1 Class 13

Adding Variables to Expressions

Now let's extend our expression datatype to include variables.

First we extend the expression language:

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

type Id = String
```

Simplification again

We look at simplication first this time.

How we handle `Val` does not change, but what about `Var`?

```
simp (Var v) = (Var v)
```

So, now, at best, `simp` will return either a `(Val value)` or a `(Var var)`.

We can no longer assume that `simp` always returns a `Val`!

This complicates simplification somewhat.

Simplification for Operators

We now have to pattern-match on the results of recursive calls to `simp`, in order to see what to do.

```
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)
```

Since `simp` returns `Val` and `Var` unchanged, we can use a catch-all pattern at the end to handle them.

Evaluating `Expr`s with Variables

Remember our extended expression language:

```
type Id = String
data Expr = Val Double
          | Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
```

```
     | Dvd Expr Expr
     | Var Id
deriving Show
```

We can't fully evaluate these without some way of knowing what values any of the variables (`Var`) have.

We can imagine that `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` (datum values) with `Id` (key values).

How to model a lookup Dictionary?

A *Dictionary* maps keys to datum values

- An obvious approach is to use a list of key/datum pairs:

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

- Defining a link between key and datum 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. This is the role of the `Maybe` type.

Maybe (`Prelude`)

```
data Maybe a
  =  Nothing
  |  Just a
  deriving (Eq, Ord, Read, Show)

maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing   =  n
maybe n f (Just x) =  f x
```

Maybe (Data.Maybe)

```
import Data.Maybe -- need to explicitly import this

isJust                 :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

isNothing              :: Maybe a -> Bool
isNothing              =  not . isJust

fromJust               :: Maybe a -> a
fromJust (Just a)      =  a
fromJust Nothing       =  error "Maybe.fromJust: Nothing"

fromMaybe              :: a -> Maybe a -> a
fromMaybe d Nothing    =  d
fromMaybe d (Just a)   =  a
```

Dict at work

Building a simple Dict that maps key "speed" to datum 20.0.

```
> define [] "speed" 20.0
[ ("speed", 20.0) ]
> find (define [] "speed" 20.0) "speed"
Just 20.0
> find [] "speed"
Nothing
```

Extending the evaluator

```
eval :: Dict Id Double -> Expr -> Double
eval _ (Val x) = x
eval d (Var i) = fromJust (find d i)
eval d (Add e1 e2) = eval d e1 + eval d e2
-- similar for Add, Mul, Dvd
fromJust (Just a) = a
```

We are back to simpler code (no need for `case ... of ...`)

`Expr` Pretty-Printing

We can write something to print the expression in a more "friendly" infix style:

```
iprint :: Expr -> String
iprint (Val x) = show x
iprint (Var x) = x
iprint (Dvd x y) = "("++(iprint x)++"/"++iprint y++")"
-- similar for Add, Mul, Sub
```

There are many ways in which this could be made much prettier.

Extending `Expr` Further

We can augment the expression type to allow 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
```

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

Def example

A sample 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 (Var "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
```

Dict-based Evaluation (I)

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

```
eval :: Dict Id Double -> Expr -> Double
eval d (Val v) = v
eval d (Add e1 e2)  =  (eval d e1) + (eval d e2)
-- others similarly
```

Dict-based Evaluation (II)

Given a variable, we simply look it up:

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

fromJust (Just x) = x
```

Dict-based Evaluation (III)

Given a `Def`, we

1. evaluate the first expression in the given Dict;
2. add a binding from the defined variable to the resulting value, and then
3. evaluate the second expression with the updated Dict:

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

`Expr`: taking stock

- We have introduced a datatype `Expr` for expressions
- We have a lookup table that associates datum values with keys
- We can simplify (`simp`) the expressions (to some degree)
- We can evaluate (`eval`) the expressions (to some degree)
- We can print (`iprint`) out the expressions in a (reasonably) nice manner

`Expr`: Issues (1)

- We extended this before — perhaps we might want to do this again?
- What happens if a variable is not in the Dict?
- What happens if we divide by zero?
- A lot of very similar looking code ("boilerplate") !

`Expr`: Issues (2)

- 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:

```
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:

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

Now lets look at some other cases.

Evaluating `Mul` using `Maybe`

```
eval d (Mul 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:

```
eval d (Mul x y)
 = case (eval d x,eval d y) of
     (Just m,Just n)  ->  Just (m*n)
     _                ->  Nothing
```

Evaluating `Dvd`

Here we can now properly handle division by zero!

```
eval d (Dvd x y)
 = case (eval d x, eval d y) 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)
 = case eval d e1 of
     Nothing  ->  Nothing
     Just v1  ->  eval (define d x v1) e2
```

More boilerplate ! Error handling seems expensive! This is why most languages support exceptions.

Closing Observations

- We can add explicit error handling using `Maybe` (or `Either`).
- Exceptions *are* available, but only in an `IO` context[1]
- However we can still do a lot better, with higher-order abstractions and classes.

2 Class 14

Arbitrarily Nested Lists

Consider the following "list":

`[1,[2,3],[[4]]]`

Is this possible in Haskell?

Not as written—a Haskell list has to contain elements of the same type.

This contains `1 :: Int`, `[2,3] :: [Int]`, and `[[4]] :: [[Int]]`.

Can we have a Haskell datastructure that supports such nested lists?

[1]??? - we'll get to this...

Nested Integer Lists (I)

We shall try to define a nested `Int` list type (`NLI`)

```
data NLI
```

We need a representation for an empty list: `X`

```
 = X  -- empty version of NLI
```

We can easily define how to add an `Int` onto the front of this list:

```
 | AC Int NLI   -- "atomic" cons: Int with NLI give an NLI
```

At this point all we have is (yet another) definition of lists of `Int`.

How do we get the nesting?

Nested Integer Lists (II)

How do we get the nesting? Simple!

Just add a new data constructor to represent "consing" a nested list to the front of a nested list:

```
 | NC NLI NLI  -- "nested" cons: NLI with NLI give an NLI
```

That's It!

Note that we are not using `NC` to represent concatenation.

We interpret the first `NLI` argument as a new single entry at the start of the second `NLI` argument.

Nested Integer Lists Example

We have:

```
data NLI
 = X
 | AC Int NLI
 | NC NLI NLI
```

How do we produce `[1,[2,3],[[4]]]`?

```
AC 1 (NC (AC 2 (AC 3 X)) (NC (NC (AC 4 X) X) X))
```

COMMENTARY: We can build example as follows, starting from the back.

`[4]` — `AC 4 X`

`[[4]]` — `NC (AC 4 X) X`

`[3]` — `AC 3 X`

`[2,3]` — `AC 2 (AC 3 X)`

`[ [2,3], [[4]] ]` — `NC (AC 2 (AC 3 X)) (NC (AC 4 X) X)`

`[ 1, [2,3], [[4]] ]`
— `AC 1 (NC (AC 2 (AC 3 X)) (NC (AC 4 X) X))`

Polymorphic Nested Lists

We can have a polymorphic version too.

Just replace `NLI` by `NL a`.

```
data NL a
 = X
 | AC a (NL a)
 | NC (NL a) (NL a)
```

An important point to note:

While we have nested lists here, once we get to adding any atomic (non-list) elements, they all have to be of the same type.

Code for this, along with a pretty printer, are in `NestedLists.hs`.

2.1 Abstraction (1)

Turning Expressions into Functions

Consider the following expression:

```
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 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
                   = 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` (why not?)

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.

2.2 Abstraction (2)

Simplifying `simp` (I)

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't abstract to the same degree as for `eval`, because there is a lot of irregularity in the simplifications.

Simplifying `simp` (II)

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
```

Simplifying `simp` (III)

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 boilerpate, 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
```

  What is `cons` here?

- 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 (I)

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` on the previous slide needs to have type `Expr -> Expr -> Expr`, which is why `Add` and `Mul` are suitable arguments to pass into `uopSimp`.

Data Constructors are Functions (II)

- 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.

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 it 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 `fold`s).