Classes Based on Other Classes ”=¿” is context, stating that the second class depends on the first

Polymorphic Type Classes

How might we define an Eq instance for lists?

- For [Bool] instance Eq [Bool] where

  [ ] == [ ] = True

  (b1:bs1) == (b2:bs2) = b1 == b2 && bs1 == bs2

  _== _= False

- Cant we do this polymorphically?

- We can!

  instance (Eq a) =¿ Eq [a] where

  [ ] == [ ] = True

  (x1:xs1) == (x2:xxs2) = x1 ==x2 && xs1 == xs2

  _== _= False

We can define equality on [a] provided we have equality set up for a

Here we are defining equlity for a type constructor ([ ] for lists) applied to a type a.

How Haskell Handles a Class Name/Operator

Consider the following polymorphic function:

threewayEq :: Eq a =¿ [a] -¿ [a] -¿ [a] -¿ Bool
threewayEq xs ys ys = xs == ys && ys == zs

The code for threewayEq is polymorphic so at compilation time, we cant say which implementation of == is used. Bu tin the general case, a function like this gets an additional aprameter, a ”class dictionary”.

This is used at run-time to look up the implementation of ==, once a concrete instance for type a is known.

Type-Constructor Classes

Consider the class declaration for Functor

class Functor f where
fmap :: (a -¿ b) -¿ f a -¿ f b

The idea here is that fmap f applied to a value of type f a will apply f to all occurences of a within that value.

Here, we are asssociating a class with a type-constructor f

Type-Constructor Examples

The Maybe type-constructor

data Maybe a = Nothing — Just a

Instances of Functor

Maybe as a Functor

instance Functor Maybe where
fmap f Nothing = Nothing

fmap f (Just a) = Just (f x)

Functor Instance for Maybe

class Functor f where

fmap :: (a -¿ b) -¿ f a -¿ f b

instance Functor Maybe where

fmap (f :: a -¿ b) (Nothing :: Maybe a) = Nothing :: Maybe b

fmap f (Just (x :: a) :: Maybe a) = Just (f x :: b) :: Maybe b

An Example: Expressions

We are going to write functions that manipulate expressions in a variety of ways.

```
data Expr
  = Val Double
  | Add Expr Expr
  | Mul Expr Expr
  | Sub Expr Expr
  | Dvd Expr Expr
  deriving Show -- makes it possible to see values (DEMO!)
```

$(10 + 5) * 90$ becomes Mul (Add (Val 10) (Val 5)) (Val 90)

$10 + (5 * 90)$ becomes Add (Val 10) (Mul (Val 5) (Val 90))

We can write a function to calculate the result of these expressions:

eval :: Expr -¿ Double

eval (Val x) = x

eval (Add x y) = eval x + eval y

eval (Mul x y) = ... –similar to above. do similarly for sub and dvd

> eval Add (Val 10) (Mul (Val 5) (Val 90))

460.0

We can also write a function to simplify expressions

simp :: Expr -¿ Expr

simp (Val x) = (Val x)

We can use pattern matching in let-expressions!

simp (Add e1 e2) = let (Val x) = simp e1

(val y) = simp e2

in Val (x+y)

simp (Dvd x y) = ...

Adding Variables to Expressions

```
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Dvd Expr Expr
          | Var Id
     deriving Show

type Id = String
```

We can also see that our simplification will return either a (Val value) or a (Var var).

simp (Var v) = (Var v

This complicates our simp somewhat because we can no longer assume that simp always returns a Val.

Simplification for Operators

We now have to pattern-match on the results of recursive calls to simp

```
simp (Add e1 e2)
 = let e1' = simp e1
       e2' = simp e2


     in case (e1',e2') of
       (Val 0.0,e)  ->  e
       (e,Val 0.0)  ->  e
       _            ->  Add e1' e2'
  simp (Mul e1 e2) = ... -- similar (and Sub,Dvd)
  simp e = e -- catches all remaining cases (Val, Var)
```

Evaluating Exprs with Variables

We can't fully evaluate our extended expression language without some way of knowing what values any of the variables (Var) have. We can imagine eval should have a signature like this:

eval :: Dict Id Double -¿ Expr -¿ Double

It now has a new (first) argument, a Dict that associates Double (data values) with Id (key values).

How to model a lookup dictionary?

A dictionary maps keys to data values. An obvious approach is to use a list of key/data pairs:

type Dict k d = [(k,d)]

Defining a link between key and data is simply cons-ing such a pair onto the start of the list:

define :: Dict k d -¿ k -¿ d -¿ Dict k d

define d s v - (s,v):d

Lookup simply searches along the list:

find :: Eq k =¿ Dict k d -¿ k -¿ Maybe d
find [ ] _= Nothing
find ((s,v) : ds) name — name == s = Just v
—otherwise = find ds name

We need to handle the case when the key is not present, using the Maybe type

Expressions with Local Variable Declarations

```
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Dvd Expr Expr
          | Var Id
          | Def Id Expr Expr
```

Def x e1 e2 means: x is in scope in e2, bu tno in e1; compute value of e1, and assign value to x; then evaluate e2 as the overall result

Abstracting Functions

Consider the following function definitions:
f a b = sqr a + sqrt b
g x y = sqrt x * sqr y
They all have the same general form:
fname arg1 arg2 = someF arg1 'someOp' anotherF arg2
We can abstract this by adding parameters to represent the bits of the general form.
absF someF anotherF someOp arg1 arg2 = omeF arg1 'someOp' anotherF arg2
Now f and g can be defined using absF:
f a b = absF sqr sqrt (+) a b
g x y = absF sqrt sqr (*) x y