1 Prototype 1

This chapter looks into solving the issue of conflicting type systems of the languages in question. HASKELL is a strong statically typed language requiring type signature for programming constructs at compile time while PROLOG is strong dynamically typed which lets through untyped programs. 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.

1.1 Creating a data type

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

Transitional paragraph hello

To start with, replicating the single type "term" in PROLOG one must consider the distinct constructs it can be associated to such as complex structures (for example, predicated clauses etc.), don't cares, cuts, variables and so on.

```
1 --david-0.2.0.2
2
3 data VariableName = VariableName Int String
4       deriving (Eq, Data, Typeable, Ord)
```

```
5
6  data Atom          = Atom      !String
7                     | Operator  !String
8        deriving (Eq, Ord, Data, Typeable)
9
10 data Term = Struct Atom [Term]
11           | Var VariableName
12           | Wildcard
13           | PString   !String
14           | PInteger  !Integer
15           | PFloat    !Double
16           | Flat [FlatItem]
17           | Cut Int
18       deriving (Eq, Data, Typeable)
19
20 data Clause = Clause { lhs :: Term, rhs_ :: [Goal] }
21             | ClauseFn { lhs :: Term, fn :: [Term] -> [Goal] }
22       deriving (Data, Typeable)
23
24 type Program = [Sentence]
25
26 type Body    = [Goal]
27
28 data Sentence = Query   Body
29               | Command Body
30               | C Clause
31       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 data type is recursive as seen in the constructor,

```
Struct Atom [Term]
```

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 [**?**]. Consider the following, a reduced version of the above data type,

```
1  type Atom          = String
2
3  data VariableName = VariableName Int String
4        deriving (Eq, Data, Typeable, Ord)
5
6  data Term = Struct Atom [Term]
```

```
            | Var VariableName
            | Wildcard -- Don't cares
            | Cut Int
        deriving (Eq, Data, Typeable)
```

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

```
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". Simply speaking we can have something like

```
FlatTerm Bool
```

and a generic fuinction like,

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

1.2 Working with the language

Creating instances,

```
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 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
```

```
20          zipMatch Wildcard _ = Just Wildcard
21          zipMatch _ Wildcard = Just Wildcard
22          zipMatch (Cut i1) (Cut i2) = if (i1 == i2)
23                  then Just (Cut i1)
24                  else Nothing
25
26  instance Applicative (FlatTerm) where
27          pure x = Struct "" [x]
28          _ <*> Wildcard        =         Wildcard
29          _ <*> (Cut i)         =         Cut i
30          _ <*> (Var v)        =        (Var v)
31          (Struct a fs) <*> (Struct b xs) = Struct (a ++ b) [f x | f <- f
```

After flattening do fixing,
Opening up the language somehow so as to accommodate your own variables .

1.3 Black box