Chapter 1

Prototype 1

1.1 About this chapter

This chapter demonstrates a ”fairly generic” procedure of creating an open embedded domain specific language in HASKELL along with *monadic unification*. As a proof of concept, the implementation consists of creating a PROLOG like open language whose unification procedure is carried out in a monad.

1.2 Components

There are four main components that we work with to develop a working implementation of embedded PROLOG using the concepts mentioned above.

1. PROLOG

   The language itself has a number of sub components, the ones relevant to this thesis are,

   (a) Language, the syntax, semantics.

   (b) Database, or the knowledge base where the rules are stored.

(c) Unification

(d) The search strategy which is used to list and accomplish goals.

(e) And finally the query resolver which combines the unification and search strategy to return a result.

2. prolog-0.2.0.1 [**?**]

   One of the existing implementation of PROLOG in HASKELL though partial provides a starting point for the implementation providing certain components to exercise our approach. The main components of this library are adopted from PROLOG and modified,

   (a) Language, adopted from PROLOG but trimmed down.

   (b) Database

   (c) Unifier

   (d) REPL

   (e) Interpreter which consists of a parsing mechanism and resembles the query resolver.

3. unification-fd [**?**]

   This library provides tools for first-order structural unification over general structure types along with mechanisms for a modifiable generic unification algorithm implementation.

   The relevant components are,

   (a) Unifiable Class

   (b) UTerm data type

   (c) Variables, STVar, IntVar

(d) Binding Monad

(e) Unification (unify and unifyOccurs)

4. Prototype 1

   This implementation applies to practice the procedure to create an open language to accommodate types, custom variables, quantifiers and logic and recovering primitives while preserving the structure of a language commonly defined by a recursive abstract syntax tree. The resulting language is then adapted to apply a PROLOG like unification.

   The implementation consists of the following components,

   (a) An open language

   (b) Compatibility with the unification library [**?**]

   (c) Variable Bindings

   (d) Monadic Unification

Each of the components are discussed in the following sections.

1.3 How Prolog works ?

To replicate PROLOG we look into how it works [**?**].

Most PROLOG distributions have three types of terms:

1. Constants.

2. Variables.

3. Complex terms.

Two terms can be unified if they are the same or the variables can be assigned to terms such that the resulting terms are equal.

The possibilities could be,

1. If term1 and term2 are constants, then term1 and term2 unify if and only if they are the same atom, or the same number.

```
?- =(mia,mia).
yes
```

2. If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 unify, and term2 is instantiated to term1 . (So if they are both variables, theyre both instantiated to each other, and we say that they share values.)

```
?- mia = X.
X = mia
yes
```

```
?- X = Y.
yes
```

3. If term1 and term2 are complex terms, then they unify if and only if:

   (a) They have the same functor and arity, and

   (b) all their corresponding arguments unify, and

   (c) the variable instantiations are compatible.

```
?- k(s(g),Y) = k(X,t(k)).
X = s(g)
Y = t(k)
yes
```

4. Two terms unify if and only if it follows from the previous three clauses that they unify.

Unification is just a part of the process were the language attempts to find a solution for the given query using the rules provided in the knowledge base. The other part is actually reaching a point where two terms need to be unified i.e searching. Together they form the query resolver in PROLOG.

For example, consider the append function

```
1 append([],L,L).
2 append([H|T],L2,[H|L3])  :-  append(T,L2,L3).
```

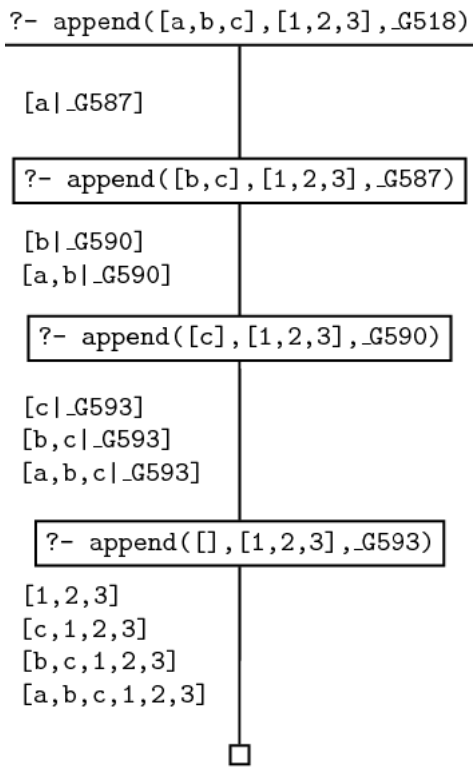

Figure 1.1: Trace for append [**?**]

In this prototype we explore the unification aspect only.

1.4 What we do in this Prototype

This prototype throws light on the process of tackling the issues involved in creating a data type to replicate the target language type system while conforming to the host language restrictions and also utilizing the benefits.

We have a PROLOG like language in HASKELL defined via *data*.

The language defined is recursive in nature.

We convert it into a non recursive data type.

Basically we do Unification monadically.

1.5 Creating a data type

To start we need to define a abstract syntax for the PROLOG like language. But there is a conflict between the type systems as we shall discuss.

A type system consists of a set of rules to define a ”type” to different constructs in a programming language such as variables, functions and so on. A static type system requires types to be attached to the programming constructs before hand which results in finding errors at compile time and thus increase the reliability of the program. The other end is the dynamic type system which passes through code which would not have worked in former environment, it comes of as less rigid.

The advantages of static typing [**?**]

1. Earlier detection of errors
2. Better documentation in terms of type signatures
3. More opportunities for compiler optimizations
4. Increased run-time efficiency
5. Better developer tools

For dynamic typing

1. Less rigid
2. Ideal for prototyping / unknown / changing requirements or unpredictable behaviour
3. Re-usability

Since HASKELL is statically type we would need to define a ”typed” language which would have a number of constructs representing different terms in PROLOG such as complex structures (for example predicates, clauses etc.), don’t cares, cuts, variables and so on.

Consider the language below which has been adopted from [**?**],

```
data VariableName = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)
data Atom         = Atom      !String
                  | Operator  !String
      deriving (Eq, Ord, Data, Typeable)
data Term = Struct Atom [Term]
          | Var VariableName
          | Wildcard
          | Cut Int
      deriving (Eq, Data, Typeable)
data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
            | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
      deriving (Data, Typeable)
type Program = [Sentence]
type Body    = [Goal]
data Sentence = Query   Body
              | Command Body
              | C Clause
      deriving (Data, Typeable)
```

Even though *Term* has a number of constructors the resulting construct has a single type. Hence, a function would still be untyped / singly typed,

```
append :: [Term] -> [Term] -> [Term]
```

The above is a classic example of a recursive grammar to define the abstract syntax of a language. One of the issues with the above is that it is not possible to distinguish the structure of the data from the data type itself [**?**]. Moreover, the primitives of the language are not accessible as the language can have expressions of only one type i.e. ”Term”. The solution to would be to add a type constructor

split the data type into two levels, a single recursive data type is replaced by two related data types. Consider the following,

```haskell
data FlatTerm a =
                  Struct Atom [a]
          |       Var VariableName
          |       Wildcard
          |       Cut Int deriving (Show, Eq, Ord)
```

One result of the approach is that the non-recursive type *FlatTerm* is modular and generic as the structure "FlatTerm" is separate from it's type which is "a". The above language can be of any type *a*. A more accurate way of saying it would be that *a* can be a *kind* in HASKELL.

In type theory, a kind is the type of a type constructor or, less commonly, the type of a higher-order type operator. A kind system is essentially a simply typed lambda calculus 'one level up,' endowed with a primitive type, denoted * and called 'type,' which is the kind of any (monomorphic) data type for example [**?**],

```haskell
Int :: *
Maybe :: * -> *
Maybe Bool :: *
a -> a :: *
[] :: * -> *
(->) :: * -> * -> *
```

Simply speaking we can have something like

```haskell
FlatTerm Bool
```

and a generic fuinction like,

```haskell
map :: (a -> b) -> FlatTerm a -> FlatTerm b
```

Although one problem remains, how does one represent infinitely nested / deep expressions of the above language, for example something of the form,

```haskell
FlatTerm(FlatTerm (FlatTerm (FlatTerm (..... (a)))))
```

and how to represent it generically to perform operations on it since,

```
(FlatTerm a) != (FlatTerm (FlatTerm a))
```

because with our original grammar all the expression that could be defined would be represented by a single entity ”Term” no matter how infinitely deep they were.

The approach to tackling this problem is to find the ”fixed-point”. After infinitely many iterations we should get to a fix point where further iterations make no difference. It means that applying one more ExprF would not change anything a fix point does not move under FlatTerm.

HASKELL provides it in two forms,

1. The fix function in the `Control.Monad.Fix` module allows for the definition of recursive functions in HASKELL. Consider the following scenario,

   ```
   fix :: (a -> a) -> a
   ```

   The above function results in an infinite application stream,

   ```
   f s : f (f (f (... )))
   ```

   A fixed point of a function f is a value a such that f a == a. This is where the name of fix comes from: it finds the least-defined fixed point of a function.

2. And in type constructor form,

   ```
   newtype Fix f = f (Fix f)
   ```

   which we apply to our abstract syntax.

The resulting language is of the form,

```
data Prolog = P (Fix FlatTerm) deriving (Show,Eq,Ord)
```

simply speaking all the expressions resulting from *FlatTerm* can be represented by the type signature *Fix FlatTerm*.

A sample function working with such expressions would be of the form,

```
func :: Fix FlatTerm -> Fix FlatTerm
```

Generically speaking, the language can be expanded for additional functionality without changing or modifying the base structure. Consider the scenario where the language needs to accommodate additional type of terms,

1. Manually modifying the structure of the language,

```
type Atom                 = String

data VariableName         = VariableName Int String
      deriving (Eq, Data, Typeable, Ord)

data Term                           = Struct Atom [Term]
                                    | Var VariableName
                                    | Wildcard
                                    | Cut Int
                                    | New_Constructor_1 .........
                                    | New_Constructor_2 .........
      deriving (Eq, Data, Typeable)
```

   This would then trigger a ripple effect thorughout the architecture because accomodations need to be made for the new functionality.

2. The other option would be to *functorize* language like we did by adding a type variable which can be used to plug something that provides the functionality into the language. Consider the following example,

```
data Box f = Abox | T f (Box f) deriving (.........)
```

   then something like,

```
T (Struct 'atom' [Abox, T (Cut 0)])
```

is possible. Since we needed the fixed point of the language we used *Fix* but generically one could add multiple custom functionality.

1.6 Working with the language / Making language compatible with unification-fd

Our language now opened up and is ready for expansion but it still needs to conform to the requirements of the [**?**] library so that the generic unification algorithm upon customization works with it.

The library provides functionality for first-order structural unification over general structure types along with mutable variable bindings.

In this section we discuss

1. Functor Hierarchy.
2. Required instances the language must have.
3. Mutable variables.
4. Variable Bindings.
5. Monadic Unification.
6. Replicating PROLOG unification in HASKELL

Classes in HASKELL are like containers with certain properties which can be thought of as functions. When a data type creates an instance of a class the function(s) can be applied to each element / primitive in the data type.

The data here is the PROLOG abstract syntax and the containers are *Functor, Foldable, Traversable, Applicative* and *Monad*.

Below 1.2 shows the relation between the different classes.

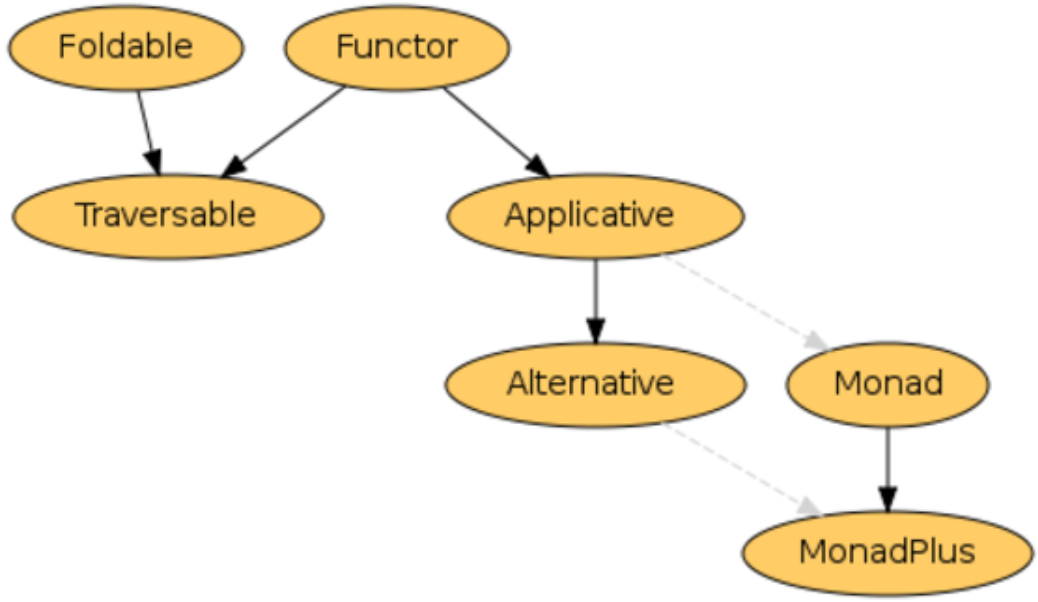

Figure 1.2: Functor Hierarchy [**?**]

The Functor and Foldable instances providing functions for applying map-reduce to the data structure. The primary issue is that at the end of the operation the structure if the data type is lost which would not help our cause since the result of a query must be a list of substitutions which are essential pairs of language variable with language values(language constructs).

Enter, Traversable, it allows reduce whilst preserving the shape of the structure. Lastly, the Applicative instance is an intermediate between a functor and a monad.

We create the necessary instances as follows,

```
instance Functor (FlatTerm) where
        fmap = T.fmapDefault

instance Foldable (FlatTerm) where
         foldMap = T.foldMapDefault

instance Traversable (FlatTerm) where
          traverse f (Struct atom x)        =        Struct atom <$>
                                    sequenceA (Prelude.map f x)
```

```
          traverse _ (Var v)        =        pure (Var v)
          traverse _ Wildcard        =        pure (Wildcard)
          traverse _ (Cut i)         =        pure (Cut i)

instance Applicative (FlatTerm) where
        pure x = Struct "" [x]
        _ <*> Wildcard        =        Wildcard
        _ <*> (Cut i)         =        Cut i
        _ <*> (Var v)         =       (Var v)
        (Struct a fs) <*> (Struct b xs) = Struct (a ++ b) [f x | f <- fs, x <- xs]
```

The above lay the foundation to work with the library. Coming back to the library, the language must have the Unifiable instance. This works in tandom with the UTerm. The UTerm data type captures the recursive structure of logic terms i.e, given some functor t which describes the constructors of our logic terms, and some type v which describes our logic variables, the type UTerm t v is the type of logic terms: trees with multiple layers of t structure and leaves of type v. The Unifiable class gives one step of the unification process. Just as we only need to specify one level of the ADT (i.e., T) and then we can use the library's UTerm to generate the recursive ADT, so too we only need to specify one level of the unification (i.e., zipMatch) and then we can use the library's operators to perform the recursive unification, subsumption, etc

```
instance Unifiable (FlatTerm) where
        zipMatch (Struct al ls) (Struct ar rs) =
                if (al == ar) && (length ls == length rs)
                        then Struct al <$>
                                pairWith (\l r -> Right (l,r)) ls rs
                        else Nothing
        zipMatch Wildcard _ = Just Wildcard
        zipMatch _ Wildcard = Just Wildcard
        zipMatch (Cut i1) (Cut i2) = if (i1 == i2)
                then Just (Cut i1)
                else Nothing
```

Logic Variables

The library ships with two implementations of logic variables. The IntVar implementation uses Int as the names of variables, and uses an IntMap to keep track of the environment.

The STVar implementation uses STRefs, so we can use actual mutation for binding logic variables, rather than keeping an explicit environment around. This implementation uses STVars.

Performing unification has the side effect of binding logic variables to terms. Thus, we'll want to use a monad in order to keep track of these effects. The BindingMonad type class provides the definition of what we need from our ambient monad. In particular, we need to be able to generate fresh logic variables, to bind logic variables, and to lookup what our logic variables are bound to. The library provides the necessary instances for both IntVar and STVar.

You can provide your own implementations of Variable and BindingMonad.

For our language expressions to be unifiable we must deal with the variables in the expressions being compared. For that we extract the variables and then convert them into a dictionary consisting of a free variable for each language variable.

```
variableExtractor :: Fix FlatTerm -> [Fix FlatTerm]
variableExtractor (Fix x) = case x of
        (Struct _ xs)         ->        Prelude.concat $ Prelude.map variableExtra
        (Var v)                       ->        [Fix $ Var v]
        _                               ->        []

variableNameExtractor :: Fix FlatTerm -> [VariableName]
variableNameExtractor (Fix x) = case x of
        (Struct _ xs)        -> Prelude.concat $ Prelude.map variableNameExtractor
        (Var v)                       -> [v]
        _                               -> []

variableSet :: [Fix FlatTerm] -> S.Set (Fix FlatTerm)
variableSet a = S.fromList a

variableNameSet :: [VariableName] -> S.Set (VariableName)
variableNameSet a = S.fromList a

varsToDictM :: (Ord a, Unifiable t) =>
    S.Set a -> ST.STBinding s (Map a (ST.STVar s t))
varsToDictM set = foldrM addElt Map.empty set where
  addElt sv dict = do
    iv <- freeVar
```

```
    return $! Map.insert sv iv dict
```

The next step would wrap the language terms into a UTerm,

```
uTermify
  :: Map VariableName (ST.STVar s (FlatTerm))
  -> UTerm FlatTerm (ST.STVar s (FlatTerm))
  -> UTerm FlatTerm (ST.STVar s (FlatTerm))
uTermify varMap ux = case ux of
  UT.UVar _                              -> ux
  UT.UTerm (Var v)                      -> maybe (error "bad map") UT.UVar $ Map.looku
 -- UT.UTerm t                              -> UT.UTerm £! fmap (uTermify varMap)
  UT.UTerm (Struct a xs)          -> UT.UTerm $ Struct a $! fmap (uTermify varMap) x
  UT.UTerm (Wildcard)                -> UT.UTerm Wildcard
  UT.UTerm (Cut i)                        -> UT.UTerm (Cut i)

translateToUTerm ::
    Fix FlatTerm -> ST.STBinding s
            (UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)),
             Map VariableName (ST.STVar s (FlatTerm)))
translateToUTerm e1Term = do
  let vs = variableNameSet $ variableNameExtractor e1Term
  varMap <- varsToDictM vs
  let t2 = uTermify varMap . unfreeze $ e1Term
  return (t2,varMap)
```

and for later use to convert them back,

```
vTermify :: Map Int VariableName ->
            UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)) ->
            UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm))
vTermify dict t1 = case t1 of
  UT.UVar x  -> maybe (error "logic") (UT.UTerm . Var) $ Map.lookup (UT.getVarID x
  UT.UTerm r ->
    case r of
      Var iv   -> t1
      _          -> UT.UTerm . fmap (vTermify dict) $ r

translateFromUTerm ::
    Map VariableName (ST.STVar s (FlatTerm)) ->
    UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)) -> Prolog
translateFromUTerm dict uTerm =
  P .  maybe (error "Logic") id . freeze . vTermify varIdDict $ uTerm where
    forKV dict initial fn = Map.foldlWithKey' (\a k v -> fn k v a) initial dict
    varIdDict = forKV dict Map.empty $ \ k v -> Map.insert (UT.getVarID v) k
```

Unification

Starting from scratch each of the two language expressions needs to be functorized and then translated to UTerms

```haskell
monadicUnification :: (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  ErrorT (UT.UFailure (FlatTerm) (ST.STVar s (FlatTerm)))
            (ST.STBinding s) (UT.UTerm (FlatTerm) (ST.STVar s (FlatTerm)),
             Map VariableName (ST.STVar s (FlatTerm)))))
monadicUnification t1 t2 = do
--  let
--    t1f = termFlattener t1
--    t2f = termFlattener t2
  (x1,d1) <- lift . translateToUTerm $ t1
  (x2,d2) <- lift . translateToUTerm $ t2
  x3 <- U.unify x1 x2
  --get state from somehwere, state -> dict
  return $! (x3, d1 `Map.union` d2)

goUnify ::
  (forall s. (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  =>
      (ErrorT
          (UT.UFailure FlatTerm (ST.STVar s FlatTerm))
          (ST.STBinding s)
          (UT.UTerm FlatTerm (ST.STVar s FlatTerm),
             Map VariableName (ST.STVar s FlatTerm)))
     )
  -> [(VariableName, Prolog)]
goUnify test = ST.runSTBinding $ do
  answer <- runErrorT $ test --ERROR
  case answer of
    (Left _)            -> return []
    (Right (_, dict))   -> f1 dict


f1 ::
  (BindingMonad FlatTerm (STVar s FlatTerm) (ST.STBinding s))
  => (forall s. Map VariableName (STVar s FlatTerm)
      -> (ST.STBinding s [(VariableName, Prolog)])
     )
f1 dict = do
  let ld1 = Map.toList dict
  ld2 <- Control.Monad.Error.sequence [ v1 | (k,v) <- ld1, let v1 = UT.lookupVar v
```

```
40    let ld3 = [ (k,v) | ((k,_),Just v) <- ld1 `zip` ld2]
41        ld4 = [ (k,v) | (k,v2) <- ld3, let v = translateFromUTerm dict v2 ]
42    return ld4
```

1.7 Chapter Recap