Functors Applicatives

d.morgan1

July 2022

1 Functors

Functors increase the level of generality in Haskell. Think of them as functions over a range of parameterised types such as lists, trees.

Functors have both an infix notation where `(<$>) = fmap`, `e.g. fmap (>1) [1,2,3]` and a prefix notation where `fmap (g.h) = fmap g . fmap h`, `e.g. (>1) <$> (Just 1)`

Functor Laws

1. `fmap id = id`
2. `fmap (g.h) = fmap g . fmap h`

Application of Composition (f.g) x, f.g *x*, *f* g x, f *g* x

Structural Induction Suppose S is some recursively defined structure (e.g. [a] or Tree a) that has substructure (e.g. sublist, subtree) and there is partial ordere.g. length or number of nodes. Structural induction implies if...

1. P is true for all minimum structures, and (Base Case)
2. P(x') is true for x' any immediate substructure of x (Induction)

2 Applicatives

Functors can map a function over each element in a structure. Suppose we wish to generalise the idea to allow functions with any number of arguments to be mapped.

Applicative Laws

1. `pure id <*> x = x` *mapping identity has no effect*
2. `pure (g x) = pure g <*> pure x` *pure distribution*
3. `x <*> pure y = pure (-> g y) <*> x` *effectful function disregards evaluation order*
4. `x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z` *associativity*

Corollary `g <$> x = pure g <*> x`

Effectful Programming Applicative functors abstract the idea of applying pure functions to effectful arguments, where the effects depend on the underlying functor. Effects may be: possibility of failure, having many ways to succeed or I/O actions.

Applicative Style

Note that: `fmap g x = pure g <*> x`, so this means

```
pure g <*> x1 <*> x2 <*> ... <*> xn
```

is same as

```
g <$> x1 <*> x2 <*> ... <*> xn
```