Predicative Tries

Athan Clark
athan.clark@gmail.com

Abstract

Traditional lookup tables and 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

Rose trees [K.A. Heller, 2010] are a classic example of an elegant, purely functional data strucure. To reccollect, rose trees are, constructively, very similar to lists. Traditionally, lists are represented by a union type between a [] unit data constructor, and a : data constructor, with a type signature $(:) :: \alpha \to [\alpha] \to [\alpha]$. For verbosity, here is the traditional implementation:

$$\begin{aligned}[\alpha] &= \alpha : [\alpha] \\ &\mid\ []\end{aligned}$$

The *Maybe* data type, popular with Haskell [S. Marlow, 2010] developers, has a similar shape:

$$\begin{aligned}Maybe\ \ \alpha &= Just\ \ \alpha \\ &\mid\ Nothing\end{aligned}$$

From this, we can refactor the traditional list design into one with a more explicit possibly nonexistent tail:

$$[\alpha]' = \alpha :' (Maybe\ \ [\alpha]')$$

If we substitute *RTree* for $[]'$, and replace *Maybe* with a *set* of tails, we can see the corrospondance to lists:

$$RTree\ \ \alpha = More\ \ \alpha\ \ [RTree\ \ \alpha]$$

Trivial Trie

We model our lookup table after a trivial trie [E. Fredkin, 1960], 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:

$$Trie\ \ t\ \ \alpha \sim RTree\ \ (t,\ \ Maybe\ \ \alpha)$$

Now we have potential contents and a path to find them. Implementing a *lookup* function is trivial:

$$\begin{aligned}
&lookup :: (Eq\ \ t) \Rightarrow [t] \to Trie\ \ t\ \ a \to Maybe\ \ a \\
&lookup\ \ []\ \ _{=}\ \ Nothing \\
&lookup\ \ (t : ts)\ \ (More\ \ (t',\ \ mx)\ \ xs) \\
&\quad\quad \mid\ t \equiv t' = \text{case}\ \ ts\ \ \text{of} \\
&\quad\quad\quad []\to mx \\
&\quad\quad\quad _ \to firstJust\ \ \$\ \ map\ \ (lookup\ \ ts)\ \ xs \\
&\quad\quad \mid\ otherwise = Nothing \\
&\quad where \\
&\quad firstJust :: [Maybe\ \ \alpha] \to Maybe\ \ \alpha \\
&\quad firstJust\ \ [] = Nothing \\
&\quad firstJust\ \ (Nothing : xs) = firstJust\ \ xs \\
&\quad firstJust\ \ ((Just\ \ x) : xs) = Just\ \ x
\end{aligned}$$

The implementation is fairly 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 Trie

Existential types have a bad rap - they quantify types scope, forbidding us from realizing their virtue without knowing it in advance. Using GHC's *ExistentialTypes* [A. Dijkstra, 2010] language extension, we can create such attrocities without obligation.

The predicates we use in our lookup tables will leverage such freedom. Each predicate will be a *conditional mutation* of our tag type t, to **some** r, such that our predicates will have a type $\forall r.t \to Maybe\ \ r$ - a boolean condition that generates a new value, of some unknown type.

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

$$
\begin{aligned}
PTrie\ \ t\ \ \alpha = PMore\ & \\
& (t, \qquad\qquad Maybe\ \ \alpha) \\
& [PTrie\ \ t\ \ \alpha] \\
\mid\ PPred\ \forall r. & \\
& (t \to Maybe\ \ r,\ \ Maybe\ \ (r \to \alpha)) \\
& [PTrie\ \ t\ \ (r \to \alpha)]
\end{aligned}
$$

This may look strange, but indeed it is useful. However, there is a major caveat - in **every children set**, *PPred* constructors must be **after** *PMore* constructors. This is because predicates have a *wider* capture range than literal lookups. This is a crucial, yet subtle, prerequisite for correct operation. Likewise, we cannot have identical predicates and expect them to behaive independently - *all overlapping predicates* will seive down the list of child tries - the behaviour is "first come, first serve" with overlapping predicates.

We may now adjust our lookup function above to reciprocate the results of our predicate:

$$
\begin{aligned}
& lookup :: (Eq\ \ t) \Rightarrow [t] \to Trie\ \ t\ \ a \to Maybe\ \ a \\
& lookup\ \ []\ \ =\ \ Nothing \\
& lookup\ \ (t : ts)\ \ (PMore\ \ (t',\ \ mx)\ \ xs) \\
& \quad \mid\ t \equiv t' = \text{case}\ \ ts\ \ \text{of} \\
& \qquad [] \to mx \\
& \qquad \_ \to firstJust\ \ \$\ \ map\ \ (lookup\ \ ts)\ \ xs \\
& \quad \mid\ otherwise = Nothing \\
& lookup\ \ (t : ts)\ \ (PPred\ \ (p,\ \ mrx)\ \ xrs) = \\
& \quad p\ \ t\ \ \gg= \\
& \qquad \lambda r.\ \text{case}\ \ ts\ \ \text{of} \\
& \qquad\quad [] \to fmap\ \ (\$\ \ r)\ \ mrx \\
& \qquad\quad \_ \to fmap\ \ (\$\ \ r) \\
& \qquad\qquad\qquad (firstJust\ \ \$\ \ map\ \ (lookup\ \ ts)\ \ xrs) \\
& \ \ where \\
& \ \ firstJust :: [Maybe\ \ \alpha] \to Maybe\ \ \alpha \\
& \ \ firstJust\ \ [] = Nothing \\
& \ \ firstJust\ \ (Nothing : xs) = firstJust\ \ xs \\
& \ \ firstJust\ \ ((Just\ \ x) : xs) = Just\ \ x
\end{aligned}
$$

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.

I. 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.

References

[K.A. Heller, 2010] FC. Blundell, Y.W. Teh and K.A. Heller (2010). "Bayesian Rose Trees". In the 26th Conference on *Uncertainty in Artificial Intelligence*, UIA 2010.

[S. Marlow, 2010] Simon Marlow, Simon Peyton Jones (2010). The 2010 Haskell Language Report

[E. Fredkin, 1960] Edward Fredkin (1960). "Trie Memory". In *Communications of the ACM*, doi:10.1145/367390.367400

[A. Dijkstra, 2010] Atze Dijkstra (2010). "Existential Haskell".