Predicative Tries

Abstract

Traditional lookup tables / tries are limited to comparing paths *literally* - a path must match exactly with the tag accompanying content. Here we present a simple, but useful notion - *predicative* lookup tables, that give us *reflection* in our lookups - the ability to orient the content of a lookup based on the result of our condition.

Background

The rose tree will suit as the backbone for our lookup tables. To reccollect, a rose tree is a list with a nondeterministic number of tails:

```
data RTree a = More a [RTree a]
```

We model our lookup table after a trivial trie, where steps down the path are merely constructor elements of our rose tree, and each element of the tree is paired with a tag implementing equality:

```
type Trie t a = RTree (t, Maybe a)
```

We now have potential contents and a path to find them. Implementing a `lookup` function is trivial:

```
lookup :: Eq t => [t] -> Trie t a -> Maybe a
lookup [] _ = Nothing
lookup (t:ts) (More (p,mx) xs)
  | t == p = case ts of
               [] -> mx
               _  -> firstJust $ map (lookup ts) xs
  | otherwise = Nothing

  where
    firstJust :: [Maybe a] -> Maybe a
    firstJust [] = Nothing
    firstJust (Nothing  : xs) = firstJust xs
    firstJust ((Just x) : xs) = Just x
```

The implementation is simple - walk down the tree, testing for equality for each chunk of the path, and if the endpoint is found, return the possible contents, otherwise fail.

Predicative Tries

Existential types have a bad rap - they quantify types beyond the top-level scope, forbidding us from realizing their virtue without knowing it in advance. Using GHC's `-XExistentialTypes` language extension, we can create such attrocities freely.

The predicates we use in our lookup tables will be of such shameful types. Each predicate will be a *mutation* of our tag type `t`, to *some* `r`, such that our predicate will have a type `pred :: forall r. t -> Maybe r` - a boolean condition that also generates a new value, of some unknown type.

What might we do with such type? For one thing, we may prefix our lookup contents by `r` to give us the reflection we desire. Indeed, this is what we do by giving a new constructor for our rose trees:

```
data PTrie t a =
             PMore (t           , Maybe       a) [PTrie t a]
 | forall r. PPred (t -> Maybe r, Maybe (r -> a) [PTrie t (r -> a)]
```

This may look strange, but indeed it is useful. We may now adjust our lookup function above to reciprocate the results of our predicate:

```
lookup :: Eq t => [t] -> PTrie t a -> Maybe a
lookup [] _ = Nothing
lookup (t:ts) (PMore (p,mx) xs)
  | t == p = case ts of
               [] -> mx
               _  -> firstJust $ map (lookup ts) xs
  | otherwise = Nothing
lookup (t:ts) (PPred (q,mrx) xrs) =
  q t >>=
    \r -> case ts of
            [] -> ($ r) <$> mrx
            _  -> ($ r) <$> firstJust $ map (lookup ts) xrs

  where
    firstJust :: [Maybe a] -> Maybe a
    firstJust [] = Nothing
    firstJust (Nothing  : xs) = firstJust xs
    firstJust ((Just x) : xs) = Just x
```

In the predicative constructor case, we leverage the monadic / functorial behaviour of `Maybe`, in that the success of the condition pulls the content out of `Just` and applies it to `r`, which we then use as a parameter to the possible content of `mrx` or later content as we walk down the tree.

Conclusion

We may now have lookup tables who's contents *interact* with the results of mutative acceptor functions. This has many uses, and is critically important for `nested-routes`, a Haskell library for url routing, where parsers are the predicative mutator.