```
module Trie where

import Types

import Data.Maybe
import qualified Data.Map as Map
import qualified Data.Set as Set
import Debug.Trace

data Trie a = Trie { children :: Map.Map a (Trie a) }
  deriving (Show)

emptyTrie :: Trie PatToken
emptyTrie = Trie { children = Map.empty }

isEnd :: Trie PatToken -> Bool
isEnd t = Map.null (children t)

expandPat :: Env -> PatToken -> Pattern -> Pattern
expandPat env pt pat =
  let len = arityOfPatToken env pt in
  generateWildcards len pat

arityOfPatToken :: Env -> PatToken -> Int
arityOfPatToken env pat = case Map.lookup pat env of
  Nothing -> error $ "Unable to find constructor: " ++ show pat
  Just kids -> fromJust $ Map.lookup pat kids

generateWildcards :: Int -> Pattern -> Pattern
generateWildcards 0 continuation = continuation
generateWildcards len continuation =
  Pat PTWild $ Just $ generateWildcards (len - 1) continuation

isCompleteLevel :: Env -> Map.Map PatToken (Trie PatToken) -> Bool
isCompleteLevel env children =
  let nonWildcards = Map.filterWithKey (\k _ -> k /= PTWild) children in
  trace ("IS COMPL: " ++ show nonWildcards) $
  if Map.null nonWildcards then False else
    -- Now grab an example elem
    let (exampleElem, _) = Map.findMin nonWildcards
        -- Grab the constructors that we expect
        sigmaSize = Map.size $ fromJust (Map.lookup exampleElem env)
        -- Grab the constructors that we _have_
        constrSize = Map.size nonWildcards in
    sigmaSize == constrSize

-- | Given a pattern @p, determine if that pattern is in the trie
useful :: Env -> Pattern -> Trie PatToken -> (Trie PatToken, Bool)
useful env (Pat PTWild Nothing) (Trie t) =
  case isCompleteLevel env t of
    True -> (Trie t, False)
    False -> case Map.lookup PTWild t of
                  Nothing -> (Trie (Map.insert PTWild emptyTrie t), True)
                  Just l -> (Trie t, False)
useful env (Pat p Nothing) (Trie t) =
  case Map.lookup p t of
    Nothing -> (Trie (Map.insert p emptyTrie t), True)
    Just l -> (Trie t, False)
useful env (Pat p (Just rest)) (Trie children) =
  case p of
    PTConstr str ->
      case Map.lookup p children of
        Nothing ->
          let (trie, _) = useful env rest emptyTrie in
          (Trie (Map.insert p trie children), True)
        Just node ->
          let (trie, b) = useful env rest node in
          (Trie (Map.insert p trie children), b)
    PTWild ->
       if isCompleteLevel env children
       then
          -- for each pattern token in children generate wildcards of the proper
          -- arity and then go down that constructors path
```

```
          -- Expand out the pattern so that it has the proper number of wildcards
          -- to match the arity of the constructor of edge that we are traversing.
          let res = (flip Map.mapWithKey) children (\k node -> let rest' = expandPat
 env k rest in
                          let (trie, b) = useful env rest' node in
                          (Trie (Map.insert p trie children), b)) in
          let res' = Map.filter snd res in
          if not (Map.null res')
          then let (_, (trie, b)) = (Map.findMin res') in
               (Trie (Map.insert p trie children), b)
          else (Trie children, False)
       else
         let wildcard = Map.filterWithKey (\k _ -> k == PTWild) children in
          case Map.size wildcard of
            0 ->
              let (trie, b) = useful env rest emptyTrie in
              (Trie (Map.insert p trie children), b)
            _ -> let (trie, b) = useful env rest (snd $ Map.findMin wildcard) in
                 (Trie (Map.insert p trie children), b)

exhaustive :: Env -> Match -> (Trie PatToken, Bool)
exhaustive env (Match str ps) =
  let check trie pats = case pats of
        [] ->
          let (t, b) = useful env patwild trie in
          if b then (t, False)
          else (t,True)
        (p:ps) ->
          let pp = convertPat p Nothing in
          let (trie', b) = useful env pp trie in
          if b then check trie' ps
          else trace (show trie') $
          error ("Dead pattern found " ++ show p)
  in check emptyTrie ps
```