Informatics 1

Functional Programming Lecture 10

Expression Trees as Algebraic Data Types

Don Sannella

University of Edinburgh

Part I

Expression Trees

Now we can use the ideas behind the definitions of List etc. to define EXPRESSIONS
and functions that manipulate them.

Expression Trees

Arithmetic expressions first. Called expression TREES because they reflect the tree-like structure of expressions.
Unlike (for example) arithmetic expressions represented using String.

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

An Exp is either a Lit (LITERAL) integer
or Add of two expressions
or Mul (Multiplication) of two expressions.

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

Evaluating expressions: given an Exp, return its value.
Left-hand side pattern :: Exp. Right-hand side :: Int

```
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)
```

Converting an Exp into a String.

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

Expression Trees

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

Two expressions with same literals and operators but different structure.

```
*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Here is how to draw them as trees:

```
e0 is        Add
            /   \
          Lit    Mul
           |    /   \
           2  Lit   Lit
               |     |
               3     3
```

```
e1 is        Mul
            /   \
         Add     Lit
        /   \     |
      Lit   Lit   3
       |     |
       2     3
```

Trees in Computer Science are drawn upside-down
Same in Linguistics. Terminology: ROOT, BRANCH, LEAF

Expression Trees, Infix

We could do the same thing using infix notation
Makes things a little clearer.

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

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 ++ ")"
```

Expression Trees, Infix

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

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Expression Trees, Symbols

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

Or, we can use infix SYMBOLS.
Remember, constructors always start with a capital letter.
Symbols used as constructors need to start with :
Here we put : at the end too, just for symmetry.

```
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 ++ ")"
```

Expression Trees, Symbols

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

*Main> showExp e0
"(2+(3*3))"

*Main> evalExp e0
11

*Main> showExp e1
"((2+3)*3)"

*Main> evalExp e1
15
```

Part II

Propositions

Propositions

We can do the same thing for propositions from Inf1-CL

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

A Prop (proposition) is either a Var (variable) with a name
or F (False - using F to avoid reuse of Bool constructor)
or T (True)
or Not (negation) of a Prop
or :|: (disjunction) of two Props
or :&: (conjunction) of two Props
(we could add :->: for implication etc.)

This part will be explained in a later lecture (type classes)

```
type Names = [Name]
type Env = [(Name,Bool)]
```

Names - will be used for sets of names
Env (environments) - will be used to map names to values

Note, the first case is Var Name, not just Name - we need the constructor to distinguish between cases.

Showing a proposition

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

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

Converting a Prop to a String. Notice how recursion is essential for this definition.
Notice how the structure of the definition follows exactly the structure of the type definition:
- there is one equation for each constructor
- the clauses for Var, F and T aren't recursive
- the clauses for Not, :|:, :&: are recursive, in just the same way that the type definition is recursive

You can read off the "shape" of the function definition from the form of the type definition.
You've seen this before for recursive definitions over lists.

You can write function definitions on algebraic types that don't follow the shape of the type definition, for instance

```
falsify :: Prop -> Prop
falsify p = F
```

but following the shape is common and a good starting point.

Names in a proposition

```
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)
```

Computing the set of all of the variable names in a Prop.
Important if you want to build a truth table for a proposition.
The built-in function nub removes duplicates from a list.

Evaluating a proposition in an environment

```
eval :: Env -> Prop -> Bool
eval e (Var x)          =  lookUp e x
eval e F                =  False
eval e T                =  True
eval e (Not p)          =  not (eval e p)
eval e (p :|: q)        =  eval e p || eval e q
eval e (p :&: q)        =  eval e p && eval e q

lookUp :: Eq a => [(a,b)] -> a -> b
lookUp xys x  =  the [ y | (x',y) <- xys, x == x' ]
  where
  the [x]  =  x
```

Evaluating a Prop tells us if it's true or false.
Only makes sense if we provide an ENVIRONMENT which gives the values of the variables.

Evaluation is along similar lines to evaluation of arithmetic expressions.
Left-hand pattern :: Prop. Right-hand side :: Bool

lookup is for looking up the value of a variable in an Env.
An Env is a list of (variable name, value) pairs.
The comprehension gives a list, which should contain one value.
the returns that value.

Propositions

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

e0 :: Env
e0 =  [("a",True)]

*Main> showProp p0
(a&(~a))

*Main> names p0
["a"]

*Main> eval e0 p0
False

*Main> lookUp e0 "a"
True
```

```
p0 is        :&:
            /   \
          Var   Not
           |     |
          "a"   Var
                 |
                "a"
```

How eval works

```
eval e (Var x)         =  lookUp e x
eval e F               =  False
eval e T               =  True
eval e (Not p)         =  not (eval e p)
eval e (p :|: q)       =  eval e p || eval e q
eval e (p :&: q)       =  eval e p && eval e q
```

```
  eval e0 (Var "a" :&: Not (Var "a"))
=
  (eval e0 (Var "a")) && (eval e0 (Not (Var "a")))
=
  (lookup e0 "a") && (eval e0 (Not (Var "a")))
=
  True && (eval e0 (Not (Var "a")))
=
  True && (not (eval e0 (Var "a")))
= ... =
  True && False
=
  False
```

Here's how eval works for this example.
The result will also be False if the environment says that a is False.
So this proposition is a CONTRADICTION.

Propositions

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

e1 :: Env
e1 =  [("a",False), ("b",False)]

*Main> showProp p1
((a&b)|((~a)&(~b)))

*Main> names p1
["a","b"]

*Main> eval e1 p1
True

*Main> lookUp e1 "a"
False
```

p1 is

```
            :|:
           /   \
          /     \
       :&:       :&:
      /   \     /   \
    Var   Var  Not   Not
     |     |    |     |
    "a"   "b"  Var   Var
                |     |
               "a"   "b"
```

All possible environments

```
envs :: Names -> [Env]
envs []       =  [[]]
envs (x:xs)   =  [ (x,False):e | e <- envs xs ] ++
                 [ (x,True ):e | e <- envs xs ]
```

Alternative

```
envs :: Names -> [Env]
envs []       =  [[]]
envs (x:xs)   =  [ (x,b):e | b <- bs, e <- envs xs ]
  where
  bs = [False,True]
```

To write functions for checking whether a Prop is a tautology, a contradiction, satisfiable etc.
we need to compute all of the possible environments over a set of variables.
Consider envs (x:xs) - combine the possible choices for x with all the possible choices for the other variables.
Careful with envs [] - there is one environment over the empty set of variables, namely the empty environment.
If you define envs [] = [], then envs anything = []

All possible environments

```
    envs []
=   [[]]

    envs ["b"]
=   [("b",False):[]] ++ [("b",True ):[]]
=   [[("b",False)],
     [("b",True )]]

    envs ["a","b"]
=   [("a",False):e | e <- envs ["b"] ] ++
    [("a",True ):e | e <- envs ["b"] ]
=   [("a",False):[("b",False)],("a",False):[("b",True )]] ++
    [("a",True ):[("b",False)],("a",True ):[("b",True )]]
=   [[("a",False),("b",False)],
     [("a",False),("b",True )],
     [("a",True ),("b",False)],
     [("a",True ),("b",True )]]
```

Here's an example.

Satisfiable

```
satisfiable :: Prop -> Bool
satisfiable p  =  or [ eval e p | e <- envs (names p) ]
```

A Prop is satisfiable if there is at least one environment that makes it evaluate to True.
Combining:
- getting the set of variables in a Prop (names)
- getting all the environments over those names (envs)
- evaluating Prop in each of those environments (eval)

Apply or to the list of results to get the answer - False only if Prop evaluates to False in all those environments.

Propositions

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

*Main> envs (names p1)
[[("a",False),("b",False)],
 [("a",False),("b",True)],
 [("a",True ),("b",False)],
 [("a",True ),("b",True )]]

*Main> [ eval e p1 | e <- envs (names p1) ]
[True,
 False,
 False,
 True]

*Main> satisfiable p1
True
```

Here's an example.

Part III

Maybe, Maybe Not

Optional Data

Maybe a is a built-in type that is handy when a value of type a may be missing.

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

A value of type Maybe a is either Nothing (value missing)
or a value of type a "wrapped up" with the constructor Just.

Optional argument

Useful for the situation where there is an optional argument,
with a DEFAULT when it is not supplied.

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

Optional result

Useful when a function may have no 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)
```

Using an optional result requires "unwrapping" it from the constructor.

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

Using "normal" division: n `div` m + 3
Doesn't work: divide n m :: Maybe Int
and + adds an Int to an Int

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

Just r :: Maybe Int, so r :: Int

case syntax is new: case expr of pat1 -> exp1 ... pat n -> expn

Part IV

Union of Two Types

Either a or b

This built-in type can be used to get lists with values of different types
For instance, [Either Int Bool]

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

A value of type Either a b is either a value of type a, wrapped up using Left,
or a value of type b, wrapped up using Right.

```
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
```

A function to add all of the integers in a list of type [Either Int String],
ignoring the strings.

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

The same function, written using comprehension.
Notice the use of the pattern Left n to select only the values of this form.

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
```

A function to concatenate all of the strings in a list of type [Either Int String], ignoring the integers.

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

The same function written using comprehension.

These examples haven't shown:
- types with multiple parameters
- mutually recursive types
- functional types as arguments of constructors

Part V

Aside:

All sublists of a list

All sublists of a list

```
subs :: [a] -> [[a]]
subs []       =  [[]]
subs (x:xs)   =  subs xs ++ [ x:ys | ys <- subs xs ]
```

All sublists of a list

```
     subs []
=    [[]]

     subs ["b"]
=    subs [] ++ ["b":ys | ys <- subs []]
=    [[]] ++ ["b":[]]
=    [[], ["b"]]

     subs ["a","b"]
=    subs ["b"] ++ ["a":ys | ys <- subs ["b"]]
=    [[], ["b"]] ++ ["a":[], "a":["b"]]
=    [[], ["b"], ["a"], ["a","b"]]
```

Part VI

The Universal Type and Micro-Haskell

Optional material: an interpreter for a tiny subset of 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 ]
eqUniv u v                     =  False
```

Can't show functions or test them for equality.

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

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

Test data

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

test_h0 = eqUniv (hEval h0 []) (UInt 7)
```