Lava: Hardware Design in Haskell

Per Bjesse, Koen Claessen, Mary Sheeran
Chalmers University of Technology, Sweden
{bjesse, koen, ms}@cs.chalmers.se

Satnam Singh
Xilinx, USA
satnam@xilinx.com

Abstract

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.

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 [She85] and Ruby [JS90], and on the use of non-standard interpretation in circuit analysis [Sin91].

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 [SB98]. 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

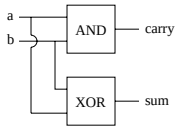

Figure 1: A half adder circuit

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 [SR93].

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 [CLM98]. We chose the second.

2 Overview of the System

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

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* [Wad92]. 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 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.

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.

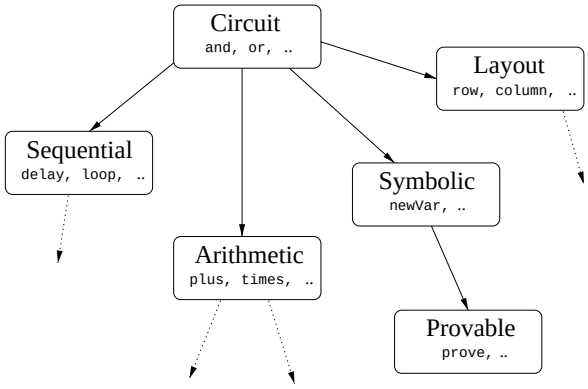

Figure 2: Type Class structure for Interpretations

2.2 Type Classes

Some circuit operations are meaningful only to certain interpretations; Lava is therefore structured with type classes (see figure 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 4.1).

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

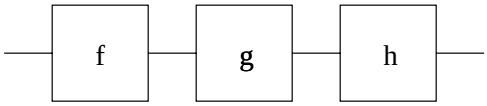

Figure 3: `compose` [f,g,h]

will slightly extend the datatype later (see section 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 4.1).

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 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 4 and 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.

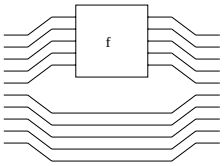

Figure 4: `one` f

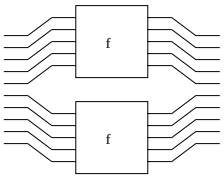

Figure 5: `two` f

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

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.

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))
  ...
```

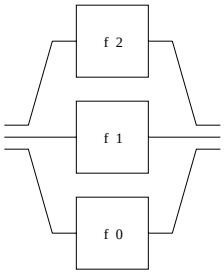

Figure 6: `decmap 3 f`

We can now simulate the example of section 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 beyond the scope of this paper.

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 in this paper are theorem provers. We briefly sketch other possibilities in section 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 2.1, 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"`.

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`.

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 [Stå89] and the first order logic theorem provers Otter [MW97] and Gandalf [Tam97].

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 2.1) 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.

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 [Jon90] and specifying signal processing software in Haskell [Bje97].

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 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.

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 [He95].

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 [PM92]. 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 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.

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 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.

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 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 7 and 8).

The FFT butterfly stages are constructed by riffling together two halves of a sequence of length k, processing them by a

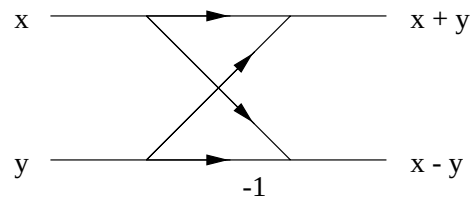

Figure 9: A butterfly

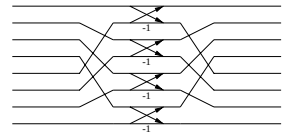

Figure 10: A butterfly stage of size 8 expressed with riffling

column of $k/2$ butterfly circuits, and unriffling the result (see figure 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 $-j$.

```
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 [PM92]. 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]
          ]
```

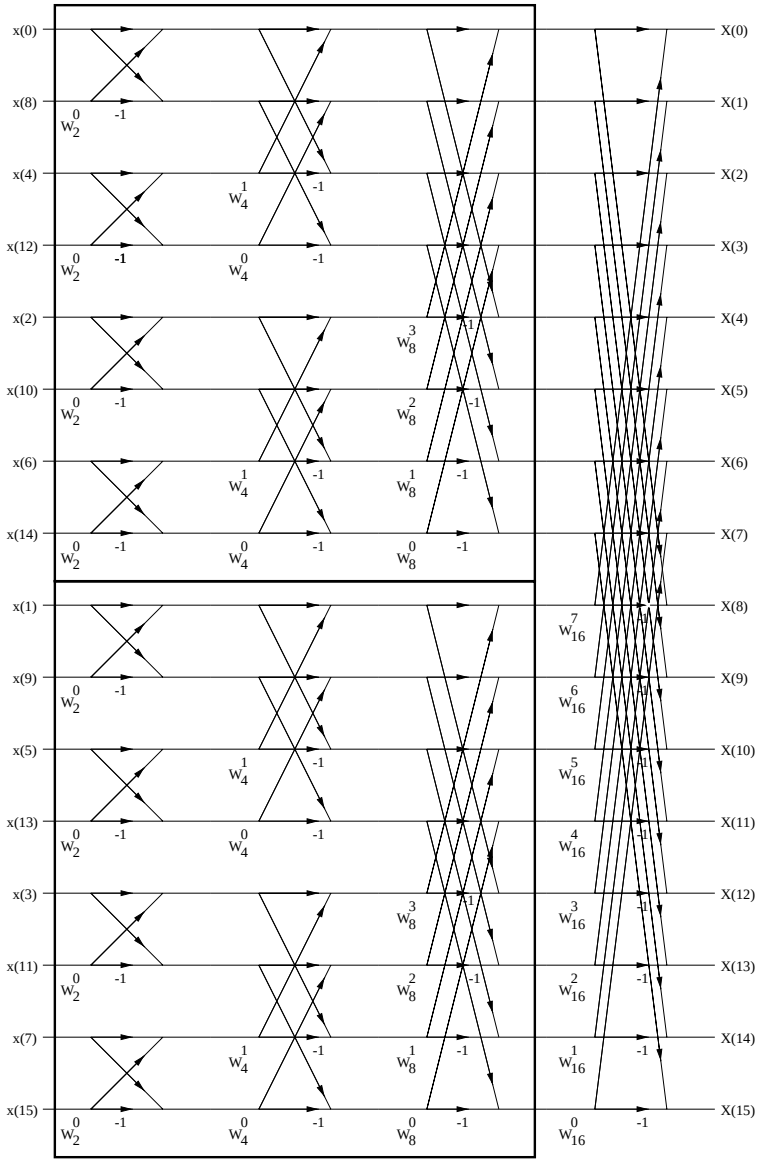

Figure 7: A size 16 Radix 2 FFT network

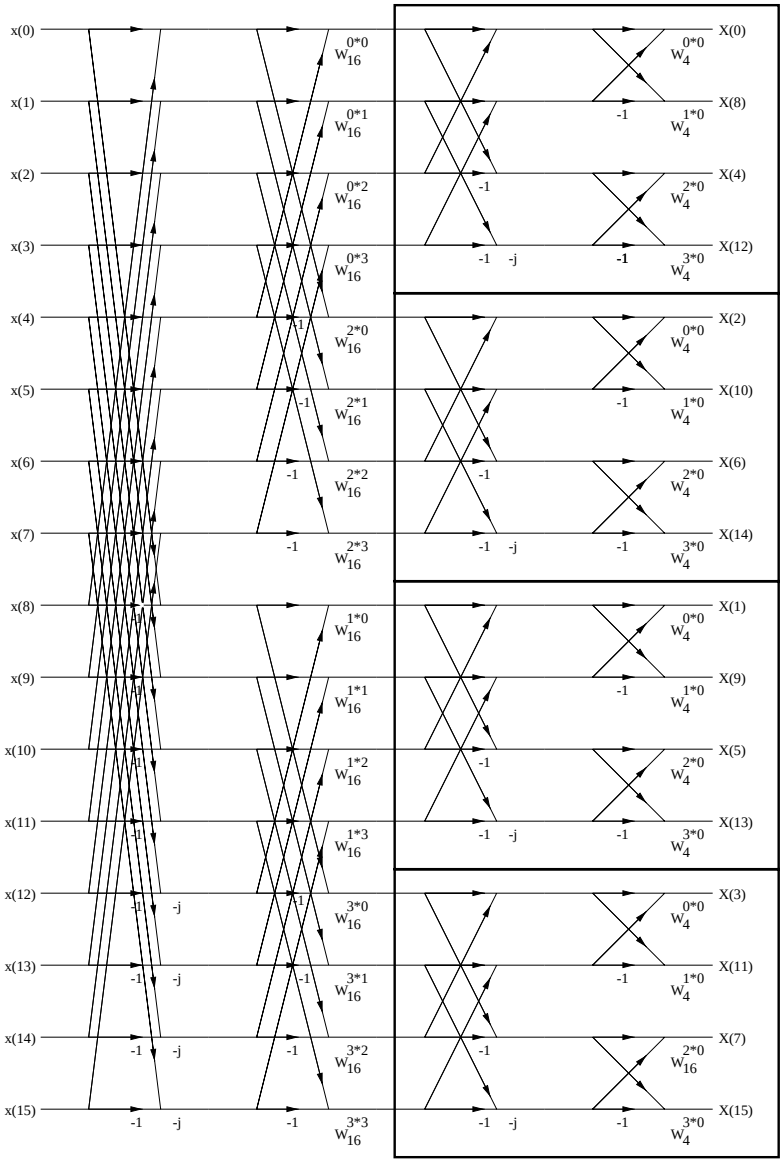

Figure 8: A size 16 Radix-2^2 FFT network

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.

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.

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 [MW97]. 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 11 shows the formula that is generated as input to Otter (notice the arithmetic and twiddle factor theory).

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 [Gam98]. 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.

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 [She85]. 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

```
%% 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 11: Otter input for size 4 FFT comparison

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 [O'D96]. 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 [CLM98]. 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 [HD92]. 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 [Cyr96], 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 [LL95]. 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.

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 us 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.

7 Future Work

We are continuing to develop the Lava system; this paper is a report of work in progress rather than a description of a finished project. 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 [Sin91]. 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.

References

[Bje97] 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.

[CLM98] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*. Marstrand, Sweden, 1998.

[Cyr96] 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.

[Gam98] 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.

[HD92] Keith Hanna and Neil Daeche. Dependent types and formal synthesis. *Phil. Trans. R. Soc. Lond. A*, (339), 1992.

[He95] Shousheng He. *Concurrent VLSI Architectures for DFT Computing and Algorithms for Multi-output Logic Decomposition*. PhD thesis, Lund Institute of Technology, 1995.

[Jon90] Geraint Jones. A fast flutter by the Fourier transform. In *Proceedings IVth Banff Workshop on Higher Order*. Springer Workshops in Computing, 1990.

[JS90] Geraint Jones and Mary Sheeran. The study of butterflies. In *Proceedings IVth Banff Workshop on Higher Order*. Springer Workshops in Computing, 1990.

[LL95] Yanbing Li and Miriam Leeser. HML: An innovative hardware design language and its translation to VHDL. In *Computer Hardware Description Languages (CHDL'95)*, 1995.

[MW97] William W. McCune and L. Wos. Otter: The CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997.

[O'D96] John 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.

[PM92] John Proakis and Dimitris Manolakis. *Digital Signal Processing*. Macmillan, 1992.

[SB98] 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.

[She85] Mary Sheeran. Designing regular array architectures using higher order functions. In *Int. Conf. on Functional Programming Languages and Computer Architecture, (Jouannaud ed.)*, volume 201 of *Lecture Notes In Computer Science*. Springer Verlag, 1985.

[Sin91] Satnam Singh. *Analysis of Hardware Description Languages*. PhD thesis, Computing Science Dept., Glasgow University, 1991.

[SR93] Robin Sharp and Ole Rasmussen. Transformational rewriting with Ruby. In *Computer Hardware Description Languages (CHDL'93)*. Elsevier Science Publishers, 1993.

[Stå89] Gunnar Stålmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (1994), European Patent No. 0403 454 (1995).

[Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[Wad92] Philip 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.