Type Directed Observable Sharing

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

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 so-called 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 tenant of pure functional programming: 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, including a hardware description DSL. The addition makes nominal impact on the abstract language syntax tree; the tree itself remains a purely functional value, and the shape of this tree guides the structure of 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 compare with pure functions the same abstract syntaxes sans observable sharing.

2. Observable Sharing and Domain Specific Languages
Consider a simple description of a bit-level parity checker.

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

xor is a function which take two arguments of the abstract type Bit, performs 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 providing an interface to a µLava. These abstract primitives allows 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.

[Copyright notice will appear here once 'preprint' option is removed.]
acts as a supporting literal interpreter of the Bit DSL, but has a different implementation. An interpreter function
The run Haskell syntax tree.
ply Haskell data constructors, and a circuit description becomes a native
deep a DSL that allows extraction, we can give our primitives an alter-
cific circuit could not be directly extracted. In order to construct
its primitives, which could be simulated, but the meaning of a spe-
Hawk used a similar shallow embedding to provide semantics for
its primitives, which could be simulated, but the meaning of a spe-
cific circuit could not be directly extracted. In order to construct
a DSL that allows extraction, we can give our primitives an alter-

--- 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)

--- 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 -> M Bit
interp (Xor b1 b2) = zipWith (/=) (interp b1) (interp b2)
interp (Delay b) = False : b

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. Or circuit is a function, Bit -> Bit,
so we provided the argument (Name "x"), where "x" is unique to
this circuit, giving us a Bit, with the Name 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 (Name "x")
Xor (Delay (Xor (Delay (Xor (Delay (Xor (...

This looping structure can be used for interpretation, but not for fur-
ther 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 prob-
lem, 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. Unfortunately, they are cumbersome in
  practice, forcing a specific style of DSL idiom for all loops.
  This is the direct analog of DSL recursion in Haskell

- Explicit Labels can be used to allow later recovery of a graph
  structure, as proposed by O’Donnell [1992]. This means pass-
  ing 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 label implicitly, or capture a graph structure as a net-list
directly. This is the solution used in the early Lava implementa-
tions [Bjesse et al. 1998], and continued in Xilinx Lava [Singh
and James-Roxby 2001]. Using monads directly impact the type
of a circuit, and our parity would now have type

interp :: Bit -> M Bit

Tying the knot of the back edges can no longer be per-
formed using the Haskell where clause, but instead have to
used the non-standard and complex recursive-do mechanism
Erkok:2002:A-recursive-do-

References can be provided as a non-conservative exten-
s (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 con-
tains a Ref at every node.

In this paper, we advocate another approach to the problem of
observable sharing, namely a 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 com-
  ponent in a small number of lines of Haskell.

- 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 num-
  ber 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.

- Finally, our solution to observable sharing may be more palat-
  able to the community than the Ref type, given we accept IO
  functions routinely.
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 tenant of functional programming. How do we contain -- and reason about -- referential opaqueness in Haskell? In this section, we introduce our reify function, and sidestep this effect, 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.

```haskell
data BitGraph = BitGraph [(Unique, BitNode Unique)]

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

We parameterize Node over the Unique graph “edges”, to facilitate generic processing of nodes later, but this parameterization is not critical to our ideas; it is a design choice.

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

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

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 Node, we can generically change the types of our nodes labels.

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

```haskell
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 be 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 true for 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. In the above example Bit contains a tree of nodes of type S, and Graph contains a graph of nodes of type S. What is their relationship? For this example, the generic reifyGraph needs to

- Be able to look inside the structure Bit, to see a shared node representation.
- Be able to traverse the datatype S, recursively looking for sharing.
- Build a Graph, with nodes of type S.

We can incorporate these ideas, and present our generalized graph reification function, reifyGraph.

```haskell
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, using a common structure.

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

```haskell
data Graph e = Graph [(Unique, e Unique)]
```

We parameterize Node over the Unique graph “edges”, to facilitate generic processing of nodes later, but this parameterization is not critical to our ideas; it is a design choice.

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

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

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 Node, we can generically change the types of our nodes labels.

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

```haskell
reifyGraph :: 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 be 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 true for all usage models, having IO does not appear to be a handicap to this function.

```haskell
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.

```haskell
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

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

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

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

The use of the type function `DeRef` to find the BitNode data-type was critical to tying the input tree to type node representation type, though functional dependencies [Jones and Diatchki 2008] could also have been used here.

The complete `MuRef` class definition has the definition.

```haskell```
class MuRef a where
  mapDeRef :: (Applicative f) => a -> f (DeRef a e)  
  where
    mapDeRef f (Var nm) = pure $ Var nm
    mapDeRef f (Input bs) = pure $ Input bs
    mapDeRef f (Delay b) = Delay <$> b
    mapDeRef f (Xor a b) = Xor <$> f a <*> f b

type DeRef a :: * -> *
```

`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 indexes, of type `e`, 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.

```haskell```
instance MuRef Bit where
  type DeRef Bit = BitGraph
  mapDeRef f (Xor a b) = Xor <$> f a <*> f b
  mapDeRef f (Delay b) = Delay <$> b
  mapDeRef f (Input bs) = pure $ Input bs
  mapDeRef f (Var nm) = pure $ Var nm
```

This is a complete definition of the necessary generics to provide `reifyGraph` with the ability to perform type-directed 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.

```haskell```
> reifyGraph $ parity (Name "x")
BitGraph [(1,GraphXor 2 3), (2,GraphDelay 1), (3,GraphInput "x")]
```

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 and semantics.

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 encoding state machine for a viterbi decoder.

![State Machine Diagram]

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.

```haskell```
data State = ZeroZero | ZeroOne | OneZero | OneOne

type Input = Bool

data State = ZeroZero | ZeroOne | OneZero | OneOne

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

step :: Input -> State -> (Output,State)

step i (State ts) = (output,st)

step :: Input -> State -> (Output,State)

step i (State ts) = (output,st)

where (output,st) = lookup i ts
```

Arguably more declarative would be to use the `binding` as the state unique identifier.

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

step :: Input -> State i o -> (Output,State i o)

step step i (State ts) = (output,st)

where (output,st) = lookup i ts
```

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

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

Submitted to the Haskell Symposium, 2009. This version has additional typo corrections.
Extracting the sharing, for example to allow the display in the graph viewing tool dot, is not possible in a purely functional setting. But extracting the sharing using our `reifyGraph` allows the deeper embedding to be gathered, and other tools can manipulate and optimize this graph.

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

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

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

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

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 original design 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 used `reifyGraph`.

newtype Signal a = Signal Wire 
newtype Wire = Wire (Block Wire) 
data Block s = Xor s s |
    And s s |
    Not s |
    Delay s |
    ...

xor2 :: Signal a -> Signal a -> Signal a 
xor2 (Signal a) (Signal b) = Signal $ Xor a b 
and2 :: Signal a -> Signal a -> Signal a 
and2 (Signal a) (Signal b) = Signal $ And a b 
not :: Signal a -> Signal a 
not (Signal a) = Signal $ Not a 
...

We also admit a purely functional variant; because our Signal type is a pure Haksell structure, we can simulate our circuits without resorting to any IO actions.

In both Kansas Lava and Chalmers Lava, phantom types (Leijen and Meijer 1999) 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.

```haskell
mux :: Signal Bool -> Signal a -> Signal a -> Signal a 
mux (Signal p) (Signal t) (Signal f) = Signal $ Mux p t f 
```

Even though we construct trees of type `Signal a`, we want to observe graphs of type `Wire`, because every `Signal` is a constructor wrapper round a tree of `Wire`. We share the same node data-type between our Haskell tree underneath `Signal`, and inside our reified graph. This data-type, `Block`, is parametrized over block 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:

```haskell
import qualified Data.Traversable as T
instance MuRef Wire where 
    type DeRef Wire = Block 
    mapDeRef f (Signal s) = T.mapM f s 
```

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

```haskell
instance T.Traversable Block where 
    traverse f (And2 e1 e2) = And <$> f e1 <*> f e2 
    traverse ... 

instance F.Foldable Block where 
    foldMap f (And2 e1 e2) = f e1 `mappend` f e2 
    foldMap ... 

instance Functor Block where 
    fmap f (And2 e1 e2) = And2 (f e1) (f e2) 
    fmap ... 
```

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.

```haskell
data Ref a = ...
instance Eq (Ref a) 
ref :: a -> Ref a 

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

\[
\begin{align*}
\text{data Ref a} &= \text{Ref a Unique} \\
\text{instance Eq (Ref a) where} \\
\text{(Ref _ u1) == (Ref _ u2)} &= u1 == u2 \\
\text{ref a} &= \text{unsafePerformIO} \ \{ \text{do} \ \\
&\quad u \gets \text{newUnique} \\
&\quad \text{return} \ \{ \text{Ref a u} \} \ \\
\text{deref (Ref a _)} &= a
\end{align*}
\]

with the usual caveats associated with the use of \( \text{unsafePerformIO} \).

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

\[
\begin{align*}
\text{newtype Signal s} &= \text{Signal Wire} \\
\text{newtype Wire} &= \text{Wire (Ref (Block Wire))} \\
\text{data Block s} &= \text{Xor s s} \\
&\quad | \ldots \\
\text{xor2 :: Signal a} &\rightarrow \text{Signal a} \\
\text{xor2 (Signal a) (Signal b)} &= \text{Signal} \ \{ \text{ref} \ \{ \text{Xor a b} \} \}
\end{align*}
\]

The differences between this definition and the Kansas Lava definition are

- The type \( \text{Wire} \) includes an extra \( \text{Ref} \) indirection.
- The DSL primitives include an extra \( \text{ref} \).

\( \text{Wire} \) in Chalmers Lava admits observable sharing directly, while Kansas Lava only admits observable sharing using \( \text{reifyGraph} \). The structure in Kansas Lava can be consumed by an alternative, purely functional simulation function, without the possibility of accidentally observing sharing. Furthermore, \( \text{reifyGraph} \) 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 \( \text{Mu} \) type (Jones 1995). For example, consider a list specified in this fashion.

\[
\begin{align*}
\text{newtype Mu a} &= \text{In (a (Mu a))} \\
\text{data List a b} &= \text{Cons a b | Nil} \\
\text{type MyList a} &= \text{Mu (List a)}
\end{align*}
\]

Now, we can write a list using \( \text{Cons}, \text{Nil}, \) and \( \text{In} \) for recursion. The list \([1,2,3]\) would be represented using the following expression.

\[
\text{In (Cons 1 (In (Cons 2 (In Cons 3 (In Nil))))})
\]

The generality of the recursion, captured by \( \text{Mu} \), allows a general instance of \( \text{Mu} \) for \( \text{MuRef} \). Indeed, this is why \( \text{MuRef} \) is called \( \text{MuRef} \).

\[
\begin{align*}
\text{instance (T.Traversable a) => MuRef (Mu a) where} \\
\text{type DeRef (Mu a) = a} \\
\text{mapDeRef} &= \text{T.traverse}
\end{align*}
\]

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

\[
\begin{align*}
\text{instance Functor (List a) where} \\
\text{fmap f Nil} &= \text{Nil} \\
\text{fmap f (Cons a b)} &= \text{Cons a (f b)}
\end{align*}
\]

\[
\begin{align*}
\text{instance F.Foldable (List a) where} \\
\text{foldMap f Nil} &= \text{mempty} \\
\text{foldMap f (Cons a b)} &= f b
\end{align*}
\]

\[
\begin{align*}
\text{instance T.Traversable (List a) where} \\
\text{traverse f (Cons a b)} &= \text{Cons <$> pure a <*> f b} \\
\text{traverse f Nil} &= \text{pure Nil}
\end{align*}
\]

Now a list, written using \( \text{Mu} \), can have its sharing observed.

\[
\begin{align*}
> \text{let xs} &= \text{In (Cons 1 (In (Cons 2 xs)))} \\
> \text{reifyGraph xs} \\
\text{Graph} &= \{ \text{(100,Cons 1 101)} \\
&\quad \text{, (101,Cons 2 100)} \} \\
&\quad 1
\end{align*}
\]

The type \( \text{List} \) was used both for expressing trees and graphs. We can reuse \( \text{List} \) and the instances of \( \text{List} \) to observe sharing in regular Haskell list.

\[
\begin{align*}
\text{instance MuRef [a] where} \\
\text{type DeRef [a] = List} \\
\text{mapDeRef \{ x:xs \} = Cons x <$> f xs} \\
\text{mapDeRef []} &= \text{pure Nil}
\end{align*}
\]

That is, regular Haskell lists are represented as a graph, using \( \text{List} \), and \( \text{Mu List} \) lists are also represented as a graph, using \( \text{List} \). Now we can capture spine-level sharing in our list.

\[
\begin{align*}
> \text{let xs} &= \text{1 : 2 : xs} \\
> \text{reifyGraph xs} \\
\text{Graph} &= \{ \text{(100,Cons 1 101)} \\
&\quad \text{, (101,Cons 2 100)} \} \\
&\quad 1
\end{align*}
\]

There is not way to observe built-in Haskell data structures using \( \text{Ref} \), which is an advantage of our \( \text{reify-based observable sharing} \). 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 is possible we observe the difference between

\[
\text{let x = X 1 in [x,x] and [X 1,X 1]}
\]
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 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 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 map to C, the same type. We can express this, using the new ~ notation to type equivalence. The type

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

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

In order to observe sharing on nodes of types that are both 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 ,
      DeRef a ~ DeRef b
    )
  => b -> d 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 [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 common data-type to represent the graph nodes.

```
data Node u = Cons u u
  | Nil
instance ( Typeable a
    , DeRef [a] ~ DeRef a
  ) => MuRef [a] where
  type DeRef [a] = Node
  mapDeRef f (x:xs) = liftA2 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.

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) ]
```

Figure renders this graph, showing we have successfully captured the sharing at two different 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
\[
x \rightarrow (1, [1..x])
\]
gives the same result as
\[
x \rightarrow (1, x)
\]
when x is bound to an exception-raising thunk.
We can instead use the type class system, again, to help us.

```haskell
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.

```haskell
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).

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

```haskell
instance NewVar Exp where
    mkVar = ExpVar
```

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

With these definitions, we can capture our function

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

The idea of passing in an 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 straightforward multi-type reification, given a Lambda node in our graph node data-type.

```haskell
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 `let v = ...` inside the `mapDeRef` definition.

We need to add our `MuRef` instance, so we can observe structures of the type `Exp`, add their paired type to `Node`, as well as `Lambda` as required by the instance of `MuRef` at `a -> b`.

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

```haskell
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!

```haskell
gerifyGraph (let t = \x -> x :: Exp > , \x -> head t 9 > in t)
```

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.][1], an expression DSL like our example here was used to capture a field of values on a two dimensional surface. The DSL generated C and Java code, allowing real time manipulation of some of the free variables of the observed function. A crucial piece of technology to make their implementation viable was a common sub-expression eliminator, or CSE. We get CSE for free here, 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.][999], 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.

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

Provided 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. The garbage collector does not taint these pointers, but there is no guarantee that the an object before and after evaluation will generate two `StableNames` that are equal. 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 not recursing 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`. This is to make our implementation thread-safe.
- 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 n Unique", where `n` 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 type.

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 (from `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 expecting, and for some structure, was not found, and does not state which one. It would be nice if we could somehow parameterize the error messages, or augment them with a secondary message.

12. Performance Measurements

We performed some basic performance measurements of our reifyGraph function. Table 1 contains a small number of test runs 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. While reifyGraph is not linear, we can handle $2^n$ (around a million) nodes in a few seconds.

<table>
<thead>
<tr>
<th>Tree Depth</th>
<th>Original No Sharing</th>
<th>Sharing</th>
<th>Dynamic No Sharing</th>
<th>Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.147</td>
<td>0.010</td>
<td>0.199</td>
<td>0.208</td>
</tr>
<tr>
<td>17</td>
<td>0.357</td>
<td>0.225</td>
<td>0.465</td>
<td>0.460</td>
</tr>
<tr>
<td>18</td>
<td>0.981</td>
<td>0.593</td>
<td>1.202</td>
<td>1.211</td>
</tr>
<tr>
<td>19</td>
<td>3.912</td>
<td>1.797</td>
<td>4.432</td>
<td>4.443</td>
</tr>
<tr>
<td>20</td>
<td>14.195</td>
<td>7.528</td>
<td>15.625</td>
<td>15.729</td>
</tr>
</tbody>
</table>

Table 1. Performance of reifyGraph

13. Conclusions and Further Work

We have introduced an IO based solution to observable sharing that uses type functions to provide type directed observable sharing. This is not a hindrance 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 TableName 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 [Launchbury1993] for a core functional language with a reify primitive, and compare it with the semantics for Ref-based observable sharing [Claessen and Sands1999].

References


A. Implementation

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

import Data.Unique

5

data Graph e = Graph [(Unique, e Unique)] Unique

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

5

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

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

20

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 []  
                    root <- findNodes rt1 rt2 m  
                    pairs <- readMVar rt2  
                    return (Graph pairs root)

findNodes :: (MuRef s)  
                  => MVar (IntMap [(StableName s, Unique)])  
                  -> MVar [(Unique, DeRef s Unique)]  
                  -> s  
                  -> IO Unique
findNodes rt1 rt2 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  
                  putMVar rt1 $  
                  M.insertWith (+)  
                  (hashStableName st)  
                  [(st, var)]  
                  tab  
    res <- mapDeRef (findNodes rt1 rt2) 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

5

{-# LANGUAGE UndecidableInstances, TypeFamilies,  
RankNTypes, ExistentialQuantification,  
DeriveDataTypeable, RelaxedPolyRec,  
FlexibleContexts #}  
module Data.Dynamic.Reify  
(MuRef(..), module Data.Reify.Graph, reifyGraph) where

5

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

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

20

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 []  
                    root <- findNodes rt1 rt2 m  
                    pairs <- readMVar rt2  
                    return (Graph pairs root)

findNodes :: (MuRef s, Typeable s)  
                  => MVar (IntMap [(Dynamic, Unique)])  
                  -> MVar [(Unique, DeRef s Unique)]  
                  -> s  
                  -> IO Unique
findNodes rt1 rt2 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  
                  putMVar rt1 $  
                  M.insertWith (+)  
                  (hashStableName st)  
                  [(fromDynamic st, var)]  
                  tab  
    res <- mapDeRef (findNodes rt1 rt2) 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

mylookup h tab =  
  case M.lookup (hashStableName h) tab of  
  Just tab2 -> Prelude.lookup (Just h)  
  [(fromDynamic c, u)]  
  [(c, u) <- tab2]  
  Nothing -> Nothing

Submitted to the Haskell Symposium, 2009. This version has additional typo corrections.

2009/5/9