Informatics 1A

Functional Programming Lectures 10

Expression Trees as Algebraic Data Types

Ohad Kammar

The University of Edinburgh

Temporary changes

Topics swap back next week:

- Mon, Tue: Functional Programming
- Thu, Fri: Computation and Logic lectures

This means the quizzes swap too, so pay attention!

Informatics 1A

Functional Programming Lectures 10

Expression Trees as Algebraic Data Types

Ohad Kammar

The 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")))
=
           evalProp valn (Not (Var "a"))
= ... =
  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
```

Informatics 1A

Functional Programming Lectures 11

Expression Trees as Algebraic Data Types

Ohad Kammar

The University of Edinburgh

Part III

Propositions

Recap: Propositions

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

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 ++ ")"
```

Recap: 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
```

Satisfiability

A proposition `P` is satisfiable when:

there is a valuation `vn` such that `evalProp vn P == True`

Satisfiability

A proposition `P` is satisfiable when:

there is a valuation `vn` such that `evalProp vn P == True`

Our plan:

```
satisfiable :: Prop -> Bool
satisfiable p
  = or [ evalProp vn p | vn <- _ {- all valuations -} ]
```

How to search all valuations?

We can enumerate all the valuations, but it is an infinite list.

We can search this list for a satisfying valuation if it exists.

If `P` isn't satisfiable, we cannot exhaust the infinite list of valuations.

Idea: search only the relevant valuations.

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"]
```

Variables that Occur in a Proposition

Do the same mathematically:

$$\begin{aligned}
&\operatorname{names} p \subseteq \mathbf{Name} \\
&\operatorname{names} x = \{x\} \qquad\qquad (x \text{ variable}) \\
&\operatorname{names} \mathrm{F} = \emptyset \\
&\operatorname{names} \mathrm{T} = \emptyset \\
&\operatorname{names}(\neg p) = \operatorname{names} p \\
&\operatorname{names}(p \vee q) = \operatorname{names} p \cup \operatorname{names} q \\
&\operatorname{names}(p \wedge q) = \operatorname{names} p \cup \operatorname{names} q
\end{aligned}$$

For example:

$$\begin{aligned}
&\operatorname{names}(a \wedge \neg a) = \{a\} \\
&\operatorname{names}((a \wedge b) \vee ((\neg a) \wedge (\neg b))) = \{a, b\}
\end{aligned}$$

Relevant valuations

Theorem. *The value of a proposition* `p` *depends only on the value of variables in* names `p`*: for every two valuations* `vn0`, `vn1`, *if,*

$$\texttt{vn0}\ x == \texttt{vn1}\ x \quad \textit{for all } x \in \text{names}\, p,$$

then,

```
evalProp vn0 p == evalProp vn1 p.
```

Relevant valuations

Theorem. *The value of a proposition* `p` *depends only on the value of variables in* names `p`*: for every two valuations* `vn0`, `vn1`, *if,*

$$\texttt{vn0}\ x == \texttt{vn1}\ x \quad \textit{for all } x \in \text{names}\, p,$$

then,

```
evalProp vn0 p == evalProp vn1 p.
```

We will then search for a valuation `vn` that are only defined for variables in names `p`. Let's call these the relevant valuations:

Corollary. *A proposition* `p` *is satisfiable iff it has a satisfying relevant valuation.*

Relevant valuations

Theorem. *The value of a proposition* `p` *depends only on the value of variables in* names `p`*: for every two valuations* `vn0`, `vn1`, *if,*

$$\texttt{vn0}\ x == \texttt{vn1}\ x \quad \textit{for all } x \in \text{names}\, p,$$

then,

```
evalProp vn0 p == evalProp vn1 p.
```

We will then search for a valuation `vn` that are only defined for variables in names `p`. Let's call these the relevant valuations:

Corollary. *A proposition* `p` *is satisfiable iff it has a satisfying relevant valuation.*

Proof. ($\Leftarrow$) If there is a satisfying relevant valuation for `p`, it shows `p` satisfiable.
($\Rightarrow$) If there is a valuation `vn` that satisfies `p`, take this relevant valuation:

```
vn' x | x `elem` names p = vn x
vn' x | otherwise        = error "undefined"
```

Then `vn x == vn' x` for every `x` $\in$ names p. By the theorem:

```
evalProp vn' p == evalProp vn p == True
```

So `vn'` is a satisfying relevant valuation.

All Relevant Valuations

We will then search for a valuation `vn` that are only defined for variables in names `p`. Let's call these the relevant 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 Relevant Valuations

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

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

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

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

If names `p` has n elements, there are 'only' 2^n such valuations.

If n is small, we can search all of them quickly.

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 valuations 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
```

Satisfiability

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

This pattern is very common in Informatics: we analyse the computational problem mathematically (prove a theorem about it), and design and implement an algorithm based on it.

If the number of elements in `names p` is large, this algorithm is intractable.

We have algorithms that are faster in practice, such as the DPLL algorithm.

We, humanity, don't know if there is a fast algorithm, but we believe there isn't. This belief is the celebrated $\mathbf{P} \neq \mathbf{NP}$ conjecture.

Relevant valuations (proof)

Theorem. *The value of a proposition* `p` *depends only on the value of variables in* names `p`*: for every two valuations* `vn0`, `vn1`, *if,*

$$\texttt{vn0}\ x == \texttt{vn1}\ x \quad \textit{for all } x \in \text{names}\ p,$$

then,

```
evalProp vn0 p == evalProp vn1 p.
```

Proof. By structural induction on `p` (see textbook). Example base case:

- Base case `(Var x)`, then `x` $\in$ names p:

```
evalProp vn0 (Var x)
=
vn0 x
=
vn1 x
=
evalProp vn1 (Var x)
```

and two other base cases for `T` and `F`.

Relevant valuations

Theorem. *The value of a proposition* `p` *depends only on the value of variables in* names `p`*: for every two valuations* `vn0`, `vn1`, *if,*

$$\texttt{vn0}\ x == \texttt{vn1}\ x \quad \textit{for all } x \in \text{names}\ p,$$

then,

```
evalProp vn0 p == evalProp vn1 p.
```

Proof. By structural induction on `p` (see textbook). Example inductive case:

- Inductive case `(p :||: q)`: assuming
  `evalProp vn0 p == evalProp vn1 p` and
  `evalProp vn0 q == evalProp vn1 q` then:

```
evalProp vn0 (p :||: q)
  = (valProp vn0 p) || (valProp vn0 q)
  = (valProp vn1 p) || (valProp vn1 q)
  = evalProp vn1 (p :||: q)
```

  and two other inductive cases for `Not` and `(:&&:)`.

Part IV

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 V

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 VI

The Universal Type and Micro-Haskell

Micro-Haskell and the Universal Type

```
--  Represent ( x y -> x + y) 3 4
data  Hask  =  HTrue
            |  HFalse
            |  HIf Hask Hask Hask
            |  HLit Int
            |  HEq Hask Hask
            |  HAdd Hask Hask
            |  HVar Name
            |  HLam Name Hask
            |  HApp Hask Hask

-- hEval :: Hask -> HEnv -> ???

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

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)
```