Embedding a Hardware Description Language in Template Haskell

John T. O'Donnell

University of Glasgow, United Kingdom

Abstract. Hydra is a domain-specific language for designing digital circuits. Research on techniques for implementing Hydra have focused on embedding it within a pure functional language. Many features required for hardware specification fit well within functional languages, leading in many cases to a perfect embedding. There are some situations, however, where the DSL does not fit exactly within the host functional language. A new solution to these problems is based on program transformations performed automatically by metaprograms in Template Haskell.

1 Introduction

The development of a computer hardware description language, called Hydra, provides an interesting perspective on embedding as an implementation technique for domain specific languages (DSLs). Hydra has many features that fit smoothly within a higher order, nonstrict pure functional language, and these aspects of Hydra demonstrate the effectiveness and elegance of embedding. There are several areas, however, where Hydra does not fit perfectly within the host functional language. This creates technical difficulties that must be solved if the embedding is to work. Therefore an overview of the history of Hydra implementations provides a number of insights into embedded implementations of DSLs.

The Hydra project grew out of the work by Steven Johnson on using recursion equations over streams to describe the behavior of digital circuits [9]. Hydra was also inspired by Ruby [11], a relational language for circuit design. Hydra has been developed gradually over the last twenty years [15], [16], [17], [19], [20], concurrently with the development of functional languages. Hydra has been embedded in six related but distinct languages over the years: Daisy, Scheme, Miranda, LML, Haskell [5], and now Template Haskell [23].

The rest of this paper describes some of the main concepts in Hydra and how they were embedded in a functional language. The presentation follows a roughly chronological approach, showing how the implementation has gradually become more sophisticated. However, one major liberty will be taken with the history: Haskell notation will be used throughout the paper, even while discussing embedding techniques that were developed for earlier functional languages. Some of the earlier papers on Hydra, cited in the bibliography, use the earlier language notations.

One consequence of the design philosophy of embedding used in Hydra is that the concrete syntax of Hydra ha changed radically as it moved from one language to the next. Indeed, some of the earlier papers are now difficult to read for anyone unfamiliar with the history of functional programming.

2 A perfect embedding

The central concept of functional programming is the mathematical function, which takes and argument and produces a corresponding output. The central building block of digital circuits is the component, which does the same. This fundamental similarity lies at the heart of the Hydra embedding: a language that is good at defining and using functions is also likely to be good at defining and using digital circuits.

2.1 Circuits and functions

Hydra models a component, or circuit, as a function. The arguments denote inputs to the circuit, and the result denotes the outputs. An equation is used to give a name to a wire (normally called a *signal*):

```
x = or2 (and2 a b) (inv c)
```

We can turn this circuit into a black box which can be used like any other component by defining function (Figure 1).

```
circ a b c = x
  where x = or2 (and2 a b) (inv c)
```

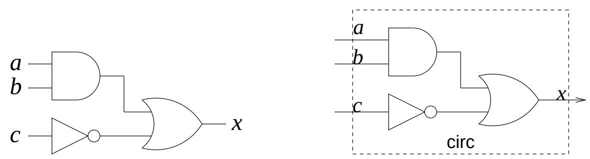

Fig. 1. Modeling a black box circuit as a function

In order to make the circuit executable, the basic logic gates need to be defined. A simple approach is to treat signals as booleans, and to define the logic gates as the corresponding boolean operators.

```
inv = not
and2 = (&&)
or2 = (||)
```

A useful building block circuit is the multiplexor `mux1`, whose output can be decribed as "if $c = 0$ then a else b" (Figure 2):

```
mux1 c a b =
  or2 (and2 (inv c) a) (and2 c b)
```

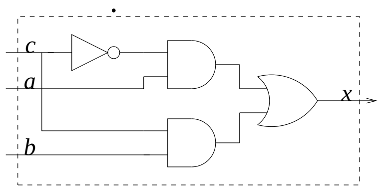

Fig. 2. The multiplexor circuit

2.2 Modeling state with streams

The functional metaphor used above requires each circuit to act like a pure mathematical function, yet real circuits often contain state. A crucial technique for using streams (infinite lists) to model circuits with state was discovered by Steven Johnson [9]. The idea is to treat a signal as a sequence of values, one for each clock cycle. Instead of thinking of a signal as something that changes over time, it is a representation of the entire history of values on a wire. This approach is efficient because lazy evaluation and garbage collection combine to keep only the necessary information in memory at any time.

The delay flip flop `dff` is a primitive component with state; at all times it outputs the value of its state, and at each clock tick it overwrites its state with the value of its input. Let us assume that the flip flop is initialized to 0 when power is turned on; then the behavior of the component is defined simply as

```
dff x = False : x
```

Now we can define synchronous circuits with feedback. For example, the 1-bit register circuit has a load control `ld` and a data input `x`. It continuously outputs its state, and it updates its state with `x` at a clock tick if `ld` is true.

```
reg1 ld x = s
  where s = dff (mux1 ld s x)
```

The specification of `reg1` is natural and uncluttered, and it is also an executable Haskell program. The following test case defines the values of the input signals for several clock cycles, and performs the simulation:

```
sim_reg1 = reg1 ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]
```

This is now executed using the interactive Haskell interpreter `ghci`, producing the correct output:

```
*Main> sim_reg1
[False,True,True,True,False,False,False]
```

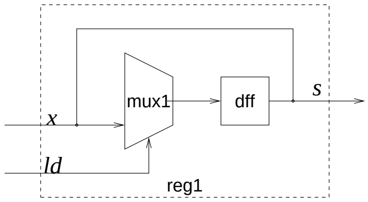

Fig. 3. Feedback in a synchronous register circuit

2.3 Design patterns and higher order functions

Digital circuits tend to have highly regular structures with enormous amounts of replication. These patterns are captured perfectly by higher order functions. For example, several important circuits have a structure (Figure 4) described by the `ascanr` function, one of an extremely useful family of map, fold and scan combinators.

```
ascanr ::  (b->a->a) -> a -> [b] -> (a,[a])
ascanr f a xs =
  (foldr f a xs,
   [foldr f a (drop (i+1) xs) | i <- [0 .. length xs - 1]])
```

A ripple carry adder can now be defined using `ascanr` to handle the carry propagation, and the `map2` combinator (similar to `zipWith` to compute the sums.

```
add1 :: Signal a => a -> [(a,a)] -> (a,[a])
add1 c zs =
 let (c',cs) = ascanr bcarry c zs
     ss = map2 bsum zs cs
  in (c',ss)
```

This specification operates correctly on all word sizes. In other words, it defines the infinite class of n-bit adders, so the circuit designer doesn't need to design a 4-bit adder, and then an 8-bit one, and so on. Furthermore, we can reason formally about the circuit using equational reasoning. A formal derivation of an $O(\log)$ time parallel adder, starting from this $O(n)$ one, is presented in [22].

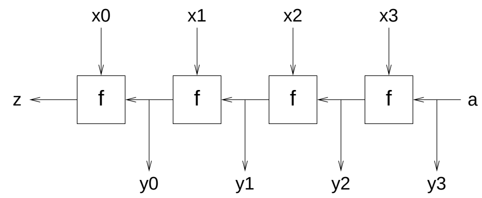

Fig. 4. The ascanr pattern.

3 Waiting for type classes

An unusual aspect of Hydra is that a circuit specification has several semantics, not just one. This led to a mismatch between the target and host language which persisted for about ten years and through four host programming languages (Daisy, Scheme, Miranda and LML). The mismatch was finally resolved when type classes were introduced in Haskell.

3.1 Hardware models and multiple semantics

There is generally only one thing to be done with a computer program: run it. For this reason, we are accustomed to thinking about *the* denotational semantics of a program. In contrast, there are several tasks that must be performed with a circuit specification, including simulation, timing analysis, netlist generation, and layout. Furthermore, there are many different ways to simulate a circuit, corresponding to various models of hardware.

One way to handle the multiple semantics of a circuit specification is to introduce, for each semantics, a special representatation for signals and corresponding implementations of the primitive components. These definitions are organized into modules, one for each semantics. A specification is a function definition containing free variables (`dff`, `and2`, etc.), which are resolved by loading up the module containing definitions for the desired semantics.

The early versions of Hydra used modules in exactly this way. To simulate a circuit, the designer loads a simulation module and the circuit specification module, and then simply executes the circuit. The other semantics are obtained just by loading the appropriate module.

This method worked adequately, but it did have some drawbacks. The main problem is that it forces the designer to work entirely within one semantic world at a time, yet it is quite natural to want to evalute an expression according to one semantics in the course of working within another one.

3.2 The Signal class

Type classes in Haskell provide a more flexible, elegant and secure way of controlling the multiple circuit semantics. Static (untimed) signals are treated as a class, with methods for the constant values and the primitive components:[1]

```
class Static a where
  zero, one :: a
  inv :: a -> a
  and2 :: a -> a -> a
   ...
```

One semantics for combinational circuits uses the `Bool` type to represent signals:

```
  instance Static Bool where
  zero = False
   ...
  inv = not
  and2 a b = a && b
  or2 a b = a || b
   ...
```

There are other static signal instances that allow for richer circuit models, allowing techniques like tristate drivers, wired or, and bidirectional buses to be handled.

A static signal can be lifted to a clocked one in the usual way, using streams:

```
  instance Static a => Clocked [a] where
  zero = repeat zero
   ...
  dff xs = zero : xs
  inv xs = map inv xs
   ...
```

Now we can execute a single circuit specification in different ways, simply by applying it to inputs of the right type:

```
circ :: Signal a => a -> a -> a -> a
circ a b c = x
  where x = or2 (and2 a b) (inv c)

test_circ_1 = circ False False False
test_circ_2 = circ
--                  0      1      2      3
```

[1] The form of the Signal class is somewhat different in Hydra. In particular, logic gates like `inv` are are actually circuits built from lower level primitives.

```
                [False, False, True,  True]
                [False, True,  False, True]
                [False, True,  True,  True]
test_circ_3 = circ (Inport "a") (Inport "b") (Inport "c")
```

The following session with the Haskell interpreter `ghci` executes `circ` to perform a boolean simulation, a clocked boolean simulation, and a netlist generation (see Section 4).

```
*Main> test_circ_1
True
*Main> test_circ_2
[True,False,False,True]
*Main> test_circ_3
Or2 (And2 (Inport "a") (Inport "b")) (Inv (Inport "c"))
```

4 Preserving referential transparency

The preceding sections describe the implementation of Hydra in Haskell as a perfect embedding. The whole approach works smoothly because the foundation of functional programming—the mathematical function—is also the most suitable metaphor for digital circuits. Circuit specifications can be written in a style that is natural and concise, yet which is also a valid functional program.

In this section, we consider a crucial problem where the embedding is highly imperfect. (Another way of looking at it is that the embedding is *too* perfect, preventing us from performing some essential computations.) The problem is the generation of netlists; the state of the problem as of 1992 is described in [17]. A new solution, based on Template Haskell, will be presented in detail in a forthcoming paper, and some of the techniques used in the solution are described briefly in this section.

4.1 Netlists

A netlist is a precise and complete description of the structure of a circuit. It contains a list of all the components, and a list of all the wires. Netlists are unreadable by humans, but there are many automated systems that can take a netlist and fabricate a physical circuit. In a sense, the whole point of designing a digital circuit is to obtain the netlist; this is the starting point of the manufacturing process.

A result of the perfect embedding is that the simulation of a Hydra specification is faithful to the underlying model of hardware. The behavioral semantics of the circuit is identical to the denotational semantics of the Haskell program that describes it.

Sometimes, however, we don't want the execution of a Hydra specification to be faithful to the circuit. Examples of this situation include handling errors in feedback loops, inserting logic probes into an existing circuit, and generating netlists. The first two issues will be discussed briefly, and the rest of this section will examine the problem of netlists in more detail.

In a digital circuit with feedback, the maximum clock speed is determined by the critical path. The simulation speed is also affected strongly by the critical path depth (and in a parallel Hydra simulator, the simulation speed would be roughly proportional to the critical path depth). If there is a purely combinational feedback loop (as the result of a design error), then the critical path depth is infinite. But a faithful simulation may not produce an error message; it may go into an infinite loop, faithfully simulating the circuit failing to converge. (It is possible that such an error will be detected and reported by the Haskell runtime system as a "black hole error", but there is no guarantee of this—and such an error message gives little understanding of where the feedback error has occurred.

A good solution to this problem is a circuit analysis that detects and reports feedback errors. However, that requires the ability to traverse a netlist.

Circuit designers sometimes test a small scale circuit using a prototype board, with physical chips plugged into slots and physical wires connecting them. The circuit will typically have a modest number of inputs and outputs, and a far larger number of internal wires. A helpful instrument for debugging the design is the logic probe, which has a tip that can be placed in contact with any pin on any of the chips. The logic probe has light emitting diodes that indicate the state of the signal. This tool is the hardware analogue of inserting print statements into an imperative program, in order to find out what is going on inside.

Logic probes are not supported by the basic versions of Hydra described in the previous sections, because they are not faithful to the circuit: the Hydra simulation computes exactly what the real circuit does, and in a real circuit there is no such thing as a logic probe.

4.2 Netlist semantics

Hydra generates netlists in two steps. First, a special instance of the signal class causes the execution of a circuit to produce a graph which is isomorphic to the circuit. Second, a variety of software tools traverse the graph in order to generate the netlist, perform timing analyses, insert logic probes, and so on.

The idea is that a component with n inputs is represented by a graph node tagged with the name of the component, with n pointers to the sources of the component's input signals. There are two kinds of signal source:

– An input to the circuit, represented as an `Inport` node with a `String` giving the input signal name.
– An output of some component, represented by a pointer to that component.

The following algebraic data type defines graphs where nodes correspond to primitive components, or inputs to the entire circuit. (This is not the representation actually used in Hydra, which is much more complex—all of the pieces of Hydra implementation given here are simplified versions, which omit many of the capabilities of the full system.)

```
data Net = Inport String | Dff Net | Inv Net
  | And2 Net Net | Or2 Net Net
```

The `Net` type can now be used to represent signals, so it is made an instance of the `Signal` class.

```
instance Signal Net where
  dff = Dff
  inv = inv
  and2 = And2
  or2 = Or2
```

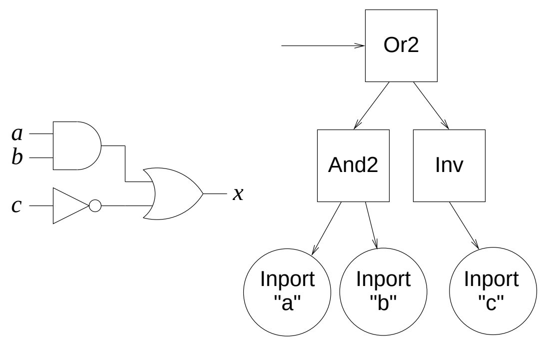

Fig. 5. The schematic and graph of a combinational circuit

Figure 5 shows the graph of `circ` (defined in a previous section). This is constructed simply by applying the `circ` function to `Inport` nodes with signal names. Since `Inport`, applied to a string, returns a value of type `Net`, the entire execution of the circuit will select the `Net` instances for all signals, and the final output will be the circuit graph. This is illustrated by entering an application of `circ` to input ports, using the interactive `ghci` Haskell system. The output is an `Or2` node, which points to the nodes that define the input signals to the `or2` gate.

```
*Main> circ (Inport "a") (Inport "b") (Inport "c")
Or2 (And2 (Inport "a") (Inport "b")) (Inv (Inport "c"))
```

Given a directed acyclic graph like the one in Figure 5, it is straightforward to write traversal functions that build the netlist, count numbers of components, and analyze the circuit in various ways.

4.3 Feedback and equational reasoning

When a circuit contains a feedback loop, its corresponding graph will be circular. Consider, for example, a trivial circuit with no inputs and one output, defined as follows:

```
x = dff (inv x)
```

Assuming that the flip flop is initialized to 0 when power is turned on, the circuit will oscillate between 0 and 1 forever. The Hydra simulation shows this:

```
*Main> oscillate
[False,True,False,True,False,True,False,True,False,True,False, ...
```

Perhaps the deepest property of a pure functional language like Haskell (and Hydra) is referential transparency, which means that we can always replace either side of an expression by the other side. Now, in the equation

$$x = \mathit{dff}\ (\mathit{inv}\ x),$$

we can replace the x in the right hand side by any value α, as long as we have an equation $x = \alpha$. And we do: the entire right hand side is equal to x. The same reasoning can be repeated indefinitely:

```
x = dff (inv x)
  = dff (inv (dff (inv x)))
  = dff (inv (dff (inv (dff (inv x)))))
  ...
```

All of these circuits have exactly the same behavior. But it is less clear whether they have the same structure. A designer writing a specification with n flip flops might expect to end up with a circuit containing n flip flops, so the sequence of equations should arguably specify the sequence of circuits shown in Figure 6.

With existing Haskell compilers, the sequence of distinct equations above would indeed produce distinct graph structures. However, these graphs are circular, and there is no way for a pure Haskell program to traverse them without going into an infinite loop. Consequently, a Hydra tool written in Haskell cannot generate a netlist for a circuit with feedback written in the form above, and it cannot determine whether two of the graphs in Figure 6 are equal.

4.4 The impure solution: pointer equality

The first language in which Hydra was embedded was Daisy, a lazy functional language with dynamic types (like Lisp). Daisy was implemented by an interpreter that traversed a tree representation of the program, and the language offered constructors for building code that could then be executed. In short,

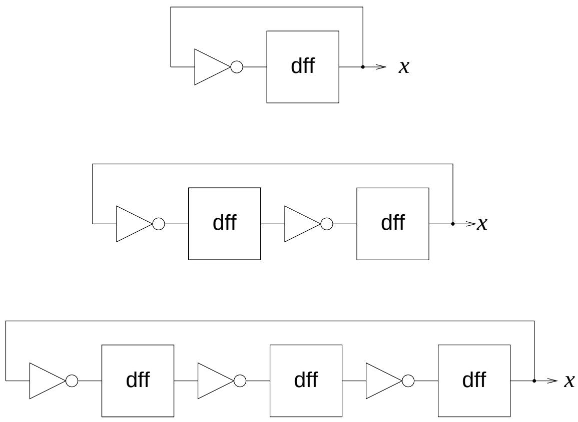

Fig. 6. The dff-inv circuit

Daisy supported metaprogramming, and this was exploited in early research on debugging tools, operating systems, and programming environments.

Daisy took the same approach as Lisp for performing comparisons of data structures: there was a primitive function that returned true if two objects are identical—that is, if the pointers representing them had the same value. This was used in turn by the standard equality predicate.

In the true spirit of embedding, the Daisy pointer equality predicate was used in the first version of Hydra to implement safe traversal of circuit graphs. This is a standard technique: the traversal function keeps a list of nodes it has visited before; when it is about to follow a pointer to a node, the traversal first checks to see whether this node has already been processed. This technique for netlist generation is described in detail in [16].

There is a serious and fundamental problem with the use of a pointer equality predicate in a functional language. Such a function violates referential transparency, and this in turn makes equational reasononing unsound.

The severity of this loss is profound: the greatest advantage of a pure functional language is surely the soundness of equational reasoning, which simplifies the use of formal methods sufficiently to make them practical for problems of real significance and complexity. For example, equational reasoning in Hydra can be used to derive a subtle and efficient parallel adder circuit [22].

The choice to use pointer equality in the first version of Hydra was not an oversight; the drawbacks of this approach were well understood from the outset, but it was also clear that netlist generation was a serious problem that would require further work. The simple embedding with pointer equality provided a convenient temporary solution.

Lava [1] [3], a recent hardware description language which is essentially a clone of Hydra, has adopted Hydra'88's pointer equality method for generating netlists [2]. The differences in Lava are that the method is called "observable sharing" rather than the traditional name "pointer equality", and the name of the function that compares two pointers for equality is also changed. In substance, however, the observable sharing technique for generating netlists is identical to the one presented in [16], and it suffers from the same drawbacks. In particular, papers on Lava stress the use of formal methods like theorem provers and model checkers, but they downplay the fact that the input to such tools is provided by a netlist generator which has an unsound semantic foundation. This approach is justified by the claim that current Haskell compilers probably do not perform optimizations that would lead to incorrect results in the circuits.

Meanwhile, the unsafe pointer equality method was abandoned in Hydra around 1990, and replaced by the labelling transformation discussed in the next section.

4.5 The labelling transformation

It is absolutely essential for a hardware description language to be able to generate netlists. There are only two possible approaches: either we must use impure language features, such as the pointer equality predicate, or we must find a way to write circuit specifications in a form that enables us to work around the problem.

The fundamental difficulty is that we need a way to identify uniquely at least one node in every feedback loop, so that the graph traversal algorithms can determine whether a node has been seen before. This can be achieved by decorating the circuit specification with explicit labeling function:

```
label :: Signal a => Int -> a -> a
```

Now labels can be introduced into a circuit specification; for example, a labelled version of the `reg1` circuit might be written as follows:

```
reg1' ld x = s
  where s = label 100 (dff (mux1 ld s x))
```

It is straightforward to add `label` to the Signal class, and to define an instance of it which ignores the label for behavioral signal types:

```
instance Signal Bool where
   ...
  label s x = x
```

Now the labelled circuit `reg'` can be simulated just like the previous version:

```
sim_reg' = reg1' ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]
```

In order to insert the labels into circuit graphs, a new Label node needs to be defined, and the instance of the label function for the `Net` type uses the `Label` constructor to create the node:

```
data Net = ...
  | Label Int Net

instance Signal Net where
    ...
  label = Label
```

Now we can define executions of the labelled specification for both simulation and netlist types:

```
sim_reg' = reg1' ld x
  where ld = [True, False, False, True, False, False]
        x  = [True, False, False, False, False, False]

graph_reg' = reg1' (Inport "ld") (Inport "x")
```

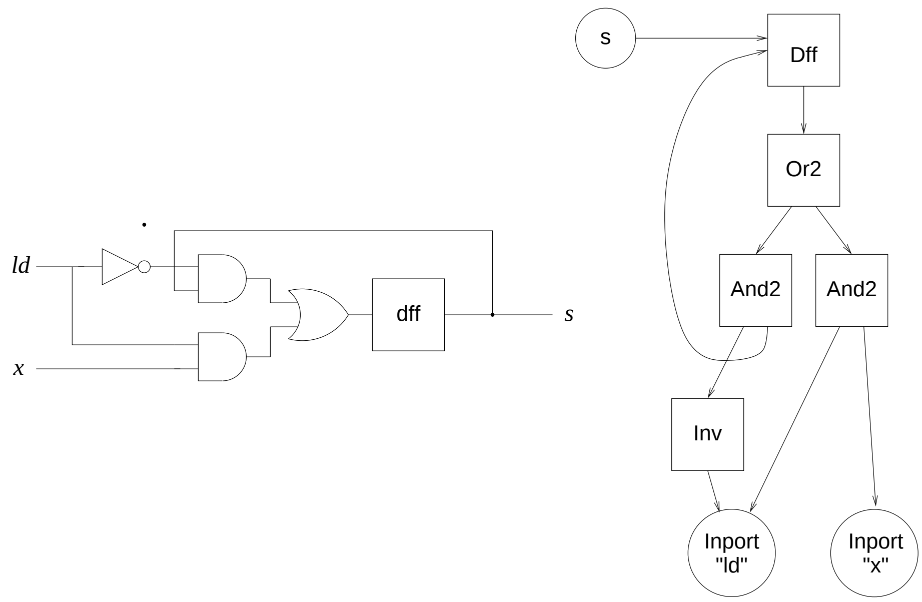

Fig. 7. Schematic and netlist for `reg1`

Figure 7 shows the schematic diagram and the corresponding circuit graph. The results of executing these test cases using `ghci` are shown below. The simulation produces the correct outputs. The graph can be printed out, but it is a circular graph so the output will be infinite (and is interrupted interactively by typing Control-C).

```
*Main> sim_reg'
[False,True,True,True,False,False,False]
*Main> graph_reg'
Label 100 (Dff (Or2 (And2 (Inv (Inport "ld")) (Label 100 (Dff (Or2
(And2 (Inv (Inport "ld")) (Label 100 (Dff (Or2 (And2 (Inv (Inport
"ld")) (Label 100 (DfInterrupted.
```

Although it is not useful to print the graph directly, the presence of labels makes it possible to traverse it in order to build a netlist. Rather than show a full netlist traversal here, consider the equivalent but simpler problem of counting the number of flip flops in a circuit. This is achieved by the following function (there is an equation corresponding to each constructor in the `Net` type, but only a few are shown here):

```
count_dff :: [Int] -> Net -> Int
count_dff ls (Inport s) = 0
   ...
count_dff ls (Dff x) = 1 + count_dff ls x
   ...
count_dff ls (And2 x y) = count_dff ls x + count_dff ls y
   ...
count_dff ls (Label p x) =
  if p `elem` ls
    then 0
    else count_dff (p:ls) x
```

Using labels, the sequence of circuits shown in Figure 6 can be defined unambiguously:

```
circloop, circloop1, circloop2, circloop3 :: Net
circloop = x
  where x = dff (inv x)
circloop1 = x
  where x = Label 1 (dff (inv x))
circloop2 = x
  where x = Label 1 (dff (inv (Label 2 (dff (inv x)))))
circloop3 = x
  where x = Label 1 (dff (inv (Label 2
              (dff (inv (Label 3 (dff (inv x))))))))
```

Executing the test cases produces the following results. First, the number of flip flops in the labelled register circuit is counted correctly. The unlabelled circuit results in a runtime exception, since the circuit graph is equivalent to an infinitely deep directed acyclic graph. The labelled circuits are all distinguished correctly.

```
*Main> count_dff [] graph_reg'
1
*Main> count_dff [] circloop
*** Exception: stack overflow
*Main> count_dff [] circloop1
1
*Main> count_dff [] circloop2
2
*Main> count_dff [] circloop3
3
```

The use of labelling solves the problem of traversing circuit graphs, at the cost of introducing two new problems. It forces a notational burden onto the circuit designer which has nothing to do with the hardware, but is merely an artifact of the embedding technique. Even worse, the labelling must be done correctly and yet cannot be checked by the traversal algorithms.

Suppose that a specification contains two different components that were mistakenly given the same label. Simulation will not bring out this error, but the netlist will actually describe a different circuit than the one that was simulated. Later on the circuit will be fabricated using the erroneous netlist. No amount of simulation or formal methods will help if the circuit that is built doesn't match the one that was designed.

When monads were introduced into Haskell, it was immediately apparent that they had the potential to solve the labelling problem for Hydra. Monads are often used to automate the passing of state from one computation to the next, while avoiding the naming errors that are rife with ordinary `let` bindings. Thus a circuit specification might be written in a form something like the following:

```
circ a b =
  do p <- dff a
     q <- dff b
     x <- and2 p q
     return x
```

The monad would be defined so that a unique label is generated for each operation; this is enough to guarantee that all feedback loops can be handled correctly.

However, there are two disadvantages of using monads for labelling in Hydra. A relatively minor problem is that monads introduce new names one at a time, in a sequence of nested scopes, while Hydra requires the labels to come into scope recursively, all at once, so that they are all visible throughout the scope of a circuit definition. In the above example, the definition of `p` cannot use the signal `q`.

A far more severe problem is that the circuit specification is no longer a system of simultaneous equations, which can be manipulated formally just by "substituting equals for equals". Instead, the specification is now a sequence of computations that—when executed—will yield the desired circuit. It feels like writing an imperative program to draw a circuit, instead of defining the circuit directly. Equational reasoning would still be sound with the monadic approach, but it would be far more difficult to use: the monadic circuit specification above contains no equations at all, and if the monadic operations are expanded out to their equational form, the specification becomes bloated, making it hard to manipulate formally.

To the author of this paper, the indirect nature of the specification and the loss of simple equational reasoning are absolutely fatal flaws in the monadic approach. If an embedding into Haskell requires us to give up the most fundamental advantages of Hydra, then the embedding is worse than useless and a standalone Hydra compiler should be constructed instead. These issues were fully understood in the early 1990s—one of this author's first thoughts, on hearing about monads, was that here is a possible way to generate Hydra netlists—but the idea was rejected almost immediately.[2]

Not everyone shares this view. The hardware description Hawk [6] incorporates many of the techniques developed for Hydra, including stream recursion, higher order functions for design patterns, multiple semantics and type classes, and labelling for netlist generation. However, Hawk uses monads to introduce the labels. In order to get around the scoping problem, the designers of Hawk added a new recursive `do` construct to Haskell.

Hawk is claimed to be suitable for formal reasoning about circuits, but this is apparently limited to the application of heavyweight packages like model checkers and theorem provers [7] to a completed design [4] [14]. One could argue, however, that mathematics is especially effectuve during the process of designing a circuit. It seems a great loss to limit the use of formal methods to seeking bugs in completed designs that had to be carried out without the assistance of formal methods. The monadic solution has a terrible cost.

4.6 Hydra in Template Haskell

Instead of requiring the designer to insert labels by hand, or using monads, the labels could be inserted automatically by a program transformation. This requires developing a parser for Hydra, an algebraic data type for representing the language, and functions to analyze the source and generate the target Haskell code, which can then be compiled. This work was actually underway when Template Haskell became available, offering a much more attractive approach.

Template Haskell [23] provides the ability for a Haskell program to perform computations at compile time which generate new code that can then be spliced into the program. It is similar in many ways to macros in Scheme, which have

[2] Some functional programmers, including the author of this paper, feel that monads are overused. Monads are exactly the right tool for some problems, but they also allow imperative styles of thinking to be grafted thoughtlessly into functional programs that might better be written in equational style. Beginners in Haskell are especially prone to use monads to solve every problem in sight, and they become a crutch enabling the programmer to avoid rethinking a problem in a more functional style.

long been used for implementing domain specific languages within a small and powerful host.

Template Haskell defines a standard algebraic data type for representing the abstract syntax of Haskell programs, and a set of monadic operations for constructing programs. These are expressible in pure Haskell. Two new syntactic constructs are also introduced: a reification construct that gives the representation of a fragment of code, and a splicing construct that takes a code representation tree and effectively inserts it into a program.

The following definition uses the reification brackets `[d| ... |]` to define `circ_defs_rep` as an algebraic data type representing the code:

```
circ_defs_rep = [d|
  circ1 a b c = (x,y)
    where x = and2 a b
          y = or2 b c
  |]
```

The `transform_module` function is applied to the code tree; this is just ordinary Haskell. The transformation function analyzes the code, checks for a variety of errors and issues domain-specific error messages if necessary, inserts the labels correctly, and also performs some other useful tasks. The result of this is a new code tree. The `$(...)` syntax then splices the new code into the program, and resumes the compilation. The effect is the same as if the programmer had written the transformed code in the first place.

```
$(transform_module circ_defs_rep)
```

The actual transformation is rather complex, and is beyond the scope of this paper. The algebraic data type used to represent netlists contains far more information than the simple `Net` type used earlier in this paper. It does more than support simple netlist generation; it also provides the information needed for several other software tools. The organization of the transformation is straightforward, consisting of the following main steps:

- Use pattern matching to determine the kind of declaration.
- Traverse the argument patterns to discover the names, and construct a renaming table.
- Build new patterns incorporating the alpha conversions.
- Copy the expressions with alpha conversions.
- Construct the black box graph structure.
- Generate unique labels and attach them to the nodes denoting the local equations.
- Assemble the final transformed function definition.
- Print out the original and transformed code, as well as temporary values, if requested by the user.

The embedding of Hydra in Template Haskell allows the circuit designer to write specifications in a simple, uncluttered form. Most of the features of Hydra come for free from the embedding, while others are obtained via an automated program transformation.

5 Conclusion

The design and implementation of Hydra is an extreme example of embedding a domain specific language in a general purpose language. The embedding has always been fundamental to the whole conception of Hydra. At the beginning, in 1982, the project was viewed as an experiment to see how expressive functional languages really are, as well as a practical tool for studying digital circuits. It took several years before Hydra came to be seen as a domain specific language, rather than just a style for using functional languages to model hardware. Hydra didn't receive its name[3] until 1986.

There are several criteria for judging whether an embedding is successful:

- *Does the host language provide all the capabilities needed for the domain specific language?* It's no good if essential services—such as netlist generation—must be abandoned in order to make the DSL fit within the host language.
- *Does the host language provide undesirable characteristics that would be inherited by the domain specific one?* This criterion is arguably a compelling reason not to use most imperative programming languages for hosting a hardware description language. But users of VHDL would probably disagree, and there is always a degree of opinion in assessing this criterion. For example, many people find the punctuation-free syntax of Haskell, with the layout rule, to be clean, readable, and elegant. Others consider it to be an undesirable feature of Haskell which has been inherited by Hydra.
- *Have the relevant advantages of the host language been retained?* The most uncompromising requirement of Hydra has always been support for equational reasoning, in order to make formal methods helpful during the design process. This is a good reason for using a functional language like Haskell, but it is also essential to avoid any techniques that make the formal reasoning unsound, or unnecessarily difficult.
- *Does the embedding provide a notation that is natural for the problem domain?* A domain specific language really must use notation that is suitable for the application domain. It is unacceptable to force users into a completely foreign notation, simply in order to save the DSL implementor the effort of writing a new compiler. Before Template Haskell became available, Hydra failed badly on this criterion. If the hardware designer is forced to insert labels manually, in order to solve the netlist problem, one could argue that the embedding is causing serious problems and a special purpose compiler would be more appropriate.

These criteria have been used as guiding principles through the entire project. Of course they are not simple binary success/failure measures. It is always fair

[3] It is appropriate that the name was suggested by Steve Johnson, whose groundbreaking work on Daisy and modelling hardware with streams was the starting point for the project that evolved into Hydra. The system had been called HDRE, an acronym for "hardware description with recursion equations", and Steve wrote approximately the following comment on a draft paper: "HDRE is pronounced Hydra, I presume?"

to ask, for example, whether the notation could be made more friendly to a hardware designer if we did not need to employ the syntax of Haskell, and opinions differ as to what notations are the best. This situation is not peculiar to Hydra: all other hardware description languages also inherit syntactic clutter from existing programming languages, even when they are just inspired by the languages and are not embedded in them.

Template Haskell offers a significant improvement to the ability of Haskell to host domain specific languages. Wherever there is a mismatch between the DSL and Haskell, a metaprogram has the opportunity to analyze the source and translate it into equivalent Haskell; there is no longer a need to make the original DSL code serve also as the executable Haskell code.

In effect, Template Haskell allows the designer of a domain specific language to find the right mixture of embedding and compilation. It is no longer necessary for an embedding to be perfect in order to be useful. Haskell has excellent facilities for general purpose programming, as well as excellent properties for formal reasoning. Now that the constraints on embedding have been relaxed, Haskell is likely to find much wider use as the host for domain specific languages.

References

1. Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, April 2001.
2. Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*. ACM SIGPLAN, 1999.
3. Koen Claessen and Mary Sheeran. *A Tutorial on Lava: A Hardware Description and Verification System*. Chalmers University of Technology, April 2000. `www.cs.chalmers.se//`*sim*`koen/Lava`.
4. Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, 1998.
5. Simon Peyton Jones (ed.). Haskell 98 language and libraries. *Journal of Functional Programming*, 13(1):1–255, January 2003.
6. John Launchbury et. al. Hawk. Web page, 1998–. www.cse.ogi.edu/PacSoft/Projects/Hawk/.
7. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
8. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
9. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984. The ACM Distinguished Dissertation Series.
10. G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second Int. Conf. on Mathematics of Program Construction*, LNCS. Springer, 1992.
11. Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods in VLSI Design*, chapter 1, pages 13–70. North-Holland, 1990. IFIP WG 10.5 Lecture Notes.
12. Ian Lynagh. Template Haskell: A report from the field. May 2003.

13. Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. May 2003.
14. John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings ICCL'98*. IEEE Press, 1998.
15. John O'Donnell. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382, Amsterdam, April 1987. North-Holland.
16. John O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328, Amsterdam, 1988. North-Holland.
17. John O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 178–194. Springer-Verlag, 1992.
18. John O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.
19. John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *FPLE'95: Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 195–214. Springer-Verlag, 1995.
20. John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings 16th International Parallel & Distributed Processing Symposium*, page 234 (abstract). IEEE Computer Society, April 2002. Workshop on Parallel and Distribued Scientific and Engineering Computing with Applications—PDSECA.
21. John O'Donnell, Timothy Bridges, and Sidney Kitchel. A VLSI implementation of an architecture for applicative programming. *Future Generation Computer Systems*, 4(3):245–254, October 1988.
22. John O'Donnell and Gudula Rünger. Derivation of a carry lookahead addition circuit. *Electronic Notes in Theoretical Computer Science*, 59(2), September 2001. Proceedings ACM SIGPLAN Haskell Workshop (HW 2001).
23. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.