Contents

Haskell vs Prolog

Lists

Mathematically we might write lists as items seperated by commas, enclosed in angle-brackets

$\sigma_0 =<>, \sigma_1 =< 1 >, \sigma_2 =< 1, 2 >, \sigma_3 =< 1, 2, 3 >$

Haskell:

```
s0=[]
s1=1:[]      -- or [1]
s2=1:2:[]    -- or [1, 2]
s3=1:2:3:[] -- or [1, 2, 3]
```

Prolog:

```
s0 = []
s1 = [1]
s2 = [1, 2]
```

Patterns

Haskell:

```
[]
(x:xs)
(x:y:xs)
```

Prolog:

```
[]
[X|Xs]
[X, Y|Xs]
```

Extending Expr Further

We can augment the expression type to allow expressions with local variable declerations:

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

The intended meaning of `Def x e1 e2` is `x` is in the scope of `e2` but not in `e1`; compute value of `e1` and assign value to `x`; then evaluate `e2` as overall result

Def Example

A simple expression in this form could look like this:

```
testExpr = Def "a" (Mul (Val 2) (Val 3)) (
             Def "b" (Sub (Val 8) (Val 1)) (
               Sub (Mul (VEar "a" ) (Var "b"))
                 (Val 1)))
```

A nice way to print this out might be:

```
let a = 2 * 3
  in let b = 8 - 1
    in (a * b) - 1
```

Dictionary Evaluation

For the non-identifier parts of the expressions we simply pass the dictionary around, but otherwise ignore it

```
eval :: Dictionary Id Float -> Expr -> Float
eval d (Val v) = v
eval d (Add e1 e2) = (eval d e1) + (eval d e2)
eval d (Mul e1 e2) = (eval d e1) * (eval d e2)
eval d (Sub e1 e2) = (eval d e1) - (eval d e2)
eval d (Dvd e1 e2) = (eval d e1) / (eval d e2)
```

Given a variable, we simply look it up

```
eval d (Var n) = fromJust (find d n)
fromJust (Just x) = x
```

Given a `Def` we

1. Evaluate the first expression in the given dictionary
2. Add a binding from the defined variable to the resulting value and then
3. Evaluate the second expression with the updated dictionary

```
eval d (Def x e1 e2) = eval (define f x (eval d e1) ) e2
```

Eval

Taking Stock

- We have introduced the datatype `Expr` for expressions
- We have a lookup table that associates datum values with keys
- We can simplify (`simp`) the expressions
- We can evaluate (`eval`) the expressions
- We can print (`print`) out the expressions
- Lets review in detail what we have

The Datatyle

```
type Id = String

data Expr
  = Val Double
  | Var Id
  | Add Expr Expr
  | Mul Expr Expr
  | Sub Expr Expr
  | Dvd Expr Expr
  | Def Id Expr Expr
  deriving (Eq, Show)
```

Evaluation

```
type Dict = Tree Id Double

eval :: Dict -> Expr -> Double
eval _ (Val x) = x
eval d (Var i) = fromJust (find d i)
eval d (Add x y) = eval d x + eval d y
```

```haskell
eval d (Mul x y) = eval d x * eval d y
eval d (Sub x y) = eval d x - eval d y
eval d (Dvd x y) = eval d x / eval d y
eval d (Def x e1 e2) = eval (define x (eval d e1) e) e2
```

Simplification

```haskell
simp :: Expr -> Expr

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)
  = let e1' = simp e1
        e2' = simp e2
    in case (e1',e2') of
        (Val 1.0,e)  ->  e
        (e,Val 1.0)  ->  e
        _            ->  Mul e1' e2'

simp (Sub e1 e2)
  = let e1' = simp e1
    e2' = simp e2
  in case (e1',e2') of
    (e,Val 0.0)  ->  e
    _            ->  Sub e1' e2'

simp (Dvd e1 e2)
  = let e1' = simp e1
    e2' = simp e2
  in case (e1',e2') of
    (e,Val 1.0)  ->  e
    _             ->  Dvd e1' e2'

simp (Def x e1 e2) = (Def x (simp e1) (simp e2))
simp e = e
```

Lots of similar code - “boilerplate”!

Fancy Printing

- Only use parentheses when they are required by operator precedence rules
- Higher precedence numbers mean tighter binding
- Start by rendering an expression at the top-level at precedence level zero

```
pPrint e = pP 0 e

pP _ (Val x) = show x

pP _ (Var v) = v

pP p (Add e1 e2) -- precedence in Haskell is 6
  | p > 6 = "("++ppadd++")"
  | otherwise ppadd
  where ppadd = pP 6 e1 ++ "+" ++ pP 6 e2

pP p (Mul e1 e2) -- precedence in Haskell is 7
  | p > 7 = "("++ppmul++")"
  | otherwise ppmull
  where ppmul = pP 7 el ++ "*" ++ pP 7 e2
```

- `Sub` is very similar to `Add`
- `Dvd` is very similar to `Dvd`

Issues

- We need proper error handling
- We need to reduce the amount of boilerplate
  - This is important if we hope to extend the expression type in any way
- Three mechanisms are available to help
  - The type system - we can define types that help with error handling
  - Abstraction - we can capture common boilerplate patterns as functions
  - Classes - we can capture common boilerplate control patterns as classes

Using Maybe to handle errors

Remember the Maybe type

```haskell
data Maybe t = Nothing | Just t
```

We can revise our `eval` function to return a value of type `Maybe Double`, using `Nothing` to signal an error:

```haskell
eval :: Dict -> Expr -> Maybe Double
eval _ (Val x) = Just x
eval d (Var i) = find d i -- returns a Maybe type anyway!
```

Evalulating using Maybe

```haskell
eval d (Add x y) = Just ( (eval d x) + (eval d y) )
```

Won't work - `eval` no longer returns a `Double`, but a `Maybe Double`!

We have to pattern-match against the recursive `eval` outcomes to see what to do next

```haskell
eval d (Add x y)
  = let r = eval d x
        s = eval d y
    in case (r, s) of
      (Just m, Just n) -> Just (m+n)
      _                -> Nothing

eval d (Mul x y)
  = let r = eval d x
        s = eval d y
    in case (r, s) of
      (Just m, Just n) -> Just (m*n)
      _                -> Nothing

eval d (Sub x y)
  = let r = eval d x
        s = eval d y
    in case (r, s) of
      (Just m, Just n) -> Just (m-n)
      _                -> Nothing

eval d (Dvd x y)
  = let r = eval d x
        s = eval d y
    in case (r, s) of
      (Just m, Just n) -> if n==0.0 then Nothing else Just (m/n)
      _                -> Nothing
```

More boilerplate

Evaluating Def

```
eval d (Def x e1 e2)
  = let r = eval d e1
    in case (r, s) of
      Nothing -> Nothing
      Just v1 -> eval (define x v1 d) e2
```

Even more boilerplate

Error handling seems expensive.

This is why more languages support exceptions.

Closing Observations

- We can add explicit error handling using `Maybe` (or `Either`)
- Exceptions are available, but only in an `IO` context
- However we can still do a lot better, with higher order abstractions and classes

Turning Expressions into Functions

Consider the following expressions:

```
a * b + 2 - c
```

There are at least four ways we can turn this into a function of one numeric argument

```
f a where f x = x * b + 2 - c
f b where f x = a * x + 2 - c
f c where f x = a * b + 2 - x
f 2 where f x = a * b + x - c
```

This process of converting expressions into functions is called *abstraction*

Abstracting Functions

Consider the following function definitions:

```
f a b = sqr a + sqrt b
g x y = sqrt x * sqr y
h p q = log p - abs q
```

They all have the same general form:

```
fname arg1 arg2 = someF aqr1 `someOp` anotherF arg2
```

We can abstract this adding parameters to represent the "bits" of the general form:

```
absF someF anotherF someOp arg1 arg2 = someF arg1 `someOp` anotherF arg2
```

Now `f`, `g` and `h` can be defined using `absF`

```
f a b = absF sqr sqrt (+) a b
g x y = absF sqrt sqr (*) x y
h = absF log abs (-) -- we can use partial application
```

The "shape" of eval using Maybe

A typical binary operation case in `eval` looks like

```
eval d (Sub x y)
  = let r = eval d x; s = eval d y
    in case (r, s) of
      (Just m, Just n) -> Just (m-n)
      _                -> Nothing
```

We just need to process the two sub-expressions, with a binary operator for the result, so we come up with

```
evalOp d op x y
  = let r = eval d x; s = eval d y
    in case (r, s) of
      (Just m, Just n) -> Just (m `op` n)
      _                -> Nothing
```

This works for `Add`, `Mul`, and `Sub` (but not `Dvd`)

Revised eval

The following cases get simplified

```
eval d (Add x y) = evalOp d (+) x y
eval d (Mul x y) = evalOp d (*) x y
eval d (Sub x y) = evalOp d (-) x y
```

We can't do `Dvd` because it will need to return `Nothing` if `y` evaluates to 0.

At least those operators that cannot raise errors are now easy to code.

Simplifying simp

We have code as follows (let's use `Sub` again)

```
simp (Sub e1 e2)
  = let e1' = simp e1
        e2' = simp e2
    in case (e1', e2') of
      (e, Val 0.0) -> e
      _            -> Sub e1' e2'
```

We can at least isolate the simplifications out

```
simpOp (opsimp e1 e2)
  = let e1' = simp e1
        e2' = simp e2
    in opsimp e1' e2'
```

`simp` itself is simpler

```
simp (Add e1 e2) = simpOp addSimp e1 e2
simp (Mul e1 e2) = simpOp mulSimp e1 e2
simp (Sub e1 e2) = simpOp subSimp e1 e2
simp (Dvd e1 e2) = simpOp dvdSimp e1 e2
```

Each operator simplifier has its own case-analysis, e.g.

```
mulSimp (Val 1.0) e = e
mulSimp e (Val 1.0) = e
mulSimp e1 e2       = Mul e1 e2
```

Still boilerplate, but perhaps it is clearer this way (no explicit use of case)

Some operators are "nice"

- Some operators have nice properties, like having unit values, e.g. `0+a=a=a+0` and `1*a=a=a*1`
- 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
```

Usage:

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

Data Constructors are Functions

The data constructors of `Expr`, are in fact functions, whoes 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` on the previous slide needs to have the 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 | ...
```

the 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

Abstraction: Summary

- Abstraction is the process of turning expressions into functions
- If done intelligently, it greatly increases code re-use and reduces boilerplate
- We saw is applied to `eval` and `simp`
- 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`)