Informatics 1A

Functional Programming Lectures 10–11

Expression Trees as Algebraic Data Types

Don Sannella

University of Edinburgh

Part I

Arithmetic Expressions

Arithmetic Expressions

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving Eq
```

```
e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)
```

Arithmetic Expressions

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving Eq


evalExp :: Exp -> Int
evalExp (Lit n)    = n
evalExp (Add e f) = evalExp e + evalExp f
evalExp (Mul e f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)    = show n
showExp (Add e f) = par (showExp e ++ "+" ++ showExp f)
showExp (Mul e f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"
```

Arithmetic Expressions

```
e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)
```

```
> showExp e0
"(2+(3*3))"
> evalExp e0
11
> showExp e1
"((2+3)*3)"
> evalExp e1
15
```

Arithmetic Expressions with Infix Constructors

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving Eq

e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)
```

```
data Exp = Lit Int
         | Exp `Add` Exp
         | Exp `Mul` Exp
         deriving Eq

e0, e1 :: Exp
e0 = Lit 2 `Add` (Lit 3 `Mul` Lit 3)
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3
```

Arithmetic Expressions with Infix Constructors

```
data Exp = Lit Int
         | Exp `Add` Exp
         | Exp `Mul` Exp
         deriving Eq

evalExp :: Exp -> Int
evalExp (Lit n)     = n
evalExp (e `Add` f) = evalExp e + evalExp f
evalExp (e `Mul` f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)     = show n
showExp (e `Add` f) = par (showExp e ++ "+" ++ showExp f)
showExp (e `Mul` f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"
```

Arithmetic Expressions with Infix Constructors

```
e0, e1 :: Exp
e0 = Lit 2 `Add` (Lit 3 `Mul` Lit 3)
e1 = (Lit 2 `Add` Lit 3) `Mul` Lit 3
```

```
> showExp e0
"(2+(3*3))"
> evalExp e0
11
> showExp e1
"((2+3)*3)"
> evalExp e1
15
```

Arithmetic Expressions with Symbolic Constructors

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving Eq

e0, e1 :: Exp
e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)


data Exp = Lit Int
        | Exp :+: Exp
        | Exp :*: Exp
        deriving Eq

e0, e1 :: Exp
e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
e1 = (Lit 2 :+: Lit 3) :*: Lit 3
```

Arithmetic Expressions with Symbolic Constructors

```
data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp
         deriving Eq

evalExp :: Exp -> Int
evalExp (Lit n)   = n
evalExp (e :+: f) = evalExp e + evalExp f
evalExp (e :*: f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)   = show n
showExp (e :+: f) = par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"
```

Arithmetic Expressions with Symbolic Constructors

```
e0, e1 :: Exp
e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
e1 = (Lit 2 :+: Lit 3) :*: Lit 3
```

```
> showExp e0
"(2+(3*3))"
> evalExp e0
11
> showExp e1
"((2+3)*3)"
> evalExp e1
15
```

Part II

Propositions

Propositions

```
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :||: Prop
          | Prop :&&: Prop
          deriving Eq


p0 :: Prop
p0 =  (Var "a" :&&: Not (Var "a"))
```

Showing a Prop

```
showProp :: Prop -> String
showProp (Var x)     = x
showProp F           = "F"
showProp T           = "T"
showProp (Not p)     = par ("not " ++ showProp p)
showProp (p :||: q) = par (showProp p ++ " || " ++ showProp q)
showProp (p :&&: q) = par (showProp p ++ " && " ++ showProp q)

par :: String -> String
par s  =  "(" ++ s ++ ")"
```

Evaluating a Proposition

```
type Valn = Name -> Bool

evalProp :: Valn -> Prop -> Bool
evalProp vn (Var x)     = vn x
evalProp vn F           = False
evalProp vn T           = True
evalProp vn (Not p)     = not (evalProp vn p)
evalProp vn (p :||: q) = evalProp vn p || evalProp vn q
evalProp vn (p :&&: q) = evalProp vn p && evalProp vn q
```

Example

```
p0 :: Prop
p0 =  (Var "a" :&&: Not (Var "a"))

valn :: Valn
valn "a" = True
valn "b" = True
valn "c" = False
valn "d" = True
```

```
> showProp p0
(a && (not a))
> evalProp valn p0
False
```

How evalProp Works

```
evalProp vn (Var x)     = vn x
evalProp vn F           = False
evalProp vn T           = True
evalProp vn (Not p)     = not (evalProp vn p)
evalProp vn (p :||: q) = evalProp vn p || evalProp vn q
evalProp vn (p :&&: q) = evalProp vn p && evalProp vn q

  evalProp valn (Var "a" :&&: Not (Var "a"))
=
  (evalProp valn (Var "a")) && (evalProp valn (Not (Var "a")))
=
  valn "a" && (evalProp valn (Not (Var "a")))
=
  True && (evalProp valn (Not (Var "a")))
= ... =
  True && False
=
  False
```

Another Example

```
p1 :: Prop
p1 = (Var "a" :&&: Var "b")
     :||: (Not (Var "a") :&&: Not (Var "b"))
```

```
> showProp p1
((a && b) || ((not a) && (not b)))
> evalProp valn p1
True
```

Variables that Occur in a Proposition

```
type Names = [Name]

names :: Prop -> Names
names (Var x)     = [x]
names (F)         = []
names (T)         = []
names (Not p)     = names p
names (p :||: q) = nub (names p ++ names q)
names (p :&&: q) = nub (names p ++ names q)


> names p0
["a"]
> names p1
["a","b"]
```

All Possible Valuations

```
empty :: Valn
empty = error "undefined"


extend :: Valn -> Name -> Bool -> Valn
extend vn x b y | x == y    = b
                | otherwise = vn y


valns :: Names -> [Valn]
valns []     = [ empty ]
valns (x:xs) = [ extend vn x b
                 | vn <- valns xs, b <- [True, False] ]
```

All Possible Valuations

```
valns :: Names -> [Valn]
valns []     = [ empty ]
valns (x:xs) = [ extend vn x b
               | vn <- valns xs, b <- [True, False] ]
```

$$\texttt{valns []} = \texttt{[}\{\textit{anything} \mapsto \textit{error}\}\texttt{]}$$

$$\begin{aligned}\texttt{valns ["b"]} = \texttt{[}&\{\texttt{"b"} \mapsto \texttt{False}, \textit{anything else} \mapsto \textit{error}\},\\ &\{\texttt{"b"} \mapsto \texttt{True}, \textit{anything else} \mapsto \textit{error}\}\texttt{]}\end{aligned}$$

$$\begin{aligned}\texttt{valns ["a","b"]} = \texttt{[}&\{\texttt{"a"} \mapsto \texttt{False}, \texttt{"b"} \mapsto \texttt{False}, \textit{anything else} \mapsto \textit{error}\},\\ &\{\texttt{"a"} \mapsto \texttt{False}, \texttt{"b"} \mapsto \texttt{True}, \textit{anything else} \mapsto \textit{error}\},\\ &\{\texttt{"a"} \mapsto \texttt{True}, \texttt{"b"} \mapsto \texttt{False}, \textit{anything else} \mapsto \textit{error}\},\\ &\{\texttt{"a"} \mapsto \texttt{True}, \texttt{"b"} \mapsto \texttt{True}, \textit{anything else} \mapsto \textit{error}\}\texttt{]}\end{aligned}$$

Satisfiable

```
satisfiable :: Prop -> Bool
satisfiable p = or [ evalProp vn p | vn <- valns (names p) ]
```

Another Example

```
p1 :: Prop
p1 = (Var "a" :&&: Var "b")
      :||: (Not (Var "a") :&&: Not (Var "b"))

> names p1
["a","b"]
> valns (names p1) -- can't print in Haskell!!
```

$$[\{\texttt{"a"} \mapsto \texttt{False}, \texttt{"b"} \mapsto \texttt{False}, \textit{anything else} \mapsto \textit{error}\},$$

$$\{\texttt{"a"} \mapsto \texttt{False}, \texttt{"b"} \mapsto \texttt{True}, \textit{anything else} \mapsto \textit{error}\},$$

$$\{\texttt{"a"} \mapsto \texttt{True}, \texttt{"b"} \mapsto \texttt{False}, \textit{anything else} \mapsto \textit{error}\},$$

$$\{\texttt{"a"} \mapsto \texttt{True}, \texttt{"b"} \mapsto \texttt{True}, \textit{anything else} \mapsto \textit{error}\}]$$

```
> [ evalProp vn p1 | vn <- valns (names p1) ]
[True, False, False, True]
> satisfiable p1
True
```

Part III

Optional Values

The Maybe Type

```
data Maybe a = Nothing | Just a
```

Optional argument

```
power :: Maybe Int -> Int -> Int
power Nothing n  = 2 ^ n
power (Just m) n = m ^ n
```

Optional result

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

Using an Optional Result

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)


wrong :: Int -> Int -> Int
wrong n m = divide n m + 3


right :: Int -> Int -> Int
right n m = case divide n m of
              Nothing -> 3
              Just r -> r + 3
```

Part IV

Disjoint Union of Two Types

Either a or b

```
data Either a b = Left a | Right b


mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]


addints :: [Either Int String] -> Int
addints []                = 0
addints (Left n : xs)     = n + addints xs
addints (Right s : xs)    = addints xs


addints' :: [Either Int String] -> Int
addints' xs = sum [n | Left n <- xs]
```

Either a or b

```
data Either a b = Left a | Right b


mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]


addstrs :: [Either Int String] -> String
addstrs []                = ""
addstrs (Left n : xs)     = addstrs xs
addstrs (Right s : xs)    = s ++ addstrs xs


addstrs' :: [Either Int String] -> String
addstrs' xs = concat [s | Right s <- xs]
```

Part V

The Universal Type and Micro-Haskell

The Universal Type and Micro-Haskell

```
data  Univ  =  UBool Bool
            |  UInt Int
            |  UList [Univ]
            |  UFun (Univ -> Univ)

data  Hask  =  HTrue
            |  HFalse
            |  HIf Hask Hask Hask
            |  HLit Int
            |  HEq Hask Hask
            |  HAdd Hask Hask
            |  HVar Name
            |  HLam Name Hask
            |  HApp Hask Hask

type  HEnv  =  [(Name, Univ)]
```

Show and Equality for Universal Type

```
showUniv :: Univ -> String
showUniv (UBool b)    =   show b
showUniv (UInt i)     =   show i
showUniv (UList us)   =
  "[" ++ concat (intersperse "," (map showUniv us)) ++ "]"

eqUniv :: Univ -> Univ -> Bool
eqUniv (UBool b) (UBool c)     =   b == c
eqUniv (UInt i) (UInt j)       =   i == j
eqUniv (UList us) (UList vs)   =
  and [ eqUniv u v | (u,v) <- zip us vs ]
```

Can't show functions or test them for equality.

```
lookUp :: HEnv -> Name -> Univ
lookUp r x =  the [ v | (y,v) <- r, x == y ]
  where
  the [v] = v
```

Micro-Haskell in Haskell

```
hEval :: Hask -> HEnv -> Univ
hEval HTrue r                =  UBool True
hEval HFalse r               =  UBool False
hEval (HIf c d e) r          =
  hif (hEval c r) (hEval d r) (hEval e r)
  where
  hif (UBool b) v w          =  if b then v else w
hEval (HLit i) r             =  UInt i
hEval (HEq d e) r            =  heq (hEval d r) (hEval e r)
  where
  heq (UInt i) (UInt j)      =  UBool (i == j)
hEval (HAdd d e) r           =  hadd (hEval d r) (hEval e r)
  where
  hadd (UInt i) (UInt j)     =  UInt (i + j)
hEval (HVar x) r             =  lookUp r x
hEval (HLam x e) r           =  UFun (\v -> hEval e ((x,v):r))
hEval (HApp d e) r           =  happ (hEval d r) (hEval e r)
  where
  happ (UFun f) v            =  f v
```

Test data

```
h0 =
 (HApp
   (HApp
     (HLam "x" (HLam "y" (HAdd (HVar "x") (HVar "y"))))
     (HLit 3))
   (HLit 4))

prop_h0 = eqUniv (hEval h0 []) (UInt 7)
```