Thesis for the Degree of Doctor of Philosophy

Embedded Languages for Describing and Verifying Hardware

Koen Claessen

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, April 2001

Embedded Languages for Describing and Verifying Hardware
Koen Claessen
ISBN 91-7291-014-3

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr 1698

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, 2001

Embedded Languages for Describing and Verifying Hardware
Koen Claessen
Department of Computing Science
Chalmers University of Technology and Göteborg University

Abstract

Lava is a system for designing, specifying, verifying and implementing hardware. It is embedded in the functional programming language Haskell, which means that hardware descriptions are first-class objects in Haskell. We are thus able to use modern programming language features, such as higher-order functions, polymorphism, type classes and laziness, in hardware descriptions.

We present two rather different versions of Lava. One version realises the embedding by using *monads* to keep track of the information specified in a hardware description. The other version uses a new language construct, called *observable sharing*, which eliminates the need for monads so that descriptions are much cleaner. Adding observable sharing to Haskell is a non-conservative extension, meaning that some properties of Haskell are lost. We thus investigate to what extent we are still allowed to use a normal Haskell compiler or interpreter.

We also introduce an embedded language for specifying properties. The use of this language is two-fold. On the one hand, we can use it to specify and later formally verify properties of the described circuits. On the other hand, we can use it to specify and randomly test properties of normal Haskell programs. As a bonus, since hardware descriptions are embedded in Haskell, we can also use it to test our circuit descriptions.

Further, we present a method for formal verification of safety properties of circuits, based on temporal induction. Temporal induction proves a property by proving it for the initial state (base case), and, by assuming it holds for a certain state, proving it also holds for the following states (step case). It is well-known that induction can be improved by strengthening the inductive hypothesis. We propose several techniques that can automatically strengthen a given property.

Keywords: embedded languages, functional programming, synchronous hardware description, specification languages, formal verification.

This dissertation is based on the following papers and reports:

1. *Lava: Hardware Design in Haskell*, written together with Per Bjesse, Mary Sheeran and Satnam Singh, and published at the International Conference on Functional Programming (ICFP) in Baltimore, Maryland, in 1998 [17].
2. *Observable Sharing for Functional Circuit Description*, written together with David Sands, and published at the Asian Computing Science Conference (ASIAN) in Phuket, Thailand, in 1999 [24].
3. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, written together with John Hughes, and published at the International Conference on Functional Programming (ICFP) in Montreal, Canada, in 2000 [23].
4. *A Tutorial on Lava: A Hardware Description and Verification System*, written together with Mary Sheeran, used as teaching material and documentation of the Lava system, in 2000 [25].
5. *An Embedded Language Framework for Hardware Compilation*, written together with Gordon Pace, and submitted for publication to the conference on Correct Hardware Design and Verification Methods (CHARME), in 2001.
6. *SAT-based Model Checking without State Space Traversal*, written together with Per Bjesse, and published at the conference on Formal Methods for Computer Aided Design (FMCAD) in Austin, Texas, in 2000 [16].

My contributions to these papers are as follows:

1. I designed and implemented the particular version of Lava described in the paper, and was heavily involved in writing most of the sections. I was not involved in the work presented in Section 2.4.
2. The problem statement, motivation and solution came from me. My co-author did most of the proofs in Section 3.4. The writing of the rest paper was divided among my co-author and me.
3. The work described in the paper, the implementation, and the writing of the paper was equally divided among my co-author and me.
4. My co-author and I discussed the contents and many of the examples together. I wrote most of the sections, except for Section 5.7, which was written by my co-author. I designed and implemented the particular version of Lava described in the paper.
5. The work described in the paper, the implementations, and the writing of the paper was equally divided among my co-author and me.
6. The work described in the paper, and the writing of the paper was equally divided among my co-author and me. I was responsible for the first prototype implementation, and my co-author made the first full-strength implementation.

Contents

Acknowledgements

I would like to thank my advisor, colleague and friend Mary Sheeran, who made me become interested in the area of hardware description and verification. She has been an excellent advisor over the years. Further, I would like to thank the co-authors of the papers in this thesis, with whom I have collaborated with pleasure: Per Bjesse, Mary Sheeran, Satnam Singh, David Sands, John Hughes, and Gordon Pace. My opponent Jean Vuillemin's suggestions have significantly improved some parts of the thesis. Without these people, the thesis would obviously not have looked the way it does now.

In the last few years, I have had several office mates: Bengt Johansson, Ilona Heldal, Dennis Björklund, Mia Indrika, Carl-Johan Lillieroth, and Niklas Eén. Thanks guys, there is nothing like that special bond with an office mate! (Though I must say I was fortunate enough to find an empty room all by myself when I got to the last phases of writing `:-)`.)

I have had many fruitful technical discussions over the years with Devdatt Dubhashi, Niklas Eén, Birgit Grohe, Jörgen Gustavsson, Thomas Hallgren, Rogardt Heldal, Peter Ljunglöf, Aarne Ranta, Andrei Sabelfeld, Josef Svenningsson, Niklas Sörensson, Walid Taha, Tuomo Takkula, Dag Wedelin, and many others at the department. Though not all of these discussions might have contributed to the contents of this thesis, they contributed to creating a vibrant working atmosphere in which I had the privilege to work in.

During the last couple of months, I was lucky enough to be able to share experiences (pleasant ones, but also some *ångest*) with my "brothers in arms" Per Bjesse, Jörgen Gustavsson, Rogardt Heldal, and Andrei Sabelfeld. We will all defend our theses within the same period of a few months. And soon we can all say: "We did it!".

Finally, I want to thank my family: my Mum, Dad, and my brother Bas, who have supported me all this time from far away, and my fiancé Elin, who supports me from nearby.

Chapter 1

Overview

This chapter gives an overview of the other chapters in the thesis, and presents background and related work.

1.1 Introduction

Currently, the most popular hardware description languages in use in industry are VHDL and Verilog. Both of these languages were designed in order to describe models of all kinds of hardware, ranging from the gate level to the system level. The need that is filled in by these languages in the hardware design process can be explained as follows [6]:

First, they allow a *structural* description of a system, that is, a system can be described in terms of its components and how these components are connected. Second, they allow a *behaviourial* description of a system, that is, the functionality of a component can be described algorithmically, in terms of normal programming language constructs. Third, designs can be efficiently *simulated*, so that designers can make decisions about design at an early stage, by comparing alternatives and testing without the need for fabricating actual hardware. Fourth, high-level descriptions of a system can be *synthesized*, in order to obtain the necessary detailed structure of a design at a lower level. Fifth, there is virtually no restriction on what kind of hardware systems can be modelled. VHDL, for example, allows the designer to specify the timing behaviour of signal delays and components in a very detailed way.

The generality of VHDL and Verilog leads to the following question: What is the *meaning* of a VHDL or a Verilog program? The current situation is that the various different simulation and synthesis tools do not at all agree on this subject. This is not very surprising, since the language designs for VHDL and Verilog seem to optimise the need for efficient simulation rather

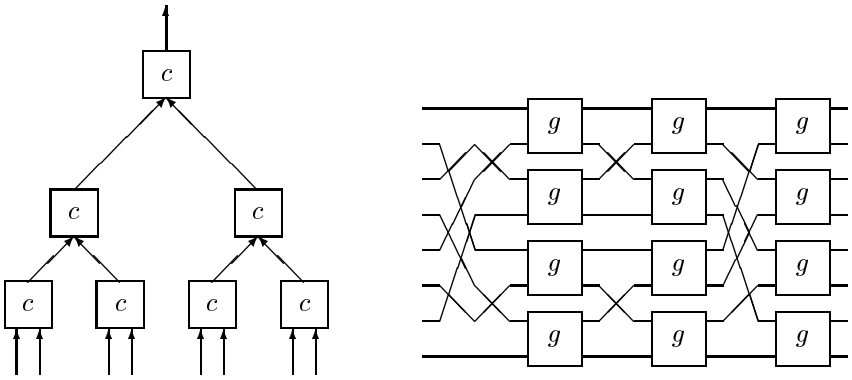


Figure 1.1: A tree-shaped circuit and a butterfly circuit

than ease of synthesis. This makes synthesis of behaviourial descriptions a very difficult process. The synthesis tools which are currently available are only able to perform correct synthesis for a small subset of the language. It is not an uncommon situation to design systems which behave differently before and after synthesis.

Another problem, which both VHDL and Verilog suffer from, is that there are inherent limitations in the language when making structural descriptions. In order to make the synthesis process easier, and to increase the amount of control over the low-level structure of the system, it is imperative that a good structural description of the system can be made. However, the abstraction mechanisms offered by VHDL and Verilog are far from expressive. In early versions, it was for example impossible to make structural descriptions parametrised by their input size. To alleviate this problem, the `generic` construct was later added to VHDL. However, in practice, it is still only used to describe generic linearly shaped patterns, like rows of (specific) components. As far as we know, the only way to describe more complicated patterns (like tree-shaped circuits or butterfly circuits, see Figure 1.1) in VHDL or Verilog, is by describing them for a particular size. It is not possible either to parametrise the descriptions by the used subcomponents (c and g in Figure 1.1).

Both of these problems, the lack of proper semantics and the lack of expressivity, have led to a major crisis in the field of hardware description. System designers out in the real world are crying out for proper semantic foundations and powerful language features. Conferences like FDL [38] and initiatives like SLDL [114] have been founded in order for people from academia and industry to unite so as to share ideas and experience and to overcome these problems.

This thesis presents material that is part of our modest contribution to solve some of these problems. However, we realise that the problems that for example VHDL and Verilog face are not easy! Therefore, we solve a much smaller problem, by limiting ourselves to only describing *synchronous structural* hardware.

This means we can use a much simpler semantic model for our circuits. In the next section, we describe the motivation and consequences for this decision. The hope is though that many of the ideas we present here can still be used for more general hardware description languages.

One of the main contributions of the thesis is to show that it is desirable and possible to have a hardware description language where the circuit descriptions are *first-class objects*. This means that:

- *Circuit descriptions can be generated.* We can write functions, that, given some parameters, generate a circuit. This is the equivalent of a parametrised circuit definition.
- *Circuit descriptions can be passed around as parameters.* We can write parametrised circuits which take other circuits as parameters. We call these circuits *connection patterns*.
- *Circuit descriptions can be analysed and transformed.* We can define functions that inspect the structure of a circuit and generate some result. An example of this is a longest combinational path analysis. The result of such a function can also be another circuit, allowing us to define circuit transformations. Examples of this are retiming transformations.

The other contributions of the thesis are:

- We have designed a formal circuit property specification language which is tightly integrated with our high-level design language.
- We have integrated (existing and home grown) automatic formal verification tools with our design system and specification language, in such a way that design, specification, correctness proof obligations, proof strategies, and implementation of a hardware system can all be described in the same language.

How to Read this Thesis. The thesis mainly presents two rather different versions of our hardware description system called *Lava* and topics related to the design and implementation of them. Each of the rest of the chapters in the thesis is a self-contained article or report, most of which have previously been published, and were co-authored with other people.

Chapter 1 (this chapter) gives an overview of the other chapters and presents some background. Chapter 2, written together with Per Bjesse, Mary Sheeran and Satnam Singh for a functional programming audience, presents the first version of Lava. Chapter 3 describes "observable sharing", an implementation technique we used in the implementation of the second Lava. Chapter 4, written together with John Hughes, describes "QuickCheck", a system with which we can describe and test properties of programs and circuits. Chapter 5, written

together with Mary Sheeran, is a tutorial of the second version of Lava, which is the version we currently prefer. Chapter 6, written together with Gordon Pace, shows how to use Lava as a framework for describing hardware compilation processes. Finally, Chapter 7, written together with Per Bjesse, presents a method of automatic formal verification of safety properties, which is not based on the usual Binary Decision Diagrams, but instead uses a propositional theorem prover.

1.2 Structural Synchronous Hardware

Structural versus behavioural description. The kind of hardware description we shall mainly deal with in this thesis is *structural* description. This means that we describe a circuit in terms of its components and how these components are connected. The components themselves are in turn described structurally, or can be provided as primitive building blocks by the system.

Another kind of description is *behavioural* hardware description, where a component can be described solely by specifying its behaviour or functionality. In such a behavioural description we can say what a circuit should do, but not exactly how it is done. This is handled by a compiler, also called *synthesiser*, which translates a behavioural program into structural hardware.

Many commercial hardware description languages, such as VHDL or Verilog, have a way of specifying hardware in a behavioural way. The reason we choose to mainly focus on structural description is three-fold. First, it seems the simplest usable level of hardware description one can work at. The semantics of behavioural VHDL or Verilog is very complicated; we want to intially avoid problems related to behavioural synthesis. Second, we want to formally verify properties of specific structural hardware implementations. The required reasoning has to happen at the component level, because for one behavioural description there are often many possible structural implementations. Third, from our descriptions, we want to generate configurations for reconfigurable hardware, such as FPGAs. Again, specifying information at the primitive component level is essential to get an optimal use of the area of the FPGA.

Note that later in Chapter 5 and more extensively in Chapter 6 we will see how to handle behavioural hardware description, based on our simple structural approach. This is possible as long as we are prepared to formally define a semantics for the behavioural description language that we want to use.

Synchronous versus asynchronous hardware. The kind of hardware we deal with in this thesis is *synchronous* hardware. In synchronous hardware, we assume that every component obeys the same omnipresent global clock. Synchronous circuits have a quite simple semantics; we can model them as functions from input bit streams to output bit streams. We do however *not* require the absence of combinational loops, i.e. cycles in the purely logical part

of the circuit. We deal with combinational loops in the way that Malik proposed in 1994 [69].

A more general approach is taken by *asynchronous* hardware, where different components listen to different and possibly unrelated clocks, or sometimes even to no clock at all. Asynchronous systems, and asynchronous hardware in particular, are very difficult to reason about. It is hard to find semantic models which are powerful enough to predict what is really going on in the circuit at the electronic level, and simple enough to reason with from a designer's point of view.

The restriction to synchronous hardware might seem a severe one, because it is clearly not as general as asynchronous hardware. These days, asynchronony is used more and more in designs, but mainly in two different ways. First, some smaller targeted components can sometimes be implemented in a more efficient way by using asynchronous logic. However, these components are usually given a synchronous interface and model. Also, lifting the restriction on combinational loops still allows many of these circuits to be considered synchronous.

Second, a system can work asynchronously at top level, but often its subcomponents still work synchronously. For large designs one is sometimes forced to design the system in this way, because of *clock skew*. This happens when the desired clock frequency is too high compared to the area of the circuit, so that one clock tick is not completely propagated over the chip before the next clock tick. In this case, the circuit can be split up into *isochronous* parts with their own clocks [2], a technique which is for example used in Globally Asynchronous Locally Synchronous systems (GALS).

Concluding, even if asynchrony is used more and more, it is still used in such a way that one can think of a main part of the design as synchronous.

The goal is set, we want to create a language for describing synchronous hardware, structurally. In the next section, we describe the approach we take in order to realise such a language.

1.3 The Embedded Language Approach

How do we design such a new language? One way is to take the traditional approach. We could first think of what kind of descriptions we would like to make, and design a syntax for them. Then we could decide on an appropriate underlying semantic model. After that we would have to implement a parser, an interpreter, a compiler. Lastly, we would make people learn our new language, and we would create debuggers and other development tools.

However, this takes a lot of time. It is difficult to design a good syntax, and it is a lot of work to write interpreters and compilers. The idea behind the embedded language approach is to *reuse* syntax, language features and tools of an already existing programming language. This language is usually called the

host language. To embed our new language in the host language, we develop libraries which allow us to define programs in our new language as *objects* in the host language.

Typically, such a library contains an *abstract datatype*, of which the embedded programs are elements, a number of *primitive* programs, these correspond to basic building blocks of the embedded language, some *combinators*, in order to write bigger programs, and one or more *run* functions, that allow executing or analysing embedded programs. Usually, also a particular *idiom* (a preferred style of writing the embedded programs) is provided, in order to avoid having to learn the full host language from scratch, before being able to write embedded programs.

An example of an embedded language is a language for specifying string grammars. A library implementing this would contain an abstract datatype (corresponding to grammars), some basic grammars (one-character grammars), combinators (sequencing and choice), and a run function (a function which parses a string according to a given grammar). A possible idiom would be to require grammars to be written as a sum-of-products, so that embedded programs look like BNF grammars.

The advantage of embedding a language is two-fold. First, writing programs in our new language means defining objects on the host language, so we can reuse all language tools (compilers, interpreters, debuggers), language features (the module system, the type system, the syntax), and even users (programmers who do not have to learn a new language) of the host language. Second, programs are now first class objects in the host language, so we can freely generate, and sometimes even inspect and transform programs we have written, all this in the same host language.

The embedded language approach has of course also some disadvantages, and indeed we must make a trade-off. Sometimes the features of the host language simply do not match the desired features of the embedded language. For example, we may want to use subtyping in the embedded programs, despite having a host language that does not support subtyping. Or the desired syntax of the embedded language may not fit with the syntax of the host language. In these cases, we can still use the embedded language approach in developing the language, using a work around such as an extra runtime pass that implements subtyping. When the language is finalised however, we probably want to develop a dedicated syntax, and implement processing tools such as compilers and interpreters.

1.4 A Simple Hardware Description Language

To demonstrate the idea of embedding a language, we present an embedding of a very simple hardware description language into the functional language Haskell [93]. This is a simplified version of the Lava language we present in

Chapter 5. We use Haskell to embed our language because it seems to have the right combination of properties that makes embedding easy to do and usable for the programmer. Haskell has successfully been used to embed various other languages (see for example [56, 55, 36, 88, 92] and many more).

We briefly present the simple approach to embedding of hardware description that we later take in Section 3.2.1. This approach is a simplification of the approach we took in the actual Lava system described in Chapter 5. We create a Haskell library, containing the definition of a type `Signal`, that models an input, output or internal wire in a circuit. For now, we can view the type `Signal` as a possibly infinite stream of booleans. This is sufficient because we only describe synchronous hardware.

```
type Signal = [Bool]
```

The above declares a new type `Signal`, to be the type `[Bool]`, which stands for the type of lists or streams of type `Bool`. Furthermore, the library contains definitions of operators that work on this type. These operators are the primitive components from which we can build circuits.

```
inv :: Signal -> Signal
reg :: Signal -> Signal
and :: (Signal, Signal) -> Signal
xor :: (Signal, Signal) -> Signal
```

The above are type declarations of two unary components: `inv` (logical inverse) and `reg` (a register), and two binary components: `and` and `xor` (an **and** and an **xor** gate). Under the assumption that the type `Signal` is implemented as a stream of booleans, a possible implementation of these components is the following.

```
inv bs      = map not bs
reg bs      = False : bs
and (as,bs) = zipWith (&&) as bs
xor (as,bs) = zipWith (/=) as bs
```

The Haskell function `map` lifts a unary operation on booleans to an operation that works on streams of booleans. The function `zipWith` does the same for binary operations. Note that function application in Haskell is written using juxtaposition: `map not bs` means the function `map` applied to the two arguments `not` and `as`. Lastly, the `(:)` operator adds a new element at the head of a list or stream, in effect shifting all the other elements to the 'next' position. It follows that the initial value of the register is `False`, corresponding to a low voltage.

Using these implementations of primitive components, we can describe bigger circuits. Here are two examples: a half adder, which is a basic component used to implement arithmetic circuits, and a toggle circuit, which is a simple *sequential* circuit (a circuit containing state).

```
halfAdd :: (Signal, Signal) -> (Signal, Signal)
halfAdd (a, b) = (sum, carry)
  where
    sum   = xor (a, b)
    carry = and (a, b)

toggle :: () -> Signal
toggle () = out
  where
    out  = inv out'
    out' = reg out
```

We use the `where` construct to make definitions of the output signals, and possibly local signals. The order of definitions in a `where` block does not matter.

For reasons that become apparent later, we consider `Signal` to be an abstract type, i.e. the circuit designer does not know how it is implemented.

1.5 Features of the Host Language

Even though the description language we have just defined is extremely simple, by embedding it in Haskell, we can already make powerful circuit descriptions. We can profit from the following language features that Haskell offers.

Data Structures and Recursion. In order to describe circuits that have multiple inputs, we can use Haskell's lists. Often, such circuits can be defined for any input size, a property that is called *genericity* in VHDL. A common way of defining such circuits is to use recursion. Here is an example of a *bit adder*, a circuit that adds one bit to a binary number of any size. The binary number is represented as a list of bits, with the least significant bit first.

```
bitAdder :: (Signal, [Signal]) -> ([Signal], Signal)
bitAdder (carryIn, [])   = ([],   carryIn)
bitAdder (carryIn, a:as) = (b:bs, carryOut)
  where
    (b,  carry)    = halfAdd  (carryIn, a)
    (bs, carryOut) = bitAdder (carry, as)
```

We use pattern matching on the argument, which is a list of signals. In Haskell, `[]` stands for the empty list, and `(:)`, which we saw earlier, matches a non-empty list by splitting it into the first element and the rest of the list.

This circuit can be used on binary numbers of any size. Other examples of data structures one can use as input or output types to circuits are tuples, trees, and records with named fields.

Polymorphism. Some circuits do not care about the type of the data of some of the input wires. This is often called *data independence*. One class of circuits

with this property is the class of *rewiring patterns*. Here are two examples of such polymorphic circuits. The first, `copy`, duplicates its in-signal, and the second, `zipp`, is the circuit equivalent of the standard Haskell function `zip`.

```
copy :: a -> (a, a)
copy inp = (inp, inp)

zipp :: ([a], [b]) -> [(a, b)]
zipp ([],   [])   = []
zipp (a:as, b:bs) = (a,b) : zipp (as, bs)
```

In Haskell, polymorphic types are expressed by using type names which start with a lower-case letter, in this case `a` and `b`. Note that it is not necessary for the circuit designer to write the types of the circuits, since Haskell has type inference. We just do so here for documentation purposes.

Other examples of rewiring patterns are `swap`, which switches the order of two elements in a tuple, and `unzipp`, which is the inverse of `zipp`.

Higher-Order Functions. A circuit description describing a shape that can be instantiated with different subcomponents is called a *connection pattern*. Connection patterns frequently occur when describing regularly structured circuits. We can realise them using higher-order functions, also called combinators [108, 85].

A very simple connection pattern is *serial composition*, denoted by the binary operator `->-`. It is realised as a circuit description parametrised by two circuits:

```
(->-) :: (a -> b) -> (b -> c) -> a -> c
(circ1 ->- circ2) inp = out
  where
    mid = circ1 inp
    out = circ2 mid
```

A simple example of the use of this connection pattern is a circuit that takes one bit as an argument, and uses a half adder to add that bit to itself.

```
selfAdd :: Signal -> Signal
selfAdd = copy ->- halfAdd
```

We can see that using connection patterns induces a new idiom for describing circuits.

Here is an other example of a connection pattern describing a *row* of subcomponents, connected through their respective carry wires (see Figure 1.2). The number of subcomponents depends on the number of inputs.

```
row :: ((c, a) -> (b, c)) -> (c, [a]) -> ([b], c)
row circ (carryIn, [])   = ([],   carryIn)
```

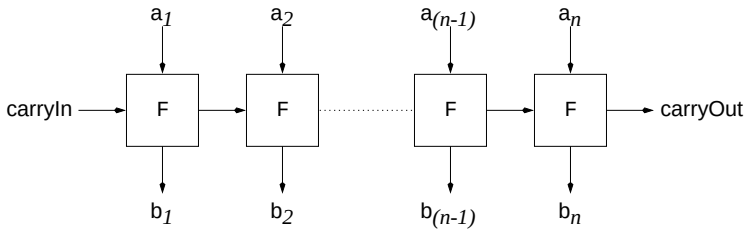


Figure 1.2: The connection pattern `row`

```
row circ (carryIn, a:as) = (b:bs, carryOut)
  where
    (b,  carry)    = circ     (carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

This pattern actually generalises the description of the bit adder we saw earlier. Using `row`, we can describe that bit adder alternatively as follows.

```
bitAdder :: (Signal, [Signal]) -> ([Signal], Signal)
bitAdder = row halfAdd
```

Note that connection patterns are often also polymorphic. Other examples of connection patterns are the dual to row, `column`, and their combination, `grid`, butterfly patterns, and parallel circuit composition. Sometimes, we even use connection patterns which can be instantiated with other connection patterns!

Overloading. Some circuit descriptions can handle most regular input types, such as signals, tuples, and lists thereof, but are not polymorphic, because the implementation of the functionality differs for each input type. In this case we use *overloading*. An example where we can use overloading is when we implement a general register circuit; a circuit that can delay and store input of any regular type. Let us call this circuit `delay`.

To overload delay in order for it to work on all regular types, we use Haskell's type classes. The type class mechanism allows a programmer to declare *properties* that a type can have. An example of such a property is called `Delayable`.

```
class Delayable a where
  delay :: a -> a
```

We can read this declaration as: A type `a` is delayable, if it supports the operation `delay`, with the specified type. In order to claim that a type has a property, we have to define the operations that come with the property, in this case the `delay` operation.

In the case where the type is a signal, we use a plain `reg`.

```
instance Delayable Signal where
  delay = reg
```

In the case where the type is a tuple or a list, we simply delay their components.

```
instance (Delayable a, Delayable b) => Delayable (a,b) where
  delay (a,b) = (delay a, delay b)

instance Delayable a => Delayable [a] where
  delay = map delay
```

And so on.

Other examples of general circuits that work for all regular types are the equality operator (`<==>`) and the muliplexer `mux`.

Laziness. Some circuits conceptually have an infinite size! This happens for circuits, which are size-generic in the number of outputs, but where the number of outputs does not depend on the number of inputs. In this case, we can say that the circuit has an infinite number of outputs. It depends on the way the component is used how many outputs will actually occur in the final circuit.

An example of such a circuit is a sequential counter. As an internal state, it has an (infinite) binary number. As an input, it takes one bit, and at every step, it adds this bit to the internal state, also outputting it. Here is a way we can define this circuit.

```
counter :: Signal -> [Signal]
counter carryIn = sum : sums
  where
    (sum, carryOut) = halfAdd (carryIn, sum')
    sums            = counter carryOut
    sum'            = reg sum
```

The size of this circuit is infinite. However, because of laziness, only the signals from the output we actually use will be computed or generated.

We already use laziness in the *implementation* of our hardware description language (we use infinite streams and datastructures with loops), but that is only visible to the implementor, and not something the hardware designer notices.

We have argued in favor of embedding a language because we can reuse the features of the host language in the embedding. A disadvantage is that it is often difficult to go beyond the features of the host language. In the case of hardware description in Haskell, there are some desirable features that we have to live without because we chose the embedded language approach.

Type System. Although Haskell's type system is very powerful, some desirable properties are not compatible with it. One such property is *sized types*. Many generic circuits have certain constraints on the sizes of their inputs, and information on how the sizes of their inputs relate to those of their outputs. An example is the `zipp` circuit, which assumes that both its arguments are of the same length, and produces an output of that same length. Other examples are

arithmetic circuits such as addition and multiplication. In the current implementation, we get a run-time error when we make a wrong assumption about these sizes. A type system knowing about sizes would be able to detect such a mistake at an earlier stage.

On the more general side, the type system should even be able to distinguish between *parameters* to circuit descriptions (for example the circuits in connection patterns, and in a less obvious way the length of input lists) and *inputs* to circuits. Currently, these are both presented as arguments to a function. However, there is a big difference between them.

Parameters should be considered *static*, i.e. known at circuit generation time. Circuit inputs are *dynamic*, i.e. provided to the circuit only after circuit generation time. It should not be allowed to mix these two levels. Parameters to a description, such as circuit size, are allowed to influence the shape of a circuit. But circuit inputs (the bits flowing through the wires) are certainly not allowed to do that. Currently however, it is possible to describe circuits in Lava which have this behaviour. It is possible to simulate these "circuits", but it will lead to an error during circuit generation time.

It would be more desirable if the type system could distinguish between static and dynamic arguments, such as is done in two- or multi-level languages [116]. Unfortunately, the Haskell type system does not allow us to express these kinds of restrictions.

1.6 Descriptions as First-Class Objects

Another advantage of embedding a language is the fact that programs in the embedded language become first-class objects in the host language. We can immediately profit from this by writing functions that *construct* circuits. In fact, we have already seen such functions; all parameterised circuits we have seen are examples of functions that, given the desired parameters, construct a circuit. Here is another example, a circuit that delays its argument n times according to its parameter.

```
delayN :: Delayable a => Int -> a -> a
delayN 0 inp = inp
delayN n inp = out
  where
    inp' = delay inp
    out  = delayN (n-1) inp'
```

Things get really interesting when we increase the potential complexity of the parameters. We can actually write *synthesisers*, functions that take a description of the functionality of a circuit and then generate a structural description of the circuit. We can see synthesisers as circuit descriptions parametrised by functionality descriptions. Examples of such functionality descriptions are truth

tables, state machines or even programs in a behavioural hardware description language! See Section 5.8 for some examples in some detail, and Chapter 6 for a more extensive discussion on this topic with respect to behavioural languages.

Having circuit descriptions as first-class objects does not only mean we can construct them, it also means that we can pass them around. Connection patterns are a nice example of this. Using connection patterns, the `delayN` circuit can be defined as follows:

```
delayN 0 = id
delayN n = delay ->- delayN (n-1)
```

We use the identity function `id` to represent the circuit that just wires its inputs to its outputs.

Analysing and Transforming Circuits. Unfortunately, there is not much else you can do with circuit descriptions as we have seen them so far. Since we model them as functions, the only thing we can really do with them is *apply* them, which means simulate them on inputs. There is no way we can *analyse* the circuit structure. We would like to do this for three reasons. First, being able to analyse circuits allows us to write functions that collect information about the circuit, for example its size or longest combinational path. Second, we would be able to generate explicit circuit descriptions in terms of VHDL, EDIF or logic, in order to implement the circuit or automatically verify functional properties of it. Third, by analysing the circuit, we could apply transformations on the circuit, such as *retiming transformations*.

A first attempt to do this might be the following. Instead of implementing signals as streams of booleans, we implement them as a datatype which explicitly keeps track of which gates that were used to construct it. Furthermore, a signal can also be an explicitly named input.

```
data Signal
  = Inv  Signal
  | Reg  Signal
  | And  (Signal, Signal)
  | Xor  (Signal, Signal)
  | Name String
```

An obvious implementation of the operators is as follows:

```
inv = Inv
reg = Reg
and = And
xor = Xor
```

In order to analyse the half adder circuit, we just evaluate the expression `halfAdd (Name "a", Name "b")`. The result is a concrete description of the circuit as an element in a Haskell datatype.

```
(Xor (Name "a", Name "b"), And (Name "a", Name "b"))
```

However, when we want to analyse the toggle circuit, we get into trouble because of the loop in the circuit. Evaluating `toggle (Name "x")` yields the infinite expression:

```
Inv (Reg (Inv (Reg (Inv (Reg ...
```

The problem is is that there is no way *within Haskell* to decide whether the above expression just represents a very large, possibly infinite circuit or a small circuit with a loop in it. At the heart of this lies the fact that Haskell does not provide a way to distinguish between the following two expressions:

```
let x = <expr>
 in ...  x ...  x ...
```

and

```
let x1 = <expr>
    x2 = <expr>
 in ...  x1 ...  x2 ...
```

This property, also called *referential transparency*, makes Haskell very suitable for mathematical reasoning; "one can replace equals for equals". On the other hand, when interpreted as circuits, the two expressions above lead to two quite different results; the latter circuit contains the component corresponding to `<expr>` twice. So, here we have an ultimate clash between the philosophy of the host language (Haskell's referential transparency) and the goal of the embedded language (definitions correspond to circuit components).

Chapter 3 discusses this problem and presents a number of solutions. The two solutions that are used in this thesis are the following.

- **Monads.** The monadic solution is the one we used in the version of Lava we present in Chapter 2. The idea is the following: If sharing information is important, we make it explicit. A convenient way to do this is by using a monad (see the next section). Using a monad `M` changes the circuit types from `Input -> Output` to `Input -> M Output`. The type `M` then keeps track of giving different names to each wire, and how the used components are connected. The style of programming changes drastically.
- **Observable sharing.** Using observable sharing is the solution we currently prefer, and is used in the version of Lava we present in Chapter 5. The idea is to extend Haskell with an extra feature that allows us to observe the sharing present in a certain datatype. The advantage is that we do not need to change our programming style. The disadvantage is that we have made a non-conservative extension to Haskell, meaning that we are not using the same programming language anymore. In Chapter 3 we investigate under what circumstances we can still use compilers and interpreters for Haskell.

1.7 Monads

Using monads is a way of dealing in a purely functional language with computations requiring features that otherwise might be considered impure, such as state or non-determinism [120]. The idea is to express these features in terms of a pure datastructure, the monad, much in the same way as when giving a denotational semantics to these features. There are many different kinds of monads, but they all share two common operators: *unit*, or `return` in Haskell, which creates a computation given a value, and *bind*, or (`>>=`), which sequences two computations into one computation.

Monad. More precisely, a monad is a type constructor, `M`, for which the following two operators are defined:

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

An object of type `M a` can be interpreted as "a computation of an object of type `a`, possibly making use of the features that `M` provides". The unit operation, `return`, takes an `a` and turns it into a (trivial) computation of that `a`. The bind operation, (`>>=`), combines a computation of an `a`, and a computation of a `b` that depends on the `a`, into a computation of a `b`.

Monad Examples. A monad that is often used is the *state monad*. It allows a computation to make use of a piece of state and update it. As an example, we present a monad that can generate unique identifiers (represented as strings) during a computation. The monad type is implemented as a function, which passes an integer number around as a state. The integer is increased every time a new identifier is needed.

```
type IdentifierM a = Int -> (a, Int)
```

A computation of type `IdentifierM a` must take the current integer as a parameter, and it returns the new integer as a result, together with the computed `a`.

```
return :: a -> IdentifierM a
return x = \n0 -> (x, n0)

(>>=) :: IdentifierM a -> (a -> IdentifierM b) -> IdentifierM b
m >>= k  = \n0 -> let (x, n1) = m n0
                      (y, n2) = k x n1
                  in (y, n2)
```

The `return` operator does not change the internal integer. Note that we use a lambda expression (in Haskell written as `\x -> f`) to create the function. The bind operator passes the new integer computed by the first computation

on to the next computation. Lastly, for a monad to be useful, we need specific operators that allow access to the provided features. In this case, we provide an operator `new`, that computes a fresh identifier, by turning the integer into a string, and updates the integer to use for the next computation.

```
new :: IdentifierM String
new = \n -> ("s" ++ show n, n+1)
```

Another often used monad is the *writer monad*, also called *output monad*. The feature provided by this monad does not influence the final computed value. During the computation, the monad gathers objects that are deemed as output. As an example, we present a monad that outputs *definitions* of signals during a computation. A definition is just a pair of a name (of type `String`) and a signal.

```
type Definition = (String, Signal)
type DefineM a  = (a, [Definition])
```

A computation of type `DefineM a` not only produces an `a`, but also a list of definitions. So, we implement it as a pair of the actual computed value and a list of definitions.

```
return :: a -> DefineM a
return x = (x, [])

(>>=) :: DefineM a -> (a -> DefineM b) -> DefineM b
m >>= k  = let (x, defs1) = m
               (y, defs2) = k x
           in (y, defs1 ++ defs2)
```

The `return` operator does not produce any such definitions, but the bind operator gathers the definitions for the final computation. The extra operator that we provide for this monad is called *define*. It produces no interesting result, just the empty tuple `()`, but outputs one definition.

```
define :: String -> Signal -> DefineM ()
define name sig = ((), [(name, sig)])
```

Monads for Describing Hardware. How are we going to use monads for describing hardware? The idea is that a hardware component is not represented as a function from input signals to output signals, but as a function from input signals to a *monadic computation* of output signals. In this way, we can be explicit about the cases where we want to have one component and share the output signals, or when we want to have multiple copies of the same component.

In the case of Lava, we have chosen a monad which outputs a netlist of gates, where each signal has its own name. This is implemented as a combination

of the two monads presented above: the `IdentifierM` monad to generate new names when we need them, and the `DefineM` to keep track of the netlist. (The new definitions of `return`, (`>>=`), `new` and `define` are not shown here.)

```
type LavaM a = Int -> (a, Int, [Definition])
```

For example, to implement an `and` gate, we could write the following:

```
and :: (Signal, Signal) -> LavaM Signal
and (x, y) =
  new >>= \idf ->
    define idf (And (x, y)) >>= \() ->
      return (Name idf)
```

On the first line of the body, we create a new identifier. On the second line, we make sure to remember what the definition of this identifier is. On the third line, we return the created identifier as the result of the computation.

Syntactic Support. It can be quite cumbersome to program in the style presented above, using the (`>>=`) combinator and lambdas explicitly. To alleviate this, Haskell provides syntactic support called the *do-notation*. Using that notation, the `and` gate described above would look like:

```
and :: (Signal, Signal) -> LavaM Signal
and (x, y) =
  do idf <- new
     define idf (And (x, y))
     return (Name idf)
```

We can see that instead of (`>>=`) and a lambda, the do-notation allows us to use a back arrow (`<-`) instead. If we are not interested in the result, such as on the second line, the back arrow can be completely left out.

Programs written in the do-notation can look like (impure) imperative programs. We should always keep in mind however that the do-notation is just syntactic sugar for a pure program using the bind operator.

1.8 Monads vs. Observable Sharing

We have developed two different versions of Lava, one using monads, and the other one using observable sharing. Currently, we prefer the latter version. In this section, we briefly discuss the main differences between using monads and observable sharing.

Programming Style. The programming style is the main argument in favor of using observable sharing. Circuits with feedback or loops are particularly

inconvenient to describe when we use a monadic style of programming. Usually, we introduce a *loop combinator*, a monadic operator which introduces a loop. Here is how one would define a simple sequential adder in monadic style:

```
adderSeq :: (Signal, Signal) -> LavaM Signal
adderSeq (a,b) =
  loopPair (\carryIn ->
    do (sum,carryOut) <- fullAdd (carryIn, (a,b))
       carryIn'       <- reg carryOut
       return (sum, carryIn')
  )
```

Here, we use the `loopPair` combinator, which takes a circuit with a pair as output, and feeds the right hand side of the pair back as input. We use a lambda expression in order to define the circuit to which we apply the loop combinator.

The left hand side of the pair becomes the output of the resulting circuit. There are a number of other typical loop patterns in circuits. Though every one of these can be expressed using the `loopPair` combinator, we have introduced a new combinator for each loop pattern, because there is often a lot of extra plumbing involved in expressing one loop combinator in terms of another.

Compare the above definition to the description in the functional style:

```
adderSeq :: (Signal, Signal) -> Signal
adderSeq (a,b) = sum
  where
    (sum,carryOut) = fullAdd (carryIn, (a,b))
    carryIn        = reg carryOut
```

It is easy to see why we prefer the latter style. Loop combinators are unfortunate because they introduce extra clutter in the code that is hard to motivate. They often force a restructuring of a circuit description, because we need to externalise the loops in a circuit just to be able to use a certain loop combinator.

Extending Haskell. Using observable sharing changes the properties of the host language—in fact, it is not the same language anymore. This is called a *non-conservative extension* to the language. This is the main objection against using observable sharing. However, in Chapter 3, we investigate the properties that a Haskell compiler must have in order not to destroy the sharing behaviour of a program. We believe that the popular Haskell interpreter Hugs has these properties. In practice, the major Haskell compiler GHC also behaves compatibly.

Relational definitions. Though capturing sharing information is the main motivation for introducing monads, monads can also be used for keeping track of other information. One possibility is to model circuits as *relations* instead of functions, a view which is also taken in the relational language Ruby [105]. This

is possible because a relation `A <-> B` can be modelled by a monadic function `A -> M B`, for a particular monad M.

This is implemented by having a monad with the following operations: `fresh`, which operationally creates unrestricted objects of a certain type, and relationally is the universal relation, and `assert`, which operationally requires its argument to be true, and relationally only relates true arguments.

Using these two operations, we can for example define a parametrised circuit combinator which creates the inverse of a relation:

```
inverse f y =
  do x  <- fresh
     y' <- f x
     assert (y <==> y')
     return x
```

A drawback is that these relations really only make sense in a logical interpretation of the circuits, i.e. when we perform verification. It is for example very difficult to simulate relations like this [59].

Laziness and Evaluation Order. A disadvantage of using monads is the impact they often have on evaluation order of subcomputations. In particular, when using a state monad, the subcomponents of circuits are created in the exact order the programmer has specified them. This leads to two problems. First of all, it is not possible to create infinite circuits, since all of the components have to be created before other components can be created afterwards. Second, it seems that in practice, the forced evaluation order due to state monads leads to worse space behaviour than generating and producing a tree-shaped datastructure lazily, as we do in observable sharing. However, we have no theoretical motivation to support this claim.

1.9 An Embedded Language for Properties

One of the goals of the Lava system is to formally verify properties about circuits. In order to specify these properties, we have developed a new embedded language. This language is heavily inspired by the property language of a system called QuickCheck, which we developed independently of the Lava system. In QuickCheck, one can specify properties of normal Haskell functions. These properties can then be automatically tested on random data. QuickCheck is described in Chapter 4.

A property of the Haskell functions `minimum` and `maximum` is stated in QuickCheck as follows:

```
prop_MinMax :: [Int] -> Property
prop_MinMax xs =
```

```
not (null xs) ==>
  minimum xs <= maximum xs
```

Read this property as follows: For all lists `xs`, if `xs` is not empty, then the minimum of `xs` is less than or equal than the maximum of `xs`. Function arguments are to be seen as implicitly universally quantified. The operator (`==>`), implication, is an example of a combinator provided by the QuickCheck library. The property can automatically be tested by applying it to random input a number of times (the default is 100):

```
Main> quickCheck prop_MinMax
OK, passed 100 tests.
```

The testing function `quickCheck` uses overloading to decide how to generate random elements in the input datatypes. One can also specify an explicit way of generating random data, using the `forAll` combinator.

```
prop_SumList4 :: Property
prop_SumList4 =
  forAll (list 4) $ \xs ->
    sum xs >= 4 * minimum xs
```

This can be read as: For all lists `xs` of length 4, the sum of all `xs` is at least as big as 4 times the minimum element. Here, `list 4` is a *data generator* that knows how to generate random lists of length 4. To avoid writing lots of parentheses, we use the (`$`) operator, which is function application.

As QuickCheck stands now, we can easily adapt it to test properties about circuit descriptions. Here is an example property: A half adder is commutative.

```
prop_HalfAddComm :: (Signal, Signal) -> Signal
prop_HalfAddComm (a,b) =
  halfAdd (a,b) <==> halfAdd (b,a)
```

We simply tell the `quickCheck` operator how to test properties defined using `Signal`s—namely by simulating them.

All that remains is to adapt the QuickCheck framework to include formal verification of properties as well. There is a problem however. It is possible to generate random elements in almost every Haskell datatype, but it is in general undecidable to formally verify properties that are universally quantified over these types. In fact, in Lava's formal verification, we only allow universal quantification over types whose size is statically known (so lists in general are not included, but booleans, pairs, and lists of fixed size are). The reason for this is that we use propositional logic to encode the properties.

There are two solutions to this: (1) we can make new Lava-specific implementations of the type classes for the data generators, and most other combinators

in QuickCheck, and restrict them to only work on finite types; (2) we can integrate formal verification and testing: where possible (on statically sized types), we universally quantify, otherwise we generate a random element in the type. We have investigated both of these methods, and we decided to incorporate the simplest one (the first one) in Lava.

To demonstrate the subtle differences between randomly testing and formally verifying, we present a simple example. We define a property that states that a binary adder is associative:

```
prop_AdderAssoc :: ([Signal],[Signal],[Signal]) -> Signal
prop_AdderAssoc (xs,ys,zs) =
  adder (adder (xs,ys),zs) <==>
    adder (xs,adder (ys,zs))
```

The above property can be tested by QuickCheck in the following way:

```
Main> quickCheck prop_AdderAssoc
OK, passed 100 tests.
```

In order to formally verify the property (this means to prove that it is true for all inputs), we need to be explicit about the input sizes. It is in general undecidable to prove these kinds of properties for all input sizes automatically. Suppose that we want all three inputs to have the same size, then we can write:

```
prop_AdderAssocForSize :: Int -> Property
prop_AdderAssocForSize n =
  forAll (list n) $ \xs ->
    forAll (list n) $ \ys ->
      forAll (list n) $ \zs ->
        prop_AdderAssoc (xs,ys,zs)
```

Fortunately, we can use a generator combinator called `triple` to make this a bit less painful to write:

```
prop_AdderAssoc :: Int -> Property
prop_AdderAssoc n =
  forAll (triple (list n)) $ \(xs,ys,zs) ->
    prop_AdderAssoc (xs,ys,zs)
```

Note that the implementations of `forAll` and `Property` are different from the ones we used in QuickCheck! We have just reused their names in the Lava property language. When we want to formally verify this property, we use the Lava function `verify`. However, we first have to pick a size `n` for which we want to verify this property:

```
Main> verify (prop_AdderAssoc 16)
Proving: ... Valid.
```

The property is verified, but *only* for inputs of size 16. So, we have chosen to either be able to randomly test a property for arbitrary input sizes, or to formally verify it for a particular size. An abvious generalisation is to randomly pick sizes for inputs, and then formally verify the resulting property. This is solution (2) mentioned earlier. We have indeed made a prototype implementation of this, but felt that the concept was too fragile to be introduced into Lava yet.

1.10 Verification using Temporal Induction

In order to formally verify the properties about the circuits described above, we can use existing verification tools. We have connected several tools to Lava, such as several implementations of Stålmarck's method [115, 44] for combinational verification, and model checking tools such as SMV [79] and VIS [119] for sequential verification.

Most current model checking tools use Binary Decision Diagrams (BDDs) [20] to verify properties. The BDDs are used internally to represent transition functions and sets of states. To verify a safety property, one usually calculates the set of reachable states, by starting with the set of initial states, and iterating the transition function until a fixpoint is reached. An excellent presentation of these techniques can be found in [26].

Unfortunately, the BDDs representing the transition function and the sets of states sometimes grow extremely large, sometimes even too large to be practical. Well-known circuits which are difficult to represent are multipliers and butterfly circuits (see Section 5.7.3). Therefore, we have developed a sequential verification method which is based on propositional logical theorem proving rather than BDDs.

The basic method we use is called temporal induction [106]. Temporal induction establishes a circuit property by proving that it holds for the circuit's initial state (base case), and, by assuming it holds for a certain state, proving that it also holds for the following states (step case). The base and step case can both be expressed as a propositional logic formula, thus making it possible to use a propositional logic theorem prover (sometimes called *satisfiabilty solver*, or *SAT solver*) to prove them rather than BDDs.

However, temporal induction has its restrictions. The main problem shows up when verifying relatively weak properties. In the step case of the induction, assuming that the property holds is simply not enough to establish the property in the following states – the induction hypothesis is too weak. This is a well-known problem when using induction as a proof method.

In Chapter 7, we propose a method which can automatically strengthen a given safety property. The strengthening is given in the form of an equivalence relation over the internal points of the circuit. The method calculates the greatest equivalence relation which is provable by induction, by means of a fixpoint iter-

ation. Calculating this strengthening greatly improves the usability of temporal induction in practice.

1.11 Related Work

μFP and Ruby. The work on the Lava family of description languages has its basis in earlier work by Sheeran on μFP, an extension of Backus' FP language to synchronous streams, designed particularly for describing and reasoning about regular circuits [110]. Like Backus' original FP, μFP advocates descriptions only using built-in connection patterns; one is not allowed to give names to inputs and outputs of circuits. This is one particular style of circuit description which we sometimes prefer, but is unfortunately not always desirable. It has the same disadvantages as the forced use of the loop connection pattern in monadic Lava. In fact, the μ-combinator in μFP is very much like the `loopPair` combinator, but it also introduces an extra delay, to ensure that every cycle contains a delay component.

A big advantage of the connection pattern style that μFP advocates is the ease of algebraic reasoning about circuit descriptions: Every built-in connection pattern comes with a set of algebraic laws. This idea is taken further in the relational hardware description language Ruby [105, 63, 104], which can be seen as the successor of μFP. In Ruby, circuits and circuit specifications are seen as relations on streams. This allows for a clean treatment of concepts like delay and anti-delay, non-determinism, and a uniform treatment of inputs and outputs. Ruby also supports built-in connection patterns that have an interpretation in terms of layout. Again, Ruby descriptions sometimes get awkward because one is forced to use the connection pattern style.

In Lava, we continue to use combinators for describing the ways in which circuits are built. But we also allow other common styles of description, where naming of arguments is allowed. In the monadic version of Lava, we even get some of the benefits of relational descriptions. Further, in Lava, we have the availability of a full-blown programming language, which allows the circuit designer to describe her own set of connection patterns.

HDRE and Hydra. One of the early embedded hardware description languages was John O'Donnell's HDRE, which stands for *Hardware Descriptions with Recursive Equations* [83]. The first version of HDRE was embedded in the non-strict functional language Daisy, and used lazy streams for simulating synchronous hardware. Later versions of HDRE have been implemented in Scheme, Miranda, and LML.

The most recent version of HDRE is embedded in Haskell and is now called Hydra [85]. Hydra has been a source of inspiration for Lava. Nowadays, Hydra is developed as a tool to teach circuit specification at an undergraduate level. It supports several basic signal datatypes at different levels of abstraction, such as voltages for the transistor level, bits for the gate level, and so on. The Hydra

system has not, as far as we know, been used to generate formulas from circuit descriptions for input to theorem provers. In order to generate netlists from Hydra descriptions, the circuit designer needs to annotate descriptions, so that cycles can be detected [84]. This topic is discussed more in Chapter 3.

Lustre and Pollux. The synchronous programming language Lustre, made by Halbwachs et al. [47], can be used to create synchronous dataflow components, called *nodes*, by means of structural composition. Lustre's nodes are very reminiscent of synchronous hardware components, and ideed there exists a variant of Lustre, called Pollux [99], which can be used to describe hardware. To aid regular circuit descriptions, Pollux offers arrays and recursion. One powerful feature, which Pollux inherited from Lustre, is the `when` construct, which allows concise descriptions of hierarchical clock schemes.

Lava's basic circuit idiom is heavily inspired by Lustre. Previous designs of Lava even contained some of the more Lustre-specific constructs, such as (1) the separation between delay and initialization of signals (resp. `pre` and `->` in Lustre), and (2) Lustre's `when` construct to describe hierarchical clock schemes. The reason why the current version of Lava does not contain these constructs is a very pragmatic one. We chose to have Lava's primitive components as close to hardware as possible, as to require as little *synthesis* as possible in the generation of actual, say, FPGA configurations. (The Pollux synthesizer provides optimisation of the `pre` and `->` constructs.) In hardware, delay and initialization of signals are implemented by one component. Singh (our collaborator at Xilinx) reports that he hardly needs hierarchical clocks in typical Lava applications, such as Digital Signal Processing (DSP) circuits. Currently, when multiple different clock signals are needed, we can use explicitly clocked delay components, and pass clock signals explicitly as a parameter to the circuits.

Extra features which Lava inherits from Haskell, and which Lustre or Pollux do not have, is a more advanced type system which allows for polymorphism and overloading, and the possibility to define connection patterns.

Hawk. Launchbury and his group have embedded a microarchitecure description language in Haskell. Their system is called Hawk [28]. Like in Hydra, there is a basic signal datatype that represents streams. In Hawk, these streams can carry objects from any Haskell datatype. In fact, Hawk circuits can be represented by any Haskell program. Architectures can thus be described at the behavioural level, which is a much higher level of abstraction. In this way, Hawk has served as a specification language for various superscalar microprocessor architectures.

The high level of abstraction of the descriptions makes it more difficult to generate netlists or to perform automatic verification of circuit properties. Instead, the Hawk group has been concentrating on doing algebraic proofs, which are performed by hand or assisted by a graphical tool or an interactive higher-order logic theorem prover, such as Isabelle [75].

When the datatypes flowing through streams in Hawk descriptions are restricted

to certain predefined abstract types, a form of symbolic evaluation of architectures becomes possible, which provides a way of automatic verification [33].

Interactive Theorem Provers. A different approach to circuit verification is taken by people from the interactive theorem proving community. The idea is to describe the circuit in a particular logic, and prove properties about the description using that logic. This has as a big advantage that properties of parametrised circuits can be proven once and for all for all parameters. The downside is that, often, the interactive proofs are quite tedious to do.

An example of work along these lines is done by Harrison. He uses his HOL-Light system, a higher-order logic theorem prover, to specify and verify floating-point unit algorithms [52]. Other examples of theorem provers used in hardware description and verification are PVS [31], in which one can use dependent types for the circuit descriptions, and ACL2 [41], in which one can use first-order logic to describe and verify circuits. ACL2 actually provides a link from its description language to real hardware.

PamDC. PamDC is a hardware description language, embedded in the popular programming language C++ [118, 12]. It is mainly used for simulating and generating configurations for Xilinx FPGAs. It uses C++ syntax overloading in a very elegant way to make descriptions look nice. For example, the assignment symbol = is overloaded for signals, so that the proper internal datastructure is built when making an assignment to a signal. (It is not clear however what happens when one makes two different assignments to the same signal, something which C++ does not prevent you from doing.) In fact, PamDC's idiom, like Lava's, is very close to Lustre.

In PamDC, it is easy to write generic circuits and circuit generators, because PamDC descriptions can use full C++. For example, a row of components can be expressed by a simple for-loop, for example. Generically sized circuits can be expressed using vectors.

In order to write connection patterns in a language, it should be possible to pass circuits as arguments. In PamDC, circuits are represented by C++ functions. However, C++ functions are not entirely first-class. They can be passed around, but not dynamically created. However, it is possible to encode this by using the class mechanism of C++. One way is to define a C++ class for every connection pattern, with a virtual function for each of its circuit arguments. When we want to instantiate the connection pattern, we make a subclass, where we define each virtual function to be the right argument. The only disadvantages of this are that the syntax is rather awkward, and that the number of connection patterns and instances used in a system should be known at compile time.

The PamDC documentation does not mention analysing or transforming described circuits. It is at present unknown to us if this is possible or not, though there is nothing that indicates that this is not technically possible.

Jazz. The language which is perhaps closest related to Lava, in terms of both its objectives and its features, is made by Frey, Berry, and Vuillemin et al., and

is called Jazz [40]. Jazz and Lava have been developed completely independently however! Jazz' syntax is inspired by Java, its type system is object oriented and supports polymorphism and subtyping, its semantics is purely functional and lazy, and its features include higher-order functions.

Since Jazz supports most of the features that allow Lava descriptions to look the way they look, and vice versa, there exists a quite simple mapping between most hardware descriptions in Jazz and in Lava, in both directions. There are some differences in idioms however. Jazz supports guarded definition blocks and arrays, which usually map to pattern matching and lists in Lava.

There are some differences in the type systems of the two languages as well. Jazz' datatypes are modelled after Java's objects, and the type system offers subtyping and type inference. An anomaly in the type system however requires every loop in a circuit to go through a `reg` component in order for it to be decidable. Haskell does not have subtyping, but instead offers type classes and overloading. One advantage with overloading in Lava compared to Jazz is that the `delay` component in Lava works on other structures than `Signal`s (in Jazz, `reg` only works on `Net`s).

Finally, in Lava, circuits are first-class objects, so we can for example define circuit analyses and circuit transformations within the language, which allows for transformed circuits to be used within other Lava circuits. In Jazz, circuits are processed using external tools.

1.12 Conclusions and Future Work

We have succeeded in embedding a structural hardware description language in the functional programming language Haskell. Links to many external tools, such as formal verification tools and VHDL processing tools are provided. The result is a system that assists hardware designers both in the initial stages of a design and in the final construction of a working circuit.

The implementation of the system, the provided interface to the hardware designer, and the circuit descriptions all heavily rely on the advanced programming features that Haskell offers: higher-order functions, polymorphism, type classes, laziness and monads. But even Haskell's syntax and type inference has made it possible to embed circuit descriptions with very little extra syntactic overhead.

The system is an interesting practical application of Haskell, which has proved to be an ideal tool, both as a hardware description language and as an implementation language. Our circuit descriptions are short and sweet, when one can find a suitable set of combinators.

On a more general level, we have thoroughly investigated the two orthogonal approaches of using monads and observable sharing to keep track of sharing behaviour. This issue frequently turns up when embedding languages in Haskell

[92, 37]. We believe we have found the right solution with observable sharing, at least for embedded languages where programs possibly contain loops.

The availability of a full-blown programming language has proven to be very powerful. In particular, we are able to write behavioural descriptions of circuits in Lava, which was chosen to be a structural description language from the beginning. This is possible because we can write parametrisable circuit descriptions. Parameters usually are integers or members of other simple datatypes to indicate size or shape of circuits, but they can in fact be of any Haskell type, including a datatype representing a behavioural description.

A big disadvantage of the embedded language approach is, as mentioned earlier, the fact that one is dependent on the features of the host language. The current design space of Lava has clearly set limitations, since we are naturally quite reluctant to make changes to our host language. If we for example need a change to Haskell's type system, we have to rely on the members of the Haskell committee and the compiler implementors. Compared to a from-scratch approach, as for example taken with the Jazz language, which has many of the rich language features of the our host language Haskell built-in, it is much more difficult for us to respond to the needs of the users of a particular embedded language, in this case hardware designers.

Future work includes a merge with Satnam Singh's version of Lava. Singh is our collaborator at Xilinx, Inc., and has made a version of Lava which includes combinators for specifying layout of FPGA configurations. These make it possible to be precise about how components are laid out on an FPGA, something which is extremely useful when generating FPGA configurations from recursive descriptions. Singh's combinators are monadic at the moment, but we believe we can adopt them to the functional style of descriptions we advocate in our most recent version of Lava. Singh has already made an experimental version of Lava which does precisely this.

Other future work consists of extending Lava with the right combinators for doing hierarchical verification. When verifying generically sized circuit descriptions for particular sizes, it is often the case that the difficulty of the verification problem increases exponentially with respect to an increasing size. This happens for example for multiplication circuits and sorting networks. The original (recursive) circuit descriptions contain useful structural information, which is lost after instantiating the circuit for a particular size by flattening the description. Knowing about the recursive structure of the description might be useful when doing the proof, even when the proof is done at a low level and for a particular size. By adding new combinators to Lava, we can for example add potential lemmas to subcomponents and substructures, which a propositional theorem prover might be able to prove and use in the overall proof.

Lastly, we would like to find an elegant and at the same time usable and understandable integration of randomly testing and formally verifying specified properties. We feel that random testing and formal verification are complementary methods, and that random input generation can avoid some common

pitfalls in formal verification.

Chapter 2

Lava: Hardware Design in Haskell

Lava is a tool to assist circuit designers in specifying, designing, verifying and implementing hardware. It is a collection of Haskell modules. The system design exploits functional programming language features, such as monads and type classes, to provide multiple interpretations of circuit descriptions. These interpretations implement standard circuit analyses such as simulation, formal verification and the generation of code for the production of real circuits.

Lava also uses polymorphism and higher order functions to provide more abstract and general descriptions than are possible in traditional hardware description languages. Two Fast Fourier Transform circuit examples illustrate this.

This chapter was written together with Per Bjesse, Mary Sheeran and Satnam Singh, and published at the International Conference on Functional Programming 1998 [17].

2.1 Introduction

The productivity of hardware designers has increased dramatically over the last 20 years, almost keeping pace with the phenomenal development in chip technology. The key to this increase in productivity has been a steady climb up through levels of abstraction. In the late seventies, designers sat with 'coloured rectangles' and laid out individual transistors. Then came the move through gate-level to register-transfer level descriptions, and the important step from schematic capture to the use of programming languages to describe circuits. Standard Hardware Description Languages like VHDL and Verilog have revolutionised hardware design.

However, problems remain. VHDL was designed as a simulation language, but now subsets of it are used as input to many kinds of tools, from synthesis engines to equivalence checkers. VHDL is poorly suited to some tasks, for example formal verification.

Ideally, we would like to be able to describe hardware at a variety of levels of abstraction, and to analyse circuit descriptions in many different ways. The analyses (or *interpretations*) that we consider to be essential are simulation (checking the behaviour of a circuit by giving it some inputs and studying the resulting outputs), verification (proving properties of the circuit), and the generation of code that allows a physical circuit to be produced. We want to be able to perform all of these tasks on one and the same circuit description.

The temptation to go away and design yet another hardware description language is strong, but we have resisted it. Instead, we would like to see how far we can get using the functional programming language Haskell. We call our design system Lava. The idea of using a functional hardware description langauge is, of course, not new, and the work described here builds on our earlier work on μFP [110] and Ruby [62], and on the use of non-standard interpretation in circuit analysis [113].

What is new about Lava is that we have built a complete system in which real circuits can be described, verified, and implemented. An earlier version of the system was used to generate filters and Bezier curve drawing circuits for implementation in a Field Programmable Gate Array based PostScript accelerator. Using the current system, very large combinational multipliers have been verified [111]. The largest formula produced so far from a circuit description had almost a million connectives. The system is constructed in a way that systematically makes use of important features of Haskell: monads, type classes, polymorphism and higher order functions.

We use ideas from Ruby, for example the use of combinators to build circuits, but in using Haskell, we gain access to a fully fledged programming language, with a rich type system and higher order functions. Having higher order functions available has greatly eased circuit description in real circuit examples. Circuits themselves still correspond to first order functions, but we use higher order

functions to construct circuit descriptions. Although we knew in theory that it is a good idea to have circuits as first class objects, we were surprised by how useful it is in practice. For example, higher order functions make it very easy to describe circuits containing look-up-tables. VHDL descriptions of such circuits tend to be long and hard to read, precisely because of the absence of suitable combinators. And even in Ruby, it is hard to deal with circuits that have a regular structure but components that vary according to their position in the structure.

Although we have moved from a relational to a functional programming language, we can retain as much of the generality of relations as we need, because the logical interpretation described later produces formulas that are relational, in that they do not distinguish between input and output. What we have lost, in moving away from Ruby, is machine support for high level design [104].

After choosing to use Haskell for hardware description, we again had two options: to make a Haskell variant and write specialised tools (compilers, synthesis engines and so on) to process it, or to make use of existing Haskell compilers by embedding a hardware description language in Haskell. Launchbury and his group are investigating the first option [28]. We chose the second.

2.2 Overview of the System

This section presents the types and abstractions used in the Lava system.

2.2.1 Monads

Dealing with an embedded language in a functional language requires a significant amount of information plumbing. A good way to hide this is to use *monads* [120]. Defining a monad means defining the language's features; a monadic expression is a program in the embedded language. Moreover, Haskell provides syntactic support and general combinator libraries for monads.

Let us take a look at a small example, and see how we can define a *half adder* circuit (figure 2.1):

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

This circuit has two input wires (bits) and two output wires. By convention, wires are grouped together so that a circuit always has one input value, and one output value. The `halfAdd` circuit consists of an `and` gate and an `xor` gate.

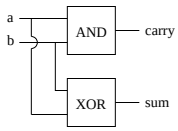


Figure 2.1: A half adder circuit

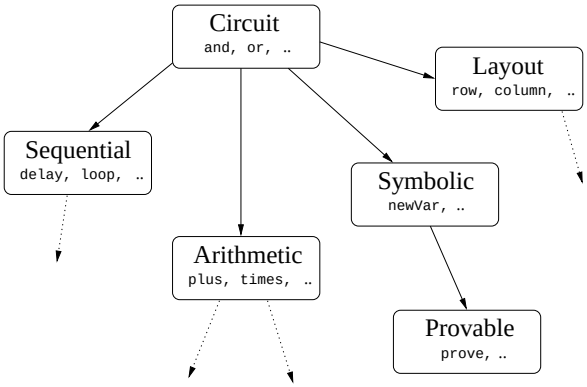


Figure 2.2: Type Class structure for Interpretations

Note that the type of a circuit description contains a type variable `m`, indicating that it is overloaded in the underlying monad. This means that we can later decide how to interpret the description by choosing an appropriate implementation of `m`. The same description can be interpreted in many ways, giving various different semantics to the embedded language. Examples of such *interpretations* are simulation (where we run the circuit on specific values), and the symbolic evaluation that is used to produce VHDL code.

2.2.2 Type Classes

Some circuit operations are meaningful only to certain interpretations; Lava is therefore structured with type classes (see figure 2.2). For example, a higher-level abstract circuit can deal with arithmetic operators, such as `plus` and `times`, where a physical circuit has no notion of numbers at all. We can point out groups of operations, which are supported by some interpretations but not by others, thus forming a hierarchy of classes.

The base class of the hierarchy is called `Circuit`. To be a `Circuit`, means to be a `Monad`, and to support basic operations like `and` and `or`.

```
class Monad m => Circuit m where
  and2, or2 :: (Bit, Bit) -> m Bit
  ...
```

Subclasses of `Circuit` are for example the `Arithmetic` class, for higher-level interpretations supporting numbers, and the `Sequential` class, for interpretations containing delay operations.

```
class Circuit m => Arithmetic m where
  plus, times :: (NumSig, NumSig) -> m NumSig
  ...

class Circuit m => Sequential m where
  delay :: Bit  -> Bit    -> m Bit
  loop  :: (Bit -> m Bit) -> m Bit
  ...
```

A circuit description will typically be constrained in the type to indicate what interpretations are allowed to run the description. The following circuit can only be run by interpretations supporting arithmetic:

```
square :: Arithmetic m => NumSig -> m NumSig
square x = times (x, x)
```

The architecture of the system makes it easy for the user to add new classes of operations to the hierarchy, and new interpretations that give semantics to them (see section 2.4.1).

2.2.3 Primitive Data Types

We use the datatype `Bit` to represent a bit. For now, this datatype can be regarded as just a boolean value, but we will slightly extend the datatype later (see section 2.3.2). We provide two constant values of this type:

```
data Bit = Bool Bool | ...

low, high :: Bit
low  = Bool False
high = Bool True
```

To describe circuits at a higher level, we add another primitive datatype, a `NumSig`, which represents an abstract wire through which numbers (integers) can flow. The `NumSig` wires will of course never appear in a physical circuit as an interpretation needs to be in the type class `Arithmetic` to handle this datatype.

```
data NumSig = Int Int | ...

int :: Int -> NumSig
int n = Int n
```

It is possible for the user to add other datatypes to Lava (see section 2.4.1).

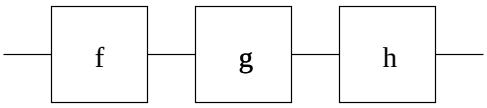


Figure 2.3: `compose` [f,g,h]

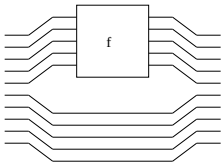


Figure 2.4: `one` f

2.2.4 Combinators

Common circuit patterns are captured using combinators which allow the designer to describe regular circuits compactly and in a way that makes the patterns explicit. This section describes some simple combinators that will be useful later.

The composition combinator `>->` passes the output of the first circuit as input to the second circuit. We also provide a version that works on lists (figure 2.3).

```
(>->) :: Circuit m
      => (a -> m b) -> (b -> m c) -> (a -> m c)

compose :: Circuit m => [a -> m a] -> (a -> m a)
compose = foldr (>->) return
```

The combinators `one` and `two` build circuits operating on $2n$-lists from circuits operating on n-lists. While `one f` applies the circuit `f` to one half of the wires and leaves the rest untouched, `two f` maps it to both halves (see figure 2.4 and 2.5).

```
one :: Circuit m
     => ([a] -> m [a]) -> ([a] -> m [a])

two :: Circuit m
     => ([a] -> m [b]) -> ([a] -> m [b])
```

Repeated application of a function is captured by `raised`: The expression `raised 3 two f` results in 8 copies of the `f` circuit, each applied to one eighth of the input wires.

```
raised :: Int -> (a -> a) -> (a -> a)
raised n f = (!! n) . iterate f
```

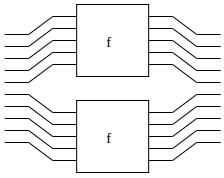


Figure 2.5: `two` f

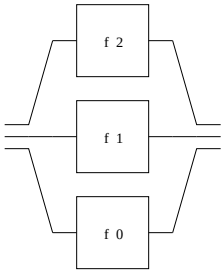


Figure 2.6: `decmap` 3 f

The circuit `decmap n f` processes an n-list of inputs by applying `f (n-1)`, `f (n-2)`, .. , `f 0` consecutively to each element (see figure 2.6).

```
decmap :: Circuit m
       => Int -> (Int -> a -> m b) -> ([a] -> m [b])
decmap n f = zipWithM f [n-1,n-2 .. 0]
```

The user can define new combinators as needed.

2.3 Interpretations

In this section, we present some interpretations dealing with concrete circuit functionality. *Standard* interpretations calculate outputs of a circuit, given input values. *Symbolic* interpretations connect Lava to external tools, by generating suitable circuit descriptions.

2.3.1 Standard Interpretation

The standard interpretation we present here is one that can only deal with *combinational* circuits, which have no notion of time or internal state. In this case, it suffices to use the identity monad since no side effects are needed.

```
data Std a = Std a
```

```
simulate :: Std a -> a
simulate (Std a) = a

instance Monad Std where ...
```

The resulting `Std` interpretation is integrated into the system by specifying the `Circuit` operations.

```
instance Circuit Std where
  and2 (Bool x, Bool y) = return (Bool (x && y))
  ...

instance Arithmetic Std where
  plus (Int x, Int y) = return (Int (x + y))
  ...
```

We can now simulate the example of section 2.2.1.

```
Hugs> simulate (halfAdd (high, high))
(high, low)
```

To deal with time and state, we can *lift* a combinational circuit interpretation into a sequential one. How this is done is not discussed here.

2.3.2 Symbolic Interpretation

Lava provides connection to external tools through the symbolic interpretations. These generate descriptions of circuits, rather than computing outputs. External tools process these descriptions, and in turn give feedback to the Lava system. The tools we focus on here are theorem provers. We briefly sketch other possibilities in section 2.3.5.

A circuit description is symbolically evaluated by providing abstract variables as input. The result of running the circuit is a symbolic expression representing the circuit. To implement this idea, we need some extra machinery. First of all, the signal datatypes are modified by adding a constructor for a variable, since a signal in this context can be both a value and a variable:

```
type Var = String

data Bit              data NumSig
  = Bool Bool           = Int Int
  | BitVar Var          | NumVar Var
```

It is important to keep the constructors of these datatypes abstract as the `Std` interpretation is unable to handle variables. By introducing the class `Symbolic`, we ensure that functions for variable creation are only available in interpretations which recognise variables.

```
class Circuit m => Symbolic m where
  newBitVar :: m Bit
  newNumVar :: m NumSig
```

When a circuit operation is applied to symbolic inputs, we create a fresh variable, and remember internally in the monad how this variable is related to the parameters of the operation.

An implementation for this interpretation is a state monad in an (infinite) list of unique variables, and a writer monad in a list of assertions. The type `Expression` is left abstract here.

```
type Sym a = [Var] -> (a, [Var], [Assertion])

data Assertion  = Var := Expression
type Expression = ...
```

The instance declaration for `Circuit Sym` is:

```
instance Circuit Sym where
  and2 (a, b) =
    do v <- newSymbol
       addAssertion (v := And [a,b])
       return (BitVar v)
  ...
```

When this interpretation is run on the half adder from section 2.2.1, the following internal assertion list is generated:

```
[ "b3" := And [ BitVar "b1", BitVar "b2" ]
, "b4" := Xor [ BitVar "b1", BitVar "b2" ]
]
```

The inputs to the circuit are called `"b1"` and `"b2"`.

2.3.3 Using a Symbolic Circuit

How can we now prove properties of circuits? We need to be able to formulate the circuit properties we want to verify. To do this, we create an *abstract* circuit that contains both the circuit and the property we want to prove.

To show a full adder with its leftmost bit set to False equivalent to a half adder, we write the question:

```
type Form = Bit

question :: Symbolic m => m Form
question =
```

```
do a <- newBitVar  -- free variables
   b <- newBitVar

   out1 <- halfAdd (a, b)
   out2 <- fullAdd (low, a, b)

   equals (out1, out2)
```

Two fresh variables `a` and `b` are given as inputs to both the half adder and the restricted full adder. The resulting formula (of type `Form`) is true if the outputs of these circuits are the same. The type `Form` is the same as `Bit`, so that we can use the logical operators (`and2`, `or2`, etc.) on both types.

The function `question` is polymorphic in the underlying interpretation; any symbolic interpretation is applicable. Here, we shall instantiate `m` with `Sym`.

2.3.4 Verification

The `Sym` interpretation is not very interesting on its own; it needs to be connected to the outside world in some way. The function `verify` takes a description of a question (which is of type `m Form`) and generates a file containing a (possibly very large) logical formula. This file is then processed by one of the automatic theorem provers that is connected to Lava by means of the `IO` monad.

```
verify :: Sym Form -> IO ProofResult

data ProofResult
  = Valid
  | Indeterminate
  | Falsifiable Model
```

The result from the theorem prover interaction has type `ProofResult` and indicates whether the desired formula was valid or not. If a countermodel (a valuation making the formulas false) can be found, it is also returned.

Using Hugs, the user of Lava can run proofs from inside the interpreter.

```
Hugs> verify question >>= print
Valid
```

This invocation generates input for a theorem prover, containing the variable definitions and the question, separated by an implication arrow:

```
AND( b3 <-> b1 & b2,      b4 <-> (b1 #! b2)
   , b5 <-> FALSE & b1,   b6 <-> (FALSE #! b1)
   , b7 <-> b6 & b2,      b8 <-> (b6 #! b2)
   , b9 <-> b5 # b7,      b10 <-> (b3 <-> b9)
```

```
, b11 <-> (b4 <-> b8), b12 <-> b10 & b11
) -> b12
```

Currently Lava interfaces to the propositional tautology checker Prover [115] and the first order logic theorem provers Otter [78] and Gandalf [117].

2.3.5 Other Interpretations

Using the same idea, we can generate input for other tools as well. An interesting target format is VHDL, which is one of the standard hardware description languages used in industry. There are many tools that can process VHDL, for purposes such as synthesis and efficient simulation.

Running the `Vhdl` interpretation on the half adder circuit (section 2.2.1) produces structural VHDL:

```
-- Automatically generated by Lava --
library circuit; use circuit.all;
entity halfadd is
  port ( b1, b2 : in std_logic;
         b3, b4 : out std_logic );
end halfadd;

library circuit; use circuit.all;
architecture structural of halfadd is
begin
  comp1 : and2 port map (b3, b1, b2);
  comp2 : xor2 port map (b4, b1, b2);
end structural;
```

An extended form of symbolic evaluation generates *layout* information. This is done by not only keeping track of how the components of a circuit are functionally composed, but also how they can be laid out on a gate array. `A >-> B` in this interpretation also indicates that `A` should be laid out to the left of `B`. Similarly, `row 5 fa` makes 5 full adders and lays them out horizontally with left to right data-flow.

The layout interpretation can generate VHDL and EDIF (another standard format) containing layout attributes that give the location of each primitive component.

Combining layout and behaviour in this way allows us to give economical and elegant descriptions of circuits, which in VHDL would require the user to attach complicated arithmetic expressions to instances.

2.4 An Example: FFT

This section illustrates how Lava is extended for signal-processing applications by the introduction of a complex number datatype and new combinators that allow two FFT circuits to be described.

The work presented here builds on previous work on deriving the FFT within Ruby [61] and specifying signal processing software in Haskell [14].

2.4.1 Complex numbers

Two flavours of complex numbers are needed for simulation and verification: concrete values and variables representing complex numbers. The implementation datatype `CmplxSig` reflects this:

```
data CmplxSig
  = Abstract NumSig
  | Concrete (Complex Double)
```

A complex datatype has to support operations like addition and multiplication. The FFT circuits also need *twiddle factors*, constants computed by `w` (see section 2.4.2). The appropriate operations are grouped together into a class.

```
class Arithmetic m => CmplxArithmetic m where
  cplus  :: (CmplxSig, CmplxSig) -> m CmplxSig
  ctimes :: (CmplxSig, CmplxSig) -> m CmplxSig
  ...
  w      :: (Int,Int) -> m CmplxSig

  cplus  = clift plus  (+)
  ctimes = clift times (*)
  ...

instance CmplxArithmetic Std where ...
instance CmplxArithmetic Sym where ...
```

To extend the existing interpretations with the complex datatype, we must write appropriate instance implementations. In this case it is simple, as the complex arithmetic operations can be implemented by lifting the existing arithmetic operations on symbolic `NumSig` variables and concrete `Complex Double` values. The twiddle factors have different meanings for different interpretations: the `Std` interpretation will get constant complex values, while `Sym` expects symbolic values.

2.4.2 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) computes a sequence of complex numbers X, given an initial sequence x:

$$X(k) = \sum_{n=0}^{N-1} x(n) \times W_N^{kn}, \qquad k \in \{0 \ldots N-1\}$$

where the constant W_N is defined as $e^{-j2\pi/N}$.

Each signal in the transformed sequence $X(k)$ depends on every input signal $x(n)$; the DFT operation is therefore expensive to implement directly.

The Fast Fourier Transforms (FFTs) are efficient algorithms for computing the DFT that exploit symmetries in the *twiddle factors* W_N^k. The laws that state these symmetries are:

$$\begin{aligned} W_N^0 &= 1 \\ W_N^N &= 1 \\ W_n^k \times W_n^m &= W_n^{k+m} \\ W_n^k &= W_{2n}^{2k}, \qquad (n, k \leq N) \end{aligned}$$

We will later use the fact that W_4^1 equals $-j$.

These laws, together with a restriction of sequence length (for example to powers of two), simplify the computations. An FFT implementation has fewer gates than the original direct DFT implementation, which reduces circuit area and power consumption. FFTs are key building blocks in most signal processing applications.

We discuss the description of circuits for two different FFT algorithms: the Radix-2 FFT and the Radix-2^2 FFT [54].

2.4.3 Two FFT circuits

The *decimation in time* Radix-2 FFT is a standard algorithm, which operates on input sequences of which the length is a power of two [97]. This restriction makes it possible to divide the input into smaller sequences by repeated halving until sequences of length two are reached. A DFT of length two can be computed by a simple *butterfly* circuit. Then, at each stage, the smaller sequences are combined to form bigger transformed sequences until the complete DFT has been produced.

The Radix-2 FFT algorithm can be mapped onto a combinational network as in figure 2.7, which shows a size 16 implementation. In this diagram, digits and twiddle factors on a wire indicate constant multiplication and the merging of two arrows means addition. The bounding boxes contain two FFTs of size 8.

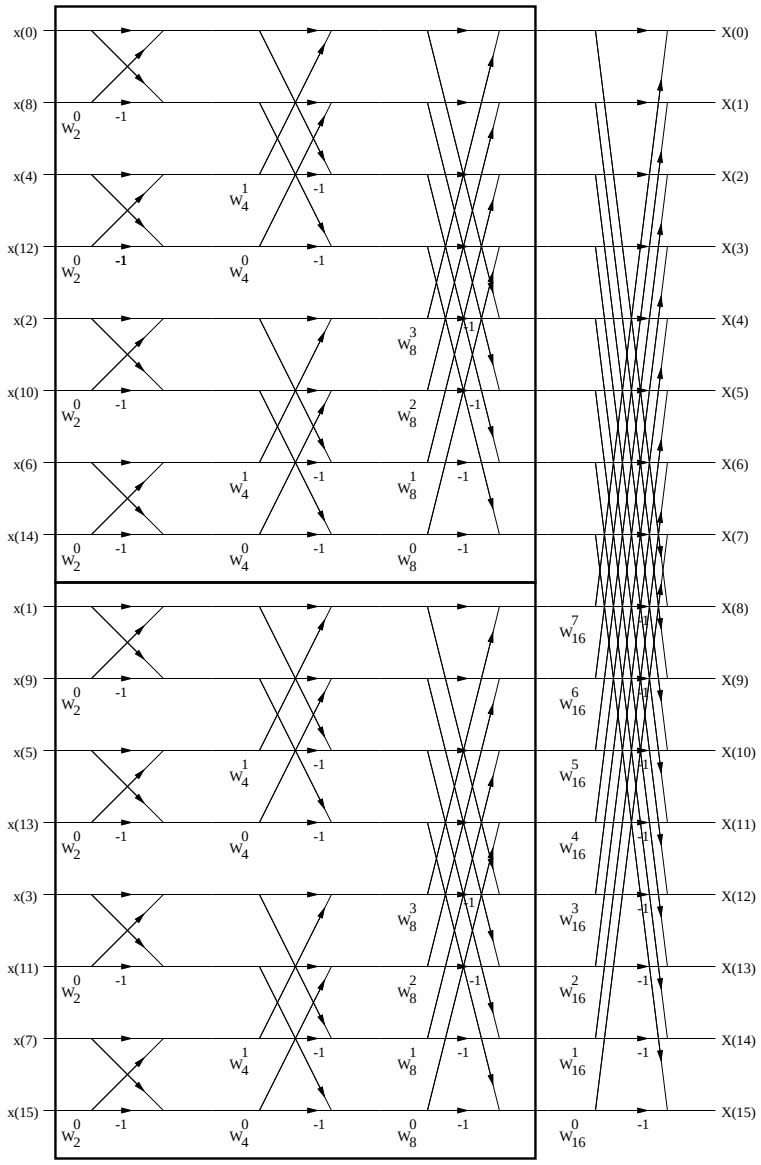


Figure 2.7: A size 16 Radix 2 FFT network

A less well-known algorithm for computation of the DFT is the *decimation in frequency* Radix-2^2 FFT, which assumes that the input length N is a power of four.

The corresponding circuit implementation (in figure 2.8) is also very regular and might be mistaken for a reversed Radix-2 circuit at a passing glance. However, it differs substantially in that *two* different butterfly networks are used in each stage, the twiddle factor multiplications are modified, and $-j$ multiplication stages have been inserted.

2.4.4 Components

We need three main components to implement FFT circuits. The first is a *butterfly circuit*, which takes two inputs x_1 and x_2 to two outputs $x_1 + x_2$ and $x_1 - x_2$ (see figure 2.9). It is the heart of FFT implementations since it computes the 2-point DFT. Systems of such components will be applied to the in-signals in many stages (figures 2.7 and 2.8).

The FFT butterfly stages are constructed by riffling together two halves of a sequence of length k, processing them by a column of $k/2$ butterfly circuits, and unriffling the result (see figure 2.10). Here `riffle` is the shuffle of a card sharp who perfectly interleaves the cards of two half decks.

```
bfly :: CmplxArithmetic m
     => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
  do o1 <- csubtract (i1, i2)
     o2 <- cplus (i1, i2)
     return [o1, o2]

bflys :: CmplxArithmetic m
      => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
  riffle >-> raised n two bfly >-> unriffle
```

Another important component of an FFT algorithm is multiplication by a complex constant, which can be implemented using a primitive component called a twiddle factor multiplier. This circuit maps a single complex input x to $x \times W_N^k$ for some N and k. The circuit `w n k` computes W_N^k.

```
wMult :: CmplxArithmetic m
      => Int -> Int -> CmplxSig -> m CmplxSig
wMult n k a =
  do twd <- w (n, k)
     ctimes (twd, a)
```

The multiplication of complete buses with $-j$ is defined as follows, using the fact that W_4^1 equals $-j$.

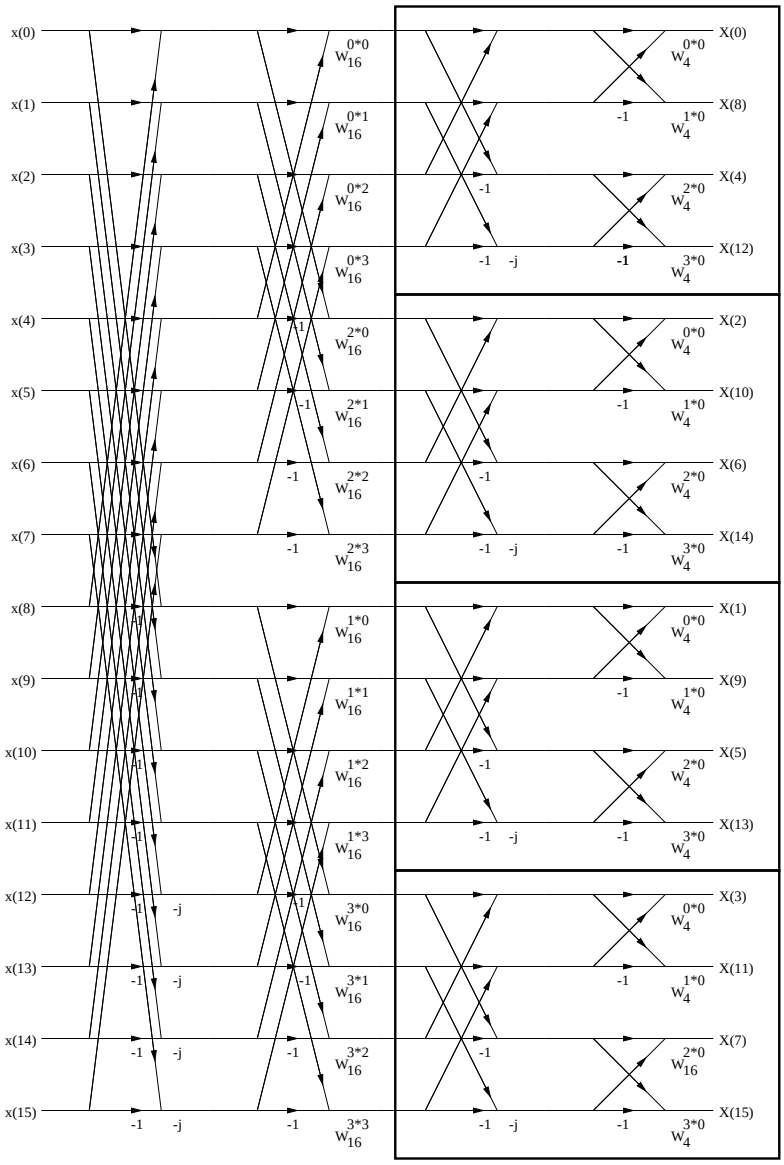


Figure 2.8: A size 16 Radix-2^2 FFT network

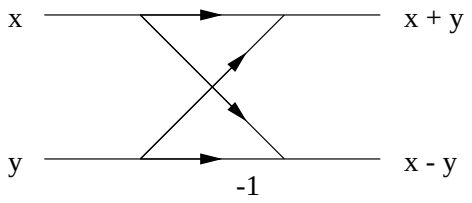


Figure 2.9: A butterfly

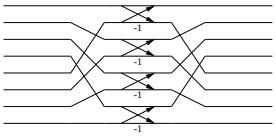


Figure 2.10: A butterfly stage of size 8 expressed with riffling

```
minusJ :: CmplxArithmetic m
       => [CmplxSig] -> m [CmplxSig]
minusJ = mapM (wMult 4 1)
```

Another useful component is the *bit reversal permutation*, used in the first or last stage of the FFT circuits. A new wire position is the reversed binary representation of the old position [97]. The permutation can be expressed using `riffle`:

```
bitRev :: Monad m => Int -> [a] -> m [a]
bitRev n =
  compose [ raised (n-i) two riffle
          | i <- [1..n]
          ]
```

Note that these components are not shown in the diagrams; either the in-data is permuted from the start, or the out-sequence needs to be rearranged.

2.4.5 The Circuit Descriptions in Lava

Inspired by the circuit diagrams we describe the two FFT circuits in Lava using higher-order combinators.

We begin by defining the type of an FFT parameterised by the interpretation monad `m`. A circuit description takes the exponent of the size of the circuit, and the list of inputs, and returns the outputs.

```
type Fft m = Int -> [CmplxSig] -> m [CmplxSig]
```

The Radix-2 FFT is a bit reversal composed with the different stages.

```
radix2 :: CmplxArithmetic m => Fft m
radix2 n =
  bitRev n >-> compose [ stage i | i <- [1..n] ]
 where
  stage i = raised (n-i) two
                $ twid i
              >-> bflys (i-1)

  twid  i = one (decmap (2^(i-1)) (wMult (2^i)))
```

The Radix-2^2 FFT is the sequence of stages composed with the final bit reversal.

```
radix22 :: CmplxArithmetic m => Fft m
radix22 m =
  compose [ stage i | i <- [m,m-1..1] ]
    >-> bitRev (2*m)
 where
  stage i = raised (m-i) (two.two)
                $ bflys (2*i-1)
              >-> one (one minusJ)
              >-> two (bflys (2*i-2))
              >-> twid i

  twid  i = column
               [ decmap (4^(i-1))
                          (wMult (4^i) . (wt *))
               | wt <- [3,1,2,0]
               ]
```

The corresponding VHDL descriptions would be several times longer.

2.4.6 Running Interpretations

We can now run some interpretations on our FFT circuits. Simulation is possible in the standard interpretation, if we provide an exponent and specific inputs to the circuit.

```
input :: [CmplxSig]
input = map cmplx [1:+4,2:+(-2),3:+2,1:+2]

Hugs> simulate (radix2 2 input)
[1.0:+6.0,(-1.0):+(-6.0),(-3.0):+2.0,7.0:+6.0]
```

The symbolic interpretation can be applied to verify that two circuit instances are equivalent, using the first order theorem prover Otter [78]. We create an abstract circuit stating the equivalence:

```
fftSame :: (Symbolic m, CmplxArithmetic m)
        => Int -> m Form
fftSame n =
  do inp <- newCmplxVector (4^n)

     out1 <- radix2 (n*2) inp
     out2 <- radix22 n inp

     equals (out1, out2)
```

The `newCmplxVector` function generates a list of complex symbolic variables. After applying both of the circuits to these inputs, we ask if the outputs are the same.

Before we can verify this equation, we have to add some knowledge to Otter: laws about complex arithmetic, and in particular the laws about twiddle factors. This information is added in the form of *theories*, which are defined by the user in Lava, and given to the prover as a proof option. Otter now shows circuit equivalence for size 4 FFTs (we have proven circuits of size 16 and 64 equivalent as well).

```
options :: [ProofOptions]
options = [ Prover otter
          , Theory arithmetic
          , Theory (twiddle 4)
          ]

Hugs> verify' options (fftSame 1) >>= print
Valid
```

Figure 2.11 shows the formula that is generated as input to Otter (notice the arithmetic and twiddle factor theory).

2.4.7 Related work on FFT description and verification

The equivalence of a Radix-2 FFT algorithm and the DFT has been shown using ACL2, a descendant of the Boyer-Moore theorem prover [41]. Our approach in the example is slightly different in that we want to show automatically generated logical descriptions of *circuits* of a fixed size equivalent, rather than proving mathematical theorems about the *algorithms*. The verifications are similar however, in that both methods use relationships between abstract twiddle factors.

2.5 Related Work

In this section, we discuss related work on the use of functional languages for hardware description and analysis.

The work described here has its basis in our earlier work on μFP, an extension of Backus' FP language to synchronous streams, designed particularly for describing and reasoning about regular circuits [110]. We continue to use combinators for describing the ways in which circuits are built. What we have gained through the embedding in Haskell, is the availability of a full-blown programming language. The synchronous programming languages Lustre, Esterel and Signal can all be used to describe hardware in much the style used here. Further experiments in this direction are being carried out in the EU project SYRF.

A source of inspiration has been John O'Donnell's Hydra system [85]. In Hydra, circuit descriptions are more direct because they are written in 'ordinary' Haskell. There are no monads cluttering up the types, and this must be an advantage. It is our use of monads, however, that makes Lava easily extensible, while Hydra is less so. The Hydra system has not, as far as we know, been used to generate formulas from circuit descriptions, for input to theorem provers, although the idea of having multiple interpretations has been a recurring theme in O'Donnell's work.

Launchbury and his group are experimenting with a different approach to using Haskell for hardware description [28]. In Hawk, a type of signals and Lustre-like functions to manipulate it are provided. Circuits are modelled as functions on signals, and the lazy state monad is used locally to express sequencing and mutable state. The main application so far has been to give clear and concise specifications of superscalar microprocessors. Simulation at a high level of abstraction has been the main circuit analysis method. Work on using Isabelle to support formal proof is under way, however. Also, it seems likely that Lava input could be generated from Hawk circuit descriptions. We plan to explore this possibility in a joint project. Hawk has, at present, no means of producing code for the production of real circuits, although work on circuit synthesis is in progress.

Keith Hanna has long argued for the use of a functional language with dependent types in hardware description and verification [51]. Hanna's work inspired much research on using Higher Order Logic for hardware verification. The PVS theorem prover, which is increasingly used in hardware verification [31], is also based on a functional language with dependent types. We do not know of work in which circuit descriptions written in this language are used for anything other than proof in PVS.

HML is a hardware description language based on ML, developed by Leeser and her group [67]. The language benefits from having higher order functions, a strong type system and polymorphism, just as ours does. The emphasis in HML is on simulation and synthesis, and not on formal verification.

2.6 Conclusions

The Lava system is an easily extensible tool to assist hardware designers both in the initial stages of a design and in the final construction of a working circuit. The system allows a single circuit description to be interpreted in many different ways, so that analyses such as simulation, formal verification and layout on a Field Programmable Gate Array are supported. Furthermore, new interpretations can be added with relatively little disturbance to the existing system, allowing us to use Lava as the main workbench for our research in hardware verification methods for combinational and sequential circuits. To be able to provide these features, we rely heavily on advanced features of Haskell's type system: monads for language embedding, polymorphism and type classes to support different interpretations, and higher order functions for capturing regularity.

The system is an interesting practical application of Haskell, which has proved to be an ideal tool, both as a hardware description language and as an implementation language. As demonstrated in the FFT examples, our circuit descriptions are short and sweet, when one can find a suitable set of combinators. Our experience with Ruby indicates that each domain of application (such as signal processing, pipelined circuits or state machines) gives rise to a small and manageable set of combinators.

The largest circuit that has been tackled so far is a 128 bit by 128 bit combinational multiplier. To deal with this circuit, we needed to use a Haskell compiler (HBC) rather than Hugs.

Writing the Lava system has been an educational exercise in software engineering. More than once, we have thrown everything away and started again. The latest version exploits Haskell's type system to impose a clear structure on the entire program, in a way that we find appealing. We have all been taught to think about types very early in the design of a system. Lava demonstrates the advantages of doing so.

2.7 Future Work

Until recently, we had several specialised versions of Lava, each concentrating on a particular aspect of design such as verification or the production of VHDL. The work of merging these versions has only just begun; it was really the need for fusion that pushed us towards the current system design. Incorporating the interpretation that takes care of layout production is a non-trivial task, as this code is necessarily large and complicated. This may lead to further changes to the top level design of Lava.

To make the system more usable, we need to add many new interpretations. For example, we would like to work on test pattern generation and testability

analysis, using earlier work by Singh as a basis [113]. All of these interpretations must be tested on real case studies.

We would be able to generalise our system further if multiple parameter type classes were provided in Haskell. At present, all of the interpretations share the same primitive datatypes. Using multiple parameter type classes, each interpretation could support its own data types, with the required features.

In the area of verification, we are working on interpretations involving sequential operations, such as delay, and on related methods to automatically prove properties of sequential circuits. We are working on a case study of a sequential FFT implementation provided by Ericsson CadLab. Inspired by the Hawk group, we find it hard to resist investigating verification of the next generation of complex microprocessors. In particular, we are interested in the question of how to *design* processors to enable verification to proceed smoothly.

```
%% Automatically generated by Lava %%

%% THEORY Arithmetic %%
list(demodulators).
eq(tim(x, plus(y, z)), plus(tim(x, y), tim(x, z))).
eq(tim(x, sub(y, z)), sub(tim(x, y), tim(x, z))).
eq(tim(1, x), x).
eq(tim(x, tim(y, z)), tim(tim(x, y), z)).
end_of_list.

%% THEORY Twiddle Factors size 4 %%
list(demodulators).
eq(W(x, 0), 1).
eq(W(x, x), 1).
$LE(x, 4) -> eq(W(x, y), W($PROD(2,x), $PROD(2,y))).
eq(tim(W(x,y),W(x,z)),W(x,$SUM(y,z))).
end_of_list.

%% SYSTEM + QUESTION %%
list(sos).
eq(x,x).
-eq(sub(sub(a4, tim(W(2,0), a2)), tim(W(4,1),
   sub(a3,tim(W(2,0), a1)))), tim(W(4,0),
   sub(sub(a4, a2),tim(W(4, 1), sub(a3, a1))))) |
-eq(sub(plus(a4, tim(W(2,0), a2)), tim(W(4,0),
   plus(a3, tim(W(2,0), a1)))), tim(W(4, 0),
   sub(plus(a4,a2), plus(a3, a1)))) |
-eq(plus(sub(a4, tim(W(2,0), a2)), tim(W(4,1),
   sub(a3,tim(W(2,0), a1)))), tim(W(4,0),
   plus(sub(a4, a2),tim(W(4,1), sub(a3, a1))))) |
-eq(plus(plus(a4, tim(W(2,0), a2)), tim(W(4,0),
   plus(a3, tim(W(2,0), a1)))), tim(W(4, 0),
   plus(plus(a4,a2), plus(a3, a1)))).
end_of_list.
```

Figure 2.11: Otter input for size 4 FFT comparison

Chapter 3

Observable Sharing

Pure functional programming languages have been proposed as a vehicle to describe, simulate and manipulate circuit specifications. We propose an extension to Haskell to solve a standard problem when manipulating data types representing circuits in a lazy functional language. The problem is that circuits are finite graphs – but viewing them as an algebraic (lazy) datatype makes them indistinguishable from potentially infinite regular trees. However, implementations of Haskell do indeed represent cyclic structures by graphs. The problem is that the sharing of nodes that creates such cycles is not observable by any function which traverses such a structure. In this paper we propose an extension to call-by-need languages which makes graph sharing observable. The extension is based on non updatable reference cells and an equality test (sharing detection) on this type. We show that this simple and practical extension has well-behaved semantic properties, which means that many typical source-to-source program transformations, such as might be performed by a compiler, are still valid in the presence of this extension.

This chapter was written together with David Sands, and a condensed version was published at the ASIAN'99 conference in Thailand, as an article titled 'Observable Sharing for Functional Circuit Description' [24].

3.1 Introduction

In this paper we investigate a particular problem of embedding a hardware description language in a lazy functional language – in this case Haskell. The "embedded language" approach to domain-specific languages typically involves the designing a set of combinators (higher-order reusable programs) for an application area, and by constructing individual applications by combining and coordinating individual combinators. There are a number of advantages in developing an embedded language rather than building a language from scratch. One advantage is that the embedded language can inherit desirable language properties and tools that the host language already provides. Examples of these which are relevant here are a strong type system, expressive syntax, higher order functions, compilers and interpreters. Further, if the host language enjoys a formal semantics and rich reasoning principles, then these can also be inherited by the the embedded language to formally reason about the embedded program. See [56] for examples of domain-specific languages embedded in Haskell.

In the case of hardware design the objects constructed are descriptions of circuits; by providing different interpretations of these objects one can, for example, simulate, test, model-check or compile circuits to a lower-level description. For this application (and other embedded description languages) we motivate an extension to Haskell with a feature which we call *observable sharing*, that allows us to detect and manipulate cycles in data-structures – a particularly useful feature when describing circuits containing feedback. Observable sharing is added to the language by providing immutable reference cells, together with a reference equality test. In the first part of the paper we present the problem and motivate the addition of observable sharing.

A problem with *observable sharing* that it is not a conservative extension of a pure functional language. It is a "side effect" – albeit in a limited form – for which the semantic implications are not immediately apparent. This means that the addition of such a feature risks the loss of many of the desirable semantic features of the host language. O'Donnell [84] considered a form of observable sharing (Lisp-style pointer equality `eq`) in precisely the same context (i.e., the manipulation of hardware descriptions) and dismissed the idea thus:

> " This ⟨pointer equality predicate⟩ is a hack that breaks referential transparency, destroying much of the advantages of using a functional language in the first place."

But how much is actually "destroyed" by this construct? In the second part of this paper we show – for our more constrained version of pointer equality – that in practice almost nothing is lost.

We formally define the semantics of the language extensions and investigate their semantic implications. The semantics is an extension to a call-by-need abstract machine which faithfully reflects the amount of sharing in typical Haskell implementations.

Not all the laws of pure functional programming are sound in this extension. The classic law of beta-reduction for lazy functional programs, which we could represent as:

$$\mathsf{let}\ \{x{=}M\}\ \mathsf{in}\ N\ =\ N[^{M}/_{x}] \quad (x \notin M)$$

does *not* hold in the theory. However, since this law could duplicate an arbitrary amount of computation (via the duplication of the sub-expression M, it has been proposed that this law is not appropriate for a language like Haskell [4], and that more restrictive laws should be adopted. Indeed most Haskell compilers (and most Haskell programmers?) do not apply such arbitrary transformations – for efficiency reasons they are careful not to change the amount of sharing (the internal graph structure) in programs. This is because all Haskell implemetations use a *call-by-need* parameter passing mechanism, whereby the argument to a function in a given call is evaluated at most once.

We develop the theory of operational equivalence for our language, and demonstrate that the extended language has a rich equational theory, containing, for example, all the laws of Ariola et al's call-by-*need* lambda calculus [4]. We also show that the semantics satisfies evaluation-order independence properties meaning that compiler optimisations such as strictness analysis will not change the semantics of programs.

The ideas in this paper are not only relevant to manipulating circuit descriptions. They appear useful for programming with other embedded description languages in Haskell. In the conclusion of the paper we mention two other possible applications.

3.2 Functional Hardware Description

We will deal with the description of synchronous hardware circuits in which the behaviour of a circuit and also its components can be modelled as *functions* from streams of inputs to streams of outputs.

The description is realised using an embedded language in the pure functional language Haskell. There are good motivations in literature for being able to use higher-order functions, polymorphism and laziness to describe hardware [110, 85, 28, 17]. In this paper however, we are concerned with some specific details related to the realisation of such an embedded language.

3.2.1 Describing Circuits

Viewing circuits as functions provides us with a way of embedding them in a functional language: as functions from in signals to out signals. This approach was taken as early as in the days of μFP [110], and later modernised in systems

like Hydra [85] and Hawk [28]. The following introduction to functional circuit description owes much to the description in [84].

Here are some examples of primitive circuit components modelled as functions. We assume the existence of a datatype `Signal`, which represents an input, output or internal wire in a circuit.

```
inv   :: Signal -> Signal
latch :: Signal -> Signal
and   :: Signal -> Signal -> Signal
xor   :: Signal -> Signal -> Signal
```

We can put these components together in the normal way we compose functions; by abstraction, application, and local naming. Here is an example of a circuit consisting of an and-gate and an xor-gate. It takes in two inputs and has two outputs.

```
halfAdd :: Signal -> Signal -> (Signal, Signal)
halfAdd a b = (xor a b, and a b)
```

Here is an example of a more complicated circuit. We use local naming of results of subcomponents using a `let` expression.

```
fullAdd :: Signal -> Signal -> Signal
        -> (Signal, Signal)
fullAdd a b c =
  let  (s1, c1) = halfAdd a b
       (s2, c2) = halfAdd s1 c
   in  (s2, xor c1 c2)
```

Here is a third example of a circuit. It consists of an inverter and a latch, put together with a loop, also called *feedback*. The result is a circuit that toggles its output.

```
toggle :: Signal
toggle =
  let  output = inv (latch output)
   in  output
```

Note how we express the loop; by naming the wire and using it recursively.

3.2.2 Simulating Circuits

By interpreting the type `Signal` as streams of bits, and the primitive components as functions on these streams, we can run, or *simulate* circuit descriptions with concrete input.

Here is a possible instantiation, where we model streams by Haskell's lazy lists.

```
type Signal = [Bool] -- possibly infinite

inv    bs = map not bs
latch  bs = False : bs
and as bs = zipWith (&&) as bs
xor as bs = zipWith (/=) as bs
```

We can simulate a circuit by applying it to inputs. Here are the results of evaluating some of the circuits we have defined so far:

```
> halfAdd [False, True] [True, True]
[(True, False), (False, True)]

> fullAdd [False, True] [True, True] [True, True]
[(False, True), (True, True)]

> toggle
[True, False, True, False, True, ...
```

As parameters we provide lists or streams of inputs and as result we get a stream of outputs. Note that the toggle circuit does not take any parameter and results in an infinite stream of outputs. The ability to both specify and execute (and perform other operations) hardware as a functional program is a claimed strength of the approach.

3.2.3 Generating Netlists

Suppose we have described a circuit component-wise in this way, and perhaps tested its behaviour. Now we want to realize the circuit – for example for implementation on an FPGA. Or perhaps we want to use a theorem prover or model checker to prove properties about the description. In all of these cases, we want to know what components the circuit are, and how they are connected. Such a description is usually called a *netlist*. We can reach this goal by *symbolic evaluation*. This means that we supply variables as inputs to a circuit rather than concrete values, and construct an expression representing the circuit.

In order to do this, we have to reinterpret the `Signal` type and its operations. A first try might be along the following lines. A signal is either a variable name (a wire), or the result of a component which has been supplied with its input signals.

```
data Signal
  = Var String
  | Comp String [Signal]

inv b   = Comp "inv"   [b]
```

```
latch b = Comp "latch" [b]
and a b = Comp "and"   [a, b]
xor a b = Comp "xor"   [a, b]
```

Now, we can for example symbolically evaluate a half adder:

```
> halfAdd (Var "a") (Var "b")
(Comp "xor" [Var "a", Var "b"],
Comp "and" [Var "a", Var "b"])
```

And, similarly a full adder. But what happens when we try to look at the toggle circuit?

```
> toggle
Comp "inv" [Comp "latch" [Comp "inv" [Comp "latch" ...
```

Since the `Signal` datatype is essentially a tree, and the toggle circuit contains a cycle, the result is an infinite structure. This is of course not usable as a symbolic description in an implementation. We get an infinite data structure representing a finite circuit.

We encounter a similar problem when we provide inputs to the half adder that are not simple variables, but the result of another component, for example an xor gate.

```
> halfAdd (xor (Var "x") (Var "y")) (Var "b")
(Comp "xor" [Comp "xor" [Var "x", Var "y"], Var "b"],
Comp "and" [Comp "xor" [Var "x", Var "y"], Var "b"])
```

The desired description here is *one* xor gate, whose output is a wire which is used twice in the half adder. Instead, because our signals are trees, the whole component is copied because sharing of subtrees cannot be expressed in the datatype. We have basically hit a wall because we have used trees (algebraic data types) to represent circuits, where as physically, circuits have a richer graph-like structure.

3.2.4 A Previous solution: Explicit Tagging

One possible solution, proposed by O'Donnell [84], is to give every use of component a unique tag, explicitly. The signal datatype is then still a tree, but when we then traverse that tree, we can keep track of what tags we have already encountered, and thus avoid cycles and detect sharing.

In order to do this, we have to change the signal datatype slightly by adding a *tag* to every use of a component. To make it easier to decorate every component in the circuit with a unique tag, an operator `(!)` is introduced that can create a new tag using an old tag and an integer.

```
data Signal = Var String
            | Comp Tag String [Signal]

type Tag = ...
(!) :: Tag -> Int -> Tag
```

Here are the definitions of the primitive components.

```
inv b   t = Comp t "inv"   [b]
latch b t = Comp t "latch" [b]
and a b t = Comp t "and"   [a, b]
xor a b t = Comp t "xor"   [a, b]
```

This is how the toggle circuit would look.

```
toggle :: Tag -> Signal
toggle t =
  let  wir = latch out (t!1)
       out = inv wir   (t!2)
   in  out
```

Though presented as "the first real solution to the problem of generating netlists from executable circuit specifications [...] in a functional language", it is awkward to use. A particular weakness of the abstraction is that it does not enforce that two components with the same tag are actually identical; there is nothing to stop the programmer from mistakenly introducing the same tag on different components.

3.2.5 Another Solution: the Monadic Approach

If explicit tagging is not the desired solution, why not let some underlying machinery take care of it? *Monads* are a standard approach for such problems (see e.g., [120]). The monadic approach is taken in Lava [17]. A monad `M` is a data structure that can abstract from an underlying computation model. A monadic computation resulting in something of type `a` has type `M a`.

A very common monad is the *state monad*, which threads a changing piece of state through a computation. We can use such a state monad to generate fresh tags for the signal datatype. Here is a simple implementation of the monad. Readers not familiar with the monadic style of programming in Haskell may safely skim through to the next section.

```
type Tag = Int
type M a = Tag -> (a, Tag)

return :: a -> M a
```

```
return a = \t -> (a, t)

(>>=) :: M a -> (a -> M b) -> M b
m >>= k = \t ->
  let (a, t') = m t  in  k a t'
```

There are two basic operations; `return` inserts a value into the computation type, and `>>=`, also called *bind*, sequences two computations, where the second can depend on the result of the first. Introducing a monad implies that the types of the primitive components and circuit descriptions become *monadic*, that is, their result is wrapped up in the monad type `M`.

```
inv   :: Signal -> M Signal
latch :: Signal -> M Signal
and   :: Signal -> Signal -> M Signal
xor   :: Signal -> Signal -> M Signal

inv b   = comp "inv"   [b]
latch b = comp "latch" [b]
and a b = comp "and"   [a, b]
xor a b = comp "xor"   [a, b]

comp name args =
  \t -> (Comp t name args, t+1)
```

A big disadvantage of this approach is not only that we must change of types, but also that the syntax must change. We can no longer use normal function abstraction and local naming anymore, we have to express this using the monadic bind `(>>=)`. This means that, just as in the previous solution, we have to change the definitions of the circuits. Here is what the half adder looks like in monadic style.

```
halfAdd :: Signal -> Signal -> M (Signal, Signal)
halfAdd a b =
  xor a b >>= \sum ->
  and a b >>= \carry ->
  return (sum, carry)
```

Another unwanted consequence of not being able to use local naming, is that we cannot use recursion anymore to express feedback in circuits. We have to define an explicit *monadic fixpoint* combinator to express loops.

```
loop :: (a -> M a) -> M a
loop f = \t -> let (a, t') = f a t in (a, t')
```

The definition of the toggle circuit serves as an example of how to introduce loops in this style:

```
toggle :: M Signal
toggle = loop (\out ->
  latch out >>= \wir ->
  inv wir   >>= \out' ->
  return out'
  )
```

All this turns out to be very inconvenient for the programmer. Furthermore, the monadic approach forces us to specify the order in which we create components in a circuit. This sequentiality is unnatural when specifying the components of circuits, which are more naturally thought of as executing in parallel. [1]

What we are looking for is a solution that does *not* require a change in the natural circuit description style of using local naming and recursion, but allows us to detect sharing and loops in a description from *within* the language.

3.3 Proposed Solution

The core of the problem is: a description of a circuit is basically a graph, but we cannot observe the sharing of the nodes from within the program.

The solution we propose is to make the graph structure of a program *observable*, by adding a new language construct. This can be done in several ways. In the beginning of this section we explain and motivate the particular constructs we chose to enrich the language. At the end, we will discuss and compare our choice with other possible solutions.

3.3.1 Objects with Identity

The idea is that we want the weakest extension that is still powerful enough to observe if two given objects have actually previously been created as one and the same object. The reason for wanting as weak an extension as possible is that we want to retain as many semantic properties from the original language as possible. This is not just for the benefit of the programmer – it is important because compilers make use of semantic properties of programs to perform program transformations, and because we do not want to write our own compiler to implement this extension.

Since we know in advance what kind of objects we will compare in this way, we choose to be explicit about this at *creation* time of the object that we might end

[1] Also, a possible problem that we noticed in practice when modelling larger circuits (e.g. multipliers) is that the linearisation of component creation seemed (in our programs) to have a disastrous effect on run-time memory behaviour of the Haskell program. Although these space-leaks could probably be fixed within the monadic program, it is gratifying to note that these problems evaporated when we used the solution described in the next section.

up comparing. In fact, you can view the objects as *non-updatable references*. We can create them, compare them for equality, and dereference them.

Here is the interface we provide to the references. We give a formal description of the semantics in section 3.4.

```
type Ref a = ...

ref   :: a -> Ref a
deref :: Ref a -> a
(<=>) :: Ref a -> Ref a -> Bool
```

The following two examples show how we can use the new constructs to detect sharing. In the first example, we create one reference, and compare it with itself, which yields `True`.

```
> let  x = undefined in  let  r = ref x in  r <=> r
True
```

In the second example, we create two *different* references to the same variable, and so the comparison yields `False`.

```
> let  x = undefined in  ref x <=> ref x
False
```

Thus, we have made a *non conservative extension* to the language; previously it was not possible to distinguish between a shared expression and two different instances of the same expression. We call the extension *observable sharing*.

In appendix A, we will present a possible implementation of references as presented here, suitable for most Haskell compilers. The extension makes use of a standard "unsafe" (side-effecting) operation that is included with most Haskell implementations, but is not part of the language proper.

3.3.2 Back to Circuits

How can we use this extension to help us to symbolically evaluate circuits? Let us take a look at the following two circuits.

```
circ1 =
  let  output = latch output
  in  output

circ2 =
  let  output = latch (latch output)
  in  output
```

In Haskell's denotational semantics, these two circuits would be identified, since `circ2` is just a recursive unfolding of `circ2`. But we would like these descriptions to represent different circuits; `circ1` has one latch and a loop, where as `circ2` has two latches and a loop. If the signal type includes a reference, we could compare the identities of the latch components and conclude that in `circ1` all latches are identical, where as in `circ2` we have two *different* latches.

3.3.3 A New Signal Type

We can now modify the signal datatype in such a way that the creation of identities happens transparently to the programmer. We play a similar trick as with the tagging, but instead we use references.

```
data Signal = Var String
            | Comp (Ref (String, [Signal]))

inv b   = comp "inv"   [b]
latch b = comp "latch" [b]
and a b = comp "and"   [a, b]
xor a b = comp "xor"   [a, b]

comp name args =  Comp (ref (name, args))
```

3.3.4 Subtleties of Sharing

Using the references we gain the ability to express different degrees of sharing of subcomponents. The subtlety is that the programmer must have a clear understanding of the sharing properties of the semantics. Here are two different definitions of `toggle`.

```
toggle1 =     -- wire
  let  output = inv (latch output) in  output

toggle2 () =  -- circuit
  let  output = inv (latch output) in  output
```

Without references, `toggle1` would be indistinguishable from `toggle2 ()`. But `toggle1` defines a *wire*. This means that it will only occur at most *once* as a component; every use of `toggle1` refers to the same wire. In contrast, `toggle2` defines a *circuit* with no inputs. Using `toggle2 ()` creates a *new* component every time you apply it.

Both views are relevant in circuit descriptions – but the programmer needs to be aware of such differences. To do this the programmer must understand the basics of the *call-by-need* execution mechanism. In `toggle1` the output

is executed (constructed) exactly once, the first time that it is needed. In the output is constructed once *for every application of the* `toggle2` *function*. Most reasonably experienced Haskell programmers are already aware of this difference; with observable sharing it becomes essential knowledge.

3.3.5 Other Possible Solutions

We present two other possible solutions, both are more or less well known extensions to functional programming languages.

Pointer Equality. The language is extended with an operator `(>=<) :: a -> a -> Bool` that investigates if two expressions are *pointer equal*, that is, they refer to the same bindings. There are a number of different semantics we can give to the extension; they involve how much evaluation of the arguments is done before comparing the pointers.

In our extension, we basically provide pointer equality in a more controlled way; you can only perform it on references, not on expressions of any type. This means we can implement our references using a certain kind of pointer equality:

```
type Ref a = a
ref a      = a
deref a    = a
r1 <=> r2  = r1 >=< r2
```

The other way around is not possible however, which shows that our extension is weaker. This corresponds to our goal, as explained at the beginning of this section.

Gensym. The language is extended with a new type `Sym` of abstract symbols with equality, and an operator that generates fresh such symbols, `gensym`.

```
type Sym = ...
gensym :: (Sym -> a) -> a
(==)   :: Sym -> Sym -> Bool
```

The symbol type and its operators can be implemented as a reference to the unit type:

```
type Sym = Ref ()
gensym f = f (ref ())
s1 == s2 = s1 <=> s2
```

The other way around is also possible; we implement a reference as a pair of a symbol and the value that the reference points to.

```
type Ref a          = (Sym, a)
ref a               = gensym (\s -> (s, a))
deref (_, a)        = a
(s1, _) <=> (s2, _) = s1 == s2
```

Since the approaches can be implemented in terms of each other, it is not clear which one is preferable. With the reference approach however, by get an important law by *definition*, which is:

$$\texttt{r1 <=> r2} = True \;\Rightarrow\; \texttt{deref r1} = \texttt{deref r2}$$

In the gensym approach, this becomes a proof obligation as part of the implementation.

3.4 The Semantic Theory

In this section we formally define the operational semantics of observable sharing, and study the induced notion of operational equivalence.

3.4.1 Language

For the technical development we work with a de-sugared core language based on an untyped lambda calculus with recursive lets, structured data, and case expressions.

The language of terms, Λ_{ref} is given by the following grammar:

$$\begin{array}{rcl} L, M, N & ::= & x \mid \lambda x.M \mid M\ x \mid c\,\vec{x} \\ & \mid & \mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ N \\ & \mid & \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i{\rightarrow}N_i\} \\ & \mid & M\,;N \\ & \mid & \mathsf{ref}\ x \mid \mathsf{deref}\ M \mid M \rightleftharpoons N \end{array}$$

where the term $M\,;N$ is *strict sequential composition*, corresponding to Haskell's ‘`seq`‘ operator. We include it here since it is useful for describing the compiler optimisations which follow *strictness analysis*. Note that we work with a restricted syntax in which the arguments in function applications and the arguments to constructors are always variables. It is trivial to translate programs into this syntax by the introduction of let bindings for all non-variable arguments. Such syntactic restrictions are common in compilation schemes. In this particular case we follow its use in the core language of the Glasgow Haskell compiler, *e.g.*, [90, 91], and in [65, 103]. Indeed, our language is essentially an untyped core of the intermediate language of the Glasgow Haskell Compiler, extended with immutable references and equality testing on references.

Throughout, x, y, z etc. will range over variables, and c will range over *constructors*. The set of *values*, $\mathrm{Val} \subseteq \Lambda_{\mathsf{ref}}$, ranged over by V and W are the constructor-expressions $c\,\vec{x}$ and the lambda-expressions $\lambda x.M$. The constructors are assumed to include the nullary constructors true and false. Constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1\,\cdots\,x_n$. We will write $\mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ N$ as a shorthand for

$$\mathsf{let}\ \{x_1{=}M_1, \ldots, x_n{=}M_n\}\ \mathsf{in}\ N$$

where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (i.e., all lets are potentially recursive). Similarly we write $\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i{\to}N_i\}$ for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\,\vec{x}_1{\to}N_1 | \cdots | c_m\,\vec{x}_m{\to}N_m\}.$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i{\to}N_i\}$.

The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

3.4.2 The Abstract Machine

The semantics for the standard part of the language presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [103]. Transitions in this machine are defined over configurations consisting of (i) a *heap*, containing a set of bindings, (ii) the expression currently being evaluated, and (iii) a *stack*, representing the actions that will be performed on the result of the current expression.

There are a number of possible ways to represent references in such a machine. One straightforward possibility is to use a global reference-environment, in which evaluation of the ref operation creates a fresh reference to its argument. The representation we present here is essentially equivalent, but syntactically more economical. Instead of reference environment, references are represented by a "hidden" constructor (i.e. a constructor which is not part of Λ_{ref}), which we denote by $\underline{\mathsf{ref}}$.

Let $\Lambda_{\underline{\mathsf{ref}}} \stackrel{\text{def}}{=} \Lambda_{\mathsf{ref}} \cup \{\underline{\mathsf{ref}}\,x \mid x \in \mathit{Var}\}$, and $\mathrm{Val}_{\underline{\mathsf{ref}}} \stackrel{\text{def}}{=} \mathrm{Val} \cup \{\underline{\mathsf{ref}}\,x \mid x \in \mathit{Var}\}$.

We write $\langle\, ?,\ M,\ S\,\rangle$ for the abstract machine configuration with heap $?$, expression $M \in \Lambda_{\underline{\mathsf{ref}}}$, and stack S. A *heap* is a set of bindings from variables to terms of $\Lambda_{\underline{\mathsf{ref}}}$; we denote the empty heap by $\emptyset$, and the addition of a group of bindings $\vec{x} = \vec{M}$ to a heap $?$ by juxtaposition: $?\,\{\vec{x} = \vec{M}\}$.

A stack is a list of stack elements. The stack written $b : S$ will denote the a stack S with b pushed on the top. The empty stack is denoted by ϵ, and the concatenation of two stacks S and T by ST (where S is on top of T). Stack elements are either:

- a variable x, representing the argument to a function,
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap,
- a group of case-alternatives, one of which will be chosen according to the outcome of the current evaluation,
- the second argument of a strict sequence, denoted $(; M)$, or
- a pending reference equality-test of the form $(\rightleftharpoons M)$, or $(\underline{\mathsf{ref}}\, x \rightleftharpoons)$,
- a dereference deref, indicating that the current computation should be dereferenced.

We will refer to the set of variables bound by $?$ as $\mathrm{dom}\,?$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. Since we cannot have more than one binding occurrence of a variable, a configuration is deemed *well-formed* if $\mathrm{dom}\,?$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(?, S)$ for their union. For a configuration $\langle ?,\ M,\ S \rangle$ to be closed, any free variables in $?$, M, and S must be contained in $\mathrm{dom}(?, S)$.

For sets of variables P and Q we will write $P \not\downarrow Q$ to mean that P and Q are disjoint, *i.e.*, $P \cap Q = \emptyset$. The free variables of a term M will be denoted $\mathsf{FV}\,(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}\,(\vec{M})$.

The abstract machine semantics is presented in figure 3.4.2; we implicitly restrict the definition to well-formed configurations.

The first group of rules are the standard call-by-need rules. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this evaluation is finished, and we may update the heap with the new binding for x.

Rules (*Unwind*) and (*Subst*) concern function application: rule (*Unwind*) pushes an argument onto the stack while the function is being evaluated; once a lambda expression has been obtained, rule (*Subst*) retrieves the argument from the stack and substitutes it into the body of that lambda expression.

Rules (*Case*) and (*Branch*) govern the evaluation of case expressions. Rule (*Case*) initiates evaluation of the case expression, with the case alternatives pushed onto the stack. Rule (*Branch*) uses the result of this evaluation to choose one of the branches of the case, performing substitution of the constructor's arguments for the branch's pattern variables.

$$\langle\, ?\,\{x = M\},\ x,\ S\,\rangle \to \langle\, ?,\ M,\ \#x : S\,\rangle \qquad (\mathit{Lookup})$$

$$\langle\, ?,\ V,\ \#x : S\,\rangle \to \langle\, ?\,\{x = V\},\ V,\ S\,\rangle \qquad (\mathit{Update})$$

$$\langle\, ?,\ M\ x,\ S\,\rangle \to \langle\, ?,\ M,\ x : S\,\rangle \qquad (\mathit{Unwind})$$

$$\langle\, ?,\ \lambda x.M,\ y : S\,\rangle \to \langle\, ?,\ M[y/x],\ S\,\rangle \qquad (\mathit{Subst})$$

$$\langle\, ?,\ \mathsf{case}\ M\ \mathsf{of}\ \mathit{alts},\ S\,\rangle \to \langle\, ?,\ M,\ \mathit{alts} : S\,\rangle \qquad (\mathit{Case})$$

$$\langle\, ?,\ c_j\ \vec{y},\ \{c_i\ \vec{x}_i \to N_i\} : S\,\rangle \to \langle\, ?,\ N_j[\vec{y}/\vec{x}_j],\ S\,\rangle \qquad (\mathit{Branch})$$

$$\langle\, ?,\ \mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ N,\ S\,\rangle \to \langle\, ?\,\{\vec{x} = \vec{M}\},\ N,\ S\,\rangle \quad \vec{x} \natural \mathrm{dom}(?, S) \qquad (\mathit{Letrec})$$

$$\langle\, ?,\ M\,;N,\ S\,\rangle \to \langle\, ?,\ M,\ (;N) : S\,\rangle \qquad (\mathit{Seq1})$$

$$\langle\, ?,\ V,\ (;N) : S\,\rangle \to \langle\, ?,\ N,\ S\,\rangle \qquad (\mathit{Seq2})$$

$$\langle\, ?,\ \mathsf{ref}\ M,\ S\,\rangle \to \langle\, ?\,\{x = M\},\ \underline{\mathsf{ref}}\ x,\ S\,\rangle \quad x \notin \mathrm{dom}(?, S) \qquad (\mathit{Ref})$$

$$\langle\, ?,\ \mathsf{deref}\ M,\ S\,\rangle \to \langle\, ?,\ M,\ \mathsf{deref}\ : S\,\rangle \qquad (\mathit{Deref1})$$

$$\langle\, ?,\ \underline{\mathsf{ref}}\ x,\ \mathsf{deref}\ : S\,\rangle \to \langle\, ?,\ x,\ S\,\rangle \qquad (\mathit{Deref2})$$

$$\langle\, ?,\ M \rightleftharpoons N,\ S\,\rangle \to \langle\, ?,\ M,\ (\rightleftharpoons N) : S\,\rangle \qquad (\mathit{RefEq})$$

$$\langle\, ?,\ \underline{\mathsf{ref}}\ x,\ (\rightleftharpoons N) : S\,\rangle \to \langle\, ?,\ N,\ (\underline{\mathsf{ref}}\ x \rightleftharpoons) : S\,\rangle \qquad (\mathit{Ref1})$$

$$\langle\, ?,\ \underline{\mathsf{ref}}\ y,\ (\underline{\mathsf{ref}}\ x \rightleftharpoons) : S\,\rangle \to \langle\, ?,\ b,\ S\,\rangle \quad b = \begin{cases} \mathsf{true} & \text{if } x = y \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad (\mathit{Ref2})$$

Figure 3.1: Abstract machine semantics

Rule (*Letrec*) adds a set of bindings to the heap. The side condition ensures that no inadvertent name capture occurs, and can always be satisfied by a local α-conversion.

Rules (*Seq1*) and (*Seq2*) implement the Haskell-style strict sequential evaluation, by first evaluating the left argument, then discarding the result in favour of the right argument.

The second collection of rules concern the observable sharing. Rule (*RefEq*) first forces the evaluation of the left argument, and (*Ref1*) switches evaluation to the right argument; once both have been evaluated to ref constructors, variable-equality is used to implement the pointer-equality test.

3.4.3 Convergence

Two terms will be considered equal if they exhibit the same behaviours when used in any program context. The behaviour that we use as our test of equivalence is simply termination. Termination behaviour is formalised by a conver-

gence predicate:

Definition 3.4.1 (Convergence) *A closed configuration* $\langle ?,\ M,\ S \rangle$ converges, *written* $\langle ?,\ M,\ S \rangle\Downarrow$, *if there exits heap* Δ *and value* V *such that*

$$\langle ?,\ M,\ S \rangle \rightarrow^* \langle \Delta,\ V,\ \epsilon \rangle.$$

We will also write $M\Downarrow$, identifying closed M with the initial configuration $\langle \emptyset,\ M,\ \epsilon \rangle$.

Closed configurations which do not converge are of four types: they either (i) reduce indefinitely, or get stuck because of (ii) a type error, (iii) a case expression with an incomplete set of alternatives, or (iv) a *black-hole* (a self-dependent expression as in let $x{=}x$ in x). All non-converging closed configurations will be semantically identified.

3.4.4 Approximation and Equivalence

To define equivalence we take the standard approach of defining a contextual preorder which says that M approximates (is less than) N in the ordering if whenever a program containing M terminates, then replacing M by N will not worsen the termination behaviour. Let $\mathbb{C}$, $\mathbb{D}$ range over *contexts* – terms containing zero or more occurrences of a *hole*, $[\cdot]$ in the place where an arbitrary subterm might occur. Let $\mathbb{C}[M]$ denote the result of filling all the holes in $\mathbb{C}$ with the term M, possibly causing free variables in M to become bound.

Definition 3.4.2 (Operational Approximation) *We say that* M operationally approximates N, *written* $M \lesssim N$, *if for all* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are closed,*

$$\mathbb{C}[M]\Downarrow \implies \mathbb{C}[N]\Downarrow.$$

We say that M and N are *operationally equivalent*, written $M \cong N$, when $M \lesssim N$ and $N \lesssim M$. Note that equivalence is a non-trivial equivalence relation; In Figures 3.2 and 3.3 we present a collection of basic laws of equivalence. As usual with a "semantic" definition of equivalence, it is not a recursively enumerable relation. In the statement of all laws, we follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables.

Remark: The fact that the reference constructor ref is abstract (not available directly in the language) is crucial to the variable-inlining properties. For example a (derivable) law like

$$\text{let } \{x{=}z\} \text{ in } N \cong N[^z/_x]$$

would fail if terms could contain ref. This failure could be disasterous in some implementations, because in effect a configuration-level analogy of this law is applied by some garbage collectors.

$$(\lambda x.M)\, y \cong M[y/x]$$
$$\mathsf{case}\ c_j\ \vec{y}\ \mathsf{of}\ \{c_i\ \vec{x}_i \to M_i\} \cong M_j[\vec{y}/\vec{x}_j]$$
$$\mathsf{let}\ \{x{=}V, \vec{y}{=}\vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x{=}V, \vec{y}{=}\vec{\mathbb{D}}[V]\}\ \mathsf{in}\ \mathbb{C}[V]$$
$$\mathsf{let}\ \{x{=}z, \vec{y}{=}\vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x{=}z, \vec{y}{=}\vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z]$$
$$\mathsf{let}\ \{x{=}z, \vec{y}{=}\vec{M}\}\ \mathsf{in}\ N \cong \mathsf{let}\ \{x{=}z, \vec{y}{=}\vec{M}[z/x]\}\ \mathsf{in}\ N[z/x]$$

Figure 3.2: Beta laws for call-by-need.

$$\mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ N \cong N, \quad \text{if } \vec{x} \not\downarrow \mathsf{FV}(N)$$
$$\mathsf{let}\ \{\vec{x}{=}\vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{y}{=}\vec{M}\}\ \mathsf{in}\ N$$
$$\cong \mathsf{let}\ \{\vec{x}{=}\vec{L}, \vec{y}{=}\vec{M}\}\ \mathsf{in}\ N$$
$$\mathsf{let}\ \{x{=}\mathsf{let}\ \{\vec{y}{=}\vec{L}, \vec{z}{=}\vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N'$$
$$\cong \mathsf{let}\ \{x{=}\mathsf{let}\ \{\vec{z}{=}\vec{M}\}\ \mathsf{in}\ N, \vec{y}{=}\vec{L}\}\ \mathsf{in}\ N'$$
$$\mathbb{C}[\mathsf{let}\ \{\vec{y}{=}\vec{V}\}\ \mathsf{in}\ M] \cong \mathsf{let}\ \{\vec{y}{=}\vec{V}\}\ \mathsf{in}\ \mathbb{C}[M]$$

Figure 3.3: Laws for dealing with lets.

$$\mathbb{R}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \cong \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{R}[N_i]\}$$
$$\mathbb{R}[\mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ N] \cong \mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ \mathbb{R}[N]$$
$$\mathsf{let}\ \{x{=}M\}\ \mathsf{in}\ \mathbb{R}[x] \cong \mathbb{R}[M], \quad \text{if } x \notin \mathsf{FV}(M, \mathbb{R})$$

Figure 3.4: Laws for Reduction Contexts.

$$\begin{array}{rcl@{\qquad}rcl}
M \rightleftharpoons N & \cong & N \rightleftharpoons M & V\,;N & \cong & N \\
x \rightleftharpoons x & \sqsupseteq & \mathsf{true} & M\,;N & \sqsupseteq & N \\
\mathsf{ref}\, M \rightleftharpoons N & \sqsupseteq & \mathsf{false} & M\,;M & \sqsupseteq & M \\
\mathsf{ref}\, M\,;N & \cong & N & L\,;(M\,;N) & \cong & (L\,;M)\,;N \\
\Omega & \sqsubseteq & M & L\,;M\,;N & \cong & M\,;L\,;N
\end{array}$$
$$\frac{\mathsf{let}\ \{x{=}\Omega\}\ \mathsf{in}\ M \cong \Omega}{M \sqsupseteq x\,;M}$$

Figure 3.5: Laws for Refs, Ω and Strictness.

3.4.5 Laws for Reduction Contexts

A *reduction context* $\mathbb{R}$ is a context in which the hole is the target of evaluation; in other words, evaluation cannot proceed until the hole is filled. We use the following simple grammar for reduction contexts; more complex definitions are also possible, but are not needed here.

$$\begin{array}{l}\mathbb{R} ::= [\cdot] \mid \mathsf{let}\ \{\vec{x}{=}\vec{M}\}\ \mathsf{in}\ \mathbb{R} \mid \mathbb{R}\,x \mid \mathsf{case}\ \mathbb{R}\ \mathsf{of}\ \{c_i\,\vec{x}_i{\to}M_i\} \\ \quad \mid\ \mathbb{R}\,;M \mid \mathbb{R} \rightleftharpoons M \mid \mathsf{deref}\ \mathbb{R}\end{array}$$

Figure 3.4 contains a collection of laws relating to reduction contexts.

3.4.6 Laws for Strictness

In Figure 3.5 we present a collection of laws for refs, and stricness. The term Ω denote any closed term which does not converge. For example, the "black-hole" term, $\mathsf{let}\ x{=}x\ \mathsf{in}\ x$, would suffice as a definition for Ω. Ω is the bottom element of the operational approximation ordering. The last of this collection of laws represents the transformation induced by *strictness analysis*. Operationally, a term M is strict in a free variable x if it is equivalent to Ω whenever x is bound to Ω. This corresponds to the usual denotational definition of strictness for the function $\lambda x.M$. The rule expresses that if M is strict in x then x can safely be evaluated in advance. This repressents the typical compiler optimisation that follows after performing strictness analysis. We sketch the proof of this property in the next section.

3.4.7 Proof Techniques for Equivalence

We have presented a collection of laws for approximation and equivalence – but how are they established? The definition of operational equivalence suffers from the standard problem: to prove that two terms are related requires one to examine their behaviour in *all* contexts. For this reason, it is common to seek to prove a *context lemma* [80] for an operational semantics: one tries to show that to prove M operationally approximates N, one only need compare their immediate behaviour. The following context lemma simplifies the proof of many laws:

Lemma 3.4.1 (Context Lemma) *For all terms M and N, $M \mathrel{\underset{\sim}{\sqsubset}} N$ if and only if for all $?$, S and substitutions σ,*

$$\langle ?,\ M\sigma,\ S\rangle{\Downarrow} \implies \langle ?,\ N\sigma,\ S\rangle{\Downarrow}$$

It says that we need only consider configuration contexts of the form $\langle ?,\ [\cdot],\ S\rangle$ where the hole $[\cdot]$ appears only once. The substitution σ from variables to variables is necessary here, but all the laws are closed under such substitutions, so there is no noticeable proof burden.

The proof of the context lemma follows the same lines as the corresponding proof for the *improvement theory* for call-by-need [81], and it involves uniform computation arguments which are similar to the proofs of related properties for call-by-value languages with state [74].

Here we give a flavour of such proofs by stating a few key properties, and outline the proof of the inference rule which says that if a term strict in a given variable, then the it is safe to evaluate the variable in advance.

It is useful and meaningful to allow computation over open configurations; this is a handy way to express certain properties of computations.

Proposition 3.4.2

1. *(Open Computation) If* $\langle ?,\ M,\ S\rangle \to^* \langle ?',\ N,\ S'\rangle$ *then* $\langle ?\,\Delta,\ M,\ ST\rangle \to^* \langle ?'\Delta,\ N,\ S'T\rangle$ *for any bindings* Δ *and stack* T *such that the corresponding configurations are well-formed.*

2. *(Subcomputation)* $\langle ?,\ M,\ S\rangle\Downarrow \Leftrightarrow \exists\Delta, V.\, \langle ?,\ M,\ \epsilon\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle\ \&\ \langle \Delta,\ V,\ S\rangle\Downarrow$

3. *(Value Stability)* *If* $\langle ?\,\{x{=}V\},\ M,\ S\rangle \to^* \langle \Delta,\ W,\ \epsilon\rangle$ *then* $\{x{=}V\} \subseteq \Delta$

4. *(Reordering) If* $\langle ?,\ M,\ \epsilon\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle$ *and* $\langle \Delta,\ N,\ \epsilon\rangle \to^* \langle \Delta',\ V',\ \epsilon\rangle$ *then there exists some* $?'$ *such that* $\langle ?,\ N,\ \epsilon\rangle \to^* \langle ?',\ V',\ \epsilon\rangle$ *and* $\langle ?',\ M,\ \epsilon\rangle \to^* \langle \Delta',\ V,\ \epsilon\rangle$.

The properties are established in a reasonably straightforward way by induction on the length of the computations. The last property of the list, reordering, can be established along the lines of Theorem 3.5.1 of [100]. Note that the reordering property would not hold if we had updatable references.

Let us illustrate the use of the context lemma and some of the properties above in a sketch proof of the strictness inference rule from figure 3.5.

Proposition 3.4.3 *If* $\mathsf{let}\ \{x{=}\Omega\}\ \mathsf{in}\ M \cong \Omega$ *then* $M \underset{\sim}{\sqsubset} x\,;M$

Proof. Under the assumption we will show that $M \underset{\sim}{\sqsubset} x\,;M$. By the context lemma, it is sufficient to show, for arbitrary $?$ and S ($x \in \mathrm{dom}?, S$), that if $\langle ?,\ M,\ S\rangle\Downarrow$ then $\langle ?,\ x\,;M,\ S\rangle\Downarrow$. Assume $\langle ?,\ M,\ S\rangle\Downarrow$. By (Subcomputation) and (Uniform Computation) we know that

$$\langle ?,\ M,\ \epsilon\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle \tag{3.1}$$

$$\langle ?,\ M,\ S\rangle \to^* \langle \Delta,\ V,\ S\rangle\Downarrow \tag{3.2}$$

Now, by the assumption that $\mathsf{let}\ \{x{=}\Omega\}\ \mathsf{in}\ M \cong \Omega$ we argue that there must be an intermediate state:

$$\langle ?,\ M,\ \epsilon\rangle \to^* \langle ?',\ x,\ T\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle$$

Applying the subcomputation argument again, we can see that $\langle ?', x, \epsilon \rangle \rightarrow^* \langle \Delta'\{x{=}W\}, W, \epsilon \rangle$ for some Δ'. By value stability we know that $\{x{=}W\} \subseteq \Delta$, so

$$\langle \Delta, x, \epsilon \rangle \rightarrow^2 \langle \Delta, W, \epsilon \rangle \tag{3.3}$$

Thus by (Reordering) (3.1) and (3.3), we know that

$$\langle ?, x, \epsilon \rangle \rightarrow^* \langle ?'', W, \epsilon \rangle \tag{3.4}$$
$$\langle ?'', M, \epsilon \rangle \rightarrow^* \langle \Delta, V, \epsilon \rangle \tag{3.5}$$

And thus, by open extension we can construct the following computation sequence:

$$\begin{aligned}
\langle ?, x; M, S \rangle &\rightarrow \langle ?, x, (;M) : S \rangle \\
&\rightarrow \langle ?'', W, (;M) : S \rangle && \text{(by 3.4)} \\
&\rightarrow \langle ?'', M, S \rangle \\
&\rightarrow \langle \Delta, V, S \rangle\Downarrow && \text{(by 3.5, 3.2)}
\end{aligned}$$

□

3.4.8 Relation to Other Calculi

Many authors have considered the semantics of functional languages extended with with various forms of state. The approach pioneered by Felleisen et al (e.g. [39]) has been to study term-based reduction-calculi. The advantage of this approach is that it builds on the idea of a core calculus of equivalences (generated by a confluent rewriting relation on terms) which is conservatively extended with each additional language feature. The price paid for this modularity is that the theory of equality is rather limited. The approach we have taken – studying operational equivalence – is exemplified by Mason and Talcott's work on call-by-value lambda calculi and state [74]. An advantage of the operational-equivalence approach is that it is a much richer theory, in which induction principles may be derived that are inexpressible in reduction calculi.

A reduction-calculi approach to call-by-need was introduced in [4]. An operational theory subsuming this calculus, the call-by-need *improvement theory*, was introduced by Moran and Sands [81]. In improvement theory, the operational equivalences allow the observation of the number of reduction steps to convergence. This makes sharing observable indirectly. The approach in this paper is based closely on the development of [81]. Interestingly all the laws which do not involve refs are also "cost equivalences within a constant factor" in the improvement theory. We have not been able to find a cost equivalence which is not an equivalence in Λ_{ref}, and it would be interesting and useful if it were possible to prove that equivalence in Λ_{ref} was an extension of cost equivalence.

There is also some work on the theory of side-effects to non-strict languages. Odersky [82] considered a minimal extension to the pure lambda calculus with binding constructs for local names, and with equality of names as their only primitive. (Pitts and Stark considered a similar extension for call-by-value [95] lambda calculus.) Because of the call-by-name operational model underlying this work, it is not directly relevant to the applications we have in mind, and the operational theory is somewhat simpler to develop. More relevant is the recent work of Ariola and Sabry [5], who consider the call-by-need lambda calculus extended with mutable state. Had we taken a reduction-calculus approach, rather than developing operational equivalence, we could have cut down their language and reduction theory. It would not, however, have been possible to treat e.g. strictness properties in such a reduction-theory. Their work could be very useful to prove the correctness of an implementation of our language.

We have only scratched the surface of the existing theory. Induction principles would be useful – and seem straightforward to adapt from [81]. For techniques more specific to the subtleties of references, work on parametricity properties of local names e.g., [94], might also be adaptable to the current setting.

3.5 Conclusions

We have motivated a small extension to Haskell which provides a practical solution to a common problem when manipulating data structures representing circuits. The feature is likely to be useful for other embedded description languages, and we briefly consider two such applications below.

The extension we propose is small, and turns out to be easy to add to existing Haskell compilers/interpreters in the form of an abstract data-type (a module with hidden data constructors). In fact similar functionality is already hidden away in the nonstandard libraries of many implementations.[2] A simple implementation using the Hugs-GHC library extensions is given in the appendix. The `hbc` compiler contains a module with essentially the same signature and functionality (plus a reference-update operation) in the `UnsafeDirty` library.

An important contribution of our work is to show that, in the absence of an update operation, these features are neither "unsafe" nor "dirty"! We have presented a precise operational semantics for this extension, and investigated laws of operational approximation. We have shown that the extended language has a rich equational theory, which means that the semantics is robust with respect to program transformations which respect sharing properties. For example, we have shown that standard compiler transformations which use strictness analysis to turn call-by-need in to call-by-value are still sound in this extension.

[2] www.haskell.org/implementations/

3.5.1 Other Applications of Observable Sharing

Here we mention two other potential applications of observable sharing to other embedded description languages.

Grammars and Parsers

A popular example of an embedded description language is a language for *grammars*. This is usually realised as a library with parsing combinators [58] for building more complex parsers from more basic ones. Parsing combinators are higher-order functions corresponding to grammatical constructs such as sequencing, alternation and repetition. Such parsers are simple to construct and easy to understand, since their form follows the grammar.

However, not all grammars are immediately executable when interpreted as parsers in this style. For example, the following grammar is *left recursive*:

$$T ::= T + T \mid n$$

Executing the grammar as a parser as it stands will result in an infinite loop, because a T will first parse a T, and so on. This is a known problem with parser combinators (and other top-down parsing methods), and usually one must transform a grammar so that it is not left recursive.

An alternative idea (due to Magnus Carlsson) is the following: if every parser had its own identity, and knew the identity of all its parents, it could detect the fact that it was used left recursively (if it occurs as one of its own parents). In that case, it could avoid infinite looping. We implemented a prototype parser based on this idea, using references to parsers.

Decision Trees

Another possible application area is decision trees. A decision tree is a binary tree with yes/no questions at its nodes and results at the leaves. We can obtain a result by walking down the tree and answering each question by yes or no, taking the left or right tree accordingly.

We can make such a decision tree more efficient by removing those nodes for which both subtrees are the same. In that case, the given answer does not matter. We can implement this using references, so that the comparison of the two subtrees only takes constant time.

```
type DT = Ref DecTree

data DecTree = Leaf Result
             | Node Quest DT DT
```

We introduce a special node creation function that checks if the two subtrees are identical.

```
node :: Quest -> DT -> DT -> DT
node quest yes no
  | yes <=> no = yes
  | otherwise  = ref (Node quest yes no)
```

One problem is that two subtrees might be equal, but not identical. This happens when we create two equal trees separately. We can leave it up to the programmer to make sure that if two trees are equal, they are shared. Another possibility is to use *memo functions* [27, 89] to make sure that equal subtrees are identical. Using this idea, we have made a simple functional implementation of Binary Decision Diagrams (BDD's) [20]. Exploring the semantic theory of extensions such as memo-tables or hashable references remains as a topic for future work.

Chapter 4

QuickCheck

QuickCheck *is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.*

This chapter was written together with John Hughes, and published at the International Conference on Functional Programming 2000, as an article titled 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs' [23].

4.1 Introduction

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

The cost of testing motivates efforts to automate it, wholly or partly. Automatic testing tools enable the programmer to complete testing in a shorter time, or to test more thoroughly in the available time, and they make it easy to repeat tests after each modification to a program. In this paper we describe a tool, QuickCheck, which we have developed for testing Haskell programs.

Functional programs are well suited to automatic testing. It is generally accepted that pure functions are much easier to test than side-effecting ones, because one need not be concerned with a state before and after execution. In an imperative language, even if whole programs are often pure functions from input to output, the procedures from which they are built are usually not. Thus relatively large units must be tested at a time. In a functional language, pure functions abound (in Haskell, only computations in the IO monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [49], which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reuseable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform distribution is often used instead, but for data drawn from infinite sets this is not even meaningful – how would one choose a random closed λ-term with a uniform distribution, for example? We have chosen to put distribution under the human tester's control, by defining a *test data generation language* (also embedded in Haskell), and a way to observe the distribution of test cases. By programming a suitable generator, the tester can not only control the distribution of test cases, but also ensure that they satisfy arbitrarily complex invariants.

An important design goal was that QuickCheck should be *lightweight*. Our implementation consists of a single pure Haskell'98 module of about 300 lines, which is in practice mainly used from the Hugs interpreter. We have also written a small script to invoke it, which needs to know very little about Haskell syntax, and consequently supports the full language and its extensions. It is not dependent on any particular Haskell system. A cost that comes with this decision is that we can only test properties that are expressible and observable *within* Haskell.

It is notoriously difficult to say how effective a testing method is in detecting faults. However, we have used QuickCheck in a variety of applications, ranging from small experiments to real systems, and we have found it to work well in practice. We report on some of this experience in this paper, and point out pitfalls to avoid.

The rest of this paper is structured as follows. Section 2 introduces the concept of writing properties and checking them using QuickCheck. Section 3 shows how to define test data generators for user-defined types. Section 4 briefly discusses the implementation. Section 5 presents a number of case studies that show the usefulness of the tool. Section 6 concludes.

4.2 Defining Properties

4.2.1 A Simple Example

As a first example, we take the standard function `reverse` which reverses a list. This satisfies a number of useful laws, such as

```
        reverse [x]  =  [x]
reverse (xs++ys)  =  reverse ys++reverse xs
reverse (reverse xs)  =  xs
```

In fact, the first two of these characterise `reverse` uniquely.

Note that these laws hold only for *finite, total* values. In this paper, unless specifically stated otherwise, we always quantify over completely defined finite values. This is to make it more likely that the properties are computable.

In order to check these laws using QuickCheck, we represent them as Haskell functions. Thus we define

```
prop_RevUnit x =
  reverse [x] == [x]

prop_RevApp xs ys =
  reverse (xs++ys) == reverse ys++reverse xs
```

```
prop_RevRev xs =
  reverse (reverse xs) == xs
```

Now, if these functions return `True` for every possible argument, then the properties hold. We load them into the Hugs interactive Haskell interpreter [64], and call for example

```
Main> quickCheck prop_RevApp
OK: passed 100 tests.
```

The function `quickCheck` takes a law as a parameter and applies it to a large number of randomly generated arguments — in fact 100[1] — reporting "OK" if the result is `True` in every case.

If the law fails, then `quickCheck` reports the counter-example. For example, if we mistakenly define

```
prop_RevApp xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

then checking the law might produce

```
Main> quickCheck prop_RevApp
Falsifiable, after 1 tests:
[2]
[-2,1]
```

where the counter model can be extracted by taking `[2]` for `xs`, and `[-2,1]` for `ys`.

In fact the programmer must provide a little more information: the function `quickCheck` is actually overloaded, in order to be able handle laws with a varying number of variables, and the overloading cannot be resolved if the law itself has a polymorphic type, as in these examples. Thus the programmer must specify a fixed type at which the law is to be tested. So we simply give a type signature for each law, for example

```
prop_RevApp :: [Int] -> [Int] -> Bool
```

Of course, the property `prop_RevApp` holds polymorphically, but we must specify which monomorphic instance to test it at, so that we can generate test cases. This turns out to be quite important in the case of overloaded functions. For example, + is associative for the type `Int`, but not for `Double`! In some cases, we can use parametricity [121] to argue that a property holds polymorphically.

[1] 100 is a rather arbitrary number, so our library provides a way to specify this as a parameter.

4.2.2 Functions

We are also able to formulate properties that quantify over functions. To check for example that function composition is associative, we first define extensional equality `(===)` by `(f === g) x = f x == g x`, and then write:

```
prop_CompAssoc :: (Int -> Int) -> (Int -> Int)
               -> (Int -> Int) -> Int -> Bool
prop_CompAssoc f g h =
  f . (g . h) === (f . g) . h
```

The only difficulty that function types cause is that, if a counter-example is found (for example if we try to check that function composition is commutative), then the function values are printed just as "`<<function>>`". In this case we discover that the 'law' we are checking is false, but not why.

4.2.3 Conditional Laws

Laws which are simple equations are conveniently represented by boolean function as we have seen, but in general many laws hold only under certain conditions. QuickCheck provides an implication combinator to represent such conditional laws. For example, the law

$$\texttt{x <= y} \Longrightarrow \texttt{max x y == y}$$

can be represented by the definition

```
prop_MaxLe :: Int -> Int -> Property
prop_MaxLe x y = x <= y ==> max x y == y
```

Likewise, the insertion function into ordered lists satisfies the law

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==> ordered (insert x xs)
```

Note that the result type of the property is changed from `Bool` to `Property`. This is because the testing semantics is different for conditional laws. Instead of checking the property for 100 random test cases, we try checking it for 100 test cases *satisfying the condition*. If a candidate test case does not satify the condition, it is discarded, and a new test case is tried.

Checking the laws `prop_MaxLe` and `prop_Insert` succeed as usual, but sometimes, checking a conditional law produces the output

```
Arguments exhausted after 64 tests.
```

If the precondition of a law is seldom satisfied, then we might generate many test cases without finding any where it holds. In such cases it is hopeless to search for 100 cases in which the precondition holds. Rather than allow test case generation to run forever, we generate only a limited number of candidate test cases (the default is 1000). If we do not find 100 valid test cases among those candidates, then we simply report the number of successful tests we were able to perform.

In the example, we know that the law passed the test in 64 cases. It is then up to the programmer to decide whether this is enough, or whether it should be tested more thoroughly.

4.2.4 Monitoring Test Data

Perhaps it seems that we have tested the law for insert thoroughly enough to establish its credibility. However, we must be careful. Let us modify `prop_Insert` as follows[2]

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    classify (null xs) "trivial" $
      ordered (insert x xs)
```

Checking the law now produces the message

```
OK, passed 100 tests (43% trivial).
```

The `classify` combinator does not change the meaning of a law, but it classifies some of the test cases, in this case those where `xs` is the empty list were classified as "trivial". Thus we see that a large proportion of the test cases only tested insertion into an empty list.

We can get more information than just labelling some test cases. The combinator `collect` will gather all values that are passed to it, and print out a histogram of these values. For example, if we write:

```
prop_Insert :: Int -> [Int] -> Property
prop_Insert x xs =
  ordered xs ==>
    collect (length xs) $
      ordered (insert x xs)
```

we might get as a result:

[2] `$` is Haskell's infix function application.

```
OK, passed 100 tests.
49% 0.
32% 1.
12% 2.
4% 3.
2% 4.
1% 5.
```

So we see that only 19 cases tested insertion into a list with more than one element. While this is enough to provide fairly strong evidence that the law holds, it is worrying that very short lists dominate the test cases so strongly. After all, it is easy to define an erroneous version of `insert` which nevertheless works for lists with at most one element.

The reason for this behaviour, of course, is that the precondition `ordered xs` skews the distribution of test cases towards short lists. Every generated list of length 0 or 1 is ordered, but only 50% of the lists of length 2 are. Thus test cases with longer lists are more likely to be rejected by the precondition. There is a risk of this kind of problem every time we use conditional laws, so it is always important to investigate the proportion of trivial cases among those actually tested.

The best solution, though, is to replace the condition with a custom test data generator for ordered lists. We write

```
prop_Insert :: Int -> Property
prop_Insert x =
  forAll orderedList $ \xs ->
    ordered (insert x xs)
```

which specifies that values for `xs` should be generated by the test data generator `orderedList`. Checking the law now gives "`OK: passed 100 tests`", as we would expect.

QuickCheck provides support for the programmer to define his or her own test data generators, with control over the distribution of test data, which we will look at in the next section.

4.2.5 Infinite Structures

The Haskell function `cycle` takes a non-empty list, and returns a list that repeats the contents of that list infinitely. Now, take a look at the following law, formulated in QuickCheck as[3]:

```
prop_DoubleCycle :: [Int] -> Property
```

[3]Note that leaving the condition `not (null xs)` out results in an error, because `cycle` is not defined for empty lists.

```
prop_DoubleCycle xs =
  not (null xs) ==>
    cycle xs == cycle (xs ++ xs)
```

Although intuitively the law is true, it cannot be checked since we are comparing two infinite lists using computable equality `==`, which does not terminate. Instead, we can reformulate the property as a logically equivalent one, by using the fact that two infinite lists are equal if all finite initial segments are equal.

```
prop_DoubleCycle :: [Int] -> Int -> Property
prop_DoubleCycle xs n =
  not (null xs) && n >= 0 ==>
    take n (cycle xs) == take n (cycle (xs ++ xs))
```

Another issue related to infinite structures is quantification over them. We will later see how to deal with properties that for example hold for all infinite lists, but in general it is not clear how to formulate and execute properties about structures containing bottom.

4.3 Defining Generators

4.3.1 Arbitrary

The way we generate random test data of course depends on the type. Therefore, we have introduced a type class `Arbitrary`, of which a type is an instance if we can generate arbitrary elements in it.

```
class Arbitrary a where
  arbitrary :: Gen a
```

`Gen a` is an abstract type representing a generator for type `a`. The programmer can either use the generators built in to QuickCheck as instances of this class, or supply a custom generator using the `forAll` combinator, which we saw in the previous section. For now, we define the type `Gen` as

```
newtype Gen a = Gen (Rand -> a)
```

Here `Rand` is a random number seed; a generator is just a function which can manufacture an `a` in a pseudo random way. But we will treat `Gen` as an *abstract* type, so we define a primitive generator

```
choose :: (Int, Int) -> Gen Int
```

to choose a random number in an interval, and we program other generators in terms of it.

We also need combinators to build complex generators from simpler ones; to do so, we declare `Gen` to be an instance of Haskell's class `Monad`. This involves implementing the methods of the `Monad` class

```
return :: a -> Gen a
(>>=)  :: Gen a -> (a -> Gen b) -> Gen b
```

the first one of which contructs a constant generator, and the second one being the monadic sequencing operator which generates an `a`, and passes it to its second argument to generate a `b`. The definition of `(>>=)` needs to pass *independent* random number seeds to its two arguments, and is only passed one seed, but luckily the Haskell random number library provides an operation to split one seed into two.

Defining generators for many types is now straightforward. As examples, we give generators for integers and pairs:

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)

instance (Arbitrary a, Arbitrary b) =>
                        Arbitrary (a,b) where
  arbitrary = liftM2 (,) arbitrary arbitrary
```

In the second case we use a standard monadic function, `liftM2`, which is defined in terms of `return` and `(>>=)`, to make a generator that applies the pairing operator `(,)` to the results of two other generators. QuickCheck contains such declarations for most of Haskell's predefined types.

4.3.2 Generators for User-Defined Types

Since we define test data generation via an instance of class `Arbitrary` for each type, then we must rely on the user to provide instances for user-defined types. In principle we could try to generate these automatically, in a pre-processor or via polytypic programming [8], but we have chosen instead to leave this task to the user. This is partly because we want QuickCheck to be a lightweight tool, easy to implement and easy to use in a standard programming environment; we don't want to oblige users to run their programs through a pre-processor between editing them and testing them. But another strong reason is that it seems to be very hard to construct a generator for a type, without knowing something about the desired distribution of test cases.

Instead of producing generators automatically, we provide combinators to enable a programmer to define his own generators easily. The simplest, called `oneof`, just makes a choice among a list of alternative generators with a uniform distribution. for example, if the type `Colour` is defined by

```
data Colour = Red | Blue | Green
```

then a suitable generator can be defined by

```
instance Arbitrary Colour where
  arbitrary = oneof
    [return Red, return Blue, return Green]
```

As another example, we could generate arbitrary lists using

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = oneof
    [return [], liftM2 (:) arbitrary arbitrary]
```

where we use `liftM2` to apply the cons operator `(:)` to an arbitrary head and tail. However, this definition is not really satisfactory, since it produces lists with an average length of one element. We can adjust the average length of list produced by using `frequency` instead, which allows us to specify the frequency with which each alternative is chosen. We define

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
    [ (1, return [])
    , (4, liftM2 (:) arbitrary arbitrary) ]
```

to choose the cons case four times as often as the nil case, leading to an average list length of four elements.

However, for more general data types, it turns out that we need even finer control over the distribution of generated values. Suppose we define a type of binary trees, and a generator:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency
    [ (1, liftM  Leaf   arbitrary)
    , (2, liftM2 Branch arbitrary arbitrary) ]
```

We want to avoid choosing a `Leaf` too often, hence the use of `frequency`.

However, *this definition only has a 50% chance of terminating!* The reason is that for the generation of a `Branch` to terminate, *two* recursive generations must terminate. If the first few recursions choose `Branch`es, then generation terminates only if very many recursive generations all terminate, and the chance of this is small. Even when generation terminates, the generated test data is sometimes very large. We want to avoid this: since we perform a large number of tests, we want each test to be small and quick.

Our solution is to limit the *size* of generated test data. But the notion of a size is hard even to define in general for an arbitrary recursive datatype (which

may include function types anywhere). We therefore give the responsibility for limiting sizes to the programmer defining the test data generator. We change the representation of generators to

```
newtype Gen a = Gen (Int -> Rand -> a)
```

where the new parameter is to be interpreted as some kind of size bound. We define a new combinator

```
sized :: (Int -> Gen a) -> Gen a
```

which the programmer can use to access the size bound: `sized` generates an `a` by passing the current size bound to its parameter. It is then up to the programmer to interpret the size bound in some reasonable way during test data generation. For example, we might generate binary trees using

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree

arbTree 0 = liftM Leaf arbitrary
arbTree n = frequency
  [ (1, liftM  Leaf   arbitrary)
  , (4, liftM2 Branch (arbTree (n `div` 2))
                      (arbtree (n `div` 2))) ]
```

With this definition, the size bound limits the number of nodes in the generated trees, which is quite reasonable.

Now that we have introduced the notion of a size bound, we can use it sensibly in the generators for other types such as integers and lists (so that the absolute value respective length is bounded by the size). So the definitions we presented earlier need to be modified accordingly.

We stress that the size bound is simply an extra, global parameter which every test data generator may access; every use of `sized` sees the same bound[4]. We do *not* attempt to 'divide the size bound among the generators', so that for example a longer generated list would have smaller elements, keeping the overall size of the structure the same. The reason is that we wish to avoid correlations between the sizes of different parts of the test data, which might distort the test results.

We do vary the size between different test cases: we begin testing each property on small test cases, and then gradually increase the size bound as testing progresses. This makes for a greater variety of test cases, which both makes testing more effective, and improves our chances of finding enough test cases satisfying the precondition of conditional properties. It also makes it more likely that we will find a small counter example to a property, if there is one.

[4] Unless the programmer explicitly changes it using the `resize` combinator.

4.3.3 Generating Functions

If we are to check properties involving function valued variables, then we must be able to generate arbitrary functions. Rather surprisingly, we are able to do so. To understand how, notice that a function generator of type `Gen (a->b)` is represented by a function of type `Int -> Rand -> a -> b`. By reordering parameters, this is equivalent to the type `a -> Int -> Rand -> b`, which represents `a -> Gen b`. We can thus define

```
promote :: (a -> Gen b) -> Gen (a->b)
```

and use it to produce a generator for a function type, provided we can construct a generator for the result type which somehow depends on the argument value. We take care of this dependence by defining a new class,

```
class Coarbitrary a where
  coarbitrary :: a -> Gen b -> Gen b
```

whose method `coarbitrary` modifies a generator in a way depending on its first parameter. We will think of `coarbitrary` as producing a *generator transformer* from its first argument. Given this class, we can define

```
instance (Coarbitrary a, Arbitrary b) =>
                         Arbitrary (a->b) where
  arbitrary =
    promote (\a -> coarbitrary a arbitrary)
```

which generates an arbitary function that uses its argument to modify the generation of its result.

In order to define instances of `Coarbitrary` we need a way to construct generator transformers. We therefore define the function

```
variant :: Int -> Gen a -> Gen a
```

where `variant n g` constructs a generator which transforms the random number seed it is passed in a way depending on `n`, before passing it to `g`. This function must be defined very carefully, so that all the generators we construct using it are independent. Given any list of integers `[n1,n2,...nk]`, we can construct a generator transformer

```
variant n1 . variant n2 . ... . variant nk
```

We define `variant` so that different lists of integers give rise to independent generator transformers (with a very high probability).

Now we can define instances of `coarbitrary` that choose between generator transformers depending on the argument value. For example, the boolean instance

```
instance Coarbitrary Bool where
  coarbitrary b =
    if b then variant 0 else variant 1
```

transforms a generator in independent ways for `True` and for `False`; the generators `coarbitrary True g` and `coarbitrary False g` will be independent. In a similar way, we can define suitable instances for many other types. For example, the integer instance just converts its integer argument into a sequence of bits, which are then used as generator transformers in turn.

Instances of `Coarbitrary` for recursive datatypes can be defined according to a standard pattern. For example, the list instance is just

```
instance Coarbitrary a => Coarbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (x:xs) =
    variant 1 . coarbitrary x . coarbitrary xs
```

The goal is that different lists should be mapped to independent generator transformers; we achieve this by mapping each constructor to an independent transformer, and composing these with transformers derived from each component. Other recursive datatypes can be treated in the same way. Since the programmer is responsible for making these definitions for user-defined types, it is important that they be straightforward.

Finally, we can even interpret functions as generator transformers, with an instance of the form

```
instance (Arbitrary a, Coarbitrary b) =>
                       Coarbitrary (a->b) where
  coarbitrary f gen =
    arbitrary >>= \a -> coarbitrary (f a) gen
```

The idea is that we apply the given function to an arbitrary argument, and use the result to transform the given generator. In this way, two functions which are different will give rise to different generator transformers.

Note that, if we had tried to avoid needing to split random number seeds by defining the `Gen` monad as a state transformer on the random seed, rather than a state reader, then we would not have been able to define the `promote` function, and we would not have been able to generate random functions.

4.4 Implementing QuickCheck

As we have seen, the function `quickCheck` can handle properties with a varying number of arguments and different result types. To implement this, we introduce the type `Property`, and we create the type class `Testable`.

```
class Testable a where
  property :: a -> Property
```

The `Property` type represents predicates that can be checked by testing. This means that it needs to be able to generate random input, and finally product a test result. So, a `Property` is a computation in the `Gen` monad, ending in an abstract type `Result`, which keeps track of the boolean result of the testing, the classifications of test data, and the arguments used in the test case.

```
newtype Property = Prop (Gen Result)
```

Let us take a look at some instances of `Testable`. An easy type to check is of course `Bool`. Further, functions for which we can generate arbitrary arguments can be tested. And lastly, even the property type itself is an instance, so that we can nest property combinators.

```
instance Testable Bool where
  property b = Prop (return (resultBool b))

instance (Arbitrary a, Show a, Testable b) =>
                          Testable (a->b) where
  property f = forAll arbitrary f

instance Testable Property where
  property p = p
```

Using the function `property`, it becomes easy to define the function `quickCheck`. Its type is:

```
quickCheck :: Testable a => a -> IO ()
```

More details of the implementation can be found in the appendix.

4.5 Some Case Studies

4.5.1 Unification

As a first (and rather pathological) case study, we discuss a unfication algorithm which we have developed along with a QuickCheck specification. This was quite revealing, both as regards the impact that QuickCheck has on programming, and the pitfalls that must be avoided. It is too large to present in detail, so we will just discuss the lessons we learned.

Impact on Type Definitions

First of all, the use of QuickCheck had an impact on the design of the types in the program. We defined the type of terms to be unified as

```
data Term    = Var Var | Constr Name [Term]
newtype Var  = Variable Nat
newtype Name = Name String
```

rather than the equivalent

```
data Term = Var Int | Constr String [Term]
```

which we would probably have chosen otherwise. The type we used distinguishes between a string used as a constructor name, and a string used in other contexts, and between a natural number used as a variable name, and an integer used in other contexts.

The reason we chose to make these distinctions in the type is that it enabled us to define a different test data generator for `Name`s for example, than for strings. Had we generated terms using the default test data generator for strings, then it is very unlikely that we would ever generate unifiable terms, since it is unlikely we would generate the same constructor name twice. Instead, we chose to generate constructor names using

```
instance Arbitrary Name where
  arbitrary = sized $ \n -> oneof
    [ return (Name ("v" ++ show i))
    | i <- [1..log n+1] ]
```

which gives us a good chance that generated terms will be at least partially unifiable. Likewise, we limited unification variables in test data to a small set.

Of course, we could have used the second `Term` type above and specified a custom test data generator with an explicit `forAll` in each property. But it is more convenient to let test data be automatically generated using `arbitrary`, so one is encouraged to make distinctions explicit in types. There are other advantages to doing so also: it permits the type checker to detect more errors. So, using QuickCheck changes the balance of convenience in the question of introducing new types in programs.

Checking Functional Properties

A unifier needs to manage the current substitution, and also the possibility of failures in recursive calls. A convenient way to do so is to use a monad. We defined a unification monad `M`, represented by a function, with operations

```
setV :: Var -> Term -> M ()
getV :: Var -> M Term
```

to read and write variables, among others. We were able to define an 'extensional equality' operator `eqM` on monadic values, and check both the monad laws and properties such as

```
prop_SetGet v t = (do setV v t; getV v)
            `eqM` (do setV v t; return t)
```

Errors Found

It would be nice to be able to report that QuickCheck found a large number of errors in this example. In fact, no errors at all were found in the unifier itself. This is probably more a reflection on the number of times the authors have programmed unifiers previously, than on the effectiveness of QuickCheck — we know how to do it, quite simply.

On the other hand, we did find errors in the *specification*. For example, we defined a substitution function which repeatedly substitutes until no variables in the domain of the substitution remain, and stated the obvious property

```
prop_SubstIdempotent s t =
  subst s (subst s t) == subst s t
```

QuickCheck revealed this property to be false: it holds only for acyclic substitutions (otherwise an infinite term is generated, and the equality test loops). This error was found using the function `verboseCheck`, which prints out the arguments to every test case before it checks it.

We were obliged to correct the specification to

```
prop_SubstIdempotent s t =
  acyclic s ==> subst s (subst s t) == subst s t
```

Thus QuickCheck made us think harder about the properties of our code, and document them correctly.

On the downside, formulating the specification correctly turned out to be quite a lot of work, perhaps more than writing the implementation. This was partly because predicates such as `acyclic` are non-trivial to define; a good set theory library would have helped here.

A False Sense of Security

The most serious pitfall we uncovered with this experiment was the false sense of security that can be engendered when one's program passes a large number of tests in trivial cases. We have already referred to this problem when we discussed conditional properties; in this example, it bit us hard.

Many properties of unification apply to the case when unification succeeds. They can be stated conveniently in the form

```
prop_Unify t1 t2 = s /= Nothing ==> ...
   where s = unifier t1 t2
```

since our unifier returns `Nothing` when it fails. With a little reflection, we see that two randomly chosen terms are fairly likely to be unifiable, since variables occur quite often, and if either term is a variable then unification will almost certainly succeed. On the other hand, if neither term is a variable then the probability that they will unify is small. Thus the case where one term is a variable is heavily over-represented among the test cases that satisfy the precondition — we found that over 95% of test cases had this property. Although QuickCheck succeeded in 'verifying' every property, we can hardly consider that they were thoroughly tested.

The solution was to use a custom test data generator

```
prop_Unify =
  forAll probablyUnifiablePair $ \(t1,t2) ->
    s /= Nothing ==> ...
 where s = unifier t1 t2
```

We generated 'probably unifiable pairs' by generating *one* random term, and replacing random subterms by variables in two different ways. This usually generates unifiable terms, although may fail to when variables are used inconsistently in the two terms. With this modification, the proportion of trivial cases fell to a reasonable 20–25%.

This experience underlines the importance of investigating the distribution of actual test cases, whenever conditional properties are used.

4.5.2 Circuit Properties

Lava in a Nutshell

Lava [17, 24] is a tool to describe, simulate and formally verify hardware. Lava is a so-called *embedded language*, which means that the circuit descriptions and properties are all expressed in an existing programming language, in this case Haskell.

The idea is to view a hardware circuit as a function from signals of inputs to signals of outputs. The Lava system provides primitive circuits, such as `and2`, `xor2`, and `delay`. More complicated circuits are defined by combining these. Circuits defined in Lava can be *simulated* as follows: one provides inputs and the outputs are calculated.

```
Main> simulate and2 (high, low)
high
```

Furthermore, the Lava system defines combinators for circuits, such as sequential composition (`>->`), parallel composition (`<|>`), and `column`, which takes one circuit and replicates it in a column of circuits, connecting the vertical wires.

Properties in Lava

Properties of circuits can be defined in a similar way. For example, to define the property that a certain circuit is commutative, we say:

```
prop_Commutative circ (a, b) =
  circ (a, b) <==> circ (b, a)
```

where `<==>` is logical equivalence lifted to arbitrary types containing signals, in this case a pair.

Properties can be formally verified. We do this by providing *symbolic* inputs to the circuit or property, and calculating a concrete expression in a Haskell datatype, representing the circuit.

We can then write this expression to a file and call an external theorem prover. All this is done by the overloaded Lava function `verify`. Here is how we can use it to verify that a so-called half adder component is commutative:

```
Main> verify (prop_Commutative halfAdd)
Proving: ... Valid.
```

The Lava system provides a number of functions and combinators to conveniently express properties and formally verify them.

QuickCheck in Lava

Though we are able to verify properties about circuits in Lava, we greatly benefit from extending it with a testing tool like QuickCheck. There are two main reasons for that.

The first one is that calling an external theorem prover is a very heavyweight process. When verifying, say, a 32-bit multiplier, the formulas that we generate for external theorem proving are quite big and we often have to wait for a long time to get an answer.

So, a typical development cycle is to write down the specification of the circuit first, then make an implementation, QuickCheck it for obvious bugs, and lastly, call the external theorem prover for verifying the correctness.

Here is an example of how to use QuickCheck in Lava:

```
Main> quickCheck (prop_Commutative halfAdd)
OK: passed 100 tests.
```

Adding this testing methodology to Lava turned out to be quite straightforward, because Lava already had a notion of properties. Testing can be done for all circuit types, even sequential circuits (containing latches). We simply check the circuit property for a sequence of inputs.

Higher Order Testing

The second reason for using testing in Lava is simply that we can test *more* properties than we can formally verify! The external theorem provers that are connected to Lava can only deal with at most first-order logics, and the Lava system is only able to generate formulas of that type.

Sometimes, we would like to verify properties about *combinators*. For example, proving that `column` distributes over `(>->)`:

```
prop_ComposeSeqColumn circ1 circ2 inp =
      column (circ1 >-> circ2) inp
 <==> (column circ1 >-> column circ2) inp
```

is very hard to verify in Lava *for all* `circ1` *and* `circ2`. In fact, such properties are hard to verify automatically in general (we can do it for small fixed sizes however). But since we can randomly generate functions, we can at least *test* these kind of properties for arbitrary circuits.

A drawback is that we have to fix the types of these circuits, whereas the combinators themselves, and thus the properties about them, are polymorphic in the circuits' input and output types.

Errors Found

The authors used the QuickCheck library while developing a collection of arithmetic circuits. Previously, testing was already used in the development process, but only in a very limited and ad-hoc way. Now, much more thorough testing was possible.

The errors we found in these particular circuits were of two kinds. Firstly, we found errors that our formal verification method would have found as well: logical errors in the circuits. But secondly, we also found errors due to the fact that random input also means random input *size*. For example, for an n-bit $\times$ m-bit adder, we only use and formally verify the circuit for specific input sizes. Random testing checks many more combinations, and it often turned out that we had forgotten to define one of these cases!

4.5.3 Propositional Theorem Proving

For teaching purposes, we implemented two different well-known methods of checking if a set of propositional logic clauses is contradictory. One of these

methods was the Davis-Putnam method [32], which uses backtracking to generate a list of all models. The other one was Stålmarck's method [115], which is an incomplete method and uses a variant on the dilemma proof system to gather information about the literals in the clause set.

```
type Clause = [Lit]  -- disjunction
type Model  = [Lit]  -- conjunction

davisPutnam :: [Clause] -> [Model]
stålmarck   :: Int -> [Clause] -> Maybe Model
```

The `stålmarck` function takes an extra argument, an `Int`, which is the so-called "saturation level", a parameter which limits the depth of the proofs, and usually lies between 0 and 3. If the result of `stålmarck` is `Nothing`, it means that there was a contradiction. If the result is `Just m`, it means that every model of the clause set should have `m` as a sub-model.

Since `davisPutnam` is much more straight-forward to implement than `stålmarck`, we wanted to check the latter against the first. Here is how we formulate the informal property stated above:

```
prop_Stålmarck_vs_DP :: Property
prop_Stålmarck_vs_DP =
  forAll clauses $ \cs ->
    forAll (choose (0,3)) $ \sat ->
      case stålmarck sat cs of
        Nothing ->
          collect "contradiction" $
            davisPutnam cs == []

        Just m ->
          not (null m) ==>
            collect (length m) $
              all (m `subModel`) (davisPutnam cs)
```

Note that we collect some statistics information: "contradiction" when the result was `Nothing`, and the size of `m` in the case of `Just m`. We also expressed that we disqualify a test case when `stålmarck` returns `Just []`.

With the help of this property, QuickCheck found 3 bugs! These bugs were due to implicit unjustified assumptions we had about the input. The implementations of both algorithms assumed that no clauses in the input could contain the same literal twice, and the `stålmarck` function assumed that none of the input clauses was empty.

The data generator `clauses` is defined using the same techniques as in section 4.5.1. Testing the property took about 30 seconds, and from the output we could see that the distribution of `Nothing` vs. `Just m` was about 50/50.

4.5.4 Pretty Printing

Andy Gill reported an interesting story about using QuickCheck to us. He used it in developing a variant of Wadler's pretty printing combinator library [122] in Java. First, he implemented his variant functionally, using Haskell. Then, still using Haskell, he used a state monad with exceptions to develop an imperative implementation of the same library. The idea was that the second implementation models what goes on in a Java implementation.

Then, he expressed the relationship between the two different implementations using QuickCheck properties. He writes: *"This quickly points out where my reasoning is faulty, and provides great tests to catch the corners of the implementation. Three problems were found, the third of which showed that I had merged two concepts in my implementation that I should not have."*

Furthermore, he made an improvement in the way QuickCheck reports counter examples. Sometimes, the counter examples found are very large, and it is difficult to go back to the property and understand why it is a counter example. However, when the counter example is an element of a tree-shaped datatype, the problem can often be located in one of the sub-trees of the counter example found. Gill extended the `Arbitrary` class with a new method

```
smaller :: a -> [a]
```

which is intended to return a list of smaller, but similar values to its argument – for example, direct subtrees. He adapted the `quickCheck` function so that when a counter example is found, it tries to find a smaller one using this function. In some cases much smaller counterexamples were found, greatly reducing the time to understand the bug found.

The last step Gill made in developing his Java pretty printing library was porting the state and exception monad model in Haskell to Java. He then used QuickCheck to generate a large number of test inputs for the Java code, in order to check that the Java implementation was equivalent to the two Haskell models.

4.5.5 Edison

Chris Okasaki's *Edison* is a library of efficient data structures suitable for implementation and use in functional programming languages. He has used QuickCheck to state and test properties of the library. Every data structure in the library has been made an instance of `Arbitrary`, and he has included several extra modules especially for formulating properties about these data structures. He reports: *"My experience has mostly been that of a very satisfied user.* QuickCheck *lets me test Edison with probably 25% (maybe less!) of the effort of my previous test suite, and does a much better job to boot."*

Okasaki also mentions a drawback, having to do with the Haskell module system. He often uses one specification of a data structure together with different

implementations. A natural way to do this is to place the specification in one module, and each implementation in a seperate module. But since the specification refers to the implementation, then the specification module must import the implementation one currently under test. Okasaki was obliged to edit the specification module by hand before each test, so as to import the right implementation! Much preferable would be to parameterise the specification on an implementation module; ML-style functors would be really helpful here!

4.6 Discussion

4.6.1 On Random Testing

At first sight, random selection of test cases may seem a very naive approach. Systematic methods are often preferred: in general, a *test adequacy criterion* is defined, and testing proceeds by generating test cases which meet the adequacy criterion. For example, a simple criterion is that every reachable statement should be executed in at least one test, a more complex one that every feasible control-flow path (with exceptions for loops) be followed in at least one test. A wide variety of adequacy critera have been proposed; a recent survey is [125].

We have chosen not to base QuickCheck on such an adequacy criterion. In part, this is because many criteria would need reinterpretation before they could be applied to Haskell programs – it is much less clear, for example, what a control-flow path is in a language with higher-order functions and lazy evaluation. In part, such a criterion would force us to use much more heavyweight methods – even measuring path coverage, for example, would require compiler modifications and thus tie QuickCheck to a particular implementation of Haskell (namely the one we modified to collect path information). Generating test data to exercise a particular path requires constraint solving: one must find input values which make the series of tests along the given path produce specified results. While such constraint solving may be feasible for arithmetic data, for the rich symbolic datatypes found in Haskell programs it is a difficult research problem in its own right.

However, apart from the difficulty of automating systematic testing methods for Haskell, there is no clear reason to believe that doing so would yield better results. In 1984, Duran and Ntafos compared the fault detection probability of random testing with partition testing, and discovered that the differences in effectiveness were small [34]. Hamlet and Taylor repeated their study more extensively, and corroborated the original results [50]. Although partition testing is slightly more effective at exposing faults, to quote Hamlet's excellent survey [49], *"By taking 20% more points in a random test, any advantage a partition test might have had is wiped out."*

For small programs in particular, it is likely that random test cases *will* indeed exercise all paths, for example, so that test coverage is likely to be good by

any measure. Using QuickCheck, we apply random testing at a fine grain: we check properties of individual functions, but the functions they call are tested independently. So even when QuickCheck is used to test a large program, we always test a small part at a time. Therefore we may expect random testing to work particularly well.

Given this, together with the much greater difficulty of automating systematic testing for Haskell, our choice of random testing is clear.

4.6.2 Correctness Criteria

The problem of determining whether a test is passed or not is known as the *oracle problem*. One solution is to compare program output with that of another version of the program, perhaps an older one, or perhaps a simpler, slower, but 'obviously correct' version. Alternatively, an executable specification might play the same rôle. This kind of oracle can easily be expressed as a QuickCheck property, although our properties are much more general.

However, often one can check that a program's output is correct much more efficiently than one can compute the output. Blum and Kannan exploit this in their work on *result-checking* [18]: a program checker is defined to be another program which classifies the program's output as correct or buggy, with a high probability of classifying correctly, and does so with strictly lower complexity. They distinguish program *checking* from program *testing*: their proposal is that programs should always check their output, and indeed in further work Blum et al. showed how programs which usually produce correct answers can even *correct* wrong output [19] (in particular domains). Of course, result checkers can also be expressed as QuickCheck properties, although we use them for testing rather than as a part of the final program.

QuickCheck's property language is however more general than result checking. Via conditional properties or specific test data generators, we can express properties which hold only for a subset of all possible inputs. Thus we avoid testing functions in cases which lead to run-time errors, or cases in which we do not care about the result. For example, we do not test insertion into an unordered list — there is no point in doing so. Yet a result checker must verify that a program produces the 'correct' output in *all* cases, even those which are uninteresting. Moreover, QuickCheck properties are not limited to checking the result of an individual function call — the property that an operator is associative, for example, cannot really be said to check the result of any individual use of the operator, but still expresses a useful 'global' property that can be checked by testing.

The idea of testing the properties in a specification directly was used in the DAISTS system [42] for testing abstract data types, which compiled equational properties into testing code, although test cases had to be supplied by the user. Lacking automatic test case generation, DAISTS did not need equivalents of

our conditional and quantified properties. Although the language used was imperative, abstract data type operations had to be forbidden to side-effect their arguments, thus the programs to be tested were essentially restricted to be functional. Later work aims to relax this restriction: Antoy and Hamlet describe a technique for testing C++ classes against an algebraic specification, which is animated in order to predict the correct result [3]. However, the specification language must be restricted in order to guarantee that specifications *can* be animated.

There seems to be no published work on automatic testing of *functional* programs against specifications. We simply observe that functional programs and property based specifications are a very good match: we can use the given properties directly for testing. Moreover, embedding the specification language in Haskell permits us to write very powerful and flexible properties, with a minimum of learning effort required.

4.6.3 Test Data Generation

Commercial random testing tools generate test data in limited domains, with the goal of matching the distribution of actual data for the system under test – the so-called *operational profile*. In this case, statistical inferences about the mean time between system failures can be drawn from the test results.

In order to generate more complex data, it is necessary to provide a description of the data's structure. A popular approach to doing so uses *grammars*. However, it was realised very early that context-free grammars cannot express all the desired properties of test data – for example, that a generated random program contains no undeclared identifiers. Therefore the grammars were enhanced with actions [21], or extended to attribute grammars. This approach has been most used for testing compilers, although Maurer argues for its use in many contexts [76].

Grammars have been used for systematic testing, where for example the generated test data is required to exercise each production at least once. Such an adequacy criterion maybe be particularly appropriate for compiler testing. Maurer also used grammars for random testing [76], and noted the termination problem for recursive grammars. His solution, though, was just to increase the probabilities of generating leaves so that eventual termination is guaranteed. Our experience is that this results in far too high a proportion of trivial test cases, and therefore inefficient testing – more tests must be run to exercise the program properly. We believe our method of controlling sizes is much superior.

It seems that the need to learn a complex language of extended grammars has hindered the adoption of these methods in practice. By embedding a test generator language in Haskell, we provide (at least) the same capabilities, but spare the programmer the need to learn more than a few new operators. At the same time we provide all the power and flexibility needed to generate test data

satisfying complex invariants, in a language the programmer already knows. By linking generators to types via Haskell's class system, we relieve the programmer of the need to specify generators at all in many cases, and where they must be specified, the programmer's work is usually limited to specifying generators for his or her own new types.

4.6.4 On Randomness

We have encountered some interesting problems in reasoning about programs which use random number generation. In particular, the `Gen` monad which QuickCheck is based on is not a monad at all! Consider the first monad law:

```
return x >>= f = f x
```

Since our implementation of bind splits its random number seed to yield the seeds passed to each operand, then `f` is passed *different* seeds on the two sides of the equation, and may therefore produce different results. So the law simply does not hold. Morally, however, we consider the law to be true, because the two sides produce the same *distribution* of results, even if the results differ for any particular seed.

But what, precisely, do we mean by 'morally'? We cannot fix the problem just by reinterpreting equality for the `Gen` type, claiming the two sides are just different representations of the same abstract generator. This isn't good enough, because we can actually observe the difference at other types by supplying a random number seed - something we have to be able to do if the `Gen` type is to be useful. Instead we have to reinterpret what we mean by program equivalence in the presence of random number generation.

We note that this difficulty is by no means confined to Haskell: the imperative program

```
a := random(); b := random(); c := a - b;
```

is morally equivalent to the same program with the assignments reversed in the same sense, but of course produces a different result. There is some interesting semantic theory to be done here.

4.6.5 On Lazy Evaluation

We have argued in the past that lazy evaluation is an invaluable programming tool, that radically changes the way programs can be structured [57]. Yet QuickCheck is (of course) only able to test computable properties. Is there a conflict here?

In fact, the conflict is much less than one might imagine. As we have shown above, we can perfectly well use infinite structures in specifications, provided

the properties we actually test are computable – for example, we can test that arbitrarily long prefixes of infinite lists are equal, rather than comparing the lists themselves. Our `Gen` monad has a lazy bind operation (because we split the random number seed, rather than threading it through first one operand, then the other), and so we can freely define generators that produce infinite results. What we cannot do is *observe* non-termination in a test result. So we cannot test, for example, the property

```
reverse (xs++undefined) == undefined
```

On the other hand, in a sense a human tester cannot observe non-termination either, and if we have been able to test lazy programs satisfactorily by hand so far, then we are not in a worse position if we use QuickCheck. Yet a human tester can observe that `reverse (xs++undefined)` produces an error message (from the evaluation of `undefined`) without producing any other output first, and can thus infer that the property above holds. The problem is that the Haskell standard provides no way for a *program* to make the same observation. Yet there are various extensions of Haskell which do indeed make this possible. Some work done by Andy Gill has shown that, given such extensions, we could formulate and check properties such as the one above using QuickCheck also.

4.6.6 Some Reflections

We are convinced that one of the major advantages of using QuickCheck is that it encourages us to formulate formal specifications, thus improving our understanding of our programs. While it is open to the programmer to do this anyway, few really do, perhaps because there is little *short term* payoff, and perhaps because a specification is of less value if there is no check at all that it corresponds to the implemented program. QuickCheck addresses both these issues: it gives us a short-term payoff via automated testing, and some reason to believe that properties stated in a module actually hold.

We have observed that the errors we find are divided roughly evenly between errors in test data generators, errors in the specification, and errors in the program. The first category is useless to discover (except insofar as it helps with further testing) – it tells us nothing about the actual program. The third category is obviously useful – in a sense these are the errors we test in order to find. But the second category is also important: even if they do not reveal a mistake in the code, they do reveal a misunderstanding about what it does. Correcting such misunderstandings improves our ability to make use of the tested code correctly later.

When formulating specifications one rapidly discovers the need for a library of functions that implement common mathematical abstractions. We are developing an implementation of finite set theory for use with QuickCheck; many of the abstractions in it are too inefficient to be of much use in programs, but in specifications, where the object is to state properties as clearly and simply as

possible, they come into their own. Because of this difference in purpose, there is a need for libraries specifically targeted at specifications.

The major limitation of QuickCheck is that there is no measurement of test *coverage*: it is up to the user to investigate the distribution of test data and decide whether sufficiently many tests have been run. Although we provide ways to collect this information, we cannot compel the programmer to use them. A programmer who does not risks gaining a false sense of security from a large number of inadequate tests. Perhaps we could define adequacy measures just on the generated test data, and thus warn the user at least in this kind of situation.

4.7 Conclusions

We have taken two relatively old ideas, namely specifications as oracles and random testing, and found ways to make them easily available to Haskell programmers. Firstly, we provide an embedded language for writing properties, giving expressiveness without the learning cost. The language contains convenient features, such as quantifiers, conditionals and test data monitors. Secondly, we provide type-based default random test data generators, including random functions, greatly reducing the effort of specifying them. Thirdly, we provide an embedded language for specifying custom test data generators, which can be based on the default generators, giving a finer control over test data distribution. We also introduce a novel way of controlling size when generating random elements of recursive data types.

Further, we demonstrate that the combination of these old techniques works extremely well for Haskell. The functional nature allows for local and fine-grained properties, since all dependencies of a function are explicit. And precisely random testing is known to work very well for small, fine-grained programs, and is effective in finding faults.

Lastly, the tool is lightweight and easy to use, and provides a short-term payoff for explicitly stating properties of functions in a program, which greatly increases the understanding of the program, for the programmer as well as for documentation purposes.

Acknowledgements: We would like to thank Andy Gill, Chris Okasaki, and the anonymous referees for their useful comments on this paper.

Chapter 5

Lava 2000

This chapter is based on a tutorial used at a Chalmers course *Hardware Description and Verification* in 2000, and was written together with Mary Sheeran [25].

5.1 Introduction

Lava is an experimental tool for hardware design and verification. Using Lava, one can describe circuits using a simple functional hardware description language. The descriptions are short and sweet, and do not suffer from the verbosity of more standard hardware description languages (HDLs) like VHDL and Verilog. On the other hand, we cannot express the same things as in these large, expressive (and complicated) languages. For example, we cannot express low level details about timing. What we can express very nicely, though, is the ways in which circuits are built from sub-circuits. Lava facilitates the description of *connection patterns* so that they are easily reusable. For some kinds of circuits, for example in signal processing, this is exactly what we want to do. Lava also provides many different ways of analysing our circuit descriptions. We can simulate circuits, just as with more standard HDLs, but we can also use symbolic methods to generate input to analysis tools such as automatic theorem provers and model checkers. Indeed, the same methods are used to generate structural VHDL from Lava circuit descriptions. Our aim in this tutorial is to gently introduce this new style of circuit design and analysis, by means of examples.

Lava is used at Chalmers as a platform for experiments in the formal verification of hardware [17, 15]. (Note, however, that both of these references are about an older version of Lava, in which circuit descriptions are a bit more complicated.) Satnam Singh, on the other hand, uses Lava in real industrial design projects at Xilinx Inc., one of the main suppliers of Field Programmable Gate Arrays (FPGAs). In particular, Lava has been used with great success in the development of FPGA cores such as filters and Bezier curve drawing circuits, and of customer applications such as digital signal processing for high speed networks and for high performance graphics applications.

Lava really consists of a simple hardware description language embedded in the powerful functional programming language Haskell. So it can be seen as a domain specific language embedded in a general purpose programming language. We describe circuits by writing Haskell programs – and the Lava system itself consists of a set of Haskell modules that give the user various facilities. The embedded language is quite similar to the Lustre synchronous dataflow language [47]. The idea of using a functional programming language to describe hardware was first proposed in the early eighties [108, 109, 60], and there has been quite a lot of work in the area since then [110, 105, 68, 85, 28, 40]. Our intention in building the Lava system (together with Singh) is to provide a tool that demonstrates the feasibility of doing circuit design and analysis using a functional language.

The main idea in Lava is that a single circuit description can be analysed in a variety of different ways, by giving different *interpretations* to its components (and sometimes even to its connection patterns). The simplest of these interpretations gives us ordinary simulation. But we can do much more. We can allow *symbolic* rather than concrete data to flow in the circuit, and in this way

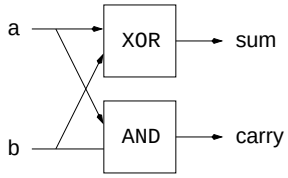


Figure 5.1: A half adder.

collect information about the circuit in various different ways. For example, we can run the circuit on symbolic data and produce expressions on the outputs that indicate how each output is related to the inputs. This can be useful when developing a first implementation. However, the expressions can get too large for humans to interpret. Then, we hook up external analysis tools, such as automatic theorem provers, to help us to analyse our circuits. When we hook up to external theorem provers, we are actually using Haskell as a proof scripting language. This turns out to be very convenient. Similarly, when we hook up to other external tools, such as VHDL-based CAD tools, we use Haskell as a scripting language. One way to view the Lava system is as a tool for linking together and controlling other tools in a unified way! Thus Haskell is used not only to construct circuit descriptions but also to control the tools that process those descriptions. The user sees only one language, rather than having to work with many, as is more usual in the CAD world.

This chapter describes the style of circuit description used in Lava, by means of very simple examples. It emphasises the way in which Lava *combinators* can be used to capture common interconnection patterns. It shows the three most important *interpretations* or circuit analysis methods – simulation, generation of VHDL code, and generation of logical formulas for input to theorem provers. The implementation of Lava is described in Appendix C.

5.2 Some First Examples

As a first circuit description, we are going to define a so-called *half adder* (see figure 5.1). A half adder is a component that is for example used in the implementation of a binary adder. It takes as an input two bits, and adds them up. The result is a *sum* and a *carry* bit. A half adder is usually realized using one `and` and one `xor` gate.

Here is how we define a half adder `halfAdd` in Lava.

```
import Lava

halfAdd (a, b) = (sum, carry)
  where
    sum   = xor2 (a, b)
```

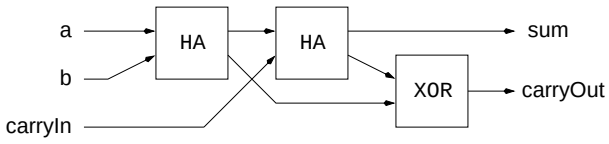


Figure 5.2: A full adder.

```
carry = and2 (a, b)
```

We import a module called `Lava`, which defines a number of operations that we can use to build circuits. Notably, it contains the definitions of the gates `xor2` and `and2`. Appendix D contains a list of such predefined operations.

Note that the order of definitions after a `where` does not matter! Since these circuit components act in parallel, we could just as well have put them the other way around.

One of the things we can do with a circuit is simulating it. Simulation is done in Lava with the operation `simulate`. It takes two arguments; one is the circuit to simulate (in this case `halfAdd`), and the other is the input to the circuit (in this case a pair of bits). Here, we show how to use this operation in the Haskell interpreter Hugs.

```
Main> simulate halfAdd (low,low)
(low, low)
Main> simulate halfAdd (high,high)
(low, high)
```

The second circuit we define is a *full adder* (see figure 5.2), a component `fullAdd` that consists of two half adders.

```
fullAdd (carryIn, (a, b)) = (sum, carryOut)
  where
    (sum1, carry1) = halfAdd (a, b)
    (sum, carry2)  = halfAdd (carryIn, sum1)
    carryOut       = xor2 (carry1, carry2)
```

Note that, just like the half adder, this circuit has *one* input. This one input consists of a pair of a bit and a pair of bits. We could also have represented the input as a triple of bits, but we shall later see why we made this particular choice.

5.2.1 Generating VHDL

Given a Lava circuit description, we can generate VHDL from it, by using the operation `writeVhdl`. It takes two arguments, the name of the VHDL definition as a string, and the circuit.

```
use work.all;

entity                architecture
  fullAdd               structural
is                    of
port                    fullAdd
  -- clock            is
  ( clk : in bit        signal w1 : bit;
  -- inputs             signal w2 : bit;
  ; cin : in bit        signal w3 : bit;
  ; a : in bit        begin
  ; b : in bit          c_w1   : entity and2 port map (clk, a, b, w1);
  -- outputs            c_sum  : entity and2 port map (clk, w1, cin, sum);
  ; sum : out bit       c_w2   : entity xor2 port map (clk, a, b, w2);
  ; cout : out bit      c_w3   : entity xor2 port map (clk, w1, carryIn, w3);
  );                    c_cout : entity xor2 port map (clk, w2, w3, cout);
end entity fullAdd;   end structural;
```

Figure 5.3: The VHDL code for the full adder in `fullAdd.vhd`.

```
Main> writeVhdl "fullAdd" fullAdd
Writing to file "fullAdd.vhd" ... Done.
```

The VHDL file that is generated will assume that there are definitions of the gates. The Lava distribution provides these definitions in the file `Lava2000/Auxiliary/lava.vhd`. We must load this file into the VHDL working library and compile it.

Normally, the VHDL generator gives names to the inputs and outputs automatically. If we want to give names to the input ourselves, we can do this by using the operation `writeVhdlInput`. Here is how we use it:

```
Main> writeVhdlInput "fullAdd" fullAdd
                       (var "cin", (var "a", var "b"))
Writing to file "fullAdd.vhd" ... Done.
```

And lastly, if we also want to give names for the outputs, we can use the operations `writeVhdlInputOutput`. Here is how we use it:

```
Main> writeVhdlInputOutput "fullAdd" fullAdd
                       (var "cin", (var "a", var "b"))
                                          (var "sum", var "cout")
Writing to file "fullAdd.vhd" ... Done.
```

See figure 5.3 for the result of this last operation.

5.2.2 Recursion over Lists

A *bit adder* takes a pair of inputs. The first part is a carry bit, the second part is a binary number, represented as a list of bits. The bit adder will add the bit

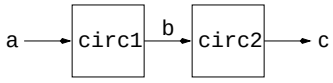


Figure 5.4: Serial composition of `circ1` and `circ2`.

to the binary number, resulting in a binary number and a carry out.

We define a bit adder `bitAdder` in Lava by recursion over the list of bits. There are two cases. Either the list is empty, denoted as `[]`, and there is nothing to add. Or the list has at least one element `a`, and we can split the list up in two parts, `a` and `as`, denoted as `a:as`. In this case, we will use an half adder to add `a` and the carry, and *recursively* add the resulting carry to the rest of the binary number.

```
bitAdder (carryIn, [])   = ([], carryIn)

bitAdder (carryIn, a:as) = (sum:sums, carryOut)
  where
    (sum, carry)     = halfAdd (carryIn, a)
    (sums, carryOut) = bitAdder (carry, as)
```

A more complicated circuit is the circuit `adder` that takes a carry and a pair of binary numbers, and adds them up. This is called a binary adder. The recursive structure is almost the same, but we are doing *simultaneous* recursion over both binary numbers.

```
adder (carryIn, ([], []))     = ([], carryIn)

adder (carryIn, (a:as, b:bs)) = (sum:sums, carryOut)
  where
    (sum, carry)     = fullAdd (carryIn, (a, b))
    (sums, carryOut) = adder (carry, (as, bs))
```

5.2.3 Connection Patterns

Looking at the two circuit definitions in the previous section, we can see that they have a lot in common. Even though the *gates* that they use are different, their *structure* is very similar.

In Lava, we can capture these common structures in *connection patterns*. Connection patterns are higher-order circuits; circuits that build circuits from other circuits.

A very common connection pattern is the *serial* composition `serial` of two circuits (see figure 5.4). It is a circuit *parametrized* by two circuits `circ1` and `circ2`. This means that `serial circ1 circ2` is a circuit, which feeds its input

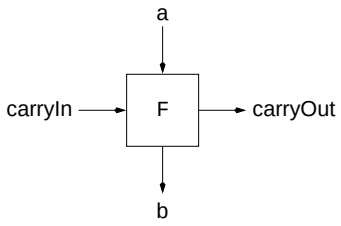


Figure 5.5: A circuit F processing an input `a` and a carry.

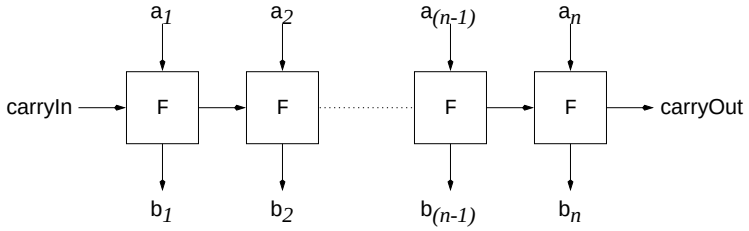


Figure 5.6: The pattern `row` F, connecting n instances of F.

`a` to `circ1`, connects the output `b` of it to the input of `circ2`, and results in that output `c`.

```
serial circ1 circ2 a = c
  where
    b = circ1 a
    c = circ2 b
```

More interesting connection patterns appear when we look at recursive circuit structures. For example, instead of the half adder circuit in the `addBit` definition, we can plug in any other circuit. The result consists of a *row* of smaller circuits (see figure 5.5 and 5.6).

Here is how we define the `row` connection pattern.

```
row circ (carryIn, [])   = ([], carryIn)

row circ (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry)     = circ (carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

Once we have made this definition, we do not need to use recursion anymore to define circuits of this specific pattern.

Here are alternative definitions of `bitAdder` and `adder`:

```
bitAdder' (carry, inps) = row halfAdd (carry, inps)
adder'    (carry, inps) = row fullAdd (carry, inps)
```

Doing so not only makes the circuit descriptions shorter. For somebody who knows how the pattern `row` is defined, it is much easier to understand the circuit definition. It is also very convenient to be able to mix these two different styles of descriptions.

Note that the reason why we chose the input to `fullAdd` to have the particular form it has is because we wanted it to be used with `row`. We will always use that convention, so that it is easier to use circuits with predefined connection patterns.

Even shorter definitions of the same circuits are:

```
bitAdder' = row halfAdd
adder'    = row fullAdd
```

Note that the *type* of `adder'` is slightly different from `adder`.

5.2.4 Arithmetic

In Lava, we can not only deal with low-level wire types like bits, and gates like `and2` and `xor2`, but also with more abstract wire types and gates. One of these types is integers.

On these integers, we have operations corresponding to abstract gates over integers. A list of these gates can be found in Appendix D.

A simple circuit using these arithmetical gates is called `numBreak`. It takes a number as input, and has a pair of a bit and a number as output. The bit in the pair corresponds to the value of the first binary digit of the number; the resulting number is the input number divided by 2.

```
numBreak num = (bit, num')
  where
    digit = imod (num, 2)
    bit   = int2bit digit
    num'  = idiv (num, 2)
```

The circuit `int2bit` converts a number into a bit, by transforming a 0 into `low`, and any other number into `high`.

We can use this arithmetical circuit to build a circuit that converts a number into a binary number, that is, a list of bits. The circuit takes a parameter, corresponding to the size of the list it has to produce, and has as input the number that needs to be converted.

The converter `int2bin` converts an integer to a binary number. It has an extra *parameter*, which specifies the number of bits the binary number should have. Note again that parameters of circuits are different from inputs; `int2bin` is not really a circuit, but `int2bin 16` is. We define this circuit by recursion over the size of the binary number.

```
int2bin 0 num = []

int2bin n num = (bit:bits)
  where
    (bit,num') = numBreak num
    bits       = int2bin (n-1) num'
```

Other arithmetical gates include `plus`, `times`, etc.

Here are some example simulations of these circuits:

```
Main> simulate numBreak 7
(high,3)

Main> simulate (int2bin 3) 7
[high, high, high]
```

5.3 Verification

In this section we describe how we can define properties of circuits, and how we can formally verify these properties using a theorem prover.

5.3.1 Simple Properties

The main kind of properties of circuits we deal with in Lava are so-called *safety properties*. These are properties which can be defined in such a way that they state that some condition is *always* true (or, equivalently, never false).

Here is an example; a property that checks that the outputs of a half adder are never both true.

```
prop_HalfAddOutputNeverBothTrue (a, b) = ok
  where
    (sum, carry) = halfAdd (a, b)
    ok           = nand2 (sum, carry)
```

Note that this property looks pretty much like a normal circuit definition, and in fact it is.

The actual verification question is: does this property circuit always yield true, no matter what the input is? To answer the question, we use the Lava operation `verify`.

```
Main> verify prop_HalfAddOutputNeverBothTrue
Proving: ... Valid.
```

This process works in the following way. Just as we can generate VHDL from a circuit description, we can also generate a logical formula representing the circuit. This logical formula is then given to an external theorem prover which will prove (or disprove) the validity of the formula. The result is then taken back into Lava.

Here is another example; we formulate that a full adder does not care about the order of the arguments, it will always produce the same result. This property is in general called *commutativity*.

```
prop_FullAddCommutative (c, (a, b)) = ok
  where
    out1 = fullAdd (c, (a, b))
    out2 = fullAdd (c, (b, a))
    ok   = out1 <==> out2
```

Note that, since we are not interested in the exact shape of the output of the two full adders, we can just give a name to the whole output, in this case `out1` and `out2`. Another thing to notice is that we use the general equality `<==>`.

5.3.2 Quantification

The commutativity property is not only true for full adders, but also in general for binary adders. Here is how we state that property:

```
prop_AdderCommutative (as, bs) = ok
  where
    out1 = adder (low, (as, bs))
    out2 = adder (low, (bs, as))
    ok   = out1 <==> out2
```

The problem is that this property holds for *all* circuit sizes, but we can only verify it for specific sizes! This is because it is very hard to verify properties automatically for all sizes.

So, instead of verifying it for all sizes, we will pick specific sizes and verify the property for those. Thus, we define a new property, which is explicit about what size of input we want to verify the property.

```
prop_AdderCommutative_ForSize n =
  forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      prop_AdderCommutative (as, bs)
```

This property means: "for all lists of size n called `as`, and for all lists of size n called `bs`, the property that the adder is commutative holds for `(as, bs)` as input".

Now, we can verify the property using `verify`. We can of course do this for more than one size.

```
Main> verify (prop_AdderCommutative_ForSize 4)
Proving: ... Valid.

Main> verify (prop_AdderCommutative_ForSize 32)
Proving: ... Valid.
```

5.3.3 Helping Verification

The formulas that we generate can sometimes be quite hard. It is often the case that, the larger the formula, the harder it is to verify it. So sometimes, it is necessary to help the prover a bit.

One way of doing this is to split up a large proof into a number of smaller ones. Then, we can prove these parts one by one, and for each new proof, we can assume that the parts that are already proved are actually true.

In Lava, we can do this by verifying not just *one* question, but a whole list of questions. For example, to verify that two circuits have the same output, we can prove this pin-by-pin instead of at one go. To do this, we have to define *one-by-one equality*.

```
equalOneByOne ([],   [])   = []

equalOneByOne ([],   y:ys) = [low]

equalOneByOne (x:xs, [])   = [low]

equalOneByOne (x:xs, y:ys) = eq : eqs
  where
    eq  = equal (x, y)
    eqs = equalOneByOne (xs, ys)
```

Note that the comparison fails if the lists are not of the same length. To check that a binary adder is commutative, we can now instead say:

```
prop_AdderCommutative' (as, bs) = oks
  where
    out1 = adder2 (as, bs)
    out2 = adder2 (bs, as)
    oks  = equalOneByOne (out1, out2)

prop_AdderCommutative_ForSize' n =
  forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      prop_AdderCommutative' (as, bs)
```

After defining a property that poses such a list of questions, we can use the theorem prover to prove them.

```
Lava> verify (prop_AdderCommutative_ForSize' 4)
Proving: pin 1 ... Valid.
Proving: pin 2 ... Valid.
Proving: pin 3 ... Valid.
Proving: pin 4 ... Valid.
Proving: pin 5 ... Valid.
--
Result: Valid.
```

Note that in the proof for `pin 2`, we can assume that `pin 1` holds, and in the proof for `pin 3`, we can assume that `pin 1` and `pin 2` holds, etc.

5.4 Sequential Circuits

In this section we describe how to deal with sequential circuits in Lava. Sequential circuits in Lava are synchronous circuits, which means that there is one global clock affecting all delay components in the circuit.

5.4.1 The Delay Component

A new component in sequential circuits is the *delay* component. It is a circuit with one parameter (the initial output of the delay) and one input, which becomes its output in the next clock cycle.

Here is an example of a simple circuit called `edge`, that checks if its input has changed with respect to its previous input. It uses a delay component to remember the previous input.

```
edge inp = change
  where
    inp'   = delay low inp
    change = xor2 (inp, inp')
```

We can simulate a sequential circuit by using the operation `simulateSeq`. It needs a circuit and a list of inputs. The list of inputs is interpreted as the different inputs at each clock tick.

```
Main> simulateSeq edge [high, low, low, high]
[high, high, low, high]
```

Here is another sequential circuit, which is called `toggle`. It has an internal state, which it outputs, and it takes one input. If the input is high, it changes the state. If not, it stays the same.

```
toggle change = out
  where
    out' = delay low out
    out  = xor2 (change, out')
```

As we can see, the definition of `out'` is dependent on `out`, whose definition is dependent on `out'`. Thus, there is a loop in the circuit. Loops are not allowed in combinational circuits, since the meaning of such circuits is unclear. But in sequential circuits, they are essential to implement any interesting behavior.

Simulating `toggle` gives:

```
Main> simulateSeq toggle [high, low, low, high]
[high, high, high, low]
```

5.4.2 Multiple Delays

We have seen how we can delay a signal *one* time instant, so that we can refer to the signal's previous value. Sometimes, we want to delay a signal multiple time instances. We can do this by defining a parametrized circuit, called `delayN`. It has two parameters, n, the number of delays to use, and `init`, the initial values of these delays.

We use recursion over n to define this circuit.

```
delayN 0 init inp = inp

delayN n init inp = out
  where
    out  = delay init rest
    rest = delayN (n-1) init inp
```

A useful sequential circuit that we can implement using `delayN`, is called `puls`. It has no inputs, one output, and one parameter n. Its output is normally low, except on the n-th, $2n$-th, $3n$-th, ... clock tick, where it outputs high.

We implement the circuit by creating $n-1$ delay components in a row, initialized by `low`, ended with one delay component initialized by `high`.

```
puls n () = out
  where
    out  = delayN (n-1) low last
    last = delay high out
```

Note that we need to use a loop back here. This implementation is not optimal, in the sense that it uses too many delay components.

Simulating `puls 3` gives:

```
Main> simulateSeq (puls 3) [(), (), (), (), (), (), ()]
[low, low, high, low, low, high, low]
```

5.4.3 Counters

An n-bit counter is a circuit that outputs an n-bit binary number at every clock tick, starting with 0, and increasing it by 1 every clock tick. We implement this by keeping an internal state, which is a binary number. The circuit takes one parameter, which indicates the number of bits to use, and has no inputs.

```
counter n () = number'
  where
    number'            = delay (zeroList n) number
    (number, carryOut) = bitAdder (high, number')
```

We use the function `zeroList`, which creates a list of n zeros, denoting the initial value. Note that the delay component not only works for bits, but also for example for pairs of bits and lists (as in this case).

Simulating `counter` gives:

```
Main> simulateSeq (counter 3) [(), (), ()]
[[low, low, low], [high, low, low], [low, high, low]]
```

A variant on this circuit is the *up-counter*, which takes an input, which indicates if the number should increase or not. In this case, we want the desired increase to take effect immediately, so we output the number *before* we delay it.

```
counterUp n up = number
  where
    number'            = delay (zeroList n) number
    (number, carryOut) = bitAdder (up, number')
```

Simulating `counterUp` gives:

```
Main> simulateSeq (counterUp 3) [high, low, high]
[[high, low, low], [high, low, low], [low, high, low]]
```

An alternative definition of the up-counter is the following. We define a so-called *infinite circuit*, a circuit with an infinite amount of outputs.

```
counterUpInfinite up = number
  where
    number'            = delay (repeat zero) number
    (number, carryOut) = bitAdder (up, number')
```

In this way, the user of the circuit has to determine how many of these actual outputs will be used.

5.4.4 Sequentialization

In chapter 5.2, we have seen a combinational binary adder. As an input, it takes two n-bit binary numbers, and adds them up. For big n, this circuit can get quite large, which means it takes a lot of circuit area and consumes a lot of power, and needs a low clock frequency to work properly.

We can make use of the regularity in the circuit to make a small version of the circuit that however needs several clock cycles to compute the result. If we apply this technique to the binary adder, we obtain a *sequential adder*. It takes one digit of both binary numbers at each clock cycle. This is sometimes called *bit serial*.

We can implement this by storing the carry as an internal state, so that the current carry-in of the circuit is the previous carry-out.

```
adderSeq (a,b) = sum
  where
    carryIn        = delay low carryOut
    (sum,carryOut) = fullAdd (carryIn, (a,b))
```

Simulating `adderSeq` gives:

```
Main> simulateSeq adderSeq [(high,low), (high,high), (low,high)]
[high, low, low]
```

Because we find that many sequential circuits have this structure, we define a *sequential* connection pattern, called `rowSeq` which builds a virtual row of circuits, but interpreted *over time*.

```
rowSeq circ inp = out
  where
    carryIn         = delay zero carryOut
    (out, carryOut) = circ (carryIn, inp)
```

Worth noting is that we make use of the generic `delay` and `zero` component here. The structure is exactly the same as in the sequential adder.

Recalling the definition of a binary adder in terms of `row`, we can repeat it and implement a sequential adder in terms of `rowSeq`:

```
adder'    = row    fullAdd  -- combinational
adderSeq' = rowSeq fullAdd  -- sequential
```

In this way, using a connection pattern to define a combinational circuit helps us to define the sequential version of the circuit.

5.4.5 Variations on `rowSeq`

The sequential row connection pattern is sometimes useful, but certainly not always. If we use it to implement a sequential adder, as we did, we can only use it to add up "infinitely big" binary numbers. The addition never ends, so we can never start over by adding two new binary numbers.

Therefore, it is handy to have a variant on the `row` connection pattern that allows starting over. We call it `rowSeqReset`, and it takes one extra input `reset`. When `reset` is high, the internal carry state is reset to `zero`.

```
rowSeqReset circ (reset,inp) = out
  where
    carryIn         = delay zero carry
    carry           = mux (reset, (carryOut, zero))
    (out, carryOut) = circ (carryIn, inp)
```

We use the standard multiplexer component `mux` here, which chooses the left or right component of an input pair, depending on if the first incoming signal is low or high, respectively.

Now we can define a resettable sequential adder `adderSeqReset` as follows:

```
adderSeqReset = rowSeqReset fullAdd
```

Very often, it is the case that the internal carry state has to be reset periodically, that is, on every n-th, $2n$-th, ... clock tick. Therefore, we create a third sequential row variation, which takes a parameter n, which indicates the reset period.

```
rowSeqPeriod n circ inp = out
  where
    reset = puls n ()
    out   = rowSeqReset circ (reset, inp)
```

Now we can define a sequential adder `adderSeqPeriod` adding n-bit numbers as follows:

```
adderSeqPeriod n = rowSeqPeriod n fullAdd
```

5.5 Sequential Verification

In this section we describe how we can verify properties of sequential circuits. We restrict ourselves to sequential safety properties.

5.5.1 Sequential Safety Properties

When defining properties about sequential circuits, we can in principle use the same techniques as we did with combinational circuits. Let us take a look at some examples.

Here is how we can compare the two sequential adders from section 5.4.4.

```
prop_SameAdderSeq inp = ok
  where
    out1 = adderSeq inp
    out2 = adderSeq' inp
    ok   = out1 <==> out2
```

Here is another example; the composition of `edge` and `toggle` from section 5.4.1 gives the identity circuit.

```
prop_ToggleEdgeIdentity inp = ok
  where
    mid = toggle inp
    out = edge mid
    ok  = out <==> inp
```

The properties we can describe in this way are called *sequential safety properties*. Recall that safety properties are properties which can be described as a circuit with one output, which should always be true (or never be false) for the property to hold.

Examples of properties which are not safety properties are for example *liveness* properties. These can assert that a certain condition must hold at some time in the future, for example.

5.5.2 Sequential Logic

Apart from the techniques we used to define combinational properties, there are also special techniques we can apply to define sequential properties.

- When we want to refer to values of signals at different time instances, we can use a `delay` to get access to previous values. But one should be careful about what initial value you choose for this use of delay.
- When we want a certain property only to be true when a certain condition holds, which does not necessarily hold all the time, we can use logical implication. Implication is implemented by the Lava gate `impl`, and also by the binary operator `==>`.

Here is an example. Suppose we want to define the following property about the `toggle` circuit: "if the input is high, then the current output is different from the previous output".

The way we define this in Lava is:

```
prop_ToggleTogglesWhenHigh inp = ok
  where
    out    = toggle inp
    out'   = delay low out
    differ = xor2 (out, out')
    ok     = inp ==> differ
```

First, we compute the output `out` from `toggle`. Then, we use a delay component to get access to the previous output `out'`. We define the situation `differ` in which these outputs differ. And then we say: "if the input is `high`, then the outputs differ".

5.5.3 Verification

After defining these properties, we would like to formally verify them. Verification of a sequential property means that we have to prove that the property holds at all times. In Lava, we do this by *induction* over time [106, 16]. It works as follows. (See also Chapter 7.)

Firstly, we have to do the *base case*: proving that the property holds for the *first* time instance. Since looking at just one time instance does not involve time at all, we can use the same techniques as we did in the combinational case.

Then, we do the inductive *step*. We want to prove that *if* the property holds at time t, it also holds at time $t+1$. We do this as follows: we create an *arbitrary* time instance by filling the states of the circuits with fresh variables. Then, we run the circuit once on that state, obtaining an output and new state values. Then we assert that the output is true, and run the circuit on the new state values. Finally, we need to prove that the new output is true.

After proving the base case and the inductive step, we have proved our property. Here is what happens in Lava:

```
Main> verify prop_ToggleEdgeIdentity
Proving: base 1 ... Valid.
Proving: step 1 ... Valid.
--
Result: Valid.

Lava> verify prop_ToggleTogglesWhenHigh
Proving: base 1 ... Valid.
Proving: step 1 ... Valid.
```

```
--
Result: Valid.
```

We give a more detailed explanation of induction in the next section.

5.5.4 Induction

To perform induction on a Lava property, we convert it to a logical formula relating input 'inp' and old state variables '$\mathsf{q_{old}}$' to output 'ok' and new state variables '$\mathsf{q_{new}}$'. Whenever we use a signal-level delay component in a circuit or property, we introduce one state variable.

In this translation, we also introduce a special input, called 'init', which is true only in the first time instance. So, after we have translated the property, we have a logical formula of the following form:

$$T(\mathsf{init}, \mathsf{inp}, \mathsf{q_{old}}, \mathsf{q_{new}}, \mathsf{ok})$$

This formula is usually called the *transition relation*.

A very simple way to prove that the output 'ok' is always true, would be to try proving the following:

$$T(\mathsf{init}, \mathsf{inp}, \mathsf{q_{old}}, \mathsf{q_{new}}, \mathsf{ok}) \Rightarrow \mathsf{ok} \tag{5.1}$$

Unfortunately however, this method does not work very often, because even when the property is always true in any run of the circuit, it might not be true in every possible configuration of the state variables. This is why we use induction.

First, we prove the *base case*, that is: 'ok' is true at the first time instance. In this case, we know that the variable 'init' is true, so we prove:

$$T(\mathsf{true}, \mathsf{inp}, \mathsf{q_{any}}, \mathsf{q_{new}}, \mathsf{ok}) \Rightarrow \mathsf{ok} \tag{5.2}$$

This is usually easy, since initially, we know the values of the state variables.

Then, we prove the *induction step*, that is: if 'ok' is true at time t, it is also true at time $t+1$. So, we are looking at two time instances of the property.

$$\left.\begin{array}{l} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathsf{true}) \\ T(\mathsf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{ok}_2) \end{array}\right\} \Rightarrow \mathsf{ok}_2 \tag{5.3}$$

Note how we connect the different time instances 1 and 2 by reuse of the state variables 'q_2' as new states in the first time instance and as old states in the second time instance. Also note that we use false for the value of 'init' in the second time instance, because we know it is not the initial time instance. And we use true for the value of ok in the first time, since we may assume that the induction hypothesis holds.

If we have proven the two formulas 5.2 and 5.3, then we know that 'ok' must be true at all time instances. This is the basic notion of induction.

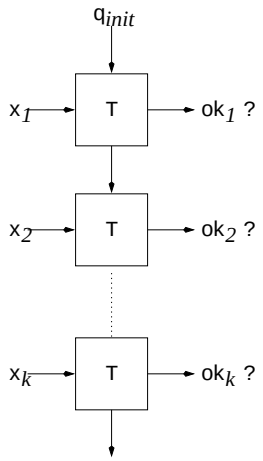


Figure 5.7: Base case for induction with depth k.

5.5.5 Induction With Depth

Unfortunately, the method of induction mentioned in the previous section is not *complete*. This means that there are properties which are true, which cannot be proven by simple induction.

Here is an example: Consider the `toggle` circuit from section 5.4.1 and the `puls` circuit from section 5.4.2. We might want to verify that these circuits do exactly the opposite if `toggle` always has a high input, and `puls` has a period of 2.

```
prop_Toggle_vs_Puls () = ok
  where
    out1 = toggle high
    out2 = puls 2 ()
    ok   = inv (out1 <==> out2)
```

This cannot be proven by normal induction, since the `puls` circuit has two delay components in a row, so it is not enough to look at two time instances at a time.

So, instead, we look at more time instances in the induction proof. We introduce the concept of *induction with depth* k, which means that the base case proves that the first k steps are okay, and the step case may assume that a sequence of k steps went okay, in order to prove that the $k+1$-th step is okay.

Here is the concrete formula for the base case (see also figure 5.7):

$$\left.\begin{array}{r} T(\mathsf{true}, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathsf{ok}_1) \\ T(\mathsf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{ok}_2) \\ \ldots \\ T(\mathsf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathsf{ok}_k) \end{array}\right\} \Rightarrow \mathsf{ok}_1,\ \mathsf{ok}_2,\ \ldots,\ \mathsf{ok}_k \qquad (5.4)$$

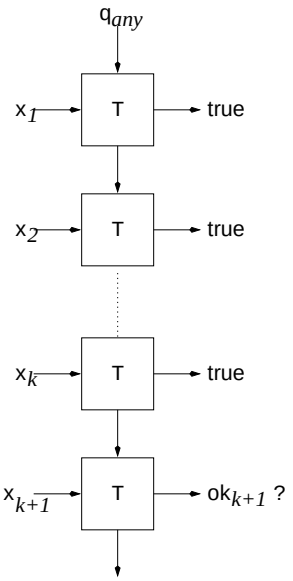


Figure 5.8: Inductive step for induction with depth k.

Note that we use the same trick of reusing the state variables of consecutive times to line up the time instances. Here is the concrete formula for the step case (see also figure 5.8):

$$\left.\begin{array}{r} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathsf{true}) \\ T(\mathsf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{true}) \\ \ldots \\ T(\mathsf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathsf{true}) \\ T(\mathsf{false}, \mathsf{inp}_{k+1}, \mathsf{q}_{k+1}, \mathsf{q}_{k+2}, \mathsf{ok}_{k+1}) \end{array}\right\} \Rightarrow \mathsf{ok}_{k+1} \tag{5.5}$$

So, for any depth k, if we can prove the formulas 5.4 and 5.5, we have proved that 'ok' holds at every time instance. Note that if we choose $k = 1$, then we are back to normal induction again.

Here is what happens when we verify `prop_Toggle_vs_Puls` in Lava:

```
Main> verify prop_Toggle_vs_Puls
Prover: base 1 ... Valid.
Prover: step 1 ... Falsifiable.
Prover: base 2 ... Valid.
Prover: step 2 ... Valid.
--
Result: Valid.
```

So, the verifier realizes that induction depth 1 is not enough for the step to go through, and increases the induction depth automatically. It will keep increasing

the depth until either the base case turns out to be false, or until it manages to prove both the base case and the step case.

If we want to specify a specific depth to do the induction for, we can use the operation `verifyWith`, which takes an extra list of verify options.

```
Main> verifyWith [Depth 2] prop_Toggle_vs_Puls
Prover: base 2 ... Valid.
Prover: step 2 ... Valid.
--
Result: Valid.
```

The operation `verify` is actually just a short-hand for `verifyWith [Depth 1, Increasing]`. With the option `Depth`, one can specify the induction depth. `Increasing` means that it will keep increasing the depth until it proves or disproves the property.

5.5.6 Induction With Restricted States

Unfortunately, even induction with depth is not a complete method. This means that there exists properies which are always true, but for which there exists no k such that the property can be proven by induction with depth k.

An example of such a property is to check if a periodic sequential adder of period 2 is equivalent to a resettable adder which we reset every second clock tick.

```
prop_AdderPeriod2 ab = ok
  where
    sum1 = adderSeqPeriod 2 ab
    two  = delay low (inv two)  -- 010101...
    sum2 = adderSeqReset (two, ab)
    ok   = sum1 <==> sum2
```

Verifying this property results in an infinite loop:

```
Main> verify prop_AdderPeriod2
Prover: base 1 ... Valid.
Prover: step 1 ... Falsifiable.
Prover: base 2 ... Valid.
Prover: step 2 ... Falsifiable.
...
```

The problem is that there exist a lot of state variable configurations that never occur when we run the circuit, but are logically possible. In some cases, these so-called *unreachable states* mess up the induction proof. Even assuming that the property we want to prove is true for a very large number k of consecutive running steps (like we do in the induction step) is not enough to ensure we are

in a reachable state. The reason for this is that we might be running around in the unreachable states in circles for these k steps, so increasing k does not help.

Instead, we will strengthen the induction step by saying that all $k+1$ states we visit in the formula must be distinct. In this way, we ensure that we are not running around in circles.

The new formula for the inductive step becomes:

$$\left.\begin{array}{r} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathsf{true}) \\ T(\mathsf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{true}) \\ \ldots \\ T(\mathsf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathsf{true}) \\ T(\mathsf{false}, \mathsf{inp}_{k+1}, \mathsf{q}_{k+1}, \mathsf{q}_{k+2}, \mathsf{ok}_{k+1}) \\ \mathsf{q}_1 \neq \mathsf{q}_2,\ \mathsf{q}_1 \neq \mathsf{q}_3,\ \ldots,\ \mathsf{q}_{k-1} \neq \mathsf{q}_{k+1},\ \mathsf{q}_k \neq \mathsf{q}_{k+1} \end{array}\right\} \Rightarrow \mathsf{ok}_{k+1} \qquad (5.6)$$

For this method, proving formulas 5.4 and 5.6 for some k is enough to prove the 'ok' holds at all time instances. Moreover, this is a complete method! This means that, if the property holds, there is always a k such that we can prove it by induction with depth k with restricted states.

To use induction with restricted states in Lava, we can use the option `RestrictStates`:

```
Main> verifyWith [RestrictStates,Increasing] prop_AdderPeriod2
Proving: base 1 ... Valid.
Proving: step 1 ... Falsifiable.
...
Proving: base 5 ... Valid.
Proving: step 5 ... Valid.
--
Result: Valid.
```

We needed induction depth 5 for this property. Note that we used the option `Increasing` also, otherwise the verification would have stopped at depth 1. An alternative would have been to use the options `[RestrictStates,Depth 5]`, but it is often difficult to predict the needed induction depth.

Lava also interfaces to other verification tools such as FixIt [1] and VIS [119].

5.6 Time Transformations

In this section, we will see some techniques with which we can compare circuits that operate at different clock rates.

5.6.1 Timing Issues

So far, when we were comparing two circuits, we always assumed that they consumed their inputs and produced their outputs at the same rate. Let us

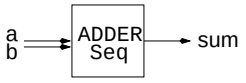


Figure 5.9: A sequential adder.

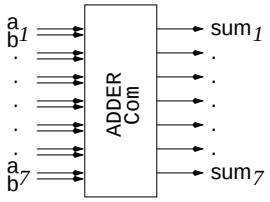


Figure 5.10: A combinational adder.

take a look at an example where this is not the case: comparing a sequential adder against a combinational adder.

The sequential adder (see figure 5.9) takes in a pair of bits every clock tick, and outputs the sum, and remembers the carry for the next clock cycle. The carry is reset every n-th clock tick. Here is how we defined it:

```
adderSeqPeriod n = rowSeqPeriod n fullAdd
```

The combinational adder (see figure 5.10) takes in two n-bit binary numbers and produces the sum as a n-bit binary number in one clock tick. For convenience, we abstract away from the carry. Here is how we define it:

```
adderCom abs = sum
  where
    (sum, carryOut) = adder (low, abs)
```

There are two basic methods for comparing these two circuits.

The first method involves slowing down the combinational adder, so that it takes more clock ticks to calculate the sum. So instead of taking n pairs of bits at a time, it takes them in one-by-one, and when it has gotten all of them, it outputs the sums one-by-one. The circuits now operate at the same rate, and can be compared by conventional methods.

The second method involves speeding up the sequential adder, so that it computes several results in one clock tick. So instead of taking in one pair of bits at a time, it takes in n pairs of bits, and produces n sums in one clock cycle.

5.6.2 Slowing Down

The first technique we describe *slows down* the combinational circuit. So, instead of computing everything in one clock tick, we force it to take n clock ticks

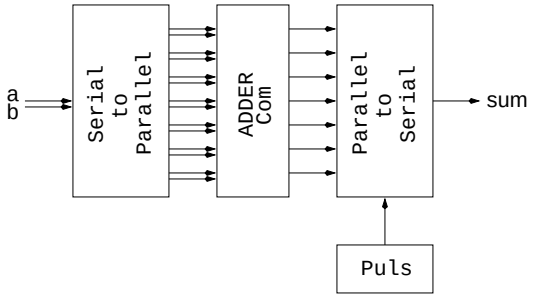


Figure 5.11: The slowed down combinational adder.

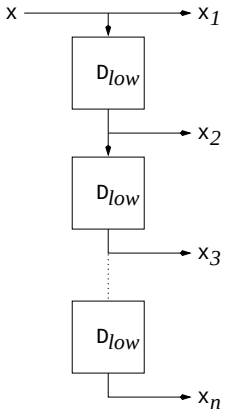


Figure 5.12: A serial to parallel converter.

instead. We do this by transforming the circuit into a circuit that looks just like the sequential version: it takes one input and produces one output at a time (see figure 5.11).

Since the inputs come in one-by-one, we have to wait for n clock ticks until we have the full input available for the circuit. This is done by the *serial to parallel converter* (see figure 5.12). We can implement this component as follows:

```
serialToParallel 1 inp = [inp]

serialToParallel n inp = inp : rest
  where
    inp' = delay zero inp
    rest = serialToParallel (n-1) inp'
```

Then we have to take care of the outputs. At every clock tick, the combinational circuit produces n outputs, but they only make sense on every n-th, $2n$-th, ... clock tick, because then we have the right input. Therefore, we need to add a

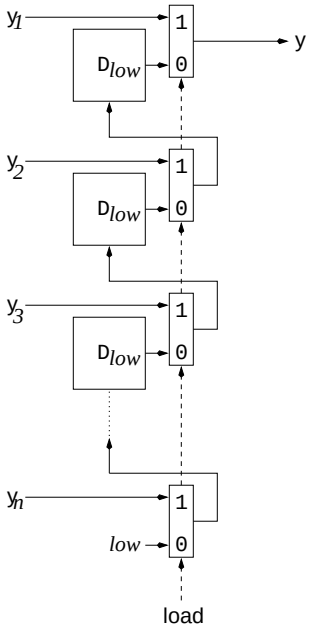


Figure 5.13: A parallel to serial converter.

component on the outputs that spreads out the outputs of the important clock ticks over the other clock ticks. This is done by the *parallel to serial converter* (see figure 5.13). We can implement this component as follows:

```
parallelToSerial (load, [inp]) = out
  where
    out  = mux (load, (low, inp))

parallelToSerial (load, inp:inps) = out
  where
    from = parallelToSerial (load, inps)
    prev = delay low from
    out  = mux (load, (prev, inp))
```

Then, we can put these components together in a new sequential adder:

```
adderSlowedDown n ab = sum
  where
    abs  = serialToParallel n ab
    sums = adderCom abs
    load = puls n ()
    sum  = parallelToSerial (load, sums)
```

The `load` input to the parallel to serial converter is a puls with period n. Let

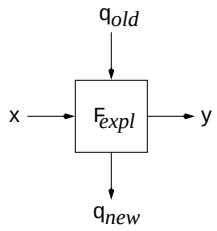


Figure 5.14: A combinational circuit with explicit states q_{old} and q_{new}.

us take a look at how this sequential adder adds up binary numbers for $n = 4$.

clock	1	2	3	4	5	6	7	8	9
input	ab_1	ab_2	ab_3	ab_4	ab'_1	ab'_2	ab'_3	ab'_4	$ab"_1$
output	0	0	0	s_1	s_2	s_3	s_4	s'_1	s'_2

As we can see, the results s_i are delayed by $n-1$ clock ticks. This is of course because the result is computed at the n-th, $2n$-th, ... clock tick. So, when we compare this with the original sequential adder, we have to slow the output of that one down with $n-1$ delay components. Here is the property:

```
prop_AdderSeqSlowedDown n ab = ok
  where
    sum1  = adderSeqPeriod n ab
    sum1' = delayN (n-1) low sum1
    sum2  = adderSlowedDown n ab
    ok    = sum1' <==> sum2
```

Unfortunately, this way of specifying the property introduces a lot of extra logic, and moreover, extra state. This makes the verification of these kind of properties very hard. In particular, the induction methods need an extremely high induction depth. In the next section, we will see a simpler and more direct method for specifying retiming properties.

5.6.3 Speeding Up

Another technique for retiming works as follows. Instead of slowing down the combinational circuit, we speed up the sequential circuit. Unfortunately, this cannot be done by adding retiming components around the circuit. Instead, we *transform* the circuit into another circuit. This is done by a built-in Lava operation, called `timeTransform`.

The idea is that we make the state of the sequential circuit explicit by turning a sequential circuit F into a combinational circuit F_{expl}, that takes in the old state as an extra input, and has the new state as an extra output (see figure 5.14).

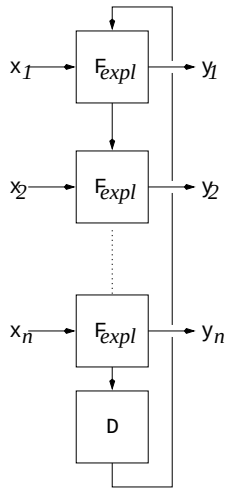


Figure 5.15: A time transformed sequential circuit F.

The next step is to create a column of F_{expl}, where we thread the states through as carry. The last step is to make the state implicit again by adding delay components and a loop back (see figure 5.15).

All this is implemented by Lava's primitive operation `timeTransform`. So, we can make a new adder from the sequential adder, by using time transformation:

```
adderSpedUp abs = sums
  where
    sums = timeTransform (adderSeqPeriod n) abs
    n    = length abs
```

The function `length` computes the length of a list, so that we know what period the sequential adder requires.

The property of comparing the two different adders now looks as follows:

```
prop_AdderSeqSpedUp abs = ok
  where
    sum1 = adderSpedUp abs
    sum2 = adderCom abs
    ok   = sum1 <==> sum2
```

Because this is a property that has a list as an input, we need to be explicit about the length of the list:

```
prop_AdderSeqSpedUp_ForSize n =
  forAll (list n) $ \abs ->
    prop_AdderSeqSpedUp abs
```

Verifying this by induction is easy, and needs induction depth 2 for any n.

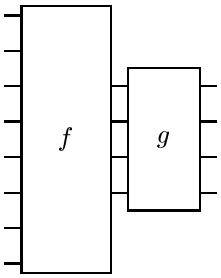


Figure 5.16: `f ->- g`

5.7 More Connection Patterns

In this section, we first review some standard connection patterns, and then consider the problem of describing *tree shaped circuits* and *butterfly circuits*. These are common circuit structures in digital signal processing.

5.7.1 Connection patterns revisited

In an earlier chapter, we saw the `serial` connection pattern, which connects two circuits in series. It is convenient to have an infix version, so that we can write `f ->- g`, instead of `serial f g`, see figure 5.16. Note that serial composition is associative:

```
f ->- (g ->- h)   ===   (f ->- g) ->- h
```

Sometimes we want to compose a list of circuits. We call this `compose`.

```
compose []           inp = inp
compose (circ:circs) inp = out
  where
    x   = circ inp
    out = compose circs x
```

Note that we could have written this definition in a different style, using the serial connection pattern.

```
compose1 []           inp = inp
compose1 (circ:circs) inp = out
  where
    out = (circ ->- compose1 circs) inp
```

We could go even further and drop the circuit inputs (`inp`) from each side of the definitions. The identity circuit (which just returns its input) is written `id`.

This is a definite change of style to one in which the emphasis is on connection patterns.

```
compose2 []           = id
compose2 (circ:circs) = circ ->- compose2 circs
```

All of these styles are equally good, and the choice is really just a matter of taste. In fact it is quite convenient to be able to mix styles, sometimes choosing one and sometimes the other.

Out of `compose`, we can easily make a connection pattern, called `composeN`, that composes several copies of the same circuit in sequence.

```
composeN n circ = compose (replicate n circ)

Main> simulateSeq (composeN 5 inc) [0,2,4,6]
[5,7,9,11]
```

Here `inc` is the circuit that adds one to its integer input.

We also saw the `par` connection pattern: `par f g` takes a pair of inputs, passing the first to `f` and the second to `g`, and combining the results into a pair. The infix version of `par f g` is written `f -|- g`.

A version of `par` that "does" `f` to the first half of a list and `g` to the second half also turns out to be useful. We call this pattern `parl`. First, we define a helper function, `halveList`, which divides a list in two.

```
halveList inps = (left,right)
  where
    left  = take half inps
    right = drop half inps
    half  = length inps `div` 2

Main> simulate halveList [high,low,high,low]
([high,low],[high,low])
```

Then, we define the circuit `append`, which takes a pair of lists of length m and n, and joins them together (or *concatenates* them), to give a list of length $m+n$. This circuit is defined in terms of Haskell's built-in infix list concatenate operator (`++`).

```
append (a,b) = a ++ b
```

Lastly, we define `parl`:

```
parl circ1 circ2 =
  halveList ->- (circ1 -|- circ2) ->- append
```

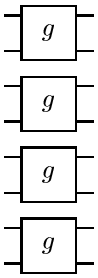


Figure 5.17: `map g`

```
Main> simulate (parl reverse id) [1..16]
[8,7,6,5,4,3,2,1,9,10,11,12,13,14,15,16]
```

Sometimes, we want to perform an operation of each element of a list of signals or bus. For this we use the connection pattern `map`. For example, `map inv` inverts each of a list of bits.

```
Main> simulate (map inv) [high, low, high, low]
[low,high,low,high]
```

Buses need not contain only lists of bits. They can be more structured, so that our circuit descriptions can match the *logical* structure of the circuit. For example, the circuit `map fullAdd` makes perfect sense.

```
Main> simulate (map fullAdd)
        [(low,(high,low)),(high,(high,high)),(low,(high,high))]
[(high,low),(high,high),(low,high)]
```

Figure 5.17 shows a `map` in the case where the input is a 4-list (of pairs or 2-lists).

5.7.2 Tree shaped circuits

Circuits in the shape of trees, like that shown in figure 5.18, can be used to systematically apply a function that combines data values together to a collection of data. A binary tree circuit first combines each half of the input values, using two smaller trees and then combines the two remaining results. One example of such a circuit is an adder tree that adds up a list of numhers.

The outline of the recursive definition of a tree connection pattern is:

```
tree circ [inp] = ... inp ...
tree circ inps  = ... tree circ ... tree circ ... circ ... inps
```

We call the parameter `circ` the *component circuit*. The first line in this outline defines what should be done when we get down to the base case of the recursion.

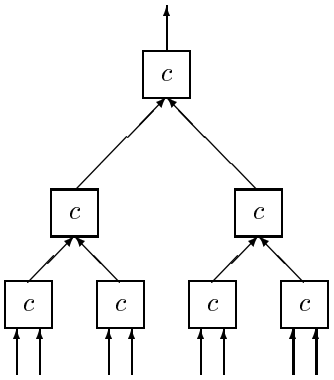


Figure 5.18: A tree shaped circuit

The second line should use two copies of `tree circ` and combine their results using `circ`. Exactly how these definitions should look depends partly on what the component `circ` looks like, and in particular on its type.

For example, if `circ` is a binary function taking a pair of inputs and returning a single output, then it makes sense to make the following definition of a binary tree connection pattern, `binTree`.

```
binTree circ [inp] = inp
binTree circ inps  =
  (halveList ->- (binTree circ -|- binTree circ) ->- circ) inps
```

This gives the behaviour that we expect: a binary tree of `circ` components gets built.

An example use of a tree connection pattern is when we want to build a circuit that adds up a lot of numbers. One way of doing this to make a so-called *adder tree*. To do this, we need a binary adder that adds two n bit numbers, to give an $n+1$ bit number. This means that we must include the carry out in the result. The resulting adder is therefore slightly different from those that we saw earlier. We call it `binAdder`.

```
binAdder (as, bs) = cs ++ [carryOut]
  where
    (cs, carryOut) = adder (low, (as, bs))
```

And here is the definition of our adder tree `addTree`:

```
addTree = binTree binAdder
```

To test it, we wrap the circuit in converters from integer to binary and back.

```
wrapAddTree n =
  map (int2bin n) ->- addTree ->- bin2int
```

```
Main> simulate (wrapAddTree 8) [3,4,5,6,10,9,8,7]
52
```

Beware, this adder tree works only for input lists whose length is a power of two.

5.7.3 Describing Butterfly Circuits

Butterfly circuits are circuits with a particular recursive structure. Figures 5.20 and 5.21 show two such circuits and also indicate their recursive structures by showing, by means of dotted boxes, where to find sub-circuits that themselves have the same recursive structure. It turns out that these two circuits are in fact equivalent: the same network of components can be recursively described in two completely different ways. And indeed it turns out that there are many more ways to describe the same network. We will study some of them.

Butterfly circuits are used for example to build routing networks from switches, and in building efficient sorting circuits. Perhaps the best known butterfly-like circuit is the standard Cooley-Tukey algorithm [29] for computing the Fast Fourier Transform (FFT). We will not consider the FFT here. The twiddle-factors complicate matters a bit. The circuit is not quite as uniform as those that we consider. However, the interested reader is referred to [17], which shows how to describe and compare various FFT circuits in an older version of Lava. For more details about how the verification is actually done, see [15].

In this section, we first introduce two new connection patterns, and then show that butterfly circuits can be made with just these two patterns and serial composition.

The first of these patterns we call `two`. The circuit `two circ` contains two copies of `circ`. The first of these operates on the first half of the input list, and the second on the second half. Each copy of `circ` should have a list as output, and the two resulting lists are appended. This pattern is easily defined in terms of `parl`, which was introduced earlier in this chapter.

```
two circ = parl circ circ

Main> simulate (two reverse) [1..16]
[8,7,6,5,4,3,2,1,16,15,14,13,12,11,10,9]

Main> simulate (two (two reverse)) [1..16]
[4,3,2,1,8,7,6,5,12,11,10,9,16,15,14,13]
```

Related to `two`, we also introduce the pattern `ilv`, for *interleave*. Whereas `two f` applies `f` to the top and bottom halves of a list, `ilv f` applies `f` to the odd and even elements. We define it in terms of the wiring pattern *riffle*, which performs the perfect shuffle on a list. Think of taking a pack of cards, halving it, and then

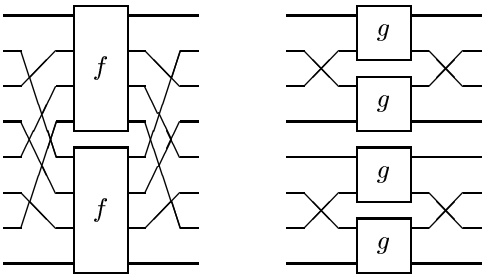


Figure 5.19: `ilv f` and `two (ilv g)`

interleaving the two half packs. If you now *unriffle* the pack, you reverse the process, returning the pack to its original condition. (This is somewhat more difficult to accomplish with aplomb at the poker table.)

```
Main> simulate riffle [1..16]
[1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16]

Main> simulate (riffle ->- unriffle) [1..16]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

Main> simulate unriffle [1..16]
[1,3,5,7,9,11,13,15,2,4,6,8,10,12,14,16]
```

Note that unriffling the sequence from 1 to n divides into its odd and its even elements. We use this fact to define `ilv` in terms of `two`.

```
ilv circ = unriffle ->- two circ ->- riffle

Main> simulate (ilv reverse) [1..16]
[15,16,13,14,11,12,9,10,7,8,5,6,3,4,1,2]

Main> simulate (ilv (ilv reverse)) [1..16]
[13,14,15,16,9,10,11,12,5,6,7,8,1,2,3,4]
```

Figure 5.19 shows `ilv f` and `two (ilv g)`.

We have seen from our examples that it makes sense to apply `two` and `ilv` repeatedly. We will do this so often in the butterfly circuits, that it is useful to define special functions.

```
twoN 0 circ = circ
twoN n circ = two (twoN (n-1) circ)

ilvN 0 circ = circ
ilvN n circ = ilv (ilvN (n-1) circ)
```

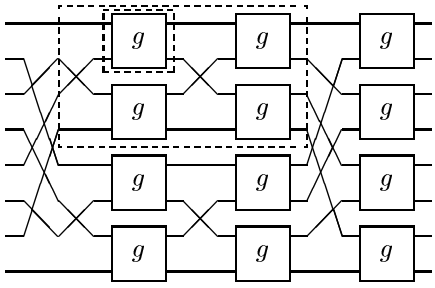


Figure 5.20: `bfly 3 g`

Clearly, there are similarities between these two definitions. We might just as well have defined a function that takes a *connection pattern* as input.

```
iter 0 comb circ = circ
iter n comb circ = comb (iter (n-1) comb circ)
```

Now, we can use `iter n two f` instead of `twoN n f` and `iter n ilv f` instead of `ilvN n f`.

Now we are in a position to define a connection pattern for butterfly circuits, that is circuits, like those shown in figures 5.20 and 5.21, that have a very particular recursive structure. Because the circuits are recursive, the corresponding connection pattern is defined using recursion.

```
bfly 0 circ = id
bfly n circ = ilv (bfly (n-1) circ) ->- twoN (n-1) circ
```

The smallest butterfly is just the identity. A butterfly of size n, for n greater than zero, consists of two interleaved butterflies of size $n-1$, the output of which is fed into a stack of `circ` components, which is made using `twoN`. This connection pattern is shown in figure 5.20, which shows `bfly 3 g`.

The larger dashed box shows one instance of `bfly 2 g`, and there is another instance just below it. These two smaller butterflies are interleaved, so there is actually an *unriffle* to their left and a *riffle* to their right. (Make sure to find these wiring patterns, and look again at the definition of `ilv`.) The two interleaved butterflies feed their outputs into four `g` components, one above the other, that is `twoN 2 g`. And if you look inside the `bfly 2 g` in the outer dashed box, you will find that it again has the same recursive structure.

Strangely enough, the same connection pattern (that is the same netlist and the same order of inputs and outputs, though a possibly different layout) can be described using a different pattern of recursion.

```
bfly1 0 circ = id
bfly1 n circ = ilvN (n-1) circ ->- two (bfly1 (n-1) circ)
```

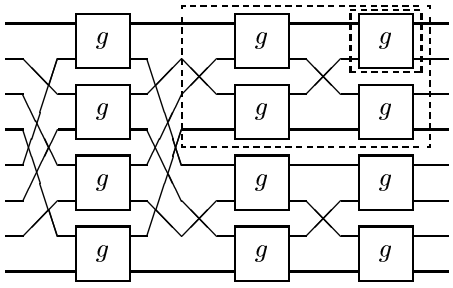


Figure 5.21: `bfly1 3 g`

This time, we start with a repeatedly interleaved stack of basic components, whose outputs are fed into two smaller butterflies, which are combined using `two`. Figure 5.21 shows this recursive decomposition.

It turns out that `ilv (bfly n circ)` is the same as `bfly n (ilv circ)`. (See the question below about `two ilv g` if you want to figure out why.) This means that we can define the butterfly network using a single recursive call, but with a larger component:

```
bfly2 0 circ = id
bfly2 n circ = ilvN (n-1) circ ->- bfly2 (n-1) (two circ)

bfly3 0 circ = id
bfly3 n circ = bfly3 (n-1) (ilv circ) ->- twoN (n-1) circ
```

The surprising thing is that all of these connection patterns give equivalent circuits (for the same size and component).

The original butterfly defintions (`bfly` and `bfly1`) can also be expressed using a tree-like combinator. Take a look at the connection pattern `listTree`, which is a version of `binTree` which works for a component circuit `circ` processing lists.

```
listTree circ [inp] = [inp]
listTree circ inps  = (two (listTree circ) ->- circ) inps
```

You should think about the types involved in this definition.

Replacing that `two` by `ilv`, we get `ilvTree`, a sort of interleaved tree.

```
ilvTree circ [inp] = [inp]
ilvTree circ inps  = (ilv (ilvTree circ) ->- circ) inps
```

If we have a component that takes a pair as input and produces a pair as output, then we can describe a stack of such components by using pairing, unpairing and map as follows.

```
pmap circ = pair ->- map circ ->- unpair

Main> simulate (pmap swap) [1..16]
[2,1,4,3,6,5,8,7,10,9,12,11,14,13,16,15]
```

Then, for inputs of length 2^n, `ilvTree (pmap circ)` is the same as `bfly n circl`, where `circl` is the same as `circ` except that it relates a 2-list to a 2-list.

So what kinds of circuits can we build with these remarkably recursive structures? Well, it turns out that `bfly 3 id` is a complicated way to write the identity function (on lists of length $8n$.) And `bfly n swapl` reverses a list of length 2^n.

```
swapl [a,b] = [b,a]

Main>  simulate (bfly 4 swapl) [1..16]
[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]

Main> simulate (ilvTree (pmap swap)) [1..16]
[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

If we choose our basic component to be the perfect shuffle on lists of length 4, the circuit that we call `s2`, then we find that a butterfly of such components performs the perfect shuffle!

```
s2 [a,b,c,d] = [a,c,b,d]

Main> simulate (bfly 3 s2) [1..16]
[1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16]
```

But all of these examples were just wiring functions. What happens when we add some functionality to the component?

5.7.4 Batcher's Bitonic Merger

One of the best known uses of the butterfly network is in the building of mergers and sorters based on a two-input two-output comparator. Let us start with two abstract comparators that work on integer inputs. One sorts into ascending order, and the other into descending.

```
compUp   [x,y] = [imin (x,y), imax (x,y)]
compDown [x,y] = [imax (x,y), imin (x,y)]

Main> simulate (two compUp) [1,2,4,3]
[1,2,3,4]

Main> simulate (ilv compDown) [1,2,4,3]
[4,3,1,2]
```

It turns out that `bfly n compUp` sorts (into ascending order) a list whose first half is ascending and second half is descending or vice-versa. We call such lists *inc-dec* and *dec-inc* lists. (The merger sorts many other lists too, the so-called *bitonic* lists, but we don't need to worry about them.) This network is known as *Batcher's bitonic merger* [9]. Also, `bfly n compDown` sorts *inc-dec* and *dec-inc* lists into descending order.

```
Main> simulate (bfly 3 compUp) [1,3,5,7,8,6,4,2]
[1,2,3,4,5,6,7,8]

Main> simulate (bfly 3 compDown) [1,3,5,7,8,6,4,2]
[8,7,6,5,4,3,2,1]
```

Knowing that the merger sorts *inc-dec lists* allows us to build a recursive sorter. In fact, we can parameterise the circuit on the comparator (the `comp` parameter), and define both an up and a down sorter at the same time. `sorter n compUp` sorts into ascending order, while `sorter n compDown` sorts into descending order.

```
sorter 0 comp [inp] = [inp]
sorter n comp inps  = outs
  where
    sortL  = sorter (n-1) comp
    sortR  = sorter (n-1) (comp ->- swapl) -- reversed comp
    merger = bfly n comp                   -- bitonic merger
    outs   = (parl sortL sortR ->- merger) inps

Main> simulate (sorter 3 compUp) [8,7,1,2,3,4,6,5]
[1,2,3,4,5,6,7,8]

Main> simulate (sorter 3 compDown) [8,7,1,2,3,4,6,5]
[8,7,6,5,4,3,2,1]
```

Note that our sorter is parameterised on the comparator or two-sorter component. So we have really designed the connection pattern that must be used to connect comparators. We have not in any way tied ourselves down to comparators of a particular type. So, as long as we provide a comparator component of the right type, then we get back a function of the same type that acts as a sorter.

The next step is to refine the comparator component, by choosing a concrete representation for the integer data. Examples of such representations are parallel least significant bit first binary, or serial signed twos complement. The point is that whatever refinement we choose, we can simply plug in the new component into our sorter function. This is an example of how Lava allows us to design connection patterns and then reuse them.

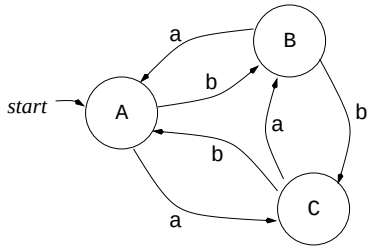


Figure 5.22: An example of a state machine.

An interesting property of sorting circuits made from comparators is that they obey the *zero-one principle*. If such a sorter works correctly on lists of integers containing only zeros and ones, then it works correctly for arbitrary integers. So, we can test an integer sorter by checking that it works on bits! Here, we need the circuit `twoBitSortl` that sorts a two-list of bits:

```
twoBitSortl [a,b] = [min, max]
  where
    min = and2 (a, b)
    max = or2  (a, b)
```

Now, all we need to do is to plug this component into our sorter.

```
Main> simulateSeq (sorter 2 twoBitSortl) (domainList 4)
[[low,low,low,low],[low,low,low,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,high,high,high],[high,high,high,high]
]
```

5.8 Synthesizing Lava Circuits

In this section, we present two examples where we generate a structural Lava circuit from a behavioral specification. We do this by embedding a behavioral description language in Lava. (See also Chapter 6.)

5.8.1 State Machines

A very common way of specifying a sequential system is by constructing a *state machine*. A state machine consists of four parts: a set of states, a set of inputs,

a set of initial states and a transition function. The transition function maps a state and an input to a set of next states. Usually, we draw state machines as pictures. An example of a state machine is pictured in figure 5.22.

In Haskell, here is how we might specify a datatype for representing state machines. We parametrize over the types of the states and the inputs.

```
data StateMachine state inp
  = StateMachine
      { states     :: [state]
      , inputs     :: [inp]
      , initial    :: [state]
      , transition :: state -> inp -> [state]
      }
```

Here is how we can describe the state machine in figure 5.22:

```
theStateMachine =
  StateMachine
    { states     = ["A", "B", "C"]
    , inputs     = ['a', 'b']
    , initial    = ["A"]
    , transition = \state inp ->
                     [ next | (state', inp', next) <-
                                [ ("A", 'a', "C")
                                , ("A", 'b', "A")
                                , ...
                                ]
                            , state == state'
                            , inp == inp'
                            ]
    }
```

Note that the somewhat clumsy definition of the transition function would be easier in an application where the states and inputs actually *mean* something.

Given a specification in terms of a state machine, we would like to be able to translate in into a circuit. One reason for this might be because we want a prototype implementation of the state machine. Another reason might be because we want to verify that a given circuit implementation is equivalent to the translated version.

One method of translating a state machine into a circuit is pictured in figures 5.23 and 5.24. The idea is that every state in the state machine maps to a component in the circuit. The component has a delay element that keeps track of if we are in that state. The component receives messages from other component that activate it, and, depending on the inputs, also sends messages to other components activating them.

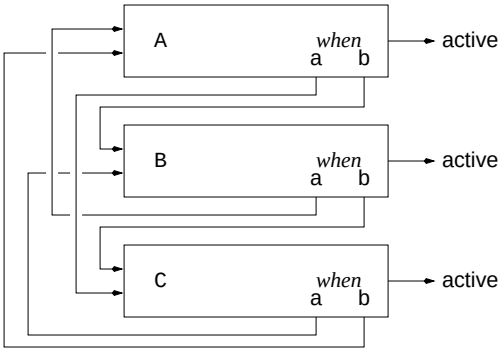


Figure 5.23: A schematic translation of the state machine of figure 5.22.

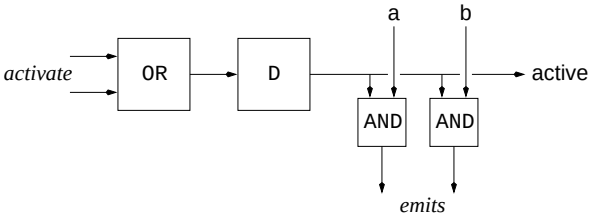


Figure 5.24: A more detailed view of the component belonging to a state.

An advantage of this translation method is that we can be in several states at the same time, allowing for non-deterministic execution of our state machine. A disadvantage is that, even when our state machine is deterministic, we still have one delay component per state, which is often too much.

The type of circuits we are translating state machines to is a circuit from input signals to a list of indicators for each state.

```
type StateCircuit
  = [Signal Bool] -> [Signal Bool]
```

From the two defined types, we can declare the type of our translation function, which takes a state machine into a state circuit.

```
stateMachine :: (Eq inp, Eq state)
             => StateMachine state inp -> StateCircuit
stateMachine machine inSignals = outSignals
  where
    ...
```

First, we define the function `inSignal` which maps an input from the state machine to the corresponding signal wire.

```
    inSignal input =
```

```
  head [ sig
       | (input',sig) <- inputs machine `zip` inSignals
       , input == input'
       ]
```

Then, we create a list of the components, which we use as a lookup table in the rest of the translation.

```
components =
  [ component state
  | state <- states machine
  ]
```

A component for a certain state consists of a pair `(active, emits)`, where `active` is the indicator signal for the state, and `emits` is a lookup table, representing what signal to send to what state.

```
component state = (active, emits)
  where
    init   = state `elem` initial machine
    active = delay (bool init) (activating state)

    emits  =
      [ ( state'
        , and2 (active, inSignal input)
        )
      | input  <- inputs machine
      , state' <- transition machine state input
      ]
```

The declaration of `active` uses one delay component, whose initial value depends on this state being an initial state or not, and whose next value depends on the signals the other components are sending to it (computed using the function `activating`).

The list `emits` is constructed as follows. For every input signal, we use the transition function to check what next states we have. We then send a signal to the component of state if and only if we are active, and we have that input as an incoming signal.

Here is how we define the function `activating`.

```
activating state =
  orl [ activate
      | (_, emits)         <- components
      , (state', activate) <- emits
      , state == state'
      ]
```

For all components, we look at what messages it wants to send, and filter out the signals going to the right state. Then, we take the or of all these signals.

Finally, we can create the list of state indicators, by taking the first output of the components.

```
outSignals =
  [ active
  | (active, _) <- components
  ]
```

Here is how we can make the circuit for the state machine we specified earlier.

```
theCircuit (a, b) = (inA, inB, inC)
  where
    [inA, inB, inC] =
      stateMachine theStateMachine [a, b]
```

5.8.2 Behavioral Descriptions

Another way of specifying the behavior of a circuit is by a *behavioral description language*. Examples of these kind of languages are behavioral VHDL, Verilog, Esterel, etc. The idea is to write a program in such a language, and then transform the program to a circuit with the same behavior.

We show how to compile programs in a very simple description language to a circuit. We call the language *Flash*. Here is a Haskell datatype respresenting Flash programs:

```
data Flash out
  = Skip
  | Emit out
  | Wait
  | IfThenElse (Signal Bool) (Flash out, Flash out)
  | While (Signal Bool) (Flash out)
  | Flash out :>> Flash out
  | Flash out :|| Flash out
```

A Flash program can send out messages of type `out`. Running a Flash program takes a number of clock cycles. Here is the informal semantics of Flash constructs:

- `Skip`: This program does not send any messages, and takes no time to execute.
- `Emit msg`: This program sends out the message `msg`, and takes no time to execute.

- `Wait`: This program does not send any messages, and takes 1 clock cycle to execute.
- `IfThenElse cond (p1, p2)`: If the signal `cond` is high, it executes `p1`, and sends the messages `p1` sends, and takes as long time as `p1` takes. If `cond` is low, the same, but for `p2`.
- `While cond p`: If `cond` is high, then it executes `p`, and sends the messages `p` sends, waits for the amount of time `p` takes to finish, and then tries to execute the program again. If `cond` is low, it finishes right away without sending any messages. For this program to be valid, `p` must at least take one clock cycle to execute if `cond` is high.
- `p1 :>> p2`: (*sequential composition*) The program executes `p1`, waits for the time it takes to finish, and then executes `p2`.
- `p1 :|| p2`: (*parallel composition*) The program executes `p1` and `p2` in parallel, waiting for both to finish until it finishes.

Here is an example of a Flash program, where we describe a toggle:

```
toggleFlash change =
  While high
    ( While (inv change)
        ( Wait
        )
  :>> Emit ()
  :>> Wait
  :>> While (inv change)
        ( Emit ()
      :>> Wait
        )
  :>> Wait
    )
```

We can read the program as follows. Forever: wait until `change` is not low, then emit a message, and wait. Then, wait until `change` is not low, and emit a message all the time, then wait. The type of messages this Flash program is using, is `()`, because there is only one message.

We can give a more formal semantics to this language by giving a translation from a program to a circuit. And then we get an implementation for free!

We are going to define a function `circuit`, which takes a Flash program to a Flash circuit.

```
type FlashCircuit out
  = Signal Bool -> (FlashEmits out, Signal Bool)
```

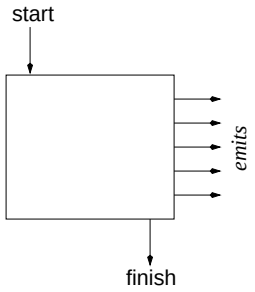


Figure 5.25: The shape of a circuit representing a Flash program.

```
type FlashEmits out
  = [(out, Signal Bool)]

circuit :: Flash out -> FlashCircuit out
```

A Flash circuit (see figure 5.25) takes in one input, called `start`, which is used to activate the program, and has two outputs, a list `emits`, and a signal `finished`, which the circuit uses to indicate that it is done. The list `emits` is a lookup table, which relates output messages and signals.

We start with `Skip`. Here, we just connect `start` to `finish`, so that we finish immediately.

```
circuit Skip start = ([], finish)
  where
    finish = start
```

In the case of `Emit`, we connect the `start` to the right output, and we finish immediately.

```
circuit (Emit out) start = (emits, finish)
  where
    emits  = [(out, start)]
    finish = start
```

When we execute a `Wait`, we connect `start` and `finish`, but with a delay, so that it takes one clock cycle to finish.

```
circuit Wait start = ([], finish)
  where
    finish = delay low start
```

To transform an `IfThenElse`, we first transform the two subprograms `prog1` and `prog2`. We start `prog1` if `start` is high and if the condition is true, and we start `prog2` if `start` is high and the condition is false. We collect all emitted messages, and finish if either one of them finishes.

```
circuit (IfThenElse cond (prog1, prog2)) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start1
    (emits2, finish2) = circuit prog2 start2

    start1 = and2 (start, cond)
    start2 = and2 (start, inv cond)

    emits  = emits1 ++ emits2
    finish = or2 (finish1, finish2)
```

To transform a `While`, we first transform the subprogram `prog`. Then, we introduce an auxiliary signal called `active`, which is high exactly when we should consider starting `prog`, that is when the whole while loop is started or when `prog` has finished. We actually start `prog` when we are active, and the condition is true. We finish the while loop when we are active but the condition is false.

```
circuit (While cond prog) start = (emits, finish)
  where
    (emits, finish') = circuit prog start'

    active = or2 (start, finish')
    start' = and2 (active, cond)
    finish = and2 (active, inv cond)
```

Transforming sequential composition just connects the `finish` of the first with the `start` of the second, and collects the emitted messages.

```
circuit (prog1 :>> prog2) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start
    (emits2, finish)  = circuit prog2 finish1

    emits = emits1 ++ emits2
```

And lastly, transforming parallel composition starts both circuits when started, collects the emitted messages, and *synchronizes* the finish signals for finishing. We use the `synchronize` circuit, which outputs a high if it has seen a high on both ints outputs.

```
circuit (prog1 :|| prog2) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start
    (emits2, finish2) = circuit prog2 start

    emits  = emits1 ++ emits2
    finish = synchronize (finish1, finish2)
```

Now we have made this translator, we can use it to turn a Flash program plus a list of output messages we are interested in into a circuit, outputting these messages.

```
compile :: Eq out => Flash out -> [out] -> [Signal Bool]
compile prog outputs = signals
  where
    start      = delay high low
    (emits, _) = circuit prog start

    signals =
      [ orl [ sig
            | (out',sig) <- emits
            , out == out'
            ]
      | out <- outputs
      ]
```

We first create a top-level `start` signal, which is to be `high` on the first clock tick, and then `low` forever, then filter out the signals we are interested in from the resulting circuit. Note that we have to take the or for these signals, since there might be several parts of the Flash program emitting the same signal.

Here is how we can create a toggle circuit from the given Flash program:

```
toggle' change = out
  where
    [out] = compile (toggleFlash change) [()]
```

We compile the Flash circuit, and say that we are only interested the `()` messages.

Chapter 6

Hardware Compilation

Various languages have been proposed to describe synchronous hardware at an abstract, yet syntesisable level. We propose a uniform framework within which such languages can be developed, and combined together for simulation, synthesis, and verification. We do this by embedding the languages in Lava — a hardware description language (HDL), itself embedded in the functional programming language Haskell. The approach allows us to easily experiment with new formal languages and language features, and also provides easy access to formal verification tools aiding program verification.

This chapter was written together with Gordon Pace, and submitted for publication, as an article titled 'An Embedded Language Framework for Hardware Compilation'.

6.1 Introduction

There are two essentially different ways of describing hardware. One way is *structural* description, where the designer indicates what components should be used and how they should be connected. Designing hardware at the structural level can be rather tedious and time consuming. Sometimes, one affords to exchange speed or size of a circuit for the ability to design a circuit by describing its behaviour at a higher level of abstraction which can then be automatically *compiled* down to structural hardware. This way of describing circuit is usually called a *synthesisable behavioural description*[1]. Behavioural descriptions are also often used to describe the specification of a circuit.

There exist a number of languages that one can use to structurally describe hardware. An example is the synchronous language Lustre [45, 47], which can be compiled into hardware structurally [99]. Languages that can be used for synthesisable behavioural description are for example Esterel [11] and Occam [87]. The popular industrial description languages VHDL and Verilog allow both kinds of descriptions.

In this paper, we will only deal with synchronous hardware, that is, all latches in a circuit listen to one omnipresent global clock. Moreover, at every clock cycle, if each input to a circuit is defined, each point in the circuit stabilises to exactly one voltage, low or high. However, we do not require that every feedback loop in the circuit contains a latch.

There are two main classes of synthesisable languages: ones where the description determines the timing behaviour (cycle by cycle) of the resultant circuit, and ones with no explicit timing control, and where the compilation only guarantees that the output at the end of the algorithm (or at designated points in the algorithm) matches that of the circuit. Languages with strict timing are necessary to describe circuits such as protocol implementations, and reactive systems, where the circuit continuously runs, sampling inputs, and behaving accordingly. In practice, some compilation schemata fall somewhere in between these two classes. In particular, commercial synthesis tools for Verilog and VHDL usually provide the user with the option of choosing how strictly the timing behaviour specified is adhered to. In the rest of the paper, we will be talking *exclusively* of strict timing compilation, but the approach is equally applicable to languages with loose timing.

6.1.1 Embedded Description Languages

Using a technique from the programming language community, called *embedded languages* [56], we present a framework to merge structural and behavioural

[1] These are to be distinguished from *behavioural descriptions* (as used in industrial HDLs such as Verilog and VHDL) which are used to describe the functionality of a circuit, but are do not necessarily have a hardware counterpart.

hardware descriptions. An embedded description language is realised by means of a library in an already existing programming language, called the *host language*. This library provides the syntax and semantics of the embedded language by exporting function names and implementations.

The basic embedded language we use is *Lava* [25]. Lava is a structural hardware description language embedded in the functional programming language Haskell [93]. Embedding a language is a powerful concept because descriptions in the embedded language are first-class objects in the host language. In the case of Lava, this means that hardware descriptions can be generated, analysed and transformed using a full-blown programming language.

The idea is now to build a layer on top of Lava, which embeds a synthesisable behavioural description language. In order to do this, we have to specify the syntax of the behavioural language, and the way it is compiled into a structural hardware description. It is possible to describe all this in the Lava framework: the syntax is described as a Haskell datatype, and the compilation process described as a Lava circuit description.

But why stop there? It is possible to embed several different behavioural description languages, each with their own features, advantages and disadvantages. In this way, we can describe a hardware system, using different languages for different parts, all within a single framework.

Examples of uses of embedding in this way are: behavioural in structural, where we use a behavioural language to describe some parts, and plug these parts together using a structural language; multiple behavioural in structural, the same, but having several different behavioural languages; structural in behavioural, so that we can describe a sub-procedure of the behavioural algorithm structurally; and even behavioural in behavioural, where we can describe sub-procedures for one behavioural language by using another behavioural language. All these examples are useful in describing circuits as well as their specifications.

Some of these examples are non-trivial to achieve, and we do not claim to have a generic solution to them. Our contribution proposes a common framework, in which one can quickly experiment with different approaches and new behavioural languages. The framework we propose, Lava, is powerful enough to use for describing new languages, giving semantics to them, implementing them, and combining them. In the context of developing behavioural description languages, it is very convenient to have circuit descriptions, analyses, transformations, and implementation and verification methods backed up by a full-blown programming language.

In section 2 we briefly introduce Lava and show how a simple high level language, that of regular expressions, can be embedded in Lava and how instances of this language can then be manipulated syntactically and compiled into circuits. In section 3 we illustrate how the embedded language approach extends easily to more complex languages by presenting a small, imperative style language, Flash. Section 4 then discusses more advanced issues: various ways of combining

different high level languages, verification of compiled programs and exploring potentially dangerous combinational loops.

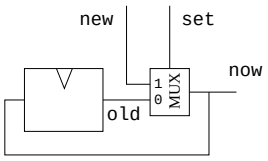
6.2 Embedding Hardware Description Languages

6.2.1 Circuit Descriptions in Lava

Circuit descriptions in Lava correspond to function definitions in Haskell. The Lava library provides primitive hardware components such as gates, multiplexers and delay components. We give a short introduction to Lava by example.

Here is an example of a description of a register. It contains a multiplexer, `mux`, and a delay component, `delay`. The delay component holds the state of the register and is initialised to `low`.

```
setRegister (set, new) = now
  where
    old = delay low now
    now = mux (set, (old, new))
```



Note that `setRegister` is declared as a circuit with two inputs and one output. Note also that definitions of outputs (`now`) and possible local wires (`old`) are given in the `where`-part of the declaration.

After we have made a circuit description, we can simulate the circuit in Lava as a normal Haskell function. We can also generate VHDL or EDIF describing the circuit. It is possible to apply circuit transformations such as retiming, and to perform circuit analyses such as performance and timing analysis. Lava is connected to a number of formal verification tools, so we can also automatically prove properties about the circuits.

6.2.2 Generic and Parametrized Circuit Definitions

We can use the one bit register to create an n-bit register array, by putting n registers together. In Lava, inputs which can be arbitrarily wide are represented by means of lists. A generic circuit, working for any number of inputs, can then be defined by recursion over the structure of this list.

```
setRegisterArray (set, [])       = []
setRegisterArray (set, new:news) = val:vals
  where
    val  = setRegister (set, new)
    vals = setRegisterArray (set, news)
```

Note how we use pattern matching to distinguish the cases when the list is empty (`[]`) and non-empty (`x:xs`, where `x` is the first element in the list, and `xs` the rest).

Circuit descriptions can also be parametrized. For example, to create a circuit with n delay components in series, we introduce n as a parameter to the description.

```
delayN 0 inp = inp
delayN n inp = out
  where
    inp' = delay low inp
    out  = delayN (n-1) inp'
```

Again, we use pattern matching and recursion to define the circuit. Note that the parameter `n` is *static*, meaning that it has to be known when we want to synthesise the circuit.

A parameter to a circuit does not have to be a number. For example, we can express circuit descriptions which take other circuits as parameters. We call these parametrized circuits *connection patterns*. Other examples of parameters include truth tables, decision trees and state machine descriptions. In this paper, we will talk about circuit descriptions which take behavioural hardware descriptions, or *programs*, as parameters.

6.2.3 Behavioural Descriptions as Objects

In order to parametrize the circuit definitions with behavioural descriptions, we have to embed a behavioural description language in Lava. We do this by declaring a Haskell datatype representing the syntax of the behavioural language. To illustrate the concepts with a small language, we will use regular expressions. The syntax of regular expressions is expressed as a Haskell datatype:

```
data RegExp = EmptyString
            | Input Sig
            | Star RegExp
            | RegExp :+: RegExp
            | RegExp :>: RegExp
```

The data objects belonging to this type are interpreted as regular expressions with, for example, $a(b+c)^*$ being expressed as:

```
Input a :>: Star (Input a :+: Input c)
```

Note that the variables `a`, `b` and `c` are of type `Sig` — they are *signals* provided by the programmer of the regular expression. They can either be outputs from

another existing circuit, or be taken as extra parameters to the definition of a particular regular expression. We interpret the signal `a` being high as the character '`a`' being present in the input.

Since regular expressions are now simply data objects, we can generate these expressions using Haskell programs. Thus, for example, we can define a `power` function for regular expressions:

```
power 0 e = EmptyString
power n e = e :>: power (n-1) e
```

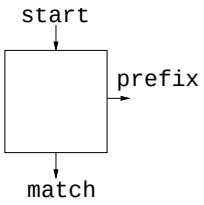
Similarly, regular expressions can be manipulated and modified. For example, a simple rewriting simplification can be defined as follows:

```
simplify (EmptyString :>: e) = simplify e
simplify (EmptyString :+: e)
  | containsEmpty e          = simplify e
  | otherwise                = EmptyString :+: simplify e
simplify (Star (Star e))     = simplify (Star e)
...
```

Another useful algorithm which can be expressed is the one presented in [98], which reduces (in linear time) a regular expression e to another one f such that the empty string does not occur in f and e^* is the same language as f^*. Thus, from now on, we assume that the body of a `Star` cannot produce the empty string.

6.2.4 Compiling Regular Expressions into Circuits

The circuits we generate for regular expressions have one input `start` and two outputs `match`, and `prefix`. When `start` is set to high, the circuit will start sampling the signals. The output `match` is then set to high when the resulting sequence of signals is included in the language represented by the expression. The output `prefix` remains true while the input is a valid prefix of the regular expression. Note that the circuit will get extra inputs, which correspond to the parsed symbols. They are part of the regular expression, by means of the `Input` construct.

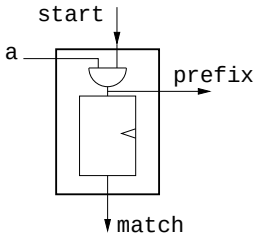


The type of the resulting circuit is thus:

```
type Circuit_RegExp = Sig -> (Sig, Sig)
```

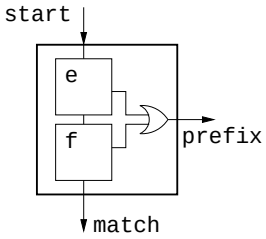
since the resulting circuit has one input and two outputs. We express the compilation process as a circuit definition parametrized by a regular expression:

```
regexp :: RegExp -> Circuit_RegExp
```



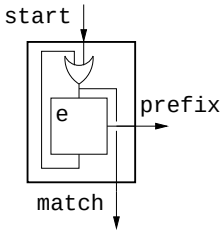
Signal input
The regular expression `Input a` is matched if, and only if the signal `a` is high when the circuit is started.

```
regexp (Input a) start = (prefix, match)
  where
    prefix = and2 (start, a)
    match  = delay low prefix
```



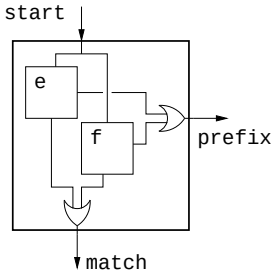
Sequential composition
The regular expression `e :>: f` must start accepting expression `e`, and upon matching it, start trying to match expression `f`.

```
regexp (rexp1 :>: rexp2) start =
                              (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match)  = regexp rexp2 match1
    prefix = or2 (prefix1, prefix2)
```



Loops
The circuit accepting regular expression `Star e` is very similar to that accepting `e`, but it is restarted every time the inputs match `e`.

```
regexp (Star rexp) start = (prefix, match)
  where
    (prefix, match') = regexp rexp match
    match = or2 (start, match')
```



Non-deterministic choice
The inputs match regular expression `e :+: f` exactly when they match expression `e` or `f`.

```
regexp (rexp1 :+: rexp2) start =
                              (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match2) = regexp rexp2 start
    prefix = or2 (prefix1, prefix2)
    match  = or2 (match1, match2)
```

A circuit resulting from such a compilation scheme is not necessarily efficient enough. Often, there are small optimisations we can make, such as: constant folding (when the input to a gate is always low or always high), sharing introduction (when we have identical gates with identical inputs), tree introduction

(changing a linear chain of associative gates into a balanced tree), and constant introduction (when a circuit point provably always has the same value). Sometimes, more rigorous optimisation methods are necessary; in this case we can use external circuit optimisation tools such as SIS [102].

6.3 Compiling Flash

In this section, we will show a slightly bigger example of a language, we will call *Flash*. It is quite a basic language, but it illustrates many of the issues one encounters when dealing with hardware compilation. As it is meant just an example, we deal quite informally with the semantics of Flash. More formal treatment of the semantics of similar languages can be found in [10, 87].

6.3.1 Flash Syntax

As before, we first declare a Haskell datatype that embeds the syntax of Flash.

```
data Flash = Skip
           | Delay
           | Shout
           | IfThenElse Sig (Flash, Flash)
           | While Sig Flash
           | Flash :>> Flash
           | Flash :|| Flash
```

Flash is a simple imperative programming language containing the usual statements like skip, sequential composition (`:>>`), if-then-else, and while. For simplicity, the language has no expressions. Instead, we can use Lava gates directly to create a signal representing the condition in both the if-then-else and the while loop.

To create some interesting output, we have added a `Shout` statement. This statement is in the spirit of the Esterel `emit` statement [11]. It makes a special output of the circuit, called `shout`, high whenever `Shout` is executed. Further, we also have parallel composition (`:||`), which has a fork-join semantics. Lastly, the delay statement is the only statement that takes time. When executed, it blocks the process until the next clock cycle. Note that `Shout` takes no time to execute.

For example, a Flash program to output a clock-like output alternating between high and low could be written as:

```
alternate = While (high) (Shout :>> Delay :>> Delay)
```

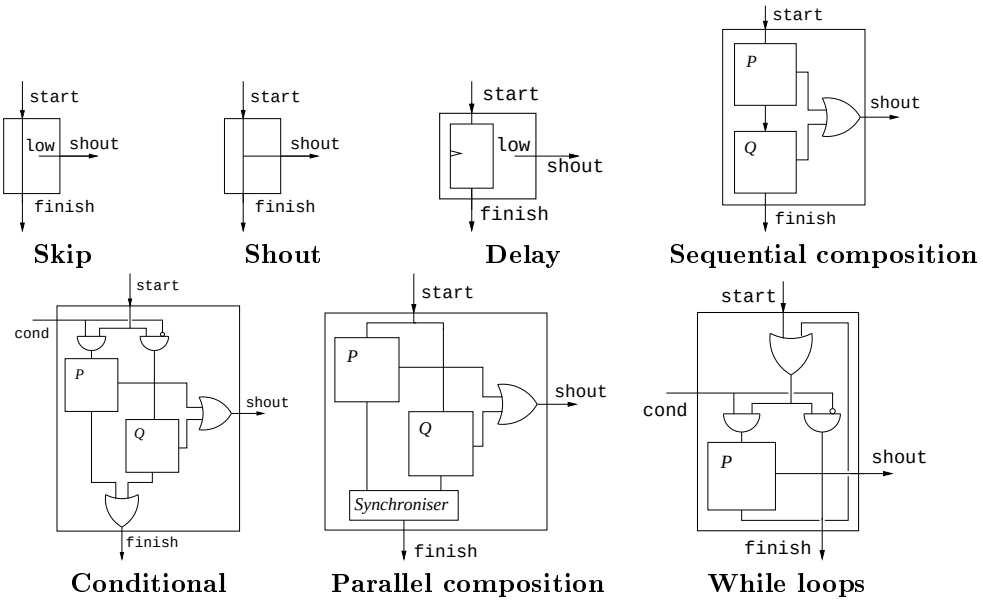
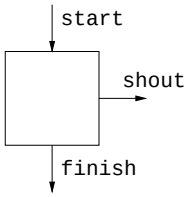


Figure 6.1: Compiling Flash

6.3.2 Compiling Flash



The circuits that we compile Flash programs into have one input, `start`, which is set to high to start the program. They will have two outputs: `shout`, which becomes high when the program shouts, and `finish`, which becomes high when the program is done.

In figure 6.1, we see the compilation schemata for the various language constructs of Flash. We show the Lava code for some of the constructs.

The case for the while loop looks as follows:

```
flash (While cond prog) start = (shout, finish)
  where
    (shout, finish') = flash prog start'
    restart = or2 (start, finish')
    start'  = and2 (restart, cond)
    finish  = and2 (restart, inv cond)
```

We might (re)start the body of the while loop, if the whole loop is started or if the body has just finished. In that case, depending on the condition, we restart the body or we finish. Note that we have created a loop since `finish'` depends

on `start'` depends on `restart` depends on `finish'`. In fact, this loop might be a combinational loop — we say more about this in section 6.4.4.

Here is how we translate parallel composition:

```
flash (prog1 :|| prog2) start = (shout, finish)
  where
    (shout1, finish1) = flash prog1 start
    (shout2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    finish = synchroniser (finish1, finish2)
```

We start both processes as soon as the parallel composition is started. We shout when one of the processes shouts. But when do we finish? We use a little circuit, called *synchroniser*, which keeps track of both processes, and generates a high on the finish signal exactly when both processes have finished.

```
synchroniser (finish1, finish2) = finish
  where
    both   = and2 (finish1, finish2)
    one    = xor2 (finish1, finish2)
    wait   = delay low (xor2 (one, wait))
    finish = or2 (both, and2 (wait, one))
```

The wire `both` is high when both processes are finishing at the same time. The wire `one` is high when exactly one process is finishing. The wire `wait` is high when one process has finished but not the other.

6.4 Advantages of Embedding

In this section, we discuss some of the advantages of embedding behavioral languages in a general hardware description framework like Lava.

6.4.1 Combining Languages

The choice of the right language to solve a problem is crucial both to simplify the algorithm, and to generate more efficient circuits. For example, regular expressions can be very useful to generate circuits which validate their input, but, since they have no outputting mechanism, it becomes very difficult (or impossible) to perform calculations and output their results.

Consider the problem of designing a circuit that accepts input sequences that behave like a clock with half-period n. This circuit might be useful for monitoring real input, or when expressing properties for later formal verification. It is easy to write a generic regular expression with the specified behaviour:

```
acceptClock n c = Star (  power n (Input c)
                       :>: power n (Input (inv c))
                        )
```

Now consider using a regular expression to design a circuit that monitors two inputs, accepting them only if they behave like clocks with half-periods n and m. The size of the smallest regular expression capable of doing this has a size of the order of magnitude of the least common denominator of n and m, which is too big in practice.

There are two solutions. One is to design a new language, in which it is easy to describe circuits as the one mentioned above. In fact, it would suffice to add *conjunction* as a regular expression operator, which is rather easy to do. The other is to combine the solutions to the two subproblems (recognising each clock) at the structural level using Lava:

```
acceptTwoClocks n m (c1,c2) = ok
  where
    (ok1,_) = regexp (acceptClock n c1) start
    (ok2,_) = regexp (acceptClock m c2) start
    start   = delay high low
    ok      = and2 (ok1, ok2)
```

Obviously, the used subprograms need not be in the same language. For example, if we want to run a Flash program `prg` only to abort it as soon as the input does not match a regular expression `rexp`, we can use the following parameterised circuit:

```
abort rexp prg start = (shout', finish)
  where
    (shout, finish) = flash  prg  start
    (prefix, _)     = regexp rexp start
    shout'          = and2 (shout, prefix)
```

6.4.2 Nesting Languages

A problem with the approach mentioned above is that we deal with the input and output of the produced circuits at a rather low-level. This is quite error-prone, and it becomes difficult to change the shape of the produced circuits.

A cleaner approach is not to express the combination of programs at the structural level, but at the behavioural one. Thus, for example, one could allow adding Flash subprograms to regular expressions by augmenting the syntax of regular expressions by:

```
data RegExp = ...   | ImportFlash Flash
```

Consider the problem of generating a circuit which recognises the input of `a`, `b` and `c` in any order. If this is required in a sub-expression of a regular expression, the result of expanding the expression can lead to a blow up in circuit size. However, a Flash program for this is rather simple to write:

```
wait s        = While (inv s) Delay
perm3 (a,b,c) = (wait a :|| wait b :|| wait c) :>> Delay
```

If this is required within a regular expression, one can easily use it as for example:

```
Star ( ImportFlash (perm3 (a,b,c))
   :+: ImportFlash (perm3 (d,e,f))
     )
```

Fiddling with the interfaces to make them match is thus done only once when the compilation of a regular expression of the form `ImportFlash p` is defined. However, this approach still has the undesirable effect that for every new language one uses, the compilers for all other languages need to be modified to be able to import programs from the new languages into the old ones.

A more extendable approach would be to add one `Import` construct for each language:

```
data RegExp = ...   | ImportRegExp Circuit_RegExp
data Flash  = ...   | ImportFlash  Circuit_Flash
```

Now, in order to import Flash programs in regular expressions, all we have to provide is a parameterised circuit `flash_regexp`, which converts from one format to the other.

```
flash_regexp flashc start = (prefix, match)
  where
    (shout, finish) = flashc start
    prefix = shout
    match  = finish
```

Needless to say, there are other ways in which a Flash circuit can be transformed into one which can be used by regular expressions. For example, one can generate (or calculate) an `active` wire from Flash circuits which corresponds to the regular expression `prefix` wire. In defining these 'conversion' circuits, we have to be careful here not to invalidate the invariants that the languages involved assume and obey. The technique mentioned in the next section can be used to help with this. 'Calling' another language now simply becomes a matter of using the `Import` construct and the right conversion circuits.

6.4.3 Error Wires

Often, something can go wrong during the execution of a program. What exactly can go wrong depends on the semantics of the language. A standard example in a language with arithmetic expressions is division by zero. It is not clear what the corresponding compiled circuit would do in that case, since we do not want the circuit simply to 'abort'.

In a language with parallel composition, things can go wrong due to parts of the circuit requiring single access: two processes trying to send a message on the same channel at the same time, two processes updating a shared variable at the same time, etc. If the semantics of the language disallows these situations, then we should make sure that the programs we compile to hardware are well-behaved.

The solution we propose is to have an extra output to the circuit which goes high as soon as something goes wrong with the program execution — an *error* wire. This wire and the logic generating it will not appear in the final implementation of the circuit, but will be used to *verify* (by means of model checking methods) that the program in question is error-free.

Consider a change to the semantics of Flash, requiring that only one process can shout at the same time. We would like to be warned at compile time if a program violates that property. Thus, we add an error wire to the output of Flash circuits, and adapt the compilation scheme accordingly. Here is the interesting case, parallel composition:

```
flash (prog1 :|| prog2) start = (shout, error, finish)
  where
    (shout1, error1, finish1) = flash prog1 start
    (shout2, error2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    both   = and2 (shout1, shout2)
    error  = orl [error1, both, error2]
    finish = synchroniser (finish1, finish2)
```

There is an error in parallel composition of two programs if there was an error in (at least) one of the processes, or if both processes shout at the same time. We can now declare a *property*, a circuit which outputs are always high if and only if a certain property holds.

```
prop_FlashProgramOk prog start = inv error
  where
    (_, error, _) = flash prog start
```

The output `ok` is high if and only if there is no error in the program `prog`. We can check this property using the Lava command `verify`.

```
Lava> verify (prop_FlashProgramOk
                          (alternate :|| (Delay :>> alternate))
Verify: ... Valid.

Lava> verify (prop_FlashProgramOk (Shout :>> Delay :|| Shout))
Verify: ... Falsifiable.
<high>
```

The error wire technique can also be used to find bugs in the compilation scheme itself. Many languages have certain invariants that hold for every program. By raising the error wire when the invariant is violated, and verifying the absence of this error for random programs (by using a technique similar to the one developed in [23]), we can find bugs or increase our confidence in the compilation scheme.

6.4.4 Combinational Loops

Looking at the compilation scheme for the while construct, we can see that it is possible to introduce combinational loops: cycles in the circuit without a delay component.

The usual solution in this case is to require that body of the while loop takes time — the execution path goes through at least one `Delay` statement. But even with this restriction, the resulting circuit might still contain combinational loops. However, these combinational loops are not *bad*, in the sense that the actual circuit never produces undefined outputs. In this case, the combinational loops are called *constructive* [112].

Even when all combinational loops in a given circuit are constructive, most of the external formal verification tools that Lava is connected to, are not able to deal with these loops. Fortunately, the method of *temporal induction* [106] can naturally verify properties of cyclic circuit definitions. However, the method is only sound if all loops in the circuit are constructive loops.

Thus, before we implement or formally verify actual circuits containing possible bad loops, we have to prove that all loops all constructive. Lava provides a circuit analysis, called `constructive`, which does exactly that [22]. Here is how we can use it:

```
Lava> verify (constructive (flash (While high Delay)))
Verify: ... Valid.

Lava> verify (constructive (flash (While high Skip)))
Verify: ... Falsifiable.
<high>
```

What about parallel composition? When is it acceptable for a body of a while

loop to contain a parallel composition? Take, for example, the following Flash program:

```
possibleProblem inp = While high
                        ( IfThenElse inp (Skip , Delay)
                      :|| Delay
                        )
```

In principle, we should be able to execute this, since for all programs `p`, the program `p :|| Delay` takes time to execute. Let us analyse the resulting circuit:

```
possibleProblemCirc inp = flash (possibleProblem inp)

Lava> verify (constructive possibleProblemCirc)
Verify: ... Falsifiable.
<low, high>
<high, low>
```

This shows that the simple compilation scheme we have used to illustrate our examples is not sufficiently robust to handle this example. Obviously, one can require that both sides of a parallel composition should take time (when appearing immediately inside the body of a while loop). However, this is a stringent and rather unsatisfactory restriction. A better solution would be to use a more complex compilation of loops, as used, for example, in [10].

6.5 Conclusions

6.5.1 Related Work

Hardware compilation of high-level languages has been around for quite a while. The approach has been considered potentially practical mainly since the introduction of programmable circuits. The compilation for various languages have since appeared in the literature, see e.g. [77, 87, 86, 10]. An introductory overview of the methodology appears in [124].

It is widely recognised that different styles of synchronous languages lend themselves more easily to different applications. In [72, 73], Maraninchi and Rémond present Mode-Automata — a combination of state diagram based descriptions (based on Argos [71]) with the dataflow language Lustre [45]. The semantics of the resulting language are defined by a translation into plain Lustre. The approach is thus very similar to the one we use, except that they use external programs to read mode-automata and translate them into Lustre. The embedded language approach we use, allows us to translate and reason about the new language at the same level as our base HDL Lava. This allows a much more versatile approach to language combination.

Poigné and Holenderski [96] present a theoretical framework for combining synchronous languages by using synchronous automata as the common semantic level. These ideas have been implemented in the SYNCHRONY WORKBENCH where programs written in one of a number of languages (Esterel, Lustre, Argos, and Synchronous Eifel) can be combined together. The main difference between their work and that presented in this paper, is that we embed the languages we use, and our intermediate language, Lava, is itself an embedded language. This gives us certain advantages: it is easier to add new languages to the framework, and language combination can be easily adapted depending on the requirements.

6.5.2 Discussion

We have presented a uniform framework in which it is easy to implement and hence experiment with synthesisable behavioural languages. By embedding these languages in Lava, we are able to define their compilation in a natural and easy way and, at the same time, benefit from the verification tools connected to Lava to improve compilation and verify programs.

Using this approach, we have shown that we can formally reason about programs at a number of levels. First, taking advantage of the fact that our programs are just data objects in Haskell, we can apply syntactic reasoning by defining functions which modify the program. Second, using the verification tools linked to Lava, we can define observers (in one of the languages) to verify properties of hardware described using either structural Lava, or some other language. Third, the compilation process itself can make use of the verification tools to check dynamic properties which may be needed to guarantee correct compilation.

Within our framework, we have implemented various languages or subsets of them, such as Esterel, Handel and Occam, fragments of process calculi such as CSP and CBS, and some restricted temporal logics. We have also embedded state machine descriptions and specification languages in the same framework. This included different control and data features including updatable variables, buffered and unbuffered channels, exceptions and broadcast communication. Describing the compilation of a language is rather straightforward, and in fact, we have successfully used this framework in the teaching of a graduate course on hardware description languages. The definition of the compilation function for a language is usually not much different from a denotational semantics of the language in terms of a dataflow network.

One of the important issues that we have not discussed in this paper is the question of the correctness of the compilation procedure. A number of approaches have been proposed [53, 101, 10] which are applicable to our compilation scheme. We are currently exploring how such proofs can also be presented uniformly within our framework. Preliminary work is encouraging and it is not difficult to prove that, for instance, the compilation of regular expressions presented in this paper satisfies regular expression equational axioms.

Chapter 7

Temporal Induction

Binary Decision Diagrams (BDDs) have dominated the area of symbolic model checking for the past decade. Recently, the use of satisfiability (SAT) solvers has emerged as an interesting complement to BDDs. SAT-based methods are capable of coping with some of the systems that BDDs are unable to handle.

The most challenging problem that has to be solved in order to adapt standard symbolic model checking to SAT-solvers is the boolean quantification necessary for traversing the state space. A possible approach to extending the applicability of SAT-based model checkers is therefore to reduce the amount of traversal.

In this paper, we investigate a BDD-based verification algorithm due to van Eijk. Van Eijk's algorithm tries to compute information that is sufficient to prove a given safety property directly. When this is not possible, the computed information can be used to reduce the amount of traversal needed by standard model checking algorithms. We convert van Eijk's algorithm to use a SAT-solver instead of BDDs. We also make a number of improvements to the original algorithm, such as combining it with recently developed variants of induction. The result is a collection of substantially strengthened and complete verification methods that do not require state space traversal.

This chapter was written together with Per Bjesse, and published at the conference on Formal Methods in Computer Aided Design 2000, as an article titled 'SAT-based Verification without State Space Traversal' [16].

7.1 Introduction

Symbolic model checking based on satisfiability (SAT) solvers [13, 1, 123, 106] has recently emerged as an interesting complement to model checking with Binary Decision Diagrams (BDDs) [20]. There are a number of systems which are not suited to be effectively verified using BDD-based model checkers, but can be verified using SAT-based methods. The use of SAT-solvers rather than BDDs also has advantages such as freeing the user from providing good variable orderings, and making the number of variables in the system less of a bottleneck. However, the boolean quantification that is necessary for computing characterisations for sets of predecessors (and successors) of states can sometimes lead to excessively large formulas in SAT adaptions of standard model checking algorithms.

In the hope of alleviating these problems, we investigate a BDD-based algorithm due to van Eijk [35] that attempts to verify safety properties of circuits without performing state-space traversal. The main idea behind the algorithm is to use induction to cheaply compute points in the circuit that always have the same value (or always have opposite values) in the reachable state space. This information sometimes directly implies the safety properties. If such a direct proof is not possible, the computed information can be used to decrease the number of necessary fixpoint iterations in backwards reachability algorithms. Van Eijk [35] has used the algorithm to directly prove equivalence between the original circuits and synthesised and optimised versions of 24 of the 26 circuits in the ISCAS'89 benchmark suite.

We are specifically interested in using van Eijk's algorithm to prove safety properties of circuits that are hard to represent using BDDs. Also, when a direct proof is not possible, we want to use the computed information to reduce the amount of state space traversal in exact SAT-based model checking methods as this could decrease the amount of necessary quantification drastically. As a consequence, we want to find alternatives to the use of BDDs in the original analysis. Van Eijk's algorithm also has the drawback of always computing the *largest* possible set of equivalences, even when this is not needed for the verification of the particular safety property at hand. In some cases this can become too costly; we would therefore like to be able to control how much work we put into finding equivalences.

We solve the two problems by converting the algorithm to use propositional formulas to represent points in the circuit, and by applying Stålmarck's saturation algorithm [115, 107] rather than BDDs for discovering equivalences between points.

The resulting algorithm is generalised in three ways. First, we make the algorithm complete by changing the induction scheme that is used in the method to some recently developed stronger variants of induction [106]. Second, we modify the algorithm to also discover implications between points in the circuit. Third, we demonstrate that van Eijk's algorithm can be viewed as an approxi-

mate forwards reachability analysis, and use this insight to construct the dual approximate backwards reachability algorithm and a mutual improvement algorithm.

The information that is computed by the resulting algorithms can in principle be used together with any BDD- or SAT-based model checking method. We show some benchmarks that demonstrate that the methods on their own can be very powerful tools for checking safety properties. For example, we use the algorithms to verify a non-trivial industrial example that previously has been out of reach for the SAT-based model checker FixIt [1].

7.2 Van Eijk's method: finding equivalences

In this section, we describe van Eijk's method [35]. In the original paper it is presented as a method for equivalence checking of sequential synchronous circuits. However, while using the method we have observed that it can work well also for general safety property verification.

Basic idea. The idea behind van Eijk's algorithm is to find the points in the circuit which have the same value (or have opposite values) in all reachable states. This information can then be used to either directly prove the safety property or to strengthen other verification methods.

The information is represented as an equivalence relation over the points of the circuit and their negations. The algorithm computes such an equivalence relation by means of a fixed point iteration. It starts with the equivalence relation that necessarily holds between the points in the initial state. Then it improves the relation by assuming that the equivalences hold at one time instance and deriving the subset of these equivalences that must hold in the next time instance. After a number of consecutive improvements, a fixed point is reached. The resulting equivalence relation is satisfied by the initial states, and is moreover preserved by any circuit transition. Therefore, it must hold in all reachable states.

Before we give a more precise description of van Eijk's algorithm, we first introduce some definitions.

Formulas. We describe the systems we are dealing with using propositional logic formulas. These are syntactic objects, built from variables like x and y, boolean values 1 and 0, and connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$. We say that a formula is *valid* if and only if it evaluates to 1 for all variable assignments under the usual interpretation of the connectives.

State machines. We represent sequential synchronous circuits as state machines in the standard way [26], where the set of states is the set of boolean valuations of a vector s of variables; one variable for each input and internal latch. As we do not restrict the input part of the states, these state machines

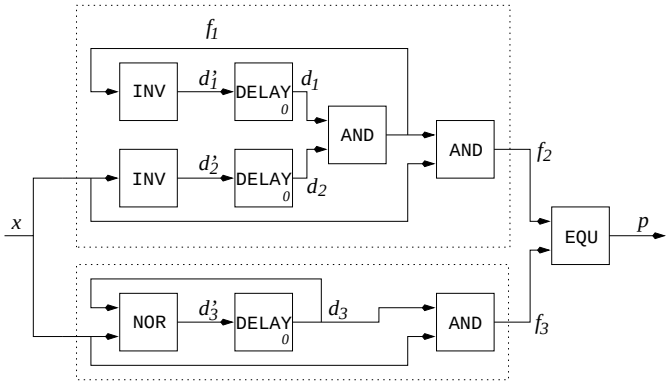


Figure 7.1: An example circuit

are non-deterministic. The standard representation also guarantees that every state has at least one outgoing transition.

We characterise the set of initial states and the transition relation of the state machine by the propositional logic formulas $\mathsf{Init}(s)$ and $\mathsf{Trans}(s, s')$, respectively. In other words, $\mathsf{Init}(s)$ is satisfied exactly by the initial states, and $\mathsf{Trans}(s, s')$ is satisfied precisely when there is a transition between the states s and s'. The safety property of the system we want to verify is represented by the formula $\mathsf{Prop}(s)$.

Example 7.2.1 *Assume that we want to decide whether the two subcircuits in Figure 7.1 are equivalent. This amounts to checking whether the signal p is always true. Let us construct the necessary formulas. There are four state variables—one for every input and one for every delay component—so* $s = (x, d_1, d_2, d_3)$. *Since the delay components have an initial value of 0, the formula for the initial states becomes:*

$$\mathsf{Init}(x, d_1, d_2, d_3) \;=\; \neg d_1 \wedge \neg d_2 \wedge \neg d_3$$

Looking at the logic contained in the circuit, we can write down the formula for the transition relation:

$$\begin{aligned}\mathsf{Trans}(x, d_1, d_2, d_3, x', d_1', d_2', d_3') \;=\;& \\ (d_1' \Leftrightarrow \neg(d_1 \wedge d_2)) \;\wedge\; (d_2' \Leftrightarrow \neg x) \;\wedge\; (d_3' \Leftrightarrow \neg(x \vee d_3))&\end{aligned}$$

Lastly, we define the formula for the property p:

$$\mathsf{Prop}(x, d_1, d_2, d_3) \;=\; (d_1 \wedge d_2 \wedge x) \;\Leftrightarrow\; (d_3 \wedge x)$$

Signals. Given the formulas that characterise a state machine, we define the set Signals that models the points in the circuit. The elements of Signals are

functions taking state variable vectors to formulas instantiated with these variables. Specifically, for every subformula $f(s)$ of the system formulas $\mathsf{Trans}(s, s')$ and $\mathsf{Prop}(s)$, such that $f(s)$ is not dependent on any of the variables of s', we add the corresponding functions f and $\neg f$ to the set. Moreover, we also add the constant signals tt and ff, for which $\mathsf{tt}(s) = 1$ and $\mathsf{ff}(s) = 0$.

Example 7.2.2 *f_1 is a signal in Figure 7.1 with the definition $f_1(x, d_1, d_2, d_3) = d_1 \wedge d_2$. The negated signal $\neg f$ has the definition $\neg f_1(x, d_1, d_2, d_3) = \neg(d_1 \wedge d_2)$.*

Signal correspondence. For a given equivalence relation $\equiv$ over the set $\mathsf{Signals}$, we define the *signal correspondence condition*, denoted by $\mathsf{Holds}(\equiv, s)$, as follows:

$$\mathsf{Holds}(\equiv, s) \;=\; \bigwedge_{f \equiv g} f(s) \Leftrightarrow g(s).$$

This means that the correspondence condition for an equivalence relation is satisfied by a state when all the signals that are equivalent have the same value in that state. We define a *signal correspondence relation* as an equivalence relation whose correspondence condition holds in all reachable states.[1]

Algorithm. In order to find a signal correspondence relation, van Eijk's algorithm computes a sequence of equivalence relations $\equiv_i$, each being a better overapproximation of the desired relation. The sequence stops when an n is found such that $\equiv_n$ is equal to $\equiv_{n+1}$.

The first approximation $\equiv_0$ is the equivalence relation that holds in the initial states. We can define it as follows; for all f and g, $f \equiv_0 g$ if and only if the following formula is valid:

$$\mathsf{Init}(s_1) \quad \Rightarrow \quad f(s_1) \Leftrightarrow g(s_1).$$

This means that two signals are equivalent precisely if they must have the same value in all initial states. The original algorithm computes $\equiv_0$ by constructing a BDD for every signal, and pairwise comparing these BDDs under the assumption that the BDD for $\mathsf{Init}(s)$ holds.

The other approximations $\equiv_{n+1}$ for $n \geq 0$ are subsets of $\equiv_n$. We can define them as follows; for all f and g, $f \equiv_{n+1} g$ if and only if $f \equiv_n g$ and the following formula is valid:

$$\mathsf{Holds}(\equiv_n, s_1) \wedge \mathsf{Trans}(s_1, s_2) \quad \Rightarrow \quad f(s_2) \Leftrightarrow g(s_2).$$

This means that two signals are equivalent in the new relation, when (1) they were equivalent in the old relation and (2) they have the same value in the next state if the old relation holds in the current state. The original algorithm computes $\equiv_{n+1}$ by pairwise comparison of the BDDs for the signals related by $\equiv_n$ under the assumption that the BDD for $\equiv_{n+1}$ holds.

[1] Note that this is a slight generalisation of van Eijk's original definition [35].

```
1.  ≡1, ≡2 := ∅, ∅ ;
2.  -- compute first approximation
3.  for every f, g in Signals do
4.      form := Init(s1) ⇒ (f(s1) ⇔ g(s1)) ;
5.      if (validBdd(form)) then
6.          set f ≡2 g ;
7.  -- iterate until a fixed point is reached
8.  while (≡1 ≠ ≡2) do
9.      ≡1, ≡2 := ≡2, ∅ ;
10.     for every f, g in Signals such that f ≡1 g do
11.         form := Holds(≡1, s1) ∧ Trans(s1, s2) ⇒ (f(s2) ⇔ g(s2)) ;
12.         if (validBdd(form)) then
13.             set f ≡2 g ;
14. return ≡1 ;
```

Figure 7.2: Van Eijk's algorithm

The construction of approximations $\equiv_i$ has the shape of an *inductive* argument; it has a base case and a step that is iterated until it is provable. The final signal correspondence relation therefore holds in all reachable states.

For a schematic overview of the algorithm, see Figure 7.2. At lines 5 and 12, we use the function validBdd that checks if a formula is valid by building its BDD. At lines 6 and 13, we use **set** to modify an equivalence relation by merging the equivalence classes for f and g.

> *Example 7.2.1 (ctd.).* The signal correspondence relation found by the algorithm for Example 7.2.1 looks as follows:
>
> $$\{\ldots, (f_1, d_3), (f_2, f_3), (p, \mathsf{tt}), \ldots\}$$
>
> From this information it follows immediately that the property p is always true.

Remarks. The signal correspondence relation found by the algorithm sometimes implies the safety property directly. If this is not the case, then we can strengthen the transition formula $\mathsf{Trans}(s, s')$ to a new formula $\mathsf{Trans}(s, s') \wedge \mathsf{Holds}(\equiv, s) \wedge \mathsf{Holds}(\equiv, s')$. This is legal as we only are interested in transitions in the reachable state space. The new transition formula relates fewer states, and can consequently reduce the number of fixpoint iterations in conventional model checking methods.

Van Eijk's original paper presents a number of improvements of the basic method, such as *retiming* techniques that enlarge the set Signals so that the equivalence relation can contain more information, and *random simulation* that aims to reduce the number of pairwise comparisons by computing a better initial approximation $\equiv_0$. We will not discuss these techniques here, but refer to the original paper [35].

7.3 Stålmarck's method instead of BDDs

Van Eijk's method has a number of disadvantages. First of all, sometimes it is impossible to complete the analysis as some signals in the circuit can not be represented succinctly as BDDs. Second, the algorithm always finds the largest equivalence relation, which can be unnecessarily costly for proving the property. Third, the equivalences are computed by pairwise comparisons of signals, which means we have to build a quadratic number of BDDs. We will now focus on trying to solve these problems by using a SAT method instead of BDDs.

Stålmarck's method. Stålmarck's *saturation method* [115, 107] is a patented algorithm that is used for satisfiability checking. The method has been successfully applied in an wide range of industrial formal verification applications. The algorithm takes a set of formulas $\{p_1, \ldots, p_n\}$ as input, and produces an equivalence relation over the negated and unnegated subformulas of all p_i. Two subformulas are equivalent according to the resulting relation only when this is a logical consequence of assuming that all formulas p_i are true. The algorithm computes the relation by carefully propagating information according to the structure of the formulas.

The saturation algorithm is parameterised by a natural number k, the *saturation level*, which controls the complexity of the propagation procedure. The worst-case time complexity of the algorithm is $O(n^{2k+1})$ in the size n of the formulas, so that for a given k, the algorithm runs in polynomial time and space. For any specific k, there are formulas for which not all possible equivalences are found, but for every formula there is a k such that the algorithm finds all equivalences. A fortunate property is that this k is surprisingly low (usually 1 or 2) for many practical applications, even for extremely large formulas.

The advantage of having control over the saturation level is that the user can make a trade-off between the running time and the amount of information that is found. A disadvantage is that it is not always clear what k to choose in order to find enough information. In contrast, finding equivalences using BDDs results in discovering either all information, or no information at all due to excessive time and space usage.

Modification of van Eijk's method. We now adapt van Eijk's algorithm to use Stålmarck's method.

To compute the initial approximation $\equiv_0$, we use the saturation procedure to compute equivalence information between positive and negative subformulas of $\mathsf{Init}(s_1)$ and $\mathsf{Holds}(\mathsf{Id}, s_1)$ under the assumption that both of the these formulas are true. Here, Id denotes the identity equivalence relation on signals, relating f to g if and only if $f = g$. Note that $\mathsf{Holds}(\mathsf{Id}, s_1)$ is a valid formula, so assuming that it is true adds no real information; we just add it to the system to ensure that all subformulas that correspond to signals are present in the resulting equivalence relation. We then use the resulting information to generate the equivalence relation $\equiv_0$ on signals.

```
1.  ≡1      := ∅ ;
2.  -- compute first approximation
3.  system  := {Init(s1), Holds(Id, s1)} ;
4.  ≡2      := STÅLMARCK(system) / s1 ;
5.  -- iterate until a fixed point is reached
6.  while (≡1≠≡2) do
7.      ≡1      := ≡2 ;
8.      system  := {Holds(≡1, s1), Trans(s1, s2), Holds(Id, s2)} ;
9.      ≡       := STÅLMARCK(system) ;
10.     ≡2      := ≡1 ∩ (≡ /s2) ;
11. return ≡1 ;
```

Figure 7.3: Van Eijk's algorithm using Stålmarck's method

To improve a relation $\equiv_n$, we run the saturation procedure on a set of formulas that contains $\mathsf{Holds}(\equiv_1, s_1)$, $\mathsf{Trans}(s_1, s_2)$, and $\mathsf{Holds}(\mathsf{Id}, s_2)$. Again, we need the formula $\mathsf{Holds}(\mathsf{Id}, s_2)$ to ensure that all subformulas that correspond to signals are present. From the result we extract $\equiv_{n+1}$ by looking at the equivalences between subformulas depending on s_2, and taking the intersection with the original equivalence relation $\equiv_n$. The intersection of two equivalence relations relates two signals if both original relations relate them.

For a schematic overview of our algorithm, see Figure 7.3. The notation $\equiv /s_1$, occuring at lines 4 and 10, turns an equivalence relation $\equiv$ on formulas into an equivalence relation on signals, by relating two signals f and g if and only if their instantiated formulas $f(s_1)$ and $g(s_1)$ are related by $\equiv$.

In our modified algorithm, we have explicit control over the running time complexity of each iteration step; each step is guaranteed to take polynomial time and space. As a consequence, we do not have to worry about a possible exponential space blowup, as in the case of building BDDs. However, having this explicit control also means that we do not always compute the *largest* relation, since the saturation algorithm is possibly incomplete depending on what k we have chosen. In many cases it turns out that even for small k, the equivalence relation we compute using Stålmarck's algorithm is still large enough to decide if the property is true or not, or to considerably reduce the number of subsequent model checking iterations.

Signal implications. Finding equivalences between signals is a rather arbitrary choice. We could just as well try to find other information about signals that is easy to compute. For example, we can compute *implications* between signals.

An implication $f \Rightarrow g$ occurs between two signals f and g, if g must be true whenever f is true. The implications over the set of signals are interesting as they can capture *all* binary relations. The reason for this is that any formula that contains two variables can be characterised by a finite number of implications between these variables, their negations, and constants.

The presented algorithm can easily be extended to find implications between the computed equivalence classes of signals. Note that implications *within* equivalence classes do not give any information, since we know that every point in an equivalence class implies every other point in the same class. Our approach for generating the implications is simple: To begin with, we generate all the equivalence classes that hold over the reachable state space using the algorithm in Figure 7.3. From each equivalence class we take a representative signal. Finally, we run a modified version of the algorithm in Figure 7.3, that uses induction to find implications rather than equivalences between the representatives.

7.4 Induction

In order to further improve our adaptions of van Eijk's method, we start by investigating another safety property verification method: *induction* [106].

Simple induction. The idea behind simple induction is to attempt to decide whether all reachable states of the described system make the formula $\mathsf{Prop}(s)$ true by proving that the property holds at the initial states, and proving that if it holds in a certain state, it also holds in the next state. The inductive proof is expressible in propositional logic using the following two formulas:

$$\begin{aligned} \mathsf{Init}(s_1) &\Rightarrow \mathsf{Prop}(s_1) \\ \mathsf{Prop}(s_1) \wedge \mathsf{Trans}(s_1, s_2) &\Rightarrow \mathsf{Prop}(s_2) \end{aligned}$$

If the first formula is valid, we know that all the initial states of the system make the property true. If the second formula is valid, we know that any time a state makes the property true, all states reachable in one step from that state also make the property true. We can thus infer that all reachable states are safe.

Simple induction is not a complete proof technique for safety properties; it is easy to construct a system whose reachable states all make $\mathsf{Prop}(s)$ true, but for which the inductive proof fails. Just take a safe system and change it by adding two unreachable states s_1 and s_2 in such a way that there is a transition between s_1 and s_2, the formula $\mathsf{Prop}(s_1)$ is true, and $\mathsf{Prop}(s_2)$ is false. This system can not give a provable induction step even though all reachable states satisfy the property. A stronger proof scheme is needed for completeness.

Induction with depth. In the step case of simple induction we prove that the property holds in the current state, assuming that it holds in the previous state. One way to strengthen the induction step is to instead assume that the property holds in the previous n consecutive states. Correspondingly, the base case becomes more demanding.

Let $\mathsf{Trans}^*(s_1, \ldots, s_n)$ be the formula that expresses that $s_1, \ldots, s_n$ are states on a path $s_1, \ldots, s_n$, and let $\mathsf{Prop}^*(s_1, \ldots, s_n)$ be the formula that expresses that Prop is true in all of the states $s_1, \ldots, s_n$. *Induction with depth* n amounts

to proving that the following formulas are valid:

$$\begin{aligned} \mathsf{Init}(s_1) \wedge \mathsf{Trans}^*(s_1, \ldots, s_n) &\Rightarrow \mathsf{Prop}^*(s_1, \ldots, s_n) \\ \mathsf{Prop}^*(s_1, \ldots, s_n) \wedge \mathsf{Trans}^*(s_1, \ldots, s_{n+1}) &\Rightarrow \mathsf{Prop}(s_{n+1}) \end{aligned}$$

The modified base case expresses that all states on a path with n states starting in the initial states make the property true. The step says that if $s_1 \ldots s_{n+1}$ is a path where $s_1 \ldots s_n$ all make Prop true, then s_{n+1} also makes Prop true. We henceforth refer to n as the *induction depth*. Note that induction with depth 1 is just simple induction.

Unique states induction. The induction scheme with depth discovers paths to any state s in the reachable state space that makes $\mathsf{Prop}(s)$ false: A path with n states starting from the initial states and ending in a bad state is a counterexample to base cases of depth n or larger. As we are verifying finite systems, some depth n is therefore sufficient to discover all bugs.

Unfortunately the scheme is still not complete; it is possible to construct a safe system where the induction step fails for any depth n. Just take any safe system and change it by adding two unreachable states s_1 and s_2, so that the property holds in s_1, s_1 can both reach itself and s_2, and the property fails in s_2. Then there exist a counterexample for every depth n that loops $n-1$ times in s_1, and then visits s_2.

However, a state that is reachable from the initial states must be reachable by at least one path that only contains *unique* states. Therefore, we can add a formula $\mathsf{Uniq}(s_1, \ldots, s_n)$ to the induction step that expresses that $s_1, \ldots, s_n$ are different from each other. This restriction on the shape of considered paths makes it impossible to generate counterexamples of arbitrary length from loops in the unreachable state space. The induction step now becomes:

$$\mathsf{Prop}^*(s_1, \ldots, s_n) \wedge \mathsf{Trans}^*(s_1, \ldots, s_{n+1}) \wedge \mathsf{Uniq}(s_1, \ldots, s_{n+1}) \Rightarrow \mathsf{Prop}(s_{n+1})$$

The result is a complete scheme for verifying safety properties of finite systems: If there are paths in the unreachable state space that falsely make the induction step unprovable, they are ruled out from consideration by some induction depth n. However, a major problem is that this n can be extremely large for some verification problems, and that it is often difficult to predict what n is needed.

7.5 Stronger induction in van Eijk's method

We will now make use of the insight into induction methods we gained in the previous section. The underlying proof method that van Eijk's algorithm uses to find equivalences that always hold in the reachable state space is simple induction. Recall that we have demonstrated that this proof technique is too weak to prove all properties that hold globally in the reachable states. Consequently

```
1.  ≡1        := ∅ ;
2.  -- compute first approximation
3.  system    := {Init(s1), Trans*(s1, ..., sn), Holds*(Id, s1, ..., sn)} ;
4.  ≡         := STÅLMARCK(system) ;
5.  ≡2        := (≡ /s1) ∩ ... ∩ (≡ /sn) ;
6.  -- iterate until a fixed point is reached
7.  while (≡1≠≡2) do
8.      ≡1        := ≡2 ;
9.      system    := {Holds*(≡1, s1, ... , sn), Trans*(s1, ..., sn+1),
                                        Holds(Id, sn+1), Uniq(s1, ..., sn+1)} ;
10.     ≡         := STÅLMARCK(system) ;
11.     (≡2)      := ≡1 ∩ (≡ /sn+1) ;
12. return ≡1 ;
```

Figure 7.4: The adaption of the algorithm for depth n unique states induction

there are circuits that contain useful equivalences that van Eijk's original algorithm misses due to the incompleteness of its underlying proof method.

Generalisation to completeness. We can make van Eijk's original algorithm complete by modifying our implementation to use unique states induction with depth n rather than simple induction. In the base case of the algorithm, we compute an equivalence relation on signals that hold in the first n states on paths from the initial states. In the algorithm step, we assume that our most recently computed signal equivalence relation holds in the first n of $n+1$ consecutive unique states, and derive the subset of the signal equivalences that necessarily holds in state $n+1$.

For a detailed description of the resulting algorithm, see Figure 7.4. We use the notation $\mathsf{Holds}^*(\equiv, s_1, \ldots, s_n)$, occuring at lines 3 and 9, as a shorthand for $\mathsf{Holds}(\equiv, s_1) \wedge \cdots \wedge \mathsf{Holds}(\equiv, s_n)$. The algorithm for finding implications between signals is modified in an analogous way.

We can now discover all equivalences that hold globally in the state space. In particular, if a safety property holds in all reachable states, there exists a saturation level and an induction depth that is sufficient to discover that the corresponding signal is equivalent to the true signal.

As an additional benefit, the possibility to adjust both the saturation level and the induction depth allows a high degree of control over how much work is spent on discovering equivalences. We can now increase the number of equivalences that can be discovered for a fixed saturation level by increasing the induction depth; this can be useful as an increase in saturation level means a big change in the time complexity of the algorithm.

We note that the idea of using stronger induction not is restricted to our SAT-based version of van Eijk's algorithm; the original BDD-based algorithm can also be made complete by stronger induction.

1. $n, S_0 := 0, \text{INIT}$;
2. **loop**
3. $\quad S_{n+1} := \text{POSTIMAGE}(S_n) \cup S_n$;
4. $\quad n := n + 1$;
5. **until** $(S_{n+1} = S_n)$;
6. **return** S_{n+1} ;

Figure 7.5: A standard forwards reachability algorithm

7.6 Approximations

In this section we show that van Eijk's algorithm is an *approximative forwards reachability* analysis. We then use this insight to derive an analogous backwards approximative analysis, and combine the two algorithms into a mutual improvement algorithm.

The forwards reachability view. Figure 7.5 shows the shape of a standard forwards reachability analysis, where we use INIT to denote the set of initial states, and the operation POSTIMAGE to compute postimages (the postimage of a set of states S is the set of states that can be reached from S in one transition). We now demonstrate that van Eijk's algorithm performs such a forwards analysis approximatively, in the sense that it is a variant of the standard analysis where INIT has been replaced with an overapproximation, and the exact operations $\cup$ and POSTIMAGE have been replaced by overapproximative operators.

Van Eijk's algorithm computes a sequence of relations $\equiv_i$. Each of the corresponding formulas $\mathsf{Holds}(\equiv_i, s)$ can be seen as the characterisation of a set of states A_i.

In the base case, the algorithm computes the binary relation $\equiv_0$ that holds between points in all the initial states. The formula $\mathsf{Holds}(\equiv_0, s)$ will therefore be valid for every state s that makes $\mathsf{Init}(s)$ valid, and possibly for some other states. A_0 is consequently a superset of the initial states.

In the step, the algorithm computes the subrelation $\equiv_{n+1}$ of $\equiv_n$ that provably holds after a transition under the assumption that $\equiv_n$ holds before the transition. Every state s that is reachable in one transition from a state in A_n therefore makes $\mathsf{Holds}(\equiv_{n+1}, s)$ valid. But $\equiv_{n+1}$ is a subrelation of $\equiv_n$, so every state s in A_n also satisfies $\mathsf{Holds}(\equiv_{n+1}, s)$. Consequently, the step operation corresponds to computing A_{n+1} as the overapproximative union of A_n and the overapproximative postimage of A_n.

Finally, the algorithm checks whether $\equiv_{n+1}$ is the same relation as $\equiv_n$. This corresponds to checking whether $A_{n+1} = A_n$. If this is the case, the algorithm terminates, otherwise the step is iterated.

Approximative backward analysis. It is well known that forwards reachability analysis has a dual analysis called *backwards* reachability analysis [26]. We can perform the backward analysis using the forwards algorithm by modifying

the characterisation of the underlying system in the following way:

$$\begin{aligned} \mathsf{Init}'(s) &= \neg\mathsf{Prop}(s) \\ \mathsf{Trans}'(s,s') &= \mathsf{Trans}(s',s) \\ \mathsf{Prop}'(s) &= \neg\mathsf{Init}(s) \end{aligned}$$

The result of the computation is the set of states that are backwards reachable from the bad states—the states where the property does not hold.

We can use the system transformation together with any of our variants of van Eijk's algorithm. In particular, we can compute a relation $\equiv$ that characterises an overapproximation of the states that are backwards reachable from the states that make $\mathsf{Prop}(s)$ false. Analogously to the forwards algorithm, the system is safe if $\mathsf{Holds}(\equiv, s)$ implies the safety property, which in this case corresponds to that no initial state is in the overapproximation of the set that can be backwards reached from the bad states. Also, if we do intersect the initial states, we can still use the approximation to constrain the transition relation in order to reduce the number of necessary iterations of an exact forwards reachability algorithm.

Mutual improvement. The new approximate backward algorithms can be very useful on their own. However, there exists a general way of enhancing approximative reachability analyses that improves matters further [46].

The idea is to first generate the overapproximation of the reachable states. If the corresponding set has an empty intersection with the bad states, we are done. If it has an nonempty intersection, there are two possible reasons: Either the system is unsafe, or the approximation is too coarse. Regardless of which is the case, we know that the only possible bad states we can reach are those that are contained in the intersection. We can therefore take the intersection to be our new bad states.

But now we can apply approximate backwards reachability analysis from the new bad states. If we do not intersect the initial states, the system must be safe. If we do, we can take the intersection to be the new initial states and restart the whole process. The algorithm terminates if we generate the same overapproximations twice, as this implies that no further improvement is possible.

The resulting mutually improved overapproximations are always at least as good as the original overapproximations, and they can sometimes be substantially better as we demonstrate in the next section.

7.7 Experimental results

In this section, we present a number of experiments we have done using a prototype implementation of our variants of van Eijk's algorithm. We compare our results against the results of three other methods. The first two are reachability analysis and unique states induction, as implemented in the SAT-based model

checking workbench FixIt [1]. The third method is BDD-based model checking, as implemented in the verification tool VIS version 1.3. In the experiments with VIS we have used dynamic variable reordering and experimented with different partitionings.[2] All running times are measured on a 296 MHz Ultrasparc-II with 512 MB memory. The results are displayed in Table 7.1.

The motivation for the choice of benchmarks is as follows. We have chosen one industrial example, one example that is difficult to represent with BDDs, and one example that belongs to the easy category for BDDs.

The Lalita example. The Lalita example is an industrial telecommunications example from Lucent Technologies that was one of the motivations for the research presented in this paper. We received the example as a challenge from Prover Technology, a Swedish formal verification company. It was given to us as a black-box problem; we had no information about the structure of the system. The design was already known to be within reach of BDD technology, but not all of the properties were possible to verify using unique states induction. When we attempted to verify the design using SAT-based reachability analysis, the representations became excessively large due to the computation of pre- and postimages.

The design contains 178 latches. The problem comes with thirteen safety properties that should be verified; we present the four most interesting properties: the two properties that were most difficult for VIS (2 and 7), one of the two properties that are hard for induction (11), and the property that was hardest for our methods (10).

All of the properties except property 10 and 11 can be done using SAT-based induction. However, we can verify or refute every property except property 10 directly using our forwards equivalence algorithm. Property 10 is verified using one iteration of mutual improvement of the computed equivalences. As the table demonstrates our analyses are a factor 10 to 100 faster than BDD-based verification in VIS.

The butterfly circuits. This family of benchmarks arose when we were designing sorting circuits for implementation on an FPGA. The problem is to decide whether a butterfly network containing reconfigurable sequential components is equivalent to an optimised version where the components have been shifted around. When we attempted to verify the circuits we discovered that standard algorithms could not handle circuits of any reasonable size. In particular, BDD-based methods did not work because the BDDs representing the circuits became too large.

The model checking algorithms in VIS are unable to verify larger networks than size 4 in a reasonable amount of time and space. SAT-based reachability analysis and induction are also unable to cope with larger instances of the circuits. However, the forwards equivalence algorithm handles all the sizes we

[2] We have also tried approximate model checking in VIS, but there appears to be a bug in the implementation which makes it unsound.

Property	FixIt Reach.	FixIt Induct.	VIS	Our Method
Lalita, nr. 2	0.3	0.2	219.9	2.3^a
7	41.8	0.2	207.3	2.2^a
10	[>15min]	[>15min]	86.6	9.7^b
11	[>15min]	[>15min]	199.3	2.1^a
Butterfly, size 2	0.1	1.0	0.3	0.1^a
4	16.6	[>15min]	2.0	0.1^a
16	[>15min]	—	[>15min]	1.4^a
64	—	—	—	37.4^a
Arbiter	2.5	[>15min]	2.1	76.9^c

[a] with equivalences, [b] with mutual improvement, [c] with implications

Table 7.1: Experimental results (times are in seconds).

have tried in less than 40 seconds.

The arbiter. This example is a simple benchmark from the VIS distribution. The arbiter controls three clients that compete for bus access. We verify the property of mutual exclusion.

The problem is easy both for BDDs and SAT-based symbolic reachability analysis, but can not be done using unique states induction. The example clearly demonstrates that finding implications between equivalence classes can be stronger than only computing equivalences: Our equivalence based analysis alone is unable to verify the design in a reasonable amount of time, but we can verify the design in 77 seconds by computing implications between the equivalence classes.

7.8 Related work

The first approach in the literature to apply SAT-based techniques to model checking was Bounded Model Checking [13]. Bounded model checking of safety properties corresponds to searching for bugs by attempting to prove the base case only of induction with depth. Certain bugs that are hard to find using BDD-based model checking can be found very quickly in this way. In order to effectively also prove safety of systems, standard symbolic reachability analysis was adapted to use SAT-solvers [1] which resulted in the analysis implemented in FixIt. Currently, interesting work is being done on combinations of SAT-solvers and BDDs for model checking [123].

The idea to use approximate analyses to generate semantic information from systems originally comes from the field of program analysis. Many different such analyses can be seen as abstract interpretations [30]. In particular, Halbwachs et al. [46, 48] have used abstract interpretation techniques to generate linear constraints between arithmetic variables that always holds in the reachable state

space of synchronous programs and timed automata. This information is used both for compilation purposes and for verification. The same techniques are used for generating strengthenings in the STeP system [70] that is targeted towards deductive verification of reactive programs.

The main differences between our work and the work on synchronous programs and STeP, is (1) that the analyses we present here are specially designed for generating information about *gate level circuits* rather than programs, (2) that we focus specifically on simple relations between boolean signals, and (3) that we use Stålmarck's saturation method as a possibly incomplete but fast method for generating the relations. Also, we generate information while keeping in mind that we can apply an exact analysis later, and we have consequently optimised the algorithms for quickly generating information. In the case of synchronous program verification and STeP, a precise analysis is not even possible in general as infinite state systems are addressed.

Dill and Govindaraju [43] have developed a method for performing BDD-based approximate symbolic model checking based on overlapping projections. Their idea is to alleviate BDD blow-up by representing an overapproximation of a set of states S as a vector of BDDs, where each individual BDD characterises the relation in S between some subset of the state variables. The conjunction of the BDDs represents an overapproximation of the underlying set. The main difference between our approach and theirs, is that we consider some particular relations between *all* pairs of signals, while they consider all relations between a number of subsets of state variables. Also, the user of Dill and Govindaraju's method must manually choose the subsets of state variables to build BDDs for, whereas our methods are fully automatic.

7.9 Conclusions and Future Work

We have taken an existing BDD-based verification method which finds equivalent points in a circuit, and adapted it to use Stålmarck's method instead of BDDs. Then, we strengthened the resulting algorithm by combining it with recently developed induction techniques. We also discussed how the algorithm can be improved by computing implications rather than simple equivalences between points. Lastly, we observed that the algorithm can be transformed into a mutual improvement approximative reachability analysis.

The resulting collection of new algorithms can be seen as SAT-based improvements of van Eijk's original algorithm, where we use stronger inductive methods. Viewed from this angle, we have made van Eijk's method complete and provided a more fine-grained tuning between the time used and the information found.

Viewing our work in a different way, we can say that we have improved an inductive method by using van Eijk's algorithm to find equivalences. In some cases, such as the butterfly examples (see Section 7.7), unique states induction needs an exponentially larger induction depth than our improved analyses.

We believe the proposed methods work well for several reasons. First of all, van Eijk's original idea of finding equivalences of points in the circuit makes it very hard for properties to "hide" deep down in the logic of a circuit. Comparing this with problems occurring with methods that only look at state variables (such as conventional model checking methods) or methods that only look at the outputs (such as inductive methods) clearly shows that this is a desirable thing to do.

Second, the use of Stålmarck's saturation algorithm forms a natural fit with van Eijk's original algorithm. The possibility of controlling the saturation level pays off especially in systems where it is hard to find *all* equivalences, but sufficient to find some. Stålmarck's algorithm is also rather robust in the number of variables used in formulas.

Third, inductive methods perform well because they do not need any iteration or complicated quantification. Unfortunately, when we prove partial properties of systems, or when we prove properties about systems with a lot of logic between the latches and the property, induction performs poorly because the induction hypothesis is not strong enough to establish the inductive step. In this case, finding equivalence or implication information is just the right thing to do, because it strengthens the induction hypothesis, and provides direct information not only about the latches, but about all points in the circuit.

For future work, we would like to investigate other signal relations than equivalences and implications. General relations over three variables is a candidate, although it is not clear how to represent the found information. Furthermore, we are interested in extending the proposed algorithms to work with other properties than just safety properties. Lastly, we would like to extend the presented ideas to the verification of safety properties of synchronous reactive systems; for example, systems implemented in the programming language Lustre [47]. In order to do this, we need to add support for integer arithmetic and to investigate how Halbwachs's ideas [46] can be combined with our analyses.

Acknowledgements. Many thanks to Niklas Sörensson who implemented a large part of the algorithms and benchmarks, as well as to Niklas Eén, who helped with running some of the benchmarks. We also thank Byron Cook, Koen van Eijk, Carl-Johan Lillieroth, Gordon Pace, Mary Sheeran, and Satnam Singh for their useful comments on earlier drafts of this paper.

Appendix A

Implementation of Observable Sharing

Here is one way to implement our proposed extension in a Haskell system. Since we are making a non conservative extension to Haskell, we cannot express it using standard Haskell contructs. Therefore, we have to use an "unsafe" operation, called `unsafePerformIO`. This operation, which is not part of the Haskell standard but supported by all major compilers, allows the execution of impure actions by a function that looks pure from the outside.

The only way to perform side-effecting actions in Haskell is to embed them in the *IO monad*, an abstract datatype that allows an encapsulated treatment of imperative actions. So `IO a` is the type of a computation that performs some side-effect and produces a result of type `a`. The type of the "unsafe" operation simply hides the side-effect: `unsafePerformIO :: IO a -> a`.

In the following implementation, we use the IO references of nonstandard library *IOExts*, which is part of the Hugs-GHC extension libraries.[1] We implement our references by creating an abstract datatype `Ref`, which only supports creating, reading and the comparison of such references.

```
module Ref
  ( Ref    -- type
  , ref    -- :: a -> Ref a
  , deref  -- :: Ref a -> a
  , (<=>)  -- :: Ref a -> Ref a -> Bool
  )
 where

import IOExts -- The Hugs-GHC Extension Libraries
```

[1] www.haskell.org/libraries/

```
  ( IORef
  , newIORef
  , readIORef
  , unsafePerformIO
  )

-- Ref: the type of references

newtype Ref a = MkRef (IORef a)

-- operations

ref :: a -> Ref a
ref x = MkRef (unsafePerformIO (newIORef x))

deref :: Ref a -> a
deref (MkRef ref) = unsafePerformIO (readIORef ref)

(<=>) :: Ref a -> Ref a -> Bool
(MkRef ref1) <=> (MkRef ref2) = ref1 == ref2
```

Appendix B

Implementation of QuickCheck

Here, we show the implementation of the QuickCheck library, except for the function `quickCheck`. The source code of QuickCheck is available from `www.cs.chalmers.se/~rjmh/QuickCheck/`.

```
module QuickCheck where

import Monad
import Random

-- Gen

newtype Gen a = Gen (Int -> Rand -> a)

choose :: Random a => (a, a) -> Gen a
choose bounds = Gen (\n r -> fst (randomR bounds r))

variant :: Int -> Gen a -> Gen a
variant v (Gen m) = Gen (\n r ->
  m n (rands r !! (v+1)))
 where
  rands r0 = r1 : rands r2 where (r1, r2) = split r0

promote :: (a -> Gen b) -> Gen (a -> b)
promote f = Gen (\n r -> \a ->
  let Gen m = f a in m n r)

sized :: (Int -> Gen a) -> Gen a
```

```
sized fgen = Gen (\n r ->
  let Gen m = fgen n in m n r)

instance Monad Gen where
  return a     = Gen (\n r -> a)
  Gen m1 >>= k =
    Gen (\n r0 -> let (r1,r2) = split r0
                      Gen m2  = k (m1 n r1)
                  in m2 n r2)

elements :: [a] -> Gen a
elements xs = (xs !!) `liftM` choose (0, length xs - 1)

vector :: Arbitrary a => Int -> Gen [a]
vector n = sequence [ arbitrary | i <- [1..n] ]

oneof :: [Gen a] -> Gen a
oneof gens = elements gens >>= id

frequency :: [(Int, Gen a)] -> Gen a
frequency xs = choose (1, sum (map fst xs)) >>= (`pick` xs)
 where
  pick n ((k,x):xs) | n <= k    = x
                    | otherwise = pick (n-k) xs

-- Arbitrary ; Coarbitrary

class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Bool where
  arbitrary = elements [True, False]

instance Arbitrary Int where
  arbitrary = sized (\n -> choose (-n,n))

instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b) where
  arbitrary = liftM2 (,) arbitrary arbitrary

instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized (\n -> choose (0,n) >>= vector)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a -> b) where
  arbitrary = promote (`coarbitrary` arbitrary)

class Coarbitrary a where
```

```
  coarbitrary :: a -> Gen b -> Gen b

instance Coarbitrary Bool where
  coarbitrary b = variant (if b then 0 else 1)

instance Coarbitrary Int where
  coarbitrary n
    | n == 0    = variant 0
    | n < 0     = variant 2 . coarbitrary (-n)
    | otherwise = variant 1 . coarbitrary (n `div` 2)

instance (Coarbitrary a, Coarbitrary b)
      => Coarbitrary (a, b) where
  coarbitrary (a, b) = coarbitrary a . coarbitrary b

instance Coarbitrary a => Coarbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (a:as) =
    variant 1 . coarbitrary a . coarbitrary as

instance (Arbitrary a, Coarbitrary b)
      => Coarbitrary (a -> b) where
  coarbitrary f gen =
    arbitrary >>= ((`coarbitrary` gen) . f)

-- Property

newtype Property = Prop (Gen Result)

data Result = Result
  {ok :: Maybe Bool, stamp :: [String], arguments :: [String]}

nothing :: Result
nothing = Result
  {ok = Nothing, stamp = [], arguments = []}

result :: Result -> Property
result res = Prop (return res)

class Testable a where
  property :: a -> Property

instance Testable Bool where
  property b = result (nothing{ ok = Just b })

instance Testable Property where
```

```
  property prop = prop

instance (Arbitrary a, Show a, Testable b)
                            => Testable (a -> b) where
  property f = forAll arbitrary f

evaluate :: Testable a => a -> Gen Result
evaluate a = gen where Prop gen = property a

forAll :: (Show a, Testable b) => Gen a -> (a->b) -> Property
forAll gen body = Prop $
  do a   <- gen
     res <- evaluate (body a)
     return (arg a res)
 where
  arg a res = res{ arguments = show a : arguments res }

(==>) :: Testable a => Bool -> a -> Property
True  ==> a = property a
False ==> a = result nothing

label :: Testable a => String -> a -> Property
label s a = Prop (add `fmap` evaluate a)
 where add res = res{ stamp = s : stamp res }

classify :: Testable a => Bool -> String -> a -> Property
classify True  name = label name
classify False _    = property

collect :: (Show a, Testable b) => a -> b -> Property
collect v = label (show v)
```

Appendix C

Implementation of Lava 2000

In this section, we briefly discuss the highlights of the implementation of Lava 2000.

C.1 The Signal Type

The basic type upon which the Lava implementation is based is the `Signal` type. As discussed in the overview of this thesis, and also in the paper about observable sharing [24], this is a datatype containing symbolic operators corresponding to primitive gates and inputs. In order to keep track of the sharing information, we also use `Ref`s.

To realise the `Signal` datatype, we introduce three different datatypes. The first one, `S`, is basically an enumeration of all primitive gates the Lava system supports.

```
data S s = Bool Bool
         | Inv  s
         | And  [s]
         | ...
         | VarBool String
         | DelayBool s s

         | Int Int
         | Plus [s]
         | ...
         | VarInt String
         | DelayInt s s
```

There are two primitive types supported by Lava, booleans and integers. They are present together in one datatype in order to ensure a uniform treatment of signals. Later on, we add a layer of type safety around this.

The datatype `S` is not recursive. The reason for this is that we do not want the necessary use of `Refs` to clutter up the definition of this datatype. Also, it is convenient to be able to define shallow interpretation functions for `S`. These are also called *algebras*. An example of an algebra is the function `eval :: S (S a) -> S a`. We make `S` an instance of the Haskell type class `Functor`.

The second datatype, `Symbol`, adds the recursion and `Refs`. It is defined as:

```
newtype Symbol = Symbol (Ref (S Symbol))
```

The third datatype, the one that the user of Lava is allowed to see, `Signal`, adds a layer of type safety around the `Symbol` type. A similar approach is for example taken in [66, 37].

```
newtype Signal a = Signal Symbol
```

All primitive operations defined on `Signal a` in terms of `Symbol` should guarantee that the `a` is instantiated to the correct type. An example of such an operation is the binary gate `and2`.

```
and2 :: (Signal Bool, Signal Bool) -> Signal Bool
and2 (Signal a, Signal b) = Signal (Symbol (ref (And [a,b])))
```

It is important that we specify the type of this gate, otherwise it would have been polymorphic in the signal type of its inputs and output.

C.2 Overloading

By convention, every circuit is a function from one input object to one output object. These objects can be signals, pairs, larger tuples, lists, trees, etc. In order to deal with all circuit types in a generic way, we use overloading. The idea is that we have a *structure* type that we can convert any object type to and from.

```
data Struct a
  = Compound [Struct a]
  | Single a
```

We also make `Struct` an instance of `Functor`. The type is expressive enough to remember the structure of converted objects and to offer a uniform treatment of them. We introduce a type class `Generic`, for types that can be converted back and forth between structures of `Symbol`s.

```
class Generic a where
  struct    :: a -> Struct Symbol
  construct :: Struct Symbol -> a
```

Some instances of this type are given below. Note that the `construct` operator often has to assume something about the shape of its argument. The user of `construct`, the implementor of Lava, has to guarantee that the argument has the right shape.

```
instance Generic (Signal a) where
  struct    (Signal x) = Single x
  construct (Single x) = Signal x

instance (Generic a, Generic b) => Generic (a,b) where
  struct    (x,y)            = Compound (struct x, struct y)
  construct (Compound [x,y]) = (construct x, construct y)

instance Generic a => Generic [a] where
  struct    xs            = Compound (map struct xs)
  construct (Compound xs) = map construct xs
```

Using `struct` and `construct`, we can define many so-called *generic* operators, which work for all instances of `Generic`. These generic operators can all be defined in terms of the corresponding primitive operator that works only on `Signal`s.

```
(<==>) :: Generic a => a -> a -> Signal Bool
x <==> y = eq (struct x) (struct y)
 where
  eq (Single a)    (Single b)    = eqSignal a b
  eq (Compound as) (Compound bs)
    | length as == length bs     = andl (zipWith eq as bs)
  eq _             _             = low

mux :: Generic a => (Signal Bool, (a, a)) -> a
mux (cond, (x,y)) = construct (mx (struct x) (struct y))
 where
  mx (Single a)    (Single b)    = Single (mxSignal cond a b)
  mx (Compound as) (Compound bs)
    | length as == length bs     = Compound (zipWith mx as bs)
  mx _             _             = error "not compatible"
```

Note that the error message occurs when one for example multiplexes on two lists of different lengths. This is not allowed, since the length of the result should be known at circuit generation time. Another generic operator we can define is `delay`.

One could say we have implemented a form of poor man's polytypic programming [8]!

C.3 Circuit Analyses

The definitions of the functions that inspect the components in a circuit, like circuit transformations and interpretations, also use the `Generic` type class. However, these operations often need to perform some graph algorithm over the structure of the circuit. We have therefore defined a combinator, called `netlistST`, which makes writing these algorithms easier.

```
netlistST :: ST s v -> (v -> S v -> ST s ())
          -> Struct Symbol -> ST s (Struct v)
netlistST new define symbols = ...
```

The combinator takes three arguments, `new`, `define`, and `symbols`, and constructs an action in the `ST` monad. The `ST` monad allows defining stateful algorithms, which is necessary when implementing graph algorithms. All elements of the structure `symbols` are explored in a recursive manner. When going down in the structure, `new` is used whenever a new node is discovered. Note that we keep track of sharing information, so that we only use `new` once for every node. The argument `define` is used when coming back up from the recursion.

In order to implement algorithms that work with structural datatypes and monads, we have found it useful generalise the Haskell function `sequence`, which works on lists.

```
sequence :: Monad m => [m a] -> m [a]
sequence []     = do return []
sequence (m:ms) = do a <- m ; as <- sequence ms ; return (a:as)
```

The function `sequence` performs all monadic actions contained in the argument list in a certain order, and returns a new list containing the results of the corresponding actions. This idea works for many other functors than just lists. In particular, we want to use it for `Struct` and even for `S`.

Thus, we define a type class `Sequent`, with one member function `sequent`.

```
class Functor f => Sequent f where
  sequent :: Monad m => f (m a) -> m (f a)
```

Here, we show instance declarations for lists and `Structs`. The declaration for `S` is done in a similar way.

```
instance Sequent [] where
  sequent = sequence
```

```
instance Sequent Struct where
  sequent (Single m) =
    do a <- m
       return (Single a)

  sequent (Compound xs) =
    do as <- sequence [ sequent x | x <- xs ]
       return (Compound as)
```

A useful combinator defined using this class is a combination of `sequent` and `fmap`. We call it `fmapM`.

```
fmapM :: (Sequent f, Monad m) => (a -> m b) -> f a -> m (f b)
fmapM f = sequent . fmap f
```

An example of a use of the netlist combinator and the sequent operators is the definition of combinational simulation.

```
simulate :: Generic b => (a -> b) -> (a -> b)
simulate circ inp = runST (
  do sref <- netlistST new define (struct (circ inp))
     sout <- fmapM (fmap symbol . readSTRef) sref
     return (construct sout)
  )
 where
  new =
    do newSTRef (error "combinational loop")

  define ref sym =
    do sym' <- fmapM readSTRef sym
       writeSTRef ref (eval sym')
```

First, using `netlistST`, we traverse the structure, creating a new ST reference for every node. Then, when coming back up, we write the result of evaluating the expression belonging to the node in the reference. If we try to read the value of a node before we have defined it, we get an error; in this case the circuit contains a combinational loop.

Analyses which are implemented in a similar way are sequential and constructive simulation, circuit transformations like slowdown and other time transformations, and interpretations such as generating VHDL code and theorem proving.

C.4 Properties

Lastly, we discuss the implementation of properties in Lava. The mechanism that lies behind this is heavily inspired by QuickCheck, see Chapter 4.

First, we define the concept of a generator. This is an object of type `Gen a`. The type constructor `Gen` is a monad that is able to generate new variable names. Just like in QuickCheck, as to be able to deal with infinite structures, we want this monad to be as lazy as possible. For this reason, we do not implement it as a state monad. Instead, we will pass the generator an infinite tree of unique identifiers, which the monadic bind distributes over the subcomputations. To generate the initial tree containing unique names, we use the technique used in [7].

```
data Tree a   = Fork a (Tree a) (Tree a)

newtype Gen a = Gen (Tree Int -> a)

instance Monad Gen where
  return a    = Gen (\t -> a)
  Gen m >>= k = Gen (\(Fork _ t1 t2) ->
                  let a = m t1 ; Gen m2 = k a in m2 t2)
```

There exist default generators for all circuit types which have a statically known size (so lists do not have a default generator). We define these using the type class `Fresh`.

```
class Fresh a where
  fresh :: Gen a

instance Fresh (Signal Bool) where
  fresh = genVar VarBool x

instance Fresh (Signal Int) where
  fresh = genVar VarInt x

genVar con x =
  Gen (\(Fork x _ _) ->
    Signal (Symbol (ref (con ("v" ++ show x)))))

instance (Fresh a, Fresh b) => Fresh (a,b) where
  fresh = liftM2 (,) fresh fresh
```

We can even generate instances for functions! In order to realise this, let us start by defining a type class of types that support a choice operator.

```
class Choice a where
  ifThenElse :: Signal Bool -> (a, a) -> a

instance Choice (Signal a) where
  ifThenElse cond (x,y) = mux (cond, (y,x))
```

```
instance (Choice a, Choice b) => Choice (a,b) where
  ifThenElse cond ((x1,x2),(y1,y2)) =
    (ifThenElse cond (x1,y1), ifThenElse cond (x2,y2))
```

The difference with the `mux` operation we defined earlier using the `Generic` type class is that we can make an *extra* instance for this class, namely functions!

```
instance Choice b => Choice (a -> b) where
  ifThenElse cond (f,g) =
    \x -> ifThenElse cond (f x, g x)
```

We can now define a type class `CoFresh` which allows us to generate fresh functions.

```
class CoFresh a where
  cofresh :: Choice b => Gen b -> Gen (a -> b)
```

Every instance of this class can be used to generate fresh functions:

```
instance (CoFresh a, Choice b, Fresh b)
      => Fresh (a -> b) where
  fresh = cofresh fresh
```

Here are some instances of `CoFresh`:

```
instance CoFresh (Signal Bool) where
  cofresh gen =
    do whenTrue  <- gen
       whenFalse <- gen
       return (\b -> ifThenElse b (whenTrue,whenFalse))

instance (CoFresh a, CoFresh b) => CoFresh (a,b) where
  cofresh gen =
    do f <- cofresh (cofresh gen)
       return (\(a,b) -> f a b)
```

To generate a function taking a boolean argument as argument, we know on beforehand that we have to choose the result between two possibilities. To generate a function taking a tuple as argument, we generate the uncurried version of the function.

Surprisingly enough, we can even make lists an instance of `CoFresh`, thereby allowing the generation of fresh functions taking lists of arbitrary length as arguments!

```
instance CoFresh a => CoFresh [a] where
  cofresh gen =
    do whenNull <- gen
```

```
whenCons <- cofresh (cofresh gen)
return (\xs -> case xs of
                 []   -> whenNull
                 x:xs -> whenCons x xs)
```

This only works because `Gen` is a lazy monad. Otherwise, we would go into an infinite loop at the recursion, where we generate `whenCons`.

Appendix D

Lava 2000 Quick Reference Guide

In this appendix we present an overview of operations, predefined circuits and connection patterns in Lava 2000.

D.1 Logical Gates

Here are the logical gates defined in the Lava system. Some binary gates have a corresponding binary operator (for example, `and2` can also be written as `<&>`).

```
-- Nullary gates :: Signal Bool
low         -- constant low
high        -- constant high

-- Unary gates :: Signal Bool -> Signal Bool
id          -- identity
inv         -- inverse, negation

-- Binary gates :: (Signal Bool, Signal Bool) -> Signal Bool
and2,  <&>  -- logical and
nand2       -- inverse of logical and
or2,   <|>  -- logical or
nor2        -- inverse of logical or
xor2,  <#>  -- logical exclusive or
xnor2, <=>  -- inverse of exclusive or
equiv, <=>  -- logical equivalence
impl,  ==>  -- logical implication
```

```
-- n-ary gates :: [Signal Bool] -> Signal Bool
andl        -- logical and
nandl       -- inverse of logical and
orl         -- logical or
norl        -- inverse of logical or
xorl        -- logical exclusive or
```

D.2 Arithmetical Gates

Here are the arithmetical gates defined in the Lava system. Some binary gates have a corresponding binary operator (for example, `plus` can also be written as `+`).

```
-- Nullary gates :: Signal Int
n           -- constant integer signal

-- Unary gates :: Signal Int -> Signal Int
id          -- identity
neg,   -    -- negation

-- Unary conversion
int2bit     -- integer signal to boolean signal
bit2int     -- boolean signal to integer signal

-- Binary gates :: (Signal Int, Signal Int) -> Signal Int
plus,  +    -- addition
times, *    -- multiplication
sub,   -    -- subtraction
idiv,  /    -- integer division
imod,  %%   -- modulo
imin        -- minimum
imax        -- maximum

-- Binary gates :: (Signal Int, Signal Int) -> Signal Bool
gte,   >>== -- greater than or equal

-- n-ary gates :: [Signal Int] -> Signal Int
plusl       -- addition
timesl      -- multiplication
```

D.3 Generic Gates

Here are some generic gates defined in the Lava system.

```
equal, <==> -- equality
delay, |->  -- delay component
mux         -- multplexer, if-else-then
```

Furthermore, Lava defines some operations which can be used on some of these types:

```
domain     :: [a]
domainList :: Int -> [[a]]

zero       :: a
zeroList   :: Int -> [a]

var        :: String -> a
varList    :: Int -> String -> [a]
```

D.4 Module: Patterns

You get access to the following wiring circuits and connection patterns if you include

```
import Patterns
```

at the top of your Lava program.

```
swap      :: (a, b) -> (b, a)
swapl     :: [a] -> [a]
copy      :: a -> (a, a)

riffle    :: [a] -> [a]
unriffle  :: [a] -> [a]

zipp      :: ([a],[b]) -> [(a,b)]
unzipp    :: [(a,b)] -> ([a],[b])

pair      :: [a] -> [(a,a)]
unpair    :: [(a,a)] -> [a]

halveList :: [a] -> ([a],[a])
append    :: ([a],[a]) -> [a]

serial    :: (a -> b) -> (b -> c) -> (a -> c)
(->-)     :: (a -> b) -> (b -> c) -> (a -> c)
compose   :: [a -> a] -> (a -> a)
composeN  :: Int -> (a -> a) -> (a -> a)
```

```
par       :: (a -> b) -> (c -> d) -> ((a,c) -> (b,d))
(-|-)     :: (a -> b) -> (c -> d) -> ((a,c) -> (b,d))
parl      :: ([a] -> [b]) -> ([a] -> [b]) -> ([a] -> [b])

two       :: ([a] -> [b]) -> ([a] -> [b])
ilv       :: ([a] -> [b]) -> ([a] -> [b])
twoN      :: Int -> ([a] -> [b]) -> ([a] -> [b])
ilvN      :: Int -> ([a] -> [b]) -> ([a] -> [b])
iter      :: Int -> (b -> b) -> (b -> b)

bfly      :: Int -> ([b] -> [b]) -> [b] -> [b]
tri       :: (a -> a) -> ([a] -> [a])

pmap      :: ((a,a) -> (b,b)) -> [a] -> [b]

mirror    :: ((a,b) -> (c,d)) -> ((b,a)     -> (d,c))
row       :: ((c,a) -> (b,c)) -> ((c,[a])   -> ([b],c))
column    :: ((a,c) -> (c,b)) -> (([a],c)   -> (c,[b]))
grid      :: ((a,b) -> (b,a)) -> (([a],[b]) -> ([b],[a]))
```

D.5 Module: `Arithmetic`

You get access to the following arithmetical circuits if you include

```
import Arithmetic
```

at the top of your Lava program.

```
halfAdd  :: (Signal Bool,Signal Bool)
                  -> (Signal Bool,Signal Bool)
fullAdd  :: (Signal Bool,(Signal Bool,Signal Bool))
                  -> (Signal Bool,Signal Bool)

bitAdder :: (Signal Bool,[Signal Bool])
                  -> ([Signal Bool],Signal Bool)
adder    :: (Signal Bool,([Signal Bool],[Signal Bool]))
                  -> ([Signal Bool],Signal Bool)
binAdder :: ([Signal Bool],[Signal Bool])
                  -> [Signal Bool]

bitMulti :: (Signal Bool,[Signal Bool])
                  -> [Signal Bool]
multi    :: ([Signal Bool],[Signal Bool])
                  -> [Signal Bool]
```

```
numBreak :: Signal Int -> (Signal Bool,Signal Int)
int2bin  :: Int -> Signal Int -> [Signal Bool]
bin2int  :: [Signal Bool] -> Signal Int
```

D.6 Module: SequentialCircuits

You get access to the following often used sequential circuits if you include

```
import SequentialCircuits
```

at the top of your Lava program.

```
edge         :: Signal Bool -> Signal Bool
toggle       :: Signal Bool -> Signal Bool
delayClk     :: a -> (Signal Bool,a) -> a
delayN       :: Int -> a -> a -> a
always       :: Signal Bool -> Signal Bool
puls         :: Int -> () -> Signal Bool
outputList   :: [a] -> () -> a

rowSeq       :: ((a,b) -> (c,a)) -> (b -> c)
rowSeqReset  :: ((a,b) -> (c,a)) -> ((Signal Bool,b) -> c)
rowSeqPeriod :: Int -> ((a,b) -> (c,a)) -> (b -> c)
```

Note that these functions are not completely polymorphic in a, but there are certain restrictions.

D.7 Interpretations

Here are the various interpretations for circuits that Lava provides.

```
-- simulations
simulate circuit input
simulateSeq circuit inputs
simulateCon circuit inputs
test circuit

-- VHDL
writeVhdl name circuit
writeVhdlInput name circuit input
writeVhdlInputOutput name circuit input output

-- verification
```

```
verify property
verifyWith options property
fixit property
```

Possible verification opions are:

```
Name name
ShowTime
Sat level
NoBacktracking
Depth depth
Increasing
RestrictStates
```

D.8 Errors

Here, we list a number of error messages that might occur when running the Lava system.

- `ERROR: Garbage collection fails to reclaim sufficient space`

  This means that Lava does not have enough memory to execute the circuit. Try to start up Lava with more memory, do this by saying `lava -h9999999`. You can increase the number if you need more.

  If this does not work, you might have an error in your circuit definition. Do you have a circular definition somewhere?

- `Program error: evaluating a delay component`

  You get this error when you try to use combinational simulation `simulate` to simulate a sequential circuit. Use `simulateSeq` instead.

- `Program error: evaluating a symbolic value`

  You get this error when you have used the `forAll` or `var` property constructors, and then later tried to simulate the circuit.

- `Program error: combinational loop`

  You get this error when you have defined a circuit which has a loop in it, on which there is no delay. In general, these circuits are hard to give meaning to, and are therefore not allowed in normal Lava simulation. You have probably made a mistake somewhere.

  You might try the *constructive* simulation `simulateCon` when this happens.

- `Program error: combining incompatible structures`

  You get this error when you use a `delay` component or `mux` component on structures of a different shape, for example two lists of different lengths. This is not allowed, since the length of a list needs to be known when you evaluate the circuit.

- `Program error: there is no equality defined for this type`

  Sigh ... you get this error when you use the Haskell equality `==` on a signal type. You probably want to use signal equality `<==>` instead.

- `Program error: undriven output`

  This also happens when you have a *bad* combinational loop in your circuit. The output wire is not driven by any component. An example is the following circuit:

```
undrivenOutput () = out
  where
    out = and2 (out, out)
```

- `Program error: you can not enumerate symbolic values`

  You get this error when you use `..` on wires from a circuit instead of on constants. Use `..` only on constants!

- `Program error: INTERNAL ERROR ...`

  Oops! This probably means that there is a bug in the Lava system. Please report this bug by sending your program to us, so that we can fix it.

If you have some typical error that you would have liked to appear here, please e-mail us so that we can make this list more complete.

Bibliography

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[2] F. Anceau. A synchronous approach for clocking VLSI systems. *IEEE Journal of Solid-State Circuits*, 17:51–56, Feb. 1982.

[3] S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. In *Irvine Software Symposium*, pages 29–48, March 1992.

[4] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proc. POPL'95, the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, January 1995.

[5] Z. M. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. POPL'98, the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–74. ACM Press, January 1998.

[6] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc., 1996.

[7] Lennart Augustsson, Mikael Rittri, and Dan Synek. Functional Pearl: On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.

[8] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming - an introduction. In *Lecture notes in Computer Science*, volume 1608, 1999.

[9] K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, 1969.

[10] Gérard Berry. The constructive semantics of Pure Esterel. Unfinished draft, available from `http://www.esterel.org`, 1999.

[11] Gérard Berry. The Esterel primer. Available from `http://www.-esterel.org`, 2000.

[12] Patrice Bertin and Hervé Touati. PAM programming environments: Practice and experience. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[13] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '99, 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[14] Per Bjesse. Specification of signal processing programs in a pure functional language and compilation to distributed architectures. Master's thesis, Chalmers University of Technology, 1997.

[15] Per Bjesse. Automatic verification of combinational and pipelined FFT circuits. In *Computer Aided Verification*. Springer Verlag, July 1999.

[16] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer Aided Design*, 2000.

[17] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava - Hardware design in Haskell. In *International Conference on Functional Programming*. ACM SigPlan, Sept. 1998.

[18] M. Blum and S. Kannan. Designing programs that check their work. In *Proc. 21st Symposium on the Theory of Computing*, pages 86–97. ACM, May 1989.

[19] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Symposium on the Theory of Computing*, pages 73–83. ACM, May 1990.

[20] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[21] A. Celentano, S. C. Reghizzi, P. Della Vigna, and C. Ghezzi. Compiler testing using a sentence generator. *Software – Practice & Experience*, 10:897–918, 1980.

[22] K. Claessen. Safety property verification of cyclic circuits. Submitted for publication, 2001.

[23] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming*, Montreal, 2000. ACM.

[24] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, Phuket, Thailand, 1999. ACM Sigplan.

[25] Koen Claessen and Mary Sheeran. A Tutorial on Lava: A Hardware Description and Verification System. Available from `http://www.cs.chalmers.se/~koen/Lava`, April 2000.

[26] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[27] B. Cook and J. Launchbury. Disposable memo functions. In *Haskell Workshop*. Amsterdam, ACM SigPlan, 1997.

[28] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*. Marstrand, Sweden, 1998.

[29] J.W. Cooley and J.W. Tukey. An algorithm for the machine computation of complex Fourier series. In *Mathematics of Computation, 19*, pages 297–301, 1965.

[30] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc.* 4^{th} *ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[31] David Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In *Formal Methods for Computer Aided Design of Electronic Circuits (FMCAD)*, number 1166 in Lecture Notes In Computer Science. Springer-Verlag, 1996.

[32] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[33] Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. In *CHARME'99 Poster Session*, 1999.

[34] J. Duran and S. Ntafos. An evaluation of random testing. *Transactions on Software Engineering*, 10(4):438–444, July 1984.

[35] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Conf. on Design, Automation and Test in Europe*, 1998.

[36] Conal Elliott. An embedded modeling language approach to interactive 3D and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3), May/June 1999.

[37] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *Workshop on Semantics, Applications and Implementation of Program Generation*, 2000.

[38] FDL: Forum on Design Languages. Current conference information on `http://www.ecsi.org/ecsi/fdl/`, 2001.

[39] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[40] Alexandre Frey, Gérard Berry, Patrice Bertin, François Bourdoncle, and Jean Vuillemin. Jazz. Available from `http://www.cma.ensmp.fr/jazz`, 1998.

[41] Ruben Gamboa. Mechanically verifying the correctness of the Fast Fourier Transform in ACL2. In *Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, 1998.

[42] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *Trans. Prog. Lang. and Systems*, (3):211–223, 1981.

[43] S. G. Govindaraju and D. L. Dill. Approximate symbolic model checking using overlapping projections. In *Electronic Notes in Theoretical Computer Science*, July 1999. Trento, Italy.

[44] Jan Friso Groote. The propositional theorem prover HeerHugo. Technical report, CWI, 1997 (?).

[45] N. Halbwachs. A tutorial of Lustre. Available from `http://-www-verimag.imag.fr/SYNCHRONE`, 1993.

[46] N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur, Belgium, September 1994. LNCS 864, Springer Verlag.

[47] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[48] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[49] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[50] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. *Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[51] Keith Hanna and Neil Daeche. Dependent types and formal synthesis. *Phil. Trans. R. Soc. Lond. A*, (339), 1992.

[52] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999.

[53] Jifeng He, Ian Page, and Jonathan Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, number 683 in LNCS. Springer, 1993.

[54] Shousheng He. *Concurrent VLSI Architectures for DFT Computing and Algorithms for Multi-output Logic Decomposition.* PhD thesis, Lund Institute of Technology, 1995.

[55] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996.

[56] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, December 1996.

[57] J. Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[58] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[59] Graham Hutton. Functional programming with relations. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, 1990. Springer-Verlag.

[60] Steven Johnson. *Synthesis of Digital Designs from Recursion Equations.* The ACM Distinguished Dissertation Series, The MIT Press, 1984.

[61] Geraint Jones. A fast flutter by the Fourier transform. In *Proceedings IVth Banff Workshop on Higher Order.* Springer Workshops in Computing, 1990.

[62] Geraint Jones and Mary Sheeran. The study of butterflies. In *Proceedings IVth Banff Workshop on Higher Order.* Springer Workshops in Computing, 1990.

[63] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. *Science of Computer Programming*, 22(1–2), 1994.

[64] M. P. Jones. The Hugs distribution. Currently available from `http://-haskell.org/hugs`, 1999.

[65] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, January 1993.

[66] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages*, 1999.

[67] Yanbing Li and Miriam Leeser. HML: An innovative hardware design language and its translation to VHDL. In *Computer Hardware Description Languages (CHDL'95)*, 1995.

[68] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In J McCanny, J McWhirter, and E Swartzlander, editors, *Systolic Array Processors*, pages 589 – 598. Prentice-Hall International, 1989.

[69] S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7), July 1994.

[70] Z. Manna and the STeP group. STeP: The Stanford Temporal Prover. Technical report, Computer Science Department, Stanford University, July 1994.

[71] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, number 630 in LNCS. Springer, 1992.

[72] F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS. Springer, 1997.

[73] Florence Maraninchi and Yann Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*. Springer, 1998.

[74] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[75] John Matthews. Recursive function definition over coinductive types. In *Theorem Proving in Higher-Order Logics*, 1999.

[76] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–56, 1990.

[77] David May. Compiling Occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley, 1990.

[78] William W. McCune and L. Wos. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[79] Ken McMillan. The SMV Model Checker. Available from `http://-www-cad.eecs.berkeley.edu/~kenmcmil/smv/`, 2000.

[80] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[81] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, pages 43–56. ACM Press, January 1999.

[82] Martin Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994.

[83] J. O'Donnell. Hardware description with recursion equations. In *IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382. North-Holland, 1987.

[84] J. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Springer-Verlag Workshops in Computing, pages 178–194, 1993.

[85] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languagues in Education*, volume 1125 of *Lecture Notes In Computer Science*, pages 221–234. Springer Verlag, 1996.

[86] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.

[87] Ian Page and Wayne Luk. Compiling Occam into field-programmable gate arrays. In Wayne Luk and Will Moore, editors, *FPGAs*, pages 271–283. Abingdon EE&CS books, 1991.

[88] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declerative Languages*, 1999.

[89] S. Peyton Jones, S. Marlow, and C. Elliot. Stretching the storage manager: weak pointers and stable names in Haskell. In *Implementation of Functional Languages*. Nijmegen, 1999.

[90] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, May 1996.

[91] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[92] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering. In *International Conference on Functional Programming*, 2000.

[93] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[94] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*, pages 152–163. IEEE Computer Society Press, Washington, 1996.

[95] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Intl. Symp, Gdansk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

[96] A. Poigné and L. Holenderski. On the combination of synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS, pages 490–514. Springer, 1997.

[97] John Proakis and Dimitris Manolakis. *Digital Signal Processing*. Macmillan, 1992.

[98] Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, number 1099 in LNCS. Springer, 1996.

[99] F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, June 1991.

[100] P. Sansom and S. Peyton Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):334–385, January 1997.

[101] M. Schenke and M. Dossis. Provably correct hardware compilation using timing diagrams. Available from `http://semantik.Informatik.Uni-Oldenburg.DE/persons/michael.schenke/`, 1997.

[102] Sentovich, E. M. and K. J. Singh et al. SIS: A system for sequential circuit synthesis. Technical Report Berkeley, UCB/ERL M92/41, 1992.

[103] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[104] Robin Sharp and Ole Rasmussen. Transformational rewriting with Ruby. In *Computer Hardware Description Languages (CHDL'93)*. Elsevier Science Publishers, 1993.

[105] M. Sheeran. Describing hardware algorithms in Ruby. In *Functional Programming, Glasgow 1989*, Springer Workshops in Computing. Springer, 1990.

[106] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.

[107] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000.

[108] Mary Sheeran. μFP, an algebraic VLSI design language. PhD thesis, Programming Research Group, Oxford University, 1983.

[109] Mary Sheeran. μFP, a language for VLSI design. In *ACM Symp. on LISP and Functional Programming*, 1984.

[110] Mary Sheeran. Designing regular array architectures using higher order functions. In Jouannaud, editor, *Int. Conf. on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes In Computer Science*. Springer Verlag, 1985.

[111] Mary Sheeran and Arne Borälv. How to prove properties of recursively defined circuits using Stålmarck's method. In *Formal Techniques for Hardware and Hardware-like Systems*. Marstrand, Sweden, 1998.

[112] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, 1996.

[113] Satnam Singh. *Analysis of Hardware Description Languages*. PhD thesis, Computing Science Dept., Glasgow University, 1991.

[114] SLDL: System Level Design Languages. SLDL initiative homepage at `http://www.inmet.com/SLDL/`, 1998.

[115] G. Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula. *Swedish Patent No. 467076 (1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995)* , 1989.

[116] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[117] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[118] H. Touati and Mark Shand. PamDC: a C++ library for the simulation and generation of Xilinx FPGA designs. Available from `http://research.compaq.com/SRC/pamette/PamDC.pdf`, 1999.

[119] VIS: Verification Interacting with Synthesis. Available from `http://vlsi.colorado.edu/~vis`, 2000.

[120] P. Wadler. Monads for Functional Programming. In *Lecture notes for Marktoberdorf Summer School on Program Design Calculi*, NATO ASI Series F: Computer and systems sciences. Springer Verlag, August 1992.

[121] Philip Wadler. Theorems for free! In *International Conference on Functional Programming and Computer Architecture*, London, September 1989.

[122] Philip Wadler. A prettier printer, March 1998. Draft paper.

[123] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, 2000.

[124] Niklaus Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, 1998.

[125] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *Computing Surveys*, 29(4):366–427, December 1997.