Type-Safe Observable Sharing in Haskell

Andy Gill

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

Abstract

Haskell is a great language for writing and supporting embedded Domain Specific Languages (DSLs). Some form of observable sharing is often a critical capability for allowing so-called deep DSLs to be compiled and processed. In this paper, we describe and explore uses of an `IO` function for reification which allows direct observation of sharing.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

General Terms Design, Languages

Keywords Observable Sharing, DSL Compilation

1. Introduction

Haskell is a great host language for writing Domain Specific Languages (DSLs). There is a large body of literature and community know-how on embedding languages inside functional languages, including shallow embedded DSLs, which act directly on a principal type or types, and deep embedded DSLs, which construct an abstract syntax tree that is later evaluated. Both of these methodologies offer advantages over directly parsing and compiling (or interpreting) a small language. There is, however, a capability gap between a deep DSL and compiled DSL, including observable sharing of syntax trees. This sharing can notate the sharing of computed results, as well as also notating loops in computations. Observing this sharing can be critical to the successful compilation of our DSLs, but breaks a central tenet of pure functional programing: referential transparency.

In this paper, we introduce a new, retrospectively obvious way of adding observable sharing to Haskell, and illustrate its use on a number of small case studies. The addition makes nominal impact on an abstract language syntax tree; the tree itself remains a purely functional value, and the shape of this tree guides the structure of

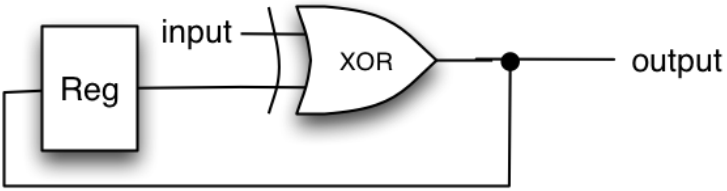
Haskell'09, September 3, 2009, Edinburgh, Scotland, UK.

a graph representation in a direct and principled way. The solution makes good use of constructor classes and type families to provide a type-safe graph detection mechanism.

Any direct solution to observable sharing, by definition, will break referential transparency. We restrict our sharing using the class type system to specific types, and argue that we provide a reasonable compromise to this deficiency. Furthermore, because we observe sharing on regular Haskell structures, we can write, reason about, and invoke pure functions with the same abstract syntaxes sans observable sharing.

2. Observable Sharing and Domain Specific Languages

At the University of Kansas, we are using Haskell to explore the description of hardware and system level concerns in a way that is suitable for processing and extracting properties. As an example, consider a simple description of a bit-level parity checker.



This circuit takes a stream of (clocked) bits, and does a parity count of all the bits, using a bit register. Given some Haskell functions as our primitives, we can describe this circuit in a similar fashion to Lava (Bjesse et al. 1998), Hawk (Matthews et al. 1998), and Hydra (O'Donnell 2002). For example, the primitives may take the form

```
-- DSL primitives
xor :: Bit -> Bit -> Bit
delay :: Bit -> Bit
```

where `xor` is a function which takes two arguments of the abstract type `Bit`, performing a bit-wise xor operation, and `delay` takes a single `Bit` argument, and outputs the bit value on the previous clock cycle (via a register or latch). Jointly these primitives provide an interface to a μLava.

These abstract primitives allow for a concise specification of our circuits using the following Haskell.

```
-- Parity specification
parity :: Bit -> Bit
parity input = output
  where
```

```
      output = xor (delay output) input
```

We can describe our primitives using a *shallow* DSL, where `Bit` is a stream of boolean values, and `xor` and `delay` act directly on values of type `Bit` to generate a new value, also of type `Bit`.

```
-- Shallow embedding
newtype Bit = Bit [Bool]

xor :: Bit -> Bit -> Bit
xor (Bit xs) (Bit ys) = Bit $ zipWith (/=) xs ys

delay :: Bit -> Bit
delay (Bit xs) = Bit $ False : xs

run :: (Bit -> Bit) -> [Bool] -> [Bool]
run f bs = rs
  where
     (Bit rs) = f (Bit bs)
```

Hawk used a similar shallow embedding to provide semantics for its primitives, which could be simulated, but the meaning of a specific circuit could not be directly extracted. In order to construct a DSL that allows extraction, we can give our primitives an alternative *deep* embedding. In a deep embedding, primitives are simply Haskell data constructors, and a circuit description becomes a Haskell syntax tree.

```
-- New, deep embedding
```

```
data Bit = Xor Bit Bit
         | Delay Bit
         | Input [Bool]
         | Var String
         deriving Show

xor = Xor
delay = Delay

run :: (Bit -> Bit) -> [Bool] -> [Bool]
run f bs = interp (f (Input bs))

interp :: Bit -> [Bool]
interp (Xor b1 b2) = zipWith (/=) (interp b1)
                                  (interp b2)
interp (Delay b)  = False : interp b
interp (Input bs) = bs
interp (Var v)    = error $ "Var not supported"
```

The `run` function has the same behavior as the `run` in the shallow DSL, but has a different implementation. An interpreter function acts as a supporting literal interpreter of the `Bit` data structure.

```
> run parity (cycle True)
[True,False,True,False,True,...
```

The advantage of a deep embedding over a shallow embedding is that a deep embedding can be extracted directly for processing and analysis by other functions and tools, simply by reading the data type which encodes the DSL. Our circuit is a function, `Bit -> Bit`, so we provided the argument `(Var "x")`, where `"x"` is unique to this circuit, giving us a `Bit`, with the `Var` being a placeholder for the argument.

Unfortunately, if we consider the structure of `parity`, it contains a loop, introduced via the `output` binding being used as an argument to `delay` when defining `output`.

```
 > parity (Var "x")
 Xor (Delay (Xor (Delay (Xor (Delay (Xor (...
```

This looping structure can be used for interpretation, but not for further analysis, pretty printing, or general processing. The challenge here, and the subject of this paper, is how to allow trees extracted from Haskell hosted deep DSLs to have *observable* back-edges, or more generally, observable sharing. This a well-understood problem, with a number of standard solutions.

- **Cycles can be outlawed** in the DSL, and instead be encoded inside explicit looping constructors, which include, implicitly, the back edge. These combinators take and return functions that operate over circuits. This was the approach taken by Sharp (2002). Unfortunately, using these combinators is cumbersome in practice, forcing a specific style of DSL idiom for all loops. This is the direct analog of programing recursion in Haskell using `fix`.
- **Explicit Labels** can be used to allow later recovery of a graph structure, as proposed by O'Donnell (1992). This means passing an explicit name supply for unique names, or relying on the user to supply them; neither are ideal and both obfuscate the

essence of the code expressed by the DSL.

- **Monads**, or other categorical structures, can be used to generate unique labels implicitly, or capture a graph structure as a net-list directly. This is the solution used in the early Lava implementations (Bjesse et al. 1998), and continued in Xilinx Lava (Singh and James-Roxby 2001). It is also the solution used by Baars and Swierstra (2004), where they use applicative functors rather than monads. Using categorical structures directly impacts the type of a circuit, and our parity function would now be required to have the type

  ```
  parity :: Bit -> M Bit
  ```

  Tying the knot of the back edges can no longer be performed using the Haskell `where` clause, but instead the non-standard recursive-`do` mechanism (Erkök and Launchbury 2002) is used.
- **References** can be provided as a non-conservative extension (Claessen and Sands 1999). This is the approach taken by Chalmers Lava, where a new type `Ref` is added, and pointer equality over `Ref` is possible. This non-conservative extension is not to everyone's taste, but does neatly solve the problem of observable sharing. Chalmers Lava's principal structure contains a `Ref` at every node.

In this paper, we advocate another approach to the problem of observable sharing, namely an `IO` function that can observe sharing directly. Specifically, this paper makes the following contributions.

- We present an alternative method of observable sharing, using stable names and the `IO` monad. Surprisingly, it turns out that our graph reification function can be written as a reusable component in a small number of lines of Haskell. Furthermore, our solution to observable sharing may be more palatable to the community than the `Ref` type, given we accept IO functions routinely.

- We make use of type functions (Chakravarty et al. 2005), a recent addition to the Haskell programmers' portfolio of tricks, and therefore act as a witness to the usefulness of this new extension.
- We illustrate our observable sharing library using a small number of examples including digital circuits and state diagrams.
- We extend our single type solution to handle Haskell trees containing different types of nodes. This extension critically depends on the design decision to use type families to denote that differently typed nodes map to a shared type of graph node.
- We illustrate this extension being used to capture deep DSLs containing *functions*, as well as data structures, considerably extending the capturing potential of our reify function.

Our solution is built on the `StableName` extension in GHC (Peyton Jones et al. 1999), which allows for a specific type of pointer equality. The correctness and predicability of our solution depends on the properties of the `StableName` implementation, a point we return to in section 12.

3. Representing Sharing in Haskell

Our solution to the observable sharing problem addresses the problem head on. We give specific types the ability to have their sharing observable, via a *reify* function which translates a tree-like data structure into a graph-like data structure, in a type safe manner. We use the class type system and type functions to allow Haskell programmers to provide the necessary hooks for specific data structures, typically abstract syntax trees that actually capture abstract syntax graphs.

There are two fundamental issues with giving a type and implementation to such a reify function. First, how do we allow a graph to share a typed representation with a tree? Second, observable sharing introduces referential opaqueness, destroying referential transparency: a key tenet of functional programming. How do we contain – and reason about – referential opaqueness in Haskell? In this section, we introduce our reify function, and honestly admit opaqueness by making the reify function an `IO` function.

Graphs in Haskell can be represented using a number of idioms, but we use a simple associated list of pairs containing `Uniques` as node names, and node values.

```
type Unique = Int
data BitGraph = BitGraph [(Unique,BitNode Unique)]
                         Unique
```

```
data BitNode s = GraphXor s s
               | GraphDelay s
               | GraphInput [Bool]
               | GraphVar String
```

We parameterize `BitNode` over the `Unique` graph "edges", to facilitate future generic processors for our nodes.

Considering the parity example, we might represent the sharing using the following expression.

```
graph = BitGraph [ (1,GraphXor 2 3)
                 , (2,GraphDelay 1)
                 , (3,GraphInput "x")
                 ]
                 1
```

This format is a simple and direct net-list representation. If we can generate this graph, then using smarter structures like `Data.Map` downstream in a compilation process is straightforward. Given a `Functor` instance for `BitNode`, we can generically change the types of our nodes labels.

We can now introduce the type of a graph reification function.

```
reifyBitGraph :: Bit -> IO BitGraph
```

With this function, and provided we honor any preconditions of its use, embedding our μLava in a way that can have sharing extracted is trivial. Of course, the `IO` monad is needed. Typically, this reify replaces either a parser (which would use `IO`), or will call another

IO function later in a pipeline, for example to write out VHDL from the BitGraph or display the graph graphically. Though the use of IO is not present in all usage models, having IO does not appear to be a handicap to this function.

4. Generalizing the Reification Function

We can now generalize reifyBitGraph into our generic graph reification function, called reifyGraph. There are three things reifyGraph needs to be able to do

- First, have a target type for the graph representation to use as a result.
- Second, be able to look inside the Haskell value under consideration, and traverse its structure.
- Third, be able to build a graph from this traversal.

We saw all three of these capabilities in our reifyBitGraph example. We can incorporate these ideas, and present our generalized graph reification function, reifyGraph.

```
reifyGraph :: (MuRef t)
           => t -> IO (Graph (DeRef t))
```

The type for reifyGraph says, given the ability to look deep inside a structure, provided by the type class MuRef, and the ability to derive the shared, inner data type, provided by the *type function* DeRef, we can take a tree of a type that has a MuRef instance, and build a graph.

The Graph data structure is the generalization of BitGraph, with nodes of the higher kinded type e, and a single root.

```
type Unique = Int
data Graph e = Graph [(Unique,e Unique)]
                     Unique
```

Type functions and associated types (Chakravarty et al. 2005) is a recent addition to Haskell. reifyGraph uses a type function to determine the type of the nodes inside the graph. Associated types allow the introduction of data and type declarations inside a class declaration; a very useful addition indeed. This is done by literally providing *type functions* which look like standard Haskell type constructors, but instead use the existing class-based overloading system to help resolve the function. In our example, we have the type class MuRef, and the type function DeRef, giving the following (incomplete) class declaration.

```
class MuRef a where
  type DeRef a :: * -> *
  ...
```

This class declaration creates a type function DeRef which acts like a type synonym inside the class; it does not introduce any constructors or abstraction. The * -> * annotation gives the kind of DeRef, meaning it takes two type arguments, the relevant instance of MuRef, and another, as yet unseen, argument. DeRef can be assigned to any type of the correct kind, inside each instance.

In our example above, we want trees of type Bit to be represented as a graph of `BitNode`, so we provide the instance `MuRef`.

```
instance MuRef Bit where
  type DeRef Bit = BitNode
  ...
```

`BitNode` is indeed of kind `* -> *`, so the type of our `reifyGraph` function specializes in the case of `Bit` to

```
reifyGraph :: Bit -> IO (Graph (DeRef Bit))
```

then, because of the type function `DeRef`, to

```
reifyGraph :: Bit -> IO (Graph BitNode)
```

The use of the type function `DeRef` to find the `BitNode` data-type is critical to tying the input tree to type node representation type, though functional dependencies (Jones and Diatchki 2008) could also be used here.

The `MuRef` class has the following definition.

```
class MuRef a where
  type DeRef a :: * -> *
  mapDeRef     :: (Applicative f)
               => (a -> f u)
               -> a
               -> f (DeRef a u)
```

mapDeRef allows us, in a generic way, to reach into something that has an instance of the MuRef class and recurse over relevant children. The first argument is a function that is applied to the children, the second is the node under consideration. mapDeRef returns a single node, the type of which is determined by the DeRef type function, for recording in a graph structure. The result value contains unique indices, of type u, which were generated by the invocation of the first argument. mapDeRef uses an applicative functor (McBride and Patterson 2006) to provide the threading of the effect of unique name generation.

To complete our example, we make Bit an instance of the MuRef class, and provide the DeRef and mapDeRef definitions.

```
instance MuRef Bit where
  type DeRef Bit = BitNode
  mapDeRef f (Xor a b)  = GraphXor <$> f a <*> f b
  mapDeRef f (Delay b)  = GraphDelay <$> f b
  mapDeRef f (Input bs) = pure $ GraphInput bs
  mapDeRef f (Var nm)   = pure $ GraphVar nm
```

This is a complete definition of the necessary generics to provide reifyGraph with the ability to perform type-safe observable sharing on the type Bit. The form of mapDeRef is regular, and could be automatically derived, perhaps using Template Haskell (Sheard and Peyton Jones 2002). With this instance in place, we can use our general reifyGraph function, to extract our graph.

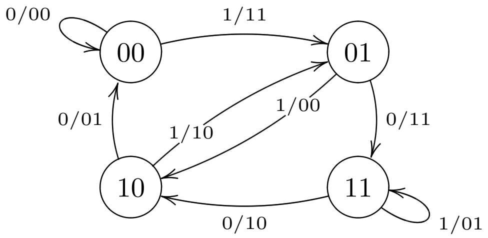
```
> reifyGraph $ parity (Name "x")
Graph [ (1,GraphXor 2 3)
```

```
        , (2,GraphDelay 1)
        , (3,GraphInput "x")
        ]
        1
```

The `reifyGraph` function is surprisingly general, easy to enable via the single `instance` declaration, and useful in practice. We now look at a number of use cases and extensions to `reifyGraph`, before turning to its implementation.

5. Example: Finite State Machines

As a simple example, take the problem of describing a state machine directly in Haskell. This is easy but tedious because we need to enumerate or label the states. Consider this state machine, a 5-7 convolutional encoder for a viterbi decoder.



One possible encoding is a `step` function, which takes input, and the current state, and returns the output, and a new state. Assuming that we use Boolean to represent 0 and 1, in the input and output, we can write the following Haskell.

```
data State = ZeroZero | ZeroOne | OneZero | OneOne
type Input = Bool
type Output = (Bool,Bool)

step :: Input -> State -> (Output,State)
step False ZeroZero = ((False,False),ZeroZero)
step True  ZeroZero = ((True ,True ),ZeroOne)
step False ZeroOne  = ((True ,True ),OneOne)
step True  ZeroOne  = ((False,False),OneZero)
step False OneZero  = ((False,True ),ZeroZero)
step True  OneZero  = ((True ,False),ZeroOne)
step False OneOne   = ((True ,False),OneZero)
step True  OneOne   = ((False,True ),OneOne)
```

Arguably more declarative encoding is to to use the *binding* as the state unique identifier.

```
data State i o = State [(i,(o,State i o))]

step :: (Eq i) => i -> State i o -> (o,State i o)
step i (State ts) = (output,st)
  where Just (output,st) = lookup i ts

state00 = State [ (False,((False,False), state01)),
                  (True, ((True ,True),  state00))]
state01 = State [ (False,((True ,True ), state11)),
                  (True, ((False,False), state10))]
state10 = State [ (False,((False,True),  state00)),
                     (True, ((True ,False), state01))]
state11 = State [ (False,((True ,False), state10)),
                  (True, ((False,True),  state11))]
```

Simulating this *binding*-based state machine is possible in pure Haskell.

```
run :: (Eq i) => State i o -> [i] -> [o]
run st (i:is) = o : run st' is
   where (o,st') = step i st
```

Extracting the sharing, for example to allow the display in the graph viewing tool `dot` (Ellson et al. 2003), is not possible in a purely functional setting. Extracting the sharing using our `reifyGraph` allows the deeper embedding to be gathered, and other tools can manipulate and optimize this graph.

```
data StateNode i o s = StateNode [ (i,(o,s)) ]
        deriving Show

instance MuRef (State i o) where
  type DeRef (State i o) = StateNode i o
  mapDeRef f (State st) = StateNode <$>
        traverse tState st
    where
         tState (b,(o,s)) = (\ s' -> (b,(o,s')))
```

```
                          <$> f s
```

Here, traverse (from the Traversable class) is a traversal over the list type. Now we extract our graph.

```
> reifyGraph state00
Graph [(1,StateNode [(False,((False,False),2))
                    ,(True,((True,True),1))
                    ])
      ,(2,StateNode [(False,((True,True),3))
                    ,(True,((False,False),4))
                    ])
      ,(3,StateNode [(False,((True,False),4))
                    ,(True,((False,True),3))
                    ])
      ,(4,StateNode [(False,((False,True),1))
                    ,(True,((True,False),2))
                    ])
      ]
      1
```

6. Example: Kansas Lava

At the University of Kansas, we are developing a custom version of Lava, for teaching and as a research platform. The intention is to allow for higher level abstractions, as supported by the Hawk DSL, but also allow the circuit synthesis, as supported by Lava. Capturing our Lava DSL in a general manner was the original motivation behind revisiting the design decision of using references for observable sharing in Chalmers Lava (Claessen 2001). In this section, we outline our design of the front end of Kansas Lava, and how it uses `reifyGraph`.

The principal type in Kansas Lava is `Signal`, which is a phantom type (Leijen and Meijer 1999) abstraction around `Wire`, the internal type of a circuit.

```
newtype Signal a = Signal Wire

newtype Wire = Wire (Entity Wire)
```

`Entity` is a node in our circuit graph, which can represent gate level circuits, as well are more complex blocks.

```
data Entity s
  = Entity Name [s] -- an entity
  | Pad Name        -- an input pad
  | Lit Integer     -- a constant

and2 :: (Signal a, Signal a) -> Signal a
and2 (Signal w1,Signal w2)
  = Signal $ Wire $ Entity (name "and2") [w1,w2]

...
```

In both Kansas Lava and Chalmers Lava, phantom types are used to allow construction of semi-sensible circuits. For example, a mux will take a `Signal Bool` as its input, but switch between polymorphic signals.

```
mux :: Signal Bool
    -> (Signal a, Signal a)
    -> Signal a
mux (Signal s) (Signal w1,Signal w2)
  = Signal
  $ Wire
  $ Entity (name "mux") [s,w1,w2]
```

Even though we construct trees of type Signal, we want to observe graphs of type Wire, because every Signal is a constructor wrapper around a tree of Wire. We share the same node datatype between our Haskell tree underneath Signal, and inside our reified graph. So Entity is parametrized over its inputs, which are Wires for our circuit specification tree, and are Unique labels in our graph. This allows some reuse of traversals, and we use instances of the Traversable, Functor and Foldable classes to help here.

Our MuRef instance therefore has the form:

```
instance MuRef Wire where
  type DeRef Wire = Entity
  mapDeRef f (Wire s) = traverse f s
```

We also define instances for the classes Traversable, Foldable and Functor, which are of general usefulness for performing other transformations, specifically:

```
instance Traversable Entity where
  traverse f (Entity v ss) = Entity v
                           <$> traverse f ss
  traverse _ (Pad v)        = pure $ Pad v
  traverse _ (Lit i)        = pure $ Lit i

instance Foldable Entity where
  foldMap f (Entity v ss) = foldMap f ss
  foldMap _ (Pad v)       = mempty
  foldMap _ (Lit i)       = mempty

instance Functor Entity where
  fmap f (Entity v ss) = Entity v (fmap f ss)
  fmap _ (Pad v)       = Pad v
  fmap _ (Lit i)       = Lit i
```

Now, with our Kansas Lava Hardware specification graph captured inside our `Graph` representation via `reifyGraph`, we can perform simple translations, and pretty print to VHDL, and other targets.

7. Comparing `reifyGraph` and `Ref` types

Chalmers Lava uses `Ref` types, which admit pointer equality. The interface to `Ref` types have the following form.

```
data Ref a = ...
```

```
instance Eq (Ref a)
ref :: a -> Ref a
deref :: Ref a -> a
```

An abstract type `Ref` can be used to box polymorphic values, via the (unsafe) function `ref`, and `Ref` admits equality without looking at the value inside the box. `Ref` works by generating a new, unique label for each call to `ref`. So a *possible* implementation is

```
data Ref a = Ref a Unique
instance Eq (Ref a) where
   (Ref _ u1) == (Ref _ u2) = u1 == u2
ref a = unsafePerformIO $ do
   u <- newUnique
   return $ Ref a u
deref (Ref a _) = a
```

with the usual caveats associated with the use of `unsafePerformIO`.

To illustrate a use-case, consider a transliteration of Chalmers Lava to use the same names as Kansas Lava. We can use a `Ref` type at each node, by changing the type of `Wire`, and reflecting this change into our DSL functions.

```
  -- Transliteration of Chalmers Lava
newtype Signal s = Signal Wire

newtype Wire = Wire (Ref (Entity Wire))
```

```
data Entity s
  = Entity Name [s]
  | ...

and2 :: Signal a -> Signal a -> Signal a
and2 (Signal w1) (Signal w2)
   = Signal
   $ Wire
   $ ref
   $ Entity (name "and2") [w1,w2]
```

The differences between this definition and the Kansas Lava definition are

- The type `Wire` includes an extra `Ref` indirection;
- The DSL primitives include an extra `ref`.

`Wire` in Chalmers Lava admits observable sharing directly, while Kansas Lava only admits observable sharing using `reifyGraph`. The structure in Kansas Lava can be consumed by an alternative, purely functional simulation function, without the possibility of accidentally observing sharing. Furthermore, `reifyGraph` can operate over an arbitrary type, and does not need to be wired into the datatype. This leaves open a new possibility: observing sharing on regular Haskell structures like lists, rose trees, and other structures. This is the subject of the next section.

8. Lists, and Other Structures

In the Haskell community, sometimes recursive types are tied using a `Mu` type (Jones 1995). For example, consider a list specified in this fashion.

```
newtype Mu a = In (a (Mu a))

data List a b = Cons a b | Nil

type MyList a = Mu (List a)
```

Now, we can write a list using `Cons`, `Nil`, and `In` for recursion. The list [1,2,3] would be represented using the following expression.

```
In (Cons 1 (In (Cons 2 (In (Cons 3 (In Nil))))))
```

The generality of the recursion, captured by `Mu`, allows a general instance of `Mu` for `MuRef`. Indeed, this is why `MuRef` is called `MuRef`.

```
instance (Traversable a) => MuRef (Mu a) where
  type DeRef (Mu a) = a
  mapDeRef = traverse
```

This generality is possible because we are sharing the representation between structures. `Mu` is used to express a tree-like structure, where `Graph` given the same type argument will express a directed graph. In order to use MuRef, we need Traversable, and therefore need to provide the instances for Functor, Foldable, and Traversable.

```
instance Functor (List a) where
  fmap f Nil        = Nil
  fmap f (Cons a b) = Cons a (f b)

instance Foldable (List a) where
  foldMap f Nil        = mempty
  foldMap f (Cons a b) = f b

instance Traversable (List a) where
  traverse f (Cons a b) = Cons a <$> f b
  traverse f Nil        = pure Nil
```

Now a list, written using Mu, can have its sharing observed.

```
> let xs =  In (Cons 99 (In (Cons 100 xs)))
> reifyGraph xs
Graph [ (1,Cons 99 2)
      , (2,Cons 100 1)
      ]
      1
```

The type List is used both for expressing trees and graphs. We can reuse List and the instances of List to observe sharing in regular Haskell lists.

```
instance MuRef [a] where
  type DeRef [a] = List
  mapDeRef f (x:xs) = Cons x <$> f xs
  mapDeRef f []     = pure Nil
```

That is, regular Haskell lists are represented as a graph, using `List`, and `Mu List` lists are also represented as a graph, using `List`. Now we can capture *spine-level* sharing in our list.

```
> let xs =  99 : 100 : xs
> reifyGraph xs
Graph [ (1,Cons 99 2)
      , (2,Cons 100 1)
      ]
      1
```

There is no way to observe built-in Haskell data structures using `Ref`, which is an advantage of our reify-based observable sharing.

A list spine, being one dimensional, means that sharing will always be represented via back-edges. A tree can have both loops and acyclic sharing. One question we can ask is can we capture the second level sharing in a list? That is, is it possible we observe the difference between

`let x = X 1 in [x,x]` and `[X 1,X 1]`

using reifyGraph? Alas, no, because the type of the *element* of a list is distinct from the type of the list itself. In the next section, we extend reifyGraph to handle nodes of different types inside the same reified graph.

9. Observable Sharing at Different Types

The nodes of the graph inside the runtime system of Haskell programs have many different types. In order to successfully extract deeper into our DSL, we want to handle nodes of different types. GHC Haskell already provides the Dynamic type, which is a common type for using with collections of values of different types. The operations are

```
data Dynamic = ...
toDyn       :: Typeable a => a -> Dynamic
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

Dynamic is a monomorphic Haskell object, stored with its type. fromDyn succeeds when Dynamic was constructed and extracted at the same type. Attempts to use fromDynamic at an incorrect type always returns Nothing. The class Typeable is derivable automatically, as well as being provided for all built-in types. So we have

```
> fromDynamic (toDyn "Hello") :: Maybe String
Just "Hello"
> fromDynamic (toDyn (1,2)) :: Maybe String
Nothing
```

In this way Dynamic provides a type-safe cast.

In our extended version of reifyGraph, we require all nodes that need to be compared for observational equality to be a member of the class Typeable, including the root of our Haskell structure we are observing. This gives the type of the extended reifyGraph.

```
reifyGraph :: (MuRef s, Typeable s)
           => s -> IO (Graph (DeRef s))
```

The trick to reifying nodes of different type into one graph is to have a common type for the graph representation. That is, if we have a type A and a type B, then we can share a graph that is captured to Graph C, provided that DeRef A and DeRef B both map to C. We can express this, using the new ~ notation for type equivalence.

Specifically, the type

```
example :: (DeRef a ~ DeRef [a]) => [a]
```

expresses that a and [a] both share the same graph node type.

In order to observe sharing on nodes of types that are Typeable, and share a graph representation type, we refine the type of mapDeRef. The refined MuRef class has the following definition.

```
class MuRef a where
  type DeRef a :: * -> *

  mapDeRef :: (Applicative f)
```

```
          => (forall b .
                ( MuRef b
                , Typeable b
                , DeRef a ~ DeRef b
                ) => b -> f u)
          -> a
          -> f (DeRef a u)
```

`mapDeRef` has a rank-2 polymorphic functional argument for processing sub-nodes, when walking over a node of type `a`. This functional argument requires that

- The sub-node be a member of the class `MuRef`;
- The sub-node be `Typeable`, so that we can use `Dynamic` internally;
- Finally, the graph *representation* of the `a` node and the graph *representation* of the `b` node are the same type.

We can use this version of `MuRef` to capture sharing at different types. For example, consider the structure

```
let xs = [1..3]
    ys = 0 : xs
in cycle [xs,ys,tail ys]
```

There are three types inside this structure, `[[Int]]`, `[Int]`, and `Int`. This means we need *two* instances, one for lists with element types that can be reified, and one for `Int`, and a common data-type to represent the graph nodes.

```
data Node u = Cons u u
            | Nil
            | Int Int

instance ( Typeable a
         , MuRef a
         , DeRef [a] ~ DeRef a) => MuRef [a] where
  type DeRef [a] = Node

  mapDeRef f (x:xs) = Cons <$> f x <*> f xs
  mapDeRef f []     = pure Nil

instance MuRef Int where
  type DeRef Int = Node

  mapDeRef f n = pure $ Int n
```

The `Node` type is our reified graph node structure, with three possible constructors, `Cons` and `Nil` for lists (of type `[Int]` or type `[[Int]]`), and `Int` which represents an Int.

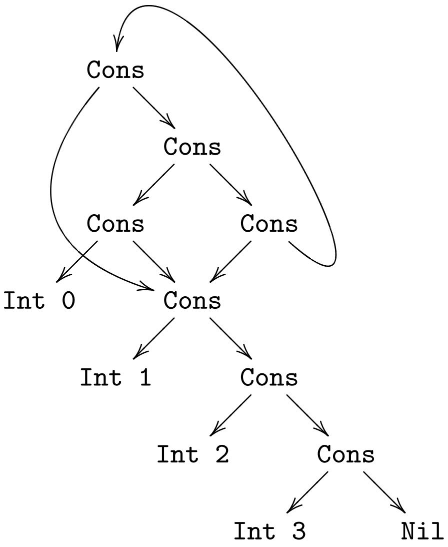


Figure 1. Sharing within structures of different types

Reifying the example above now succeeds, giving

```
> reifyGraph (let xs = [1..3]
>                 ys = 0 : xs
>             in cycle [xs,ys,tail ys])
Graph [ (1,Cons 2 9)
      , (9,Cons 10 12)
```

```
       , (12,Cons 2 1)
       , (10,Cons 11 2)
       , (11,Int 0)
       , (2,Cons 3 4)
       , (4,Cons 5 6)
       , (6,Cons 7 8)
       , (8,Nil)
       , (7,Int 3)
       , (5,Int 2)
       , (3,Int 1)
       ]
       1
```

Figure 1 renders this graph, showing we have successfully captured the sharing at multiple levels.

10. Observing Functions

Given we can observe structures with distinct node types, can we use the same machinery to observe functions? It turns out we can!

A traditional way of observing functions is to apply a function to a dummy argument, and observe where this dummy argument occurs inside the result expression. At first, it seems that an exception can be used for this, but there is a critical shortcoming. It is impossible to distinguish between the use of a dummy argument in a sound way and examining the argument. For example

```
 \ x -> (1,[1..x])
```

gives the same result as

```
 \ x -> (1,x)
```

when x is bound to an exception-raising thunk.

We can instead use the type class system, again, to help us.

```
 class NewVar a where
   mkVar :: Dynamic -> a
```

Now, we can write a function that takes a function and returns the function argument and result as a tuple.

```
 capture :: (Typeable a, Typeable b, NewVar a)
         => (a -> b) -> (a,b)
 capture f = (a,f a)
   where a = mkVar (toDyn f)
```

We use the `Dynamic` as a unique label (that does not admit equality) being passed to `mkVar`. To illustrate this class being used, consider a small DSL for arithmetic, modeled on the ideas for capturing arithmetic expressions used in Elliott et al. (2003).

```
 data Exp = ExpVar Dynamic
          | ExpLit Int
          | ExpAdd Exp Exp
          | ...
   deriving (Typeable, ...)
```

```
instance NewVar Exp where
  mkVar = ExpVar

instance Num Exp where
  (+) = ExpAdd
  ...
  fromInteger n = ExpLit (fromInteger n)
```

With these definitions, we can capture our function

```
> capture (\ x -> x + 1 :: Exp)
(ExpVar ..., ExpAdd (ExpVar ...) (ExpLit 1))
```

The idea of passing in a explicit `ExpVar` constructor is an old one, and the data-structure used in Elliott et al. (2003) also included a `ExpVar`, but required a threading of a unique `String` at the point a function was being examined. With observable sharing, we can observe the sharing that is present inside the `capture` function, and reify our function without needing these unique names.

`capture` gives a simple mechanism for looking at functions, but not functions inside data-structures we are observing for sharing. We want to add the `capture` mechanism to our multi-type reification, using a `Lambda` constructor in the graph node data-type.

```
instance ( MuRef a, Typeable a, NewVar a,
           MuRef b, Typeable b,
           DeRef a ~ DeRef (a -> b),
           DeRef b ~ DeRef (a -> b) )
     => MuRef (a -> b) where
```

```
   type DeRef (a -> b) = Node
  mapDeRef f fn = let v = mkVar $ toDyn fn
                  in Lambda <$> f v <*> f (fn v)
```

This is quite a mouthful! For functions of type `a -> b`, we need `a` to admit `MuRef` (have observable sharing), `Typeable` (because we are working in the multi-type observation version), and `NewVar` (because we want to observe the function). We need `b` to admit `MuRef` and `Typeable`. We also need `a`, `b` and `a -> b` to all share a common graph data-type. When observing a graph with a function, we are actually observing the sharing created by the `let v = ...` inside the `mapDeRef` definition.

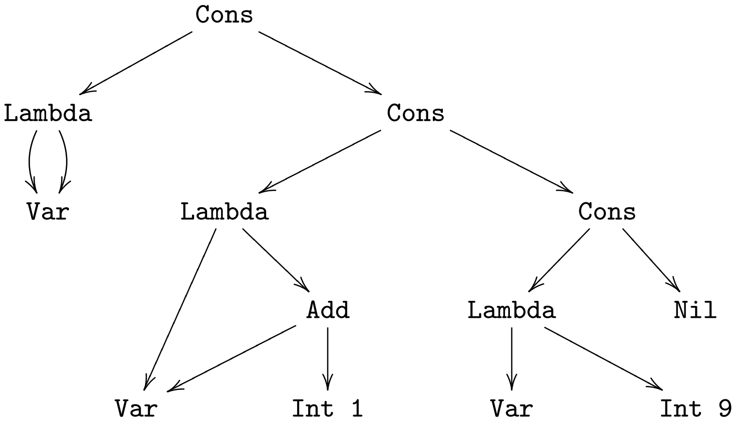


Figure 2. Sharing within structures and functions

We need to add our `MuRef instance` for `Exp`, so we can observe structures of the type `Exp`.

```
data Node u = ... | Lambda u u | Var | Add u u

instance MuRef Exp where
  type DeRef Exp = Node

  mapDeRef f (ExpVar _)   = pure Var
  mapDeRef f (ExpLit i)   = pure $ Int i
  mapDeRef f (ExpAdd x y) = Add <$> f x <*> f y
```

Finally, we can observe functions in the wild!

```
> reifyGraph (let t = [ \ x -> x :: Exp
>                     , \ x -> x + 1
>                     , \ x -> head t 9 ]
>             in t)
Graph [ (1,Cons 2 4)
      , (4,Cons 5 9)
      , (9,Cons 10 13)
      , (13,Nil)
      , (10,Lambda 11 12)
      , (12,Int 9)
      , (11,Var)
```

```
        , (5,Lambda 6 7)
        , (7,Add 6 8)
        , (8,Int 1)
        , (6,Var)
        , (2,Lambda 3 3)
        , (3,Var)
        ]
        1
```

Figure 2 shows the connected graph that this reification produced. The left hand edge exiting `Lambda` is the argument, and the right hand edge is the expression.

In Elliott et al. (2003), an expression DSL like our example here was used to synthesize and manipulate infinite, continuous images. The DSL generated C code, allowing real time manipulation of image parameters. In Elliott (2004), a similar expression DSL was used to generate shader assembly rendering code plus C# GUI code. A crucial piece of technology needed to make both these implementations viable was a common sub-expression eliminator, to recover lost sharing. We recover the important common sub-expressions for the small cost of observing sharing from within an `IO` function.

11. Implementation of `reifyGraph`

In this section, we present our implementation of `reifyGraph`. The implementation is short, and we include it in the appendix.

We provide two implementations of `reifyGraph` in the hackage library `data-reify`. The first implementation of `reifyGraph` is a depth-first walk over a tree at single type, to discover structure, storing this in a list. A second implementation also performs a depth-first walk, but can observe sharing of a predetermined set of types, provided they map to a common node type in the final graph.

One surprise is that we can implement our flexible observable sharing functions in just a few lines of GHC Haskell. We use the `StableName` abstraction, as introduced in Peyton Jones et al. (1999), to provide our basic (typed) pointer equality, and the remainder of our implementation is straightforward Haskell programming.

Stable names are supplied in the library `System.Mem.StableName`, to allow pointer equality, provided the objects have been declared comparable inside an `IO` operation. The interface is small.

```
data StableName a
makeStableName :: a -> IO (StableName a)
hashStableName :: StableName a -> Int
instance Eq (StableName a)
```

If you are inside the IO monad, you can make a `StableName` from any object, and the type `StableName` admits `Eq` without looking at the original object. `StableNames` can be thought of as a pointer, and the `Eq` instance as pointer equality on these pointers. Finally, the `hashStableName` facilitates a lookup table containing `StableNames`, and is stable over garbage collection.

We use stable names to keep a list of already visited nodes. Our graph capture is the classical depth first search over the graph, and does not recurse over nodes that we have already visited. `reifyGraph` is implemented as follows.

- We initialize two tables, one that maps `StableNames` (at the same type) to `Uniques`, and a list that maps `Uniques` to edges in our final node type. In the first table, we use the `hashStableName` facility of `StableNames` to improve the lookup time.
- We then call a recursive graph walking function `findNodes` with the two tables stored inside `MVars`.
- We then return the second table, and the `Unique`

Inside `findNodes`, for a specific node, we

- Perform `seq` on this node, to make sure this node is evaluated.
- If we have seen this node before, we immediately return the `Unique` that is associated with this node.
- We then allocate a new `Unique`, and store it in our first `MVar` table, using the `StableName` of this node as the key.
- We use `mapDeRef` to recurse over the children of this node.
- This returns a new node of type "`DeRef s Unique`", where `s` is the type we are recursing over, and `DeRef` is our type function.
- We store the pair of the allocated unique and the value returned by `mapDeRef` in a list. This list will become our graph.
- We then return the `Unique` associated with this node.

It should be noted that the act of extracting the graph performs like a deep `seq`, being hyperstrict on the structure under consideration.

The `Dynamic` version of `reifyGraph` is similar to the standard `reifyGraph`. The first table contains `Dynamics`, not `StableNames`, and when considering a node for equality, the `fromDynamic` is called at the current node type. If the node is of the same type as the object inside the `Dynamic`, then the `StableName` equality is used to determine point equality. If the node is of a different type (`fromDynamic` returns `Nothing`), then the pointer equality fails by definition.

One shortcoming with the `Dynamic` implementation is the obscure error messages. If an `instance` is missing, this terse message is generated.

```
Top level:
    Couldn't match expected type 'Node'
    against inferred type 'DeRef t'
```

This is stating that the common type of the final `Graph` was expected, and for *some* structure was not found, but does not state which one was not found. It would be nice if we could somehow parameterize the error messages or augment them with a secondary message.

12. Reflections on Observable Sharing

In this section, we consider both the correctness and consequences of observable sharing. The correctness of `reifyGraph` depends on the correctness of `StableNames`. Furthermore, observing the heap, even from within an `IO` function, has consequences for the validity of equational reasoning and the laws that can be assumed.

In the `System.Mem.StableName` library, stable names are defined as providing "a way of performing fast [...], not-quite-exact comparison between objects." Specifically, the only requirement on stable names is that if two stable names are equal, then "[both] were created by calls to `makeStableName` on the same object." This is a property that could be trivially satisfied by simply defining equality over stable names as `False`!

The intent of stable names is to implement the behavior of pointer equality on heap representations, while allowing the heap to use efficient encodings. In reality, the interface does detect sharing, with the advertised caveat that an object before and after evaluation may not generate stable names that are equal. In our implementation, we use the `seq` function to force evaluation of each graph node under observation, just before generating stable names, and this has been found to reliably detect the sharing we expect. It is unsettling, however, that we do not (yet) have a semantics of when we can and can not depend on stable names to observe sharing.

An alternative to using stable names would be to directly examine the heap representations. Vacuum (Morrow) is a Haskell library for extracting heap representations, which gives a literal view of the heap world, and has been successfully used to both capture and visualize sharing inside Haskell structures. Vacuum has the ability to generate `dot` graphs for observation and does not require that a graph be evaluated before being observed.

Vacuum and `reifyGraph` have complementary roles. Vacuum allows the user to see a snapshot of the real-time heap without necessarily changing it, while `reifyGraph` provides a higher level interface, by forcing evaluation on a specific structure, and then observing sharing on the same structure. Furthermore `reifyGraph` does not require the user to understand low-level representations to observe sharing. It would certainly be possible to build `reifyGraph` on top of Vacuum.

Assuming a reliable observation of sharing inside `reifyGraph`, what are the consequences to the Haskell programmer? Claessen and Sands (1999) argue that little is lost in the presence of observable sharing in a *call-by-name* lazy functional language, and also observe that all Haskell implementations use a call-by-name evaluation strategy, even though the Haskell report (Peyton Jones 2003) does not require this. In Haskell let-β, a variant of β-reduction, holds.

$$\texttt{let}\,\{x = M\}\,\texttt{in}\,N \quad = \quad N[^{M}/_{x}] \quad (x \notin M) \tag{1}$$

Over *structural* values, this equality is used with caution inside Haskell compilers, in either direction. To duplicate the construction of a structure is duplicating work, and can change the time complexity of a program. To common up construction (using (1) from right to left) is also problematic because this can be detrimental to the space complexity of a program.

It is easy in Haskell to lose sharing, even without using (1). Consider one of the `map` laws.

$$\texttt{map}\,id\,M \quad = \quad M \tag{2}$$

Any structure that the spine of 'M' has is lost in '`map` id M'.

Interestingly, this loss of sharing in `map` is not mandated, and a version of `map` using memoization could preserve the sharing. This is never done because we can not depend on – or observe – sharing.

One place where GHC introduces unexpected sharing is when generating overloaded literals. In Kansas Lava, the term `9 + 9` unexpectedly shares the same node for the value `9`.

```
> reifyGraph (9 + 9)
Graph [ (1,Entity + [2,2])
      , (2,Entity fromInteger [3])
      , (3,Lit 9)
      ]
      1
```

Literal values are like enumerated constructors, and any user of `reifyGraph` must allow for the possibility of such literals being shared.

What does all this mean? We can have unexpected sharing of constants, as well as lose sharing by applying what we considered to be equality holding transformations.

The basic guidelines for using `reifyData` are

- Observe only structures built syntactically. Combinators in our DSLs are lazy in their (observed) arguments, and we do not deconstruct the observed structure before `reifyData`.
- Assume constants and enumerated constructors may be shared, even if syntactically they are not the same expression.

There is a final guideline when using observable sharing, which is to allow a DSL to have some type of (perhaps informal) let-β rule. In the same manner as rule (1) in Haskell should only change how *fast* some things run and not the final outcome, interpreters using observable sharing should endeavor to use sharing to influence performance, not outcome. For example, in Lava, undetected acyclic sharing in a graph would result in extra circuitry and the same results being computed at a much greater cost. Even for undetected loops in well-formed Lava circuits, it is possible to generate circuits that work for a preset finite number of cycles.

If this guideline is followed literally, applying (1) and other equational reasoning techniques to DSLs that use observable sharing is now a familiar task for a functional programer, because applying equational reasoning changes performance, not the final result. A sensible let-β rule might not be possible for all DSLs, but it provides a useful rule of thumb to influence the design.

13. Performance Measurements

We performed some basic performance measurements on our `reifyGraph` function. We ran a small number of tests observing the sharing in a binary tree, both with and without sharing, on both the original and `Dynamic reifyGraph`. Each extra level on the graph introduces double the number of nodes.

Tree	**Original**		**Dynamic**	
Depth	**Sharing**	**No Sharing**	**Sharing**	**No Sharing**

16	0.100s	0.154s	0.147s	0.207s
17	0.237s	0.416s	0.343s	0.519s
18	0.718s	1.704s	0.909s	2.259s
19	2.471s	7.196s	2.845s	8.244s
20	11.140s	25.707s	13.377s	32.443s

While `reifyGraph` is not linear, we can handle 2^{20} (around a million) nodes in a few seconds.

14. Conclusions and Further Work

We have introduced an `IO` based solution to observable sharing that uses type functions to provide *type-safe observable sharing*. The use of `IO` is not a hinderance in practice, because the occasions we want to observe sharing are typically the same occasions as when we want to export a net-list like structure to other tools.

Our hope is that the simplicity of the interface and the familiarity with the ramifications of using an `IO` function will lead to `reifyGraph` being used for observable sharing in deep DSLs.

We need a semantics for `reifyGraph`. This of course will involve giving at least a partial semantics to `IO`, for the way it is being used. One possibility is to model the `StableName` equality as a non-deterministic choice, where IO provides a `True/False` oracle. This would mean that `reifyGraph` would actually return an infinite tree of possible graphs, one for each possible permutation of answers to the pointer equality. Another approach we are considering is to extend Natural Semantics (Launchbury 1993) for a core functional language with a reify primitive, and compare it with the semantics for `Ref`-based observable sharing (Claessen and Sands 1999).

Acknowledgments

I would like to thank all the members of CDSL at ITTC for the creative research environment, many interesting discussions, and detailed feedback. I would also like to thank Conal Elliott, Kevin Matlage, Don Stewart, and the anonymous reviewers for their many useful comments and suggestions.

References

Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 69–79. ACM Press, 2004. ISBN 1-58113-850-4.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, April 2001.

Koen Claessen and David Sands. Observable sharing for functional circuit description. In P. S. Thiagarajan and Roland H. C. Yap, editors, *Advances in Computing Science - ASIAN'99*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999. ISBN 3-540-66856-X.

Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003.

J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

Levent Erkök and John Launchbury. A recursive do for Haskell. In *Haskell Workshop'02, Pittsburgh, Pennsylvania, USA*, pages 29–37. ACM Press, October 2002.

Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5.

Mark P. Jones and Iavor S. Diatchki. Language and program design for functional dependencies. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 87–98, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: http://doi.acm.org/10.1145/1411286.1411298.

John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999.

John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *ICCL '98: International Conference on Computer Languages*, pages 90–101, 1998.

Conor McBride and Ross Patterson. Applicative programing with effects.

Journal of Functional Programming, 16(6), 2006.

Matt Morrow. Vacuum. `hackage.haskell.org/package/vacuum`.

John O'Donnell. Overview of Hydra: a concurrent language for synchronous digital circuit design. In *Parallel and Distributed Processing Symposium*, pages 234–242, 2002.

John O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 178–194. Springer-Verlag, 1992.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. In *Proceedings of the 11th International Workshop on the Implementation of Functional Languages*, LNCS, The Netherlands, September 1999. Springer-Verlag.

Richard Sharp. Functional design using behavioural and structural components. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 324–341, London, UK, 2002. Springer-Verlag. ISBN 3-540-00116-6.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Satnam Singh and Phil James-Roxby. Lava and jbits: From hdl to bitstream in seconds. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 91–100, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-2667-5.

A. Implementation

```
{-# LANGUAGE FlexibleContexts, UndecidableInstances #-}
module Data.Reify.Graph ( Graph(..), Unique ) where

import Data.Unique

type Unique = Int
data Graph e = Graph [(Unique,e Unique)] Unique
```

```
{-# LANGUAGE UndecidableInstances, TypeFamilies #-}
module Data.Reify
   ( MuRef(..), module Data.Reify.Graph, reifyGraph
   ) where

import Control.Concurrent.MVar
import Control.Monad
import System.Mem.StableName
import Data.IntMap as M
import Control.Applicative
import Data.Reify.Graph

class MuRef a where
  type DeRef a :: * -> *
  mapDeRef :: (Applicative m)
           => (a -> m u) -> a -> m (DeRef a u)

reifyGraph :: (MuRef s) => s -> IO (Graph (DeRef s))
reifyGraph m = do rt1 <- newMVar M.empty
```

```
                    rt2 <- newMVar []
                    uVar <- newMVar 0
                    root <- findNodes rt1 rt2 uVar m
                    pairs <- readMVar rt2
                    return (Graph pairs root)

findNodes :: (MuRef s)
          => MVar (IntMap [(StableName s,Int)])
          -> MVar [(Int,DeRef s Int)]
          -> MVar Int
          -> s
          -> IO Int
findNodes rt1 rt2 uVar j | j `seq` True = do
        st <- makeStableName j
        tab <- takeMVar rt1
        case mylookup st tab of
          Just var -> do putMVar rt1 tab
                         return $ var
          Nothing -> do var <- newUnique uVar
                        putMVar rt1 $ M.insertWith (++)
                            (hashStableName st)
                            [(st,var)]
                            tab
                        res <- mapDeRef
                                (findNodes rt1 rt2 uVar)
                                j
                        tab' <- takeMVar rt2
                        putMVar rt2 $ (var,res) : tab'
                        return var
   where
        mylookup h tab =
           case M.lookup (hashStableName h) tab of
             Just tab2 -> Prelude.lookup h tab2
```

```
              Nothing ->  Nothing

newUnique :: MVar Int -> IO Int
newUnique var = do
  v <- takeMVar var
  let v' = succ v
  putMVar var v'
  return v'
```

```
{-# LANGUAGE UndecidableInstances, TypeFamilies,
     RankNTypes, ExistentialQuantification,
     DeriveDataTypeable, RelaxedPolyRec,
     FlexibleContexts  #-}

module Data.Dynamic.Reify
  ( MuRef(..), module Data.Reify.Graph, reifyGraph
  ) where

class MuRef a where
  type DeRef a :: * -> *
  mapDeRef :: (Applicative f) =>
              (forall b . (MuRef b, Typeable b,
                             DeRef a ~ DeRef b)
                                       => b -> f u)
                          -> a
                          -> f (DeRef a u)

reifyGraph :: (MuRef s, Typeable s)
           => s -> IO (Graph (DeRef s))
reifyGraph m = do rt1 <- newMVar M.empty
                  rt2 <- newMVar []
                  uVar <- newMVar 0
```

```
                    root <- findNodes rt1 rt2 uVar m
                    pairs <- readMVar rt2
                    return (Graph pairs root)

findNodes :: (MuRef s, Typeable s)
          => MVar (IntMap [(Dynamic,Int)])
          -> MVar [(Int,DeRef s Int)]
          -> MVar Int
          -> s
          -> IO Int
findNodes rt1 rt2 uVar j | j `seq` True = do
        st <- makeStableName j
        tab <- takeMVar rt1
        case mylookup st tab of
          Just var -> do putMVar rt1 tab
                         return $ var
          Nothing -> do var <- newUnique uVar
                        putMVar rt1 $ M.insertWith (++)
                            (hashStableName st)
                            [(toDyn st,var)]
                            tab
                       res <- mapDeRef
                                 (findNodes rt1 rt2 uVar)
                                 j
                       tab' <- takeMVar rt2
                       putMVar rt2 $ (var,res) : tab'
                       return var

mylookup :: (Typeable a)
         => StableName a
         -> IntMap [(Dynamic,Int)]
         -> Maybe Int
mylookup h tab =
           case M.lookup (hashStableName h) tab of
             Just tab2 -> Prelude.lookup (Just h)
                             [ (fromDynamic c,u)
                             | (c,u) <- tab2 ]
             Nothing ->  Nothing

newUnique :: MVar Int -> IO Int
newUnique var = do
  v <- takeMVar var
  let v' = succ v
  putMVar var v'
  return v'
```