is to exploit the modularity and ease of reasoning afforded by monads, at least—is that it is not a monad! In particular, we cannot define **return** and $\gg\!=$ operations, because there is no form of *Hardware* value that corresponds to a "finished" computation.

To obtain a monad, we will extend the underlying type to be a sum ($T_1 + T_2$, written in Haskell as *Either* T_1 T_2). Now a computation may either be a return value of type a, *or* it output and a function waiting on an input and producing a new computation. With this, we obtain the monad of reactive computation, called *React*.

```
newtype React i o a =
          React (Either a (o,i -> React i o a))
```

The monad instance for *React* is as follows.

```
instance Monad (React i o) where
  return v = React (Left v)
  React m >>= f =
     React (case m of
              Left v      -> f v
              Right (o,k) ->
```

```
                     Right (o,\ i -> k i >>= f))
```

In other words, **return** v represents a finished computation with a return value of v. As for $\gg=$, when given a finished computation on the left it simply takes the return value and feeds it to the function on the right. When given a paused computation, it produces a new paused computation that, once the left-hand computation finishes, will pass its return value on to f.

The up-counter can now be rewritten in monadic style.

```
type Input  = Bit
type Output = Bit

count :: Int -> React Input Output ()
count x = do rst <- signal x
             case rst of
               1 -> count 0
               _ -> count (x+1)

main :: React Input Output ()
main = count 0
```

This definition makes use of a non-proper morphism for *React* called *signal*, which may be thought of as writing its argument to the output line, waiting for the next clock tick, and returning the value of the input at that tick.

```
signal :: o -> React i o i
signal x = React (Right (x,\ y -> return y))
```

Finally, we can generalize the *React* monad to the resumption monad *transformer* *ReactT* [**?**] as follows.

```
newtype ReactT i o m a =
    ReactT (m (Either a (o,i -> ReactT i o m a)))

deReactT :: ReactT i o m a ->
              m (Either a (o,i -> ReactT i o m a))
deReactT (ReactT m) = m

instance Monad m => Monad (ReactT i o m) where
  return v = ReactT (return (Left v))
  ReactT m >>= f =
   ReactT (do r <- m
              case r of
               Left v      -> deReactT (f v)
               Right (o,k) ->
                 return (Right (o,\ i -> k i >>= f)))

instance MonadTrans (ReactT i o) where
  lift m = ReactT (m >>= return . Left)
```

This reactive resumption monad transformer has already proved its usefulness in the semantics of concurrency [**?**, **?**], including the construction of verified separation kernels [**?**, **?**]. In that context, reactive resumption computations are used to represent processes on a multitasking system. The structure of the monad underlying *ReactT* controls the notion of effects that is available to processes, and a kernel (consisting of a scheduler and a set of handlers for system calls) interleaves processes (i.e., computations in *ReactT*) in a controlled fashion.

For modular monadic hardware, we will use reactive resumptions for two purposes that are related to, but distinct from, the above. First, reactive resumptions at the top level of a program represent the interactions of a hardware system with the outside world, namely the ability to read input lines and write to output lines. When reactive resumptions are used at the top level, the *i* and *o* parameters of *React* and *ReactT* are exactly analogous to the I/O signals of a VHDL entity or architecture.

Second, reactive resumptions may be used as a way of coordinating resource sharing among logically separate hardware units. For example, a dual-core processor may be realized in MMS by instantiating two single-core CPU specifications, and applying a parallel composition operator to these two instances. The parallel composition operator determines how resource contention is handled. This approach is broadly similar to Harrison's monadic separation kernels [**?**, **?**]—but here, the "processes" are hardware subsystems.

1.4 Reasoning about Security with Monads

Monads are not just a useful abstraction for programming with effects. They also provide a powerful set of *equational reasoning* principles that have a wide array of applications to program verification. In particular, Harrison and collaborators have extensively explored the application of monadic equational reasoning to security kernels [**?**, **?**, **?**, **?**]. A monadic security kernel in this style is built around *layered state monads* that make use of multiple state monad transformers. Such monads have a number of useful properties by construction, providing an elegant formalism for expressing security properties like non-interference. Specifically, it follows from the type construction of layered state monads that computations operating exclusively at one state level *commute* with those operating exclusively at a different state level; in other words, computations operating in one state domain have no effect on those operating at another. Thus if two different processes (computations in a reactive resumption monad over layered state) can be shown to operate at a single state level, it follows that their atomic state operations may safely be interleaved without introducing information flow. Put simply, storage channels via internal storage are *a priori* impossible. In this way, the verification of security properties at the kernel level is reduced to the problem of proving that the kernel *itself* does not introduce information flow via indirect channels. In practice, this leads to very concise and manageable separation proofs.

This by-construction style of non-interference proof can also be applied to hardware. For example, using the layered-state approach to construct the separating CPU of Figure 1.1c substantially lightens the proof burden, as one only needs to prove that the context-switch logic does not introduce information flow between domains. A convincing demonstration of the applicability of monadic control of effects to the construction and verification of secure hardware is one of the contributions of this dissertation.

1.5 Generating Circuit Implementations

Programming with monads requires an expressive language like Haskell with a rich type system supporting functional abstraction. But an unfortunate consequence of Haskell's expressiveness is that there are many constructs in Haskell that cannot readily be mapped onto efficient hardware implementations. In particular, higher-order functions and the unrestricted use of recursion make the synthesis of hardware directly from Haskell a very tall order indeed.

To cut this problem down to size, we can identify a *subset* of Haskell that may be translated to hardware. Chapter 3 identifies just such a subset, which we refer to as *ReWire*. The ReWire language contains, essentially, those programs that

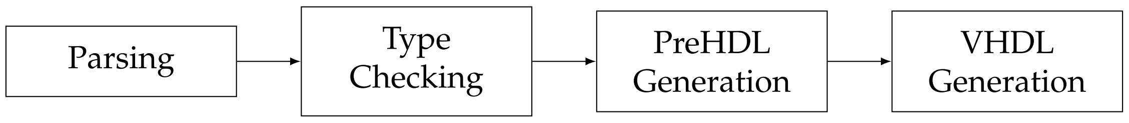


Figure 1.2: Structure of the ReWire Compiler

consist of a finite set of *guarded, mutually tail-recursive* equations whose codomains are all typed in a reactive state monad, and whose arguments are all of *finitely representable type* (e.g., bits, words, or enumerated types). Here "guarded" refers to a syntactic criterion [**?**] that ensures each recursive call ultimately produces either a *Left* or *Right* value, meaning that the next state and output signals are always well defined. ReWire maps each such equation onto a state in a finite state machine with data registers.

The structure of the ReWire compiler, which is discussed in much greater detail in Chapter 4, is outlined in Figure 1.2. Once a ReWire program has passed the parsing and type checking phase provided by what is essentially a vanilla Haskell front end, it is translated to an intermediate language called PreHDL. A few source-to-source transformations (not directly reflected in Figure 1.2) are made to the resulting PreHDL program, resulting in a final form that can be translated to an efficient VHDL program, suitable for implementation on an FPGA.

1.6 Structure of the Dissertation

The remainder of this dissertation starts with a discussion of related work in Chapter 2. This is followed in Chapter 3 with a formal definition of the ReWire language. In Chapter 4, the ReWire compiler is discussed in detail. Chapters 5 and 6 each contain a significant case study in using ReWire to specify and synthesize circuits. The former comprises a simple 8-bit CPU, and the latter a framework that uses ReWire to generate fast hardware-based regular expression matchers for packet inspection. In Chapter 7, we explore how ReWire enables support for formal verification of hardware security. Chapter 8 concludes with a summary of results and a discussion of future work.

Chapter 2

Related Work

This chapter reviews related work. It is not intended to be an exhaustive review of the literature on high-level synthesis, hardware languages, or secure hardware design, but rather to situate this research in the context of especially closely related work. Particular attention is given to ReWire's place in the not-insubstantial body of existing work on hardware design via functional programming languages. Here the overarching theme is that the emphasis on *semantics, not structure* significantly distinguishes ReWire from existing approaches.

2.1 Functional Languages in Hardware Design

There is a fairly long history of attempts to apply functional language technology to hardware specification and synthesis, most of them using Haskell. These attempts may be divided into two broad categories. On the one hand, Haskell has been used as a *host* for a number of embedded domain specific languages (EDSL). On the other, a few attempts have been made at using Haskell as a *source* language for hardware compilation. The following slightly reductive slogan may elucidate the difference between the EDSL approach and the Haskell-as-hardware-description-language (Haskell-as-HDL) approach: The EDSL approach is about circuit design *with* Haskell. The Haskell-as-HDL approach is about circuit design *in* Haskell. The general design flow in an EDSL-based approach is to use Haskell as a language for constructing circuit descriptions within a smaller, embedded language syntax. Higher-order combinators can be used to mask this fact, but in general EDSL-based circuit design techniques do not *directly* compile Haskell terms to hardware; rather, Haskell terms are used to *generate* circuit descriptions, and it is these descriptions that are then compiled to hardware. By contrast, the Haskell-as-HDL approach identifies type structures in Haskell that are amenable to direct hardware compilation. This avoids the complexity and overhead of an EDSL-based design framework. ReWire falls squarely in the Haskell-as-HDL camp.

2.1.1 Embedded Domain-Specific Languages: Lava, Hawk, and ForSyDe

Lava [**?**] is a family of Haskell-hosted domain specific languages for circuit specification, simulation, synthesis, and verification. In its original implementation, Lava provides a monadic interface for specifying circuits. A hierarchy of several different type classes allows multiple interpretations of the same specification for purposes of simulation, synthesis, and verification. The original paper on Lava [**?**] gives the following example of a half-adder. (Note that *Circuit* is a subclass of *Monad*.)

```
halfAdd :: Circuit m => (Bit,Bit) -> m (Bit,Bit)
halfAdd (a,b) = do carry <- and2 (a,b)
                   sum   <- xor2 (a,b)
                   return (carry,sum)
```

This specification is overloaded to work in any monad in the *Circuit* class. This means that the same circuit specification can be interpreted for simulation (by instantiating *m* to the "standard" simulation monad called *Std*), for synthesis to VHDL (by instantiating *m* to a member of *Symbolic*), and for verification (by instantiating *m* to an instance of *Provable*). The underlying type *Bit* can represent either a boolean literal or a bit-valued variable in Lava (distinct from Haskell variables), enabling symbolic properties of circuits to be expressed and verified via external backends such as a SAT solver.

Roughly contemporary with Lava is Hawk [**?**, **?**], a Haskell-hosted domain specific language for processor specifications that bears a great deal of similarity to Lava but was never intended to support hardware synthesis. It seems largely to have been superseded by Lava.

Lava has been extended and revised by several research groups over the years, resulting in variant implementations known as York Lava, Xilinx Lava, and Kansas Lava [**?**, **?**, **?**]. Xilinx Lava differs from classic Lava (often referred to as "Chalmers Lava" since it was developed primarily at Chalmers University of Technology) by providing a number of low-level primitives, specific to Xilinx FPGAs, that allow circuit designers to specify circuit layout precisely. Kansas Lava represents a major overhaul of Lava's internals, designed to leverage advances in Haskell's type system that have emerged since Lava was first designed in the late 1990s. One major difference is that Kansas Lava uses a unified *Signal* type to represent circuits both for simulation and for synthesis purposes. Internally, this type encompasses both a functional representation of the circuit's behavior used for simulation, and a symbolic representation of its structure used for synthesis. This dual deep and shallow embedding of circuit descriptions supplants the hierarchy of monad classes present in Chalmers Lava. Kansas Lava also introduces sized types, allowing sized bit vectors to be represented naturally in the Haskell type system.

Another Haskell DSL-based design methodology is ForSyDe ("**For**mal **Sy**stem **De**sign") [**?**, **?**, **?**]. With ForSyDe, one begins by specifying a design in terms of high-level process networks. Mechanisms of synchronization and communication are kept abstract at this level. A series of design transformation maps such a high-level design onto a low-level implementation. These design transformations are (by design) not fully automated; thus ForSyDe may be seen more as a design framework than a language.

The major way in which Hawk, Lava, and ForSyDe all differ from ReWire is that the former are all *embedded* domain-specific languages (EDSL), using Haskell as a host language. In other words, they are best understood not as tools for translating Haskell itself into hardware, but as hardware design languages embedded (implemented) within Haskell. The embedding of domain-specific languages may take one of two forms: *shallow* embedding and *deep* embedding. In a shallow embedding, one creates a small language of higher order combinators whose *implementations* are given as Haskell functions. For example, a monad is an example of a shallowly embedded domain specific language, whose primitive operations are **return** and $\gg=$. By contrast, a deep embedding consists of an abstract syntax for a small special-purpose language, and a set of functions that interpret that syntax in various ways—e.g., by compilation. The fundamental tension between shallow and deep embeddings is one of *observability* versus *expressiveness*. With a shallow embedding, it is often impossible to observe the inner workings of an EDSL program. This is a major problem in the hardware synthesis space [**?**], as we are ultimately concerned with a translation to a particular kind of structure (e.g., netlists). On the other hand, a deep embedding must often forgo the expressiveness of higher-order functional abstraction, and in some cases even type safety is compromised.

ReWire is not an embedded DSL. Instead, it is a compiler that takes Haskell itself as a *source* language. The result is that ReWire supports a full range of functional language features, thus bypassing the problems of expressiveness that are inherent to deep embeddings. At the same time, the fact that the full Haskell source is available to the compiler means that observability is not a problem. The major disadvantage is that the implementation is much more complex. A few other attempts have been made at high-level hardware synthesis directly from Haskell, and these are discussed in Section 2.1.2.

Ultimately, the embedded DSLs described here are all founded on a small set of primitives which are actually rather low level. They achieve abstraction by using Haskell as a means of automatic circuit construction with the primitives of the EDSL forming the foundation. ReWire, by contrast, takes the abstract notion of a reactive resumption as primitive. This has the disadvantage of wresting much of the control over implementation away from the designer, possibly resulting in less efficient implementations. Qualitative comparisons along these lines are an important line of future work where ReWire is concerned. In applications where semantic flexibility is important, however, I believe that the expressiveness and intuitiveness of ReWire's reactive resumption-based model make it worth the trade off.

2.1.2 Compiling Functional Languages to Hardware

The CλaSH project is an effort to produce a compiler from Haskell to hardware. It supports only a subset of the language, but has made quite a bit of progress lately on dealing wih sequential and stateful circuits [**?**]. Its support for sequential circuits is built around the automaton arrow [**?**], and it uses similar techniques to ReWire's partial evaluation to factor out un-hardware-like constructions at compile time. Its implementation techniques are thus quite similar to ReWire. It is not clear from published work on CλaSH, however, whether designs of the complexity exhibited here—in particular, modular monadic designs—can be handled.

Another major effort that is underway is Stephen Edwards' experiments with generating hardware from high-level Haskell specifications [**?**, **?**, **?**]. Edwards employs transformational techniques akin to defunctionalization [**?**] to derive what are essentially specialized state machines implementing particular recursive functions. So far, these techniques have not been fully automated. Unlike ReWire, Edwards does not limit the Haskell source program to a top-level interface typed in a reactive structure, nor indeed to *any* particular type structure. Another distinguishing feature of Edwards' work is that it is capable of representing recursive data structures at runtime. This is achieved by embedding RAM into the implemented circuit, and using an in-memory representation of the recursive structures that is broadly similar to what would be used by a typical Haskell compiler. It might be possible to extend ReWire with similar features, but so far it does not seem wise to take on this extra layer of complexity. A particular challenge here would be reconciling the expected deterministic timing properties of reactive resumption computations with the timing behavior of on-board RAM; this concern seems not to apply to Edwards' work, as he is more interested in compiling general recursive Haskell functions, which do not have predictable timing to begin with.

As applied to processor design, Edwards' approach is closely related to *semantics directed architecture* [**?**] as described by Wand in his classic paper. In semantics directed architecture, the implementation of a machine architecture is derived directly from the denotational semantics of a source language. Wand is more concerned, however, with deriving specialized architectures that are meant to implement abstract machines whose operations are derived from the semantics of a high-level source language. ReWire has broader goals than this, as it seeks to extend monadic abstractions to all kinds of hardware designs, not just to the automatic derivation of specialized instruction sets.

SAFL+ [**?**] represents yet another recent effort to compile a functional language (but not Haskell) to hardware.

2.2 Other Language Paradigms in Hardware Design

Many other language paradigms, including imperative and data flow programming, have been applied to hardware synthesis. Edwards gives a nice survey of the landscape [**?**].

The quintessential examples of conventional hardware description languages are VHDL and Verilog. Such languages are generally focused on abstractions that are built around the notion of a signal over the time domain. A hardware circuit in VHDL is built by connecting components with input and output signals in parallel, though a behavioral style of specification is often used to specify the behavior of individual components. The notion of composition offered by ReWire is more general than this: it is possible, through the use of a handler akin to Harrison's monadic kernels [**?**, **?**] to connect together components according to a variety of communication protocols. While a ReWire program must produce and consume signals *externally*, one is not forced to think in these terms *internally*. In principle, this should bring about a much greater degree of modularity. It should be noted that conventional hardware description languages have been extended to include first-class notions of secure information flow, yielding in particular an experimental language called Caisson [**?**].

Conventional high-level programming languages have been explored as source languages for compilation, but experience suggests that they are ill suited to the diversity of programming models embodied by modern hardware. Edwards has argued [**?**] that the C language, for example, is inextricably joined to the assumptions that underlie classic architecture: flat address spaces, no inherent support for parallelism. Edwards points out that the most successful C-like hardware languages actually tend to borrow only surface syntax features from C, while liberally extending and contracting its semantics, particularly with respect to timing and concurrency. Nevertheless, C-to-VHDL compilers do seem to find substantial use in the reconfigurable computing world. An important conceptual advantage would seem to arise here when dealing with hardware/software co-design. Here the C-to-VHDL paradigm relieves programmers from the cognitive burden of having to switch between languages. In any case, the working hypothesis of the line of work presented in this dissertation (as well as many other active research programs) is that programmer productivity is best enhanced not by forcing hardware design into the imperative language paradigm, but rather by raising the level of abstraction for hardware design. ReWire in particular reflects the belief that *semantics* must drive the design process.

The *synchronous* languages [**?**] such as LUSTRE [**?**] and Esterel are based on the fundamental assumption that a system is reacting to outside events, and that its reactions to those events happen "instantaneously." Computation in such a language may be manifested either as interactions among streams (as in a data flow language like LUSTRE), or with classical imperative constructs (as in Esterel). Lee and Messerschmitt give an overview of the synchronous data flow paradigm [**?**]. Multiparadigm languages that mix the data flow and imperative paradigms have also been proposed [**?**].

The synchronous approach has applications to verification [**?**, **?**], as the notion of interacting streams represents a quite elegant mathematical formalism. One disadvantage of such languages, or at least the data flow languages, is that (much like conventional HDLs) they tend to enforce a certain structure on the inner working of the circuit that may not be appropriate. While any synchronous hardware device clearly can be viewed *externally* as consuming a stream of inputs and producing a stream of outputs, there is no particular reason to assume that this is the best way to understand the interactions of its internal components. The semantic modularity of MMS allows one flexibility in this regard.

2.3 Monads and Modular Monadic Semantics

Monads as a mathematical structure first arose in category theory [?]. Moggi first demonstrated [?] that they represent a useful unifying formalism for notions of computation with effects. In the functional programming world, Wadler [?] has popularized the use of monads as a way of embedding—one might say simulating—effectful computation *inside* the pure language Haskell. Interactions with the outside world are modeled in Haskell via a built-in abstract monad (arguably a pseudomonad, since it has no formal mathematical definition) called the IO monad [?]. This provides a way of isolating effectful parts of the program from pure parts—in essence, once a program gets into the IO monad, it never gets out—but it does not necessarily provide a useful framework for reasoning about these external interactions; for this reason, Simon Peyton Jones has described the IO monad as a sort of "sin bin".

2.3.1 Modular Monadic Semantics

The use of monad transformers in modular semantics originates with Liang [?]. A similar and roughly contemporary approach can be found in Espinosa's Semantic Lego [?]. Harrison showed that modular monadic semantics, in conjunction with partial evaluation, can be used to derive compilers from modular monadic semantics [?]. The general trend of this past research suggests that while monadic semantics makes extensive use of higher order features, program transformation is an effective way of transforming MMS into efficient implementations, e.g., in the area of compiler construction. One of the techniques behind the research described in this dissertation is to push this paradigm even further, down to the level of specification of the hardware itself.

2.3.2 Monads and Security

Monads have been considered as an organizing principle for security reasoning in a number of contexts. As mentioned several times already, Harrison and collaborators have done extensive work on modular monadic semantics as a way of obtaining by-construction security properties such as separation [**?**]. A different approach treats monads essentially as a way of "tagging" computations with information on what security domains they access; this is the approach taken by Crary et al [**?**] and Russo et al [**?**]. In this treatment, the actual type structure of the monad is largely irrelevant; instead, the type system is used extensionally to account for information flow, with the use of a monad simply ensuring that this type system cannot be subverted. A synthesis of these approaches is explored in recent work by Bill Harrison and me [**?**].

2.3.3 Monadic Hardware Semantics

An active project at Cambridge utilizes monads to model relaxed-memory concurrency in modern multicore architectures [**?**]. This project does not, however, aim at synthesizable hardware specifications. Monads are used instead as a basis for *reasoning* about memory access events, with a formalization written in HOL. In the long run, the Cambridge semantics may provide an interesting test case for ReWire, to determine if monadic semantics that are innocent of performance concerns—that is, not originally intended for synthesis—can be used to derive efficient implementations of modern ISAs. In practice I expect that there will be a considerable gap between the reasoning-oriented semantics designed by the Cambridge group and an efficient semantics designed for implementation with ReWire. Measuring and bridging this gap may be a fascinating area of future work.

2.3.4 Related Structures

Comonads [?], the categorical dual of monads, have been used to embed data flow programming into Haskell. It is possible that ReWire could serve as a host for an embedded comonadic data flow DSL, building on the cited work.

Another related structure called *arrows* [?, ?] has seen considerable use in the functional modeling of hardware. The general intuition behind arrows—which are said to generalize monads—is that an arrow is an abstract notion of something that "consumes" an input and "produces" an output. Formally, an arrow is a binary type constructor T with three associated operations:

```
(>>>)  :: T a b -> T b c -> T a c
first  :: T a b -> T (a,c) (b,c)
second :: T a b -> T (c,a) (c,b)
```

The >>> operator is a composition operator that connects the output of one arrow to the input of another, while the *first* and *second* operators allow the construction of paired signals, with one of the signals being filtered through the supplied arrow and the other being passed unchanged. For example, Haskell's function type constructor $(\rightarrow)$ is itself an arrow. An arrow can also be constructed from any monad M, comprising a type constructor mapping types a and b to the type $a \rightarrow M\, b$. This is known as a Kleisli arrow. And as previously discussed, the automata arrow [?] closely resembles a reactive resumption monad. Recent work by Adam Megacz has thoroughly explored this line of work [?], demonstrating that standard Haskell can be used as a host language for metaprogrammatically generating hardware designs [?] with *generalized arrows*. Generalized arrows are a superclass of arrows, expanded to include arrow-like constructions that (unlike "ungeneralized" arrows) cannot subsume arbitrary Haskell functions. This enables a phase distinction to be drawn between the Haskell program that generates an arrow-based program, and the arrow-based program itself; Megacz refers to this as *heterogeneous metaprogramming*.

The appeal of arrows in hardware design is that they directly reflect the kind of structural composition that hardware designers are used to thinking about: hardware is composed from modules that produce outputs, connected to modules that produce inputs, possibly with feedback loops. For this reason it has been claimed [?] that arrows are superior to monads when it comes to hardware construction. I believe that this view holds up only insofar as one assumes that the classical model of hardware design—that hardware should be thought of simply as a set of components running in parallel and communicating over wires—is the right way to think. The present work takes a different view, advocating the recasting of the internal behavior of hardware circuits in terms of semantic, rather than structural, notions. For this, monads are the right abstraction.

Chapter 3

The ReWire Language

This chapter describes the ReWire language, a domain-specific language for modular monadic hardware design. Both syntactically and semantically, the ReWire language is a subset of Haskell, meaning that all ReWire programs are Haskell programs. The ReWire language contains built-in support for an important class of monads (all of which can be emulated in Haskell by means of a library) that enable the construction of verified hardware systems in a semantically modular way. At the same time, the subset of Haskell embodied by ReWire has been carefully selected to ensure synthesizability in hardware. Support for features like recursive functions and data types is, therefore, more limited than that provided by Haskell. A full exploration of the design choices made to enable semantic modularity and ensure synthesizability is given in Section 3.1.

Section 3.2 presents the semantic core of ReWire as a first-order computational λ-calculus [**?**]. Distilling the full language into a compact core simplifies the presentation of ReWire's formal semantics given in Section 3.3. By itself, however, the core language is rather uncomfortable to program with, as it lacks, for example, algebraic data types, polymorphism, and type inference. Thus the actual compiler of Chapter 4 accepts a richer surface language that includes support for (non-recursive) algebraic data types, polymorphism, and type inference, and shares a concrete syntax with (a subset of) Haskell. The nature of this extended language is discussed in more detail in Section 3.4, but we note at the outset that all of the extra features of the full ReWire language can be encoded in terms of the core calculus. Where the distinction between the core calculus and the full language is important, we will refer to them respectively as "ReWire Core" and the "ReWire language". Where the distinction is unimportant, we will refer to both as "ReWire".

3.1 Design of the ReWire Language

Two major criteria inform the design of ReWire. The first of these is the need to support semantically modular hardware design with monads. The second is the necessity that every valid ReWire program be synthesizable to an efficient hardware implementation. These two concerns are explored in Sections 3.1.1 and 3.1.2, respectively. In a nutshell, these considerations taken together mean that ReWire must be "expressive enough without being too expressive". That is, ReWire must be expressive enough to support modular monadic hardware design abstractions, but not so expressive as to require a runtime system burdened with features such as dynamic memory allocation that are impractical in typical hardware designs.

With these criteria in mind, the remainder of this subsection consists of an exploration of the potential design space. Ultimately we converge on the conclusion that (1) the combination of reactive resumptions and layered state monads is sufficient for the kinds of monadic hardware designs we wish to express, and (2) these may be implemented efficiently in hardware. Throughout this discussion the definitions of various type structures, monads, etc., will be expressed in Haskell syntax for purposes of exposition, but it is important to note that the design abstractions we choose will ultimately be taken as primitive in the ReWire language.

The language design that we converge on here will restrict the family of available monads to that formed by the reactive resumption and state monad transformers. This is motivated by a desire to have a simple language as a starting point for research, and the observation that resumptions and state monads are by far the most important ingredients for hardware construction. The basic design, however, can easily be extended to support a broader class of built-in monads. This point will be touched on at the end of Section 3.1.1 and in Chapter 8.

3.1.1 Supporting Semantically Modular Hardware Design

Digital circuit design is commonly divided into two domains: *combinational* circuit design and *sequential* circuit design. In this section we shall divide the problem of choosing design abstractions for ReWire along the same lines. Combinational circuits consist only of asynchronous (unclocked) logic gates that map one or more binary input signals to one or more binary output signals. Sequential circuits, by contrast, *do* exhibit memory, and operate synchronously with a shared clock signal that has the effect of imposing a discrete timeline on the circuit. These sorts of circuits may be implemented in terms of a combination of asynchronous logic gates and synchronous memory elements such as flip flops.

Combinational Logic Represented by Pure Functions

In logic design it is quite common to represent combinational circuits in terms of *truth tables* that map input bits to output bits. The following truth table, for example, describes the behavior of a two-input AND gate:

Inputs		Outputs
a	*b*	*z*
0	0	0
0	1	0
1	0	0
1	1	1

By the same token, combinational circuits can be expressed in a functional/-monadic language as *pure* functions, i.e. functions which do not have any side effects. A binary AND gate, for example, may be expressed in Haskell according to the following definition which directly reflects the truth table:

```
and :: Bit -> Bit -> Bit
and 0 0 = 0
and 0 1 = 0
and 1 0 = 0
and 1 1 = 1
```

Of course, we could also leverage the extra expressiveness of the language to obtain something equivalent but more compact:

```
and :: Bit -> Bit -> Bit
and 0 _ = 0
and 1 b = b
```

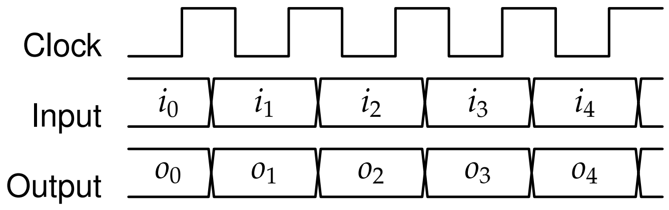
We believe this style is more readily understood by a human reader, especially for more complex logical functions.

Generalizing slightly from this example, the ReWire language adopts the basic approach of implementing any pure function—i.e., any function whose codomain is not typed in a monad—in terms of combinational logic constructs. The alert reader may note that this approach would seem to present a handful of puzzling implementation challenges, such as how to deal with higher-order functions, recursion, and pattern match failures. The basic answer is that these problematic constructs are forbidden; further details are given in Section 3.1.2. Some support for abstract types, however, is afforded by the use of non-recursive data types, which can be encoded as bit vectors. The implementation details of this are given in Chapter 4.

Sequential Logic Represented by Monadic Functions

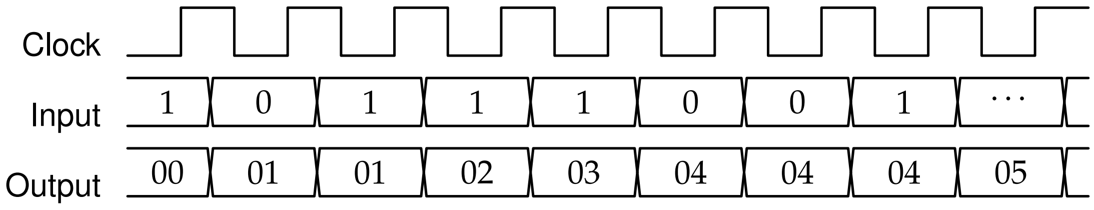
The purely functional nature of combinational circuitry means that simple functions are a sufficient model of combinational logic. The picture for sequential logic, however, is considerably trickier, as we will need to account for functions that may have some memory of past inputs, and whose behavior, i.e., the mapping they they make between inputs from outputs, may change over time.

To begin with, we will note that ReWire in its current design is limited to *single* clock domains. This enables us to treat the problem somewhat more abstractly, while still covering a very large class of realistic circuit designs. A sequential logic circuit can be viewed as sampling a stream of input values $i_0, i_1, \cdots$ of some type I at each tick of a clock signal, and producing a stream of output values $o_0, o_1, \cdots$ of some type O in response, as illustrated by the following timing diagram.



We will assume that the output stream is a *causal* stream, meaning that output o_j is determined fully by the inputs $i_0, i_1, \cdots, i_{j-1}$. In other words, we cannot "look into the future" when producing an output.

As an example of a sequential circuit which we will revisit later in this section, consider an up-counter that takes as its input a stream of bits, and outputs at each clock tick an 8-bit integer indicating how many ones have been received on the input stream, illustrated as follows.



We now face the task of choosing a functional/monadic structure to represent this kind of behavior. We can reject out of hand the possibility of using simple functions of type $I \to O$, for this would preclude any sort of memory of past inputs. As a second attempt, we might consider modeling sequential circuits as functions mapping input *histories* to outputs, i.e., functions of type $I^* \to O$ where I^* represents lists (or strings) of I values. As an abstract mathematical model this does indeed suffice, but it is not immediately clear how to implement directly, as it potentially requires us to store the entire input history for later examination, even as its size grows without bound. Besides this, it does not seem to us like a convenient structure for programming.

A more realistic approach to the memory problem is to use a recursive type like the following.

```
data Seql i o = Seql o (i -> Seql i o)
```

In other words, a sequential circuit from inputs of type *I* to outputs of type *O* consists of a current output value, and a function that maps an input to a "new" sequential circuit; think of this as a continuation. This structure has the benefit that any influence of past input values is encoded in the continuation, giving us some hope (though no inherent guarantee) that we may be able to bound memory usage.

Our up-counter may be modeled in terms of *Seql* as follows (assuming the existence of a numeric *Int8* type of eight-bit integers).

```
upcount :: Seql Bit Int8
upcount = loop 0
  where loop :: Int8 -> Seql Bit Int8
        loop n = Seql n
                   (\ i -> case i of
                             0 -> loop n
                             1 -> loop (n+1))
```

Thus *Seql* seems to be sufficient to express the behavior of sequential hardware circuits. It is also, as it happens, a monad [?].

```
-- Left-to-right Kleisli composition operator
(>=>) :: Monad m =>
  (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \ x -> f x >>= g

instance Monad (Seql i) where
  return x        = Seql x (\ _ -> return x)
  Seql o k >>= f =
    case f o of
      Seql o' _ -> Seql o' (k >=> f)
```

This is not a very *useful* monad for our purposes, however, as the monadic interface provided here does not (and is not intended to) allow us to express computations that take more than one clock cycle. Thus it is not possible to express our up-counter circuit in *Seql* without breaking the monadic abstraction. (It should be noted that *Seql* as implemented in the cited work, where it is referred to as *Moore*, is far more useful when treated as a comonad.)

Let us consider a closely related structure, called a *reactive resumption*. In contrast with *Seql*, a computation of type *React i o a* not only has an input and output channel, but also has the ability to *terminate*, producing a value of type *a*. On a conceptual level we may say that *React* provides output as a side effect, while *Seql* treats the output channel as the computational result.

```
newtype React i o a =
        React (Either a (o,i -> React i o a))

instance Monad (React i o) where
  return x = React (Left x)
  React (Left x) >>= f       = f x
  React (Right (o,k)) >>= f =
           React (Right (o,\ i -> k i >>= f))
```

We will often make use of an additional operation in *React* called *signal*, which in informal operational terms has the effect of sending an output signal, waiting for the next input, and returning that input as its result value.

```
signal :: o -> React i o i
signal o = React (Right (o,return))
```

Equipped with *React*, we can now express our up-counter in monadic style.

```
upcount :: React Bit Int8 ()
upcount = loop 0
  where loop :: Int8 -> React Bit Int8 ()
        loop n = do i <- signal n
                    case i of
                      0 -> loop n
                      1 -> loop (n+1)
```

Semantic Modularity via Monad Transformers

Thus far we have established design abstractions for combinational logic (namely pure functions) and sequential logic (namely the reactive resumption monad). The final piece of the puzzle is to generalize these abstractions in such a way as to enable semantic modularity.

Well-established techniques in the programming languages world [**?**, **?**] center on the use of monad *transformers* to structure monadic specifications in a semantically modular way. If we are to adopt the same approach, we will need to generalize *React* from a monad to a monad transformer. Indeed, the *React* monad generalizes cleanly to a monad transformer [**?**] as follows.

```
newtype ReactT i o m a =
     ReactT (m (Either
                  a
                  (o,i -> ReactT i o m a)))
instance Monad m => Monad (ReactT i o m) where
  return x = ReactT (return (Left x))
  ReactT m >>= f = ReactT (m >>= \ r ->
    case r of
      Left x      -> deReactT (f x)
      Right (o,k) -> return (Right
                (o,\ i -> k i >>= f))
   where deReactT (ReactT m) = m

instance MonadTrans (ReactT i o) where
  lift m = ReactT (m >>= return . Left)

signal :: Monad m => o -> ReactT i o m i
signal o = ReactT (return (Right (o,return)))
```

We may now refactor the definition of our up-counter in a semantically modular style, layering *ReactT* over the state monad transformer *StateT* so as to eliminate the need to explicitly thread the counter value through to the next loop iteration.

```
type M = ReactT Bit Int8 (StateT Int8 Identity)

putctr = lift . put
getctr = lift get

upcount :: M ()
upcount = do putctr 0
             loop
  where loop :: M ()
        loop = do n <- getctr
                  i <- signal n
                  case i of
                    0 -> loop
                    1 -> putctr (n+1) >> loop
```

In implementation terms, however, it is not clear how to account for the *StateT* element, if for no other reason than that the initial value of the state is undefined. One could simply choose an arbitrary initial value for state variables, but this seems rather ad hoc and potentially unsafe.

To deal with the situation in a slightly more parsimonious way, we will supply an operator we call "extrude" that, given a computation in a monad of the form *ReactT i o* (*StateT s m*), allows us to "pull out" one state monad transformer, producing a new computation in the monad *ReactT i o m*. We may then require that the "top-level" definition for a ReWire program is typed in *ReactT i o Identity*. Doing so will require us to supply the initial value for the state.

```
extrude :: Monad m =>
  ReactT i o (StateT s m) a ->
    s -> ReactT i o m (a,s)
extrude (ReactT phi) s =
  ReactT (do (res,s') <- runStateT phi s
             case res of
               Left x      -> return (Left (x,s'))
               Right (o,k) -> return (Right (o,
                       \ i -> extrudeStateT (k i) s')))
```

This allows us to again refactor the code as follows, eliminating the need for the introductory *putctr* call.

```
type M = ReactT Bit Int8 (StateT Int8 Identity)
type R = ReactT Bit Int8 Identity

putctr = lift . put
getctr = lift get

loop :: M ()
loop = do n <- getctr
          i <- signal n
          case i of
            0 -> loop
            1 -> putctr (n+1) >> loop

upcount :: R ((),Int8)
upcount = extrude loop 0
```

The current version of ReWire includes **extrude** as a primitive, largely because implementing it as a non-primitive requires recursion on *ReactT*. However, if in a future version of ReWire we choose to support a broader class, a more general construction allowing extrusion from other monad transformers is possible. In the general case we may think of an extrusion operator as "flipping" *ReactT* on top of the transformer stack with the next-innermost transformer, as reflected in the *MonadExtrude* class below.

```
class MonadTrans t => MonadExtrude t where
  extrudeM :: Monad m => ReactT i o (t m) a
                          -> t (ReactT i o m) a
```

After *extrudeM* is applied, a transformer-specific "run" function can then be applied to dispense with the extruded transformer. The already-defined state monad extrusion operator would then fit into the generalized picture as illustrated by the following imaginary `ghci` session.

```
> :t m
ReactT I O (StateT S Identity) ()
> :t extrudeM m
StateT S (ReactT I O Identity) ()
> :t runStateT (extrudeM m)
S -> ReactT I O Identity ((),S)
```

Many, if not all, useful monad transformers can be a member of this class. For example, the *MaybeT* monad transformer, which allows the possibility of runtime failure to be combined with other effects, could be extruded as follows.

```
newtype MaybeT m a = MaybeT (m (Maybe a))
runMaybeT (MaybeT x) = x

instance Monad m => Monad (MaybeT m) ...
instance MonadTrans MaybeT ...

instance MonadExtrude MaybeT where
  extrudeM (ReactT phi) =
   MaybeT $ ReactT $
    do res <- runMaybeT phi
       case res of
         Nothing            ->
           return (Left Nothing)
         Just (Left x)      ->
           return (Left (Just x))
         Just (Right (o,k)) ->
           return (Right (o,runMaybeT . extrudeM . k))
```

Thus the basic language design advocated here could be generalized, if desired, to a broader class of effects.

Facilitating Security Verification

One of the major motivations of the research presented in this dissertation is to support the construction of formally verified secure hardware. The discussion in this chapter is more concerned with language implementation trade-offs than with formal reasoning concerns, but it is worth noting that the combination of reactive resumption and layered state monads is already known to be a powerful basis for reasoning about the security of concurrent systems [**?**, **?**]. Our use of the same class of monads in ReWire means that the very same techniques are applicable to secure hardware design, further bolstering the argument in favor of our chosen design abstractions. This is explored in greater detail in Chapter 7.

3.1.2 Ensuring Synthesizability

Having ensured that our language is expressive enough for modular monadic hardware design, we now must take care that it is not *too* expressive for implementation in hardware. If this were not a concern, the most obvious course of action would be to take the Haskell definitions given above for *ReactT* and *StateT*, retarget an existing Haskell compiler to FPGA platforms, and use this as our compiler. But Haskell as a source language presents a number of major problems when it comes to implementation in hardware. A Haskell implementation requires a runtime system with a number of features that are undesirable in hardware, and may even be impossible to implement efficiently. (Some of these features, such as dynamic memory allocation, may be useful for certain hardware-based applications; but the much tighter demands made of hardware with respect to timing and resource utilization mean it is essential that control of them be kept in the programmer's hands. A hardware design language that lacks such features is more useful than one that gives the programmer no control over them at all.)

Memory allocation and garbage collection. For tightly integrated, timing-critical embedded systems, dynamic memory allocation (including both stack and heap allocation) and garbage collection present significant problems. First, dynamic memory allocation requires the use of RAM. This is no problem in a software-based runtime system, but in hardware it is often desirable to eschew the use of RAM altogether in favor of statically sized, non-addressable storage elements (i.e., flip flops and registers). Second, dynamic memory allocation raises the possibility that the system may run out of memory. This condition must be detected and handled either by the runtime system or by the user; either way, this imposes a substantial performance overhead on the design. Finally, garbage collection in particular may wreak havoc on timing, though recent research on real-time garbage collection for reconfigurable hardware [**?**] may mitigate this particular concern somewhat.

Divergence, unpredictable timing, and undefinedness. The presence of general recursion in Haskell means that it is possible to write an expression whose evaluation diverges, i.e. runs forever without producing a well-defined value. In general, this is highly undesirable in hardware. While it is true that most hardware systems are designed to run indefinitely, one usually requires that the system produce a well-defined stream of outputs along the way.

Even if we could somehow guarantee that recursive functions will always terminate, the use of recursion is still problematic in terms of timing. Generally we require that a synchronous circuit *always* produces its next output value in time for the next clock tick. Suppose, then, that we have a tail recursive function *fib* that computes the n^{th} Fibonacci number in linear time and constant space, and at each clock tick we are writing *fib*(n) on the output stream, where n varies from clock tick to clock tick over a large range. In general there is no upper bound on how long the evaluation of *fib*(n) will take, meaning that the minimum acceptable clock period for the circuit is ∞; in other words, our circuit may operate no faster than 0Hz. In some cases, we may know *a priori* that the value of n is never greater than some maximum value, say 2^{32}. But this does not help us very much with respect to language implementation, as we are still forced to slow the clock for the entire circuit enough that *fib*(2^{32}) may be calculated between clock ticks, bringing our maximum clock speed above zero, but likely not very far. (A quick benchmark test computing this value via a C program running on a modern desktop CPU as of 2014 suggests that we could expect this value to take at least one minute to compute, implying a maximum clock frequency somewhere in the centihertz range.)

Additionally, there is another kind of undefinedness present in Haskell, arising

Semantic Feature	**Runtime Problems**
Higher-order functions	Dynamic memory allocation Garbage collection
Recursive data structures	Dynamic memory allocation Garbage collection
General recursive functions	Divergence Unpredictable timing Dynamic memory allocation
Incomplete pattern matching	Undefined values

Table 3.1: The semantic antecedents of certain runtime features of Haskell that are problematic in hardware.

from pattern match failure. While undefinedness in this sense presents no inherent problem with respect to timing, it does require us to decide how to represent an undefined value, and how the system must behave when one arises. Hardware designers typically demand control over such decisions.

Identifying the Culprits

The approach we will take with ReWire is to identify the semantic features of Haskell—the "culprits"—that cause undesirable runtime behaviors, and either eliminate these features or restrict them. Table 3.1 identifies a number of problematic semantic features in Haskell, along with the problems they cause with respect to hardware synthesis.

The good news is that these problematic features are *not* essential for our purposes. By eliminating or placing restrictions on them, it is possible to have a language that allows modular monadic hardware design, yet also guarantees synthesizability of reasonably efficient FPGA implementations.

Culprit 1: Higher-Order Functions Higher-order functions are not allowed in the ReWire language. This includes functions that take functions as arguments, functions that return other functions (meaning all curried functions must be fully applied), and functions that take reactive resumptions as arguments. Furthermore, the state and input/output type parameters to monad types may only be instantiated with simple non-recursive data types. For example, types like *ReactT* (*StateT Int Identity* ()) *Int* (), which would represent a synchronous computation that takes a stream of stateful computations as input, are disallowed.

Culprit 2: Recursive Data Structures User-defined recursive data structures, such as lists and trees, are not allowed in the ReWire language. Support for recursive data is limited only to functions that produce values whose codomain is in *ReactT*, and this is subject to the restrictions on function recursion described below.

Culprit 3: General Recursive Functions Recursive functions are only allowed under certain limited circumstances:

- Recursive functions must only take simple data types as arguments, and produce as a result a computation in a well-formed reactive resumption monad. Formally, this means something of the form

$$\begin{aligned}&a_1 \to a_2 \to \cdots \to a_n \to \\ &\quad \mathit{ReactT}\ t_i\ t_o\ (\mathit{StateT}\ s_0\ (\mathit{StateT}\ s_1\ (\cdots\ (\mathit{StateT}\ s_m\ \mathit{Identity}))))\ b\end{aligned}$$

where every a_j, t_i, t_o, every s_k, and b are simple data types.

- Recursive functions must be *productive*. This means that every recursive function must always either terminate with a final value, or produce an output signal, in finite time. This may be ensured by a *guardedness* condition [**?**], a syntactic condition that is sufficient (but not necessary) for productivity.

- All recursive calls must be tail calls.

Culprit 4: Incomplete Pattern Matching All pattern matching in the ReWire language must be exhaustive.

3.2 Syntax of ReWire Core

The design decisions laid out informally in Section 3.1 guarantee that the ReWire language provides enough expressiveness for modular monadic hardware design, while guaranteeing synthesizability. In the remainder of this chapter, we give a more formal definition of the language. We begin with the syntax of the ReWire Core calculus, given in Figure 3.1.

3.2.1 Types

ReWire Core makes a critical but syntactically subtle distinction between "pure" types (simple data) and general types, which include monadic computations. The set of pure types (whose metavariables conventionally are t or any decoration

$$
\begin{array}{rcl}
x, y, f \in \mathit{Variable} & ::= & \mathit{identifiers} \\
t \in \mathit{PureType} & ::= & () \mid t + t' \mid t \times t' \\
B \in \mathit{BaseMonad} & ::= & \mathbb{I} \mid \mathbb{S}_t\ B \\
R \in \mathit{ResMonad} & ::= & \mathbb{R}_{t,t'}\ B \\
M \in \mathit{Monad} & ::= & B \mid R \\
\tau \in \mathit{Type} & ::= & t \mid M(t) \\
\zeta \in \mathit{ResType} & ::= & R(t) \\
e \in \mathit{Expression} & ::= & x\ e_1 \cdots e_n\ \boxed{n \geq 0} \mid \textbf{let}\ x = e\ \textbf{in}\ e'\ \textbf{end} \\
 & \mid & \textbf{nil} \mid \langle e,e' \rangle \mid \textbf{inl}\ e \mid \textbf{inr}\ e \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\
 & \mid & \textbf{case}\ e\ \textbf{of}\ \textbf{inl}\ x \rightarrow e'\ ;\ \textbf{inr}\ y \rightarrow e''\ \textbf{end} \\
 & \mid & \textbf{return}\ e \mid \textbf{bind}\ x \leftarrow e\ \textbf{in}\ e'\ \textbf{end} \mid \textbf{lift}\ e \\
 & \mid & \textbf{get} \mid \textbf{put}\ e \mid \textbf{signal}\ e \mid \textbf{extrude}\ e\ e' \\
l \in \mathit{LetDefn} & ::= & x\ x_1 \cdots x_n = e\ \boxed{n \geq 0} \\
Y \in \mathit{ProgBody} & ::= & \textbf{letfun}\ l\ \textbf{in}\ Y\ \textbf{end} \\
 & \mid & \textbf{letrec}\ l_1\ ; \cdots ;\ l_n\ \textbf{in}\ e\ \textbf{end}\ \boxed{n \geq 0} \\
P \in \mathit{Prog} & ::= & \textbf{program}\ Y\ \textbf{end}
\end{array}
$$

Figure 3.1: Abstract Syntax for ReWire Core Calculus. (Metavariables may appear subscripted or with prime marks.)

thereof) consists only of those types composed of the unit type (), the product constructor $\times$, and the coproduct constructor $+$. The broader set of general types (with metavariables τ, τ', etc.) also contains monadic types, expressing the type of a computation. Note that monads may not be stacked, i.e., one cannot construct the type of a "computation that produces a computation". Three constructors are available for monads: the base identity monad $\mathbb{I}$, the state monad transformer $\mathbb{S}_t$ (where t represents the type of the state, i.e. the `s` in `StateT s`), and the reactive resumption monad transformer $\mathbb{R}_{t,t'}$ where t and t' are (pure) input and output types, respectively. Where a reactive resumption monad transformer is present, there must be only one, and it must be on the top of the monad transformer stack. The subset *ResType* of *Type* is restricted to the types of computations in a resumption monad, i.e. types of the form $\mathbb{R}_{t,t'}\mathbb{S}_{t_1}\mathbb{S}_{t_2}\cdots\mathbb{S}_{t_n}\mathbb{I}(t'')$. Later we will see that recursion is permitted only over types of this form. Arrow types are absent from the syntax altogether, as ReWire Core does not support higher-order functions—functions are not values.

3.2.2 Expressions

Apart from the monadic constructs (in the last two lines of the grammar for *Expression*), most features of the expression language are exactly what one would expect to find in a vanilla functional language. The expression **nil** constructs a value of type (); tuples may be constructed with the form $\langle -, - \rangle$; sum values may be constructed with **inl** and **inr**; and tuples (respectively, sums) may be destructed with the projection operators **fst** and **snd** (respectively, **case**-expressions). Variable reference and function application are slightly nonstandard; these are folded into a single syntactic form merely to emphasize the first-order nature of the language. Note also that λ is absent from the language. As a consequence, **let** permits only the definition of values, not of functions. (Such definitions can always be λ-lifted [**?**] to the top level anyway.)

The basic monad operations take the form of built-in operations **return** and **bind**. Lifting through monad transformers is also built-in via the **lift** operator, as are the monad operations **get** and **put** (for state monads), **signal** (for reactive resumption monads), and **extrude** (for supplying initial state values to state monads underneath resumptions). Fundamentally, the internal structure of the monads is kept abstract, meaning that it is not possible as it is in Haskell to construct a computation by directly applying data constructors like *StateT* and *ReactT*. (As we will see in Section 3.3, though, the semantic structures that underlie ReWire's monads are identical to their Haskell counterparts, modulo the nonexistence of $\perp$ in ReWire Core.)

3.2.3 Programs

Finally, a program in ReWire Core is constructed by wrapping an expression with a single **letrec** defining recursive functions, and a stack of **letfun**s that allows the definition of zero or more nonrecursive functions. All aforementioned restrictions on recursion are encoded not in the syntax, but in the type system.

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{T}[\![()]\!] & = & 1 & \mathcal{M}[\![\mathbb{I}]\!]\,x & = & x \\
\mathcal{T}[\![t+t']\!] & = & \mathcal{T}[\![t]\!] + \mathcal{T}[\![t']\!] & \mathcal{M}[\![\mathbb{S}_t M]\!]\,x & = & (\mathcal{M}[\![M]\!]\,(x \times \mathcal{T}[\![t]\!]))^{\mathcal{T}[\![t]\!]} \\
\mathcal{T}[\![t \times t']\!] & = & \mathcal{T}[\![t]\!] \times \mathcal{T}[\![t']\!] & \mathcal{M}[\![\mathbb{R}_{t,t'} M]\!]\,x & = & \nu X.\mathcal{M}[\![M]\!]\,(x + (\mathcal{T}[\![t']\!] \times X_{\perp}^{\mathcal{T}[\![t]\!]})) \\
\mathcal{T}[\![M(t)]\!] & = & \mathcal{M}[\![M]\!]\,\mathcal{T}[\![t]\!] & & &
\end{array}$$

Figure 3.2: Denotational Semantics of Types

3.3 Semantics of ReWire Core

We now turn our attention to the static and dynamic semantics of ReWire Core, that is its type system and the denotational semantics of programs.

3.3.1 Semantics of Types

The denotational semantics of types is provided in Figure 3.2 on page 64, in terms of two meaning functions $\mathcal{T}[\![-]\!]$ (for base types) and $\mathcal{M}[\![-]\!]$ (for monads). Types are interpreted as complete partial orders, but with the exception of the reactive resumption monad all domains are discretely ordered. Each monad type constructor is interpreted as a mapping that takes one domain (i.e., CPO) as its argument and gives another domain as a result. (In categorical terms this is simply the object map corresponding to the underlying functor of a monad in **CPO**.) Note that while the reactive resumption monad is interpreted as a pointed CPO, the guardedness criterion (discussed in Section 3.3.2) guarantees that the bottom value will never actually be constructed by a program; it is needed only so we may make use of a fixpoint operator in defining the semantics of **letrec**, the bind operation for $\mathbb{R}$, and **extrude**.

3.3.2 Type System

Figure 3.3 on page 67 contains the typing rules for ReWire Core. Typing judgments have the form $\Gamma \vdash \theta : \tau$, where Γ is a set of assumptions about the types of bound variables, τ is a type, and θ is either an expression, a program body, or a program (i.e., $\theta ::= e \ \ | Y \mid P$). Typing assumptions take the form $x : t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow \tau$, where $n \geq 0$. Thus even though functions are not "first-class citizens" and arrow "types" are not actually part of the syntax of types, the typing environment may contain assumptions about functions. In understanding the type system, it is important to bear in mind the distinction between the metavariables τ, ζ, and t. The first of these includes all syntactically valid types, both data types and computational types; the second includes *only* computational types in a reactive resumption monad; and the third contains only data types. The distinction among these three categories is the mechanism by which computations-as-values are kept out of the language, and recursion is restricted only to resumption-monadic computations.

Guardedness and Tail Recursiveness

The typing rule T-Prog requires judgments that the program being typed be guarded and tail recursive. The exact rules for these judgments are formalized in Figures 3.4 and 3.5. Guardedness judgments take the form $\Sigma \vdash \mathcal{G}(\theta)$, where Σ is a set containing the names of recursively defined functions currently in scope and θ is either a program body or an expression. The key rule is G-Bind2, which, in the presence of **signal** on the left-hand side of a **bind**, does not demand further scrutiny of the right-hand side expression. Tail recursiveness judgments take the form $\langle \Pi, \Sigma \rangle \vdash \mathcal{S}(\theta)$, where Σ and θ play the same role as in guardedness judgments, and $\Pi \in \{\bot, \top\}$ indicates whether a recursive call made in the current context would ($\top$) or would not ($\bot$) be a tail call. (Thus Π is forced to $\bot$ when making judgments about, for example, the subexpressions of a function application, but Π is not forced to $\bot$ when making judgments about, for example, the body of a **let** expression.) The fact that recursion is restricted only to reactive resumption monads simplifies both guardedness and tail recursiveness judgments significantly compared to what would be required in a general recursive language. For example, it is not necessary to inspect function argument expressions, as these expressions are, by construction, restricted to simple base types, and therefore their values cannot be constructed recursively. We note in passing that the rules both for guardedness and tail recursiveness might be simplified if they were redefined in a type-directed fashion (perhaps combining typing judgments, guardedness judgments, and tail-recursiveness judgments into a single inductively-defined predicate).

3.3.3 Semantics of Programs

Finally, we turn our attention to the semantics of programs exhibited in Figure 3.6 on page 71. The meaning function $\mathcal{E}[\![-]\!]$ is technically defined by induction on the structure of typing *derivations*. For the sake of readability, however, the presentation of Figure 3.6 is given merely in terms of expressions (or program bodies/programs), whose type will be mentioned explicitly only when immediately relevant. Thus $\mathcal{E}[\![\theta : \tau]\!]$ takes a expression, program body, or program e of type τ, an environment mapping variables to their meanings, and returns an element of the semantic domain $\mathcal{T}[\![\tau]\!]$. For **bind**, **return**, and **lift** we refer to the functions η_{-}, $\star_{-}$, $\mathcal{L}_{-}$

$$\frac{\Gamma \vdash e_i : t_i \text{ for every } i \in [1,n] \qquad x : t_1 \to \cdots \to t_n \to \tau \in \Gamma}{\Gamma \vdash x\, e_1\ \cdots e_n : \tau}\ \textsc{T-VarApp}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma, x : t \vdash e' : \tau}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e'\ \mathbf{end} : \tau}\ \textsc{T-Let}$$

$$\frac{}{\Gamma \vdash \mathbf{nil} : ()}\ \textsc{T-Nil} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash \langle e, e' \rangle : t \times t'}\ \textsc{T-Pair}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{inl}\ e : t + t'}\ \textsc{T-InL} \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash \mathbf{inr}\ e : t + t'}\ \textsc{T-InR}$$

$$\frac{\Gamma \vdash e : t \times t'}{\Gamma \vdash \mathbf{fst}\ e : t}\ \textsc{T-Fst} \qquad \frac{\Gamma \vdash e : t \times t'}{\Gamma \vdash \mathbf{snd}\ e : t'}\ \textsc{T-Snd}$$

$$\frac{\Gamma \vdash e : t + t' \quad \Gamma, x : t \vdash e' : \tau \quad \Gamma, y : t' \vdash e'' : \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x \to e'\ ; \mathbf{inr}\ y \to e''\ \mathbf{end} : \tau}\ \textsc{T-Case}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{return}\ e : M(t)}\ \textsc{T-Ret} \qquad \frac{\Gamma \vdash e : M(t) \quad \Gamma, x : t \vdash e' : M(t')}{\Gamma \vdash \mathbf{bind}\ x \leftarrow e\ \mathbf{in}\ e' : M(t')}\ \textsc{T-Bind}$$

$$\frac{\Gamma \vdash e : B(t)}{\Gamma \vdash \mathbf{lift}\ e : \mathbb{R}_{t':t''}B(t)}\ \textsc{T-LiftR} \qquad \frac{\Gamma \vdash e : B(t)}{\Gamma \vdash \mathbf{lift}\ e : \mathbb{S}_{t'}B(t)}\ \textsc{T-LiftS}$$

$$\frac{}{\Gamma \vdash \mathbf{get} : \mathbb{S}_t B(t)}\ \textsc{T-Get} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathbf{put}\ e : \mathbb{S}_t B(())}\ \textsc{T-Put}$$

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash \mathbf{signal}\ e\ :\ \mathbb{R}_{t:t'}B(t)}\ \textsc{T-Signal} \qquad \frac{\Gamma \vdash e : \mathbb{R}_{t:t'}\mathbb{S}_{t''}B(t''') \quad \Gamma \vdash e' : t''}{\Gamma \vdash \mathbf{extrude}\ e\ e' : \mathbb{R}_{t:t'}B(t''' \times t'')}\ \textsc{T-Extrude}$$

$$\frac{\Gamma, x_1 : t_1, \cdots, x_n : t_n \vdash e : \tau \quad \Gamma, f : t_1 \to \cdots \to t_n \to t \vdash Y : \zeta}{\Gamma \vdash \mathbf{letfun}\ f\ (x_1 : t_1) \cdots (x_n : t_n) = e\ \mathbf{in}\ Y\ \mathbf{end} : \zeta}\ \textsc{T-LetFun}$$

$$\frac{\begin{array}{c} l_i \text{ is } f_i\, x^i_1 \cdots x^i_{k_i} = e_i \text{ for every } i \in [1,n] \\ \Gamma' = \Gamma, f_1 : t^1_1 \to \cdots \to t^1_{k_1} \to \zeta^1, \ldots, f_n : t^n_1 \to \cdots \to t^n_{k_n} \to \zeta^n \\ \Gamma', x^i_1 : t^i_1, \ldots, x^i_{k_i} : t^i_{k_i} \vdash e_i : \zeta_i \text{ for every } i \in [1,n] \\ \Gamma' \vdash e : \zeta \end{array}}{\Gamma \vdash \mathbf{letrec}\ l_1\ ;\ l_2\ ;\ \cdots\ ;\ l_n\ \mathbf{in}\ e\ \mathbf{end} : \zeta}\ \textsc{T-LetRec}$$

$$\frac{\{\} \vdash Y : \zeta \quad \{\} \vdash \mathcal{G}(Y) \quad \langle \top, \{\} \rangle \vdash \mathcal{S}(Y)}{\{\} \vdash \mathbf{program}\ Y\ \mathbf{end} : \zeta}\ \textsc{T-Prog}$$

Figure 3.3: Type System for ReWire Core Calculus

$$\frac{x \notin \Sigma}{\Sigma \vdash \mathcal{G}\,(x\; e_1\; \cdots e_n)}\ \text{G-VarApp}$$

$$\frac{\Sigma \setminus \{x\} \vdash \mathcal{G}\,(e')}{\Sigma \vdash \mathcal{G}\,(\textbf{let}\; x = e\; \textbf{in}\; e'\; \textbf{end})}\ \text{G-Let}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{nil})}\ \text{G-Nil} \qquad \frac{}{\Sigma \vdash \mathcal{G}\,(\langle e, e'\rangle)}\ \text{G-Pair}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{inl}\; e)}\ \text{G-InL} \qquad \frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{inr}\; e)}\ \text{G-InR}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{fst}\; e)}\ \text{G-Fst} \qquad \frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{snd}\; e)}\ \text{G-Snd}$$

$$\frac{\Sigma \setminus \{x\} \vdash \mathcal{G}\,(e) \qquad \Sigma \setminus \{y\} \vdash \mathcal{G}\,(e'')}{\Sigma \vdash \mathcal{G}\,(\textbf{case}\; e\; \textbf{of}\; \textbf{inl}\; x \rightarrow e'\; ;\textbf{inr}\; y \rightarrow e''\; \textbf{end})}\ \text{G-Case}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{return}\; e)}\ \text{G-Ret} \qquad \frac{\Sigma \vdash \mathcal{G}\,(e)}{\Sigma \vdash \mathcal{G}\,(\textbf{lift}\; e)}\ \text{G-Lift}$$

$$\frac{\Sigma \vdash \mathcal{G}\,(e) \qquad \Sigma \setminus \{x\} \vdash \mathcal{G}\,(e')}{\Sigma \vdash \mathcal{G}\,(\textbf{bind}\; x \leftarrow e\; \textbf{in}\; e')}\ \text{G-Bind1} \qquad \frac{\Sigma \vdash \mathcal{G}\,(e)}{\Sigma \vdash \mathcal{G}\,(\textbf{bind}\; x \leftarrow \textbf{signal}\; e\; \textbf{in}\; e')}\ \text{G-Bind2}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{get})}\ \text{G-Get} \qquad \frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{put}\; e)}\ \text{G-Put}$$

$$\frac{}{\Sigma \vdash \mathcal{G}\,(\textbf{signal}\; e)}\ \text{G-Signal} \qquad \frac{\Sigma \vdash \mathcal{G}\,(e)}{\Sigma \vdash \mathcal{G}\,(\textbf{extrude}\; e\; e')}\ \text{G-Extrude}$$

$$\frac{\Sigma \setminus \{f\} \vdash \mathcal{G}\,(Y)}{\Sigma \vdash \mathcal{G}\,(\textbf{letfun}\; f\; x_1 \cdots x_n = e\; \textbf{in}\; Y\; \textbf{end})}\ \text{G-LetFun}$$

$$\frac{\begin{array}{c} l_i \text{ is } f_i\; x^i_1 \cdots x^i_{k_i} = e_i \text{ for every } i \in [1, n] \\ \Sigma' = \Sigma \cup \{f_1, \cdots, f_n\} \\ \Sigma' \setminus \{x^i_1, \ldots, x^i_{k_i}\} \vdash \mathcal{G}\,(e_i) \text{ for every } i \in [1, n] \end{array}}{\Sigma \vdash \mathcal{G}\,(\textbf{letrec}\; l_1\; ;\; l_2\; ;\; \cdots\; ;\; l_n\; \textbf{in}\; e\; \textbf{end})}\ \text{G-LetRec}$$

Figure 3.4: Guardedness Condition for ReWire Core Calculus

$$\frac{x \notin \Sigma}{\langle \bot, \Sigma \rangle \vdash \mathcal{S}\,(x\; e_1\; \cdots e_n)}\ \textsc{S-VarApp1}$$

$$\frac{}{\langle \top, \Sigma \rangle \vdash \mathcal{S}\,(x\; e_1\; \cdots e_n)}\ \textsc{S-VarApp2}$$

$$\frac{\langle \Pi, \Sigma \setminus \{x\} \rangle \vdash \mathcal{S}\,(e')}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{let}\ x = e\ \textbf{in}\ e'\ \textbf{end})}\ \textsc{S-Let}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{nil})}\ \textsc{S-Nil} \qquad \frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\langle e, e' \rangle)}\ \textsc{S-Pair}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{inl}\ e)}\ \textsc{S-InL} \qquad \frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{inr}\ e)}\ \textsc{S-InR}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{fst}\ e)}\ \textsc{S-Fst} \qquad \frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{snd}\ e)}\ \textsc{S-Snd}$$

$$\frac{\langle \Pi, \Sigma \setminus \{x\} \rangle \vdash \mathcal{S}\,(e) \quad \langle \Pi, \Sigma \setminus \{y\} \rangle \vdash \mathcal{S}\,(e'')}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{case}\ e\ \textbf{of inl}\ x \rightarrow e'\ ; \textbf{inr}\ y \rightarrow e''\ \textbf{end})}\ \textsc{S-Case}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{return}\ e)}\ \textsc{S-Ret} \qquad \frac{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(e)}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{lift}\ e)}\ \textsc{S-Lift}$$

$$\frac{\langle \bot, \Sigma \rangle \vdash \mathcal{S}\,(e) \quad \langle \Pi, \Sigma \setminus \{x\} \rangle \vdash \mathcal{S}\,(e')}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{bind}\ x \leftarrow e\ \textbf{in}\ e')}\ \textsc{S-Bind}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{get})}\ \textsc{S-Get} \qquad \frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{put}\ e)}\ \textsc{S-Put}$$

$$\frac{}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{signal}\ e)}\ \textsc{S-Signal} \qquad \frac{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(e)}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{extrude}\ e\ e')}\ \textsc{S-Extrude}$$

$$\frac{\langle \Pi, \Sigma \setminus \{f\} \rangle \vdash \mathcal{S}\,(Y)}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{letfun}\ f\ x_1 \cdots x_n = e\ \textbf{in}\ Y\ \textbf{end})}\ \textsc{S-LetFun}$$

$$\frac{\begin{array}{c} l_i \text{ is } f_i\ x^i_1 \cdots x^i_{k_i} = e_i \text{ for every } i \in [1, n] \\ \Sigma' = \Sigma \cup \{f_1, \cdots, f_n\} \\ \langle \Pi, \Sigma' \setminus \{x^i_1, \ldots, x^i_{k_i}\} \rangle \vdash \mathcal{S}\,(e_i) \text{ for every } i \in [1, n] \\ \langle \Pi, \Sigma' \rangle \vdash \mathcal{S}\,(e) \end{array}}{\langle \Pi, \Sigma \rangle \vdash \mathcal{S}\,(\textbf{letrec}\ l_1\ ;\ l_2\ ;\ \cdots\ ;\ l_n\ \textbf{in}\ e\ \textbf{end})}\ \textsc{S-LetRec}$$

Figure 3.5: Tail Recursiveness Predicate for ReWire Core Calculus

of Figure 3.8 on page 73 which respectively define the semantics of the bind, return/unit, and lift operations of a given monad. These semantics are equivalent to those given by Liang [**?**] (for the state monad transformer) and Papaspyrou [**?**] (for the reactive resumption monad transformer). On a minor technical note, $\mathcal{L}_{-}$, which defines the lift operations for monad transformers, is technically not a total function, as it is undefined when applied to $\mathbb{I}$. The structure of the type system, however, ensures that $\mathcal{L}_{\mathbb{I}}$ is never actually "used" by the semantics; see the rule T-Lift in Figure 3.3 and the equation for **lift** in Figure 3.6.

3.4 Extended Language Constructs

The ReWire Core calculus features a syntax that is appealingly compact from a semantic point of view. Its usefulness as a surface-level programming language, however, is greatly enhanced by the addition of a number of convenience extensions, forming what we call the ReWire language. Several such extensions, most of which are already supported by the compiler of Chapter 4, will be described informally in this section. It is important to note that none of the language extensions described here have major semantic implications; they may all be encoded in terms of the ReWire Core calculus.

3.4.1 Haskell Concrete Syntax

The ultimate goal is to support Haskell concrete syntax for the entirety of ReWire's feature set. A few examples of this are given below.

$$
\begin{array}{rcl}
\mathcal{E}\,[\![x\, e_1\ \cdots\ e_n]\!]\,\rho &=& (\rho x)(\mathcal{E}\,[\![e_1]\!]\,\rho)\cdots(\mathcal{E}\,[\![e_n]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{let}\ x = e\ \mathbf{in}\ e'\ \mathbf{end}]\!]\,\rho &=& \mathcal{E}\,[\![e']\!]\,(\rho[x \mapsto \mathcal{E}\,[\![e]\!]\,\rho]) \\
\mathcal{E}\,[\![\mathbf{nil}]\!]\,\rho &=& () \\
\mathcal{E}\,[\![\langle e, e'\rangle]\!]\,\rho &=& \langle \mathcal{E}\,[\![e]\!]\,\rho, \mathcal{E}\,[\![e']\!]\,\rho\rangle \\
\mathcal{E}\,[\![\mathbf{fst}\ e]\!]\,\rho &=& \pi_1(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{snd}\ e]\!]\,\rho &=& \pi_2(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{inl}\ e]\!]\,\rho &=& \iota_1(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{inr}\ e]\!]\,\rho &=& \iota_2(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x \to e'\ ;\mathbf{inr}\ y \to e''\ \mathbf{end}]\!]\,\rho &=& \begin{cases} \mathcal{E}\,[\![e']\!]\,(\rho[x \mapsto v]) \\ \quad \text{if } \mathcal{E}\,[\![e]\!]\,\rho = \iota_1\ v \\ \mathcal{E}\,[\![e'']\!]\,(\rho[y \mapsto v]) \\ \quad \text{if } \mathcal{E}\,[\![e]\!]\,\rho = \iota_2\ v \end{cases} \\
\mathcal{E}\,[\![\mathbf{bind}\ x \leftarrow e\ \mathbf{in}\ e' : M(t')]\!]\,\rho &=& \mathcal{E}\,[\![e]\!]\,\rho \star_M \lambda v.\mathcal{E}\,[\![e']\!]\,(\rho[x \mapsto v]) \\
\mathcal{E}\,[\![\mathbf{return}\ e : M(t)]\!]\,\rho &=& \eta_M(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{lift}\ e : TM(t)]\!]\,\rho &=& \mathcal{L}_{TM}(\mathcal{E}\,[\![e]\!]\,\rho) \\
\mathcal{E}\,[\![\mathbf{get} : \mathbb{S}_t M(t)]\!]\,\rho &=& \lambda\sigma.\eta_M\langle\sigma,\sigma\rangle \\
\mathcal{E}\,[\![\mathbf{put}\ e : \mathbb{S}_t M(())]\!]\,\rho &=& \lambda\sigma.\eta_M\langle(), \mathcal{E}\,[\![e]\!]\,\rho\rangle \\
\mathcal{E}\,[\![\mathbf{signal}\ e\ :\ \mathbb{R}_{t:t'} M(t)]\!]\,\rho &=& \eta_M(\iota_2\langle \mathcal{E}\,[\![e]\!]\,\rho, \lambda i.\eta_M(\iota_1\ i)\rangle) \\
\mathcal{E}\,[\![\mathbf{extrude}\ e\ e' : \mathbb{R}_{t:t'} M(t''' \times t'')]\!]\,\rho &=& \chi_M(\mathcal{E}\,[\![e]\!]\,\rho)(\mathcal{E}\,[\![e']\!]\,\rho) \qquad (\dagger) \\
\mathcal{E}\,[\![\mathbf{letfun}\ \cdots]\!]\,\rho &=& \textit{See Figure 3.7} \\
\mathcal{E}\,[\![\mathbf{letrec}\ \cdots]\!]\,\rho &=& \textit{See Figure 3.7} \\
\mathcal{E}\,[\![\mathbf{program}\ Y\ \mathbf{end}]\!]\,\rho &=& \mathcal{E}\,[\![Y]\!]\,\rho
\end{array}
$$

(†) In the equation for **extrude**,

$$
\begin{array}{rcl}
\chi_M &=& \mathbf{fix}\ F.\lambda\varphi.\lambda\sigma. \\
&& \qquad \varphi\sigma \star_M \lambda p.\begin{cases} \eta_M(\iota_1\langle x, \sigma'\rangle) & \text{if } p = \langle \iota_1\ x, \sigma'\rangle \\ \eta_M(\iota_2\langle o, \lambda i.F(\kappa i)\sigma'\rangle) & \text{if } p = \langle \iota_2\langle o, \kappa\rangle, \sigma'\rangle \end{cases}
\end{array}
$$

Figure 3.6: Denotational Semantics of Expressions (continued in Figure 3.7)

$$\mathcal{E}\,[\![\textbf{letfun}\ f\ (x^1 : t^1)\cdots(x^k : t^k) = e\ \textbf{in}\ Y\ \textbf{end}]\!]\ \rho = \mathcal{E}\,[\![Y]\!]\ \rho'$$

where

$$\begin{array}{rcl} g & = & \lambda y^1.\cdots\lambda y^k.\mathcal{E}\,[\![e]\!]\,(\rho[x^1 \mapsto y^1, \cdots, x^k \mapsto y^k]) \\ \rho' & = & \rho[f \mapsto g] \end{array}$$

$$\mathcal{E}\left[\!\!\left[\begin{array}{l} \textbf{letrec}\ \ f_1\ (x_1^1 : t_1^1)\cdots(x_1^{k_1} : t_1^{k_1}) = e_1 \\ \qquad\qquad\vdots \\ \qquad\quad\ f_n\ (x_n^1 : t_n^1)\cdots(x_n^{k_n} : t_n^{k_n}) = e_n \\ \textbf{in}\ e\ \textbf{end} \end{array}\right]\!\!\right]\rho = \mathcal{E}\,[\![e]\!]\ \rho''$$

where

$$\begin{array}{rcl} \langle g_1, \cdots, g_n \rangle & = & \textbf{fix}\ \langle F_1, \cdots, F_n \rangle. \\ & & \quad \textbf{let}\ \rho' = \rho[f_1 \mapsto F_1, \cdots, f_n \mapsto F_n] \\ & & \quad \textbf{in}\ \langle \lambda y_1^1.\cdots\lambda y_1^{k_1}.\mathcal{E}\,[\![e_1]\!]\,(\rho'[x_1^1 \mapsto y_1^1, \cdots, x_1^{k_1} \mapsto y_1^{k_1}]), \\ & & \qquad\quad \vdots \\ & & \qquad \lambda y_n^1.\cdots\lambda y_n^{k_n}.\mathcal{E}\,[\![e_n]\!]\,(\rho'[x_n^1 \mapsto y_n^1, \cdots, x_n^{k_n} \mapsto y_n^{k_n}])\rangle \\ \rho'' & = & \rho[f_1 \mapsto g_1, \cdots, f_n \mapsto g_n] \end{array}$$

Figure 3.7: Denotational Semantics of Expressions (continued from Figure 3.6)

Bind operators:

$$
\begin{array}{rcl}
\varphi \star_{\mathbb{I}} f & = & f\varphi \\
\varphi \star_{\mathbb{S}_t M} f & = & \lambda\sigma.\varphi\sigma \star_M \lambda p.f(\pi_1 p)(\pi_2 p) \\
\varphi_0 \star_{\mathbb{R}_{t:t'} M} f_0 & = & g\ \varphi_0\ f_0 \\
 & & \text{where } g \quad = \mathbf{fix}\ F.\lambda\varphi.\lambda f. \\
 & & \qquad \varphi \star_M \lambda r. \begin{cases} f x \\ \qquad \text{if } r = \iota_1\ x \\ \eta_M(\iota_2\langle o, \lambda i.F\ (\kappa\ i)\ f\rangle) \\ \qquad \text{if } r = \iota_2\langle o, \kappa\rangle \end{cases}
\end{array}
$$

Unit operators:

$$
\begin{array}{rcl}
\eta_{\mathbb{I}}(x) & = & x \\
\eta_{\mathbb{S}_t M}(x) & = & \lambda\sigma.\eta_M(\langle x, \sigma\rangle) \\
\eta_{\mathbb{R}_{t:t'} M}(x) & = & \eta_M(\iota_1\ x)
\end{array}
$$

Lift operators:

$$
\begin{array}{rcl}
\mathcal{L}_{\mathbb{S}_t M}(\varphi) & = & \lambda\sigma.\varphi \star_M \lambda x.\eta_M(\langle x, \sigma\rangle) \\
\mathcal{L}_{\mathbb{R}_{t:t'} M}(\varphi) & = & \varphi \star_M \lambda x.\eta_M(\iota_1\ x)
\end{array}
$$

Figure 3.8: Denotational Semantics of Monads

Layout Rule

Haskell's concrete syntax features block structuring based on indentation, the correct parsing of which is notoriously tricky to get right [**?**]. At the moment, the layout rule is not implemented by the ReWire language, requiring explicit block structuring on the part of the user, but it will be implemented in a future release.

Module Structure

In Haskell, programs are structured not as an explicitly nested set of **let**/**letrec** expressions, but as a series of top-level function definitions that form a single recursive binding group. The ReWire language adopts the same concrete syntax. Decoding this to ReWire Core necessitates a simple static check that analyzes which functions are being defined recursively and which are not, and rearranging the declarations as a set of nested **letfun**s wrapped around a single **letrec**.

Where-Clauses

Haskell allows locally scoped function definitions via a syntactic form called a **where** clause. For example, in the following code fragment for computing the n^{th} element of the Fibonacci sequence, the inner definition of `fibs` is not visible outside the scope of `fib`.

```
fib :: Int -> Int
fib n = fibs!!n
  where fibs :: [Int]
        fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

(Note that this code is not valid ReWire due to its use of recursion outside *ReactT*, but it still illustrates the salient point.) A transformation akin to λ-lifting [**?**] can dispense with where clauses, by promoting **where**-bound variables to top-level definitions. This transformation is slated for implementation in the ReWire compiler in the near future.

Pattern Binding

It is often convenient when programming in Haskell to assign values to multiple variables at a time by deconstructing a data value. The syntax for this is called *pattern binding*. For example, the standard Haskell Prelude defines a function *quotRem* that returns both the quotient and the remainder result from dividing two integers.

```
quotRem :: Int -> Int -> (Int,Int)
```

If we wish to break out the resulting values into separate named variables, using pattern binding in a **let** expression or **where** clause is often convenient.

```
let
   (q,r) = quotRem x y
in
   ...
```

This pattern is also useful in monadic programming. Suppose the input to our reactive circuit is a tuple containing (say) an address and a write-enable line. We can break out the individual elements of the tuple by applying pattern binding to the return value of **signal**.

```
foo :: ReactT (Bit,Addr) Word Identity ()
foo = do (we,a) <- signal bar
         ...
```

Pattern binding can be desugared to pattern matching, the implementation of which is described below. Note that as always, ReWire (unlike Haskell) will require pattern matching to be exhaustive; this precludes the use of pattern binding on multiple-constructor data types, as in the following example.

```
f :: Int -> Maybe Int
f x = ...

g :: Int
g = let (Just x) = f 0 in x     -- NOT legal in ReWire
```

On the other hand, pattern binding in defining equations *is* permissible, as long as all cases are handled.

```
h :: Maybe Int -> Int
h (Just x) = x          -- Both cases are handled,
h Nothing  = 0          -- so no problem for ReWire.
```

Pattern binding is slated for implementation in the ReWire compiler in the near future, and will be implemented via desugaring to **case** expressions.

Infix Operators

Haskell allows programmers to define custom infix operators, which often results in more visually appealing code. Support for this is not yet implemented in ReWire, but it will require only a minor overhaul to the parser.

`do`-Notation

Since monads are such a pervasive programming structure in Haskell, it, like ReWire Core, provides built-in support for imperative-style monadic programming via a construct called `do`-notation. Here the syntactic differences between ReWire Core and Haskell are quite shallow; a Haskell expression of the form

```
do x <- e
   y <- e'
   e''
   return (f x y)
```

can be translated to ReWire core as

$$\begin{array}{ll} & \textbf{bind}\ x \leftarrow e \\ \textbf{in} & \textbf{bind}\ y \leftarrow e' \\ \textbf{in} & \textbf{bind}\ ? \leftarrow e'' \\ \textbf{in} & \textbf{return}\ (f\ x\ y) \end{array}$$

where ? is a freshly generated variable not occurring free in the expression.

3.4.2 Algebraic Data Types

Simple product and coproduct data types are clearly a bit unwieldy from a programmer's point of view. For this reason the ReWire language includes support for nonrecursive algebraic data types.

To see how to encode this in ReWire Core let us first consider the type of bits.

```
data Bit = Zero | One
```

This type is, up to isomorphism, just the sum of two unit types. Therefore it may be translated into ReWire Core's type system as exactly that.

$$Bit \triangleq () + ()$$

Data types with fields, on the other hand, can be encoded as products. So the type of four-bit vectors

```
data W4 = W4 Bit Bit Bit Bit
```

becomes

$$\begin{array}{rcl} W4 & \triangleq & Bit \times Bit \times Bit \times Bit \\ & = & (() + ()) \times (() + ()) \times (() + ()) \times (() + ()) \end{array}$$

Semantically speaking, this sort of encoding will suffice for arbitrarily complex non-recursive data types. The actual compiler treats named data constructors as built in to the language rather than actually applying this transformation, mostly for the sake of producing readable error messages.

When pattern matching on algebraic data types in the ReWire language, all possible cases must be handled or the compiler will terminate with an error message, unlike Haskell. This avoids the introduction of an “undefined” value into the language.

3.4.3 Polymorphism and Type Classes

Haskell allows the definition of parametrically polymorphic functions and data types. *Maybe* is a prime example of a polymorphic data type.

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

Here the type variable a stands for any type, meaning that the data declaration actually defines several (indeed infinitely many) types: $Maybe\ Int$, $Maybe\ Bit$, $Maybe\ (Maybe\ (Bit, Bit, Int))$, and so on.

Function definitions, too, may be polymorphic. For example, a function that flips the elements of a pair may be defined as follows.

```
flipPair :: (a,b) -> (b,a)
flipPair (x,y) = (y,x)
```

Again, this function has infinitely many types: $(Bit, Bit) \to (Bit, Bit)$, $(Bit, Int) \to (Int, Bit)$, $(Maybe\ Bit, Bit) \to (Bit, Maybe\ Bit)$, and so on.

In Haskell, compiling parametrically polymorphic functions is a relatively simple matter since data structures are (basically) represented internally as a nest of pointers. In ReWire, however, this approach presents a bit of a problem; later in Chapter 4 we will define a compilation scheme that stores data types in a *fixed*-size bit vector. Pattern matching and the construction of data values, then, requires that we know *a priori* the exact size of the data that we are operating on, and therefore its exact type.

Since separate compilation is not really a concern for ReWire, however, a simple workaround is available: we will insist that every use of a polymorphic function has a uniquely determined type. For example, if we apply the function *flipPair* to the expression $(0, 1)$ of type (Bit, Bit), we know that the function we want is actually *flipPair* with both of its type variables instantiated to *Bit*. At compile time we will produce a separate implementation of *flipPair* for each such case. Due to the lack of polymorphic recursion in ReWire, the number of such implementations that we will require is guaranteed to be finite.

Support for polymorphism in the manner described here is already implemented in the ReWire compiler; again, the lack of polymorphic recursion means that the typical dictionary-passing implementation of type classes may be transformed away at compile time [**?**]. In the future, Haskell-style type classes may be implemented according to a similar scheme, allowing the overloading of certain functions. At the moment, however, only a limited and ad-hoc form of type class-style overloading is available, sufficient to support functions that may operate in any monad.

Chapter 4

The ReWire Compiler

This chapter comprises a detailed discussion of the design of the ReWire Compiler, which translates programs in the ReWire language into VHDL programs suitable for synthesis and implementation on FPGAs. A high-level outline of the compilation process is provided in Figure 4.1. In the first of three broad phases (tagged (a) in the figure), the compiler parses and type-checks a ReWire program expressed in Haskell concrete syntax (as discussed previously in Section 3.4). The structure of the front end is mostly standard and reuses an existing Haskell front end implementation, so the discussion of Section 4.1 is limited to broad strokes. The heart of the compiler is the second phase (Figure 4.1b), where the compiler converts a type-annotated ReWire program into a simple intermediate language called Pre-HDL, which is then transformed into a normal form more amenable to translation to VHDL. The use of a simple intermediate language allows various intermediate compiler phases to be implemented without taking into account the full syntax, semantics, and feature set of VHDL. Phase (b) of the compiler is discussed in its

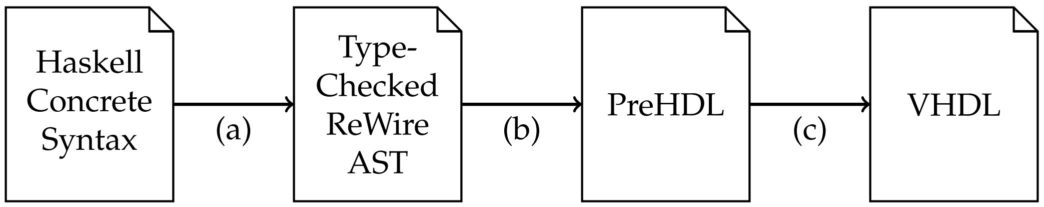


Figure 4.1: ReWire Compilation Process

entirety in Section 4.2. The final phase of the compiler (Figure 4.1c) is a relatively straightforward translation from PreHDL into a single-process VHDL state machine, discussed in Section 4.3. The ReWire Compiler is implemented entirely in Haskell.

Our running example program for the discussion of the compilation process (Figure 4.2) is an idealized two-function calculator, supporting addition, subtraction, and a "clear" operation (defined by the type `Oper` on line 3). To make things simple, we will assume that the calculator accepts a command on each clock cycle. The use of the monad `Calc = ReactT Oper W8 (StateT W8 Identity)`, defined on line 4, indicates that the circuit takes inputs of type `Oper`, produces outputs of type `W8` (a standard library type representing 8-bit vectors), and has internal state of type `W8`. Lines 6 and 7 define foreign functions written in VHDL implementing addition and subtraction on `W8`. It is, of course, quite possible to implement these functions directly in ReWire, but leaving them abstract allows us to define them at the back end in terms of VHDL's native + operator. The VHDL synthesis tools will choose the implementation most appropriate to the target device. The main program `loop` on lines 15-22 reads the current value from the store (line 16), then `signal`s it on the output and samples the next input (line 17) Then, depending on the input `Oper`, either an add, subtract, or clear operation (lines 18-21) is executed.

```
 1  module Calc where
 2
 3  data Oper = Add W8 | Sub W8 | Clr
 4  type Calc = ReactT Oper W8 (StateT W8 Identity)
 5
 6  vhdl plusW8  :: W8 -> W8 -> W8
 7  vhdl minusW8 :: W8 -> W8 -> W8
 8
 9  getVal :: Calc W8
10  getVal = lift get
11
12  putVal :: W8 -> Calc ()
13  putVal x = lift (put x)
14
15  loop :: Calc ()
16  loop = do x    <- getVal
17            oper <- signal x
18            case oper of
19              Add y -> putVal plusW8 x y
20              Sub y -> putVal minusW8 x y
21              Clr   -> putVal 0
22            loop
23
24  start :: Calc ((),W8)
25  start = extrude loop 0
```

Figure 4.2: Running Example: A Simple Two-Function Calculator

Finally, control tail-recursively returns to the top of the `loop` via a tail call (line 22). The `start` function defined on lines 24-25 (`start` is taken by the compiler to be the entry point to the program, à la `main` in C) initializes the state to 0 via `extrude`, then begins the loop.

4.1 Front End

The front end of the ReWire compiler consists of a parser written with the Parsec parser combinator library [?] and a type checker based on a pre-existing reference implementation called Typing Haskell in Haskell [?]. Expressions in the AST are annotated with sufficient type information to reconstruct the types of subexpressions without tracing the context in which they occur. Specifically, all variable binders and occurrences, and all occurrences of data constructors, carry an annotation recording their type at that occurrence. Thus, the polymorphic constructor $\mathit{Just} : a \rightarrow \mathit{Maybe}\,a$ can be tagged $\mathit{Int} \rightarrow \mathit{Maybe}\,\mathit{Int}$ in one position, and $\mathit{Bit} \rightarrow \mathit{Maybe}\,\mathit{Bit}$ in another, depending ultimately on the type of its argument.

One possibility we are currently considering is switching the front end to that provided by GHC, which has the advantages of extensive testing and performance tuning as well as a plethora of useful language extensions (which would allow us to implement, just to name two examples, metaprogramming via Template Haskell [?], and much-desired support for sized bit vectors in place of a fixed menu of types like $W8$, $W16$, and so on). We have held off on making this change primarily because of the implementation challenges posed by interfacing with GHC. While GHC is technologically unsurpassed, its internals are inconsistently documented and historically something of a moving target. It is also possible that we will wish to implement changes at a later stage that may break ReWire's subset relationship with Haskell; in this case, a simple reference implementation may give a better platform for experimentation than GHC.

4.2 Code Generation

In this section we discuss the generation of PreHDL code from ReWire programs. Ultimately, the goal of code generation is to produce synthesizable VHDL code for a circuit containing a single process of the form:

```
process(clk)
begin
  if clk'event and clk='1' then
    -- <loop body>
  end if;
end process;
```

where the loop body consists entirely of loop-free code. ReWire, however, allows for nested loops (implemented via tail recursion). The code generation function works by emitting code with goto (a construct that does not actually exist in VHDL, though it can be eliminated in a favor of structured programming constructs at a later pass [**?**]), which means that the single-loop structure we want is not guaranteed to be present. Therefore a significant amount of code transformation is needed to bring the program into this form. For this reason, the code generation pass (Figure 4.1b) generates programs in an imperative intermediate language called PreHDL, described in more detail in Section 4.2.1. Targeting PreHDL instead of VHDL directly allows us to implement the necessary code transformation passes on a much smaller language than VHDL itself.

The basic code generation procedure has three steps. First, a syntax-directed code generation process translates the ReWire source program into a PreHDL program which uses labels to represent tail call targets and contains **yield** statements that delineate the end of a clock cycle. This program may contain multiple nested loops (implemented in terms of goto). Such loops cannot be be implemented directly in VHDL, so in the second step, we convert the source program into a single-loop form where each iteration of the loop represents the action of a single clock tick. The guardedness criterion, as we will see, is critical to ensure that this transformation always succeeds. Finally, we perform a goto-elimination pass to restructure the loop body (which may contain forward gotos) in terms of structured if/then/else statements (and no gotos). In this final form, the program may be directly transliterated into a single VHDL process.

4.2.1 Definition of PreHDL

Figure 4.3 contains the grammar for PreHDL. The basic structure of a PreHDL program contains a short preamble indicating the types of the input and output channels for the circuit, followed by a collection of (pure) function definitions, and then the main body of the program, which is comprised of a sequence of variable declarations and a sequence of statements. Types are limited to booleans and bit vectors. The only nonstandard constructs relative to an ordinary imperative language are the **input** and **output** pseudovariables and the **yield** instruction. The informal semantics of **yield** is to signal the current **output** value on the circuit's output lines, wait until the next clock tick, and re-sample the **input** value. Operationally, this will correspond to an end to the current iteration of the VHDL process loop, whose statements are executed once at each clock tick. Variables declared at the top level are not considered to be in scope inside function bodies, and although this restriction is not reflected in the syntax, we assume that function bodies do not contain any **goto** or **yield** statements.

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \mathit{IODecl}\ \mathit{VarDecl}^{*}\ \mathit{FunDefn}^{*}\ \mathit{Stmt}^{*} \\
\mathit{IODecl} & ::= & \textbf{input} : \mathit{Ty}\ ;\ \textbf{output} : \mathit{Ty}\ ; \\
\mathit{VarDecl} & ::= & \mathit{Name} : \mathit{Ty}\ ; \\
\mathit{FunDefn} & ::= & \textbf{function}\ \mathit{Name}\ (\ \mathit{ParamList}\ ) : \mathit{Ty}\ \{ \\
 & & \quad \mathit{VarDecl}^{*}\ \mathit{Stmt}^{*} \\
 & & \quad \textbf{return}\ \mathit{Exp}\ ; \\
 & & \} \\
 & | & \textbf{vhdl}\ \mathit{Name}\ (\ \mathit{ParamList}\ ) : \mathit{Ty}\ ; \\
\mathit{ParamList} & ::= & \mathit{Params}\ \mid\ \epsilon \\
\mathit{Params} & ::= & \mathit{Param}\ ,\ \mathit{Params}\ \mid\ \mathit{Param} \\
\mathit{Param} & ::= & \mathit{Name} : \mathit{Ty} \\
\mathit{Stmt} & ::= & \mathit{LHS} := \mathit{Exp}\ ;\ \mid\ \textbf{if}\ \mathit{BExp}\ \{\ \mathit{Stmt}^{*}\ \}\ \textbf{else}\ \{\ \mathit{Stmt}^{*}\ \} \\
 & | & \textbf{label}\ \mathit{Name} :\ \mid\ \textbf{goto}\ \mathit{Name}\ ;\ \mid\ \textbf{yield}\ ; \\
\mathit{LHS} & ::= & \mathit{Name}\ \mid\ \textbf{output} \\
\mathit{Exp} & ::= & \mathit{BExp}\ \mid\ \mathit{Name}\ \mid\ \mathit{Name}\ (\ \mathit{ArgList}\ )\ \mid\ "\ \mathit{Bit}^{*}\ " \\
 & | & \mathit{Exp}\ [\ \mathit{Int} : \mathit{Int}\ ]\ \mid\ \textbf{concat}\ (\ \mathit{ArgList}\ )\ \mid\ \textbf{input} \\
\mathit{ArgList} & ::= & \mathit{Args}\ \mid\ \epsilon \\
\mathit{Args} & ::= & \mathit{Exp}\ ,\ \mathit{Args}\ \mid\ \mathit{Exp} \\
\mathit{BExp} & ::= & \mathit{BExp}\ \&\&\ \mathit{BExp}\ \mid\ \mathit{BExp}\ \|\ \mathit{BExp}\ \mid\ !\ \mathit{BExp} \\
 & | & \mathit{Exp} == \mathit{Exp}\ \mid\ \mathit{Name}\ \mid\ \textbf{true}\ \mid\ \textbf{false} \\
\mathit{Ty} & ::= & \textbf{boolean}\ \mid\ \textbf{bits}\ [\ \mathit{Int}\ ] \\
\mathit{Name} & ::= & \mathit{identifiers} \\
\mathit{Bit} & ::= & 0\ \mid\ 1 \\
\mathit{Int} & ::= & \mathit{Digit}^{+} \\
\mathit{Digit} & ::= & 0\ \mid\ 1\ \mid\ \cdots\ \mid\ 9
\end{array}
$$

Figure 4.3: PreHDL Syntax

For a concrete example of a PreHDL program the reader may wish to skip ahead to Figure 4.7 on page 112. This program corresponds to the final PreHDL output generated by the compiler for the example calculator program of Figure 4.2.

No formal semantics of PreHDL will be given here, but it is worth noting that such a formal semantics could be defined quite naturally in resumption-monadic terms. This should be beneficial to any future compiler-correctness proofs, as stating and proving the correctness property (up to PreHDL generation) does not require the construction of an elaborate correspondence between different semantic universes.

4.2.2 Translating ReWire into PreHDL

Most of the work of translating ReWire into synthesizable VHDL takes place in the first code generation phase, described in this section. The basic translation scheme works by translating non-resumption functions directly into PreHDL functions, and resumption-monadic computations into a sequence of statements containing gotos. For the sake of simplicity, the current implementation of the compiler, as well as the presentation of the process given in this section, will assume that any functions with monadic codomain not bound by **letrec** (including those typed in non-resumption monads) have been inlined in a previous pass; this restriction would not be difficult to eliminate if need be.

We will present here a detailed specification of the PreHDL code generator, in the form of pseudocode in a Haskell-like metalanguage. The metalanguage diverges from Haskell primarily in the sense that, for the sake of readability, we use $[\![-]\!]$ brackets as a quotation construct, representing terms in ReWire Core or

PreHDL abstract syntax. Interpolation of metalanguage variables is allowed inside quotation brackets; thus, for example, $\textbf{let}\, l = \text{“}\mathit{foo}\text{”}\ \textbf{in}\ [\![\textbf{goto}\ l;]\!]$ evaluates to the PreHDL statement $[\![\textbf{goto}\ \mathit{foo};]\!]$. We will also use ellipses inside patterns and expressions where convenient. If, for example, the list pattern $[r_1, r_2, \ldots, r_n]$ appears in an expression, it should be read as binding n variables named r_1, r_2, and so on up to r_n.

Representing Data Types

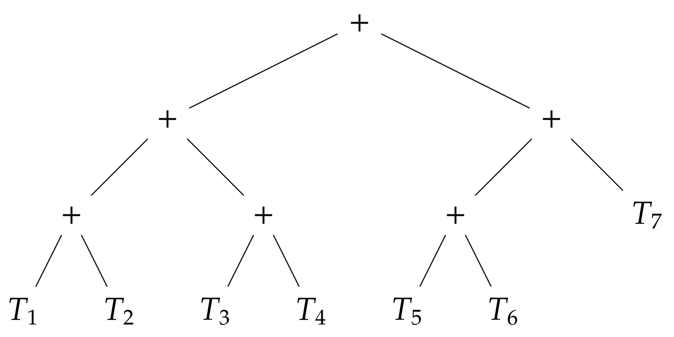
Non-recursive types in ReWire, as discussed in Chapter 3, may be desugared to ReWire Core's sum and product types. This is the representation of data types we will use here. Thus we will need to settle on a bit vector representation of sum and product types. The basic scheme adopted by ReWire is to represent the **nil** value as a zero-length bit vector; values $\langle v, u\rangle$ of product type $t \times t'$ as the concatenation of the bit vectors representing v and u; and sum values of the form $\textbf{inl}\ v : t + t'$ (respectively $\textbf{inr}\ u : t + t'$) as the bit vector representing v (respectively u) prefixed with an extra 0 (respectively 1), with 0-padding at the right-hand side of the bit vector making up any difference in size between t and t′. Thus the *sizeof* function, which computes the number of bits needed to represent a value of a given type, is defined as follows.

$$\begin{array}{lcl}
\mathit{sizeof} & :: & \mathit{RWPureTy} \rightarrow \mathit{Int} \\
\mathit{sizeof}\ [\![\textbf{nil}]\!] & = & 0 \\
\mathit{sizeof}\ [\![t + t']\!] & = & 1 + \mathit{max}\ (\mathit{sizeof}\ t)\ (\mathit{sizeof}\ t') \\
\mathit{sizeof}\ [\![t \times t']\!] & = & \mathit{sizeof}\ t + \mathit{sizeof}\ t'
\end{array}$$

Data types can be encoded in terms of simple sum and product types. Assume without loss of generality that all data constructors have an arity of 1. Then given a data type:

```
data D = C1 T1 | C2 T2 | ... | Cn Tn
```

we will assign a data constructor to each leaf node in a balanced binary tree in left-to-right order, and a + operation to each branch in that tree. Thus if we fix $n = 7$, we obtain:



Flattening this binary tree, we obtain the sum that we will use to represent D:

$$((T_1 + T_2) + (T_3 + T_4)) + ((T_5 + T_6) + T_7)$$

(where T_i here technically stands in for the *representation* of T_i in terms of sums and products). An expression of the form of $C_1\,e$ now corresponds to $\textbf{inl}\,(\textbf{inl}\,(\textbf{inl}\,e))$, $C_4\,e$ to $\textbf{inl}\,(\textbf{inr}\,(\textbf{inr}\,e))$, and $C_7\,e$ to $\textbf{inr}\,(\textbf{inr}\,e)$.

This construction induces the following relationship between expressions of D and their corresponding bit vectors. (Assume that the bit-string representation of $e : T_1$ is $d_0d_1d_2d_3$, implying that *sizeof* $T_1 = 4$, and assume without loss of generality that $T_1 = T_2 = \cdots = T_7$. "X" stands for "don't care".)

Expression	Bit Vector						
$C_1\,e$	0	0	0	d_0	d_1	d_2	d_3
$C_2\,e$	0	0	1	d_0	d_1	d_2	d_3
$C_3\,e$	0	1	0	d_0	d_1	d_2	d_3
$C_4\,e$	0	1	1	d_0	d_1	d_2	d_3
$C_5\,e$	1	0	0	d_0	d_1	d_2	d_3
$C_6\,e$	1	0	1	d_0	d_1	d_2	d_3
$C_7\,e$	1	1	d_0	d_1	d_2	d_3	0
(no valid expression)	1	1	X	X	X	X	1

Alternative Encoding One awkward thing about the encoding scheme described here is that the part of the bit vector corresponding to the data constructor variable width; as a result the data field is not always aligned, which is undesirable from an efficiency point of view. (Imagine a function that projects the *e* part out of a *D*, regardless of the constructor. If we had a fixed-width encoding of the tag, this would be a simple bit vector slice; but with the simplified encoding, the logic for such a function would have to check for the special case arising from the constructor C_7.) An alternative encoding scheme, which is actually used by the ReWire compiler, instead assigns a fixed-width tag of size $\lceil \log_2 n \rceil$, where n is the number of data constructors, producing a slightly different table for the example as follows.

Expression	Bit Vector						
$C_1\,e$	0	0	0	d_0	d_1	d_2	d_3
$C_2\,e$	0	0	1	d_0	d_1	d_2	d_3
$C_3\,e$	0	1	0	d_0	d_1	d_2	d_3
$C_4\,e$	0	1	1	d_0	d_1	d_2	d_3
$C_5\,e$	1	0	0	d_0	d_1	d_2	d_3
$C_6\,e$	1	0	1	d_0	d_1	d_2	d_3
$C_7\,e$	1	1	0	d_0	d_1	d_2	d_3
(no valid expression)	1	1	1	X	X	X	X

It is worth re-emphasizing that *this is the encoding scheme actually used by the ReWire compiler*. The implementation details surrounding the alternative encoding scheme are a bit complex, however, so we will proceed for the remainder of this section with the simplified encoding described previously.

Code Generation Monad

The code generator operates in a monad M that contains support for (1) an environment mapping ReWire names to binding information; (2) a stack of mutable *declaration frames* (the declaration frame on top of this stack consists of a list of variable declarations, which can be extended whenever a fresh temporary is generated); (3) a global list of emitted function declarations; and (4) fresh name generation:

```
type M = ReaderT [(RWName, Binding)]
           (StateT ([[PHDLVarDecl]], [PHDLFunDefn], Int)
             Identity)
```

(Here the *Int* in the third position of the state tuple is a counter for fresh names.) The basic interface for this monad is as follows:

```
withBindings   :: [(RWName, Binding)] → M a → M a
                  — executes a computation in an extended
                  — binding environment
askBinding     :: RWName → M Binding
                  — retrieves binding information for a variable
pushDeclFrame  :: [PHDLVarDecl] → M ()
                  — pushes a declaration frame onto the stack
popDeclFrame   :: M [PHDLVarDecl]
                  — pops a declaration frame from the stack
emitVarDecl    :: PHDLVarDecl → M ()
                  — emits a variable declaration in the top frame
getFunDefns    :: M [PHDLFunDefn]
```

```
                 — gets generated function definitions
emitFunDefn   :: PHDLFunDefn → M ()
                 — emits a function definition
freshName     :: M PHDLName
                 — generates a fresh name
```

The type *Binding* reflects four sorts of variable binding. If variable x maps to $RB\,l\,[r_1, \cdots, r_n]$ in the environment, this indicates that x is a recursively defined function (necessarily of resumption-monad type), whose PreHDL code begins at the label l and whose argument registers are the PreHDL variables $r_1, \cdots, r_n$. If x maps to $FB\,f$, then x is a non-recursively defined function, implemented as the PreHDL function f. (No information is needed with respect to the parameters.) If x maps to $VB\,r$, then x is a (local or global) variable mapped to the PreHDL variable r. Finally, if x maps to $PB\,r$, then x is a function parameter variable, mapped to the PreHDL function parameter r.

```
data Binding = RB PHDLName [PHDLName]
             | FB PHDLName
             | VB PHDLName
             | PB PHDLName
```

The presentation here assumes that ReWire programs always form an infinite loop. Eliminating this assumption would simply require us to declare a return value register, and decide at the VHDL level how to communicate the termination condition and return value to the outside world.

Compiling Expressions

The heart of the code generator is the function *cmp*, which compiles expressions. This function takes a ReWire expression and returns a list of PreHDL statements implementing the expression, and a PreHDL expression which will represent the result value after the execution of those statements:

$$cmp :: RWExpr \rightarrow M\,([PHDLStmt], PHDLExpr)$$

The variable/function application case proceeds by looking up the binding information for the variable/function name in the environment, and proceeding in one of three ways. In the case where the variable x corresponds to a recursive definition (meaning it is bound to a PreHDL label and list of argument registers), we emit code that fills the argument registers for that definition with the argument values, then jumps to its entry label. The result register is undefined in this case due to the assumption of non-termination. If x is bound to a PreHDL function, this is translated directly to a PreHDL function call, and if x is bound to a variable or function parameter, the expression is translated to a PreHDL variable reference.

```
$cmp\ [\![x\ e_1\ \cdots\ e_n\ :\ t]\!]\ =\ \textbf{do}$
  $b\ \leftarrow\ askBinding\ x$
  $\textbf{case}\ b\ \textbf{of}$
    $RB\ l\ [r_1, \cdots, r_n] \rightarrow\ \textbf{do}$
        $[(ce_1, ee_1), \cdots, (ce_n, ee_n)]\ \leftarrow\ mapM\ cmp\ [e_1, \cdots, e_n]$
        $return \left( \left[\!\!\left[ \begin{array}{l} ce_1;\ r_1\ :=\ ee_1; \\ \quad\vdots \\ ce_n;\ r_n\ :=\ ee_n; \\ \textbf{goto}\ l; \end{array} \right]\!\!\right], undefined \right)$
    $FB\ nf \rightarrow\ \textbf{do}$
        $[(ce_1, ee_1), \cdots, (ce_n, ee_n)] \leftarrow mapM\ cmp\ [e_1, \cdots, e_n]$
        $\textbf{let}\ st\ =\ sizeof\ t$
        $r\ \leftarrow freshName$
        $emitVarDecl\ [\![r\ :\ \textbf{bits}[st];]\!]$
        $return \left( \left[\!\!\left[ \begin{array}{l} ce_1;\ r_1\ :=\ ee_1; \\ \quad\vdots \\ ce_n;\ r_n\ :=\ ee_n; \\ r\ :=\ nf(r_1, \cdots, r_n); \end{array} \right]\!\!\right], [\![r]\!] \right)$
    $VB\ r \rightarrow\ return\ ([], [\![r]\!])$
    $PB\ r \rightarrow\ return\ ([], [\![r]\!])$
```

Let-expressions are compiled in a straightforward way, via a locally-bound PreHDL variable corresponding to the bound ReWire variable.

```
$cmp\ [\![\textbf{let}\ x\ :\ t\ =\ e\ \textbf{in}\ e'\ \textbf{end}]\!]\ =\ \textbf{do}$
  $(ce, ee) \leftarrow cmp\ e$
  $\textbf{let}\ st\ =\ sizeof\ t$
  $r\ \leftarrow freshName$
  $emitVarDecl\ [\![r\ :\ \textbf{bits}[st];]\!]$
  $(ce', ee')\ \leftarrow\ withBindings\ [(x, VB\ r)]\ (cmp\ e')$
  $return\ ([\![ce;\ r\ :=\ ee;\ ce';]\!], ee')$
```

The **nil** expression simply corresponds to an empty bit string.

$cmp\ [\![\textbf{nil}]\!]\ =\ return\ ([], [\![""]\!])$ — empty bit string

A pair expression simply results in a concatenation of the bit vectors produced by its arguments.

```
$cmp\ [\![\langle e, e' \rangle]\!] \ = \ \mathbf{do}$
  $(ce, ee) \ \leftarrow \ cmp\ e$
  $(ce', ee') \leftarrow \ cmp\ e'$
  $return\ ([\![ce;\ ce';]\!], [\![\mathbf{concat}(ee, ee')]\!])$
```

Where coproduct types are concerned, things are slightly more tricky. After obtaining the result of the argument expression, we prepend the appropriate tag, and pad the bit vector with zeros as necessary.

```
$cmp\ [\![\mathbf{inl}\ e \ : \ t \ + \ t']\!] \ = \ \mathbf{do}$
  $(ce, ee) \qquad \leftarrow cmp\ e$
  $\mathbf{let}\ st \qquad = \ sizeof\ t$
      $st' \qquad = \ sizeof\ t'$
      $padding = \ max\ (st' \ - \ st, 0)$
      $padbits \ = \ replicate\ padding\ [\![0]\!]$
  $return\ (ce, [\![\mathbf{concat}("0", ee, "padbits")]\!])$
$cmp\ [\![\mathbf{inr}\ e \ : \ t \ + \ t']\!] \ = \ \mathbf{do}$
  $(ce, ee) \qquad \leftarrow cmp\ e$
  $\mathbf{let}\ st \qquad = \ sizeof\ t$
      $st' \qquad = \ sizeof\ t'$
      $padding = \ max\ (st \ - \ st', 0)$
      $padbits \ = \ replicate\ padding\ [\![0]\!]$
  $return\ (ce, [\![\mathbf{concat}("1", ee, "padbits")]\!])$
```

To destruct tuples, we simply select out the appropriate slice of the underlying bit vector.

```
$cmp\ [\![\mathbf{fst}\ (e \ : \ t \ \times \ t')]\!] \ = \ \mathbf{do}$
  $(ce, ee) \ \leftarrow cmp\ e$
  $\mathbf{let}\ end = \ sizeof\ t - 1$
  $return\ (ce, [\![ee[0 : end]]\!])$
$cmp\ [\![\mathbf{snd}\ (e \ : \ t \ \times \ t')]\!] \ = \ \mathbf{do}$
  $(ce, ee) \ \ \leftarrow cmp\ e$
  $\mathbf{let}\ start = \ sizeof\ t$
      $end \ \ = \ start \ + \ sizeof\ t' \ - \ 1$
  $return\ (ce, [\![ee[start : end]]\!])$
```

Pattern matching (i.e., destruction of sums) takes a bit more effort to compile, mostly owing to the need for branching control flow. Essentially we compute the bit vector for e, and proceed with the code for expression e' if the resulting tag bit is zero, or for e'' otherwise. In either case, pattern matching variables are bound to the slice of the bit vector corresponding to the data field.

```
$cmp\ [\![(\textbf{case}\ (e\ :\ t\ +\ t')\ \textbf{of inl}\ x\ \rightarrow\ e'\ ;\ \textbf{inr}\ y\ \rightarrow\ e''\ \textbf{end})\ :\ t'']\!]\ =\ \textbf{do}$
  $(ce, ee) \leftarrow cmp\ e$
  $nx \quad \leftarrow freshName$
  $ny \quad \leftarrow freshName$
  $r \quad\ \leftarrow freshName$
  $\textbf{let}\ st\ =\ sizeof\ t$
      $st'\ =\ sizeof\ t'$
      $st''\ =\ sizeof\ t''$
  $emitVarDecl\ [\![nx\ :\ \textbf{bits}[st];]\!]$
  $emitVarDecl\ [\![ny\ :\ \textbf{bits}[st'];]\!]$
  $emitVarDecl\ [\![r\ :\ \textbf{bits}[st''];]\!]$
  $(ce', ee') \ \leftarrow\ withBindings\ [(x, VB\ nx)]\ (cmp\ e')$
  $(ce'', ee'') \leftarrow\ withBindings\ [(y, VB\ ny)]\ (cmp\ e'')$
  $return\ ($ $[\![$
      $ce;$
      $\textbf{if}\ ee[0:0]\ ==\ "0"\ \textbf{then}\ \{$
          $nx\ :=\ ee[1:st];$
          $ce';$
          $r\ :=\ ee';$
      $\}$
      $\textbf{else}\ \{$
          $ny\ :=\ ee[1:st'];$
          $ce'';$
          $r\ :=\ ee'';$
      $\}$
  $]\!]\ ,\ [\![r]\!]\ )$
```

Compilation of the basic monadic primitives **return**, **bind**, and **lift** is straightforward. **Bind** expressions are compiled identically to **let** expressions, while **return** and **lift** are operational no-ops.

$$cmp\ [\![\textbf{return}\ e]\!]\ =\ cmp\ e$$

```
$cmp\ [\![\mathbf{bind}\ (x\ :\ t)\ \leftarrow\ e\ \mathbf{in}\ e'\ \mathbf{end}]\!]\ =\ \mathbf{do}$
  $(ce, ee) \leftarrow cmp\ e$
  $\mathbf{let}\ st\ =\ sizeof\ t$
  $r\ \leftarrow freshName$
  $emitVarDecl\ [\![r\ :\ \mathbf{bits}[st];]\!]$
  $(ce', ee')\ \leftarrow\ withBindings\ [(x, VB\ r)]\ (cmp\ e')$
  $return\ ([\![ce;\ r\ :=\ ee;\ ce';]\!], ee')$

$cmp\ [\![\mathbf{lift}\ e]\!]\ =\ cmp\ e$
```

State monad operations may be compiled in a type-directed fashion. If the operation is typed in a monad with n state layers, we know that the operation should affect the n^{th} state variable state_n. **Get**-expressions, therefore, simply copies state_n into a temporary that is returned to the caller, and **put** overwrites state_n.

```
$cmp\ [\![\mathbf{get}\ :\ \mathbb{S}_{t_n}\mathbb{S}_{t_{n-1}}\cdots\mathbb{S}_{t_1}\mathbb{I}(t_n)]\!]\ =\ \mathbf{do}$
  $\mathbf{let}\ st_n\ =\ sizeof\ t_n$
  $r\ \leftarrow freshName$
  $emitVarDecl\ [\![r\ :\ \mathbf{bits}[st_n];]\!]$
  $return\ ([\![r\ :=\ \mathrm{state}_n;]\!], [\![r]\!])$

$cmp\ [\![\mathbf{put}\ e\ :\ \mathbb{S}_{t_n}\mathbb{S}_{t_{n-1}}\cdots\mathbb{S}_{t_1}\mathbb{I}(())]\!]\ =\ \mathbf{do}$
  $(ce, ee)\ \leftarrow\ cmp\ e$
  $return\ ([\![ce;\ \mathrm{state}_n\ :=\ ee;]\!], [\![\text{""}]\!])$
```

(Note that in the case for **get**, it would not do to simply return state_n as the result expression, since this value could be overwritten before use.)

For the reactive resumption operation **signal** we will copy the result of the argument expression into the (pseudo-)variable **output**, **yield** until the next clock tick, and read the newly sampled **input** into a freshly allocated temporary.

```
$cmp\ [\![\mathbf{signal}\ e\ :\ \mathbb{R}_{t,t'} \cdots \mathbb{I}(t)]\!]\ =\ \mathbf{do}$
    $(ce, ee) \leftarrow cmp\ e$
    $r \qquad\quad \leftarrow freshName$
    $\mathbf{let}\ st \ \ =\ sizeof\ t$
    $emitVarDecl\ [\![r\ :\ \mathbf{bits}[st];]\!]$
    $return\ ([\![ce;\ \mathbf{output}\ :=\ ee;\ \mathbf{yield};\ r\ :=\ \mathbf{input};]\!], [\![r]\!])$
```

Finally, expressions of the form **extrude** $e\ e'$ operation, whose semantics as given in Figure 3.6 seemed so complicated, compile merely to the initialization of a state variable to the value of e', followed by the execution of e.

```
$cmp\ [\![\mathbf{extrude}\ (e\ :\ \mathbb{R}_{t,t'}\mathbb{S}_{t_n}\mathbb{S}_{t_{n-1}} \cdots \mathbb{S}_{t_1}\mathbb{I}(t''))\ e']\!]\ =\ \mathbf{do}$
    $(ce', ee') \leftarrow cmp\ e'$
    $(ce, ee) \ \ \leftarrow cmp\ e$
    $r \qquad\quad \leftarrow freshName$
    $\mathbf{let}\ st_n \ \ =\ sizeof\ [\![t'' \times t_n]\!]$
    $emitVarDecl\ [\![r\ :\ \mathbf{bits}[st_n];]\!]$
    $return\ ([\![ce';\ \mathrm{state}_n\ :=\ ee';\ ce;\ r\ :=\ \mathbf{concat}(ee, \mathrm{state}_n);]\!], [\![r]\!])$
```

Compiling Programs

The main entry point of the code generator is the function *cmpProg* which takes a ReWire program and produces a PreHDL program. This function proceeds by compiling the program body via *cmpBody*, then embedding the resulting statements in a PreHDL program that contains appropriate declarations for the input and output types, as well as any function definitions and variable declarations emitted in the process of compiling the body. This function is also responsible for declaring the variables $\mathrm{state}_1, \cdots, \mathrm{state}_n$ corresponding to the state monad layers.

```
$cmpProg :: RWProg \rightarrow M\ PHDLProg$
$cmpProg\ [\![ \textbf{program}\ y\ \textbf{end}\ :\ \mathbb{R}_{t,t'} \mathbb{S}_{t_n} \mathbb{S}_{t_{n-1}} \cdots \mathbb{S}_{t_1} \mathbb{I}(t'') ]\!] = \textbf{do}$
  $pushDeclFrame\ []$
  $(cy, \_)$              $\leftarrow cmpBody\ y$
  $[d_1, \cdots, d_m]$    $\leftarrow popDeclFrame$
  $\textbf{let}\ st_i$    $= sizeof\ t$
      $st_o$              $= sizeof\ t'$
      $[st_1, \cdots, st_n] = map\ sizeof\ [t_1, \cdots, t_n]$
  $[fd_1, \cdots, fd_j]$  $\leftarrow getFunDefns$
  $return\ [\![$ $\textbf{input} : \textbf{bits}[st_i];\ \textbf{output} : \textbf{bits}[st_o];$
              $\mathrm{state}_1 : \textbf{bits}[st_1];\ \cdots\ \mathrm{state}_n : \textbf{bits}[st_n];$
              $d_1;\ \cdots\ d_m;$
              $fd_1;\ \cdots\ fd_j;$
              $cy;$ $]\!]$
```

The function *cmpBody* proceeds recursively with two cases. For **letfun**, we first compile the function being defined, which will result (inside *cmpFunDefn*) in the emission of a function declaration, then recurse on the remainder of the program body. For **letrec**, we must first pre-allocate labels and argument registers for the recursively-bound functions via *recDefnBinding*, then compile the definitions of those functions via *cmpRecDefn*, and finally compile the body of the **letrec**—which will be the main entry point to the program—with the expression compilation function *cmp*.

```
$cmpBody :: RWBody \rightarrow M\ ([PHDLStmt], PHDLExpr)$
$cmpBody\ [\![ \textbf{letfun}\ l\ \textbf{in}\ y\ \textbf{end} ]\!] = \textbf{do}$
  $b \leftarrow cmpFunDefn\ l$
  $withBindings\ [b]\ (cmpBody\ y)$
$cmpBody\ [\![ \textbf{letrec}\ l_1 ; \cdots ; l_n\ \textbf{in}\ e\ \textbf{end} ]\!] = \textbf{do}$
  $bs$                    $\leftarrow mapM\ recDefnBinding\ [l_1, \cdots, l_n]$
  $[cl_1, \cdots, cl_n]$  $\leftarrow withBindings\ bs$
                              $(mapM\ cmpRecDefn\ [l_1, \cdots, l_n])$
  $(ce, ee)$              $\leftarrow withBindings\ bs\ (cmp\ e)$
  $return\ ([\![ ce;\ cl_1;\ \cdots\ cl_n; ]\!], ee)$
```

The *recDefnBinding*, which is responsible for pre-allocating labels and argument registers, operates as follows. The *Binding* corresponding to the definition will be returned as a result.

```
$recDefnBinding$ :: $RWLetDefn \rightarrow M\ (RWName, Binding)$
$recDefnBinding\ [\![f\ (x_1 : t_1) \cdots (x_n : t_n) = e]\!] =$ **do**
  $l \leftarrow freshName$
  $[r_1, \cdots, r_n] \leftarrow mapM\ (const\ freshName)\ [x_1, \cdots, x_n]$
  **let** $[st_1, \cdots, st_n] = map\ sizeof\ [t_1, \cdots, t_n]$
  $mapM\ (\lambda\ (r, st) \rightarrow emitVarDecl\ [\![r : \mathbf{bits}[st];]\!])$
       $[(r_1, st_1), \cdots, (r_n, st_n)]$
  $return\ (f, RB\ l\ [r_1, \cdots, r_n])$
```

Function *cmpFunDefn*, which operates on non-recursive definitions, first generates fresh PreHDL names corresponding to the function name and to each function parameter, enters the parameter bindings into the environment, and compiles the function body. The resulting code is then wrapped up in a PreHDL function definition, which is added to the global function definition list via *emitFunDefn*. We then return the resulting binding to the caller, which is responsible for inserting it into the environment as appropriate (see *cmpBody*).

```
$cmpFunDefn$ :: $RWLetDefn \rightarrow M\ (RWName, Binding)$
$cmpFunDefn\ [\![f\ (x_1 : t_1) \cdots (x_n : t_n) = e : t_e]\!] =$ **do**
  $[r_1, \cdots, r_n] \leftarrow mapM\ (const\ freshName)\ [x_1, \cdots, x_n]$
  **let** $pBdgs = [(x_1, PB\ r_1), \cdots, (x_n, PB\ r_n)]$
  $pushDeclFrame\ []$
  $(ce, ee) \leftarrow withBindings\ pBdgs\ (cmp\ e)$
  $[d_1, \cdots, d_m] \leftarrow popDeclFrame$
  **let** $[st_1, \cdots, st_n] = map\ sizeof\ [t_1, \cdots, t_n]$
      $st_e = sizeof\ t_e$
  $nf \leftarrow freshName$
  $emitFunDefn$
    [[ **function** $nf\ (r_1 : \mathbf{bits}[st_1], \cdots, r_n : \mathbf{bits}[st_n])\ :\ \mathbf{bits}[st_e]$ {
         $d_1; \cdots; d_m;$
         $ce;$
         **return** $ee;$
       } ]]
  $return\ (f, FB\ nf)$
```

In the case of a recursive definition, *cmpRecDefn* simply emits straight-line code for the body of the function prefixed with the pre-allocated label. Note that because we are assuming that the program runs forever, we will simply ignore the expression returned by *cmp*.

```
$cmpRecDefn :: RWLetDefn \rightarrow M\,[PHDLStmt]$
$cmpRecDefn\ [\![f\ x_1\ \cdots\ x_n\ =\ e]\!]\ =$ do
  $RB\ l\ [r_1, \cdots, r_n] \leftarrow askBinding\ f$
  let $pBdgs$       $=\ [(x_1, PB\ r_1), \cdots, (x_n, PB\ r_n)]$
  $(ce, \_)$          $\leftarrow withBindings\ pBdgs\ (cmp\ e)$
  $return\ [\![$label $l:\ ce;]\!]$
```

With this, the code generator is complete.

4.2.3 PreHDL Transformations

The code generator of the preceding section will produce PreHDL code that, while semantically valid, is not ready for compilation to VHDL for two reasons. First, the output contains **goto** statements. These cannot directly be translated to VHDL. Second, the code may contain multiple and nested loops. This is also difficult to represent directly in VHDL, as most VHDL synthesis tools only support loops in the context of generics; put another way, all VHDL loops must be completely unrollable at compile time.

We will address the impedance mismatch with two source-to-source transformations implemented against PreHDL, both of which are discussed in this section. The first transformation, called *loop flattening*, allows us to transform a PreHDL program which may contain multiple loops into a single loop, suitable for implementation in a single VHDL process. We shall see that ReWire's guardedness criterion is critical to the success of this technique. The second transformation, called *goto elimination*, converts the loop body, which contains only *forward* gotos, into a structured program fragment where if-then-else statements stand in for the gotos. This is an instance of a more general technique due to Erosa and Hendren [**?**], whose goto-elimination algorithm also handles backward jumps (but, in the process, introduces looping constructs that, as we have said, are not directly implementable in VHDL).

Control Flow Graphs

The loop flattening transformation operates on a PreHDL control flow graph, generated from the program returned by the code generation pass. In constructing a CFG, we consider **yield** to be a control flow instruction, and it is therefore considered to end a basic block. Formally, every edge in the control flow graph will be tagged with a branch condition of the form:

$$\text{BranchCond} ::= \textit{Jump} \mid \textit{BranchF}\ \text{BExp} \mid \textit{BranchT}\ \text{BExp} \mid \textit{Yield}$$

where *Jump* denotes an unconditional jump, *BranchF* and *BranchT* denote branches on false/true, and *Yield* denotes that the source basic block terminated with a **yield** statement.

Definition (Control Flow Graph)**.** *A PreHDL* control flow graph *(CFG) is a tuple $\langle V, E, u, f, g\rangle$, containing a finite set of vertices $V \subset \mathbb{N}$, a set of edges $E \subseteq V \times V$, a designated start vertex $u \in V$, a vertex labeling function $f : V \to Stmt^*$, and an edge labeling function $g : E \to BranchCond$, with $\langle V, E\rangle$ forming a directed graph, where for every $v \in V$ exactly one of the the following conditions holds:*

1. *there exists exactly one $w \in V$ such that $g(v, w) = Jump$; or*

2. *there exist exactly two vertices $w, \bar{w} \in V$ and an expression $e \in$* BExp *such that $g(w) = BranchF\ e$ and $g(\bar{w}) = BranchT\ e$; or*

3. *there exists exactly one $w \in V$ such that $g(v, w) = Yield$.*

In other words, every node in the control flow graph may branch on a single boolean expression, *or* jump unconditionally to another node, *or* yield before jumping unconditionally to another node.

The loop flattening algorithm requires a control flow graph of a special form, called a *guarded control flow graph*, as follows:

Definition (Guarded Control Flow Graph)**.** *A* guarded control flow graph *(GCFG) is a control flow graph $\langle V, E, u, f, g\rangle$ such that every cycle in the graph $\langle V, E\rangle$ contains at least one edge e such that $g(e) = Yield$.*

Fortunately, *programs produced by the code generation pass will always have this property*. This is a result of ReWire's guardedness condition. To see this, recall that in ReWire, the only opportunity for control flow loops arises from (tail) recursive function calls. The guardedness condition guarantees that between entry into a recursively-defined function and exit to another recursively-defined function, a **signal** must occur. Since the code generated for **signal** expressions contains a **yield** statement whose execution is unconditional, we can be certain that every loop in the control flow graph will eventually **yield**.

Loop Flattening

Given a guarded control flow graph, we can produce a *linear* control flow graph which is suitable for implementation as a single loop:

Definition (Linear Control Flow Graph)**.** *A* linear control flow graph *(LCFG) is a guarded control flow graph $\langle V, E, u, f, g\rangle$ such that there exists an edge $e \in E$ where for every e', $g(e') = Yield$ implies that $e' = e$, and the graph $\langle V, E \setminus \{e\}\rangle$ is acyclic.*

The process of converting a guarded control flow graph to a linear control flow graph is called *loop flattening*. The action of loop flattening is illustrated in Figure 4.5 on page 108. On the left we have a graph that contains multiple yield edges (indicated by red dashed arrows). (Note that branch conditions are not notated here, but it is assumed that for any node with out-degree 2, both branches are labeled with complementary *BranchT* and *BranchF* conditions.) On the right, we have a semantically equivalent graph with only one yield edge, and the deletion of that edge results in an acyclic graph. Note that loop flattening has resulted in the creation of new start and end nodes called A (pronounced alpha) and Ω respectively, and the addition of a number of edges to the graph (indicated by green dotted arrows). In the figure, the added edges are tagged either with a condition (indicated by a question mark), or an action (indicated without a question mark). These correspond to an additional variable that has been added to the program indicating the *re-entry point* after a yield. The condition Rn? indicates a test that the re-entry point is n, and the action Rn indicates that the re-entry point should be assigned n just prior to the transition to Ω.

The full algorithm for loop flattening graph is given in Figure 4.4 on page 107. Both here and in Figure 4.5 we will trivially generalize the definition of a control flow graph by allowing FLATTEN to produce a graph with multiple edges between two nodes (meaning the CFG is really a multigraph), and with with more than two conditional edges outbound from a node, as long as the conditions labeling those edges are mutually exclusive (which they will be, if produced by FLATTEN). This abuse simplifies presentation but does not conceptually alter the algorithm. Note that the *input* graph to FLATTEN must still conform to the strict definition; again, this restriction exists merely to enhance the clarity of exposition.

After the control flow graph has been flattened, we may convert our program back to straight-line code containing gotos, representing the body of a single infinite loop, as follows. First, we delete the yield edge from the graph, resulting in an acyclic graph. Second, we assign to each node in the graph a freshly generated label. Third, to the code at each node in the graph except Ω we append a series of (possibly conditional) goto statements corresponding to its outbound edges. To Ω we append a **yield** statement. Fourth, we topologically sort the graph, taking care to place Ω at the end of the resulting list. (To ensure this, we can take any valid topological ordering on the nodes and simply move Ω to the end, since it has no outbound edges.) Fifth, we output the code of each node in the CFG in topological order (guaranteeing that there will be no backward jumps). The resulting code for the flattened CFG of Figure 4.5 is given in Figure 4.6.

Goto Elimination

Once the loop body has been flattened, we may eliminate unstructured gotos from the loop body by applying a goto-elimination transformation [**?**]. This transformation operates essentially by generating a boolean variable `goto_L` for each goto

```
procedure FLATTEN($G$)
    ▷ Add new start node $A$ and end node $\Omega$ with empty statement lists.
    $A \leftarrow \text{AddVertex}(G, [])$;
    $\Omega \leftarrow \text{AddVertex}(G, [])$;

    ▷ Gather yield edges.
    $Y \leftarrow \{e \mid e \in \text{Edges}(G), \text{EdgeLabel}(G, e) = \mathit{Yield}\}$;

    for $e \in Y$ do
        $s \leftarrow \text{EdgeSrc}(e)$;
        $t \leftarrow \text{EdgeDest}(e)$;

        ▷ Add to $s$ a jump to $\Omega$ and a statement to update entry point.
        $\text{AddEdge}(G, s, \Omega, \mathit{Jump})$;
        $\text{SetVertexLabel}(G, s, \text{VertexLabel}(G, s)$++[entrypoint := $t$;]);

        ▷ Add branch from $A$ to the target.
        $\text{AddEdge}(G, A, t, \mathit{BranchT}$ (entrypoint == $t$));

        ▷ Delete the old yield edge.
        $\text{DeleteEdge}(G, e)$;
    end for

    ▷ Add branch from $A$ to old start node, and set $A$ as new start node.
    $\alpha \leftarrow \text{StartVertex}(G)$;
    $\text{AddEdge}(G, A, \alpha, \mathit{BranchT}$ (entrypoint == $\alpha$));
    $\text{SetStartVertex}(G, A)$;

    ▷ Add yield edge from $\Omega$ to $A$.
    $\text{AddEdge}(G, \Omega, A, \mathit{Yield})$;
end procedure
```

Figure 4.4: Loop Flattening Algorithm. We will assume that a variable `entrypoint` is added to the program, with suitable type to represent every yield-target from the original graph, and with an initial value equal to the old start node.

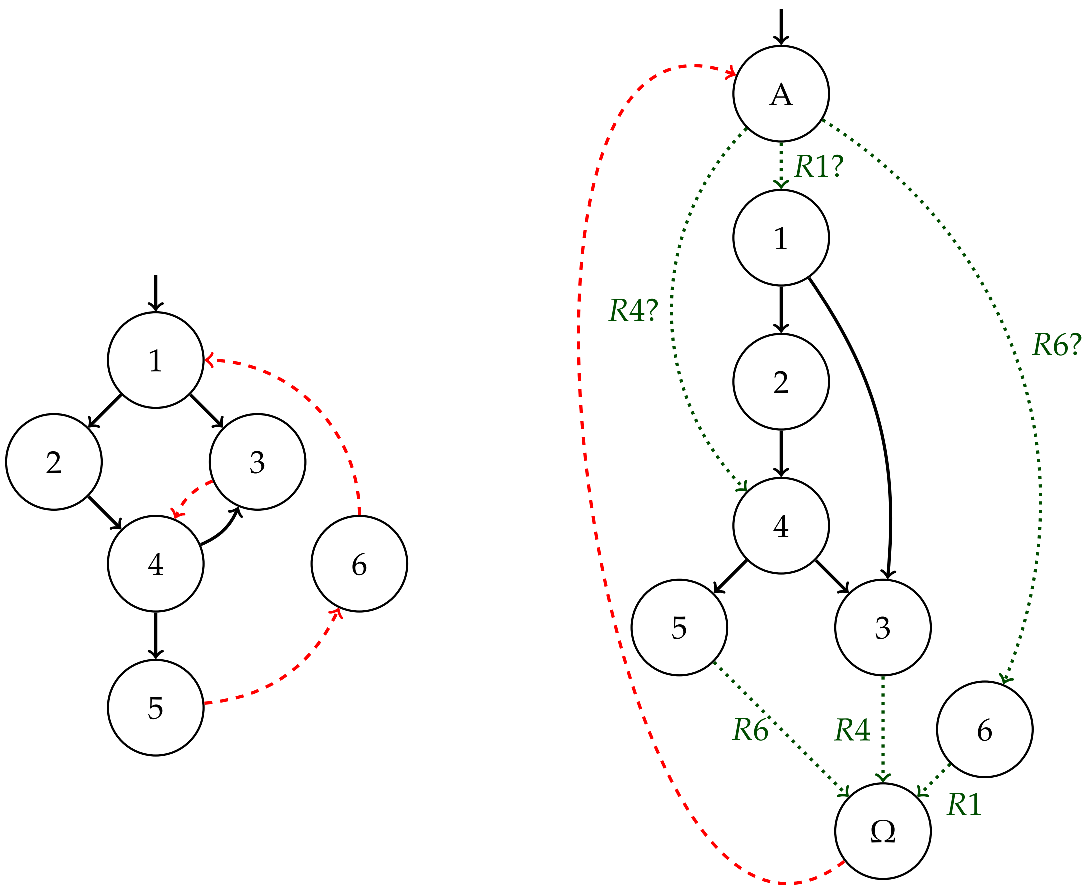


Figure 4.5: PreHDL CFG Before (l) and After (r) Loop Flattening

```
label L_A:
  if entrypoint==4 then goto L_4;
  else if entrypoint==1 then goto L_1;
  else if entrypoint==6 then goto L_6;
  else goto L_1;
label L_1:
  [[code originally at 1]];
  if [[condition for branch from 1 to 2]] goto L_2;
  else goto L_3;
label L_2:
  [[code originally at 2]];
  goto L_4;
label L_4:
  [[code originally at 4]];
  if [[condition for branch from 4 to 5]] goto L_5;
  else goto L_3;
label L_5:
  [[code originally at 5]];
  entrypoint := 6;
  goto L_Omega;
label L_3:
  [[code originally at 3]];
  entrypoint := 4;
  goto L_Omega;
label L_6:
  [[code originally at 6]];
  entrypoint := 1;
  goto L_Omega;
label L_Omega:
  yield;
```

Figure 4.6: Flattened Code From CFG of Figure 4.5

target label *L*, and performing a series of rewrites that push goto statements ever further down in the program text; for example, a goto into the body of an `if` may be pushed down as follows:

```
   if (condition) goto L;
   statements1;
   if (e) {
     statements2;  // contains label L
   }

        ==>

   goto_L := condition;
   if (!goto_L) {
     statements1;
   }
   if (e || goto_L) {
     if (goto_L) goto L;
     statements2;
   }
```

Other rewrite rules exist for gotos that exit the body of an `if`, and for all other control flow constructs. Through repeated application of these rules a condition will eventually be reached where the goto statement is at the same "height" in the syntax tree as its target label:

```
   if (condition) goto L;
   statements1;
label L:
   statements2;
```

at which point the goto may be removed from the program:

```
   goto_L := condition;
   if(!goto_L) {
     statements1;
   }
label L:
   goto_L := false;
   statements2;
```

Experience suggests that this transformation does a good job of eliminating gotos without introducing excessive overhead.

Example

For reference, the PreHDL code that is produced by the compiler for the example of Figure 4.2 is presented in Figure 4.7. The resulting code has been cleaned up cosmetically for presentation purposes. Note that variable `entrypoint` is introduced during the loop flattening process; the values it may take are in one-to-one correspondence with the **yield** statements occurring in the unflattened program generated by the code generation phase.

4.3 VHDL Generation

Once a PreHDL program has been converted to the final form exemplified by Figure 4.7, translation to VHDL is straightforward. The loop structure of Figure 4.7 is replaced with a VHDL `process`, but the body of the loop is essentially identical except for shallow syntactic differences. The final VHDL code for the calculator example of Figure 4.2 is listed in Figure 4.8. For completeness' sake we may report that this VHDL code, when synthesized by Xilinx's XST synthesis tool (ISE version

```
input      : bits[10]; output : bits[8];
s0         : bits[8];
entrypoint : bits[1];
vhdl plusW8  (bits[8], bits[8]) : bits[8];
vhdl minusW8 (bits[8], bits[8]) : bits[8];

  entrypoint := "0";
label LOOP:
  if (entrypoint == "0")
    { s0 := "00000000"; }
  else if (entrypoint == "1") {
    if ("00" == input[0:1])
     { s0 := plusW8 (s0,input[2:9]); }
    else if ("01" == input[0:1])
     { s0 := minusW8 (s0,input[2:9]); }
    else
     { s0 := "00000000"; }
  }
  output := s0;
  entrypoint := "1";
  yield;
  goto LOOP;
```

Figure 4.7: PreHDL Output for the Calculator Example

14.7) for a Kintex-7 KC705 FPGA (XC7K70T, speed grade -2), produced a circuit capable of operation at around a clock rate of around 760MHz.

VHDL veterans may notice a slight oddity in the structure of the resulting code. Namely, the *output* value, rather than the input value, is registerized. In practice this results in slightly undesirable timing behavior: given an input arriving at clock tick t, the output register will not latch until tick $t + 1$. From that moment, there is a short delay until the output value from the register stabilizes. As a result, the output value is not actually available to be sampled by another device tied to the same clock until time $t + 2$. A newer version of the ReWire compiler will remedy this deficiency by registerizing the input lines rather than the outputs, but it was not quite ready as this dissertation went to press. In the meantime we note that for streaming applications, such as the regular expression matchers of Chapter 6, this single-tick delay is not actually a problem, and even for more timing-sensitive applications workarounds are available.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.prims.all;

entity Calc is
  Port (clk    : in std_logic ;
        input  : in std_logic_vector (0 to 9);
        output : out std_logic_vector (0 to 7));
end Calc;

architecture ReWire of Calc is
begin
  process(clk)
   variable s0         : std_logic_vector(0 to 7);
   variable entrypoint : std_logic_vector(0 to 0) := "0";
  begin
    if clk'event and clk='1' then
      if entrypoint = "0" then
        s0 := "00000000";
      elsif entrypoint = "1" then
        if input(0 to 1) = "00" then
          s0 := plusW8 (s0,input(2 to 9));
        elsif input(0 to 1) = "01" then
          s0 := minusW8 (s0,input(2 to 9));
        else
          s0 := "00000000";
        end if;
      end if;
      output <= s0;
      entrypoint := "1";
    end if;
  end process;
end Calc;
```

Figure 4.8: Final VHDL Output for the Calculator Example

Chapter 5

Case Study I: A Simple CPU

This chapter presents the first of two case studies of circuit design in ReWire. The example design is a simple 8-bit CPU. The purpose of this case study is to demonstrate that ReWire provides a useful platform for semantically modular circuit design. The semantically modular design style allows the rapid development of a viable implementation 5.2 from a very high level design sketch 5.1 without getting bogged down in structural circuit details. Moreover, as sketched in Section 5.4, semantic modularity enables us to extend our design with security features without modifying the basic outline of the design.

5.1 Basic Design

The CPU we will develop in this chapter is an 8-bit CPU with a simple instruction set architecture. The instruction set architecture is given in Table 5.1. We will assume that CPU is connected to (1) an 8-bit data bus, a reset line, and an interrupt line on input; and (2) an 8-bit data bus, an 8-bit output address bus, a write-enable line, and an interrupt acknowledge line on output. The CPU has four general purpose registers r_0 through r_3, an 8-bit program counter, and flags for zero, carry, and interrupt enable. When an interrupt occurs, the zero and carry bits and the program counter will be stored in "save" registers, which are restored by the IRET instruction.

5.2 Code for the Simple CPU

5.2.1 Input and Output Types

We begin by declaring the types of input and output signals for the CPU.

```
data Inputs  = Inputs W8 Bit Bit
data Outputs = Outputs W8 W8 Bit Bit
```

The Inputs type has only one (eponymous) constructor and consists of three fields: the first, of type W8, represents the 8-bit connection to the data input bus. The second and third fields, both of type Bit, correspond to the reset and interrupt lines. The Outputs type also has one constructor, and consists of four fields: the first, of type W8, represents the address lines The second, of type W8, represents the data output bus (used for memory writes). The third, of type Bit, is a write-enable flag for memory accesses. The fourth, of type Bit, is an interrupt-acknowledge signal.

For convenience we will define "getter" functions for each of the Input fields.

Mnemonic	Encoding	Pseudocode	Flags

`MEM`	`0000RWrraaaaaaaa`	if R then reg(r) := mem(a)
		else if W then mem(a) := reg(r)
			–

`LD`	`0001ddaa`	reg(d) := mem(reg(a))	–
`ST`	`0010ddaa`	mem(reg(a)) := reg(d)	–
`ADD`	`0011ddss`	reg(d) := reg(d) + reg(s)	ZC
`ADDC`	`0100ddss`	reg(d) := reg(d) + reg(s) + C	ZC
`SUB`	`0101ddss`	reg(d) := reg(d) - reg(s)	ZC
`SUBB`	`0110ddss`	reg(d) := reg(d) - reg(s) - C	ZC
`MOV`	`0111ddss`	reg(d) := reg(s)	–
`OR`	`1000ddss`	reg(d) := reg(d) OR reg(s)	ZC
`AND`	`1001ddss`	reg(d) := reg(d) AND reg(s)	ZC
`XOR`	`1010ddss`	reg(d) := reg(d) XOR reg(s	ZC
`CMP`	`1011ddss`	compute := reg(d) - reg(s)	ZC
`BRZ`	`110000rr`	pc := reg(r) if Z=0	–
`BRNZ`	`110001rr`	pc := reg(r) if Z!=0	–
`BRC`	`110010rr`	pc := reg(r) if C=0	–
`BRNC`	`110011rr`	pc := reg(r) if c!=0	–
`JMP`	`110100rr`	pc := reg(r)	–
`IEN`	`1101010E`	IE := E	–
`IACK`	`11010110`	output iack signal	–
`IRET`	`11010111`	return from interrupt	ZC
`NOT`	`110110rr`	reg(r) := NOT (reg(r))	–
`CLRR`	`110111rr`	reg(r) := 0	–
`INCR`	`111000rr`	reg(r) := reg(r) + 1	ZC
`DECR`	`111001rr`	reg(r) := reg(r) - 1	ZC
`ROTL`	`111010rr`	reg(r) := reg(r) ROL 1	–
`ROTR`	`111011rr`	reg(r) := reg(r) ROR 1	–
`SHLA`	`111100rr`	reg(r) := reg(r) shla 1	ZC
`SHLL`	`111110rr`	reg(r) := reg(r) shll 1	ZC
`SHRA`	`111101rr`	reg(r) := reg(r) shra 1	ZC
`SHRL`	`111111rr`	reg(r) := reg(r) shrl 1	ZC

Table 5.1: Instruction Set Architecture for the Simple CPU

```
dataIn :: Inputs -> W8
dataIn (Inputs d_i _ _) = d_i

rstIn :: Inputs -> Bit
rstIn (Inputs _ r_i _) = r_i

intIn :: Inputs -> Bit
intIn (Inputs _ _ i_i) = i_i
```

We can also define “setter” functions for each of the `Output` fields.

```
setAddrOut :: Outputs -> W8 -> Outputs
setAddrOut (Outputs _ d_o we_o iack_o) a_o =
   Outputs a_o d_o we_o iack_o

setDataOut :: Outputs -> W8 -> Outputs
setDataOut (Outputs a_o _ we_o iack_o) d_o =
   Outputs a_o d_o we_o iack_o

setWeOut :: Outputs -> Bit -> Outputs
setWeOut (Outputs a_o d_o _ iack_o) we_o =
   Outputs a_o d_o we_o iack_o

setIackOut :: Outputs -> Bit -> Outputs
setIackOut (Outputs a_o d_o we_o _) iack_o =
   Outputs a_o d_o we_o iack_o
```

In fact, the CPU source code consists of a large number of such getter and setter functions. The construction of these functions is entirely regular, so henceforth only the type signatures of getter and setter functions will be given. It should be noted that Haskell has a convenient *record syntax* that makes manipulation of types like these much easier, and eliminates the need for boilerplate getter and setter definitions. This is not currently implemented in the ReWire compiler, but since record syntax is essentially just syntactic sugar for the kinds of getters and setters we are defining here, it should not be too difficult to implement it in the future.

5.2.2 CPU State

Internally, the CPU will carry around a handful of state variables, which will be stored in a state monad. We bundle these variables together in a single record type called `CPUState`.

```
data CPUState = CPUState Inputs Outputs
                         Bit Bit Bit W8
                         Bit Bit W8
                         W8 W8 W8 W8
```

In order, the fields are:

1. `inputs :: Inputs`: Stores the input value received at the beginning of the current clock cycle.

2. `outputs :: Outputs`: Stores the output value to be generated for this clock cycle. (This value may be updated multiple times in the course of a clock tick; the `tick` function defined below will simply output the final value.)

3. `zFlag, cFlag, ieFlag :: Bit`: Zero flag, carry flag, interrupt-enable flag.

4. `pc :: W8`: Program counter.

5. `zsFlag, csFlag :: Bit and pcSave :: W8`: "Save" spaces for the zero flag, carry flag, and program counter. The values of `zFlag`, etc. will be copied here when an interrupt is received.

6. `r0,r1,r2,r3 :: W8`: General purpose registers.

We define getter and setter functions for the fields of type `CPUState` with the following signatures.

```
inputs              :: CPUState -> Inputs
outputs             :: CPUState -> Outputs
zFlag,cFlag,ieFlag :: CPUState -> Bit
pc                  :: CPUState -> W8
zsFlag,csFlag       :: CPUState -> Bit
pcSave              :: CPUState -> W8
r0,r1,r2,r3         :: CPUState -> W8

setInputs                 :: CPUState -> Inputs -> CPUState
setOutputs                :: CPUState -> Outputs -> CPUState
setZFlag,setCFlag,setIEFlag
                          :: CPUState -> Bit -> CPUState
setPC                     :: CPUState -> W8 -> CPUState
setZSave,setCSave         :: CPUState -> Bit -> CPUState
setPCSave                 :: CPUState -> W8 -> CPUState
setR0,setR1,setR2,setR3   :: CPUState -> W8 -> CPUState
```

Finally, it will be convenient to have an enumerated type that we may use to name individual general purpose registers.

```
data Register = R0 | R1 | R2 | R3
```

5.2.3 VHDL Foreign Functions

A number of low-level functions dealing with bits, words, etc. are implemented as foreign functions in VHDL. First, we define a number of bitwise logical operators.

```
vhdl notBit   :: Bit -> Bit
vhdl eqBit    :: Bit -> Bit -> Bit
vhdl andBit   :: Bit -> Bit -> Bit
vhdl orBit    :: Bit -> Bit -> Bit
vhdl xorBit   :: Bit -> Bit -> Bit
```

Second, we define some logical and arithmetic operators over eight-bit words.

```
vhdl notW8    :: W8 -> W8
vhdl andW8    :: W8 -> W8 -> W8
vhdl orW8     :: W8 -> W8 -> W8
vhdl xorW8    :: W8 -> W8 -> W8
vhdl eqW8     :: W8 -> W8 -> Bit
vhdl rolW8    :: W8 -> W8
vhdl rorW8    :: W8 -> W8
vhdl plusCW8  :: W8 -> W8 -> Bit -> (Bit,W8)
vhdl plusW8   :: W8 -> W8 -> (Bit,W8)
vhdl negW8    :: W8 -> W8 -> W8
vhdl minusCW8 :: W8 -> W8 -> Bit -> (Bit,W8)
vhdl shlCW8   :: W8 -> Bit -> (Bit,W8)
vhdl shrCW8   :: W8 -> Bit -> (Bit,W8)
vhdl msbW8    :: W8 -> Bit
vhdl lsbW8    :: W8 -> Bit
```

Note that the arithmetic `plusW8`, `plusCW8`, and `minusCW8` functions take a carry-in bit as a third argument in addition to their operands. Furthermore, `plusCW8` and `minusCW8` produce a carry-out bit as part of their return values. Similar remarks apply to `shlCW8` and `shrCW8`.

Each of the functions defined here certainly *could* be implemented directly in ReWire. There are two reasons that using VHDL primitives is profitable, however. First, where the arithmetic functions are concerned, coding these functions in VHDL allows us to use VHDL's built-in arithmetic operations for addition and subtraction, which can be compiled into whatever sort of adder is most efficient for the target platform. This frees us from having to make decisions about (say) whether to implement addition with a carry-lookahead adder or with a ripple-carry adder. Second, the VHDL code generated by ReWire for "deep bit fiddling" functions is often somewhat difficult for VHDL synthesis tools to digest. For large circuits this can result in excessively long synthesis times. The latter problem could probably be remedied with some more work on back-end code generation. In any case, a more fully developed version of ReWire would probably have a large library of functions like these as part of its standard library, or even implement them as language primitives.

5.2.4 CPU Monad

The monad in which our CPU design will live is simply the layering of `ReactT Inputs Outputs`—meaning our design will have inputs of type `Inputs` and outputs of type `Outputs`—on top of `StateT CPUState Identity`, giving an internal mutable state of type `CPUState`.

```
type CPU = ReactT Inputs Outputs (StateT CPUState Identity)
```

To access and update the mutable state, we define some simple functions in the monad.

```
getState :: CPU CPUState
getState = lift get

putState :: CPUState -> CPU ()
putState s = lift (put s)
```

To ease access to the individual fields of the current CPU state, we create monadic functions to "get" and "put" the current values of individual fields. For example, the functions associated with the program counter are as follows.

```
getPC :: CPU W8
getPC = do s <- getState
           return (pc s)

putPC :: W8 -> CPU ()
putPC pc = do s <- getState
              putState (setPC s pc)
```

Since the shape of these functions is quite regular, we will again give only the type signatures for them.

```
getInputs                    :: CPU Inputs
getOutputs                   :: CPU Outputs
getZFlag,getCFlag,getIEFlag  :: CPU Bit
getPC                        :: CPU W8
getZSave,getCSave            :: CPU Bit
getPCSave                    :: CPU W8
getReg                       :: Register -> CPU W8
getDataIn                    :: CPU W8

putInputs                    :: Inputs -> CPU ()
putOutputs                   :: Outputs -> CPU ()
putZFlag,putCFlag,putIEFlag  :: Bit -> CPU ()
putPC                        :: W8 -> CPU ()
putZSave,putCSave            :: Bit -> CPU ()
putPCSave                    :: W8 -> CPU ()
putReg                       :: Register -> W8 -> CPU ()
putDataOut                   :: W8 -> CPU ()
putAddrOut                   :: W8 -> CPU ()
putWeOut                     :: Bit -> CPU ()
putIackOut                   :: Bit -> CPU ()
```

5.2.5 Instruction Fetch, Decode, and Execute

We now come to the main fetch-decode-execute loop. The loop proceeds by reading the current input, and first checking whether the reset or interrupt line is high. If the reset signal is high, a tail call to the `reset` function is made. If the interrupt line is high and interrupts are currently enabled, a tail call to the `interrupt` function is made. Otherwise we proceed to execute the instruction that has just arrived on the data bus. The instruction decode logic comprises the bulk of this function. It takes the form of a large sequence of pattern matches, dispatching in each case a handler function that corresponds to each machine instruction. There is a clean, one-to-one correspondence between the instruction mnemonics and the instruction handling functions. There is, however, a bit of inefficiency here, to be discussed in Section 5.3.

```
loop :: CPU ()
loop = do
  inp <- getInputs
  case rstIn inp of
    1 -> reset
    0 -> case (ie,intIn inp) of
      (1,1) -> interrupt
      _     -> do
        incrPC
        case dataIn inp of
          W8 0 0 0 0 rEn wEn b0 b1 ->
                                mem rEn wEn (r b0 b1)
          W8 0 0 0 1  b0  b1 c0 c1 ->
                                ld (r b0 b1) (r c0 c1)
          W8 0 0 1 0  b0  b1 c0 c1 ->
                                st (r b0 b1) (r c0 c1)
          W8 0 0 1 1  b0  b1 c0 c1 ->
                                add (r b0 b1) (r c0 c1)
          W8 0 1 0 0  b0  b1 c0 c1 ->
                                addc (r b0 b1) (r c0 c1)
          W8 0 1 0 1  b0  b1 c0 c1 ->
                                sub (r b0 b1) (r c0 c1)
          W8 0 1 1 0  b0  b1 c0 c1 ->
                                subb (r b0 b1) (r c0 c1)
```

```
W8 0 1 1 1  b0  b1 c0 c1 ->
                         mov (r b0 b1) (r c0 c1)
W8 1 0 0 0  b0  b1 c0 c1 ->
                         or (r b0 b1) (r c0 c1)
W8 1 0 0 1  b0  b1 c0 c1 ->
                         and (r b0 b1) (r c0 c1)
W8 1 0 1 0  b0  b1 c0 c1 ->
                         xor (r b0 b1) (r c0 c1)
W8 1 0 1 1  b0  b1 c0 c1 ->
                         cmp (r b0 b1) (r c0 c1)
W8 1 1 0 0 0 0 b0 b1 ->
                         brz (r b0 b1)
W8 1 1 0 0 0 1 b0 b1 ->
                         brnz (r b0 b1)
W8 1 1 0 0 1 0 b0 b1 ->
                         brc (r b0 b1)
W8 1 1 0 0 1 1 b0 b1 ->
                         brnc (r b0 b1)
W8 1 1 0 1 0 0 b0 b1 ->
                         jmp (r b0 b1)
W8 1 1 0 1 0 1 0 b0 ->
                         ien b0
W8 1 1 0 1 0 1 1 0 ->
                         iack
W8 1 1 0 1 0 1 1 1 ->
                         iret
W8 1 1 0 1 1 0 b0 b1 ->
                         not (r b0 b1)
W8 1 1 0 1 1 1 b0 b1 ->
                         clrr (r b0 b1)
W8 1 1 1 0 0 0 b0 b1 ->
                         incr (r b0 b1)
W8 1 1 1 0 0 1 b0 b1 ->
                         decr (r b0 b1)
W8 1 1 1 0 1 d b0 b1 ->
                         rot d (r b0 b1)
W8 1 1 1 1 l d b0 b1 ->
                         shft l d (r b0 b1)
_ ->                     reset
```

```
         pc <- getPC
         putAddrOut pc
         loop
   where r :: Bit -> Bit -> Register
         r 0 0 = R0
         r 0 1 = R1
         r 1 0 = R2
         r 1 1 = R3
```

5.2.6 Instructions

We may now define the exact implementation of each individual instruction. First we will define a couple of helper functions. The `tick` function delineates the end of one clock cycle by signaling the output value (which has been built up in a state monad) and storing the next input for use later in this cycle.

```
tick :: CPU ()
tick = do o <- getOutputs
          i <- signal o
          putInputs i
```

We will also define a helper to increment the program counter.

```
incrPC :: CPU ()
incrPC = do pc <- getPC
            putPC (snd (plusW8 pc oneW8))
```

Our CPU's `mem` instruction has a somewhat unconventional semantics that implements both load and store operations. On the first clock cycle, `mem` will read the immediate source/destination address (i.e. the second byte of the instruction) from memory. On the second cycle, `mem` will output that address to the address bus, along with the write enable bit and (if the write-enable bit is set) data from a register on the data bus. On the third cycle, in the case of a read operation, `mem` will store the resulting data into the register. (N.B., we assume the presence of an asynchronous RAM, or at least of one clocked fast enough that it will be able to respond within one CPU clock cycle. Setting both write-enable and read-enable may result in undefined behavior.)

```
mem :: Bit -> Bit -> Register -> CPU ()
mem rEn wEn r = do pc <- getPC
                   putAddrOut pc

                   a <- getDataIn

                   putAddrOut a
                   putWeOut wEn
                   case wEn of
                     1 -> do d <- getReg r
                             putDataOut d
                     0 -> return ()

                   incrPC
                   tick

                   case rEn of
                     1 -> do v <- getDataIn
                             putReg r v
                     0 -> return ()
```

The `ld` instruction loads a value from an address stored in register r_S into register r_D. The complementary `st` instruction will store the value of r_D into the address stored in r_S.

```
ld :: Register -> Register -> CPU ()
ld rD rS = do a <- getReg rS
              putWeOut 0
```

```
                putAddrOut a
                tick

                getDataIn
                putReg rD v

st :: Register -> Register -> CPU ()
st rD rS = do a <- getReg rS
              v <- getReg rD
              putWeOut 1
              putDataOut v
              putAddrOut a
              tick
```

The arithmetic instructions add, addc (add-with-carry), sub, and subb (subtract-with-borrow), and the logical operators or, and, xor, and cmp, all follow a very similar pattern. We will provide only the code for addc as a representative example.

```
addc :: Register -> Register -> CPU ()
addc rD rS = do vD              <- getReg rD
                vS              <- getReg rS
                cin             <- getCFlag
                let (cout,vD') =  plusCW8 vD vS cin
                putCFlag cout
                putReg rD vD'
                tick
```

The mov instruction implements a register-register move.

```
mov :: Register -> Register -> CPU ()
mov rD rS = do v <- getReg rS
               putReg rD v
               tick
```

Control flow instructions brz (branch on zero), brnz, brc (branch on carry), brnc, and jmp (unconditional jump) all follow a similar pattern; as an example, brz is implemented as:

```
brz :: Register -> CPU ()
brz r = do z <- getZFlag
           case z of
             1 -> do a <- getReg r
                     putPC a
             0 -> return ()
           tick
```

The ien (interrupt-enable), iack (interrupt-acknowledge), and iret (return-from-interrupt) instructions are used to implement interrupt handling.

```
ien :: Bit -> CPU ()
ien b = do putIEFlag b
           tick

iack :: CPU ()
iack = do putIackOut 1
          tick

iret :: CPU ()
iret = do putIEFlag 1
          pc <- getPCSave
          putPC pc
          z  <- getZSave
          putZFlag z
          c  <- getCSave
          putCFlag c
          tick
```

Our CPU's instruction set implements a handful of unary operators on register as well: not (logical negatiion), clrr (clear register), incr (increment register), and decr (decrement register).

```
not :: Register -> CPU ()
not r = do v <- getReg r
           putReg r (notW8 v)
           tick

clrr :: Register -> CPU ()
clrr r = do putReg r zeroW8
            tick

incr :: Register -> CPU ()
incr r = do v             <- getReg r
            let (cout,v') =  plusCW8 v oneW8 0
            putReg r v'
            putCFlag cout
            putZFlag (eqW8 v' zeroW8)
            tick

decr :: Register -> CPU ()
decr r = do v             <- getReg r
            let (cout,v') =  minusCW8 v oneW8 0
            putReg r v'
            putCFlag cout
            putZFlag (eqW8 v' zeroW8)
            tick
```

Implementation of the `rotl` and `rotr` (rotate left/right) instructions is collapsed into a single function:

```
rot :: Bit -> Register -> CPU ()
rot dir r = do v <- getReg r
               case dir of
                 0 -> putReg r (rolW8 v)
                 1 -> putReg r (rorW8 v)
               tick
```

Finally, the `shft` function encodes both arithmetic and logical shifts, in both the left and the right direction.

```
shft :: Bit -> Bit -> Register -> CPU ()
shft log dir r = do v         <- getReg r
                    (cout,v') <- case (dir,log) of
                      (0,0) -> shlCW8 v (msbW8 v)
                      (0,1) -> shlCW8 v 0
                      (1,0) -> shrCW8 v (lsbW8 v)
                      (1,1) -> shrCW8 v 0
                    putReg r v'
                    putCFlag cout
                    putZFlag (eqW8 v' zeroW8)
                    tick
```

5.2.7 Reset and Interrupt Handling

The handler for a reset event simply resets the carry and zero flags, initializes the output register to `initOutputs` (all zeros), and waits for one clock tick. (The caller, `loop`, will then return to the top of the fetch-decode-execute loop.)

```
reset :: CPU ()
reset = do putCFlag 0
           putZFlag 0
           putOutputs initOutputs
           tick
```

Interrupts cause the the current program counter, zero flag, and carry flag to be stored in a temporary, turns off the interrupt-enable bit (so that an interrupt cannot interrupt an interrupt), and transfers control flow to an interrupt handling vector at address FE_{16}. This function, like `reset`, is invoked by the fetch-decode-execute loop, which will return to the top of the loop after invocation.

```
interrupt :: CPU ()
interrupt = do putIEFlag Zero
               getPC
               z <- getZFlag
               c <- getCFlag
               putPCSave pc
               putZSave z
               putCSave c
               putPC interruptVectorAddr
               tick
```

5.2.8 Startup

The `start` function simply invokes `reset`, then transfers control to the `loop`.

```
start :: CPU ()
start = extrude begin initState
  where begin = do reset
                   loop
```

Finally, for lack of a better place, we will define a few initial constants here.

```
initOutputs :: Outputs
initOutputs = Outputs zeroW8 zeroW8 0 0

initInputs :: Inputs
initInputs = Inputs zeroW8 0 0

initState :: CPUState
initState = CPUState initInputs initOutputs 0 0 0 zeroW8 0 0
                     zeroW8 zeroW8 zeroW8 zeroW8 zeroW8
```

This completes the ReWire portion of the CPU spec. The VHDL-defined primitives are omitted here, but each consists of no more than a single arithmetic operation and/or straight-forward bit manipulation.

5.3 Evaluation

The CPU described here synthesizes successfully on Xilinx 7-series FPGAs. Unfortunately, logic utilization is rather high, and the resulting clock frequency is less than desired (several times slower, for example, than PicoBlaze on the same chip). The main reason for this is probably redundant code occurring in (for example) each arithmetic expression. In Section 8.2.3, a possible remedy for this problem is explored.

5.4 Extending the CPU with Multiple Security Domains

In this section we will sketch how to extend the CPU design given here in order to enforce separation of security domains. Our methodology is derived from a set of techniques developed originally by Harrison [**?**] based on layered state monad. Specifically we will derive a single-core CPU where processes in multiple security domains share an execution unit, yet information flows between the security domains are provably absent. We assume that there are two independent input and output channels present, and that an external "mode" bit input determines whether the next instruction is to be executed in domain A or B.

5.4.1 Modifying the Monad

The first step in implementing multiple security domains is to alter the monad to produce a layered state monad. We will add a second layer of *CPUState* for the second security domain, and a third state layer containing an internal bit that will track whether the CPU is operating in security domain A or B. We will then modify the output channel by changing it to the type *Maybe* (*Either Outputs Outputs*) which reflects three possibilities: (1) no output is available (this happens when the CPU is querying the mode bit); (2) output for domain A is available; or (3) output for domain B is available. We make a similar mutation to the input type while also adding a line for the mode bit (of type *Domain*).

```
data Domain = A | B
type CPU    = ReactT (Domain,Inputs,Inputs)
                     (Maybe (Either Outputs Outputs))
                     (StateT Domain
                       (StateT CPUState
                         (StateT CPUState Identity)))
```

5.4.2 Modifying the Code

We will make a slight alteration to the CPU loop, which will take one clock tick to sample the current mode input. The rest of the loop is the same.

```
loop :: CPU ()
loop = do (dom,_,_) <- signal Nothing
          putMode dom
          ...
```

This may seem slightly overcomplicated—if the current mode input is available the circuit input, why do we need to store it separately? The answer is that there is nothing stopping the mode bit from changing *during* instruction execution. We do not wish to start overwriting domain B's registers with data that was intended for domain A just because a change in the mode bit came in the middle of instruction execution. Therefore we must sample it once, at the beginning of an instruction, and stick with that value for the remainder of instruction execution.

The instruction handler functions are identical except that we replace uses of **signal**, **get**, and **put** with calls to newly defined functions that are analogous, but examine the current mode register to determine which state domain is to be affected and which input channel is to be sampled.

```
signal' :: Outputs -> CPU Inputs
signal' o = do
  dom <- getMode
  case dom of
    A -> do (_,i,_) <- signal (Just (Left o))
            return i
    B -> do (_,_,o) <- signal (Just (Right o))
            return i

get' :: CPU CPUState
get' = do
  dom <- getMode
  case dom of
    m <- getMode
    case m of
      A -> lift (lift get)
      B -> lift get

put' :: CPUState -> CPU ()
put' s = do
  dom <- getMode
  case dom of
    A -> lift (lift (put s))
    B -> lift (put s)
```

This completes the changes to the CPU specification.

5.4.3 Correctness Property

We may now formulate a correctness property for our CPU, proving that no internal information leakage channels exist. The property is a Goguen and Meseguer [?]-style non-interference property: if we alter the input stream in one domain, the output stream in the other domain is unaffected. Formally, we can define this in terms of a (Haskell, not ReWire) function that supplies a stream of inputs to the function.

```
run :: CPU () -> [(Domain,Inputs,Inputs)] ->
                 [Maybe (Either Outputs Outputs)]
```

The program has no information flow from the security domain A to domain B if the following equality holds:

```
forall insA :: [Inputs], insA' :: [Inputs],
       insB :: [Inputs], modes :: [Domain],
  map bOuts (run start inStream)
     =
  map bOuts (run start inStream')

  where bOuts Nothing            = Nothing
        bOuts (Just (Left _))    = Nothing
        bOuts (Just (Right o)) = o

        inStream   = zip3 modes insA insB
        inStream'  = zip3 modes insA' insB
```

The property may be proved using simple monadic equational reasoning techniques that have been published previously [?, ?]. Chapter 7 gives a full account of a formal type system and logic enabling such reasoning on a calculus that is closely related to ReWire. In particular, the layered state-monad structure of the

ReWire calculus helps to establish that no *direct storage channels* exist between the high and low domains: a value that is retrieved from the low domain with **get** is never written to the high domain with **put**. This, in turn, enables the use of certain cancellation and commutativity properties that establish the essential irrelevance of what is happening in the high domain to what is happening in the low domain.

Chapter 6

Case Study II: Fast Regular Expression Matchers

This chapter is reprinted from a paper I submitted to ARC 2015 with Ian Graves, Michela Becchi, William L. Harrison, and Gerard Allwein, entitled "Hardware Synthesis from Functional Embedded Domain-Specific Languages". That paper carries the following acknowledgment: *This research was supported by the Office of the Assistant Secretary of Defense for Research and Engineering, the U.S. Department of Education under GAANN grant number P200A100053, NSF CAREER Award 00017806, and NSF award CNS-1319748.*

Abstract. Although FPGAs have the potential to bring software-like flexibility and agility to the hardware world, designing for FPGAs remains a difficult task divorced from standard software engineering norms. A better programming flow would go far towards realizing the potential of widely deployed, programmable hardware. We propose a general methodology based on domain specific languages embedded in the functional language Haskell to bridge the gap between high level abstractions that support programmer productivity and the need for high performance in FPGA circuit implementations. We illustrate this methodology with a framework for regular expression to hardware compilers, written in Haskell, that supports high programmer productivity while producing circuits whose performance matches and, indeed, exceeds that of a state of the art, hand-optimized VHDL-based tool. For example, after applying a novel optimization pass, throughput increased an average of 28.3% over the state of the art tool for one set of benchmarks. All code discussed in the paper is available online [**?**].

6.1 Introduction

FPGAs are notably difficult to program and this has motivated research into high-level synthesis (HLS) from high level programming languages and, in particular, from domain-specific languages [**?**]. This language-based approach is attractive because of its potential to make hardware engineering more like software engineering with its support for modularity, reuse, and abstraction, and thereby create a wider group of developers for programmable hardware. This paper describes a methodology for deriving performant hardware implementations directly from high-level functional embedded domain-specific languages (EDSL).

The main contribution of this research is a methodology and related tools supporting the "three P's" [**?**] for programming reconfigurable hardware: productivity, performance and portability. DSLs address the first two P's directly because domain specialization supports programmer productivity and, furthermore, allows aggressive optimization of domain-specific idioms. Portability is achieved by a retargetable back-end for producing hardware called ReWire [**?**] and because functional languages naturally lend themselves to modularity and reusability. The approach we advocate opens the world of FPGA programming to the functional programming community because the methodology operates entirely within the Haskell functional programming language and every language in the EDSL pipeline is a high-level language.

The Delite DSL compiler framework [**?**] seeks to address the "three P's" with respect to implementing software on parallel, heterogeneous systems. Delite addresses portability (i.e., retargetability of DSL compilers to a broad range of parallel hardware) through *language virtualization* (LV). ReWire is also a virtualized DSL in that it has a separate compiler backend for producing FPGA-based implementations while reusing large parts of its host language's infrastructure—including Haskell's type system, front end, etc. ReWire addresses the first two P's in much the same way as Delite. In George, et al., [**?**], the Delite framework is adapted to the generation of hardware from DSLs, specifically the hardware acceleration of kernels in a heterogeneous setting. By contrast, ReWire and the present methodology is concerned exclusively with the generation of (homogeneous) synchronous hardware circuits.

New language constructs raise issues with respect to performance. Is there a performance price to be paid and, if so, is the increased expressiveness worth it? Does the increased expressiveness enable better performance and programmer productivity? In light of these questions, we evaluate our methodology via two case

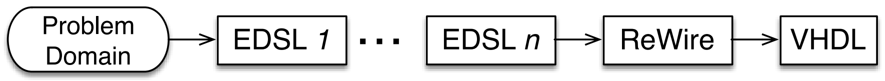


Figure 6.1: FP Methodology for HLS

studies. The case studies presented here consider a purely functional framework for REHC construction, called RexHacc (for "Regular EXpression HArdware compiler-compiler"). RexHacc is an EDSL-structured compiler-compiler, implemented in Haskell, for Perl-compatible regular expressions (PCRE) similar to those seen in popular intrusion detection systems (e.g., Snort [**?**]).

6.1.1 Overview of Methodology

The methodology factors the problem of HLS into a series of translations between EDSLs. An EDSL is a domain-specific language that is defined as a collection of constructs within an existing high level language. The methodology is illustrated in Figure 6.1. A problem domain can be realized as a DSL embedded in Haskell. DSL cross-compilers targeting ReWire enable synthesis onto an FPGA via the ReWire compiler. Section 6.2 presents a more in-depth discussion of our methodology.

The case studies (see Figure 6.2) involve regular expression to hardware compilation in which we generate artifacts that perform as well as and often better than state of the art approaches. The case studies reported here consider the problem domain of regular expression to hardware compilers (REHC) [**?**]. Following Figure 6.1, we developed a reusable and modular framework for REHC called *RexHacc* and demonstrated that circuits produced with it meet or exceed the performance of state-of-the-art REHC.

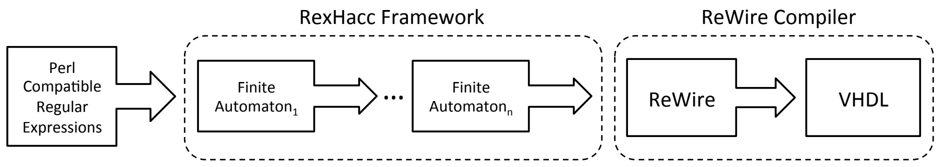


Figure 6.2: Combining the Ease of Use of Traditional EDSLs with the Power and Run-Time Performance of a Virtualized Language

The RexHacc Framework To evaluate the methodology, we performed an experiment in which we compared RexHacc to the performance of the state-of-the-art REHC of Becchi and Crowley [?] (henceforth `reg2vhdl`) against its own benchmarks. The goal is to demonstrate both the productivity gain and high performance achievable via our methodology in the construction and testing of compilers generated by RexHacc. This section is deliberately high-level. We suppress the definitions of functions and data types; the code is online [?].

The entry point for RexHacc is the function `rexhacc` with Haskell type:

```
rexhacc :: (NFA a -> NFA a) -> RegEx a -> ReWire
```

The declaration form "`::`" is pronounced "has type". The function `rexhacc` takes two inputs, an optimization function (of type `NFA a -> NFA a`) as well as a regular expression (of type `RegEx a`). The type `NFA a` (resp., `RegEx a`) represents non-deterministic finite automata (resp., regular expressions) over an alphabet of type `a`. A regular expression compiler is generated with RexHacc by applying the top-level `rexhacc` function to an optimization pass, `opt`:

```
            compiler :: RegEx a -> ReWire
(‡)         compiler = rexhacc opt
                  where opt = (o₁ . ··· . oₙ)
```

Each o_i is an optimization pass of type `NFA a -> NFA a`, all of which are composed using Haskell's function composition operator (i.e., the infix "`.`") into a single

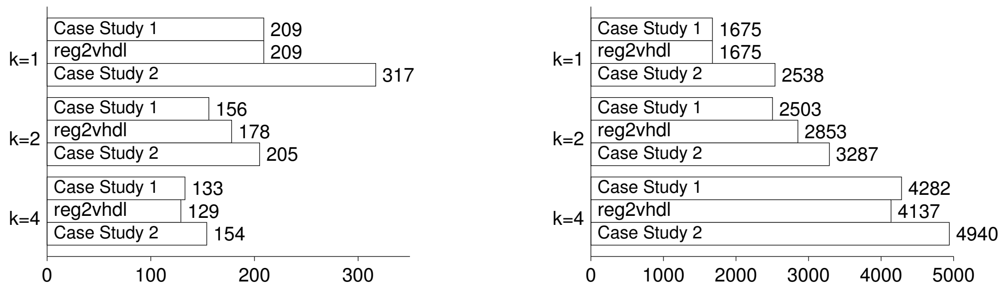


Figure 6.3: Writing Domain-Specific Optimizations for Case Studies. The `tcp25` benchmark: (left) Maximum frequency (MHz) and (right) throughput in Mbits/sec. Parameter `k` indicates stride length (defined in Section 6.4). Case study 2 shows an average of 28.3% throughput increase over `reg2vhdl`.

pass. This composition corresponds to the middle box in Figure 6.2 and each $\texttt{o}_i$ is a phase inside that box. The generated `compiler` takes a regular expression over an alphabet of type `a` and converts it into an `NFA a`, which is then fed to the optimization pass `opt`. The optimization pass produces an `NFA a` from which ReWire code is generated. The ReWire output from this compiler can either be translated into VHDL by the ReWire compiler or executed as software in any standard Haskell environment.

Summary of Case Study Results Secs. 6.4 and 6.5 each describe the definition of an REHC in the RexHacc framework. Each case study was tested against `reg2vhdl` using existing test suites [**?**] with respect to standard metrics for circuit size, clock speed and throughput (see Figure 6.3). The first case study (Section 6.4) implements the same optimization passes as `reg2vhdl`, and it was clear that this compiler generally matched or exceeded the performance of the hand-optimized compiler `reg2vhdl` with a tiny increase in circuit size. It was observed that one of the benchmarks (`tcp25`) seemed to be particularly challenging for both the first case study compiler and `reg2vhdl` with respect to throughput. This observation motivated the second case study (Section 6.5), which improves on the first with an (apparently novel) optimization pass that results in better performance than `reg2vhdl` on the `tcp25` benchmark.

6.2 A Methodology for Synthesis from Functional EDSLs

Synthesis from pure functional languages (e.g., Haskell, `www.haskell.org`) is appealing because combinational hardware is functional in nature, functional languages have powerful features supporting programmer productivity (e.g., modularity, expressive data types, static type inference, etc.), and the absence of side effects (e.g., destructive update) simplifies synthesis. But general purpose functional languages also contain a number of features that cannot be represented in hardware (e.g., general recursion and garbage collection) and this makes HLS directly from existing functional languages more challenging.

ReWire [**?**] is a proper *sublanguage* of Haskell—i.e., any ReWire program is a Haskell program, but not all Haskell programs are ReWire programs. ReWire programs, in contrast with general purpose functional languages like Haskell, are always synthesizable to hardware. ReWire restricts Haskell by disallowing the use of higher-order functions and general recursion at runtime (though techniques like partial evaluation may enable their use at compile time). RexHacc uses the ReWire hardware compiler as a back-end for producing VHDL implementations.

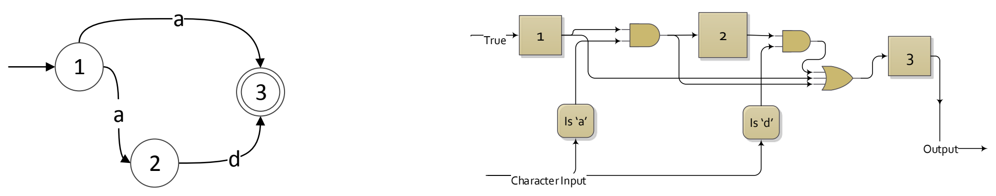


Figure 6.4: NFA and Corresponding Sidhu and Prasanna-style Implementation

6.2.1 Front End

The RexHacc compilation process begins with a collection of regular expressions written in Perl-compatible regular expression (PCRE) syntax. We use the parser combinator library Parsec in Haskell to parse the regular expressions in the source file. The regular expression is converted to the `NFA` type via a textbook translation of regular expressions to NFAs [?]. The resulting `NFA` is passed to the optimization portion of the compilation chain.

6.2.2 Simulating Circuits in Haskell

Because ReWire is a sublanguage of Haskell, we can execute ReWire code as software in any Haskell environment with a test harness for executing reactive resumptions. Figure 6.5 demonstrates the process of generating ReWire from regular expressions (lines 1-4) and compiling the generated ReWire to VHDL and to Haskell for debugging (lines 5-6). The ReWire compiler generates a Haskell file (Debug.hs) that is imported by the test harness (Harness.hs) for simulation. We load the test harness in GHCI, an interactive Haskell shell, on line 7 of Figure 6.5. The harness has a function called `test` that simulates a regular expression `matcher` with a string (line 8). What results is a list that represents the stream of output bits from `matcher` (line 9). These indicate the accept value of the device after reading each character

```
$ cat regex.pcre
a*bcde
$ rewire-regex regex.pcre
ReWire Output written to output.rw
$ rewire output.rw
Debug -> Debug.hs; VHDL -> output.vhd
$ ghci Harness.hs
ghci> test matcher "aaabcde"
["0","0","0","0","0","0","1"]
```

Figure 6.5: Simulation in Haskell

of input. The final character in the input results in a match and so the final value in the list of outputs is "1".

6.3 Related Work

The conversion of sets of regular expressions into NFAs is a well-known procedure [?]. Sidhu and Prasanna [?] have proposed an efficient FPGA implementation of NFAs. Their solution is based on the one-hot encoding scheme; the use of an NFA representation avoids the $O(2^n)$ space complexity that is characteristic of DFA (deterministic finite automata) representations, typically adopted in memory-based regular expression matching implementations [?, ?, ?, ?]. Subsequent efforts on FPGA [?, ?, ?, ?] have refined Sidhu and Prasanna's implementation and achieved gigabit/sec processing throughputs on real-world pattern sets.

There are a number of efforts to apply ideas and techniques from functional programming to hardware design and synthesis. Arvind [?] describes the Bluespec synthesis language as "a relatively simple DSL (GAAs *[Guarded Atomic Actions]* and modules) with a fully functioning Haskell-like meta programming layer on top." The methodology advocated here employs metaprogramming as well, in that ReWire programs (which are also Haskell programs) are ultimately produced by the `rexhacc` function. Within the Haskell community, perhaps the most well known system for hardware synthesis is Lava [?]. Lava is a domain-specific language for hardware specification embedded in Haskell. Primitives in Lava are essentially structural and specify circuits at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, and relies on an abstract set of behavioral primitives. The primary motivation for developing ReWire is as a vehicle for the design, implementation, and formal verification of high assurance hardware.

Cλash [?], is a compiler for a subset of Haskell to VHDL. Like ReWire, Cλash uses Haskell itself as a source language. Cλash requires some limits be placed on the kinds of algebraic data types used as well as the basic operating types. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [?]. The current research demonstrates that the ReWire compiler works at scale as the generated ReWire programs are on the order of 100K LOC. Great care was taken in the design of ReWire so that it possesses a rigorous denotational semantics to support formal verification while maintaining synthesizability for all of its programs.

6.4 Case Study 1: Matching State of the Art

We undertake the construction of a tool equivalent in functionality to the state of the art [?] (`reg2vhdl`) and to examine the feasibility of duplicating this functionality with our approach. The purpose of this case study is to demonstrate the *ease* with which such a tool can be constructed. The optimizations were chosen to match those of Becchi and Crowley [**?**] and include *head zipping*, *striding*, *alphabet compression*, and *epsilon elimination*. These results indicate that the `rexhacc`-based compiler compares favorably to and often surpasses `reg2vhdl` where throughput is concerned, and area utilization is similarly competitive. Each optimization phase was implemented in a few dozen lines of Haskell code; this is a rough indication that the amount of programmer effort required is small.

Head zipping. Head zipping is a transformation that merges outbound transitions from a state that have the same transition labels. Nodes with more than one inbound transition are not head zipped because this would result in a non-equivalent NFA. Head zipping is performed by merging the destination nodes of the matching transitions into one node that includes all of the outbound transitions from the merged nodes.

Striding. Striding is an optimization pass that doubles the number of characters an NFA matches at each transition. Striding traverses the graph's edges and looking two transitions ahead from each state, converting each two-transition sequence in the original NFA to a single transition consuming two characters.

Alphabet compression. Alphabet compression is a technique that increases sharing of logic by exploiting the identical treatment of different characters by an NFA. If two characters always result in the same transitions between all states, then these characters can be compressed into a single character class.

Epsilon elimination. Eliminating ϵ-transitions reduces the complexity and size of NFAs and simplifies code generation. NFAs with ϵ-transitions allow state transitions without consuming input. States connected to an NFA solely by ϵ-transitions can be eliminated. Eliminating unnecessary states reduces the number of flip flops required to implement the NFA on an FPGA. A textbook ϵ-elimination algorithm is used [?].

6.4.1 Experiments and Evaluation

To test the performance of RexHacc, we selected three benchmark sets of regular expressions from the literature [?, ?]. `Snort24` is a set of 24 regular expressions drawn from the Snort network intrusion detection system [?]. `Tcp25` is a set of 79 regular expressions designed to match malicious SMTP traffic, also drawn from the Snort NIDS. `Bro217` is a set of 217 regular expressions drawn from the Bro NIDS [?]. Matchers for each of these benchmarks were generated using `reg2vhdl`, as well as RexHacc. Each benchmark was tested at stride lengths $k = 1$, $k = 2$, and $k = 4$, producing circuits that consume input streams at one, two, and four bytes per clock cycle. The resulting VHDL was then synthesized using Xilinx's XST synthesis tool for the Xilinx Spartan-3E X3CS500E FPGA, speed grade -4.

Figure 6.6 compares the resulting circuits in terms of three performance metrics: (a) logic slice utilization, (b) LUT utilization, and (c) maximum throughput as measured in megabits per second. (Flip flop utilization was extremely close between the two tools and thus is not shown.) RexHacc compares favorably with `reg2vhdl` on virtually all fronts.

Throughput. RexHacc matches or exceeds `reg2vhdl`'s total throughput for all but one of the nine benchmarks. In the best case (benchmark `bro217`, $k = 1$) throughput is around 60% higher. In the worst case (benchmark `tcp25`, $k = 2$) throughput is

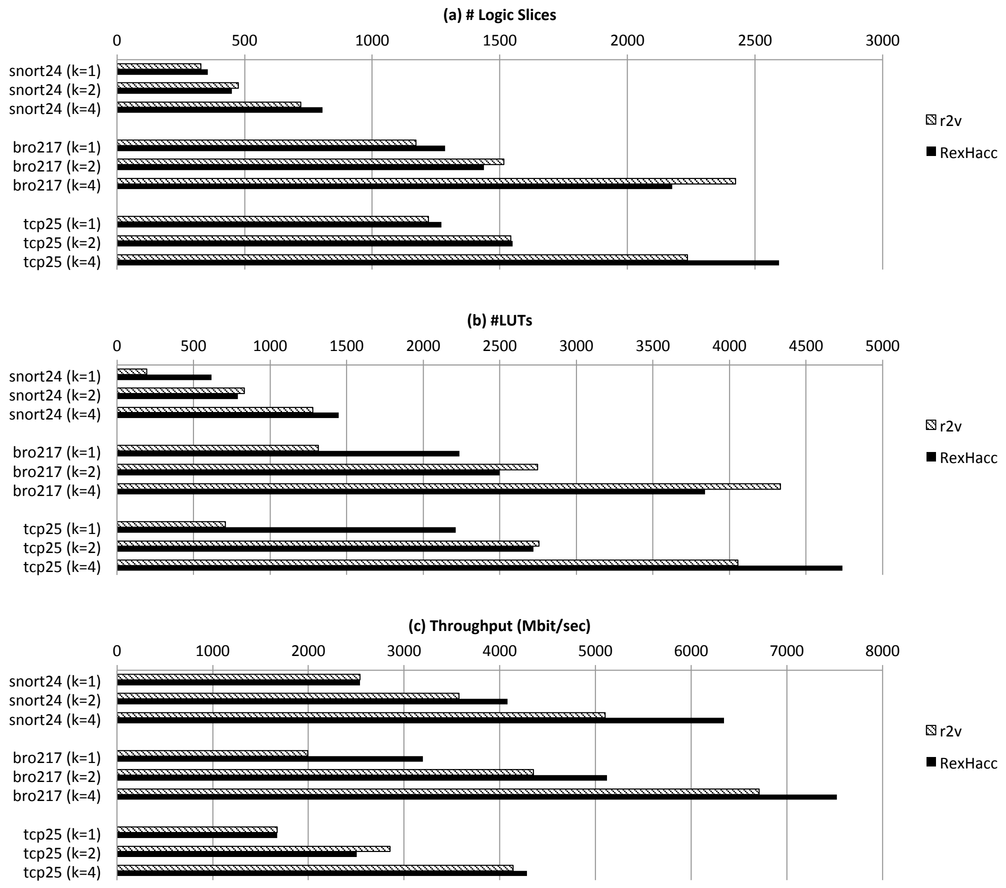


Figure 6.6: Performance Comparisons of RexHacc to `reg2vhdl`

around 13% lower. Both tools, in all cases, are capable of processing input at a rate of more than 1 Gbit/sec. In the best case, RexHacc is capable of handling input rates up to 7.5 Gbit/sec on a Xilinx Spartan-3E FPGA at a relatively low clock rate. Tests on a Xilinx 7-series platform (not presented here, but available online [**?**]) indicate that throughputs of up to 25 Gbit/sec are achievable with a more modern FPGA.

Logic utilization. With the exception of the single-strided ($k = 1$) benchmarks, LUT utilization for RexHacc-generated circuits ranged from 88% to 116% of their `reg2vhdl` counterparts. In the specific case where $k = 1$, RexHacc tends to produce circuits with higher LUT counts (up to 219% higher), suggesting that the combinational next-state logic produced by the RexHacc code generator is more complicated for these circuits. For all benchmarks, flip flop utilization for RexHacc was close to, but slightly higher than, the results generated by `reg2vhdl`. This is not surprising since each state in the NFA is represented by a single flip flop, and both tools tend to generate similar numbers of NFA states. RexHacc, however, pays a small penalty here, because it generates output signals synchronously, storing them in flip flops, while `reg2vhdl` does not. Please note, however, that the choice of synchronous outputs rather than asynchronous ones is optional in the most recent version of ReWire.

The results exhibited here suggest that the case study compiler is competitive with the state of the art. The extra flexibility of the modular, purely functional design does not come at a prohibitive cost in terms of circuit size, and indeed brings substantial benefits with respect to throughput.

6.5 Case Study 2: Surpassing State of the Art

In this case study, we demonstrate the *agility* of the RexHacc approach by identifying an opportunity for an optimization, and rapidly implementing that optimization as a compiler phase in RexHacc. The modular nature of RexHacc made it easy both to identify a key performance bottleneck, and to implement a new optimization pass to address it.

Identifying the bottleneck. While conducting the experiments of Section 6.5, we noticed that one of the benchmarks, `tcp25`, stood out for its relatively low maximum throughput when processed by RexHacc as well as by `reg2vhdl`. While striding enabled our compiler to produce circuits with maximum throughput in excess of 6 Gbit/sec for `snort24` and `bro217`, maximum throughput for `tcp25` just barely exceeded 4 Gbit/sec. The throughput advantage over `reg2vhdl` observed for `snort24` and `bro217` was essentially nonexistent for `tcp25`.

To explore the reasons for this, we instrumented our compiler pipeline by using the Haskell Functional Graph Library's built-in support for generating graph visualizations via GraphViz (www.graphviz.org). We observed that the `tcp25` NFA exhibited a structural feature that was not present in the `snort24` and `bro217` NFAs. Specifically, the `tcp25` NFA contained one state that had a large number of inbound transitions. A simplified example of this problem is exhibited in Figure 6.7 (top), where state 9 has eight inbound transitions. A large number of inbound transitions emerges when the source regular expression contains a long chain of choice operators. This pattern is not uncommon in packet inspection rulesets (e.g., consider a long chain of alternative filenames followed by the common suffix ".exe").

In the circuit implementation the inbound transitions translate to a large fan-in of signals that must be ORed together to determine whether to activate that state. As the size of this fan-in grows large, the combinational logic involved begins to dominate the critical path of the circuit. The result is a sharp reduction in maximum operating clock frequency, and therefore throughput. This suggested an opportunity for optimization: namely, to transform the NFA in such a way as to reduce the number of inbound transitions to heavily-loaded states.

Implementing an Optimization. To test our hypothesis, we extended the compiler of Section 6.4 with an optimization called *state splitting*. Suppose we have in our NFA a state s with inbound transitions $e_1, \cdots, e_n$, and assume without loss of generality that s has no self-loops. Observe that we can produce an *equivalent* NFA by "splitting" s in two: that is, introducing a new state (call it s'), and reassigning half of the inbound transitions (say, $e_1, \cdots, e_{\lceil n/2 \rceil}$) to s' instead of s. State splitting works by applying this transformation to each node whose indegree exceeds a certain fixed threshold t. Figure 6.7 (bottom) illustrates the results of applying state splitting to the NFA for $t = 2$. N.b., the maximum indegree has been reduced from 8 to 2 in this example.

The reader may note that this optimization may have the effect of *increasing* the number of inbound transitions for successor states of split nodes. This is generally not a problem for two reasons: first, as long as state splitting succeeds in reducing the *maximum* indegree, it is likely to pay off even if some states see their number of inbound transitions increased. Second, state splitting may be iterated; if the splitting of state s_1 results in state s_2 exceeding the split threshold, s_2 itself may be split.

The full code for the state-splitting optimization, consisting of 17 lines of code, is given as the `splitStates` function in the code base [**?**]. We can insert the state-splitting into the optimization pipeline simply by adding an extra phase to the `rexhacc` call; this is an instance of (‡) from Section 6.1:

```
rhcc2 :: Int -> Int -> RegEx a -> ReWire
rhcc2 k m = rexhacc
              (alphabetCmpr k . stride k . splitStates m .
                 headZip . epsElim)
```

Experimental Results. To test the state-splitting optimization, we repeated the experiments described above for the `tcp25` benchmark with state splitting enabled; the

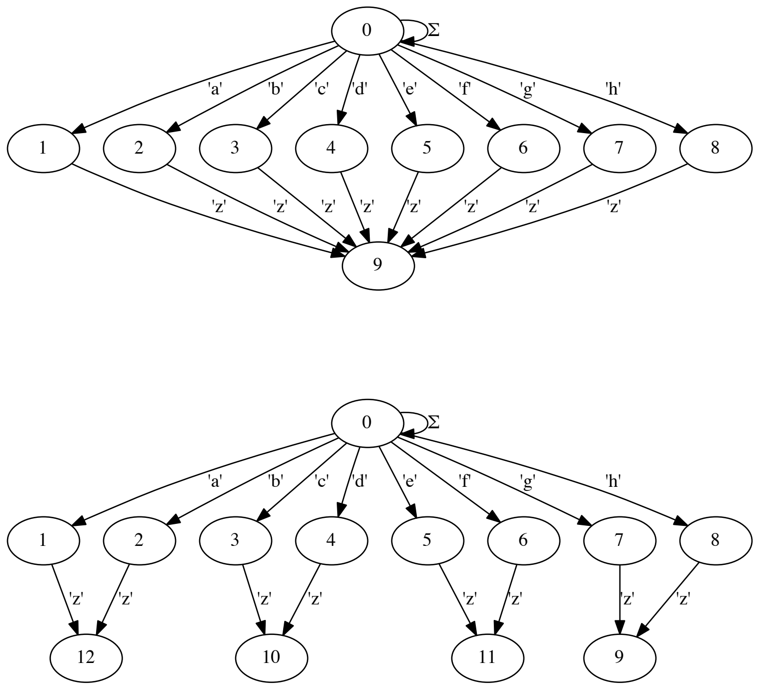


Figure 6.7: NFA for $(a|b|c|d|e|f|g|h)z$, Before State Splitting (top) and After (bottom)

in-degree threshold for state splitting was set to 3. We chose to test only against the `tcp25` benchmark for the simple reason that the other two benchmarks did not have any states with a large number of predecessors; as such, state splitting would have precisely zero effect on those benchmarks. The results are reported in Figure 6.8. With state splitting enabled, RexHacc now exceeds the maximum throughput of `reg2vhdl` on the previously troublesome `tcp25` benchmark, with throughput gains (Figure 6.8(e)) of 51%, 15%, and 19% relative to `reg2vhdl` for stride values of $k = 1$, $k = 2$, and $k = 4$ respectively. The cost in area is relatively modest, as illustrated by the comparable numbers of logic slices, LUTs, and flip flops (Figure 6.8(a-c)).

We also measured the effect of moving state-splitting to the end of the pipeline, performing it after, rather than before striding. Although this had minimal impact on the performance, it does illustrate the ease of experimenting with optimization orders in RexHacc.

6.6 Conclusions and Future Work

This research is a substantial case study utilizing the ReWire compiler at scale. ReWire is a subset of Haskell limited in expressive power to ensure the synthesizability of every ReWire program. There is a potential drawback to such restrictions: it excludes many powerful functional programming idioms. Can we maintain the sufficient expressiveness to support design while guaranteeing programs yield high-performance FPGA implementations? The key contribution of our methodology is an affirmative answer to this question.

The methodology leverages the intrinsic power of Haskell and functional programming. RexHacc is modular and customizable in the sense that optimization passes can be easily added and removed. Because the ordering of passes is exposed as function composition in Haskell, experimentation with optimization ordering is enabled. A RexHacc-generated compiler can be instrumented in a straightforward manner as we did with GraphViz and take advantage of existing external Haskell tools.

The flexibility of the RexHacc framework derives from the cross-compilation to ReWire and the ability of ReWire to generate VHDL synthesizable to efficient circuits. The methodology we have introduced lowers the barrier to entry for reconfigurable computing for functional programmers. At the same time, it provides an opportunity for hardware designers to leverage the power of the functional paradigm to improve productivity. The choice of a purely functional language does not come at a performance cost: our benchmarking demonstrates that we match or exceed the performance of a state-of-the-art hand-tuned compiler for a number of real-world tests.

The two research directions we are pursuing have to do with increasing the expressiveness of the type system to support metaprogramming and hardware security. The current methodology is based on metaprogramming (i.e., ReWire/Haskell programs are generated by Haskell programs) and there are type systems for staged programming (e.g., MetaML [**?**]) that we believe will improve programmer productivity further while automatically enforcing type safety. We developed a type system for enforcing fault isolation on ReWire [**?**] and we are currently extending to information flow security.

6.7 Acknowledgments

The authors would like to thank David Andrews of the University of Arkansas and Jason Agron of Intel Corporation for their helpful feedback.

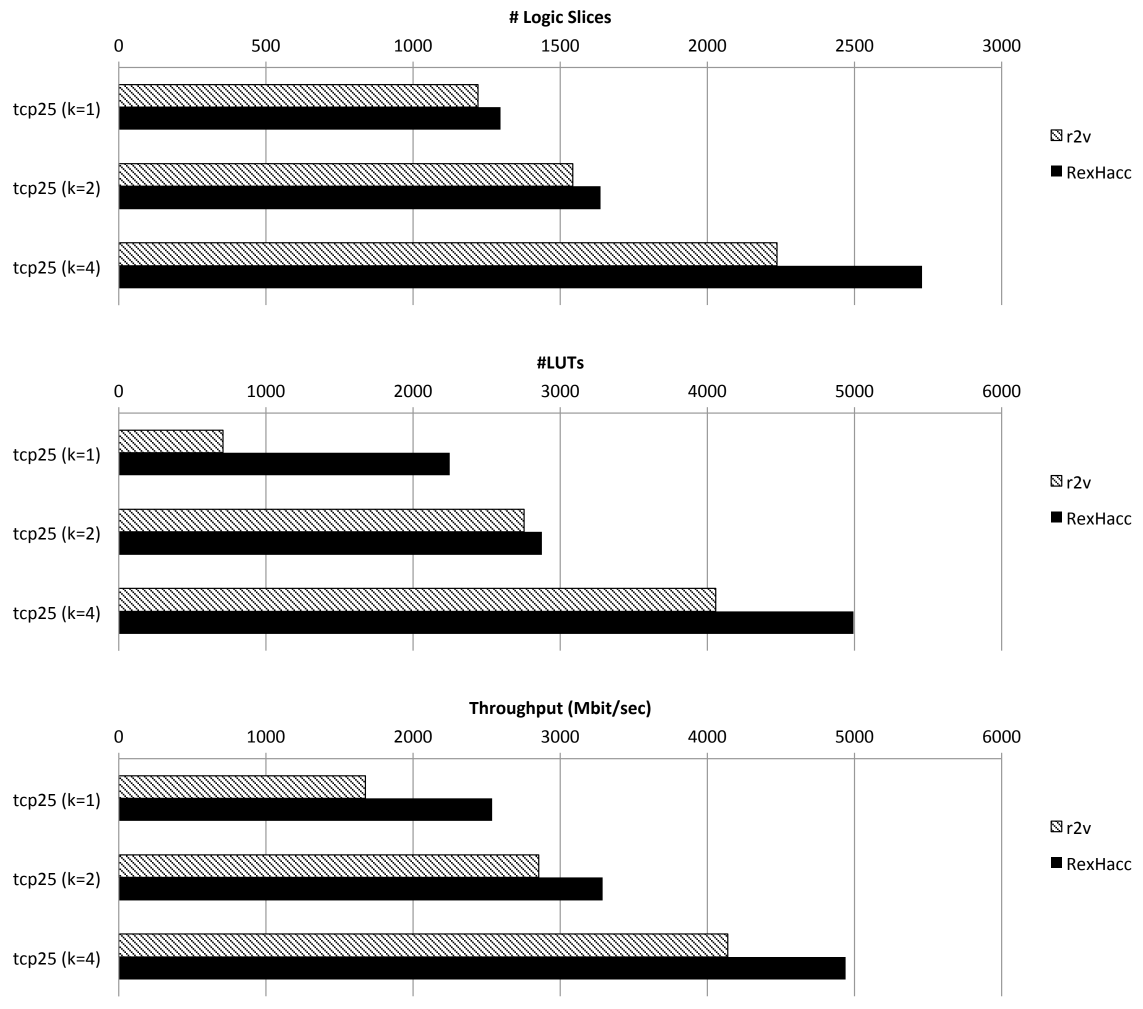


Figure 6.8: Comparisons of RexHacc with State Splitting Enabled to `reg2vhdl`

Chapter 7

Verification Techniques

This chapter is reprinted from a paper I published with William L. Harrison and Gerard Allwein at ICFEM 2012 entitled "The Confinement Problem in the Presence of Faults". That paper carried the following acknowledgment: *This research was supported by NSF CAREER Award 00017806, US Naval Research Laboratory Contract 1302-08-015S, and by the U.S. Department of Education GAANN grant no. P200A100053.*

Abstract. In this paper, we establish a semantic foundation for the safe execution of untrusted code. Our approach extends Moggi's computational λ-calculus in two dimensions with operations for asynchronous concurrency, shared state and software faults and with an effect type system à la Wadler providing fine-grained control of effects. An equational system for fault isolation is exhibited and its soundness demonstrated with a semantics based on monad transformers. Our formalization of the equational system in the Coq theorem prover is discussed.

We argue that the approach may be generalized to capture other safety properties, including information flow security.

7.1 Introduction

Suppose that you possess an executable of unknown provenance and you wish to run it safely. The cost of analyzing the binary is prohibitive, and so, ultimately, you have little choice but to explore its effects by trial and error. That is, you run it and hope that nothing irreversibly damaging is done to your system. There are two alternatives proposed in the literature to the trial and error strategy. You can attempt to detect safety and security flaws in the untrusted code with automated static analyses. This is the approach being explored by much of the literature from the language-based security [?] community. The other approach is to isolate the untrusted code so that any destructive side effects (malicious or otherwise) resulting from its execution are rendered inert.

This paper introduces the *confinement calculus* (CC) and uses it as a vehicle for exploring the design and verification of isolation kernels (defined below). CC extends Moggi's computational λ-calculus [?] with constructs for state, faults and concurrency. Furthermore, the type system for the CC also incorporates an effect system à la Wadler [?] to distinguish computations occurring on different domains. The CC concurrency metalanguage is closely related to recent work of Goncharov and Schröder [?].

Lampson coined the term *confinement problem* [?] for the challenge of confining arbitrary programs—i.e., executing arbitrary code in a manner that prevents the illegitimate leakage of information through what Lampson termed *covert channels*. Legitimate channels transfer information via a system's resources used as intended by its architects. Covert channels transfer information by using or misusing system resources in ways unintended by the system's architects. The isolation property we define—called *domain isolation*—is similar to, albeit more restrictive than, the security property from our previous work [**?**]. Delimiting the scope of effects for arbitrary programs is the essence of confinement and the combination of effect types with monads is the scoping mechanism we use to confine effects.

A simple isolation kernel written in CC is presented in Figure 7.1. We assume there are two confinement domains, named Athens (A) and Sparta (S). The kernel is a function `k` which is parameterized by *domain handler functions* for each of the input domains, with types Δ_{A} and Δ_{S} respectively. These domain handlers are applied by the kernel to produce a single effectful computation step; the effect system guarantees that the effects of the Δ_{A} (resp. Δ_{S})-typed handler are restricted to A(S). The kernel also takes as input an internal kernel state value (here just a domain tag of type `D` which serves a similar function to a process id), and domain state values of types Dom_{A} and Dom_{S}. Execution of the respective threads is interleaved according to a round robin policy. The `unfold` operator encapsulates guarded recursion.

The proof that `k` is, in fact, an *isolation* kernel rests on two important features of the CC. The effect system guarantees that the domain handlers do not themselves induce state effects outside of their respective domains. The equational logic allows us to prove, using simple monadic equational reasoning, that the interleaving of the threads by `k` does not introduce new interactions between the domains: failure in Athens will not propagate to Sparta (nor vice versa).

```
D = {A,S}
k : (ΔA×ΔS) → (D×DomA×DomS) → R^D()
k (ha,hs) (s0,α0,σ0) =
 unfold (s0,α0,σ0)
        (λ(s,α,σ). case s of
              A -> case α of
                       (Just x) -> ha x >>= λα′. return (Left (S,α′,σ))
                       Nothing  -> return (Left (S,α,σ))
              S -> case σ of
                       (Just x) -> hs x >>= λσ′. return (Left (A,α,σ′))
                       Nothing  -> return (Left (A,α,σ)))
```

Figure 7.1: A Simple Isolation Kernel in CC

The structure of the remainder of this article is as follows. The rest of this section introduces the safety property *fault isolation* and motivates our approach to it. Section 7.2 presents an overview of the literature on effect systems and monads. The Confinement Calculus is defined in Section 7.3. Section 7.4 demonstrates how to use the CC to construct and verify an isolating kernel. Formalization of the Confinement Calculus in the Coq theorem prover is discussed in Section 7.5. Related work is discussed in Section 7.6, and Section 7.7 concludes.

A Monadic Analysis of Fault Isolation

Fault isolation is a safety property which prescribes boundaries on the extent of a fault effect. Here, we take a *fault* to mean a failure within a thread that causes it to terminate abnormally. The causes of a fault can be many and system-dependent, but some typical causes include activities such as division by zero, the corruption of a runtime stack, etc. Under some circumstances, one thread's failure can "crash" other threads. Fault isolation in this context means that the failure of one thread can only effect a subset of all threads running on a system.

We assume that the threads running on a system are partitioned into *domains* where the term is adapted from the terminology of hypervisors [**?**] and separation kernels [**?**] rather than denotational semantics. Fault isolation, as we use the term, means that a thread effect may only operate on its own domain. We refer to a *fault-isolating kernel* as a multitasking, multi-domain kernel in which the imperative and fault operations on one domain have no impact on other thread domains. Our fault model applies equally as well to software traps and programmable exceptions, although we do not provide the details here.

From the perspective of an individual thread, the scope of a fault should be global. Let the thread t be a sequence of atoms, $a_0; a_1; a_2; \cdots$, then, if a_0 causes a fault, then the execution of $a_1; a_2; \cdots$ should be cancelled, thereby satisfying the (pseudo-)equation, $a_0; a_1; a_2; \cdots = a_0$. From the point of view of a concurrent system (e.g., a multitasking kernel, etc.), the scope of a fault within an individual thread must remain isolated. The execution of t is really interwoven with other actions, including potentially those of other threads (e.g., $b_0; a_0; b_1; a_1; b_2; a_2; \cdots$), and a fault within t must not effect the execution of the other actions. In other words, should a_0 cause a fault, then the following (pseudo-)equation should hold, $b_0; a_0; b_1; a_1; b_2; a_2; \cdots = b_0; a_0; b_1; b_2; \cdots$, specifying that the subsequent actions of t should be filtered from the global system execution. The pseudo- prefix on the aforementioned equations signifies that the equations capture intuitions rather than rigorous mathematical truth. The confinement calculus will allow us to make these statements rigorous.

7.2 Effect Systems and Monads

Effect systems [**?**] and monads [**?**, **?**] are means of representing the potential side effects of a program explicitly within its type. This section provides a brief overview of effect systems and monads and motivates our use of their combination.

Effect Types. Effect systems are commonly associated with impure, strongly typed functional languages (e.g., ML [**?**]) because the effect type annotations make explicit the side effects already present implicitly in the language itself. In an impure, strongly-typed functional language, the type of a function specifies its input and output behavior only. An ML function, $f : \texttt{int} \rightarrow \texttt{int}$, takes and returns integer values, but, because ML is impure, it may also have side effects (e.g., destructive update or programmable exceptions) which are not reflected in its type. An effect system would indicate the potential side effects in the type itself. Annotating the arrow in f's type with ρ (i.e., $f : \texttt{int} \xrightarrow{\rho} \texttt{int}$) could be used to indicate that f may destructively update region ρ. Effect annotations are introduced via side effecting language constructs (e.g., ML's assignment and dereference operations, `:=` and `!`, respectively). An effect type system tracks the effects within a program to indicate its potential side effects. For an excellent account of effect systems, the reader is referred to Nielson, et al. [**?**].

Monads. Pure, strongly-typed functional languages (e.g., Haskell [**?**]) do not allow side effects, so there are no implicit side effects to make explicit. Monads are used to mimic side effecting computations within a pure language. Monads in Haskell are type constructors with additional operations, bind (»=) and unit (`return`), obeying the "monad laws" (defined in Figure 7.2). What makes monads useful is that programmers can tailor the desired effects to the application being

$$\frac{\Sigma \vdash \Gamma, x : B \triangleright\ e_1 = e_2\ :\ A}{\Sigma \vdash \Gamma, x : B \triangleright\ \texttt{return}(e_1) = \texttt{return}(e_2)\ :\ MA} \quad \text{(cong1)}$$

$$\frac{\Sigma \vdash \Gamma, x : C \triangleright\ e_1 = e_2\ :\ MA \quad \Sigma \vdash \Gamma, x' : A \triangleright\ e_1' = e_2'\ :\ MB}{\Sigma \vdash \Gamma, x : C \triangleright\ e_1 \gg\!= \lambda x'.e_1' = e_2 \gg\!= \lambda x'.e_2'\ :\ MB} \quad \text{(cong2)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : MA \quad \Gamma, x_1 : B \triangleright e_2 : MB \quad \Gamma, x_2 : C \triangleright e_3 : MC}{\Sigma \vdash \Gamma, x : A \triangleright\ (e_1 \gg\!= \lambda x_1.e_2) \gg\!= \lambda x_2.e_3 = e_1 \gg\!= \lambda x_1.(e_2 \gg\!= \lambda x_2.e_3)\ :\ MC} \quad \text{(assoc)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : B \quad \Gamma, x_1 : B \triangleright e_2 : MC}{\Sigma \vdash \Gamma, x : A \triangleright\ (\texttt{return}(e_1) \gg\!= \lambda x_1.e_2) = [e_1/x_1]e_2\ :\ MC} \quad \text{(l-unit)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : MB}{\Sigma \vdash \Gamma, x : A \triangleright\ e_1 \gg\!= \lambda x_1.\texttt{return}(x_1) = e_1\ :\ MB} \quad \text{(r-unit)}$$

Figure 7.2: The Computational λ-Calculus. M stands for any monad. The "monad laws" are assoc (associativity), l-unit (left unit), and r-unit (right unit).

constructed, effectively configuring a domain-specific language for each application. Rewriting it in Haskell, f now has type `Int` $\rightarrow$ `M Int` where `M` is the monad type constructor that encapsulates desired effects. Monads are also algebraic constructions with properties useful to formal verification (more will be said about this below). Figure 7.2 presents Moggi's well-known computational λ-calculus [**?**]. The computational λ-calculus is the core of any equational logic for monadic specifications, including the logic presented in Section 7.3. An equational judgment has the form, $\Sigma \vdash \Gamma \triangleright\ e_1 = e_2\ :\ t$, where Σ, Γ and t are a set of hypotheses, a typing environment, and a type, respectively.

Effect Systems + Monads. Combining effect systems with monadic semantics (as in Wadler [**?**]) provides fine-grained tracking of effects with a semantic model of those effects. Monads give rise to an integrated theory of effects and effect propagation. The integration of multiple effects within a single monad M has consequences for formal verification. Because all of the effects are typed in M, those effects are not distinguished syntactically within the type system of a specification language.

More positively, a rich equational theory governing their interaction follows by construction.

Effect systems can reflect this semantic information in the syntax of the specification language itself, thereby making monadic specifications more amenable to logical analysis. In the setting of this research, the combination of effects systems with monads is used to abstract over computations that occur on a particular domain. Given a particular domain d and monad K, for example, any term, $\Gamma \triangleright e : K^{\{d\}}A$, is arbitrary code on domain d, i.e., its effects occur only in domain d. Combining effects systems and monads delimits the scope of effects for arbitrary programs and is the principal mechanism for designing and verifying confinement systems.

The Identity and State Monads. The identity (left) and state (right) monads are defined below (where *Sto* can be any type). The *return* operator is the monadic analogue of the identity function, injecting a value into the monad. The ≫= operator is a form of sequential application. Monadic operators other than ≫= and *return* are key to the formulation of a particular notion of computation. The state monad S encapsulates an imperative notion of computation with operators for updating and reading the state, u and g, resp.

```
data Id a      = Id a                   g : S Sto
return v       = Id v                   g = S (λσ.(σ, σ))
(Id x) ≫= f  = f x                    return v = S (λσ. (v, σ))
data S a = S(Sto → (a, Sto))            (S x) ≫= f
deS (S x)      = x                        = S (λσ0. let (v, σ1) = x σ0
u : (Sto → Sto) → S()                                in deS (f v) σ1)
u f            = S (λσ.((), f σ))
```

The Maybe Monad & Errors. The usual formulation of an error monad is called *Maybe* in Haskell (see below). An error (i.e., *Nothing*) has the effect of canceling the rest of the computation (i.e., f). The scope of *Nothing* is global in the sense that each of the following expressions evaluates to *Nothing*: $(Just\,1 \gg\!= \lambda v.\, Nothing)$, $(Nothing \gg\!= \lambda d.\, Just\,1)$, $(Just\,1 \gg\!= \lambda v.\, Nothing \gg\!= \lambda d.\, Just\,2)$.

$$\begin{array}{llll}
\textbf{data}\ Maybe\ a & = Just\ a \mid Nothing & Just\ v \gg\!= f & = f\ v \\
return & = Just & Nothing \gg\!= f & = Nothing
\end{array}$$

Observe that this behavior precludes the possibility of *fault isolation* within domains: if the *Nothing* occurs on one domain and the $(Just\,1)$ or $(Just\,2)$ occur on another, the entire multi-domain computation will be canceled. From a security point of view, this is clearly undesirable: allowing a high-security computation to terminate a low-security computation introduces information flow via a termination channel, and allowing a low-security computation to terminate a high-security computation exposes the system to a denial of service attack.

Monad Transformers. Monad transformers allow monads to be combined and extended. Monad transformers corresponding to the state monad are defined in Haskell below. Formulations of the state monad equivalent to those above are produced by the applications of this transformer to the identity monad, *StateT Sto Id*. In the following, type variable *m* abstracts over monads.

$$\begin{array}{l}
\textbf{data}\ StateT\ s\ m\ a\ =\ ST\ (s\ \rightarrow\ m\ (a, s)) \\
deST\ (ST\ x)\ =\ x \\
return\ v\ =\ ST\ (\lambda s.\, return_m\ (v, s)) \\
(ST\ x) \gg\!= f\ =\ ST\ (\lambda s_0.\, (x\ s_0) \gg\!=_m \lambda(y, s1).\, deST\ (f\ y)\ s1) \\
lift\ :\ m\ a\ \rightarrow\ StateT\ s\ m\ a \\
lift\ \varphi\ =\ ST\ (\lambda s.\, \varphi \gg\!=_m \lambda v.\, return_m\ (v, s))
\end{array}$$

For a monad *m* and type *s*, (*StateT s m*) "extends" *m* with an updateable store *s*. The *lift* morphism is used to redefine any existing operations on *m* for the monad

($StateT\ s\ m$). The process of lifting operations is analogous to inheritance in object-oriented languages. The layer ($StateT\ s\ m$) also generalizes the definitions of the update and get operators:

$$
\begin{array}{lll}
u : (s \to s) \to StateT\ s\ m\ () & \qquad & g : StateT\ s\ m\ s \\
u\, f = ST\ (\lambda s.\ return_m\ ((), f\ s)) & & g = ST\ (\lambda s.\ return_m\ (s, s))
\end{array}
$$

Layered State Monads. A layered state monad is a monad constructed from multiple applications of the state monad transformer to an existing monad.

$$
\begin{array}{l}
type\ K = StateT\ Sto\ (StateT\ Sto\ (StateT\ Sto\ Id)) \\
u_1,\ u_2,\ u_3 : (Sto \to Sto) \to K\ () \\
u_1 f = u\, f \\
u_2 f = lift\ (u\, f) \\
u_3 f = lift\ (lift\ (u\, f))
\end{array}
$$

Each application of ($StateT\ Sto$) creates a layer with its own instances of the update (u_1-u_3) and get operations (not shown). These imperative operators come with useful properties by construction [?] and some of these are included as the equational rules (clobber) and (atomic n.i.) (atomic non-interference) in Section 7.3.

Resumption-Monadic Concurrency. Two varieties of resumption monad are utilized here, the *basic* and *reactive* resumption monads [?]. Basic resumptions encapsulate a concurrency-as-interleaving notion of computation, while reactive resumptions refine this notion to include a failure signal. The basic and reactive monad transformers are defined in Haskell in terms of monad m as:

$$
\begin{array}{ll}
\mathbf{data}\ ResT\ m\ a & = Done\ a \mid Pause\ (m\ (ResT\ m\ a)) \\
return & = Done \\
(Done\ v) \gg\!= f & = f\ v \\
(Pause\ \varphi) \gg\!= f & = Pause\ (\varphi \gg\!=_m \lambda\kappa.\ return_m\ (\kappa \gg\!= f))
\end{array}
$$

$$
\mathbf{data}\ ReactT\ m\ a = Dn\ a \mid Ps\ (m\ (ReactT\ m\ a)) \mid Fail
$$

$$
\begin{array}{ll}
\mathit{return} & = \mathit{Dn} \\
(\mathit{Dn}\, v) \gg\!= f & = f\, v \\
(\mathit{Ps}\, \varphi) \gg\!= f & = \mathit{Ps}\, (\varphi \gg\!=_m \lambda\kappa.\, \mathit{return}_m\, (\kappa \gg\!= f)) \\
\mathit{Fail} \gg\!= f & = \mathit{Fail}
\end{array}
$$

We chose to formulate the reactive resumption transformer along the lines of Swierstra and Altenkirch [**?**] rather than that of our previous work [**?**] because it is simpler. We define the following monads: $Re = ReactTK$, $R = ResTK$, and $K = StateT^n\ Sto\ Id$ where n is the the number of domains and *Sto* is the type of stores (left unspecified).

Figure 7.3 presents the concurrency and co-recursion operations for R and Re. The *step* operation lifts an m-computation into the R (resp. Re) monad, thereby creating an atomic (w.r.t. R (Re)) computation. A resumption computation may be viewed as a (possibly infinite) sequence of such steps; a finite R-computation will have the form, $(step_R\ m_1) \gg\!=_R \lambda v_1. \cdots \gg\!=_R \lambda v_n. (step_R\ m_n)$. The definition of $step_R$ is below ($step_{Re}$ is analogous). The $unfold_R$ operator is used to define kernels while the $unfold_{Re}$ operator is used to define threads. The co-recursion provided by these operators is the only form of co-recursion supported by the confinement calculus. An important consequence of this limitation on recursion is that it guarantees productivity [**?**]. It should be noted that the presence of *Maybe* in the type of $unfold_{Re}$ means that threads may fail, while its absence from the type of $unfold_R$ means that kernels cannot fail. The unfold operators are defined below; note that *Either a b* is simply Haskell-inspired notation for the sum type $a + b$.

$$step_R : K A \to R\, A$$
$$step_R\ \varphi = Pause\ (\varphi \ggg= (return \circ Done))$$

$$\begin{array}{lll}
unfold_R & : & (Monad\ t) \Rightarrow a \to (a \to t\,(Either\ a\ b)) \to ResT\ t\ b \\
unfold_R\ a\ f & = & step_R\ (f\ a) \ggg= \lambda\kappa. \\
& & \quad \mathbf{case}\ \kappa\ \mathbf{of} \\
& & \qquad (Left\ a') \quad \to \quad unfold_R\ a'\ f \\
& & \qquad (Right\ b) \quad \to \quad return\ b \\
unfold_{Re} & : & (Monad\ t) \Rightarrow a \to (a \to t\,(Maybe\,(Either\ a\ b))) \to ReactT\ t\ b \\
unfold_{Re}\ a\ f & = & step_{Re}\ (f\ a) \ggg= \lambda\kappa. \\
& & \quad \mathbf{case}\ \kappa\ \mathbf{of} \\
& & \qquad (Just\ (Left\ a')) \quad \to \quad unfold_{Re}\ a'\ f \\
& & \qquad (Just\ (Right\ b)) \quad \to \quad return\ b \\
& & \qquad Nothing \quad \to \quad Fail
\end{array}$$

Figure 7.3: Monadic Concurrency and Co-recursion Operations

$$\begin{array}{lll}
e, e' \in Exp & ::= & x \mid \lambda x.e \mid e\,e' \mid \mathtt{return}\ e \mid e \gg= \lambda x.e' \mid \mathtt{get} \mid \mathtt{upd}\ e \\
& & \mid \mathtt{fail} \mid \mathtt{mask} \mid \mathtt{out} \mid \mathtt{step} \mid \mathtt{unfold} \mid \mathtt{zero} \mid \mathtt{succ} \mid \mathtt{natRec} \\
A, B \in Type & ::= & A \to B \mid \mathtt{K}^{\sigma}\ A \mid \mathtt{R}^{\sigma}\ A \mid \mathtt{Re}^{\sigma}\ A \mid \mathtt{Sto} \mid \mathtt{Nat} \mid () \mid A + B \mid A \times B
\end{array}$$

Figure 7.4: Abstract Syntax. Assume $D = \{d_1, \ldots, d_n\}$ and $\sigma \in \mathcal{P}(D)$.

7.3 The Confinement Calculus

This section introduces the confinement calculus and defines its syntax, type system and semantics. The CC proceeds from Moggi's computational λ-calculus [?] and Wadler's marriage of type systems for effects with monads [?].

Types in the CC are directly reflective of semantic domains introduced in the previous section, and as a result are named similarly. As a notational convention, we will use teletype font when expressing a type in CC (e.g. `K Nat`), and an italic font when referring to semantic domains (e.g. *K Nat*).

Abstract Syntax. Figure 7.4 presents the abstract syntax for the CC. The finite set of domains, D contains labels for all thread domains in the system (D replaces *Region*

$$\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \texttt{K}^{\emptyset} A} \qquad \frac{\Gamma \triangleright e : \texttt{K}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \texttt{K}^{\sigma'} B}{\Gamma \triangleright e \ggg\!= \lambda x.e' : \texttt{K}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \vdash e : \texttt{K}^{\sigma} A \quad \sigma \subseteq \sigma'}{\Gamma \vdash e : \texttt{K}^{\sigma'} A}$$

$$\frac{}{\Gamma \vdash \texttt{get} : \texttt{K}^{\{d\}} \texttt{Sto}} \qquad \frac{\Gamma \vdash f : \texttt{Sto} \to \texttt{Sto}}{\Gamma \vdash \texttt{upd}\ f : \texttt{K}^{\{d\}} ()} \qquad \frac{}{\Gamma \triangleright \texttt{mask} : \texttt{K}^{\{d\}} ()}$$

Figure 7.5: Type System for Imperative Effects

from Wadler's original presentation of MONAD [?]). We also diverge slightly from Wadler's language in that we do not track the "sort" of effects that a computation may cause: reading, writing, and failure are all treated the same.

The monadic expression language *Exp* has familiar computational λ-calculus constructs as well as imperative operations (`get, upd`), an imperative operation (`mask`) used in specifying the isolation of imperative effects [?], and others for resumption monadic computations (`out`, `step`, and `unfold`). Intuitively, the `mask` operation has the effect of resetting or "zeroing out" a particular domain. Expressions `unfold`, `step` and `out` are resumption-monadic operations. The `unfold` operator encapsulates corecursion, and its semantics are structured to allow only guarded recursion. More will be said about `fail`, `step` and `out` later in this section. Finally, the expressions `zero`, `succ`, and `natRec` allow construction of, and primitive recursion over, natural numbers. Note that the *only* forms of recursion permitted by CC are primitive recursion over naturals (via `natRec`), and guarded corecursion over resumptions (via `unfold`).

The type syntax in Figure 7.4 contains three monads. The monads `K`, `R` and `Re` encapsulate layered state, concurrency and system executions, and concurrent threads, respectively. The effect system can express fine-grained distinctions about computations and, in particular, allows the domain of a thread to be expressed in

$$\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \mathtt{R}^{\emptyset} A} \qquad \frac{\Gamma \triangleright e : \mathtt{R}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \mathtt{R}^{\sigma'} B}{\Gamma \triangleright e \gg\!= \lambda x.e' : \mathtt{R}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \triangleright e : \mathtt{K}^{\sigma} A}{\Gamma \triangleright \texttt{step}\ e : \mathtt{R}^{\sigma} A}$$

$$\frac{\Gamma \triangleright p : \mathtt{R}^{\sigma} A}{\Gamma \triangleright \texttt{out}\ p : \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma} A)} \qquad \frac{\Gamma \triangleright p : A \quad \Gamma, x{:}A \triangleright q : \mathtt{K}^{\sigma}(A + B)}{\Gamma \triangleright \texttt{unfold}\ p\ (\lambda x.q) : \mathtt{R}^{\sigma} B} \qquad \frac{}{\begin{array}{l}\Gamma \triangleright \texttt{natRec} : \\ \quad A \to (A \to A) \to Nat \to A\end{array}}$$

Figure 7.6: Type System for Concurrency. The rule for `natRec` is also included.

$$\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \mathtt{Re}^{\emptyset} A} \qquad \frac{\Gamma \triangleright e : \mathtt{Re}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \mathtt{Re}^{\sigma'} B}{\Gamma \triangleright e \gg\!= \lambda x.e' : \mathtt{Re}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \triangleright e : \mathtt{K}^{\sigma} A}{\Gamma \triangleright \texttt{step}\ e : \mathtt{Re}^{\sigma} A}$$

$$\frac{\Gamma \triangleright p : \mathtt{Re}^{\sigma} A}{\Gamma \triangleright \texttt{out}\ p : \mathtt{K}^{\sigma}(\mathtt{Re}^{\sigma} A)} \qquad \frac{\Gamma \triangleright p : A \quad \Gamma, x{:}A \triangleright q : \mathtt{K}^{\sigma}(A + B + ())}{\Gamma \triangleright \texttt{unfold}\ p\ (\lambda x.q) : \mathtt{Re}^{\sigma} B} \qquad \frac{}{\Gamma \triangleright \texttt{fail} : \mathtt{Re}^{\{d\}} A}$$

$$\frac{A < B \qquad \sigma_0 \subseteq \sigma_1}{\mathtt{K}^{\sigma_0} A < \mathtt{K}^{\sigma_1} B} \qquad \frac{A < B \qquad \sigma_0 \subseteq \sigma_1}{\mathtt{R}^{\sigma_0} A < \mathtt{R}^{\sigma_1} B} \qquad \frac{A < B \qquad \sigma_0 \subseteq \sigma_1}{\mathtt{Re}^{\sigma_0} A < \mathtt{Re}^{\sigma_1} B}$$

Figure 7.7: Type System for Reactive Concurrency; Subtyping Relation

its type.

Types and Effects for the Confinement Calculus. The type and effect system for the CC is presented in Figures 7.5-7.7 and that figure is divided into three sections. Figure 7.5 contains the system for the imperative core of CC—i.e., the computations in the monad `K`. Figure 7.6 contains the type system for concurrency. Figure 7.7 contains the type system for threads—i.e., computations in the `Re` monad. The `Re` monad expresses the same notion of computation as the `R` monad, except that it also contains a signal `fail`. A thread may use `fail` to generate a fault, and it is up to the kernel component to limit the extent of the fault to the thread's domain.

Figure 7.7 gives the rules for subtyping monadic computations (standard rules for reflexivity, transitivity, and for arrow, product, and sum types are omitted). The intuition is that one monadic computation φ may stand in for another computation γ without breaking type safety, if and only if the result type of φ is a subtype of that of γ, *and* φ's affected domains are a subset of γ's. An effect-free computation of type $\mathtt{K}^{\emptyset}A$ also has type $\mathtt{K}^{\{d\}}A$ (but not vice versa).

Denotational Semantics of the Confinement Calculus. The dynamic semantics of CC is a typed denotational semantics, meaning that the denotation of terms depends in part on their typing derivations. This allows us to overload the monad operations. The denotation of types does not depend on effect annotations and is completely standard. The `out` operation accesses the first step of a concurrent (`R`-typed) computation, producing a `K`-typed computation. Omitted from Figure 7.8 is the denotation of `natRec`, which is defined by structural induction on naturals. The CC term `run` and its denotation are defined as:

$$\mathtt{run\ n}\ \varphi_0\ \mathtt{=\ natRec\ (return}_K\ \varphi_0\mathtt{)\ (}\lambda\varphi\mathtt{.\ (}\varphi\ \mathtt{>>=\ out}_R\mathtt{))\ n}$$

$$\begin{array}{l} run : Nat \to RA \to K(RA) \\ run\ 0\ \varphi \qquad\quad = return\ \varphi \\ run\ (n+1)\ \varphi = out\ \varphi \ \ggg\ run\ n \end{array}$$

Equational Logic. The rules of the equational logic encode known facts about the denotational semantics proven in an earlier publication [**?**]. For instance, the *run* operator "unrolls" *R* computations:

$$run\ (n+1)\ (step\ \varphi \ggg_R f) = \varphi \ggg_K run\ n \circ f \qquad (7.1)$$

$$run\ (n+1)\ (return_R\ x) = return_K\ (return_R\ x) \qquad (7.2)$$

Properties (7.1) and (7.2) justify our introduction of the following rules:

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright \varphi : \mathtt{K}^{\sigma}A \quad \Gamma, x : A \triangleright e : \mathtt{R}^{\sigma}B}{\Sigma \vdash \Gamma \triangleright\ \mathtt{run}\ (n{+}1)\ (\mathtt{step}\ \varphi \ \mathtt{»=}\ \lambda x.e) = \varphi\ \mathtt{»=}\ \lambda x.\ \mathtt{run}\ n\ e\ : \ \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma}B)} \quad \text{(run-step)}$$

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright e : A}{\Sigma \vdash \Gamma \triangleright\ \mathtt{run}\ (n{+}1)\ (\mathtt{return}_R\ e) = \mathtt{return}_K\ (\mathtt{return}_R\ e)\ : \ \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma}A)} \quad \text{(run-return)}$$

$$
\begin{array}{lcl}
[\![\Gamma \triangleright x : A]\!]\rho & = & \rho\, x \\
[\![\Gamma \triangleright \lambda x.e : A \longrightarrow B]\!]\rho & = & \lambda v.[\![\Gamma, x : A \triangleright e : B]\!](\rho[x \mapsto v]) \\
[\![\Gamma \triangleright e\, e' : A]\!]\rho & = & ([\![\Gamma \triangleright e : B \longrightarrow A]\!]\rho)\ ([\![\Gamma \triangleright e' : B]\!]\rho) \\
[\![\Gamma \triangleright \mathtt{return}\, e : M^{\emptyset} A]\!]\rho & = & \mathit{return}_M\ ([\![\Gamma \triangleright e : A]\!]\rho) \\
[\![\Gamma \triangleright \mathtt{get} : \mathtt{K}^{d_i} \mathit{Sto}]\!]\rho & = & \mathit{lift}_i\ g \\
[\![\Gamma \triangleright \mathtt{upd}\, \delta : \mathtt{K}^{d_i}\ ()]\!]\rho & = & \mathit{lift}_i\ (u\ ([\![\Gamma \triangleright \delta : \mathit{Sto} \longrightarrow \mathit{Sto}]\!]\rho)) \\
[\![\Gamma \triangleright \mathtt{mask} : \mathtt{K}^{d_i}\ ()]\!]\rho & = & \mathit{lift}_i\ (u\ (\lambda x.s_0)) \\
\multicolumn{3}{l}{[\![\Gamma \triangleright e \gg\!= \lambda x.e' : M^{\sigma \cup \sigma'} B]\!]\rho} \\
\multicolumn{3}{l}{\qquad = [\![\Gamma \triangleright e : M^{\sigma} A]\!]\rho \gg\!=_M \lambda v.[\![\Gamma, x{:}A \triangleright e' : M^{\sigma'} B]\!](\rho[x \mapsto v])} \\
\multicolumn{3}{l}{[\![\Gamma \triangleright \mathtt{unfold}\, e\ (\lambda x.e') : \mathtt{R}^{\sigma} B]\!]\rho} \\
\multicolumn{3}{l}{\qquad = \mathit{unfold}_{\mathtt{R}}\ ([\![\Gamma \triangleright e : A]\!]\rho)\ (\lambda v.[\![\Gamma, x : A \triangleright e' : \mathtt{K}^{\sigma}(A + B)]\!]\ (\rho[x \mapsto v]))} \\
[\![\Gamma \triangleright \mathtt{out}(p) : \mathtt{K}^{\sigma}(\mathtt{R}^{\sigma} A)]\!]\rho & = & \begin{cases} \mathit{return}\,(\mathit{Done}\ v) & \text{if} [\![p]\!]\rho = \mathit{Done}\ v \\ \varphi & \text{if} [\![p]\!]\rho = \mathit{Pause}\ \varphi \end{cases} \\
[\![\Gamma \triangleright \mathtt{fail} : \mathtt{Re}^{d_i}\ A]\!]\rho & = & \mathit{Fail} \\
\multicolumn{3}{l}{[\![\Gamma \triangleright \mathtt{unfold}\, e\ (\lambda x.e') : \mathtt{Re}^{\sigma} B]\!]\rho} \\
\multicolumn{3}{l}{\qquad = \mathit{unfold}_{\mathit{Re}}\ ([\![\Gamma \triangleright e : A]\!]\rho)(\lambda v.[\![\Gamma, x : A \triangleright e' : \mathtt{K}^{\sigma}(A + B + ())]\!]\ (\rho[x \mapsto v]))} \\
[\![\Gamma \triangleright \mathtt{out}(p) : \mathtt{K}^{\sigma}(\mathtt{Re}^{\sigma} A)]\!]\rho & = & \begin{cases} \mathit{return}\,(\mathit{Just}(\mathit{Dn}\ v)) & \text{if} [\![p]\!]\rho = \mathit{Dn}\ v \\ \varphi \gg\!\!= (\mathit{return} \circ \mathit{Just}) & \text{if} [\![p]\!]\rho = \mathit{Ps}\ \varphi \\ \mathit{return}\, \mathit{Nothing} & \text{if} [\![p]\!]\rho = \mathit{Fail} \end{cases}
\end{array}
$$

Figure 7.8: Denotational Semantics. *M* stands for the *K*, *R*, or *Re* monads. *K*, *R*, and *Re* are defined in Section 7.2.

A straightforward induction on the structure of type derivations justifies the soundness of the following rules. This induction makes use of previous work (specifically Theorems 1-3 on page 17 [**?**]) and the "lifting law" of Liang [**?**]: *lift*$(x \gg\!= f) =$ *liftx* $\gg\!=$ *lift* $\circ f$.

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0} A \qquad \Gamma \triangleright \mathtt{mask} : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \subseteq \sigma_1}{\Sigma \vdash \Gamma \triangleright\ \varphi \gg \mathtt{mask} = \mathtt{mask}\ :\ \mathtt{K}^{\sigma_1}()} \quad \text{(clobber)}$$

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0}() \qquad \Gamma \triangleright \gamma : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \cap \sigma_1 = \emptyset}{\Sigma \vdash \Gamma \triangleright\ \varphi \gg \gamma = \gamma \gg \varphi\ :\ \mathtt{K}^{\sigma_0 \cup \sigma_1}()} \quad \text{(atomic n.i.)}$$

7.4 Isolation Kernels in Confinement Calculus

In this section, we turn our attention the construction of *isolation kernels* within the confinement calculus. An isolation kernel is a function which interleaves the execution of two or more threads in different domains, without introducing any interactions across domains. Put another way, an isolation kernel must have the property that a computation in domain d, when interleaved with a computation in $d' \neq d$, behaves exactly the same as it would if the d' computation had never happened.

The formal definition of isolation is made in terms of a notion called *domain similarity*. Two computations φ and γ are domain similar in a domain d if and only if for every finite prefix of φ, there exists a finite prefix of γ whose effects in d are the same.

Definition 1 (Domain Similarity Relation)**.** *Consider two computations* $\varphi, \gamma : \mathtt{R}^{d_1 \cup \ldots \cup d_n} A$. *We say* φ *and* γ *are* similar with respect to domain d_i

(*written* $\varphi \sim_{d_i} \gamma$) *if and only if the following holds.*

$$\forall\, n{\in}N.\ \exists\, m{\in}N.\ \texttt{run}\, n\, \varphi \gg_K \texttt{mask} \ = \ \texttt{run}\, m\, \gamma \gg_K \texttt{mask}$$

where $\texttt{mask} = \texttt{mask}_{d_1} \gg \ldots \gg \texttt{mask}_{d_{i-1}} \gg \texttt{mask}_{d_{i+1}} \gg \ldots \gg \texttt{mask}_{d_n}$.

Kernels. Assume in Definitions 2-4 that $D = \{d_1, \ldots, d_n\}$ is a fixed set of domains. We first define a notion of *state*: namely a tuple containing one element representing the kernel's internal state, and an additional element for the state of each confinement domain. The domain states are wrapped in a *Maybe* constructor to represent the possibility of failure within a domain.

Definition 2 (Domain and Kernel State)**.** *The state of domain i, Dom_i is defined as:*

$$Dom_i = \texttt{Re}^{\{d_i\}}()$$

The type of kernel states for a type t is:

$$S = t{\times}(\texttt{Maybe}\ Dom_1){\times}\ldots{\times}(\texttt{Maybe}\ Dom_n)$$

A kernel is parameterized by *handlers* for each domain. A handler is a state transition function on domain states, which may also have effects on the global state (hence the presence of K in the type). The only restriction on a handler for domain d_i is that its effects must be restricted to d_i.

Definition 3 (Domain Handler Function)**.** *The type of handler functions for domain d_i is:*

$$\Delta_{d_i} = Dom_i \rightarrow \texttt{K}^{\{d_i\}}(\texttt{Maybe}\ Dom_i)$$

The handler vector type for D is:

$$\Delta_D = \Delta_1 \times \ldots \times \Delta_n$$

Putting the pieces together brings us to the definition of a kernel. Note that a kernel in our sense is parameterized over handler vectors.

Definition 4 (Kernel)**.** *Given a handler vector Δ_D, kernel state type S, and an answer type Ans, the type of kernels is defined by the following:*

$$\Delta_D \rightarrow S \rightarrow \mathtt{K}^D(S + Ans)$$

Defining and Proving Isolation. To simplify the presentation, we will restrict our attention for the remainder of this section to the case of two domains, called A (for *Athens*) and S (for *Sparta*). All the results here generalize naturally to more than two domains.

We define isolation by an extensional property on kernels. A kernel is said to be isolating if the result of "eliminating" the computation in any one domain has no effect on the outcome of any other. This is a property akin to noninterference [**?**]. We express this formally by replacing the domain handler with `return` – the "do-nothing" computation – and replacing the domain state with `Nothing`.

Definition 5 (Isolation in the Presence of Faults)**.** *A kernel k is* isolating in the presence of faults *if and only if*

$$\begin{array}{lcl} k\,(f_{\mathsf{A}}, f_{\mathsf{S}})(s, d_{\mathsf{A}}, d_{\mathsf{S}}) & \sim_{\mathsf{A}} & k\,(f_{\mathsf{A}}, \mathtt{return})(s, d_{\mathsf{A}}, \mathtt{Nothing}) \\ k\,(f_{\mathsf{A}}, f_{\mathsf{S}})(s, d_{\mathsf{A}}, d_{\mathsf{S}}) & \sim_{\mathsf{S}} & k\,(\mathtt{return}, f_{\mathsf{S}})(s, \mathtt{Nothing}, d_{\mathsf{S}}) \end{array}$$

The following theorem shows that kernel `k` of Figure 7.1 satisfies this definition.

Theorem 1 (k is isolating)**.** *The kernel* $\mathtt{k}$ *of Figure 7.1 is isolating in the presence of faults.*

Proof. We will show the S-similarity side of the proof, since the A-similarity proof is analogous. N.b., Definition 1 allows the number of steps on each side of the equation (n and m) to be different. Here it suffices to fix $m = n$:

$$\begin{aligned}&\mathtt{run}\,n\,(\mathtt{k}\,(f_{\mathsf{A}}, f_{\mathsf{S}})\,(s, d_{\mathsf{A}}, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}}\\ &\qquad = \mathtt{run}\;n\;(\mathtt{k}\,(\mathtt{return}, f_{\mathsf{S}})\,(s, \mathtt{Nothing}, d_{\mathsf{S}})) \gg\; \mathtt{mask}_{\mathsf{A}}\end{aligned}$$

The proof is by induction on n. The base case is trivial. We proceed by cases on s, d_{A}, and d_{S}. The most interesting case is when $s = \mathsf{A}$ and $d_{\mathsf{A}} = \mathtt{Just}\ x$; all others involve no more than straightforward evaluation and an application of the induction hypothesis. Let $s = \mathsf{A}$ and $d_{\mathsf{A}} = \mathtt{Just}\ x$. Then:

$$\begin{aligned}
&\mathtt{run}(n+1)(\mathtt{k}(f_{\mathsf{A}}, f_{\mathsf{S}})(\mathsf{A}, \mathtt{Just}\ x, d_{\mathsf{S}}))\gg\mathtt{mask}_{\mathsf{A}} && \\
&= \mathtt{run}\,1\,(\mathtt{k}\,(f_{\mathsf{A}},f_{\mathsf{S}})\,(\mathsf{A}, \mathtt{Just}\ \ x, d_{\mathsf{S}})) \gg\!= \mathtt{run}\,n \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{prop}\ \mathtt{run}\\
&= f_{\mathsf{A}}\,x \gg\!= \lambda d'_{\mathsf{A}}.\,\mathtt{run}\,n\,(\mathtt{k}\,(f_{\mathsf{A}},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Just}\ \ x, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{evaluation}\\
&= f_{\mathsf{A}}\,x \gg\!= \lambda d'_{\mathsf{A}}.\,\mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{i.h.}\\
&= f_{\mathsf{A}}\,x \gg\!= \lambda d'_{\mathsf{A}}.\,\mathtt{mask}_{\mathsf{A}} \gg \mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) && \because \textit{atomic n.i.}\\
&= f_{\mathsf{A}}\,x \gg \mathtt{mask}_{\mathsf{A}} \gg \mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) && \because d_{\mathsf{A}}'\ \textit{does not occur}\\
&= \mathtt{mask}_{\mathsf{A}} \gg \mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) && \because \textit{clobber}\\
&= \mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{atomic n.i.}\\
&= \mathtt{return}\,() \gg \mathtt{run}\,n\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{left unit}\\
&= \mathtt{run}\,1\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{A}, \mathtt{Nothing}, d_{\mathsf{S}})) \gg\!= \mathtt{run}\,n \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{evaluation}\\
&= \mathtt{run}\,(n+1)\,(\mathtt{k}\,(\mathtt{return},f_{\mathsf{S}})\,(\mathsf{A}, \mathtt{Nothing}, d_{\mathsf{S}})) \gg \mathtt{mask}_{\mathsf{A}} && \because \textit{prop}\ \mathtt{run}
\end{aligned}$$

□

7.5 Mechanizing the Logic in Coq

The syntax, denotational semantics, and equational logic of CC have all been mechanized in the Coq [?] theorem prover. In lieu of separate syntaxes for type judgments and terms as presented in the preceding sections, the Coq formulation uses a strongly-typed term formulation as suggested by Benton et al [?]. We combine this with a dependently typed denotational semantics along the lines of Chlipala [?] (see Chapter 9). The payoff of this approach is that we do not need to establish many of the usual properties such as progress and subject reduction that usually accompany an operational approach. Furthermore, strongly-typed terms are much more amenable to the use of Coq's built-in system of parametric relations and morphisms. Just a handful of relation and morphism declarations lets us reuse many of Coq's standard tactics (e.g., `replace` and `rewrite`) when reasoning in the CC logic.

Figure 7.9 presents an example equational judgment rule from the Coq development, representing the clobber rule. The only major difference between the Coq formalism and the corresponding rule in Section 7.3 has to do with the need for explicit subtyping: the term constructor `subsume` casts a term from a supertype to a subtype, and requires as an argument a proof term showing that the subtyping relationship holds (here called `S_just`). The full development in Coq is available by request.

```
J_clobber : ∀ (Γ:list ty) (d:domain) (t:ty)
              (te:term Γ (tyK (onedom d) t)),
   let S_just := S_tyK (onedom d) (union (onedom d) (onedom d))
                      (S_refl tynil) (clobber_obligation _)
   in  eq_judgment (subsume S_just (nullbindK te (mask Γ d))) (mask Γ d)
```

Figure 7.9: Expressing the Clobber Rule in Coq

7.6 Related Work

Klein et al. [**?**] describe their experience in designing, implementing and verifying the seL4 secure kernel. Monads are applied as an organizing principle at all levels of the seL4 design, implementation and verification. Their model of effects is different from the one described here. The seL4 monadic models encapsulate errors, state and non-determinism. The notions of computation applied here include concurrency and interactivity (R and Re, respectively) as well as layered state (K). Also, the type system underlying the seL4 models does not include effect types. The present work models faults via simulation on distinct domains rather than as part of an integrated model of effects. Cock, Klein and Sewell [**?**] apply Hoare-style reasoning to prove that a design is fault free. Kernels in the CC are fault-free as a by-product of our type system and it would be interesting to investigate whether the application of CC in the seL4 construction and verification process would alleviate some verification effort.

Another similarly interesting question is to consider the impact of integrating layered state and resumption-based concurrency into their abstract, executable and machine models. One design choice, for example, concerns the placement of preemption points—i.e., places where interrupts may occur—within the seL4 kernel specifications. It seems plausible that interrupt handling in seL4 might be simplified by the presence of an explicit concurrency model—i.e., resumptions. It also seems plausible that aspects of the seL4 design and verification effort might be reduced or abstracted by the inclusion of effect-scoping mechanisms like layered state and effect types—e.g., issues arising from the separation of kernel space from user space.

Language-based security [?] seeks to apply concepts from programming languages research to the design, construction and verification of secure systems. While fault isolation is generally considered a safety property, it is also a security property as well in that an unconfined fault may be used for a denial of service attack and also as a covert channel. Monads were first applied within the context of language-based security by Abadi et al. [?], although the use of effect systems seems considerably less common in the security literature. Bartoletti et al. [?, ?] apply effect systems to history-based access control and to the secure orchestration of networked services. Bauer et al. [?] applied the combination of effect systems and monads to the design of secure program monitors; their work appears to be the first and only previously published research in security to do so. The current work differs from theirs mainly in our use of interaction properties of effects that follow by the construction of the monads themselves. These by-construction properties provide considerable leverage towards formal verification. Scheduler-independent security [?, ?]) considers the relationship between scheduling and security and investigates possibilistic models of security that do not depend on particular schedulers. A natural next step for the current research is to investigate scheduler freedom with respect to CC kernels (e.g., Definition 4).

7.7 Conclusions

The research described here seeks to apply tools and techniques from programming languages research—e.g., monads, type theory, language compilers—to the production of high assurance systems. We are interested, in particular, in reducing the cost of certification and re-certification of verified artifacts. The questions we confront are, in terms of semantic effects, what can untrusted code do and, given a semantic model of untrusted code, how can we specify a system for running it safely in isolation? Untrusted code can read and write store obviously. It can fail—i.e., cause an exception. It can also signal the operating system via a trap. These effects are inherited from the machine language of the underlying hardware.

Previous work [?] explored the application of modular monadic semantics to the design and verification of separation kernels. Each domain is associated with an individual state monad transformer [?] and the "layered" state monad constructed with these transformers has "by-construction" properties that are helpful for verifying the information flow security. The present work builds upon this in the following ways. Our previous work did not consider fault effects, and the layering approach taken in that work does not generalize to handle faults. It is also interesting, albeit less significant, that the semantics of the CC effect system is organized by state monad transformers. Structuring the semantics of Wadler's MONAD language with monad transformers was first suggested by Wadler [?] and the present work, to the best of our knowledge, is the first to actually do so. Although the current work focuses on isolation, we believe that it can be readily adapted to MILS (multiple independent levels of security) systems by refining the effect types to distinguish, for example, reads and writes on domains (as Wadler's original MONAD language did). One could then express, for example, a high security handler that is allowed to read from, but not write to, domains lower in the security hierarchy.

Kobayashi [?] proposed a general framework for reasoning about monadic specifications based in modal logic in which monads are formalized as individual modalities. Nanevski elaborated on the monad-as-modality paradigm, introducing a modal logic for exception handling as an alternative to the exception monad [**?**]. Nanevski's logic is an S4 modal logic in which necessity encodes a computation which may cause an exception—i.e., $\Box_C A$ represents a computation of an A value that may cause an exception named in set C. Modal logics have been adapted to security verification by partially ordering modalities to reflect a security lattice by Allwein and Harrison [**?**]. We are currently investigating the integration of the monad-as-modality paradigm with security-enabled partially-ordered modalities into a modal logic for verifying the security of monadic specifications in the confinement calculus and related systems.

Chapter 8

Conclusions and Future Work

In this chapter we conclude with a brief summary of major results, and a discussion of future research directions.

8.1 Results

This dissertation has presented three main results.

Semantic foundations and reasoning techniques for secure hardware systems. We have demonstrated that modular monadic semantics, which has previously been shown to provide a powerful framework for constructing and reasoning about programming languages in a modular style, also provides a solid foundation for verifiably secure hardware design. In particular we have shown that reactive resumption monads form a natural basis for constructing and reasoning about hardware systems. The computational λ-calculus/programming language of Chapter 3 and the logical treatment of Chapter 7 together provide a formal system not just for constructing resumption monadic programs, but also for reasoning about their formal properties.

A language for designing high-assurance hardware systems. We have presented the design and formal semantics of a computational λ-calculus called ReWire Core that provides expressive constructs for hardware design, yet also ensures that all well-formed programs are synthesizable to running hardware. The case study of Chapter 5 shows that this language enables the rapid development of hardware designs, and provides tools for high assurance from the very beginning of the design process.

A compiler that produces efficient implementations of semantics-directed hardware designs. The ReWire compiler of Chapter 4 produces implementations of circuits directly from ReWire specifications. Compiling directly from monads means there is no semantic gap between the language of specification and reasoning and the language of implementation. This eliminates a major source of complexity and error in the formal methods process. At the same time, circuits produced by ReWire are efficient; the regular expression matchers of Chapter 6 in some cases outperformed those generated by a state of the art hand-tuned implementation.

8.2 Future Work

8.2.1 Support for a Broader Class of Monads

Monads in the ReWire Core calculus are limited to those constructed with the reactive resumption and state monad transformers. This does not have to be the case. Haskell programmers often make use of a larger set of monads and monad transformers, including *ErrorT* or *ExceptionT* for exception throwing and handling; *ContT* for first class continuations; *ReaderT* for non-mutable state; *WriterT* for logging; the *list* monad which supports nondeterminism; and many combinations thereof. Many of these abstractions would be just as useful in hardware design. Exceptions are commonly seen in hardware design (consider, for example page faults), and we could make use of the nondeterminism monad or a probability distribution monad [**?**] to reason about random bit-flips and signal disruptions caused by cosmic rays.

It should be possible to extend the ReWire calculus to support many of these monadic constructs. In some cases, however, we may be forced to burden the language with the need for undesirable runtime features like dynamic memory allocation. For example, computations in the *ReaderT* monad transformer are conventionally built up in terms of the following operations:

```
ask   :: Monad m => ReaderT r m r
local :: Monad m => (r -> r) -> ReaderT r m a
                             -> ReaderT r m a
```

Informally, we can view *ReaderT* as adding a locally fixed environment of type *r* to the base monad *m*. The *ask* operation queries the current value of this environment; the operation *local f m* executes a subcomputation *m* in a local environment produced by applying the function *f* to the current environment—once this subcomputation returns, the environment effectively rolls back to its prior state. The rollback induced by the *local* operation, however, presents a serious problem if we wish to avoid the use of RAM—seemingly there is no way to implement this without the use of a stack to store both environments and return contexts. Similar concerns exist for *ErrorT* and *ExceptionT*, which would need to represent a stack of exception handlers.

Of course, if a hardware designer is using a monad transformer like *ReaderT*, we might reasonably assume that a stack implemented in RAM conforms to the designer's intentions. The presence of RAM, however, complicates ReWire's otherwise tidy timing semantics (where every **signal** corresponds to a clock tick), as RAM itself typically is a synchronous device. A computation like *lift* (*local f* (*local g m*)) : *ReactT i o* (*ReaderT r Identity*) *a* would have to make at least two pushes to the stack, requiring at least two clock ticks for a single "logical" ReWire time slice. Thus the inclusion of a broader class of monads will require more thought to be put into ReWire's timing properties, and may require new language constructs in order to maintain some degree of timing predictability.

8.2.2 Support for Structures Other than Monads

This research has demonstrated the efficacy of monads for semantically modular design. For designs of a more structural flavor, however, previous research has already explored a number of other functional constructs. One that stands out in particular is Lava's use of lazy streams [**?**, **?**], which model circuits essentially at the level of (mutually reactive) binary signals. Another such construct is arrows, including the automata arrow [?] and the Megacz's generalized arrows [?]. Arrows allow the structural composition of circuit features (i.e., connecting together the inputs and outputs of black boxes) in a way that corresponds quite closely to most hardware engineers' pre-existing intuition of how circuits are built.

While in Chapter 2 we have made a strong case for monadic design in *contrast* to these structures, all three paradigms (lazy streams, arrows, and monadic hardware design as embodied by ReWire) have distinct and possibly *complementary* advantages: sometimes one wishes to design at the level of signals, sometimes at the level of structure, and sometimes at the level of semantics. Indeed, one may want to make use of different abstractions within different parts of the *same* design, specifying (for example) the action of one synchronous circuit component in terms of monads, but interconnecting it with others via arrows.

Fashioning a calculus that mixes hardware monads with streams and/or arrows will require extensive future research, but at the outset there are a few intriguing possibilities. Recall the definition of the underlying type for the automata arrow:

```
newtype Auto i o = Auto (i -> (o,Auto i o))
```

We can define an operation which effectively translates a computation in a reactive resumption monad into an automaton.

```
toAuto :: ReactT i o Identity a -> Auto i (Either o a)
toAuto (ReactT (Identity (Left x))) = r
  where r = Auto (\ i -> (Right x,r))
toAuto (ReactT (Identity (Right (o,k)))) =
  Auto (\ i -> (Left o,toAuto (k i)))
```

One could envision building support for *toAuto* as a primitive into a ReWire Core-like calculus enhanced with support for the automata arrow, enabling the embedding of resumption monad computations into arrow-structured circuits. Intuitively this corresponds to something like embedding a circuit specified as an imperative program into a larger structural block diagram.

8.2.3 Support for Higher-Order Abstractions

Higher-order abstractions could substantially speed the process of circuit design in ReWire. By higher-order abstractions, we mean functions that take functions or computations as arguments and/or return functions as results. For example, the redundant code present for the arithmetic instructions in the CPU of Chapter 5 could be unified by supplying the arithmetic operator as a parameter to a single, higher-order *arithInstr* function:

```
-- (rand 1:W8, rand 2:W8, carry in:Bit, result:W8)
type ArithOp = W8 -> W8 -> Bit -> W8
plusCW8  :: ArithOp
minusCW8 :: ArithOp

...

arithInstr :: ArithOp -> Register -> Register
                      -> CPU ()
arithInstr f rD rS = do
  vD             <- getReg rD
  vS             <- getReg rS
  cin            <- getCFlag
  let (cout,vD') =  f vD vS cin
  putCFlag cout
  putReg rD vD’
  tick

...

addc :: Register -> Register -> CPU ()
addc = arithInstr plusCW8

subc :: Register -> Register -> CPU ()
subc = arithInstr minusCW8
```

The challenge here essentially boils down to one of representing functions as values within the restrictions imposed on data by ReWire. We could use a structure similar to the closures employed by a functional language runtime, but these employ linked lists, which are not available to us. A promising candidate for solving this problem is a technique originally due to Reynolds called *defunctionalization* [**?**]. Defunctionalization is a program transformation that takes an arbitrary higher-order program and transforms it into a first-order program. Occurrences of higher-order constructs in the original program (i.e., functions that occur less than fully applied in argument position) may be represented by values of a newly declared data type. In the general case, the data structures produced by defunctionalization may themselves turn out to be recursive, making them difficult to represent in hardware. In the presence of certain restrictions on recursion, however, we can eliminate this possibility. Thus the *arithInstr* example above could be defunctionalized to the following (assuming we know that *plusCW8* and *minusCW8* are the only values that will ever be passed as the first argument of *arithInstr*).

```
data ArithOp = PlusCW8 | MinusCW8

doArithOp :: ArithOp -> W8 -> W8 -> Bit -> W8
doArithOp PlusCW8 x y c  = plusCW8 x y c
doArithOp MinusCW8 x y c = minusCW8 x y c

arithInstr :: ArithOp -> Register -> Register
                      -> CPU ()
arithInstr f rD rS = do
  vD             <- getReg rD
  vS             <- getReg rS
  cin            <- getCFlag
  let (cout,vD') =  doArithOp f vD vS cin
  putCFlag cout
  putReg rD vD'
  tick

...

addc :: Register -> Register -> CPU ()
addc = arithInstr PlusCW8

subc :: Register -> Register -> CPU ()
subc = arithInstr MinusCW8
```

While it is already possible to apply defunctionalization by hand, automating it would greatly increase the expressive power of ReWire.

8.2.4 Metaprogramming

Another language paradigm that has great potential to enhance ReWire's expressiveness is metaprogramming [**?**, **?**]. Metaprogramming essentially enables a program p written in a language L to produce (either statically at compile-time or dynamically at run-time) another program p' written in L, and to invoke p' from within p—or, at least, some subset of these features, depending on the implementation. The ReWire-based regular expression framework RexHacc, described in Chapter 6, is itself metaprogrammatic according to this definition: RexHacc is a program written in Haskell, producing programs (represented as strings) in Haskell (particularly in ReWire, which is a subset of Haskell). This enables the uniform construction of circuits with varying state and input/output types. The string-based approach, however, is quite fragile, as it gives no static guarantee that programs will be syntactically well-formed (let alone well-typed). Extending ReWire with quasi-quotation and splicing constructs akin to Template Haskell [**?**] would provide a much greater degree of convenience and safety to systems like RexHacc. Metaprogramming can serve as a tool for the interconnection of a rich and extensible library of circuit components.

8.2.5 Heterogeneous Computing

Looking farther into the future, ReWire may form the kernel of a language that supports heterogeneous computing, i.e., computation that mixes CPUs, FPGAs, and possibly even other architectures like GPUs. ReWire in its current form is already a subset of Haskell, and Haskell can certainly be used to specify software programs. In a heterogeneous implementation, reactive resumption monads would provide a coordination mechanism allowing the CPU-based parts and the FPGA-based parts of a program to communicate. Since both the CPU and FPGA-based halves of such a program share a common language with a uniform semantics, many of the inconveniences that plague existing heterogeneous programming environments—marshalling and unmarshalling of data, coordinating kernel calls, and so on—would be greatly reduced.

8.2.6 Tool Support for Formal Reasoning

While ReWire provides a sound and expressive basis for reasoning about reactive hardware circuits, the inherent complexity of hardware verification means that wider adoption still will likely require proof automation tools. Chapter 7 briefly discusses a formalization of the confinement calculus logic in Coq. This formalization could also serve as the basis for a ReWire proof assistant. Another possibility would be to further develop and adapt the MProver system [**?**] I designed with Bill Harrison and Aaron Stump, which contains a lightweight logic tailored towards equational reasoning about programs in a Haskell-like language.

VITA

Adam Procter was born in Columbia, Missouri on March 6, 1982, to Brenda Procter (B.A. ’81, M.S. ’93) and Michael Procter (B.A. ’76, M.S. ’78, Ph.D. ’82). He received the B.A. degree in Computer Science *summa cum laude* with a minor in Mathematics from the University of Missouri in May 2005, and the Ph.D. degree in Computer Science from the University of Missouri in December 2014.