On embedding a microarchitectural design language within Haskell

John Launchbury Jeffrey R. Lewis Byron Cook

Oregon Graduate Institute of Science & Technology

Abstract

Based on our experience with modelling and verifying microarchitectural designs within Haskell, this paper examines our use of Haskell as host for an embedded language. In particular, we highlight our use of Haskell's lazy lists, type classes, lazy state monad, and `unsafePerformIO`, and point to several areas where Haskell could be improved in the future. We end with an example of a benefit gained by bringing the functional perspective to microarchitectural modelling.

1 Introduction

There are many ways to design and implement a language. Landin's vision of the next 700 programming languages [20], for example, was to build domain-specific vocabularies on top of a generic language substrate. In the verification community, this is known as a *shallow embedding* of one language or logic into another. In effect, every abstract data type defines a language. Admittedly, most abstract data types by themselves make impoverished languages, but when interesting combinators are provided, the language becomes rich and vibrant in its own right. This explains the continuing popularity of combinator libraries, from the time of Landin until now.

The animation language/library Fran is a beautiful example [11, 10]. Fran provides two families of abstract types in Haskell: *behaviors* and *events*. To construct a term of type `Behavior Int`, for example, is to write a sentence in the Fran language, using Fran primitives and Fran combinators. To build complex Fran entities, however, the full power of Haskell can be brought to bear. Fran objects are just another abstract data type.

So how good is Haskell as a host for embedded languages? This is one of those questions that can only be answered through experience, and is precisely where we see the contribution of this paper. We describe our use of Haskell as a host for a microarchitectural modelling language, calling attention to the aspects of Haskell that helped us, those that hindered us, and the features we wish we had. In particular, we highlight our use of Haskell's lazy lists, type classes [18], the lazy state monad [21], and `unsafePerformIO` [19]. This paper contains no deep theory, but rather a dose of measured introspection.

The remainder of this paper is organized as follows: In Section 2 we provide the motivation for our work in microarchitectural modelling. In Section 3 we introduce Hawk and show how we use lazy lists to model wires. In Sections 4, 5, and 6, we show how type classes, the lazy state monad, and `unsafePerformIO`, respectively, are put to use in Hawk, and in Section 7 we describe an application that makes use of all four features. In Section 8 we outline where Haskell has constrained us, and discuss future directions. Finally, the paper closes with an example of some new insights into microarchitectures that arose as a consequence of the functional perspective.

2 Building a Microarchitectural Description Language

Contemporary superscalar microarchitectures employ tremendously aggressive strategies to mitigate dependencies and memory latency. Their complexity taxes current design techniques to the limit. The trend continues as the size of design teams grows exponentially with each new generation of chip.

To gain an appreciation for the complexity of modern microarchitectures, take as an example the model of an *instruction reorder buffer* which occurs frequently in out-of-order microprocessors like the Pentium III. The purpose of the instruction reorder buffer is to allow instructions to be executed at the earliest possible moment. It does this by maintaining a pool of instructions, so that it can dynamically determine which of them are eligible for execution by keeping track of whether their operands have been computed. Furthermore, instructions are introduced speculatively, based upon numerous successive branch predictions. Consequently, instructions that have previously been scheduled and executed must sometimes be rescinded when a branch is discovered to have been mispredicted. Thus the instruction reorder buffer must keep track of instructions up to the point that they can either be retired (committed) or flushed. Since some instructions following a branch may already have been executed when a branch misprediction is discovered, register contents are also affected. At a branch misprediction, register mapping tables must be modified to invalidate the contents of registers that contain results of rescinded instructions. The contents of registers that are possibly live must be preserved until after the branch has

ICFP '99 9/99 Paris, France

been resolved, thus increasing the complexity of the interaction between a instruction reorder buffer and the registers.

In addition, there are all the issues of managing on-chip resources, of ensuring rapid and correct communication of results, of cache coherence and so on. It will get worse. The next generation of microarchitectures will address many more issues such as explicit instruction parallelism [14] and multiple instruction threads [35].

As if all these algorithms did not provide enough design complexity, commercially viable microarchitectures are also subject to legacy requirements. For example Intel's Pentium III must deal with dozens of exception types to remain compatible with earlier versions of the X86 architecture. Pentium III also struggles with the variable length of X86 instructions. It tries to fetch three each cycle, and it turns out that dynamically determining the length of instructions before decoding is one of Pentium III's primary performance bottlenecks. Again, this type of problem is not going to go away. Intel's recently announced Merced processor executes not only its new instruction set [9], but X86 code as well [13].

With designs of this complexity, it is hard to imagine that designers will not stumble upon subtle interaction and concurrency bugs. The need for powerful and effective modelling and verification has never been greater. By couching microarchitecture modelling in terms of higher-level abstractions and emphasizing the modularity of a design it is possible to regain control of the design space. This is what we have done. Influenced by discussions with Intel's Strategic CAD Laboratory, we have developed *Hawk* as an executable modelling language embedded in Haskell [1]. Hawk is very high level compared with other hardware description languages. Consequently, even complex microarchitecture models remain remarkably brief, allowing designers to retain a high level of intellectual control over the model. For example, the complete formal model of a speculative, superscalar, out-of-order microarchitecture based on the Pentium III required less than 1000 lines of code [5].

3 Lazy Lists: Signals in Haskell

We intend for signals to model values that change over time, like wires in a microprocessor. Following O'Donnell [28], Srivas & Bickford [34], and many others, we implement signals as lazy lists. The idea is simple: the n^{th} element of the list represents the value of the wire at clock tick n. Thus the value of each wire is a complete description of its behavior over time. This approach leads to circuit semantics with a definite denotational flavor. In contrast, state transition systems (another popular style) are much more operational in their nature. There are naturally advantages and disadvantages to each.

To represent units with clocked inputs and clocked outputs we use functions from signals to signals, known as stream transformers. Combinational circuits can be turned into clocked circuits simply by mapping them down their input lists. If `add::(Int,Int)->Int` is a simple addition circuit, then `map add :: [(Int,Int)] -> [Int]` is its clocked equivalent.

The fundamental non-combinational circuit is the *delay*. The delay is what makes feedback loops in clocked circuits possible—without any delays, a feedback loop would just generate smoke! A delay is defined so that the $(n+1)^{st}$ element of the output is equal to the n^{th} element of its input, with an initial value output for the very first clock tick. The implementation of `delay :: a -> [a] -> [a]` is simply "cons".

Some care is needed within this paradigm, however. Arbitrary use of list processing functions, especially those which add or discard elements, can cause problems in that they fail to model hardware. For example, a function which duplicates every element on its input list would require an infinite buffer to implement in hardware. To restrict the way in which a signal can be constructed or altered, we make the signal type abstract in Hawk and provide a basic set of manipulation functions that are known to be safe for the model.

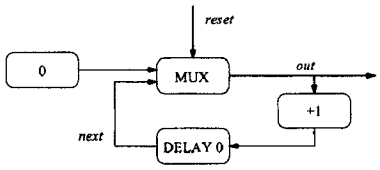
```
newtype Signal a

delay :: a -> Signal a -> Signal a
lift0 :: a -> Signal a
lift1 :: (a -> b) -> Signal a -> Signal b
      ::    .
  .   ::    .
  .   ::    .
```

The `lift` family of functions lifts n-ary (curried) functions to clocked functions over signals. The base case is `lift0`, which returns a constant signal, and `lift1` is just the map function for lists. Later we will use the derived operator `bundle`, which takes a pair of signals, and produces a signal of pairs.

Restricting access to the implementation in this way gives the usual freedoms to provide alternative implementations, or even to refine the semantics somewhat. For example, rather than using lazy lists, we could implement signals as functions from the natural numbers to values.

If the above signature seems to be missing something, it is. The rest comes from Haskell itself, in particular, lazy recursive definitions. You could say that the missing operator of the abstract type is a lazy fixpoint operator. Consider a resettable counter circuit like:



which, in Hawk, we might model as:

```
counter :: Signal Bool -> Signal Int
counter reset = out
  where
  next = delay 0 (lift1 (+1) out)
  out  = mux reset (lift0 0) next
```

The mutual recursion between signals allows for arbitrary looping in a circuit. Microprocessor models have many such feedback loops, at many different levels of specification. Note that the laziness of Haskell is vital for this mutual recursive definition to have the intended meaning. It is not merely the laziness of signals themselves that is required, but also the laziness of the definitions. Even if signals are known to be lazy, a strict language would (by default) attempt to evaluate the uses of `out` and `next` in the right-hand-sides of the definitions, leading to non-termination. This distinction between lazy structure and lazy definition is brought out

well by Okasaki and Wadler in their respective methods for adding laziness to Standard ML [29, 37]. We can summarize the principle as follows: (mutually) recursive definitions of an abstract data type require lazy definitions. This principle holds even if the abstract datatype is implemented by a function so that no lazy data structures are actually involved.

One item that is not missing from the signal definition is a way to observe a list by taking its head or tail. This is intentional. A circuit that was specified to take the tail of a list would be asking for a circuit to perform lookahead in time. We *do* allow signals to be viewed as lists for the purpose of viewing simulation results, but this operation is only provided for use at the top-level.

4 Microarchitectural Abstractions

Two of the goals of Hawk have been to build abstractions that increase the concision of microarchitectural models [5], and to facilitate the verification process [25]. For microarchitectural abstractions to be relevant, they must be extraordinarily flexible in the types that they operate over. Instruction sets differ in variety of details: size and type of data, number and types of registers, and the instructions themselves. Internally, machines may use other instruction sets. For example, the AMD K6[33] implements the X86 instruction set, but uses a RISC instruction set within its execution core.

We use type classes to facilitate the description of circuits that operate over all instruction sets. For example, the type of a primitive ALU might be:

```
alu :: (Instruction i, Word w) =>
  (Signal i, Signal w, Signal w) -> Signal w
```

This way, `alu` can be used in an X86 model (where `w` is set to 32-bit words and `i` to X86 instructions) or a 64-bit RISC instruction set, like that of the Alpha. The `Word` class is an extension of Haskell's `Num` class that adds operators related to word size, signedness, etc. The `Instruction` class captures the common elements between instruction sets.

With common architectural characteristics captured by type classes, we are then able to build abstractions that help organize microarchitectural models. For example, *transactions* [2, 27] are a simple yet powerful grouping of control and data. A transaction is a machine instruction grouped together with its current evaluation state. This state might include:

- Operand and result values.
- A flag indicating that the instruction has caused an exception.
- A predicted jump target, if the instruction is a branch.

It seems a trivial thing to do, when building multiple component values are so easy in functional languages, yet it had significant consequences. For example, we found that microarchitectural models that utilize transactions can make decisions locally rather than with a separate control unit, and to a large extent, definition of local control is far easier to get right than attempting the same task globally.

To get a feel for transactions, consider the following example. Suppose the instruction fetch unit issues an instruction that Registers 1 and 2 are to be added and the result placed in Register 4, that is, `"r4<-r2+r1"`. The initial transaction corresponding to this would lack values for each of this registers, i.e. `"(r4,_)<-(r2,_)+(r1,_)"`. As the transaction passes through the register file, its operand values are filled in: `"(r4,_)<-(r2,4)+(r1,4)"`. After the ALU, the computed result is also filled in: `"(r4,8)<-(r2,4)+(r1,4)"`, and now the transaction is ready to go back to the register file to store the result.

Hawk provides a library of functions for creating and modifying transactions. For example, `bypass` takes two transactions and builds a new transaction where the values from the destination operands of the first transaction are forwarded to the source operands of the second. If `i` is the transaction:

```
"(r4,8) <- (r2,4) + (r1,4)"
```

and `j` is the transaction:

```
"r10 <- (r4,6) + (r1,4)"
```

then `bypass i j` produces the transaction:

```
"r10 <- (r4,8) + (r1,4)"
```

That is, `bypass` inserts `i`'s more recent valuation of `r4` into the destination operand of `j`.

The bypass function is an example of a local control operator. The control function it performs is selective forwarding of newly computed results to other instruction transactions that may otherwise contain stale information.

```
bypass :: (Word w, Register r) =>
  Trans i r w -> Trans i r w -> Trans i r w
```

By parameterizing over the instances of finite words and registers, `bypass` can be used in many contexts. Within our Pentium III-like microarchitectural model we use `bypass` on instructions with both concrete register references and virtual register references (which arise as a result of dynamic register renaming for the out-of-order core of the processor). Both types of register are instances of the type class `Register`. In our Merced-like model [6], we use the same `bypass` with IA-64 instructions.

5 Lazy State: Using State-Based Components

There has been debate in the Haskell community about the merits of laziness/strictness within the state monad. In this section we describe an application where lazy state is just right [21].

Some microarchitectural components, such as register files, are more naturally (and efficiently) presented as state transition systems than as list transformers. For example, imagine modelling a primitive register file as an array which, on each clock tick, is both written to and read from. Here it is, using the basic idiom of lazy state, done first with explicit lazy-lists to show the recursion structure.

```
regFile :: [(Addr,w)] -> [Addr] -> [w]
regFile writes reads
  = runST (
    do { reg <- newSTArray (minAddr, maxAddr)
                           (error "uninitialized")
       ; regLoop reg writes reads
       }
    )
```

```
regLoop :: STArray s Addr w ->
           [(Addr,w)] -> [Addr] -> ST s [w]
regLoop reg ((a,w):aws) (r:rs)
   = do { writeSTArray reg a w
        ; v <- readSTArray reg r
        ; vs <- regLoop reg aws rs
        ; return (v:vs)
        }
```

As with both versions of encapsulated state, the state within the scope of `runST` is completely hidden from the outside world. Thus as far as the rest of the program is concerned, `reg` is completely pure, as indicated by its type. The encapsulation of the state is guarenteed by the type of `runST` [23]. Inside the implementation of `regFile`, however, the situation is quite different. The array writes are "imperative", a constant-time operation having effects immediately visible to subsequent reads.

The semantics of lazy state is as follows. The monadic structure sequentializes the operations of the monad but *forces nothing*. When the result of the state thread is demanded (in this case, the output list of values), execution proceeds to meet the demand, but in the order determined by the monadic sequentialization. Thus, while execution proceeds by demand, some of that demand is transmitted through the state sequencer. As more and more of the result signal is demanded though execution of the rest of the Hawk model, so a larger and larger prefix of the sequence of state instructions are executed. Laziness with respect to later state operations is essential here: the computed value `v` must be made available to the outside world before the recursive call to `regLoop aws rs` is performed.

To recast this in the context of Hawk abstract signals is straightforward. Within the definition of signals, we introduce a new family of functions `liftST` n, which are the monadic map on signals. For example:

```
liftST2 :: (a -> b -> ST s c) ->
           Signal a -> Signal b -> ST s (Signal c)
```

The corresponding Hawk definition of the register file is as follows:

```
reg :: Register r =>
       Signal (r,w) -> Signal r -> Signal w
reg writes reads
  = runST (
    do { reg <- newSTArray (minReg, maxReg)
                           (error "uninitialized")
       ; liftST2 (regFile reg) writes reads
       })

regFile ::Register r => STArray s Addr w ->
          (r,w) -> r -> ST s w
regFile reg (a,w) r
   = do { writeSTArray reg a w
        ; readSTArray reg r
        }
```

In the use of `liftST2` above, the state machine is executed step by step, consuming its list input and generating its list output on the way. In particular, the `liftST` construct does not attempt to execute the state machine completely before releasing the output list. It is this behavior we *require* of the state monad and fortunately, though not officially a part of Haskell, most implementations provide it.

6 Use and Abuse of unsafePerformIO

When embedding a language, one often needs "language primitives" that provide good things but could not be defined directly. Fran for example, has a function :

```
importBitmap :: Filename -> Bitmap
```

which imports a bitmap file and treats it as a pure value.

There are two basic approaches to defining this kind of primitive. The first is to write code in C, and add it as a new primitive in the run-time system of the host language. The alternative is to provide the host language with a generic, though potentially unsafe, mechanism of writing new primitives, and to make clear what extra proof obligations arise that make its use predictable.

In this vein, most Haskell implementations provide an implementor's function `unsafePerformIO :: IO a -> a` which performs an IO operation and then casts the result as a pure value. The Fran function `importBitmap`, for example, is defined in this way. The action of reading a bitmap file is performed, and then `unsafePerformIO` is used to treat the bitmap as a pure value.

As its name suggests, `unsafePerformIO` is potentially unsafe. By abusing it one can do all manner of bad things. But under the alternative scenario of hacking the run-time system in C, one can also do all manner of bad things. The question is, which is worse? Providing the extension mechanism at the source language level avoids large classes of errors that could otherwise arise from mangling the run-time system, and works uniformly across many language implementations. Over the last few years, a fairly strong consensus has emerged that if extra primitives are needed they might as well be defined at source language level through a judicious use of a mechanism like `unsafePerformIO`.

However, because it does extend the primitive base of Haskell, there is a clear sense in which any use of `unsafePerformIO` means that the resulting program is no longer written in Haskell per se, but rather in some extension to Haskell. Thus, properties that apply to all Haskell programs, may cease to apply to programs written in poorly defined extensions. It is not just the delicate properties, like parametricity for example, that are at risk, but even basic properties like referential transparency and type safety. For example, `unsafePerformIO` is strong enough to allow the definition of a new primitive function `cast`:

```
cast :: a -> b
cast x = let bot = bot
             r = unsafePerformIO (newIORef bot)
         in unsafePerformIO
              (do {writeIORef r x; readIORef r})
```

The use of `unsafePerformIO` resurrects the original ML-reference problem. The reference `r` is unconstrained at creation, and the use of `unsafePerformIO` allows it to be bound by a `let`-construct, and so has its type generalized. It can store or retrieve values of any type. Thus there is no problem storing a value of type `a` nor of reading a value of type `b`, even though precisely the same value will be written and read! Incidentally, avoiding exactly this problem (amongst others) lead to the careful use of parametricity in the definition of runST [23].

All is not lost, however. There are many examples of careful uses of `unsafePerformIO` that extend Haskell in ways entirely consistent with its underlying philosophy. We give one below.

6.1 Observing Signals

When using Hawk, we found that we often wanted to observe the values flowing across a signal. Unfortunately, Haskell's semantic purity makes this viewing rather difficult, as viewing a signal often implied recoding the model so that the stream we were interested in was available at the top level. As an alternative, we provide the function:

```
probe :: Filename -> Signal a -> Signal a
```

As far as Hawk-level models are concerned, a probe is simply the identity function on signals. However, the external world receives a different view. Probes are side-effecting, writing values to a file, even though they apparently have a pure type. Thus, probes cannot be defined within Haskell-proper. Instead, they need to be introduced as a Haskell extension through the use of `unsafePerformIO`.

```
probe name vals =
   lift2 (write name) clock vals

write name tick val = unsafePerformIO
   do { h <- openFile name AppendMode
      ; hPutStrLn h (show tick ++ ":" ++
                     show val)
      ; hClose h
      ; return val
      }
```

(`clock` is a stream that enumerates the natural numbers.) Notice that we are careful not to change the strictness of the argument stream of `probe`. Each element of the list is wrapped in an independent side-effecting closure which, when evaluated, writes its value to the file required, and then returns the value. This definition makes essential use of the strictness of the IO monad, in contrast to the laziness of the ST monad earlier. Without strictness, the final value would simply be returned, with none of the effects having been performed.

Because the Hawk models do not depend on the contents of the filestore, we can guarantee that a model is unchanged by the addition of probe functions.

We went much further than just writing the probe information to a file. We used the commercial drawing package Visio to build a front end to Hawk. We can now draw diagrams in Visio and then, at the push of a button, generate a corresponding Hawk model containing one probe function per wire on the diagram. During and after the execution of the model, double-clicking on any wire causes the corresponding probe file to be opened, displaying the contents of the wire. This provided an invaluable feedback tool for debugging microarchitectures.

In summary, we found `unsafePerformIO` to be a powerful facility for building tools to observe but not affect the microarchitectural models.

7 Verification in Hawk

We wanted Hawk to provide tools that can be used to formally verify properties of microarchitectural models. Suppose, for example,that we want to prove the following properties about the resettable counter from Section 3:

1. When the reset line is low on the next clock cycle, the output is the value at the current cycle plus 1;
2. When the reset line is high at the current clock cycle, the output is zero.

In Hawk, we might express these properties as follows. Assume that `r0` and `r1` are the values of the reset line at time t and $t+1$ respectively, and that n and m are the corresponding integer outputs from the circuit.

```
propCounter r0 r1 n m = prop_one && prop_two
  where
    prop_one = not r1 ==> (n + 1 === m)
    prop_two = r0 ==> (n === 0)
```

We would like to show that these properties hold for arbitrary values of `r0` and `r1`, and for arbitrary values of the internal state element of the counter circuit. To do this, we will use symbolic values for `r0` and `r1`, and symbolically simulate the circuit.

The approach we take to symbolic simulation is a straightforward application of polymorphism and overloading, given in more detail elsewhere [8]. We introduce a datatype of symbolic expressions (variables and additional term structure). For example, we have used the following datatype for symbolic simulation of simple arithmetic circuits.

```
data Symbo a =
       Const a
     | Var String
     | Plus (Symbo a) (Symbo a)
     | Times (Symbo a) (Symbo a)
```

Sufficiently polymorphic functions that arise in a Hawk model can be instantiated at new types and at the symbolic type `Symbo` in particular. The catch is that some care is required in making functions "sufficiently polymorphic". In brief, the parts of the program that you wish to symbolically evaluate cannot use concrete types, because those types must be able to be replaced by symbolic counterparts.

7.1 Symbolic Simulation in Haskell

In places, Haskell's prelude is remarkably amenable to symbolic simulation. Take the `Num` class, for example. As almost every numeric operator is overloaded, so too are the vast bulk of numeric expressions. Thus to symbolically execute a numeric expression, all we have to do is declare an instance of class `Num` over the `Symbo` type.

```
instance Num a => Num (Symbo a) where ...
```

Now any numeric expression is immediately symbolically executable.

In other places Haskell's prelude is not so amenable to symbolic simulation. Booleans provide an excellent example. Comparison and conditional operations in Haskell's prelude have booleans hardwired in place. The historical reason is clear. Overloading in Haskell was introduced precisely because the designers of the language already had many different versions of numbers that they wanted to add and multiply (integer, rational, floating point, complex, etc.), but only one version of booleans: simple `True` and `False`. However, there are more varieties of booleans that we are now coming across, particularly in the realm of embedded languages. For example, Fran needs to be able to compare expression that vary with time, leading naturally to the concept of a boolean result that also varies with time. In our context we

want the boolean operations to apply to symbolic expressions representing booleans.

To capture the operations of both concrete and symbolic booleans we echo the development of the Num class, and define a class Boolean, which makes all the boolean operators from the prelude abstract:

```
class Boolean b where
    true :: b
    false :: b
    (&&) :: b -> b -> b
    (||) :: b -> b -> b
    (==>) :: b -> b -> b
    not :: b -> b
```

We also define a class Eql, which is similar to the standard Eq class, except that it is also abstracted over equality's result type.

```
class Boolean b => Eql a b where
    (===) :: a -> a -> b
```

Conditional expressions, too, must be abstract:

```
class Mux c a where
    mux :: c -> a -> a -> a
```

If the condition on which we branch is symbolic, it is clear that the result must be symbolic as well. Hence there is a relationship between the type of the conditional, and the type of the result—just the sort of thing that multi-parameter type classes express well.

To capture the common usage of conditional expressions, we make Bool an instance of Mux

```
instance Mux Bool a where
    mux x y z = if x then y else z
```

Of course, we also make signals of boolean-like things instances of the Mux class.

We can now employ many implementations of Booleans. In particular we can use binary decision diagrams (BDDs) [4], which implement semantic equality between symbolic boolean expressions in constant time. Using H/Direct [12] and unsafePerformIO, we have imported the CMU BDD package into Haskell [7]. In the style of the Voss modelling language [31], Hawk treats BDDs just like Booleans. But, thanks to type classes, a user can also choose *not* to use BDDs, but some other instance of Boolean.

7.2 Proving a Property

We now have the infrastructure needed to verify our properties. Our strategy is to simulate the counter with symbolic values on the reset line for the first two ticks, and then test the desired property on the first two outputs. To ensure the result applies at any stage of the execution we also need to be able to initialize the state element (the delay component) of the counter by placing a symbolic value there as well. The new definition of counter is as follows:

```
counter :: (Num a, Boolean b) =>
           a -> Signal b -> Signal a
counter init reset = out
  where
  next = delay init (lift1 (+1) out)
  out  = mux reset (lift0 0) next
```

We can use this definition directly in verification of the property:

```
test :: BDD
test = propCounter r0 r1 n m
  where
    a = var "a" :: BDD_Vector8
    r0 = var "r0" :: BDD
    r1 = var "r1" :: BDD
    reset = r0 `delay` r1 `delay` false
    [n, m] = counter a reset @@@ [0, 1]
```

where (@@@ is an operator for sampling a signal at the specified times.) By evaluating test, we are proving that, for Boolean vectors of length 8, the counter circuit meets our specification. Using types more general than BDD_Vector8, we can prove the properties for counters of arbitrary size.

One of the unsatisfying aspects of this verification example is that it was necessary to make the internal state of the counter an explicit parameter. Doing this in a complex model would entail passing around a lot of extra parameters—just the sort of thing we'd like to avoid. Also, in forcing the model to be explicit about its internal state, it undercuts one of the strengths of the signal transformer model that sets it apart from state transformer models, making it a sort of unwelcome hybrid. However, using ideas from Symbolic Trajectory Evaluation [15], we are currently working with symbolic domains that have a partial order structure. Symbolic simulation proceeds by starting with initial states set to bottom, with iteration of the model gradually adding more information. The fit with lazy stream models looks very good indeed.

8 Where Haskell and Hawk Tangle

For our domain, Haskell has turned out to be an excellent tool for experimenting with language design. However, in a few places, Haskell is not a perfect match. In this section we point to some of the hinderences that we have encountered.

8.1 Lazy Lists

In some cases Haskell is a little too generous. Our preferred semantics for signals is that of truly infinite, or coinductive, lists—i.e., not that of finite, infinite, and partially defined lists, as in Haskell. Any feedback loop that did not include at least one delay should be rejected by Hawk as being ill-defined—the corresponding hardware would generate more smoke than data. Haskell, however, will stubbornly do its best to make sense of even such ill-defined definitions. Could Haskell be coerced to match our intended application better?

We have constructed a shallow embedding of Hawk in Isabelle [30], which is much less forgiving. In order to have Isabelle accept our recursive definitions we have had to develop a richer theory of induction over coinductive datatypes than previously available [24]. Using this theory, Isabelle is able to accept all the valid Hawk definitions that we have thrown at it, while rejecting the invalid ones. It would be useful if Haskell's type system could be extended to handle this—perhaps using unpointed types [22] to express valid coinductive definitions.

8.2 Type Classes

For generality, the type representing an instruction set must remain abstract. Consequently we cannot directly pattern

match on it. Instead, the operations of the `Instruction` class provide predicates to identify common instructions such as nops, arithmetic ops, loads and stores and jumps.

```
class (Show i, Eq i) => Instruction i where
    isNoOp :: i -> Bool
    isAddOp :: i -> Bool
    isSubOp :: i -> Bool
    ...
```

If Haskell allowed arbitrary views of datatypes then this could be handled much more nicely. Such a proposal would not need to go so far as Wadler's *views* [36] (with their problems of hidden computation) to be useful.

8.3 The State Monad

Haskell's syntactic support for state is not a perfect fit. In particular, Haskell has no way to declare storage statically, although this is exactly what is required. In the register example, the array is allocated at the beginning, and nothing else is allocated afterwards. This reflects the fact that silicon cannot be allocated on the fly. Furthermore, when we come to consider other interpretations of Hawk models, it would be useful to guarantee that the body of the state code did not affect the shape of the store, merely its contents.

8.4 Using `unsafePerformIO`

Probes often work quite well, but there are some glitches. While we have been careful to preserve the semantics of Haskell in introducing probes, the semantics of probes are not really preserved by Haskell. Due to lazy evaluation, there is nothing to ensure that probe output will appear in the order expected. The output of a probe at clock tick 9 might be put in the file before the output of a probe at clock tick 7. Another glitch arises because a given unit can be used repeatly within a microarchitectural model. If that unit has an embedded probe, the output of both uses of the probes will be merged in one file. This is not problematic for execution of the model (for probes cannot affect the models themselves), but there is no way of identifying which output is from which use of the probe.

8.5 Symbolic Simulation

Our drive to make the entire Hawk library sufficiently polymorphic to perform symbolic evaluation has made us painfully aware of the shortcomings of Haskell's type class system in describing abstract data types. Haskell's module system *can* be used in a limited way to effect abstraction, as we have used for the signal type. This allows us to work around some of the problems with type classes, because we can completely reinterpret the meaning of symbols, both their types and their values. But Haskell's module system is only intended as name space management, and is a poor match when you intend to use abstract types instantiated at many different types. Whether an ML-style module system would work better in this case is an interesting question.

The type class system at times works brilliantly. What is perhaps most impressive is how well it has worked even when we use it for tasks far beyond its original intended use (simply as a system of overloading numeric and equality types). However, the fit is not always perfect. One place is the lack of explicit control over which instances are used where. One of the neat aspects of symbolic evaluation is that it allows us to take an existing executable model and verify properties of it, without changing the model at all.

However, this does not work quite as well as it could because of limitations in the class system. Ideally, we would like to instantiate the `test` expression above at different symbolic types. However, there is no good way to parameterize `test` by the types in question, without resorting to unpleasantries like adding dummy arguments. The type of the data for `counter` is purely an intermediate value in the definition of `test`. If we were not specific about the type of the initial value `a`, Haskell would consider the declaration ambiguous. We would like a way to parameterize which instance is used without having a dummy value parameter.

8.6 Elaboration Monads

One of the shortcomings of Hawk is that it has no explicit notion of elaboration, separate from the semantics of the model. Elaboration is the process of translating a possibly higher-order Hawk model into a first-order description, such as a netlist, or utilizing primitives of hardware description languages like VHDL or Verilog. This was not always the case. Initially, Hawk was similar to Lava [3] (in fact the two languages started from a common block of definitions), and used a monad of circuits to express circuit elaboration. Different implementations of the abstract monad would be used to generate net-lists for low level tools to manipulate, or logical formulae for input to a theorem prover, or simply execution for simulation and testing. To perform simulation, for example, the circuit monad is implemented simply as the identity monad, since all we have to do is glue together functions. A richer version of simulation, however, could provide the machinery to allow the output of duplicated probes to be separated, so removing the problem with probes that we outlined earlier.

There were two reasons we departed from an explicit monadic style. First, the presence of the monad made simple function application tedious. We could live with this, or work around it. Much more serious, however, was the lack of any syntactic help for mutual recursion between the results of monadic actions. The idiom of mutually recursive streams works so well for describing circuit feedback that we wanted something similar for monadic computations. For example, restating the example of the counter in monadic form ought to come out something like this:

```
counter :: Signal Bool -> Circuit (Signal Int)
counter reset = do
  { next <- delay 0 inc
  ; inc  <- lift1 (+1) out
  ; out  <- mux reset zero next
  ; zero <- lift0 0
  ; return out}
```

Unfortunately, a corresponding recursive do-form is not currently available. We would like to see the do notation extended so that the bindings are mutually recursive, with the recursion being defined by a user-supplied definition of an `mfix` function:

```
mfix :: Monad m => (a -> m a) -> m a
```

Note that, as the `counter` example shows, the obvious generic definition of `mfix` as

```
mfix f = do { z <- mfix f
            ; f z}
```

is simply not appropriate. We want the looping to take place on the values manipulated by the monad, not on the effects the execution of the monad generates. Rather we need something with the behaviour of `fixST` [23]. Finding an appropriate axiomatization for `mfix` is the subject of current research.

9 Hardware Algebra

As promised, we close with a section describing how the functional perspective gives us new insight into the structure of microarchitectures.

Transformational laws are well known in digital hardware, and form the basis of logic simplification and minimization, and of many retiming algorithms. Traditionally, these laws occur at the gate level: de Morgan's law being a classic example. We were quite surprised when corresponding laws started to emerge at the microarchitectural level!

Perhaps we shouldn't have been surprised. After all, functional languages are especially good at expressing transformational laws, and algebraic techniques have long been used in the relational hardware-description language Ruby [32]. Sizeable Ruby circuits have been successfully derived and verified through algebraic manipulation [16, 17]. Even so, the Ruby research has emphasized circuits at the gate level and, a priori, there is no reason to think that large microarchitectural components should satisfy any interesting algebraic laws: the components are constructed from thousands of individual gates, and boundary cases could easily remove any uniformity that would have to exist for simple laws to be present. Yet we have found that when microarchitectural units are presented in a particular way, many powerful laws appear.

Before we consider one of the laws in some detail, note first that we inherit for free the ground rule of referential transparency or, in hardware terms, a *circuit duplication law*. Any circuit whose output is used in multiple places is equivalent to duplicating the circuit itself, and using each output once. Because Hawk is embedded in Haskell (and introduces no new features that would otherwise break referential transparency), every circuit satisfies this law. That is, it is impossible within Hawk for a specification of a component to cause hidden side-effects observable to any other component specification. Of course, in many specification languages this law does not hold universally. For example, duplicating a circuit that incremented a global variable on every clock cycle would cause the global variable to be incremented multiple times per clock period, breaking behavioral equivalence. Hawk circuits can still be stateful, but all stateful behavior is forced to be local (the encapsulated state example) and/or expressed using feedback.

9.1 Register-Bypass Law

The law we will discuss in some detail is the *register-bypass law*. To do so, we need to discuss register files and bypasses in more detail than we have up to now.

Consider a transaction-based specification of a register file. This component has two input signals (for reading and writing) and one output signal, each of which are signals of transactions. At each clock cycle, the read-input is expected to contain a transaction whose opcode and register name fields have been set, but whose value fields are absent, whereas the write-input contains a completed transaction from a previously executed instruction. Execution proceeds as in the simplified example in Section 5. The register-file first performs the write by updating its internal state on the basis of the destination register-name and value fields of the write-input. Then, it performs the read by filling in the value fields for the source-operands of the transaction on the read-input. The resulting transaction is placed on the output. In this model, all this work is performed in a single clock-cycle.

Now consider bypasses, and the role they have in the specification of forwarding. The purpose of forwarding logic in a pipeline is to ensure that results computed in later stages of the pipeline are available to earlier stages in time to be used. Conceptually, the forwarding logic at each pipeline stage examines its current instruction's source register names to see if they match a later stage's destination register name. For every matching source name, the corresponding value is replaced with the result value computed by the later pipeline stage. Non-matching source operands continue to use operand values given by the preceding pipeline stage.

This conceptual logic can be implemented concisely using transactions. A bypass circuit has two inputs, each a signal of transactions. The first contains the input transactions from the preceding pipeline stage, and the second is the control or update input, containing transactions from later stages in the pipeline. At each clock cycle, the bypass circuit compares the source names of the current input transaction with the destination names of the current update-transaction. The output of the bypass is identical to the input, except that source operands matching the update's destination operand are updated.

Bypasses have many nice properties by themselves. Not only are they time-invariant (delays can pass over them) but they are idempotent in their second argument:

$$\forall inp\ .\ \forall upd\ .$$
$$\texttt{bypass}\ upd\ (\texttt{bypass}\ upd\ inp) = \texttt{bypass}\ upd\ inp$$

Most interesting, however, is their interaction with register files, which can be expressed with the *register-bypass* law:

$$\forall read\ .\ \forall write\ .$$
$$\texttt{bypass}\ write\ (\texttt{reg}\ (\texttt{delay Nop}\ write)\ read) =$$
$$\texttt{reg}\ write\ read$$

In other words, we can delay writing a value into the register file, so long as we also forward the write-value to the output, in case that register was being read on the same clock cycle. We use this law repeatedly to eliminate forwarding logic when simplifying pipelines. Seen the other way around, this law *explains* the origin of forwarding logic.

Initially we considered the register-bypass law to be a theorem about register files, and accordingly we proved that it held for a number of different implementations. However, it is also tempting to view this law as an *axiom* of register files. In effect, by using the law repeatedly from right to left, we obtain a specification for how the register file must behave for any time prefix.

9.2 Transforming the Microarchitecture

Other laws of microarchitectural algebra include a hazard-bypass law, for transforming multi-cycle pipelines in the presence of data hazards, and projection laws, for expressing local properties of signals [25, 26]. Here we note that

the laws we have discovered up to now are *by themselves* sufficiently powerful to simplify a pipelined microarchitecture that uses forwarding, branch speculation and pipeline stalling for hazards. The resulting simplified pipeline is very similar to the reference machine specification (i.e. no forwarding logic), while still retaining cycle-accurate behavior with the original implementation pipeline.

10 Acknowledgements

For their contributions we would like to thank Mark Aagaard, Borislav Agapiev, Todd Austin, Robert Jones, John O'Leary, and Carl-Johan Seger of Intel Corporation; Tim Leonard and Abdelillah Mokkedem of Compaq/Digital Corporation; Simon Peyton Jones of Microsoft Corporation; Per Bjesse, Koen Claessen, and Mary Sheeran of Chalmers; Satnam Singh of Xilinx; Elias Sinderson of GlobalStar; and Tito Autrey, Nancy Day, Dick Kieburtz, Sava Krstić, John Matthews, Thomas Nordin, and Mark Shields of OGI.

This research is supported in part by the Intel Corporation, the National Science Foundation (DGE-9818388, EIA-9805542), the Defense Advanced Research Projects Agency, and Air Force Material Command (F19628-96-C-0161).

References

[1] Hawk website. http://www.cse.ogi.edu/PacSoft/projects/Hawk/.

[2] AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).

[3] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

[4] BRYANT, R. E. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24*, 3 (1992).

[5] COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

[6] COOK, B., LAUNCHBURY, J., MATTHEWS, J., AND KIEBURTZ, D. Formal verification of explicitly parallel microprocessors. In *Conference on Correct Hardware Design and Verification Methods* (1999).

[7] DAY, N. A., LAUNCHBURY, J., AND LEWIS, J. R. Logical abstractions in Haskell. submitted for publication, 1999.

[8] DAY, N. A., LEWIS, J. R., AND COOK, B. Symbolic simulation of microprocessor models using type classes in Haskell. Tech. Rep. CSE-99-005, Department of Computer Science and Engineering, Oregon Graduate Institute, 1999.

[9] DULONG, C. The IA-64 architecture at work. *IEEE Computer 31*, 7 (1998).

[10] ELLIOTT, C. An embedded modeling language approach to interactive 3D and multimedia animation. *To appear in IEEE Transactions on Software Engineering* (1999).

[11] ELLIOTT, C., AND HUDAK, P. Functional reactive animation. In *The International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

[12] FINNE, S., LEIJEN, D., MEIJER, E., AND JONES, S. P. H/Direct: A binary foreign language interface for Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

[13] GWENNAP, L. First Merced patent surfaces. *Microprocessor Report 11*, 3 (1997).

[14] GWENNAP, L. Intel, HP make EPIC disclosure. *Microprocessor Report 11*, 14 (1997).

[15] HAZELHURST, S., AND SEGER, C.-J. H. Symbolic trajectory evaluation. In *Formal Hardware Verification*. Springer-Verlog, 1997.

[16] JONES, G., AND SHEERAN, M. Collecting butterflies. Tech. rep., Oxford University Computing Laboratory, 1991.

[17] JONES, G., AND SHEERAN, M. Designing arithmetic circuits by refinement in ruby. In *Mathematics of Program Construction* (1993), vol. 669 of *LNCS*, Springer Verlag.

[18] JONES, M. P. *Qualified Types: Theory and Practice*. PhD thesis, Department of Computer Science, Oxford University, 1992.

[19] JONES, S. P., AND MARLOW, S. Stretching the storage manager: weak pointers and stable names in Haskell, 1999. Submitted for publication.

[20] LANDIN, P. J. The Next 700 Programming Languages. *Communications of the ACM 9*, 3 (March 1966), 157–164.

[21] LAUNCHBURY, J., AND JONES, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.

[22] LAUNCHBURY, J., AND PATTERSON, R. Parametricity and unboxing with unpointed types. In *The International Conference on Functional Programming* (1996).

[23] LAUNCHBURY, J., AND PEYTON JONES, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (December 1995), 293–341.

[24] MATTHEWS, J. Recursive function definition over coinductive types. In *The 12th International Conference on Theorem Proving in Higher Order Logics* (Sept. 1999).

[25] MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra. In *International Conference on Computer-Aided Verification* (Trento, Italy, July 1999).

[26] MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra: Top-level proof of pipelined microarchitecture. Tech. Rep. CSE-99-002, Oregon Graduate Institute, Computer Science Department, Portland, Oregon, Mar. 1999.

[27] MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Aug. 1998).

[28] O'DONNELL, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).

[29] OKASAKI, C. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

[30] PAULSON, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[31] SEGER, C.-J. Voss – a formal hardware verification system. Tech. Rep. 93-45, University of British Columbia, 1993.

[32] SHARP, R., AND RASMUSSEN, O. An introduction to Ruby. Teaching Notes ID–U: 1995-80, Dept. of Computer Science, Technical University of Denmark, October 1995.

[33] SHRIVER, B., AND SMITH, B. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective.* IEEE Computer Society Press, 1998.

[34] SRIVAS, M., AND BICKFORD, M. Formal verification of a pipelined microprocessor. *IEEE Software 7*, 5 (1990).

[35] TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture* (Philadelphia, PA, May 1996).

[36] WADLER, P. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages* (Munich, Germany, January 1987).

[37] WADLER, P., TAHA, W., AND MACQUEEN, D. B. How to add laziness to a strict language withouth even being odd. In *Proceedings of the 1998 ACM Workshop on ML* (Sept. 1998), pp. 24–30. Baltimore, MD.