Refactoring TypedData to FeatureSets using Rewriter Combinators

Clay Thomas

August 3, 2017

The Basic Problem

Before:

```
ints <- intsToTypedData <$> mkSafeHashMap
  [ ("x", someHaxlInt)
  , ("y", someOtherHaxlInt)
  ]
```

After:

```
ints <- (toTypedData . mconcat)
  [ genFeature "x" <@ someHaxlInt
  , genFeature "y" <@ someOtherHaxlInt
  ]
```

Retrie Review

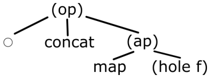
$$\forall\ \texttt{f.concat} \circ \texttt{map f} \quad \mapsto \quad \texttt{concatMap f}$$

$$\forall\texttt{.concat} \circ \texttt{concat} \quad \mapsto \quad \texttt{megaconcat}$$

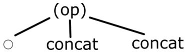
$$\forall\ \texttt{f g.first f} \circ \texttt{second g} \quad \mapsto \quad \texttt{bimap f g}$$

Retrie Review

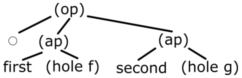
$\forall$ `f.concat ∘ map f` $\mapsto$ `concatMap f`



$\forall$`.concat ∘ concat` $\mapsto$ `megaconcat`

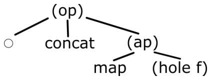


$\forall$ `f g.first f ∘ second g` $\mapsto$ `bimap f g`



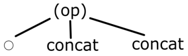
Retrie Review

$\forall$ `f.concat ∘ map f` $\mapsto$ `concatMap f`



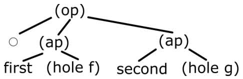
`(op) ∘ concat (ap) map (hole:f)`

$\forall$`.concat ∘ concat` $\mapsto$ `megaconcat`



`(op) ∘ concat concat`

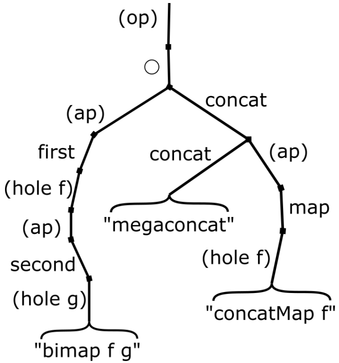
$\forall$ `f g.first f ∘ second g` $\mapsto$ `bimap f g`



```
(op) ∘ (ap) first (hole:f)
    (ap) second (hole:g)
```

Retrie Review

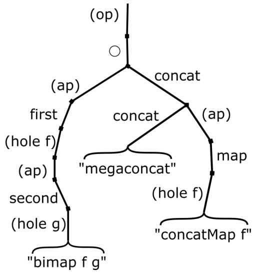
```
(op) ∘ concat (ap) map (hole:f) ↦    concatMap f
              (op) ∘ concat concat ↦    megaconcat
      (op) ∘ (ap) first (hole:f)
              (ap) second (hole:g) ↦     bimap f g
```



Retrie Review

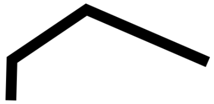
```
type Rewriter = EMap Template

data EMap a =
  { opEMap   :: EMap (EMap (EMap a))
      -- ^      op    lhs    rhs
  , apEMap   :: EMap (EMap a)
      -- ^     func  arg
  , varEMap  :: Data.Map String a
      -- ^                var_name
  , listEMap :: ListMap EMap a
      -- ^ => EMap (EMap (EMap (EMap ...)))
  }
```



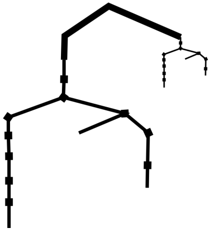
Constructing New Tries

```
ints <- intsToTypedData <$> mkSafeHashMap
    [  ("x", someHaxlInt)
    ,  ("y", someOtherHaxlInt)
    ]
```



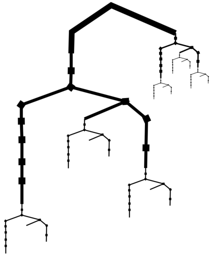
Constructing New Tries

```
ints <- intsToTypedData <$> mkSafeHashMap
    [  ("x", someHaxlInt)
    ,  ("y", someOtherHaxlInt)
    ]
```



Constructing New Tries

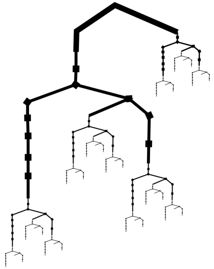
```
ints <- intsToTypedData <$> mkSafeHashMap
    [  ("x", someHaxlInt)
    ,  ("y", someOtherHaxlInt)
    ]
```



Constructing New Tries

```
ints <- intsToTypedData <$> mkSafeHashMap
    [  ("x", someHaxlInt)
    ,  ("y", someOtherHaxlInt)
    ]
```

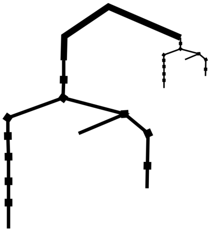
And more!



Constructing New Tries

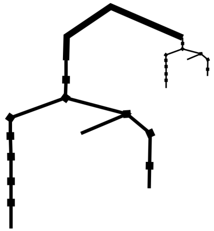
```
thenApply' :: EMap a -> EMap b -> EMap (a,b)
thenApply' func arg = EMap { apEMap = root }
  where
    root = fmap catArg func
    catArg a = fmap (a,) arg


thenApply :: EMap Template
  -> EMap Template -> EMap Template
thenApply x y
  = fmap catTemplates $ thenApply' x y
  where
    catTemplates :: Template -> Template
      -> Template
```



Constructing New Tries

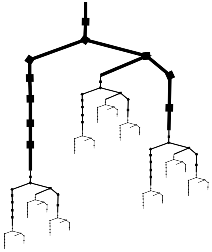
```
thenApply' :: EMap a -> EMap b -> EMap (a,b)
thenApply' func arg = EMap { apEMap = root }
  where
    root = fmap catArg func
    catArg a = fmap (a,) arg
```



```
thenApply :: EMap Template
  -> EMap Template -> EMap Template
thenApply x y
  = fmap catTemplates $ thenApply' x y
  where
    catTemplates :: Template -> Template
      -> Template
```

Constructing New Tries

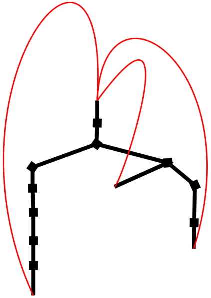
```
cyclicize' :: EMap a -> EMap [a]
cyclicize' emap = reverse <$> list
  where
    list = EMap { listEMap = root }
    root = ListMap
      { nil = [[]]
      , cons = fmap catRoot emap }
    catRoot a = fmap (a:) root
```



```
cyclicize :: EMap Template -> EMap Template
cyclicize = fmap consTemplates . cyclicize'
  where
    consTemplates :: [Template] -> Template
```

Constructing New Tries

```
cyclicize' :: EMap a -> EMap [a]
cyclicize' emap = reverse <$> list
  where
    list = EMap { listEMap = root }
    root = ListMap
      { nil = [[]]
      , cons = fmap catRoot emap }
    catRoot a = fmap (a:) root
```



```
cyclicize :: EMap Template -> EMap Template
cyclicize = fmap consTemplates . cyclicize'
  where
    consTemplates :: [Template] -> Template
```

They Work Kinda!

```
{-# RULES
  "outer" (fmap intsToTypedData . mkSafeHashMap)
    = (toTypedData . mconcat)
  "inner" forall x y. (x,y) = genFeatureInt x <@ y
#-}
... (rule loading, file shuffling)
applyRewriter $ outer `thenApply` cyclicize inner
```

They Work Kinda!

```
{-# RULES
  "outerInt" (fmap intsToTypedData . mkSafeHashMap)
    = (toTypedData . mconcat)
  "outerBool" (fmap boolsToTypedData . mkSafeHashMap)
    = (toTypedData . mconcat)
  ...
  "innerInt" forall x y. (x,y) = genFeatureInt x <@ y
  "innerBool" forall x y. (x,y) = genFeatureBool x <@ y
  ...
#-}
... (rule loading, file shuffling)
applyRewriter . fold
  $ zipWith thenApply outers (map cyclicize inners)
```

Impact

1. Help teams transition to FeatureSet
2. Reduce technical debt
3. 25K lines of code
4. Reduced usage by 2/3

Other Applications

1. Squashing lists of lists

   ```
   flatten' :: EMap a -> EMap b -> EMap [[b]]
   ```

2. Collecting and moving list elements?

   ```
   collect' :: EMap a -> EMap b -> EMap [([a],[b])]
   ```

3. Refactoring every element in a ‘do‘ block?

It Kinda Doesn't Work

1. Holes overlap
2. Formatting