Chapter 1

Work Completed

1.1 What we are doing

A partial implementation of the logic programming language PROLOG is provided by the library `prolog-0.2.0.1`. One of the objectives is to implement monadic unification using the library [**?**].

1.2 Unifiable Data Structures

For a data type to be Unifiable, it must have instances of Functor, Foldable and Traversable. The interaction between different classes is depicted in figure 1.1.

The Functor class provides the `fmap` function which applies a particular operation to each element in the given data structure. The Foldable class *folds* the data structure by recursively applying the operation to each element and

1.3 Why `Fix` is necessary?

Since HASKELL is a lazy language it can work with infinite data structures. *Type Synonyms* in HASKELL cannot be self referential.

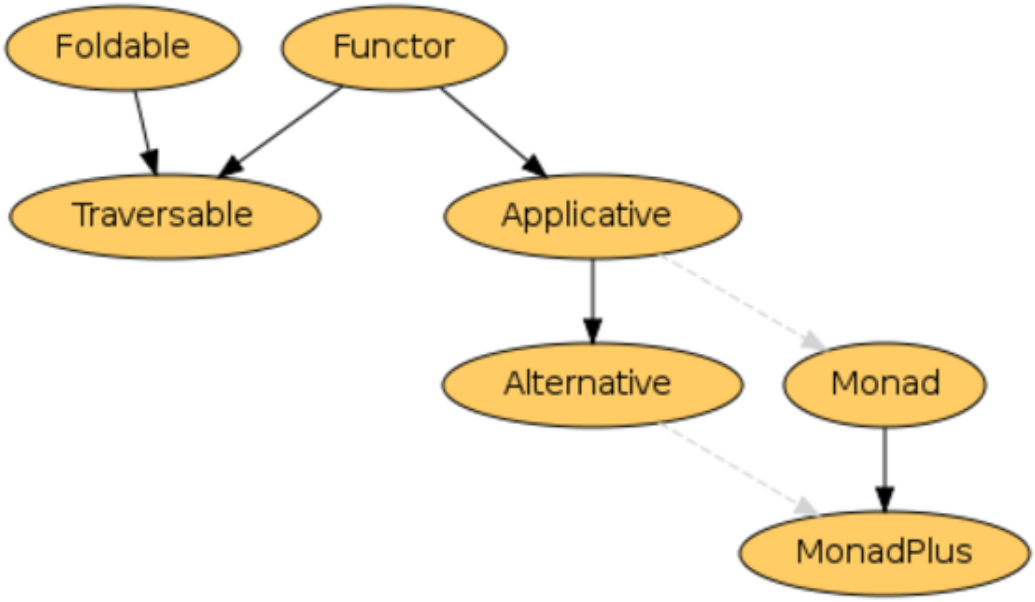

Figure 1.1: Functor Hierarchy [**?**]

In our case consider the following example,

```
-- The Prolog Syntax
type Atom = String
data VariableName = VariableName Int String  deriving (Show,Eq,Ord)
data FlatTerm a =
                  Struct Atom [a]
         |        Var VariableName
         |        Wildcard
         |        Cut Int deriving (Show,Eq,Ord)
```

A `FlatTerm` can be of infinite depth which due to the reason stated above cannot be accounted for during application function. The resulting type signature would be of the form,

```
FlatTerm (FlatTerm (FlatTerm (FlatTerm (.....))))
```

Enter the `Fix` same as the function as a data type. The above would be simply reduced to,

```
Fix FlatTerm
```

resulting in the PROLOG Data Type

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

1.4 Dr. Casperson's Explanation

A recursive data type in HASKELL is where one value of some type contains values of that type, which in turn contain more values of the same type and so on. Consider the following example.

```
data Tree = Leaf Int | Node Int (Tree) (Tree)
```

A sample `Tree` would be,

```
(Node 0 (Leaf 1) (Node 2 (Leaf 3) (Leaf 4)))
```

The above structure can be infinitely deep since HASKELL is a *lazy* programming language. But working with an infinitely deep / nested structure is not possible and will result in a *occurs check* error. This is because writing a type signature for a function to deal with such a parameter is not possible. One option would be to *flatten* the data type by the introduction of a type variable. Consider the following,

```
data FlatTree a = Leaf Int | Node Int a a
```

A sample `FlatTerm` would be similar to `Tree`.

The `FlatTree` is recursive but does not reference itself. But it too can be infinitely deep and hence writing a function to work on the structure is not possible.

1.5 The other fix

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.

1.6 The Fix we use

Fix-point type allows to define generic recursion schemes [**?**].

```
1  Fix f = f (Fix f)
```