On embedding a microarchitectural design language within Haskell

John Launchbury, Jeff Lewis and Byron Cook
Oregon Graduate Institute

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`. We also point to several areas where Haskell could be improved.

1 Introduction

There are many ways to design and implement a language — not all of them imply building from the ground up. Landin's vision of the next 700 programming languages [18], 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. From our programming language perspective we believe that, in effect, every abstract type defines a language. Admittedly, most abstract types by themselves make poor languages, but when interesting combinators are provided the language suddenly 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 [10, 9]. 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.

How good is Haskell at hosting other languages? This is one of those questions that can only be answered through experience—and is precisely where we can contribute. In this paper we describe our use of Haskell as a host to 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 [15], the lazy state monad [19], and `unsafePerformIO` [17]. 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 to 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`, respectivly, are put to use in Hawk. In Section 7 we describe an application that makes use of all four features. In the final sections we outline where Haskell has constrained us, and discuss future work.

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 (ROB) which occurs frequently in out-of-order microprocessors like the Pentium III. The function of the ROB is to maintain a pool of instructions, and to determine dynamically which of them are eligible for delivery to an execution unit once their operands have been computed. This way, instructions are executed at the earliest possible moment. 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 ROB 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 been resolved, thus increasing the complexity of the interaction between a ROB 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 [13] and multiple instruction threads [29].

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 archi-

tecture. 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 upcoming Merced processor will execute not only its new instruction set [8], but X86 as well [12].

With designs of this complexity, it is hard to imagine that designers will not stumble upon subtle 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. In conjunction with Intel's Strategic CAD Laboratory, we have developed *Hawk* as an executable modelling language embedded in Haskell. 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: adding signals to Haskell

Effectively, Hawk is an embedding of Lustre-style signals [4] into Haskell. Signals model values that change over time, like wires in a microprocessor. Following O'Donnell [24], Srivas & Bickford [28], and many others, we implement signals as lazy lists. The idea is very 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 list transformers (or stream transformers). Combinational circuits can be turned into clocked circuits simply by mapping them down their input lists. So if `add::(Int,Int)->Int` acts like 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".

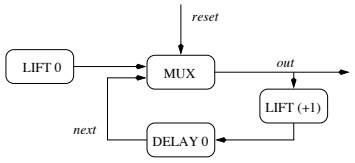
Some care is needed within this paradigm, however. Arbitrary use of list processing functions, especially those which discard elements, such as filter, can cause problems in that they may require infinite buffers to implement. To restrict the way in which a signal can be constructed or altered, we make the signal type abstract in Hawk and provide a basic set of manipulation functions that are known to be safe.

```
newtype Signal a

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

`lift0` returns a constant signal; and `lift1` is just map. 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, 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 reset = out
  where
  next = delay 0 (lift1 (+1) out)
  out  = mux reset (lift0 0) next
```

Note the mutual recursion between signals. The laziness of Haskell is vital for this definition to have the intended meaning.

One thing that is not missing 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 an infinite buffer. 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 Organizing microarchitectural abstractions with type classes

The point of Hawk has been to build abstractions that increase the concision of microarchitectural models [5], and facilitate the verification process [22].

In order 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[27] 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 an ALU might be:

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

This way `alu` can be used in a 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 `Bits` 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 different instructions sets.

With common architectural characteristics captured with type classes, we are then able to build abstractions that help organize microarchitectural models. For example, *transactions* [1, 23] are a simple yet powerful grouping of control and data. A transaction is a machine instruction grouped together with its state. This state might include:

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

Microarchitectures models that utilize transactions can then make decisions locally rather than with a seperate control unit.

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` inserted `i`'s more recent valuation of `r4` into the destination operand of `j`.

By parameterizing over the instances of finite words and registers:

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

`bypass` can be used in many contexts. Within our Pentium III-like microarchitectural model we use `bypass` on both instructions with real register references and virtual register references (both 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 strictness within the state monad. In this section we describe an application where a lazy state monad is the right thing.

Some microarchitectural components, such as register files, are more naturally (and effeciently) presented as state transition systems than list transformers. Fortunately, we can easily embed state-based models into the list transformer idiom using the *lazy* state monad and `runST` [19].

Imagine modelling a register file as an array which, on each clock tick, is both written to and read from.

```
reg :: Register r => Signal (r,w) -> Signal r ->
                     Signal w
reg writes reads
  = runST (
    do { reg <- newArray (minAddr, maxAddr) init
       ; loopST (regFile reg) (bundle writes reads)
       }
    )

regFile :: STArray s Addr Val -> ((Addr,Val), Addr)
          -> ST s Val
regFile reg ((a,w),r)
   = do { writeArray reg a w
        ; readArray reg r
        }
```

where `loopST` is a monadic map on signals:

```
loopST :: (a -> ST s b) -> Signal a
          -> ST s (Signal b)
```

The semantics of lazy state is as follows. The monadic structure sequentializes the operations of the monad but *forces nothing*. As the result of the state thread is demanded, so execution proceeds, but in the order determined by the monadic sequentialization. Thus execution proceeds on demand, but some of that demand is transmitted through the state sequencer.

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 occurs because of the type of `runST`. Inside the implementation of `regFile`, however, the situation is quite different. The array writes are "imperative", having effects immediately visible to subsequent reads.

In the use of `loopST` above, the state machine is executed step by step, consuming its list input and generating its list output on the way. In particular, the `loop` construct did 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 Monitoring circuits with `unsafePerformIO`

When embedding a language, one often needs "language primitives" that provide good things in bad ways. Fran for example, has a function :

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

which imports a bitmap file in the `IO` monad but uses `unsafePerformIO` to treat the bitmap as a pure value.

When using Hawk we find that one often wants to observe the values flowing across a signal. Unfortunately, Haskell's semantic purity makes this viewing rather difficult. Often, without re-coding a model, it is not possible to observe the signal. Therefore we provide the function:

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

As far as Hawk-level models are concerned, a probe is simply an identity. However, the external world receives a different view. Probes are fundamentally 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 required some Haskell system hacking through the use of `unsafePerformIO`.

```
probe name vals = zipWith (write name) [1..] vals

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

Notice that we are careful not to change the strictness of lazy lists.

We have found that `unsafePerformIO` is a powerful facility for building of domain-specific tools that observe, but do not affect the microarchitectural models.

7 Verification in Hawk

The past several sections have, one-by-one, demonstrated the usefulness of lazy lists, type classes, the state monad, and `unsafePerformIO`. In this section we discuss a particularly exciting application that requires all four features.

Hawk provides tools that can be used to formally verify properties of models. Suppose 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. and 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 outputs.

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

The trick is to show that these properties hold for arbitrary values of `r0` and `r1`. To do that, we will use symbolic values for `r0` and `r1`, and symbolically simulate the circuit.

The approach we take to symbolic simulation [7] is straightforward. Take a sufficiently polymorphic function, and instantiate it at a symbolic datatype. What we mean by a symbolic datatype is any datatype that is enriched with 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)
```

The catch is that some care is required in making functions "sufficiently" polymorphic. This means that over the parts of the program that you wish to symbolically evaluate, you cannot use concrete types, because those types must be able to become symbolic.

7.1 Fitting symbolic simulation into Haskell

In places, such as with the `Num` class, Haskell's prelude is remarkably amenable to symbolic simulation. In others it is not. As an example, consider Booleans. To capture the operations of both concrete and symbolic Booleans we have defined 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 have also defined the class `Eql`, which is like the standard `Eq` class, except that it is also abstracted over the result type for equality, resulting in a multi-parameter type class:

```
class 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, then 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
```

We can now employ many implementations of Booleans. In particular we can use binary decision diagrams (BDDs) [3], which implement semantic equality between symbolic boolean expressions in constant time. Using H/Direct [11], the state monad and `unsafePerformIO`, we have imported the CMU BDD package into Haskell. In the style of the modelling language of Voss [26], Hawk treats BDDs just like Booleans. But, thanks to type classes, a user can also choose *not* to use BDDs — so long as their choice is an instance of `Boolean`.

7.2 Proving a property

We now have the infrastructure 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. We have made the initial value of the delay in the counter an additional parameter so that we can place a symbolic value there as well. This makes our test independent of the internal state of the counter, and thus makes it valid to test the properties only at the first two clock ticks.

```
test :: BDD
test = prop_one && prop_two
    where
    a = var "a" :: BDD_Vector8
```

```
r0 = var "r0" :: BDD
r1 = var "r1" :: BDD
reset :: Signal BDD
reset = r1 `delay` r1 `delay` false
[n, m] = counter a reset @@@ [0, 1]
prop_one = not r1 ==> (n + 1 === m)
prop_two = r0 ==> (n === 0)
```

(`@@@` 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 that `BDD_Vector8`, we can prove the properties for counters of arbitrary size.

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 review our use of lazy lists, type classes, the lazy state monad, and `unsafePerformIO` and point to 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 as being ill-defined. Haskell, however, will stubbornly do its best to make sense of even such ill-defined definitions. Could Haskell do better? We have constructed a shallow embedding of Hawk in Isabelle [25], 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 [21]. 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 [20] to express valid coinductive definitions.

8.2 Type Classes

Because the type representing an instruction set must remain abstract, 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 [30], then this could be handled much more nicely.

8.3 The State Monad

Haskell's syntactic support for state is not a perfect fit. First, Haskell has no way to declare storage statically, but this is exactly what is required. In the register example, the array is allocated at the beginning, and nothing else is allocated afterwards. Since silicon cannot be allocated on the fly, 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.

Secondly, in our microarchitectural models, the pattern `loopST f (bundle xs ys)` occurs often enough to want a language construct to describe it. Putting these ideas together, we may ideally wish to write something like:

```
reg writes reads
  = runST (do {array reg (minAddr, maxAddr) = init
              ; loop (w<-writes, r<-reads)
                   {  writeArray reg a w
                   ;  readArray reg r
                   }
               }
          )
```

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's nothing to assure 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 is that, in a model, we are free to use a given unit more than once. But if that unit has an embedded probe, you will get the output of both probes in the file. This is not problematic, except that you have no way of identifying which output is from which probe.

But these problems have less to do with the perhaps unscrupulous nature of using `unsafePerformIO`, and more to do with a shortcoming in our overall design. In the section on future work, we will discuss an approach that will mitigate these problems.

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. 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.

The type class system at times works brilliantly. And what is most impressive is how well it has worked for us, as we use it for tasks far beyond its original intended use (simply as a system of overloading). However, the fit is not always perfect. One place is the lack of explicit control over instancing. 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 `test` 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 counter data

is purely an intermediate value in the definition of `test`. If we were not specific about the type of `a`, Haskell would consider the declaration ambiguous. Here we are limited by the type class system's restriction to type inference—the programmer is given no tool to resolve the ambiguity. Just as type inference can be augmented by type annotations to help the type system where it can't help itself, as with polymorphic recursion, we should be able to provide some sort of annotation to help Haskell resolve ambiguous uses of type classes.

9 Future work

9.1 Verification

One of the unsatisfying aspects of the 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 also 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 [14], 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.

We are also currently applying symbolic simulation to simple pipelined microarchitectures to verify correctness of hazard avoidance, using a self-consistency checking approach [16]. The technique is to simulate a stream of symbolic instructions two times. Let us assume that the pipeline has two stages. In the first case, we feed two symbolic instructions followed by a no-op. In the second case, we feed the same two symbolic instructions *separated* by the no-op. The test is that the contents of the registers is the same after the third instruction, demonstrating that the hazard logic is working correctly.

9.2 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 circuit into a first-order description, such as the hardware languages VHDL or Verilog. This was not always the case. Initially, Hawk was similar to Lava [2], using a monad to capture circuit elaboration. The monad might be used to generate net-lists for the purposes of fabrication, or it might produce logical formulae for input to a theorem prover. For simulation, the monad is essentially the identity monad, since all we have to do is glue together functions. However, during simulation, the monad could also provide the service of, for example, splitting probes that get duplicated.

One reason that we departed from an explicit monadic style is that the mutually recursive streams idiom that works so well is not supported by the `do` notation. What we propose is to extend the `do` notation so that bindings are recursive.

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; Elias Sinderson of GlobalStar; and Dick Kieburtz, John Matthews, Nancy Day, Sava Krstić, Thomas Nordin, Tito Autrey, and Mark Shields of OGI.

This research is supported in part by Intel, the National Science Foundation, the Defense Advanced Research Projects Agency, and Air Force Material Command.

References

[1] 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).

[2] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

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

[4] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages* (Munich, Germany, Jan. 1987).

[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 microarchitectures, 1999. Submitted for publication.

[7] Day, N. A., Lewis, J. R., and Cook, B. Symbolic simulation of microprocessor models using type classes in Haskell. Submitted for publication.

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

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

[10] Elliott, C., and Hudak, P. Functional reactive animation. In *The International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

[11] 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).

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

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

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

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

[16] JONES, R. B., SEGER, C.-J. H., AND DILL, D. L. Self-consistency checking. In *Formal Methods in Computer-Aided Design* (Palo Alto, California, 1996).

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

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

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

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

[21] MATTHEWS, J. Recursive function definition over coinductive types. Submitted for publication.

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

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

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

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

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

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

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

[29] 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).

[30] 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).