Chapter 1

Work Completed

1.1 What is this chapter about

1.2 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 [**?**].

1.3 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

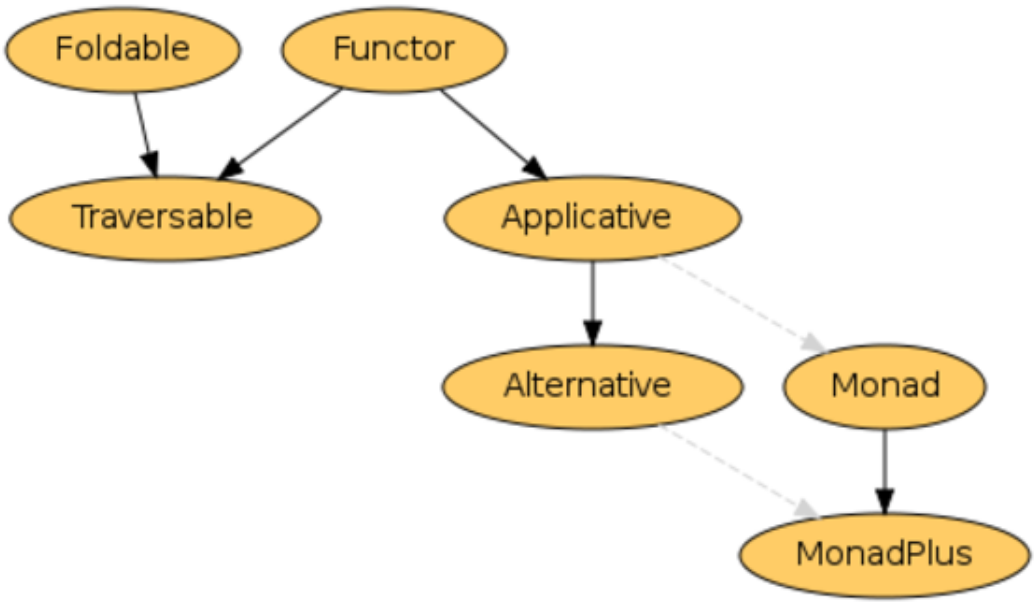

Figure 1.1: Functor Hierarchy [**?**]

1.4 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.

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)
```

1.5 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.

1.6 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.

1.7 The Fix we use

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

What is Algebra Naively speaking algebra gives us the ability to perform calculations with numbers and symbols.

What can algebra do The ability to form and evaluate expressions.

How to generate expressions Using grammars, for example

```
1 data Expr = Const Int
2           | Add Expr Expr
3           | Mul Expr Expr
```

How to uncover primitives from a recursive type Make it non-recursive by defining a type function, otherwise known as type constructor,

```
1 ExprF a = Const Int
2         | Add a a
3         | Mul a a
```

How to create a nested structure from the above The fractally recursive structure of Expr can be generated by repeatedly applying ExprF to itself.

```
(ExprF (ExprF (ExprF a)))
```

How to generate really deep expressions Keep on applying

```
ExprF
```

Is there a better way 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 ExprF. It's like adding one to infinity: you get back infinity.

How do that in HASKELL In HASKELL, we can express the fix point of a type constructor f as a type:

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

With that, we can redefine Expr as a fixed point of ExprF:

```
type Expr = Fix ExprF
```

Any other benefits Writing functions is simpler. You can have the terms of all depths encapsulated under the same type, i.e.

```
Fix ExprF
```

So rather than writing separate functions for,

```
(ExprF a)

(ExprF (ExprF a))

(ExprF (ExprF (ExprF a)))

(ExprF (ExprF (ExprF ....)))
```

We write a function from,

```
func :: Fix ExprF -> Fix ExprF
```

1.8 Opening up or Extending language Explanation using Box Analogy

This section will describe what it means to **"open up or extend a language"**.

1. Let us start with a sample language with a recursive abstract syntax,

```
type Atom                  = String

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

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

   The above language represent a stripped down version of PROLOG from [**?**]. The pool of the expressions that can be generated from *Term* are restricted to the constructors,

```
Struct "hello" [Struct "a" []]          -- hello(a).
Var (VariableName 125 "X")          -- X = 125.
Wildcard          -- _.
Cut 0         -- !.
```

   It does not allow the ability to have a **"typed"** *Term*, for example a *Term* of type *int or string* and so on.

2. So we **flatten** the language by introducing a type variable,

```
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)
```

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 [**?**],

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

Simply speaking the *a* can be changed.

3. It gives the language the capability to be expanded. Adding some functionality to the original language could be done in a no.of ways

   (a) 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.

   (b) 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.

4. If we have a grammar that support an expressions like,

   $x \cdot y + x \cdot z$

   Once the language is 'functorized' one can add quantifiers and logic to the language to do something like,

$$\forall x \forall y \forall z \qquad x \cdot y + x \cdot z \tag{1.1}$$

$$= x \cdot (y + z) \tag{1.2}$$

5. Multiple modifications

6. As is with the original language it can be wrapped with multiple other data structures,

```
Just (Strcut ......) -- A Maybe Term
[Cut 0]                              -- A List of Terms
```

   and so on. But the core expression can only be of type *Term*.

   Whereas a *FlatTerm* exspression can not only have an outer wrapper but also its type is 'open'.

1.9 Chapter Recap