What's the Matter with Kansas Lava?

Andrew Farmer, Garrin Kimmell, and Andy Gill

Information Technology and Telecommunication Center,
Department of Electrical Engineering and Computer Science,
The University of Kansas,
2335 Irving Hill Road,
Lawrence, KS 66045, USA
{anfarmer,kimmell,andygill}@ku.edu

Abstract. Kansas Lava is a functional hardware description language implemented in Haskell. In the course of attempting to generate ever larger circuits, we have found the need to effectively test and debug the internals of Kansas Lava. This includes confirming both the simulated behavior of the circuit and its hardware realization via generated VHDL. In this paper we share our approach to this problem, and discuss the results of these efforts.

1 Introduction

Lava is a Domain Specific Language (DSL) embedded in Haskell that allows for the description of hardware circuits using Haskell functions [1]. It turns out that such a DSL, known as a functional hardware description language, represents a natural way to express circuits. For instance, the definition of a half adder, which takes two bits as inputs, adds them, and returns the result bit and a carry bit, is:

```
halfAdder :: Bit -> Bit -> (Bit, Bit)
halfAdder a b = (carry,sum)
  where carry = and2 a b
        sum   = xor2 a b
```

The half adder can be run like a normal Haskell function. We call this mode of running the circuit *simulation.*

```
ghci> halfAdder true true
(T,F)
```

We can also, under the correct conditions, capture our half adder function as an abstract syntax tree, which we can render into a traditional hardware description language, such as VHDL. We call this process *synthesis.*

1.1 What is Kansas Lava?

Kansas Lava is an effort to create a modern Lava implementation that allows direct specification of circuits like traditional Lava. Kansas Lava makes extensive use of recent design patterns, like Applicative Functors, to permit concise and natural expression of hardware concerns as Haskell functions. Kansas Lava also heavily leverages advanced extensions to GHC's type system, like scoped type variables and type functions, to offer a high degree of control over signal representations, accurate simulation, and assurances of correctness.

Notice that the `halfAdder` circuit above, when defined in Kansas Lava, can be used in two contexts: as a combinatorial circuit and as a circuit operating on a sequence of clocked values:

```
ghci> :t halfAdder
halfAdder :: (Signal sig) =>
    sig Bool -> sig Bool -> (sig Bool, sig Bool)
ghci> :t true
true :: Comb Bool
ghci> halfAdder true true
(T,F)
ghci> let x = toSeq $ cycle [False,False,True,True]
ghci> let y = toSeq $ cycle [False,True]
ghci> halfAdder x y
(F :~  F :~  F :~  T :~ ..., F :~  T :~  T :~  F :~ ...)
```

1.2 In This Paper

Programmatically generating hardware presents new challenges when it comes to testing and debugging. Often, traditional lightweight testing strategies are adept at discovering errors, via simulation, but offer little help in determining the cause of the error.

When used in this debugging context, they often scale poorly to large circuits since they don't permit inspection of intermediate values, and require a functional decomposition to expose these internal streams that may be unnatural or time consuming. What is needed in such a situation is a debugging tool that permits the inspection of intermediate values and the testing of parts of a circuit in isolation.

Additionally, none of these tools allows a means of testing the synthesized circuit. This is a prime concern both for the implementors of Kansas Lava and users in the late stages of circuit design – who need to refine circuits to meet the constraints imposed by their chosen hardware substrate.

In this paper we present a solution to this problem for Kansas Lava. Specifically, we:

- show why existing solutions like QuickCheck are often ineffective in this domain (Section 2).

- introduce the notion of probes, a means of observing intermediate values without loss of modularity (Section 3), and explain their implemention (Section 4).
- show that probes offer a natural way to compare the simulated circuit to the sythesized version (Section 5).
- demonstrate the powerful ability of probes to demarcate subcircuits, permitting a form of automated algorithmic debugging which compares the behavior of the simulation to that of the synthesized VHDL (Section 6).

Using these tools has allowed us to debug large, real-world Lava circuits in a more straightforward manner than in the past.

2 Testing Functional Circuits

We specifically want to test two aspects of circuits in Kansas Lava. First, that they behave correctly in simulation, like any Haskell function. Secondly, that the generated VHDL behaves exactly the same as the simulated circuit, clock for clock.

A popular method of testing Haskell functions is to use the QuickCheck tool [2]. QuickCheck allows the Haskell programmer to define logical assertions about the bahavior of functions, known as properties, and then attempts to find counter-examples by generating random inputs to each property and checking the result. The programmer is given a means to control how the random inputs are generated via a typeclass called `Arbitrary`.

Since Lava circuits are Haskell functions, using QuickCheck is straightforward. For instance, given Kansas Lava's implementation of boolean conjunction and a suitable instance of QuickCheck's `Arbitrary` type class, one can verify properties like the following, which shows that the `and2` implementation is commutative:

```
prop_andComm x y = (x ‘and2‘ y) == (y ‘and2‘ x)
    where types = (x :: Comb Bool, y :: Comb Bool)
```

While this approach is quick and easy to implement, it is of limited use.

Foremost, equality over sequential inputs is problematic. Two sequences can certainly be unequal, and they can have equivalent prefixes, but they are unbounded data structures, so it is impossible to assert equivalence. We can use a hack to say, effectively, that equivalent prefixes are good enough:

```
prop_andCommSeq x y = prefix (x ‘and2‘ y) == prefix (y ‘and2‘ x)
    where types = (x :: Seq Bool, y :: Seq Bool)
          prefix = (take 100) . toList
```

but this is less than ideal. Nevertheless, an instance of QuickCheck's `Arbitrary` type class is defined for `Seq`, in case it proves useful for other kinds of tests.

Secondly, QuickCheck tests properties by generating random inputs based on type information. While this may be effective for small circuits, larger circuits often require complex non-random input. For instance, we may want to isolate a case where a specific control sequence elicits bad behavior and repeatedly test that case. In essence, we need a unit test instead of a randomized test.

Most problematic for us, as the developers of the Kansas Lava DSL, is the fact that QuickCheck doesn't test the generated VHDL at all. Equality between signal types, over which all Kansas Lava circuits are defined, only compares the simulation values (and in the case of sequences, as mentioned above, doesn't even do that).

It is not obvious how to define equality over the generated code, short of comparing the resulting circuit graphs for isomorphism. The worst case complexity of such a solution is often exponential.

2.1 Observing Intermediate Values

Kansas Lava circuits are opaque Haskell functions which only permit observation of the relationship between inputs and outputs. Intermediate values, defined as local Haskell bindings, cannot be observed without modifying the underlying circuit.

As a simple example, consider this archetypal (and buggy) Lava definition of a full adder, constructed by combining two half adder circuits:

```
fullAdder a b cin = (sum,cout)
  where (s1,c1) = halfAdder a a
        (sum,c2) = halfAdder cin s1
        cout = xor2 c1 c2
```

In this example, the intermediate values `s1`, `c1`, and `c2` are not exposed as outputs to the function, and are consequently not observable. In fact, the above definition has a bug in the calculation of `(s1,c1)` where the input parameter `a` is used as the second argument to the half adder instead of the parameter `b`. Observing the input/output behavior of the `fullAdder` function (with QuickCheck for example) will reveal incorrect behavior, but does not provide any insight into the location of the error. Rather, we need to be able to observe the input/output behavior of the first `halfAdder` *in the context of its use in the full adder.*

One approach to locating the bug is to simply return the intermediate values as additional outputs to the circuit:

```
fullAdder a b cin = ((sum,cout),debug)
  where (s1,c1) = halfAdder a a
        (sum,c2) = halfAdder cin s1
        cout = xor2 c1 c2
        debug = (s1,c1,c2)
```

While this succeeds in exposing the intermediate values, we have changed the interface to the circuit. This necessitates a modification to all the circuits depending on `fullAdder`, and in general leads to a loss of modularity. Moreover, the clarity of the type signature is lost completely:

Before:

```
fullAdder :: (Signal sig) => sig Bool -> sig Bool -> sig Bool
                             -> (sig Bool, sig Bool)
```

After:

```
fullAdder :: (Signal sig) => sig Bool -> sig Bool -> sig Bool
    -> ((sig Bool, sig Bool), (sig Bool, sig Bool, sig Bool))
```

If the incorrect behavior were to be observed when defining a circuit which *uses* the full adder, then that circuit must in turn be modified to propagate the debugging output.

This leaves the user with two options, both with significant drawbacks. They can either export *all* of the intermediate circuit values and sort through the global collection, or they can iteratively export a subset of the intermediate values, checking for correct behavior, until the troublesome circuit is located. The first solution requires changing large amounts of code and muddying function interfaces, the second is incredibly time consuming.

3 Circuit Instrumentation Using Probes

Kansas Lava provides a solution that sidesteps this problem using the notion of a *probe*, based on the design in the Hawk Architectural Design Language [3]. When using this construct, intermediate values can be observed without changing the circuit interface. Only those intermediate values that are probed will be observable, allowing probes to be inserted and removed as the circuit is searched to locate the source of an error.

Using this function, we can instrument the `fullAdder` circuit to expose intermediate values without changing the circuit interface:

```
fullAdder a b cin = (sum,cout)
    where (s1,c1)  = halfAdder a a
          (sum,c2) = halfAdder cin (probe "s1" s1)
          cout     = xor2 (probe "c1" c1) (probe "c2" c2)
```

Two functions are provided for extracting probe values. First, `probeCircuit` takes a circuit with probes and generates an association list of probe names and values. Then `getProbe` looks up a probe name in this list and returns the associated value. The `test` function, shown below, demonstrates the use of these two functions.

```
test = do
    probes <- probeCircuit $ fullAdder false true false
    case (getProbe probes "s1") of
        Just (ProbeValue probeName xstrm) ->
            return $ showXStream xstrm
```

Calling test at the ghci prompt yields the following trace of outputs for the s1 probe:

```
ghci> test
"F" :~  "F" :~  "F" :~  "F" :~ ...
```

The probe mechanism automatically lifts combinational Kansas Lava values into a stream of values.

While the above example demonstrated a way to observe intermediate Kansas Lava values, it doesn't shed a great deal of light on the particular bug in the full adder circuit. We have only observed the output of the halfAdder, whereas the bug is due to the incorrect input value. To make it possible to observe both the input and output of a function, we can apply probe to the *function itself*:

```
fullAdder a b cin = (sum,cout)
  where (s1,c1)  = (probe "ha1" halfAdder) a a
        (sum,c2) = halfAdder cin s1
        cout     = xor2 c1 c2
```

To print out traces from all of the probes, we modify our test function:

```
test = do
    probes <- probeCircuit $ fullAdder false true false
    mapM_ printProbe probes
    where printProbe (i, (ProbeValue name xstrm)) = do
        putStr $ name ++ ": "
        putStrLn $ show xstrm
```

The probe function generates names by simply enumerating each argument (along with the result) and adding the enumerated index to the given probe name, ha1:

```
ghci> test
ha1_0: "F" :~ "F" :~ "F" :~ "F" :~ ...
ha1_1: "F" :~ "F" :~ "F" :~ "F" :~ ...
ha1_2: "(F,F)" :~ "(F,F)" :~ "(F,F)" :~ "(F,F)" :~...
```

In this example, ha1_0 is the first argument to halfAdder, ha1_1 is the second, and ha1_2 is the result. The fact that the two inputs to halfAdder are the same is now obvious.

4 Implementation

4.1 Implementing Probes on Values

There are two concrete types in Kansas Lava: Seq and Comb, which represent sequential and combinatorial values, respectively. Combinatorial values exclude the notion of a clock, whereas sequential values encode a series of values over time:

```
    data Comb a = Comb <shallow value> <deep structure>
    data Seq a = Seq (Stream <shallow values>) <deep structure>
```

Both are instances of the `Signal` type class, over which most primitives are defined.

The shallow embedding is a regular Haskell value which can be manipulated by Haskell functions. The deep embedding is a structure of primitive entities that can be reified to a netlist, from which we generate VHDL for compilation. Kansas Lava primitives manipulate both embeddings, freeing the user to focus on circuit composition.

The ability to observe intermediate values is a pleasing consequence of maintaining both embeddings. In essense, when the `probe` function is invoked on a circuit, the deep embedding is annotated with the result of the shallow embedding.

The `probe` function itself takes a string representing a user-significant name for the intermediate value and a Kansas Lava circuit. To allow `probe` to be utilized for a variety of types, the function is overloaded via a type class and instances are provided for the range of types representable as Kansas Lava circuits.

```
class Probe a where
    probe :: String -> a -> a
```

To allow any Kansas Lava value to be stored in the deep representation, we construct an existential type, `ProbeValue`:

```
data ProbeValue = forall a. (Show a, RepWire a, Typeable a) =>
                            ProbeValue String (XStream a)
                  deriving Typeable
```

The `Typeable` constraint on `a` allows us to use Haskell's `Data.Dynamic` library to store and recover the printed representation of the value. One downside of this implementation decision is that we can no longer manipulate observed values directly, only their string representations. The `XStream` data type can be thought of as a `Seq` that has no deep embedding:

```
data XStream a = XStream (Stream a)
data Seq a = Seq (Stream a) (D a)
```

For `Seq` values, the result of the shallow embedding is simply repackaged into an `XStream` and added to a special attribute list in the deep data structure:

```
instance (Show a, RepWire a, Typeable a) => Probe (Seq a) where
    probe name (Seq shallow (D deep)) =
        Seq shallow (D (addAttr name stream deep))
            where stream = XStream shallow :: XStream a
```

The addAttr function handles the various possible entities that may make up the deep embedding at this point. Using the common case of a Port as an example, we see addAttr create a ProbeValue and add it to the attribute list:

```
addAttr :: forall a . (...) =>
    String -> XStream a -> Driver E -> Driver E
addAttr name value (Port v (E (Entity n outs ins attrs))) =
    let p = [("simValue", toDyn (ProbeValue name value))]
    in  Port v (E (Entity n outs ins $ attrs ++ p))
```

To retrieve the probe values, we use the probeCircuit function, which reifies the circuit and returns an association list of probe names and values:

```
probeCircuit :: (...) => a -> IO [(String,ProbeValue)]
probeCircuit circuit = do
    rc <- reifyCircuit circuit
    return [(n,p) | (_,Entity _ _ _ attrs) <- theCircuit rc
                  , ("simValue", val) <- attrs
                  , Just p@(ProbeValue n v) <- [fromDynamic val]]
```

The reifyCircuit function uses IO-based reification [4] to return a netlist representation of the circuit. Each item is a tuple of a unique id and the Entity from the deep embedding. We search through this list for any attributes that are probes, and recover the ProbeValue.

4.2 Implementing Probes on Functions

To probe a function, we apply probes to each argument as it arrives, and then probe the result value. In order to identify which probe matches which argument, we add a function to our Probe class, which is like probe, but additionally accepts a name supply:

```
class Probe a where
    probe :: String -> a -> a
    probe' :: String -> [Var] -> a -> a

instance (Show a, Probe a, Probe b) => Probe (a -> b) where
    probe name f = probe' name vars f
        where vars = [Var $ show i | i <- [0..]]

    probe' name ((Var v):vs) f x =
        probe' name vs $ f (probe (name ++ "_" ++ v) x)
```

The initial call to probe on a function f will generate a list of names and call probe'. As f is applied to each argument x, that argument has an appropriately named probe wrapped around it. Once the function is fully applied, probe' will be called on a Seq or Comb value. The probe' function for these values calls probe with the annotated name, discarding the name supply:

```
instance (Show a, RepWire a, Typeable a) => Probe (Seq a) where
    probe name (Seq shallow (D deep)) = ...

    probe' name ((Var v):_) seq = probe (name ++ "_" ++ v) seq
```

Using the probesFor function, we can filter the output of probeCircuit to find the specific set of probes related to a probed function:

```
probesFor :: String -> [(String,ProbeValue)]
                    -> [(String,ProbeValue)]
probesFor name plist =
    sortBy (\(n1, _) (n2, _) -> compare n1 n2) $
    filter (\(n, _) -> name `isPrefixOf` n) plist
```

5 Testing the Deep Embedding

As we attempt to use Kansas Lava to generate ever larger circuits, confirming that the shallow and deep embedding are equivalent is increasingly important. Subtle issues like differences in timing behavior between the simulated shallow functions and the VHDL entities often only manifest themselves in larger circuits.

To address this, we use recovered probe values to drive the VHDL simulation and then compare the results. The function that does this is called testCircuit:

```
testCircuit "mux2"
    (mux2 :: Seq Bool -> (Seq U4, Seq U4) -> Seq U4)
    (\ f -> let sel  = toSeq $ cycle [True,False,True,True,False]
                inp  = toSeq $ cycle [0..15]
                inp2 = toSeq $ cycle $ reverse [0..15]
            in f sel (inp, inp2))
```

The implementation of testCircuit takes a user-significant name, the circuit, and a function that applies that circuit to inputs:

```
testCircuit :: (...) => String -> a -> (a -> b) -> IO ()
testCircuit name circuit apply = do
    let probed = probe name circuit
    plist <- probeCircuit $ apply probed

    mkInputs name 50 $ probesFor name plist
    mkTestbench name probed
```

First, testCircuit wraps a probe around the circuit. Since the circuit is a function, that means both inputs and outputs will be observed. Next, the apply function applies the sample input provided by the user to this probed circuit. Using probeCircuit and probesFor, the probe data is recovered.

The mkInputs function transforms the XStream values in each probe into two ASCII files. One is a human readable info file which gives a clock value followed by each input and the output, in both Haskell and wire representations. The other file is meant to be read by the VHDL testbench:

```
mux2.info                                      mux2.shallow
(0) T/1 -> (0,15)/11110000 -> 0/0000    1111100000000
(1) F/0 -> (1,14)/11100001 -> 14/1110   0111000011110
(2) T/1 -> (2,13)/11010010 -> 2/0010    1110100100010
(3) T/1 -> (3,12)/11000011 -> 3/0011    1110000110011
(4) F/0 -> (4,11)/10110100 -> 11/1011   0101101001011
(5) T/1 -> (5,10)/10100101 -> 5/0101    1101001010101
...                                            ...
```

Running the VHDL testbench created by `mkTestbench` will generate a `mux2.deep` file that corresponds to the `mux2.shallow` file. If these two files are the same, then the simulated and compiled versions of the circuit behaved in the same way.

6 Handling Large Circuits

The framework we just described suffers because it only uses the probe data for the overall circuit. This is fine when the circuit is small, but becomes problematic for larger circuits because, while it effectively reports that something is wrong, it offers no help in pinpointing the bug.

The user can place many probes within a single circuit, so our framework should be able to generate tests for each of them. This allows the user to narrow down the problem by finding probe tests that fail within the larger circuit.

6.1 Extracting Subcircuits

Whereas probing intermediate values is a useful analogue to actual hardware probes, effectively allowing the user to watch a stream of values move along a wire in the circuit, probing functions is much more powerful. Since each input to the function is itself probed, along with the output, we have effectively tagged the boundaries of the function within the circuit graph.

To illustrate, reifying our probed `fullAdder` gives the graph in Fig. 1. Notice that nodes are annotated with the names of probes applied to them. To find the nodes that make up the `halfAdder` function named `ha2`, we start at its output node (`ha2_2`, the highest numbered `ha2` probe) and do a breadth first search (BFS) backward along the input edges. If we encounter a node that also has an `ha2` probe on it, then we have reached one of the arguments to the function. This marks the boundary of the function within the graph. Nodes encountered during this search make up the subgraph that implements the `halfAdder` function.

Using this subgraph, we can create a new sink based on the output type of the `ha2_2` node. The leaves, which all have probes on them, are sorted in argument order based on the probe name. Each leaf is replaced with a `Pad` (an externally driven input), using the output type of the leaf as the new input and output type of the `Pad`. This allows us to pass the captured probe data as input. The newly extracted self-contained circuit can be seen in Fig. 2.

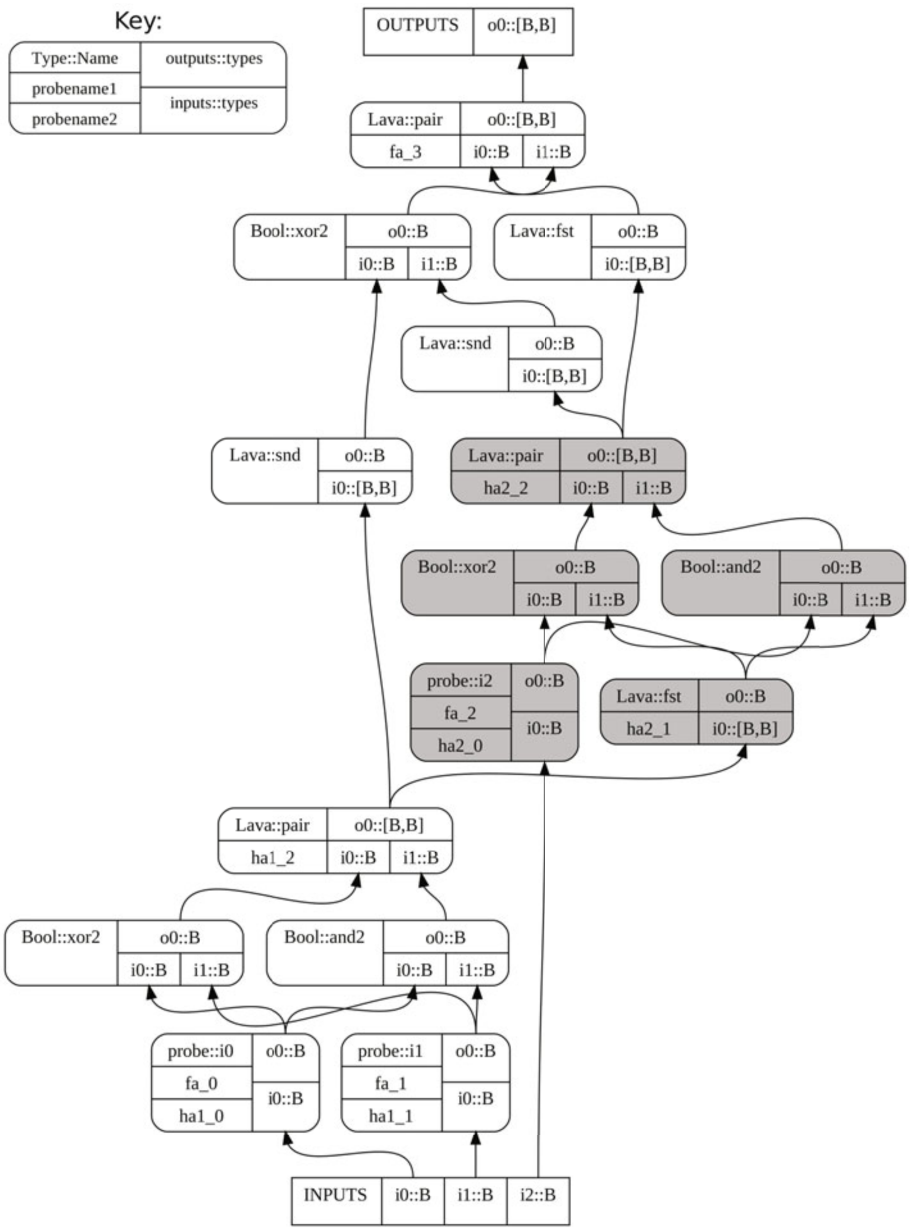

Fig. 1. A circuit graph for the `fullAdder`. Note that we have attached probes to each `halfAdder` as well as the entire circuit. Using a BFS backwards along the inputs of the `ha2_2` node, we find the subcircuit making up the second probed `halfAdder`. These nodes are shaded in gray.

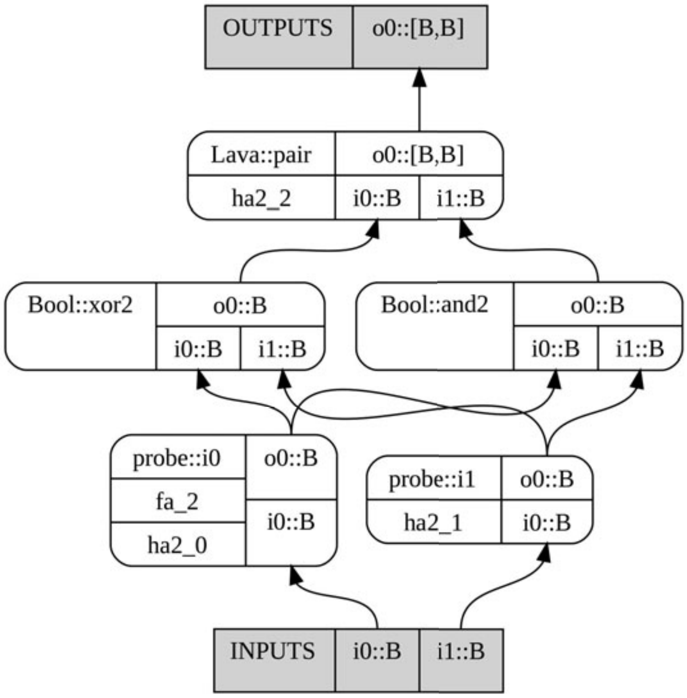

Fig. 2. The circuit for the `ha2` probe, extracted from the `fullAdder` circuit in Fig. 1. Leaf nodes have been converted to `Pads`, and new sources and sinks are derived based on type information.

We provide a function named `extract` to implement this algorithm. It takes a probe name and a reified circuit and returns a reified subcircuit which implements the probed function:

```
extract :: String -> ReifiedCircuit -> ReifiedCircuit
```

Using `extract`, we can write a more versatile version of `testCircuit` which allows us to specify *which* probed function we would like to test in the context of the overall circuit:

```
testSubcircuit :: (...) => String -> a -> (a -> b) -> IO ()
testSubcircuit name circuit apply = do
    let probed = probe "whole" circuit

    reified <- reifyCircuit probed
    plist <- probeCircuit $ apply probed

    mkInputs name 50 $ probesFor name plist
    mkTestbenchFromRC name $ extract name reified
```

6.2 Locating Errors Automatically

The ability to extract and individually test subcircuits allows us to liberally probe a circuit and use our intuition to locate the cause of a problem.

However, since we are testing the correspondence of the shallow and deep embeddings, which comes down to diffing the generated outputs, there is no reason not to automate this process.

To do so, we implement a form of algorithmic debugging [5]. The general idea is to build an execution tree, and for each node ask an oracle if the result at that node is correct, eventually finding the node that is causing the error. Normally the oracle is the programmer, but in this case we can use the recorded probe data, effectively making the shallow embedding an oracle for the deep embedding.

We do this by walking the circuit graph in breadth first order from the sinks, recording the relationship among the various probes in the circuit into a simple Rose Tree structure that records the name of the probe, the node's unique identifier, and a list of children:

```
data ProbeTree = Node String Unique [ProbeTree]
    deriving (Eq, Show)

probeForest :: ReifiedCircuit -> [ProbeTree]
```

In the context of a hardware circuit, our execution tree must capture the *contained-in* relationship between probed functions (as opposed to the *depends-on* relationship). This encodes the failure relationships between probed subcircuits. If subcircuit `A` is wholly contained by subcircuit `B`, then `A`'s failure will most likely also cause `B` to fail. As such, `A` is a child of `B`. Otherwise, `A` and `B` are siblings. For the `fullAdder` example, `probeForest` returns:

```
[Node "fullAdder" 3 [Node "ha2" 5 [],Node "ha1" 9 []]]
```

Note that while `ha2` depends on the output of `ha1` in the circuit, `ha1` is not contained *within* `ha2`. A failure within `ha1` will not necessarily lead to a failure of `ha2`, merely bad input. Both are within the `fullAdder` subcircuit, whose failure could be caused by either, meaning `ha1` and `ha2` are siblings.

Now that we have this tree structure, we use it to implement our debugger. There are various strategies for traversing the execution tree. Most of these focus on improving the experience for human oracles by reducing the number nodes tested and avoiding dramatic context switches that can slow the oracle down. Since we are using the shallow embedding as our oracle, we chose a strategy known as Top-Down Search [6]. While not the most advanced strategy available, it is both simple to implement and effective at pruning the execution tree.

Beginning at the root, we call a modified `testSubcircuit` on each node to extract and compare the deep and shallow embeddings of the probed subcircuit. If the test fails, we start testing each child, on the premise that the failure was caused by a failing child. If the test succeeds, we assume all child subcircuits functioned properly (since they are part of the circuit we just tested), and move on to a sibling in the tree:

```
algDebug :: ReifiedCircuit         -- circuit
         -> [(String, ProbeValue)] -- probe data
         -> IO ()
algDebug circuit pdata = go "" $ probeForest circuit
    where go []     [] = putStrLn "Embeddings act the same."
          go parent [] = putStrLn $ parent ++ " failed."
          go parent ((Node name _ children):siblings) = do
                code <- testSubcircuit circuit pdata name
                case code of
                    ExitSuccess -> go parent siblings
                    ExitFailure _ -> go name children

testSubcircuit :: ReifiedCircuit
               -> [(String, ProbeValue)]
               -> String
               -> IO ExitCode
testSubcircuit circuit pdata name = do
    mkInputs name 50 $ probesFor name pdata
    mkTestbenchFromRC name $ extract name circuit
```

The search terminates in two ways:

- All siblings at the top level test successfully, meaning all probed portions of the circuit are behaving equally in both embeddings.
- All siblings at another level test successfully, meaning their parent node (which failed) is the likely culprit.

As you can see, each run of this algorithmic strategy only locates a single failing subcircuit, so we may be required to run the search multiple times if there are multiple errors.

We change our `testCircuit` definition once again to take advantage of this automated search:

```
testCircuit :: (...) => String -> a -> (a -> b) -> IO ()
testCircuit name circuit apply = do
        let probed = probe name circuit

        rc <- reifyCircuit probed
        pdata <- probeCircuit $ apply probed

        algDebug rc pdata
```

7 Related Work

There are many ongoing efforts to create effective testing and debugging tools for lazy functional languages. As we have seen, QuickCheck is one of those tools. Others include HPC, a code coverage tool [7], and ThreadScope, a means of visualizing parallel computations [8].

Probes are a means of tracing, a well explored area when it comes to debugging functional languages. Chitil *et al.* compare three popular solutions for tracing Haskell program execution [9], including Freja [10], Hat [11], and Hood [12]. Our probe implementation is most like that of Hood in spirit. Both are lightweight tools that allow observation of intermediate values without greatly impacting performance. Freja and Hat are extensions to Haskell compilers, and each offers a guided traversal of the trace information to locate bugs, much like our efforts in Section 6.2. Hat in particular is known to have a large runtime overhead.

Algorithmic debugging is a promising approach to automating debugging tasks [5]. Functional languages appear to be a good match for this technique due to their lack of side effects. Laziness, however, presents a problem in that values presented to the user might not be evaluated yet. Nilsson and Fritzson make an in-depth examination of algorithmic debugging in the context of lazy functional languages [13].

8 Conclusion and Future Work

Traditional Haskell testing tools like QuickCheck are of limited use when testing Lava circuits. Many properties that are easy to express over finite data structures like `Comb` are more cumbersome over unbounded ones such as `Seq`. They also offer no easy way to test the generated VHDL code.

Probes present a method of observing intermediate values in the shallow embedding without modifying the circuit interface. Their use is intuitive in the to a hardware designer accustomed to thinking of wires and observing waveforms. The dual shallow/deep embedding used by Kansas Lava signals is crucial for their implementation. In order to test VHDL generation, they permit an automated comparison of the deep and shallow embeddings.

While the current system is primarily useful to the developers of Kansas Lava, one possible future direction is to adapt the framework to be a true algorithmic debugger, with the Lava user as the oracle. Alternatively, this framework could be used to test circuit optimizations, using the unoptimized circuit as an oracle for the optimized one.

The full implications of the ability of probes to bound and extract subcircuits also remains to be explored. We can envision a hybrid execution model, running some parts of the circuit in hardware while simulating others using the shallow embedding. Efforts to visualize circuits can also be greatly improved. Large circuits may contain millions of nodes, but probes would allow us to group related parts of the circuit and view it at a more abstract level.

References

1. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in haskell. In: International Conference on Functional Programming, pp. 174–184 (1998)
2. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. ACM SIGPLAN Notices, 268–279 (2000)
3. Matthews, J.R.: Algebraic Specification and Verification of Processor Microarchitectures. PhD thesis, University of Washington (1990)
4. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing Kansas Lava. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 18–35. Springer, Heidelberg (2010)
5. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
6. Av-Ron, E.: Top-Down Diagnosis of Prolog Programs. PhD thesis, Weizmanm Institute (1984)
7. Gill, A., Runciman, C.: Haskell Program Coverage. In: Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell. ACM Press, New York (2007)
8. Jones Jr., D., Marlow, S., Singh, S.: Parallel performance tuning for haskell. In: Haskell 2009: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, pp. 81–92. ACM, New York (2009)
9. Chitil, O., Runciman, C., Wallace, M.: Freja, hat and hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 176–193. Springer, Heidelberg (2001)
10. Nilsson, H.: Declarative Debugging for Lazy Functional Languages. PhD thesis, Linköping, Sweden (May 1998)
11. Claessen, K., Runciman, C., Chitil, O., Hughes, J., Wallace, M.: Testing and tracing lazy functional programs using quickcheck and hat. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638. Springer, Heidelberg (2003)
12. Gill, A.: Debugging haskell by observing intermediate data structures. In: Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell, Technical report of the University of Nottingham (2000)
13. Nilsson, H., Fritzson, P.: Algorithmic debugging for lazy functional languages. Journal of Functional Programming 4, 337–369 (1994)