Some operators are "nice"

Some operators have nice properties like having unit values e.g. $0 + a = a = a + 0$

We can code a simplifier for these as follows:

uopSimp cons u (Val v) e — v == u = e
uopSimp cons u e (Val v) — v == u = e
uopSimp cons u e1 e2 = cons e1 e2

We can deduce that cons is:

cons :: Expr -¿ Expr -¿ Expr

So we use Add and Mul directly:

simp (Add e1 e2) = uopSimp Add 0.0 e1 e2
simp (Mul e1 e2) = uopSimp Mul 1.0 e1 e2

The data constructors of Expr are in fact functions, whose types are as follows:

```
Val :: Double -> Expr
Var :: Id -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
Sub :: Expr -> Expr -> Expr
Dvd :: Expr -> Expr -> Expr
Def :: Id -> Expr -> Expr -> Expr
```

So, cons has to have type Expr -¿ Expr -¿ Expr, which is why Add and Mul are suitable arguments to pass into uopSimp

Given declaration:

data MyType = ... — MyCons T1 T2 ... Tn — ....

then data constructor MyCons is a function of type:

MyCons :: T1 -¿ T2 -¿ ... -¿ Tn -¿ MyType

Partial applications of MyCons are also valid

(MyCons x1 x2) :: T3 -¿ ... -¿ Tn -¿ MyType

Data constructors are the only functions that can occur in patterns.

Abstractions

A lot of the higher-order functions in the Prelude are examples of abstraction of common programming shapes encountered in functional programs (e.g. map and folds).

Turning Common Shapes Into Functions

Remember these?

sum [ ] = 0 sum (n:ns) = n + sum ns
length [ ] = 0 length (_:xs) = 1 + length xs
prod [ ] = 0 prod (n:ns) = n * prod ns

They have a common pattern which is typically referred to as folding. Can we abstract this?

Commonalities

- They all have the empty list as a base case

- They all have a non-empty list as the recursive case
- The base case returns a fixed "unit" value, which we will call u

  <abs-fold> [ ] = u

  <abs-fold> (a:as) = ... <abs-fold> as