Type-Safe Observable Sharing in Haskell

Andy Gill

The University of Kansas

September 3, 2009
An Embedded Domain Specific Language is simply a style of (Haskell) library.

- You need to know Haskell!
- User-code written in the DSL are centered round a specific type (or types).
- We want to speed up execution, or cross-compile to a different execution platform.
Deep Embedding

-- Deep Embedding of Lava light

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

xor :: Bit -> Bit -> Bit
xor = Xor

delay :: Bit -> Bit
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
Executing a Deep Embedding

```haskell
xor :: Bit -> Bit -> Bit
delay :: Bit -> Bit
run :: (Bit -> Bit) -> [Bool] -> [Bool]

-- Parity specification
parity :: Bit -> Bit
parity input = output
where output = delay output 'xor' input

> take 10 $ run parity (cycle [True,False,True])
[True,True,False,True,True,False,True,True,False,True]
```

Question: Can we capture the structure of parity?
Deep Embedding and Var

-- New, deep embedding
data Bit = Xor Bit Bit | Delay Bit | Input [Bool]
  | Var String
  deriving Show

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

> let not a = xor a (Var "high") in not (Var "x")
Xor (Var "x") (Var "high")
> parity (Var "x")
Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (...

Andy Gill (The University of Kansas)
Where are we?

- We can give a deep embedding of Lava program.
- Our representation is a tree, not a graph.
- We have no observable sharing in our representation.
- Need to recover (or never lose) the sharing.
- We have no observable loops in our representation.
- Need to find a way to represent cycles.
- Not just a Lava issue
  - PAN’s compiler critically depended on a CSE pass for compiling arithmetical pixel descriptions.
A common way to handle structures like our target is using a netlist. A netlist is just a directed graph, or a list of recursive bindings.

```haskell
type Netlist = [(Name,Op)]

data Op = XorN Name Name
         | DelayN Name
         | InputN [Bool]
         | VarN String
```

Question: Can we share the data-structure between our deep embedding and netlist?
data Bit = Bit (Circuit Bit)

data BitCircuit c
  | Xor BitCircuit c BitCircuit c
  | Delay BitCircuit c
  | Input [Bool]
  | Var String

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

type CircuitGraph = Graph Circuit
How do we turn a Lava program into a Graph?

- Use a Monad – Xilinx Lava does this.
- Tag every binding with a unique tag – Hydra does this.
- Automate the introduction of unique tags.
  - Destroys referential transparency.
  - Chalmers Lava does this.
  - Not a big problem in practice.
- Use a function that can see certain classes of sharing.
  - Like a reflective parser.
  - Kansas Lava does this.
  - Like Chalmers Lava, can be unsafe.
data Bit = Bit (Ref (Circuit Bit))

data Circuit c
| Xor c c
| Delay c
| Input [Bool]
| Var String

\(\cong\) :: Ref a -> Ref a -> Bool

\textbf{unsafe}NnewRef a :: a -> Ref a \quad -- \textbf{unsafe}

deRef :: Ref a -> a
data Bit = Bit (Circuit Bit)

data Circuit c
  | Xor c c
  | Delay c
  | Input [Bool]
  | Var String

reifyStructure :: Bit -> IO (Graph Circuit)

data Graph e = Graph [(Unique,e Unique)] Unique
reifyGraph :: MuRef s => s -> IO (Graph (DeRef s))
data Graph e = Graph [(Unique, e Unique)] Unique

DeRef is a type function

-- ‘‘conceptual’’ code
type DeRef Bit = Circuit
class MuRef a where
  type DeRef a :: * -> *
  mapDeRef :: ... -> a -> m (DeRef a u)

reifyGraph :: MuRef s => s -> IO (Graph (DeRef s))
data Graph e = Graph [(Unique, e Unique)] Unique

DeRef has kind * -> * -> *.
class MuRef

    type DeRef a :: * -> *

    mapDeRef :: (Applicative m)
    => (a -> m u)
    -> a
    -> m (DeRef a u)

reifyGraph :: MuRef s => s -> IO (Graph (DeRef s))
data Graph e = Graph [(Unique, e Unique)] Unique
instance MuRef Bit where
  type DeRef Bit = Circuit
  mapDeRef f (Xor a b) = liftA2 Xor (f a) (f b)
  mapDeRef f (Delay b) = liftA Delay (f b)
  mapDeRef f (Input bs) = pure $ Input bs
  mapDeRef f (Var nm) = pure $ Var nm

reifyGraph :: MuRef s => s -> IO (Graph (DeRef s))
data Graph e = Graph [(Unique, e Unique)] Unique

data Circuit c | Xor c c | Delay c | Input [Bool] | Var String
data Bit = Bit (Circuit Bit)
Example of capturing parity

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

> parity (Var "x")
Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (Delay (Xor (... 
> reifyGraph (parity (Bit (Var "x")))
Graph [(1,Xor 2 3),(3,Var "x"),(2,Delay 1)] 1
```
newtype Mu a = In (a (Mu a))

data List a b = Cons a b | Nil

type MyList a = Mu (List a)

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

traverse :: (Traversable t, Applicative f)
  => (a -> f b) -> t a -> f (t b)

instance Traversable (List a) where
  traverse f (Cons a b) = liftA (Cons a) (f b)
  traverse f Nil = pure Nil
Observing a Mu List

> let xs = In (Cons 1 (In (Cons 2 xs)))
> reifyGraph xs

Graph [ (100, Cons 1 101), (101, Cons 2 100) ]

100
instance MuRef [a] where
  type DeRef [a] = List
  mapDeRef (x:xs) = liftA (Cons x) (f xs)
  mapDeRef [] = pure Nil

> let xs = 1 : 2 : xs
> reifyGraph xs
Graph [ (100,Cons 1 101)
  , (101,Cons 2 100)
  ]
100
How reify works: Stable Names

data StableName a = ...
makeStableName :: a -> IO (StableName a)
instance Eq (StableName a)
hashStableName :: StableName a -> Int
instance MuRef [a] where
  type DeRef [a] = List
  mapDeRef (x:xs) = liftA (Cons x) (f xs)
  mapDeRef [] = pure Nil

let x = X 1
in [x,x]  
[X 1,X 1]
Dynamic variant of \texttt{reifyGraph}

```haskell
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))
```

Andy Gill (The University of Kansas)
Type-Safe Observable Sharing in Haskell  
September 3, 2009 22 / 27
let xs = [1..3]
ys = 0 : xs
in cycle [xs,ys,tail ys]
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) = liftA2 Cons (f x) (f xs)
  mapDeRef f []     = pure Nil

instance MuRef Int where
  type DeRef Int = Node

  mapDeRef f n = pure $ Int n
let xs = [1..3]
ys = 0 : xs
in cycle [xs, ys, tail ys]
let t = [λ x → x :: Exp, λ x → x + 1, λ x → head t 9 ] in t
Questions?