Refined Types

Dom De Re

FP-Syd, 24 Sept, 2014

Partiality

It starts off so innocently:

```
head' :: [a] -> a
head' (x : _)    = x
head' []         = undefined

tail' :: [a] -> [a]
tail' (_ : xs)   = xs
tail' []         = undefined
```

Whats the problem?

"Just don't give it an empty list." - Some Pragmatic Programmer

The Rabbit Hole

First we learn about `Maybe`, adulterate the type to add a rug to sweep the unwanted kernel under.

```
head' :: [a] -> Maybe a
head' (x : _)    = Just x
head' []         = Nothing

tail' :: [a] -> Maybe [a]
tail' (_ : xs)   = Just xs
tail' []         = Nothing
```

We have abstractions to propagate things like Maybe to the end of the control flow:

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

liftA2
    :: (Applicative f)
    => (a -> b -> c)
    -> f a -> f b -> f c

class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

But this is just a case of an unwanted pattern, rather than adulterate the output type, we could *refine* the input type:

```
data NonEmpty a = a :| [a]

nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty (x : xs)    = Just $ x :| xs
nonEmpty []          = Nothing

head' :: NonEmpty a -> a
head' (x :| _) = x

tail' :: NonEmpty a -> [a]
tail' (_ :| xs) = xs
```

Functor

And using the same machinery we can lift these simpler total functions up to the more complicated types:

```
headl :: [a] -> Maybe a
headl = fmap head' . nonEmpty

taill :: [a] -> Maybe [a]
taill = fmap tail' . nonEmpty
```

Applicative

```
liftA2
    :: (Applicative f)
    => (a -> b -> c)
    -> f a -> f b -> f c
```

Monad

```
(=<<)
    :: (Monad m)
    => (a -> m b)
    -> m a -> m b
```

Lens & Traversals

```haskell
fromList :: [a] -> Either [b] (NonEmpty a)
fromList = maybe (Left []) Right . nonEmpty

toList :: NonEmpty a -> [a]
toList (x :| xs) = x : xs

_NonEmpty :: Prism [a] [b] (NonEmpty a) (NonEmpty b)
_NonEmpty = prism toList fromList

dropTail :: NonEmpty a -> NonEmpty a
dropTail (x :| _) = x :| []

-- Provided you are happy with the "do nothing" response
-- for values in the kernel of fromList
over _NonEmpty
    :: (NonEmpty a -> NonEmpty b)
    -> [a] -> [b]
```

So. . . .

We have a **lot** of tools to lift functions on simpler types into functions on more complex types.

This all sounds great, so where does it go wrong?

Rock Bottom

A Binary Tree

```
data Tree a =
        Leaf
    |   Node a (Tree a) (Tree a)
```

Red-Black Tree

```
data Colour = Red | Black

data RBTree a =
        Leaf
    |   Node Colour a (RBTree a) (RBTree a)
```

Oh Wait!

Invariants:

1. Red nodes have no Red Children
2. All paths from the root node to the leaves have the same number of black nodes

Properties

- This is a common use for properties, write up your properties that check that the invariants are valid in the output.
- Write up your `Arbitrary` instance for your type that produces values that satisfy the invariant.

- Sometimes, writing up code that generates the invariant satisfying values ends up being very similar to the code you are testing. . .
- On top of that concern, you have to worry about the coverage of the invariant satisfying subset.

```haskell
insert
    :: (Ord a)
    => a
    -> RBTree a
    -> RBTree a

balance
    :: Colour
    -> a
    -> RBTree a
    -> RBTree a
    -> RBTree a
```

Let's Refine

```
data Colour = Red | Black

data RBTree a =
        Leaf
    |   Node Colour a (RBTree a) (RBTree a)
```

Bam!

```
-- Ignoring Invariant 2 since we only looking at inserts

data RedNode a = RedNode a (BlackNode a) (BlackNode a)

-- technically, the root node is supposed to be black, so
-- represent a red black tree in its final state.
data BlackNode a =
        Leaf
    |   BlackNode a (RedBlack a) (RedBlack a)

data RedBlack a = R (RedNode a) | B (BlackNode a)
```

Oh, and while we are inserting a value into the tree, the tree can be in an intermediate state where Invariant 1 is broken at the root:

```
-- This is assumed to e representing a Red Node at the roo
data Invariant1Broken a =
        LeftRedChild a (RedNode a) (BlackNode a)
    |   RightRedChild a (BlackNode a) (RedNode a)

data InsertState a =
        Ok (RedBlack a)
    |   Broken (Invariant1Broken a)
```

Wooo! Alright, now lets go rewrite those two simple yet incredibly bug prone functions!

Before

```
insert
    :: (Ord a)
    => a
    -> RBTree a
    -> RBTree a

balance
    :: Colour
    -> a
    -> RBTree a
    -> RBTree a
    -> RBTree a
```

After

```
balanceblackl :: a -> InsertState a -> RedBlack a -> RedBla

balanceblackr :: a -> RedBlack a -> InsertState a -> RedBla

fixBroken :: InsertState a -> BlackNode a

ins :: (Ord a) => a -> RedBlack a -> InsertState a

insBlack :: (Ord a) => a -> BlackNode a -> RedBlack a

insRed :: (Ord a) => a -> RedNode a -> InsertState a

joinRedl :: a -> RedBlack a -> BlackNode a -> InsertState a

joinRedr :: a -> BlackNode a -> RedBlack a -> InsertState a

insert :: (Ord a) => a -> BlackNode a -> BlackNode a
```

When the compiler finally released me I had realised I hadn't eaten for 5 days.

But I **don't** have a problem.

- The Rabbit hole goes further with Invariant 2 and deletes, with `DataKinds` or `GADTs`.
- The deletes involve leaving the tree in an intermediate state where invariant 2 is broken.

Not going there in this talk.

Reflection

So whats the problem here?

What's the difference between the `[a]` / `NonEmpty a` case?

- All of the types could be injected into `RBTree a` much like `NonEmpty a` can be injected into `[a]`.
- But theres a conceivable use for `[a]`, values exist.
- Other than implementing the core operations, users of the of the data structure should never encounter values that break the invariants.

Refining Types

- So far when I “refined” a type, I had to write up a completely new distinct type.
- Conversions between all these different types can be potentially inefficient.
- To “refine” `[a]`, had to throw away the `[]` and build `NonEmpty a` from `a` again.

We almost always use Type constructors to adulterate types (except for `Const` and `Identity`)

So could we get more mileage from our types if we could qualify our types to restrict or refine them instead of adulterating them?

```
type RedNode a = { t : RBTree a | ??? }

type BlackNode a = { t : RBTree a | ??? }

type RedBlack a = { t : RBTree a | ??? }

type InsertState a = { t : RBTree a | ??? }
```

Liquid Haskell

A worked example

Red-Black Trees?

Haha, goodness me no.

You can see my very very very early attempt at using `DataKinds` and `GADTs` to do it here

And a recent experiment with Liquid Types that doesn't quite work yet here.

Binomial Trees

A primitive from which Binomial heaps are built:

```
data BinomialTree a = BinomialTree a [a]
```

Defined inductively as follows:

- Binomial tree of Rank 0 is a singleton node.
- A binomial tree of rank `r + 1` is formed by linking two binomial trees of rank `r`, with one becoming a child of the other.

Measures

Liquid Haskell lets you define simple functions to use in constraints. They can't return functions though.

```
{-@
    measure binTreeRank :: BinomialTree a -> Int
    binTreeRank (BinomialTree x cs) = len cs
@-}
```

Refined Types

Or *Liquid Types* (**L**ogically **Q**ualified **D**ata Types).

Similar to Subset Types in **Coq**.

```
{-@ type BinomialTreeN a N = {t : BinomialTree a | (binTre
```

Invariants

The inductive definition results in the following invariant:

- The list of children is ordered by decreasing rank, with each element 1 rank higher than the next...

Encode invariants into the type:

```
data BinomialTreeList a =
        Nil
    |   Cons (BinomialTreeList a) (BinomialTree a)

{-@
    measure listlen :: BinomialTreeList a -> Int
    listlen (Nil)        = 0
    listlen (Cons xs x) = 1 + (listlen xs)
@-}

{-@ invariant {v : BinomialTreeList a | (listlen v) >= 0}

{-@
    type BinomialTreeListN a N = {ts : BinomialTreeList a
@-}

-- Invariant here
{-@
```

Let's store the rank in the structure and add an invariant for that also:

```
data BinomialTree a = BinomialTree Int a (BinomialTreeList

{-@
    measure binTreeRank :: BinomialTree a -> Int
    binTreeRank (BinomialTree r x cs) = r
@-}

{-@
    data BinomialTree a =
        BinomialTree (r :: Int) (x :: a) (cs :: BinomialTr
@-}
```

Can now provide guarantees on the outputs of functions that are statically checked:

```
{-@ binlength :: t : BinomialTreeList a -> {x : Int | x =
binlength :: BinomialTreeList a -> Int
binlength Nil            = 0
binlength (Cons ts _)   = 1 + binlength ts

{-@ rank :: v : BinomialTree a -> {x : Int | x = (binTreeR
rank :: BinomialTree a -> Int
rank (BinomialTree r _ _) = r
-- rank (BinomialTree _ _ cs) = binlength cs
-- rank _ = 0
```

Verify the inductive definition is preserved by our implementation of the core operations:

```
-- | Singleton node defined to have rank 0
{-@ singletonTree :: a -> BinomialTreeN a {0} @-}
singletonTree :: a -> BinomialTree a
singletonTree x = BinomialTree 0 x Nil

-- | Rank r + 1 tree is created by linking together two ran
{-@
    link
        :: (Ord a)
        => w : BinomialTree a
        -> z : BinomialTreeN a {(binTreeRank w)}
        -> BinomialTreeN a {1 + (binTreeRank w)}
@-}
link :: BinomialTree a -> BinomialTree a -> BinomialTree a
link w@(BinomialTree rw x ws) z@(BinomialTree rz y zs)
    | x < y     = BinomialTree (rw + 1) x (Cons ws z)
    | otherwise = BinomialTree (rz + 1) y (Cons zs w)
```

Final Thoughts

Pros

- Don't have to manipulate proofs in parallel with program values.
- Some of the expressive capacity of Dependent Types

Limitations 1

- Only some of the expressive capacity of Dependent Types
- Can use any SMT solver backend apparently, but z3 is the only with a reliable enough reputation
- z3 is not free (Non-Commercial Research use only)
- Using an SMT solver is a little “black boxy”, not sure I would ever want it in the compiler, don’t know if that will ever take off
  - Then again, I didn’t think the IPod was going to be a big deal.

Limitations 2

- If refined types rule out specific patterns (e.g Red Black trees), fails exhaustivity checking in GHC since the function is effectively partial as far as GHC is concerned.
- A lot of the time, expressing properties/invariants/constraints is really just as challenging as doing so in the existing type system.
  - So I don't think we have solved the Type complexity problem yet.
- At the moment its like a separate type system running in parallel, gets a little schizophrenic.
- Terrible error messages

```
{-@ binlength :: t : BinomialTreeList a -> {x : Int | x = (
binlength :: BinomialTreeList a -> Int
binlength Nil           = 0
binlength (Cons ts _)   = 2 + binlength ts
```

```
src/Data/Binomial.hs:75:26-41: Error: Liquid Type Mismatch
   Inferred type
     VV : Int | (VV == (?c + ?a))

   not a subtype of Required type
     VV : Int | (VV == (listlen ?b))

   In Context
     ts : (BinomialTreeList a) | ((listlen ts) >= 0)
     ?b : (BinomialTreeList a) | ((listlen ?b) >= 0)
     ?a : Int | (?a == (listlen ts))
     ?c : Int | (?c == (2  :  int))
```

References And Further Reading

- Z3
- N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell
- Try Liquid Haskell - An Online Interactive Liquid Haskell Demo