

---

# Predicative Tries

ATHAN CLARK

Unaffiliated

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 structure. To recollect, 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  $(:) :: \alpha \rightarrow [\alpha] \rightarrow [\alpha]$ . For verbosity, here is the traditional implementation:

$$[\alpha] = \alpha : [\alpha] \\ | []$$

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

$$\text{Maybe } \alpha = \text{Just } \alpha \\ | \text{Nothing}$$

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

$$[\alpha]' = \alpha :' (\text{Maybe } [\alpha]')$$

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

$$\text{RTree } \alpha = \text{More } \alpha [\text{RTree } \alpha]$$

## 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:

$$\text{Trie } t \ \alpha \sim \text{RTree } (t, \text{Maybe } \alpha)$$

Now we 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 (t', mx) xs)
  | t == t' = 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 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 atrocities 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  $\forall r. t \rightarrow \text{Maybe } r$  - 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:

$$\begin{aligned} \text{PTrie } t \ \alpha &= \text{PMore} \\ &\quad (t, \quad \text{Maybe } \alpha) \\ &\quad [\text{PTrie } t \ \alpha] \\ &\quad | \text{PPred } \forall r. \\ &\quad (t \rightarrow \text{Maybe } r, \text{Maybe } (r \rightarrow \alpha)) \\ &\quad [\text{PTrie } t \ (r \rightarrow \alpha)] \end{aligned}$$

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

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

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".