# Another Look at Hardware Design Languages:
Freeing ourselves from the tyranny of the host language

Tim Sheard
Computer Science Department
Maseeh College of Engineering and Computer Science
Portland State University
sheard@cs.pdx.edu

December 28, 2005

## 1 Introduction

Using a functional language (mostly Haskell) as a host language for embedding hardware description languages is a popular theme. Examples include Ruby, Hydra, Hawk, and Lava. The reasons[3] for this approach to hardware description language design are numerous:

- The syntax of the host language can be reused. There is no need to invent a new syntax or write parsers.
- All the machinery of the host language can be reused. Tools for the language such as compilers and debuggers, and language features such as its module system and type system.
- Even the programmers who already know the host language can be reused without re-training.
- Finally, hardware descriptions are now first class objects in the host language and are subject to all the properties of host language objects.

Of course there are disadvantages to such an approach, and these cannot be overlooked. They include:

- The syntax of the two languages may be radically different. So users of the hardware description language feel that they are being forced into an unnatural programing style.
- The features of the host language may not match the features of the embedded language. Type systems, in particular are an example of this kind of mismatch. In this case, unnatural or awkward embeddings must be used. These embeddings increase the complexity of the system, and make it harder to use and maintain.

Most proponents of the embedded language approach recognize these disadvantages, and note that a stand-alone language (as opposed to an embedding) with its own syntax, type system, and implementation might be more useful. But most, pay only lip service to actually building such an implementation. We believe it is time to actually take such a leap.

# 2 Advantages of a stand-alone implementation.

A stand-alone implementation of an HDL could provide its users with several important advantages. These advantages include a type system (or systems, see Section 5) specific to the domain of hardware description, and a cleaner distinction between the multiple stages inherent in the design of hardware. An embedded approach is stuck with the type system of the host language. A stand-alone system is free from this constraint. A stand-alone implementation is also free to address the phase distinction in new and novel ways. We now discuss some of the problems inherent in choosing an embedded approach.

## 2.1 Intertwined recursion

In many current embedded HDLs two different forms of recursion are implemented with the same mechanism in the host language. These two forms play very different roles, and because they are implemented with the same mechanism, they cannot later be easily separated. The two forms of recursion are:

- Constructing large circuits with repeated patterns from smaller ones can be described by writing inductive functions that induct over some index that describes the connection pattern between the patterns. This index is often something as simple as an integer, but can be much more complicated, such as a tree.
- Feed back loops in sequential circuits can be described by recursive equations.

There is a phase distinction between the two kinds of recursion. The first describes a computation in the first phase when we are constructing the design. It is a real program. It should be inductive, in that such programs should always terminate (we don't want to wait forever for our design to be produced). We may, or may not, want to enforce this.

The second describes the recurring behavior described in the design. This behavior "runs" in the second phase when the design is implemented. As far as the HDL is concerned, the second is a piece of data. These are truly recursive, in that they define infinite behavior patterns, but they should always be captured by a finite description (after all, we cannot build infinitely large hardware devices).

Current embedded approaches mix these two types of recursion in ways that are hard to separate. They are both implemented by recursion in the host language. This has two unfortunate consequences. First it forces us to use a lazy semantics. This may or may not be a good thing, but we should have a choice. Second, hardware designs with feedback loops always create infinite structures (why we need laziness). This forces us to use "tricks" to observe the sharing in the cyclic heap objects that represent the data, so that we can recover the finite description (of the infinite behavior) we should have built in the first place.

## 2.2 Type Systems that are to Weak.

An embedded approach forces the HDL to share the host languages type system. Even the rich type system of Haskell (the most common host language) with its rich class mechanism and rank-N polymorphism is too weak to capture some simple static properties of HDL programs. Examples include:

- **Typed HDL Object Descriptions.** HDL programs produce hardware descriptions. Such descriptions are data in the HDL, but they also represent programs (or hardware). Such descriptions could have types, and we would like to ensure that the designer only produce well-typed designs. Current languages use *phantom-types* [2, 4, 6] to help ensure that we only *build* well typed designs, but phantom types are truly phantom when *analyzing* designs. Type information is lost in the analysis phase and cannot be recovered.
- **Structures with Fixed Width.** Hardware is full of structures with fixed width: 32-bit registers, 64-bit busses, 8-bit adders. Can we specify generic algorithms that manipulate n-bit entities, then specialize them to a fixed width, and be sure we never mix two similar objects with different widths?

- **Powers of 2.** Many abstract designs (such as a butterfly circuit) can be precisely described using regular patterns only when the size of the design is a power of 2, (or in some examples a perfect square). Such a property can be statically enforced, but the type system of the host language is usually too weak.

- **Interface Types.** Many signals have directionality (input v.s. output). A static type system could prevent the error of connecting two inputs, or leaving a signal dangling.

- **Protocol Types.** Many hardware descriptions must behave according to a certain protocol. For example, a simple handshake protocol depends on *request* and *acknowledge* signals strictly alternating. Some protocols look more like complex automata than types. Can a static type system enforce such behavior?

- **Counting the Delay in a Circuit.** Good designs follow patterns. For example, some design patterns insist that every cyclic loop nest includes a delay. Can we statically enforce such design patterns?

# 3 The next HDL.

A stand-alone implementation of a HDL should have a number of desirable properties. We discuss these properties at a high level of abstraction here, and then in the ensuing sub-sections, describe how these properties might be implemented.

- **Descriptions as first class data.** Hardware descriptions are both data and programs. But, it is important that they be first class data structures. Their very purpose is to be manipulated as data. Anything that makes this hard is in direct competition with their primary role. In particular, they should not be encoded as functions or as cyclic heap structures, because this makes it difficult, if not impossible, to perform the analyses they were designed for.

- **Typed representations.** Hardware descriptions should be typed. It should be possible to describe rich abstractions such as tuples, and busses of width $n$. These abstractions should be available in all stages. In particular it should be possible to construct both tuples of circuits, and circuits of tuples (as well as other richer data structures). The type system should apply in the analysis phase as well as the construction phase of the descriptions.

- **Special syntactic sugar.** A stand-alone implementation should support syntactic sugar to make it easy to capture hardware descriptions.

- **Multiple static analyses.** A stand-alone implementation can support multiple static analyses, one for each important static property. The specification and checking of different analyses should be separable to minimize the conceptual effort needed to "debug" errors when they occur.

- **Multiple interpretations.** A hardware description has multiple interpretations. The same piece of data should be interpretable in multiple ways.

Haskell makes an excellent host language. Its support for abstraction and first class functions make it an excellent meta-language to describe the manipulation of HDL object-language programs. But, its type system is too weak, and its syntax is too restrictive to describe the phase distinction between the meta-programs and the hardware descriptions as data. For these reasons, we propose building a stand-alone HDL with much the same look and feel of Haskell, but with features that allow it to overcome these restrictions. Imagine an Haskell-like language with a stronger type system, a slightly altered syntax, and features designed for describing hardware. We have, and this paper describes what we came up with.

## 3.1 Generalized Algebraic Datatypes

A hardware description is a piece of data. We will represent it as a *type-indexed* algebraic datatype. Algebraic data types (ADT's) are an abstraction available in many functional languages (such as Haskell, ML, or O'Caml) that allow users to define inductively formed structured data. A type-indexed algebraic datatype is an extension to ADTs we will explain in a moment.

ADTs generalize other forms of structuring data such as enumerations, records, and tagged variants. For example, in Haskell we might write:

```
-- An enumeration type
data Color = Red | Blue | Green
-- A record structure
data Address = MakeAddress Number Street Town
-- A tagged Variant
data Person = Teacher [Class]  | Student Major
```

We assume the reader has a certain familiarity with ADTs. In particular that values of ADTs are constructed by *constructors* and that they are taken apart by the use of *pattern matching*. Two valuable features of ADTS are their ability to be *parameterized by a type* and to be *recursive*. A simple example that employs both these features is:

```
data Tree a
   = Fork (Tree a) (Tree a)
   | Node a
   | Tip
```

This declaration defines the polymorphic `Tree` type constructor. Example tree types include `(Tree Int)` and `(Tree Bool)`. In fact the type constructor `Tree` can be applied to any type whatsoever. Note how the constructor functions (`Fork`, `Node`) and constants (`Tip`) are given polymorphic types.

```
Fork ::  forall a .  Tree a -> Tree a -> Tree a
Node ::  forall a .  a -> Tree a
Tip ::  forall a .  Tree a
```

When we define a parameterized algebraic datatype, the syntactic formation rules of the `data` declaration enforce the following restriction: The range of every constructor function, and the type of every constructor constant must be a polymorphic instance of the new type constructor being defined. Notice how the constructors for `Tree` all have range (`Tree` $a$) with a polymorphic type variable $a$. We can remove this restriction by strengthening the type system of the language. The restriction is **syntactically** removed by the following mechanism. In a `data` declaration, rather than leaving the range of the constructor functions implicit, the user replaces the enumeration of the constructors and the type of their domains with a full explicit typing of each constructor. The only restriction being that the range must be some instance of the type being defined. **Semantically**, this requires new type checking rules. This simple extension allows us to build indexed-datatypes. A simple example with relevance to hardware design can be found in Figure 1.

Circuits are nothing more than a data structure. They are built by the constructor functions of their datatype. Simple combinational circuits can be described as values of the `Exp` datatype which is defined in Figure 1. The type constructor `Exp` takes a type as an argument (which we call its type index). The type index of `Exp` tells us what kind of circuit it encodes – `(Exp Bit)` if it represents a combinational circuit of bits, and `(Exp Int)` if it represents a combinational circuit of integers. To fully understand this we need to take a short side trip into the world of kinds.

## 3.2 Values, Types, Kinds, and Sorts

The kind of `Exp` is `*0 ~> *0`. Kinds are similar to types in that, while types classify values, kinds classify types. We indicate this by the overloaded *classifies* relation (`::`). For example: `5::Int`, and `Int::*0` .

```
data Exp ::  *0 ~> *0 where
  O      :: Exp Bit                          -- Bits
  I      :: Exp Bit
  And    :: Exp Bit -> Exp Bit -> Exp Bit
  Or     :: Exp Bit -> Exp Bit -> Exp Bit
  Not    :: Exp Bit -> Exp Bit
  Xor    :: Exp Bit -> Exp Bit -> Exp Bit

  Lit    :: Int -> Exp Int                   -- Integers
  Equal  :: Exp Int -> Exp Int -> Exp Bit
  Plus   :: Exp Int -> Exp Int -> Exp Int
  Minus  :: Exp Int -> Exp Int -> Exp Int
  Times  :: Exp Int -> Exp Int -> Exp Int
```

Figure 1: Type-indexed ADT capturing simple circuit descriptions. Notice how the full type of every constructor is explicitly supplied, and how the range of each constructor is not necessarily a polymorphic instance of `Exp`.

We say `5` is classified by `Int`, and `Int` is classified by `*0` (star-zero). The kind `*0` classifies all types that classify values (things we actually can compute).

**New types** are introduced by the **data** declaration along with the value constructors that produce values of that type. **New kinds** are introduced by the **kind** declaration along with the type constructors that produce types of that kind. Several declarations, for types and kinds of interest in this paper follow:

```
kind Nat = Z | S Nat

data Nat':: Nat ~> *0 where
   Z:: Nat' Z
   S::  (Nat' x) -> Nat' (S x)

kind Row (x :: *0) = RNil | RCons x (Row x)

kind Ctype = Comb | Seq
```

The `Nat` declaration introduces two new type constructors `Z` and `S` which encode the natural numbers at the type level. The type `Z` has kind `Nat`, and `S` has kind `Nat ~> Nat`. The type `S` is a type constructor, so it has a higher-order kind. We indicate this using the classifies relation as follows:

```
Z :: Nat
S :: Nat ~> Nat
Nat :: *1
```

The classification `Nat::*1` indicates that `Nat` is at the same "level" as `*0` — they are both classified by `*1`. The kinds `Ctype` and `Row` are also kinds different from `*0`. We summarize the relationship between values, types, kinds, and sorts in the table below.

| value name space | | type name space | | | | |
|---|---|---|---|---|---|---|
| value | | type | | kind | | sort |
| 5 | :: | Int | :: | *0 | :: | *1 |
| [2] | :: | [Int] | :: | *0 | :: | *1 |
| | | [] | :: | *0 $\rightsquigarrow$ *0 | :: | *1 |
| | | | | Nat | :: | *1 |
| | | Z | :: | Nat | :: | *1 |
| | | S | :: | Nat $\rightsquigarrow$ Nat | :: | *1 |
| | | Nat' | :: | Nat $\rightsquigarrow$ *0 | :: | *1 |
| Z | :: | Nat' Z | :: | * 0 | :: | *1 |
| S | :: | Nat' m $\rightsquigarrow$ Nat' (S m) | :: | *0 | :: | *1 |
| | | Exp | :: | *0 $\rightsquigarrow$ * 0 | :: | *1 |
| | | | | Ctype | :: | *1 |
| | | Comb | :: | Ctype | :: | *1 |
| | | Seq | :: | Ctype | :: | *1 |
| | | | | Row | :: | *1 $\rightsquigarrow$ *1 |
| | | RCons | :: | k $\rightsquigarrow$ Row k $\rightsquigarrow$ Row k | :: | *1 |
| | | RNil | :: | Row k | :: | *1 |

Note that the table is somewhat incomplete. Not all types classify values. For example there are no values of type `[]` (list) or `Z`, but there are values of type `[Int]`. The same holds at the kind level. Not all kinds classify types. For example, there are no types classified by `Row`, but `RNil` could be classified by `(Row Int)`. As is done in Haskell, there are two name spaces. One name space names values, and the other name space names types, kinds, sorts etc. Two objects can have the same name if they live in different name spaces.

We put this principal to work in the `Nat'` datatype. `Nat'` is a reflection of the `Nat` type into the value world where the value constructors of the `data` declaration for `Nat'` mirror the type constructors in the `kind` declaration of `Nat`, and the type index of `Nat'` exactly captures the kind reflected. We are able to use the same names for the constructor functions of `Nat'` as the type constructors of `Nat` because the name space for values and types is separate. `Nat'` is a singleton type. There is only one element of any singleton type. For example only `S (S Z)` inhabits the type `Nat' (S (S Z))`. Using the same names for the type constructors of kind `Nat` (`S` and `Z`), and the constructor functions of `data` type `Nat'` (`S` and `Z`), is deliberate. This is possible because of the separate name spaces for values and types.

We will find natural numbers at the type level (and their `Nat'` reflections at the value level) to be so useful we introduce some syntactic sugar for constructing such types (or values). For example: `#0 = Z`, and `#1 = (S Z)`, and `#2 = (S (S Z))` etc. We also support the syntax #(n + x) = ($S_1$($S_2$ ... ($S_n$ x))). This is one advantage of not using an embedded approach, the ability to introduce new syntax for commonly used idioms. This is analogous to the use of square brackets to describe lists in addition to the use of the constructor `(:)`.

### 3.3 Returning to Hardware Descriptions as First Class Type-Indexed Data

A value of type `(Exp Bit)` is interpreted as a combinational circuit with type `Bit`. A simple example of a function that manipulates circuits is:

```
halfAdd :: (Exp Bit) -> (Exp Bit) -> (Exp Bit, Exp Bit)
halfAdd a b = (Xor a b,And a b)
```

We can give meaning to such a data structure by writing a meaning or evaluation function for values of type `(Exp t)`. Note how the meaning of a value of type `(Exp t)` is a value of type `t`.

```
data Bit = Low | High

eval :: Exp t -> t
```

```
eval O = Low
eval I = High
eval (And x y) = andB (eval x) (eval y)
eval (Or x y) = orB (eval x) (eval y)
eval (Xor x y) = xorB (eval x) (eval y)
eval (Not x) = notB (eval x)
eval (Lit n) = n
eval (Plus x y) = (eval x) + (eval y)
eval (Minus x y) = (eval x) - (eval y)
eval (Times x y) = (eval x) * (eval y)
eval (Equal x y) = if (eval x)==(eval y) then High else Low
```

Note how the types of the constructor functions of `Exp` play a role when they are analyzed as well when they are constructed. The types are not phantom here. Type-indexed ADTs are an instance of a more general mechanism called Generalized Algebraic Datatypes (GADTs). The typing rules for GADT's are beyond the scope of this short paper, but they are well understood[5, 7].

### 3.4 Composite Data at the Circuit Level

The function `halfAdd` produces a pair of circuits of type `Bit` from two circuits of type `Bit`. Its type is `halfAdd :: Exp Bit -> Exp Bit -> (Exp Bit, Exp Bit)`. Notice how it returns a tuple of two circuits. Can we build circuits which encode tuples and other data structures inside the circuit? Such a function might have type: `Exp Bit -> Exp Bit -> Exp(Bit,Bit)`. This would allow designers to build structured circuits. To do this we would just enrich the GADT describing circuits.

```
data Vec:: *0 ~> Nat ~> *0 where
  Empty:: Vec a Z
  Add:: a -> Vec a m -> Vec a (S m)

data Exp ::  *0 ~> *0 where
  . . .
  Pair   :: Exp a -> Exp b -> Exp (a,b)
  CEmpty :: Exp (Vec a Z)
  CAdd   :: Exp a -> Exp (Vec a m) -> Exp (Vec a (S m))
```

Now we can build circuits of four different kinds of data: bits, integers, heterogeneous products, and homogeneous vectors of fixed width. Here are some examples

```
#2          :: Nat' #2           -- (S (S Z))
I           :: Exp Bit           -- I
(Lit 5)     :: Exp Int           -- (Lit 5)
#(I,Lit 5)  :: Exp (Bit,Int)     -- (Pair I (Lit 5))
#[I,O,I]    :: Exp (Vec Bit #3)  -- (CAdd I (CAdd O (CAdd I CEmpty)))
```

In addition to the new constructors for `Exp` (`Pair`, `CAdd`, and `CEmpty`) we are also illustrating a considerable amount of syntactic sugar as well. The syntax on the left expands to the calls to the constructor function expressions after the comment marks (`--`) on the right. Notice how we use syntactic sugar to model object-level numbers, tuples, and vectors using the meta-level syntax, just by prefixing them with the object level marker hash (`#`). This is just another thing we couldn't do in an embedded approach. Here is second version of the half-adder that returns a single circuit representing a pair of wires.

```
halfAdd2 :: (Exp Bit) -> (Exp Bit) -> Exp(Bit,Bit)
halfAdd2 a b = #(Xor a b,And a b)
```

We can also mix and match between meta-level and object-level data. Consider constructing a full adder from a n-bit adder

```
bitAdder :: Exp Bit -> Vec (Exp Bit) n -> (Vec (Exp Bit) n, Exp Bit)
bitAdder carry_in Empty = (Empty,carry_in)
bitAdder carry_in (Add x xs) = (Add y ys,carry_out)
  where #(y,carry) = halfAdd2 carry_in x
        (ys,carry_out) = bitAdder carry xs
```

Note how the `halfAdd2` returns a structured circuit, which we take apart using pattern matching, and then reassemble into a meta-level pair. Note also that the size of the adder is indicated in the type of the function that constructs it!

### 3.5 Binding and Naming Circuits

Reconsider the half-adder function.

```
halfAdd :: (Exp Bit) -> (Exp Bit) -> (Exp Bit, Exp Bit)
halfAdd a b = (Xor a b,And a b)
```

Note how the inputs `a` and `b` are duplicated by the function. If they represented large circuits the size of the output might be larger than one might naively expect. Of course duplicating circuits is not always bad. One might want to duplicate a small circuit to improve the delay in a larger one, of which it is a part. The point is, duplication is neither inherently good nor bad. What the designer needs is control over naming and duplication.

Introducing naming into the object-level means the object-level data structure must have binding constructs. Binding constructs are notoriously hard to use if exposed naively to the user. The study of meta-programming systems has provided a number of excellent solutions to this problem. These solutions require some syntactic sugar, and are thus not available in an embedded approach. Fortunately, we have avoided that trap by our initial design choices. Consider

```
halfAdd :: (Exp Bit) -> (Exp Bit) -> Exp(Bit,Bit)
halfAdd a b = circuit #(carry,sum)
  where carry = Xor a b
        sum = And a b
```

Here we introduce the `circuit` syntactic sugar for introducing named circuits. It looks and behaves a like a meta-level `where` clause, except it builds an `Exp` data-structure and names some of its sub-expressions. Further uses of the named sub-expressions will mention the name of the structure not their value, so duplication can be controlled. To implement this we again need to enrich the GADT that represents circuits by adding variables, patterns, and binding constructs.

```
data Pat :: *0 ~> *0 where
  PVar :: String -> (Rep t) -> Pat t
  PPair :: Pat a -> Pat b -> Pat (a,b)

data Exp ::  *0 ~> *0 where
  . . .
  Var    :: String -> Rep t -> Exp t
  Let    :: Decs t -> Exp t

Decs :: *0 ~> *0
data Decs t
   = In (exp t)
   | exists s . Decl (Pat s) (Exp s) (Decs t)
```

We introduce the `Pat` type, and add two new constructors to the `Exp` datatype (`Var`, and `Let`), and a new auxiliary datatype `Dec` to represent declarations. So the definition of the circuit in the half adder function above might expand into the data structure:

```
Let (Decl (PVar "carry" Bit) (Xor a b)
    (Decl (PVar "sum" Bit) (And a b)
    (In #(Var "carry" Bit,Var "sum" Bit))))
```

Because we want our variables to be typed, we must introduce the representation for types `Rep`.

```
data Rep:: *0 ~> *0 where
  Bit:: Rep Bit
  Int:: Rep Int
  Prod:: (Rep a) -> (Rep b) -> Rep (a,b)
  Vec:: Rep a -> Nat' n -> Rep(Vec a n)
```

A value of type `(Rep t)` is a value isomorphic in shape to the type `(t)`, but is a value rather than a type. Like `Nat'`, `Rep` is a singleton type, because there is exactly one value with type `Rep t` and that value is isomorphic in shape to the type `t`. For example:

```
Bit :: Rep Bit
Prod Bit Int :: Rep (Bit,Int)
Vec Int #1 :: Rep (Vec Int #1)
```

This allows users to create run-time values that encode types. By storing these singletons in `Var` and `PVar` objects the user can encode the type of the variables they are representing. Again, this uses the trick of naming the constructors of the `Rep` values with the same name as the types they represent, by exploiting the distinction between the value and type name space.

Patterns are a subset of expressions, essentially those expressions constructed from only variables and products. Many times it is useful to convert between patterns and this subset of expressions.

```
exp2pat :: Exp t -> Pat t
exp2pat (Var nm t) = PVar nm t
exp2pat (Pair x y) = PPair (exp2pat x) (exp2pat y)

pat2exp :: Pat t -> Exp t
pat2exp (PVar nm t) = Var nm t
pat2exp (PPair x y) = Pair (pat2exp x) (pat2exp y)
```

Including `Rep`s inside of variables and patterns makes it possible to compute the type of any expression. The function `typeOf` is an example of a function that analyzes an `Exp` to compute some property of that expression.

```
typeOf :: Exp t -> Rep t
typeOf (Var s t) = t
typeOf (If x y z) = typeOf y
typeOf O = Bit
typeOf I = Bit
typeOf (And x y) = Bit
typeOf (Or x y) = Bit
typeOf (Xor x y) = Bit
typeOf (Not x) = Bit
typeOf (Lit n) = Int
typeOf (Plus x y) = Int
```

```
typeOf (Minus x y) = Int
typeOf (Times x y) = Int
typeOf (Equal x y) = Bit
typeOf (Pair  x y) = Prod (typeOf x) (typeOf y)
typeOf (CAdd x y) = case typeOf  y of Vec _ n -> Vec (typeOf x) (S n)
typeOf (Let (In e)) = typeOf e
typeOf (Let (Decl p e ds)) = typeOf (Let ds)
typeOf (RLet ds) = typeOf (last ds)
  where last (Decl p e ds) = last ds
        last (In e) = e
```

Note, that because all `Exp`s are well formed, in some instance it is not necessary to compute the type of a an expression's sub-expressions to compute its type.

### 3.6 Sequential Circuits

Feed back loops capture recursion inside the circuit and are semantically different than the recursion we used in `bitAdder`. There the recursion indicated that the circuit producing function was being called repeatedly, not that the circuit being built had a feed back loop. Sequential circuits also require several new primitives to capture the semantics of a circuit as an infinite sequence of values one for each clock cycle. These primitive include the `(Const n)` operator that returns a sequential circuit that returns `n` on all clock cycles, and the `(Delay n x)`, which returns `n` in the first clock cycle, and then returns the values returned by `x`, only delayed by one clock cycle. To do this, we again enrich the `Exp` GADT.

```
 data Exp ::  *0 ~> *0 where
   . . .
   Delay  :: a -> Exp a -> Exp a
   Const  :: a -> Exp  a
   RLet   :: Decs t -> Exp t       -- Like "Let" but denotes recursive binding
```

We can indicate feed back loops by expanding the use of named circuits. If we allow the definition of names inside of a `circuit` to be mutually recursive, then feed back loops become possible with no additional syntax. Consider:

```
toggle change =
  circuit  out2  where
     out1 = Delay Low out2
     out2 = Xor change out1
```

This is transformed by the compiler into a definition that replaces the recursive equations by an expression built by using the constructor functions of the `Exp` datatype. These constructor functions make explicit the recursion implicit in the equations. There is considerable analysis going on here, and this is one reason why an embedded approach comes up short. The expanded declaration produced is as follows:

```
toggle change =
  (RLet (Decl (PVar "out1" Bit) (Delay Low (Var "out2" Bit))
        (Decl (PVar "out2" Bit) (Xor (Input "x" Bit) (Var "out1" Bit))
        (In (Var "out2" Bit)))))
```

The translation process is the same as for named circuits, except here the naming is mutually recursive, as expressed by the use of `RLet` instead of `Let`. The analysis discovers the recursion and inserts the `RLet`. Note also, that the `Rep` types inside of `Var` and `PVar`, which is only implicit in the unexpanded `circuit` syntactic sugar, is made explicit by the expansion process. The type checker reconstructs these so that the user does not have to. It would be possible to have only one kind of naming, where all naming is potentially recursive. We intend to explore that design decision in the near future.

### 3.7 The Importance of Syntactic Sugar

We wish to emphasize that users of the HDL need know nothing about the constructors for declarations (`In` and `Decl`), variables (`Var` and `PVar`), and binding (`Let` and `RLet`). These constructors are completely abstracted by the `circuit` syntactic sugar. In an embedded approach none of this would be possible.

### 3.8 The Truth Revealed

The story of named circuits is plausible, but it doesn't quite stand up to inspection. First, it is susceptible to inadvertent name capture. Second, it doesn't really solve the duplication problem. To illustrate the first point consider.

```
f x = circuit (halfAdd carry x)
       where carry = I
```

This expression uses nested `circuit` syntax. It expands to nested `Let` expressions, but they both employ the same local name `carry`.

```
f x =
  Let (Decl (PVar "carry" Bit) I
      (In (Let (Decl (PVar "carry" Bit) (Xor (Var "carry" Bit) x)
               (Decl (PVar "sum" Bit) (And (Var "carry" Bit) x)
               (In #(Var "carry" Bit,Var "sum" Bit)))))))
```

Note how the second `Decl` for `carry` hides the first one. To illustrate the second problem consider.

```
f x = And x x
g x y = f (halfAdd x y)
```

This just duplicates the `Let` expression in each argument of the `And` constructor. What we need is to push the `And x x` inside the body of the `Let` expression. Fortunately, the meta-programming community has developed standard solutions to both of these problems. The first is solved by a generative approach to naming. I.e. every time a `circuit` expression is expanded, a fresh version for each name is generated. This approach has been well studied in both MetaML[1, 12, 11] and Template Haskell[8]. There are even techniques for proving properties of programs generated in this way[10]. The second is solved by the use of continuations. The great thing about both these approaches is that users can be completely unaware of the implementation mechanism. Using these approaches the first example expands to:

```
f x =
  Let (Decl (PVar "carry_1" Bit) I
      (In (Let (Decl (PVar "carry_2" Bit) (Xor (Var "carry_1" Bit) x)
               (Decl (PVar "sum_3" Bit) (And (Var "carry_1" Bit) x)
               (In #(Var "carry_2" Bit,Var "sum_3" Bit)))))))
```

And the second: `f (halfAdd x y)` expands to:

```
Let (Decl (PVar "carry_1" Bit) (Xor x y)
    (Decl (PVar "sum_2" Bit) (And x y)
    (In (And (Var "carry_1" Bit) (Var "sum_2" Bit)))))
```

We discuss the implementation details of a generative approach to name generation and the use of continuations in more detail in Section 4

# 4 The Signal Datatype

To handle possible inadvertent name capture and to control duplication, we introduce the `Sig` type constructor. A signal is just an abstraction which produces an `Exp` datatype, but which hides a few details. These details solve the problems discussed in Section 3.8. Signals employ a combination of two techniques: the state monad, and the continuation monad. It is useful to discuss each technique in isolation.

**The state monad for name generation.** The generative approach to naming must generate new, unique names when a named signal is created. To do this we need a little bit of state from which to generate new names. For example, if we were only interested in name generation we might define `Sig` as follows:

```
data Sig t = Sig (Int -> (Exp t,Int))

and :: Sig Bit -> Sig Bit -> Sig Bit
and (Sig f) (Sig g) = Sig h
    where h n = let (x,n1) = f n
                    (y,n2) = g n1
                in (And x y,n2)
```

Each of the operators of primitive circuits (`And`, `Or`, `Xor`, etc.) are easily lifted to the signal type as illustrated for the `And` case. Each signal (implicit in the function `f` and `g`), is called, possibly generating a new integer (`n1` and `n2`) which is propagated unchanged inside the definition of `And` because it does no naming. To illustrate how generative naming is performed consider the simple named circuit.

```
f :: Sig Bit -> Sig Bit -> Sig Bit
f x y = circuit z where z = and x y
```

This might be expanded to

```
f x y = Sig h
  where h n = let (z,zpat) = gensym n "z" Bit
                  (ans,n2) = runSig (and x y) (n+1)
              in (Let (Decl zpat (In ans)),n2)

runSig (Sig f) n = f n

gensym :: Int -> String -> Rep t -> (Exp t,Pat t)
gensym s t = (Var name t,PVar name t)
  where name = s++show n
```

The current integer (`n`) is used to generate a new name, and the body of the `circuit` is "run" with the next integer (`n+1`). The resulting `Exp` (`ans`), and the ultimate integer (`n2`) are used to assemble the result.

**The continuation monad.** To avoid duplication, naming a sub-circuit must propagate the name of the circuit to all possible future circuits. To do this we use continuations. For example, if we were only interested in name propagation we might define `Sig` as follows:

```
data Sig a = Sig (forall c . (Exp a -> Decs c) -> Decs c)

and :: Sig Bit -> Sig Bit -> Sig Bit
and (Sig f) (Sig g) = Sig h
    where h k = f (\ x ->
                g (\ y -> k(And x y)))
```

Here we lift each primitive into the Sig type as illustrated by the definition of and. Note how the continuation k is pushed inside on the result of anding together the result of running the two input computations. Now, to propagate new names to all future computations we proceed by using the new name operation.

```
name :: Pat a -> Sig a -> Sig a
name p (x@(Sig f)) = Sig h
  where h k = Decl p (Let (f In)) (k (pat2exp p))
```

Given a name p and a computation x we want to name in all future circuit expressions, we start by building a new declaration for p. Note how all future signals will be in the scope of this name since these signals are implicit in the continuation k, and k expands in the body of the new declaration. The tricky part is to run x in the simplest continuation In, and then use the resulting declarations to build an Exp whose value will be bound to the name p. Note that when f is applied to a continuation, it always expands into its own sequence of Decls. Thus the shape of the result will be something like.

```
Decl p (Let (Decl_1 p_1 e_1 ... (Decl_n p_n e_n (In f_result))))
       (k (Var nm ty))
```

In many interpretations of Exp this is equivalent to the following expression which normalizes the form of declarations:

```
(Decl_1 p_1 e_1 ...
(Decl_n p_n e_n
(Decl p f_result
(In (k (Var nm ty))))))
```

so it might be ok to define a name function that builds declarations in normal form.

```
name :: Pat a -> Sig2 a -> Sig2 a
name p (Sig2 f) = Sig2 h
  where h k = f (\ res -> Decl p res (k (pat2exp p)))
```

**The combined approach.** So while this propagates names, it doesn't generate fresh ones. To do this we need to combine name generation and continuations. Thus we might consider defining a signal as follows

```
Sig a = (forall c . Int -> (Int -> Exp a -> Decs c) -> Decs c)
```

It turns out that it is more convenient to define a slightly more generic type, which is a monad, and then to define a signal as an instance of this monadic type.

```
data M a = M (forall c . Int -> (Int -> a -> Decs c) -> Decs c)
unM (M x) = x
runM (M f) = f 0 (\ s -> \x -> x)

return x = M h where h s k = k s x
bind (M a) f = M h where h s k = a s (\ s' b -> unM (f b) s' k)
fail s = error s
monadM = Monad return bind fail
next s = M h where h n k = k (n+1) (s++"_"++show n)

data Sig a = Sig (M (Exp a))
unSig (Sig x) = x
```

All of the primitive circuit constructors are easily lifted to this new type. We follow the convention that the lifted operators (except in a few cases) use the same name as the Exp constructor functions, but don't use an uppercase letter as the first character in their name.

```
lift0 :: Exp a -> Sig a
lift0 x = Sig(return x)

lift1 :: (Exp a -> Exp b) -> (Sig a -> Sig b)
lift1 f (Sig x) =
  Sig(do { a <- x; return(f a)}) where monad monadM

lift2 :: (Exp a -> Exp b -> Exp c) -> (Sig a -> Sig b -> Sig c)
lift2 f (Sig x) (Sig y) =
  Sig(do { a <- x; b <- y; return(f a b)}) where monad monadM

mux = lift3 If
pair = lift2 Pair
cempty = lift0 CEmpty
cadd = lift2 CAdd
o = lift0 O
i = lift0 I
and = lift2 And
or = lift2 Or
xor = lift2 Xor
not1 = lift1 Not
xor2 = lift2 Xor
lit n = lift0(Lit n)
equal = lift2 Equal
plus = lift2 Plus
minus  = lift2 Minus
times  = lift2 Times
delay a = lift1 (Delay a)
const a = lift0 (Const a)
```

Because the new definition of signals includes a mechanism for generating fresh names, we can define a `name` operation that does not need a name as input, but generates a fresh one on demand.

```
name (Sig (M f)) = Sig(M h) where
  h n k = f n (\ n2 res ->
                 let t = (typeOf res)
                     name = ("x"++"_"++show n2)
                 in Decl (PVar name t) res (k (n2+1) (Var name t)))
```

Now the user simply uses the lifted versions of the primitive operators. All the naming machinery is still hidden inside the `circuit` syntactic sugar.

```
toggle3 change =
  circuit out2 where
     out1 = delay Low out2
     out2 = xor change out1
```

## 4.1 Obtaining Expressions from Signals

Now suppose we wish to test one of our signal description generators. We will need two things: (1) a way of generating abstract inputs, and (2) a way of turning `(Sig t)` into `(Exp t)`. Both are easy. First enrich the `Exp` datatype with a notion of an abstract input with a new constructor `Input`, and then define the function `go` that turns a signal into a data structure and then prints out the result.

```
data Exp ::  *0 ~> *0 where
  . . .
  Input :: String -> Rep t -> Exp t

arg :: String -> (Rep a) -> Sig a
arg s t = lift0 (Input s t)

go :: Sig a -> String
go (Sig (M f)) = showC(RLet (f 0 (\ n x -> In x)))
```

The function `arg` lifts abstract inputs from the `Exp` level to the `Sig` level. The function `go` forces the computation inside a signal by applying it to an initial integer `0`, and an initial continuation `In`, and then prints the result. We can now test one of our circuit descriptions by evaluating `go (toggle3 (arg "A" Bit))` which produces the following string as output

```
"circuit out2_1 where
   out1_0::Bit = (Delay Low out2_1)
   out2_1::Bit = (Xor A out1_0)"
```

If we want to analyze a circuit description, we simply skip the use of the printing function `showC`. Note that expressions are still typed, and the types help us ensure that are analysis are correct.

# 5 Static Analyses

A static analysis computes a static property of a value or function. The property can be determined by looking at the program without running it. Type systems are one type of static analysis. An easy way to specify a static analysis is by using an inference system. The inference system states how to build the property of a large program from the properties of its smaller sub-programs. We have already seen a simple static property of `Exp`. That property is its type index. It states that *every* expression with type `(Exp t)` denotes a circuit with type `t`. Note that functions over `Exp` propagate this property. For example: `halfAdd2 :: (Exp Bit) -> (Exp Bit) -> Exp(Bit,Bit)` states that given two expressions with static property `Bit` it creates an expression with static property `(Bit,Bit)`.

We specified the property for the constructors of `Exp` using the `data` declaration and then we inferred (or checked) the property for every function. There is no reason we cannot duplicate this process. Give rules for inferring the property for each constructor, then infer (or check) the rules for every function. For example, a circuit built without using the `RLet`, `Const`, or `Delay` constructor can always be given an interpretation as a combinational circuit. Can we track this property statically? The answer is yes.

```
kind Ctype = Comb | Seq

property circuitType of
  Pat :: Ctype ~> *0 where
    PVar :: String -> (Rep t) -> Pat ct
    PPair :: Pat ct -> Pat ct -> Pat ct

  Exp ::  Ctype ~> *0 where
    Var    :: String -> Rep t -> Exp ct
    If     :: Exp ct -> Exp ct -> Exp ct -> Exp ct
    Let    :: Decs ct -> Exp ct
    Pair   :: Exp ct -> Exp ct -> Exp ct
    CEmpty :: Exp ct
    CAdd   :: Exp ct -> Exp ct -> Exp ct
```

```
    O      :: Exp ct
    I      :: Exp ct
    And    :: Exp ct -> Exp ct -> Exp ct
    Or     :: Exp ct -> Exp ct -> Exp ct
    Not    :: Exp ct -> Exp ct
    Xor    :: Exp ct -> Exp ct -> Exp ct
    Lit    :: ct -> Exp ct
    Equal  :: Exp ct -> Exp ct -> Exp ct
    Plus   :: Exp ct -> Exp ct -> Exp ct
    Minus  :: Exp ct -> Exp ct -> Exp ct
    Times  :: Exp ct -> Exp ct -> Exp ct
    Delay  :: a -> Exp ct -> Exp Seq
    Const  :: a -> Exp  Seq
    RLet   :: Decs t -> Exp Seq
    Input :: String -> Rep t -> Exp ct


  Decs :: Ctype ~> *0  where
    In :: Exp t -> Decs ct
    Decl :: Pat ct -> Exp ct -> Decs ct -> Decs ct
```

To introduce a new property, define a new kind (`Ctype`) whose structure is the carrier of the property. Here we have two kinds of circuits: combinatorial (`Comb`), and sequential (`Seq`). Then for each datatype involved (`Pat`, `Exp`, and `Decs`), give an inference system where the new property is an index of the datatype. The similarity between `data` declaration and `property` declaration is intentional. The property defines an alternate type system for the constructors. In the property `circuitType` most of the constructors are polymorphic in their `Ctype` index. Only `RLet`, `Const`, and `Delay` force the introduction of the `Seq` type index. Thus an expression without one of these constructors will have a polymorphic type in the alternate `circuitType` inference system.

When we define a new function we can give multiple prototype declarations. One for its type, and one for each static property we want to check. For example:

```
circuitType bitAdder :: Exp a -> Vec (Exp a) n -> (Vec (Exp a) n, Exp a)

bitAdder :: Exp Bit -> Vec (Exp Bit) n -> (Vec (Exp Bit) n, Exp Bit)

bitAdder carry_in Empty = (Empty,carry_in)
bitAdder carry_in (Add x xs) = (Add y ys,carry_out)
  where #(y,carry) = halfAdd2 carry_in x
        (ys,carry_out) = bitAdder carry xs
```

Here we state that `bitAdder` does not necessarily force its output to be a sequential circuit, but propagates the `Ctype` property of its inputs. The compiler checks this property by type checking the body twice. First checking that it has the normal type, and second by using the alternate `circuitType` types for the constructors (`Add`), and helper functions (`halfAdd2`). An error in either check causes a compile-time error.

# 6 Multiple Interpretations

Once a circuit description is built, users want to interpret it in many ways. It should be possible to do this, in fact this is why we build descriptions in the first place. We've already seen several interpretations of `Exp`: `eval` and `typeOf`. Other examples of interpretations are:

```
eval1 :: Exp t -> Env1 -> [t]             -- an infinite sequence of t values
                                          -- one for each clock cycle

eval2 :: Exp t -> Env2 -> (Time -> t)     -- A function from Time to t

eval3 :: Exp t -> Env3 -> Code(Time -> t) -- The Haskell code of a function
                                          -- from Time to t

numGates :: Exp t -> Int                  -- Number of primitive gates

depth :: Exp t -> Int                     -- depth of the circuit
```

In the first three examples we need an environment to handle free variables in expressions. The structure of the environment will differ, depending on what the function is doing. For space reasons we illustrate only `eval2`, and we only show some of the cases. The missing cases are all similar to the `And` and `Plus` cases.

```
type Time = Int
data Entry2 = exists t . E2 String (Rep t) (Time -> t)
type Env2 = [Entry2]

find2 :: String -> Rep t -> Env2 -> (Time -> t)
find2 s t [] = error ("Not found: "++show s)
find2 s t ((E2 nm z v):xs) = case (eqStr s nm,testRep t z) of
                               (True,Just Eq) -> v
                                _ -> find2 s t xs

eval2 :: Exp t -> Env2 -> (Time -> t)
eval2 (Var s t) env = find2 s t env
eval2 (And x y) env = \ t -> andB (eval2 x env t) (eval2 y env t)
eval2 (Plus x y) env = \ t -> (eval2 x env t) + (eval2 y env t)
eval2 (Pair x y) env = \ t -> (eval2 x env t, eval2 y env t)
eval2 (Const n) env = \ t -> n
eval2 (Delay n x) env = \ t -> if t==0 then n else (eval2 x env (t-1))
eval2 (Let (In e)) env = eval2 e env
eval2 (Let (Decl (PVar nm t) e ds)) env =
  eval2 (Let ds) ((E2 nm t (eval2 e env)):env)
eval2 (RLet ds) env = eval2 (last ds) env2
  where env2 = map ds
        map (Decl p e ds) =  bindPat2 p (lazy(eval2 e env2)) (map ds)
        map (In e) = env
        last (Decl p e ds) = last ds
        last (In e) = e

bindPat2 :: Pat t -> (Time -> t) -> Env2 -> Env2
bindPat2 (PVar nm t) v env = (E2 nm t v):env
```

# 7 Smart Circuits

Recently Mary Sheeran has described how to use clever circuits[9] to build not-quite regular structures. The key idea here is to tag each circuit with a value that circuit building combinators can observe to direct how they should combine the smaller circuits into larger ones. We can accommodate this by adding an extra slot in each expression to store the tag.

```
data TagExp ::  *0 ~> *0 ~> *0 where
  Var    :: tag -> String -> Rep t -> Exp tag t
  If     :: tag -> Exp tag Bit -> Exp tag b -> Exp tag b -> Exp tag b
  Let    :: tag -> Decs tag t -> Exp tag t
  O      :: tag -> Exp tag Bit
  I      :: tag -> Exp tag Bit
  And    :: tag Exp Bit -> Exp tag Bit -> Exp tag Bit
```

Of course, each combinator that does something special with the tags will have to be individually encoded.

# 8 Hardware Description and Domain Specific Languages

A Hardware description language (HDL) is a domain specific language. As such it can have a number of advantages over using a general purpose language for hardware design. In this section we try to open a discussion about the future of DSL's for hardware design. I am deeply indebted to John O'Leary for many of the ideas discussed in this section. First let's take stock of the current state of affairs.

- A HDL should adopt an existing notation, and allow designers to create large designs with little or no programming experience. The HDL language syntax is often different than that of existing notation because high-level HDL's attempt to do things existing notations were not designed to do.

  While syntactical issues are important, a bigger problem is achieving a smooth `semantic` integration between a high-level HDL and an existing notation. Ideally one should not have to switch mind sets when switching between "existing notation mode" and "meta mode". This is a challenge in any meta-programming system.

- A HDL program provides a common language that allows different database users to share precisely the description of the hardware they are interested in. An HDL communicates ideas.

- A HDL program relieves the designer from needing to know about various aspects of the hardware design process, such as the data structures that store the low level description, or the layout process that maps the design onto silicon. These issues must eventually be addressed, but they can be abstracted away in early steps of the design.

- A HDL program encodes constraints. Constraints could include area, timing, and power requirements. It's not clear how analysis of these properties would fit with type checking, or even whether it should. In general, design constraints get handed down from management or marketing (e.g. it has to be an x86, consume less than 100 watts of power, run at 3 GHz, and fit on a die so big). As the implementation is developed the constraints need to be maintained and pushed down to sub-blocks, but there's often not enough good information to say for certain whether a block fulfills its constraints. This is because the most accurate estimates come bottom-up from the detailed levels of design, and you don't get there until the final few months of a project.

  Also, some things are quite difficult to estimate statically - we might be interested not in worst-case power but in average power over some benchmark suite. In this case there's no substitute for dynamic simulation.

- A HDL program can take advantage of domain specific properties. Especially in the logic synthesis process. Given a Boolean function and a set of constraints (e.g. area, power, delay), generate a gate-level implementation meeting the constraints. The constraints (and physics) are what make it more than just simplification, and design principles used here would be hard to encode in a general purpose programming language. There's a whole research community built around "physically aware synthesis", and an HDL should incorporate this knowledge.

- A HDL program is a rapid prototype. An HDL description should be easy to write and lets designer experiment with different solutions. If the results of a design does not have the properties intended, the designer is free to throw the original design away and try another without excessive losses in time or resources.

- An HDL design should integrate with a number of existing contexts or environments, such as timing analysis, and layout systems.

- An HDL design should support a natural notion of refinement. For example, may design a sequential circuit over integers, and later refine integer to a fixed 32-bit, twos-complement bus. The vision is that one writes high level models of various blocks, validates that they work together correctly, and then uses these models as a test harness for more detailed designs (e.g. by substituting a gate-level block for one of the high level blocks and running simulation again).

  But what is missing from this picture is a formal account of refinement - when can a block be said to refine another? Does refinement preserve properties that were verified on the abstract model? Besides data refinement designers are interested in refinement in several other dimensions.

  1. **Temporal refinement.** An abstract model of a pipeline might have an incompletely-specified delay from input to output (e.g. between four and six cycles). A more concrete model may have the delays nailed down.
  2. **Interface refinement.** An abstract model might communicate via CSP-style channels:

     c!v - send value v along channel c
     c?x - receive a value on channel c and bind it to variable x

     A more concrete model would implement some sort of handshaking scheme. There are several interesting things here. At the abstract level, the sender views a channel as an output and the receiver views it as an input. At the concrete level, channel_imp has a mixture of inputs and outputs - to the sender, request and data are outputs while acknowledge is an input (and vice versa for the receiver). A type system supporting interface abstraction had better account for this. Another interesting thing is that the abstract level has atomic send and receive operations (! and ?). The concrete level implements these by a protocol involving a sequence of events. What does refinement mean here?

- A HDL design is an efficient division of labor. Programming language experts design the HDL language, algorithm experts design the algorithms used, and domain experts design the devices.

  1. HDLs facilitate division of labor in a design project because a small set of architects can develop the overall product design as an interconnection of major blocks with each block modeled abstractly, and farm the detailed design of each block to specialists. This also occurs within major blocks as they are partitioned into sub-blocks.
  2. HDLs facilitate division of labor in CAD tool development. The intermediate data format or "data model" is more important. The data model is the object code for the HDL compiler, and should preserve all the content of the original HDL description - logical functionality, constraints, etc. With a perfect data model experts in the backend tasks of synthesis, optimization, layout can all do their thing without having to understand the whole problem.

### 8.1 Domain-Specific Language Evolution

The life cycle of a DSL may be thought of as having three stages, each calling for different implementation approaches.

**Infancy.** In this stage, language design ideas are in flux and domain abstractions are not well understood. While some domains have existing conceptual frameworks and notations that can be used for problem specification, many domains must be analyzed and studied to identify the key abstractions. At this stage, modeling flexibility is at a premium, performance is less important, and embedded implementations are likely to be cost-effective. *I believe we have passed this point in the life-cycle of HDLs.*

**Adolescence.** In this stage, domain abstractions stabilize and become better-understood as the language begins to be exercised by end-users. Implementation flexibility is still necessary. Actual use uncovers the limits of the implementation, which leads to new features, abstractions, and implementation techniques. In this stage users begin to develop domain-specific logics for their DSLs. As more users exercise the system, the need for specialized language syntax, type discipline, and error reporting become more apparent. These are difficult or impossible to provide in a conventional host language. Because most host languages lack the ability to reflect on their own code – *i.e.*, to analyze it as well as execute it – they make it very difficult to enhance an embedded DSL implementation with validation or optimization passes. *I believe this is exactly where we currently stand in the HDL life cycle. Consider the work on monads[**?**], observable sharing[3], and labeling[**?**], as attempts to over come this restriction.*

**Maturity.** In this stage, domain abstractions are well-understood, and the DSL design is fixed. Features that enhance usability – such as type systems, debuggers, and programming environments – become important as the DSL gains adherents. Performance of the DSL implementation itself may become a problem as DSL program size and complexity increase.

A stand-alone implementation for a DSL can have its own syntax and type system appropriate for just that domain. The DSL can be "restricted" to enforce constraints on what can be expressed, and it can have its own optimizer that relies on domain-specific knowledge so that performance bottlenecks can be addressed. If investing in a stand-alone DSL implementation ever makes sense, it will be at this phase, when the language has become relatively stable and user tools are important. *This is where we are headed, but we need to rethink a few of our assumptions if we're ever to get here.*

## References


[1] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, Rice University, 2002.

[2] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003. Also available from: `http://www.informatik.uni-bonn.de/~ralf/publications/Phantom.pdf`.

[3] Koen Claessen. *An Embedded Language Approach to Hardware Description and Verification*. PhD thesis, Chalmers University of Technology, 2000.

[4] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *IFIP International Conference on Theoretical Computer Science (TCS)*, pages 448–460, August 2002.

[5] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. `http://research.microsoft.com/Users/simonpj/`, 2004.

[6] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. *ACM SIGPLAN Notices*, 35(1):109–122, January 2000.

[7] Tim Sheard. Languages of the future. *Onward Track, OOPSLA'04. Reprinted in: ACM SIGPLAN Notices, Dec. 2004.*, 39(10):116–119, October 2004.

[8] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM Press, 2002.

[9] Mary Sheeran. Generating fast multipliers using clever circuits. In *FMCAD'04*, volume 3312 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[10] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.

[11] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. Or, the theory of MetaML is non-trivial. In *2000 SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, January 2000.

[12] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.