Practical Observable Sharing

Alexander Konovalov

June 2019

Introduction

Consider this simple DSL, representing a context free Grammar.

```
data CFG = Str String | Seq [CFG] | Alt [CFG]

digit  = Alt [Str "0", Str "1"]
number = Alt [Seq [], Seq [digit, number]]
```

$$D = 0|1$$
$$N = \varepsilon|D\ N$$

We can easily write a “parser” for this DSL:

```
parse :: CFG -> String -> Maybe String
parse (Str p) s        = stripPrefix p s
parse (Alt []) s       = Nothing
parse (Alt (h : t)) s = parse h s <|> parse (Alt t) s
parse (Seq []) ""      = Just ""
parse (Seq []) _       = Nothing
parse (Seq (h : t)) s = do
  r <- parse h s
  parse (Seq t) r
```

Now consider the problem of producing a corresponding regular expression to this grammar, or dumping it out in EBNF notation.

$$D = 0|1$$
$$N = \varepsilon|DN$$

$$N = (0|1)*$$

We can't! Because of referential transparency we can not detect loops.

Referential transparency ensures we can not distinguish between

$$N = \varepsilon | DN$$

and

$$\begin{aligned} N &= \varepsilon | D(\varepsilon | DN) \\ &= \varepsilon | D | DDN \end{aligned}$$

In fact, as long as we stick to referential transparency we will be observing essentially

$$N = \varepsilon | D | DD | DDD | DDDD | \ldots$$

(in reality there will be a lot more nesting due the alternation between Seq and Alt constructors)

Now consider a different DSL,

```
data Signal = Latch Signal
            | Xor Signal Signal
            | And Signal Signal
            | Inv Signal

-- 1010101010...
clock = Inv (Latch clock)
-- 001000100010001...
halfClock = let wire0 = Xor clock (Latch wire0)
                wire1 = And clock (Latch wire0)
            in wire1
```

Once again, we can simulate this circuit

```
simulate :: Signal -> [Bool]
simulate (Latch s) = False : (simulate s)
simulate (Xor a b) = zipWith (/=) (simulate a)
                                  (simulate b)
simulate (And a b) = zipWith (&&) (simulate a)
                                  (simulate b)
simulate (Inv a)   = map not (simulate a)
```

but can we dump it out as a circuit diagram, or produce a gate configuration for an FPGA? Can we find common subcircuits?

Finally, suppose we want to search for a subgraph of a certain shape in a graph,

```
data Graph = Map Int [Int]
find :: Graph -> Graph -> Bool
```

wouldn't it be nice to be able to express a triangle subgraph as

```
graph :: Graph
graph = magic a where
    a = [b]
    b = [c]
    c = [a]
```

In all of these cases, we have to violate referential transparency in some way to achieve our goals. How can we do that **tastefully**?

Outlawing cycles

We could outlaw cycles and use explicit

```
data CFG = ... | Fix (CFG -> CFG) | Magic Integer

number = Fix (\x -> Alt [Seq [], Seq [digit, x]])
```

data constructors. Horrifying for many different reasons.

Explicit labels

We could give all nodes an explicit label

```
number = Named 0 (Alt [Seq [], Seq [digit, number]])
```

Doesn't compose. Bug-prone.

Recursive do

We could use MonadFix to tie the knot in a monadic computation that produces the graph.

```
wire1 = do
    rec
        c <- clock
        lw0 <- latch wire0
        wire0 <- xor c lw0
        wire1 <- and c lw0
    return wire1
```

It works, but the syntax is more verbose and the semantics are tricky and rely *crucially* on laziness.

Unguarded Reference Equality

The most straightforward way would be to somehow allow comparing two values by reference,

```
refEq :: a -> a -> Bool
foo = let a = 10
       in refEq a a
```

This breaks referential transparency.

Guarded Reference Equality (1)

We could try to guard reference comparison using IO,

```
refEq :: a -> a -> IO Bool
refEq = do
  a <- makeStableName x
  b <- makeStableName y
  return (eqStableName a b)
```

Works, but not entirely satisfying. Can't use standard data structures, even with a newtype can't define Eq (Ref a).

Guarded Reference Equality

Instead of trying to guard reference comparison, we can guard "getting a reference":

```
newtype Name a = Name (StableName a)
instance Eq (Name a)
instance Hashable (Name a)

name :: a -> IO (Name a)
name = Name <$> makeStableName
```

Can use standard data structures, doesn't break referential transparency.

Stable Names

- Stable Names is a GHC feature. Think of them as comparable weak references, with a constructor guarded by IO.
- Can't make a valid Ord instance, but can easily make Hashable and Eq.
- Can implement in Scala, PureScript, Haskell, with the same exact interface.

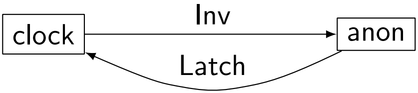
- So far we have talked only about how to observe reference equality.
- But we are interested in the actual practical solutions to our problems. Can't be bothered to implement graph traversal manually every time.

Graph representation

Recall our circuit DSL,

```
data Signal = Latch Signal
            | Xor Signal Signal
            | And Signal Signal
            | Inv Signal
```

```
anon  = Latch clock
clock = Inv anon
```



Typed adjacency lists (1)

One potential representation of this graph is

```
graph :: Map Integer ???
graph = Map.fromList [(0, Inv 1)
                      (1, Latch 0)]
```

Think adjacency lists, but instead of a node connecting to a list of other nodes we have typed entries. Doesn't typecheck though...

Typed adjacency lists (2)

In order to represent our typed adjacency lists, we need a type that is very similar to **Signal** but that allows arbitrary entries (e.g. integers) in recursive positions.

```
data SignalF a = LatchF a
               | XorF a a
               | AndF a a
               | InvF a

graph :: Map Integer (SignalF Integer)
graph = Map.fromList [(0, InvF 1)
                      (1, LatchF 0)]
```

Recursion Schemes

But of course, **SignalF** is just the signature/pattern functor of **Signal**. In general, a signature functor is a single layer of a recursive data structure. For any signature functor you can usually provide:

```
-- Remove one layer of a recursive data
-- structure off of the top.
project :: Signal -> SignalF Signal
-- Add one layer.
embed   :: SignalF Signal -> Signal
```

In fact, those two functions define two typeclasses,

```
class Recursive t f | t -> f where
  project :: t -> f t
class Corecursive t f | t -> f where
  embed :: f t -> t
```

```haskell
instance Recursive Signal SignalF where
    project (Latch x) = LatchF x
    project (Xor x y) = XorF x y
    project (And x y) = AndF x y
    project (Inv x)   = InvF x
```

Remember how graph traversal algorithms work:

1. Given a queue or a stack of nodes N, take the first element n from N.
2. If you have already visited it, continue to step 1.
3. Otherwise, find its children $\{c_i\}$ and add them to N.

We will focus primarily on DFS.

Remember how DFS works:

1. Given a node n,
2. Check if you have already visited it, and if so, return its identifier.
3. Otherwise, assign it a new id.
4. Find its children $\{c_i\}$ and visit them in any order. Use the returned identifiers to populate a map from ids to nodes.

In our case,

1. Nodes are **Signal**s.
2. We can check whether we have visited a node using guarded referential equality.
3. We can find children of a **Signal** by using **project** or by direct pattern matching on **Signal**.

Easy enough for **Signal**, but how do we generalize?

In general,

1. Nodes are some *recursive* types t (we can require Recursive t f instance).
2. We can check whether we have visited a node using guarded referential equality.
3. Since t is recursive, we can use **project** to extract one layer of t, f t.
4. But now we need some way to go over all children nodes inside of f t...

In general, how do we "fold over" every x in $f\ x$?

We could use **Foldable**, but we will require monadic folds because of our use of guarded referential equality (and hence IO). We need **Traversable** f.

```
instance Traversable f where
  traverse :: Applicative m
           => (a -> m b)
           -> f a -> m (f b)
```

Traversable SignalF allows you to replace every *a* in an *SignalF a* with some applicative (and hence monadic) effect *mb* and then sequence all of them, obtaining *m (SignalF b)* in the end.

```
instance Traversable SignalF where
  traverse :: Applicative m
           => (a -> m b)
           -> SignalF a -> m (SignalF b)
    traverse f (Latch x) = Latch <$> f x
    traverse f (Xor x y) = Xor <$> f x <*> f y
    ...
```

This fits perfect for us, once we obtain a *SignalF Signal*, we can visit each sub-node and replace them with its integer identifier, obtaining *SignalF Integer*.

Back to our graph traversal algorithm:

1. Given a node $n :: t$, where we have *Recursive t f* for some f,
2. Check if you have already visited it using guarded referential equality, and if so, return its identifier.
3. Otherwise, assign it a new id.
4. Otherwise, find its children by extracting one layer of t using *project*.
5. Visit all children using *traverse*.

```haskell
newtype Graph f = Graph (Map Integer (f Integer))

toGraph :: (Recursive t f, Traversable f)
        => t -> IO (Graph f)

fromGraph :: (Corecursive t f, Functor f)
          => Graph f -> t
```

The standard *recursion-schemes* package uses type families instead of *functional dependencies*

```
newtype Graph f = Graph (Map Integer (f Integer))

toGraph :: (Recursive t, Traversable (Base t))
        => t -> IO (Graph (Base t))

fromGraph :: (Corecursive t, Functor (Base t))
          => Graph (Base t) -> t
```

The original paper on this topic actually defined a combined *Recursive* and *Traversable* typeclass.

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

This slide is just a cue for us to check out the source code.

What can we do with this?

So what does this representation allow us to do?

1. We can, given a parser, print its EBNF (demo).
2. We can turn recursive structures into graphs and back.
3. We can check nullability of parsers.
4. Nice DSLs for specifying graphs.

Caveats

So what does this representation allow us to do?

1. We can, given a parser, print its EBNF (demo).
2. We can turn recursive structures into graphs and back.
3. We can check nullability of parsers.
4. Nice DSLs for specifying graphs.

Caveats

```haskell
newtype Node = Node [Node]
data NodeF a = NodeF [a]

magic :: Node -> Graph NodeF
magic = unsafePerformIO . toGraph

test :: Graph NodeF
test = magic a where
    a = Node [b]
    b = Node [c]
    c = Node [a]
```

```haskell
test :: Graph NodeF
test = magic a where
    a = Node [b]
    b = Node [c]
    c = Node [a]

main' :: IO ()
main' = print $ test
-- [(0,NodeF [1]),(1,NodeF [2]),(2,NodeF [3]),
--  (3,NodeF [4]),(4,NodeF [5]),(5,NodeF [3])]
```

```
[(0,NodeF [1]),(1,NodeF [2]),(2,NodeF [3]),
 (3,NodeF [4]),(4,NodeF [5]),(5,NodeF [3])]
-- combine 5 and 2
[(0,NodeF [1]),(1,NodeF [2]),(2,NodeF [3]),
 (3,NodeF [4]),(4,NodeF [2])]
-- combine 4 and 1
[(0,NodeF [1]),(1,NodeF [2]),(2,NodeF [3]),
 (3,NodeF [1])]
-- combine 3 and 0
[(0,NodeF [1]),(1,NodeF [2]),(2,NodeF [0])]
```

Caveats

GHC is allowed to do some non-trivial transformations of our expressions, which may result in duplication and de-duplication of nodes.

HKT

In principle, the same approach works for types of the kind $* \to *$, e.g.

```
data P a where
  Map :: (a -> b) -> P a -> P b
  Str :: a -> String -> P a
  Eps :: a -> P a
  Seq :: (a -> b -> z) -> P a -> P b -> P z
  Alt :: P a -> P a -> P a
```

```haskell
data PF f a where
  MapF :: (a -> b) -> f a -> PF f b
  StrF :: a -> String -> PF f a
  EpsF :: a -> PF f a
  SeqF :: (a -> b -> z) -> f a -> f b -> PF f z
  AltF :: f a -> f a -> PF f a
```

```haskell
newtype Index a = Index Integer
data GraphK (f :: (* -> *) -> *) x =
  GraphK (Index x)
    (forall a. Index a -> f Index a)

parser :: P Int
graph :: GraphK PF Int
```

MonadFix

```
do rec
  a <- f ... a b c
  b <- f ... a b c
  c <- f ... a b c
  return c
```

Is non-trivial to implement in non-lazy languages.