
A Compositional Interpreter for Expressions
Talen & Compilers

Utrecht University

# Last week

ASTs (datatypes) and corresponding folds …

… applied in several analyses

Utrecht University

Today

# A compositional Interpreter for expressions

Lecture notes: 5.4-5

# Computing with parsers

Lecture notes: 6


# Learning goals

Learning goals:

- associate inherited and synthesised attributes with the different alternatives of (possibly mutually recursive) datatypes (or the different nonterminals of grammars)
- define algebraic semantics in terms of compositional (or syntax driven) code that is defined using algebras of computations which make use of inherited and synthesised attributes
- explain deforestation in the context of computing with parsers

Utrecht University

# A compositional Interpreter for expressions

Utrecht University

# Simple expressions again

```
data E = Add E E
       | Neg E
       | Num Int

type Ealgebra r = (r → r → r -- add
                  ,r → r     -- neg
                  ,Int → r)  -- num

foldE :: EAlgebra r → E → r
foldE (add,neg,num) = f
  where f (Add e1 e2) = add (f e1) (f e2)
        f (Neg e)     = neg (f e)
        f (Num n)     = num n
```

Utrecht University

# Evaluation

Direct recursion:

```
eval :: E              → Int
eval    (Add e1 e2)    =  eval e1 + eval e2
eval    (Neg e)        =  negate (eval e)
eval    (Num n)        =  n
```

Using folds:

```
evalAlgebra :: EAlgebra Int
evalAlgebra =  ((+),negate,id)

eval        =  foldE evalAlgebra
```


# Expressions with variables

Utrecht University

# Adding variables

We want to write expressions like `-x+1` and `x+y+y+z`

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id

type Id = String
```

Utrecht University

# Extending the algebra and fold

```
type Ealgebra r = (r → r → r -- add
                  ,r → r     -- neg
                  ,Int → r   -- num
                  ,Id → r)   -- var


foldE :: EAlgebra r → E → r
foldE (add,neg,num,var) = f
  where f (Add e1 e2) = add (f e1) (f e2)
        f (Neg e)     = neg (f e)
        f (Num n)     = num n
        f (Var x)     = var x
```

Utrecht University

# Evaluation revisited

What is the value of –x+1?

And how about x+y+y+z?

```
eval :: E              → Int
eval    (Add e1 e2)    =  eval e1 + eval e2
eval    (Neg e)        =  negate (eval e)
eval    (Num n)        =  n
eval    (Var x)        =  ???
```

x,y,z are **free variables**

Utrecht University

# Evaluating expressions with free variables

The value of an expression with free variables is a function

```
Env → Int
```

where `Env` is an environment mapping free variables to integers


# Representing environments

Using lists

```
type Env = [(Id,Int)]
```

Insert constant time, lookup linear

Using finite maps (balanced trees):

```
import Data.Map
type Env = Map Id Int
```

Insert/lookup O(log n)

Utrecht University

# Finite Map interface

```
import Data.Map as M -- short name to disambiguate

type Map k v -- abstract

M.empty  ::  Map k v -- not the parser combinator
insert   ::  Ord k ⇒ k → v → Map k v → Map k v
(!)      ::  Ord k ⇒ Map k v → k → v
```

Utrecht University

# Evaluation revisited II

```
eval :: E          → Env → Int
eval    (Add e1 e2) env  =  eval e1 env + eval e2 env
eval    (Neg e)     env  =  negate (eval e env)
eval    (Num n)     env  =  n
eval    (Var x)     env  =  env!x

evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra =  (\r1 r2 → \env → r1 env + r2 env
               ,\r     → \env → negate (r env)
               ,\n     → \env → n
               ,\x     → \env → env!x)
```

Utrecht University

# Where should the environment go?

```
evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra =  (\r1 r2 → \env → r1 env + r2 env
               ,\r     → \env → negate (r env)
               ,\n     → \env → n
               ,\x     → \env → env!x)


evalAlgebra :: Env → EAlgebra Int
evalAlgebra    env =
  (\r1 r2 → r1 + r2
  ,\r     → negate r
  ,\n     → n
  ,\x     → env!x)
```


# Expressions with definitions

Utrecht University

# Adding definitions

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def Id E E -- let x = e1 in e2
```

Utrecht University

# Extending the algebra and fold

```
type Ealgebra r = (r → r → r    -- add
                  ,r → r        -- neg
                  ,Int → r      -- num
                  ,Id → r       -- var
                  ,Id → r → r) -- def

foldE :: EAlgebra r → E → r
foldE (add,neg,num,var,def) = f
  where f (Add e1 e2)   = add (f e1) (f e2)
        f (Neg e)       = neg (f e)
        f (Num n)       = num n
        f (Var x)       = var x
        f (Def x e1 e2) = def x (f e1) (f e2)
```

Utrecht University

# Evaluation revisited again

What is the value of:

```
let x=1 in x
let x=y in x+x
let x=1 in let x=2 in x
let x=1 in let x=x+1 in x
```

Design decisions:

- We still need an environment, although we can define **closed terms** using definitions
- Inner definitions shadow outer definitions
- No recursive definitions: variable not available in rhs of definition

Utrecht University

# Evaluation revisited again II

```
eval :: E → Env → Int
... -- as before
eval (Def x e1 e2) env =
  eval e2 (insert x (eval e1 env) env)


evalAlgebra :: EAlgebra (Env → Int)
evalAlgebra =
  (... -- as before
  ,\x r1 r2 → \env → r2 (insert x (r1 env) env))
```

Utrecht University

# Q

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
**CYNXXG**

Enable answers by SMS

Utrecht University

# Mutually recursive datatypes declarations and expressions

Utrecht University

# Mutually recursive datatypes

In many real-life situations, datatypes are mutually recursive.

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def Id E E -- let x = e1 in e2
```

Utrecht University

# Mutually recursive datatypes

In many real-life situations, datatypes are mutually recursive.

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def D    E -- let x = e1 in e2

data D = Dcl Id E
```

Utrecht University

# An algebra for a family of datatypes

```
Add  ::  E → E → E
Neg  ::  E → E
Num  ::  Int → E
Var  ::  Id → E
Def  ::  D → E → E
Dcl  ::  Id → E → D
```

```
type EDAlgebra e d =
  (e → e → e
  ,e → e
  ,Int → e
  ,Id → e
  ,d → e → e
  ,Id → e → d)
```

Utrecht University

# A fold for a family of datatypes

```
foldE :: EDAlgebra e d → Expr → e
foldE (add,neg,num,var,def,dcl) = fe
where fe (Add e1 e2) = add (fe e1) (fe e2)
      fe (Neg e)     = neg (fe e)
      fe (Num n)     = num n
      fe (Var x)     = var x
      fe (Def d e)   = def (fd d) (fe e)
      fd (Dcl x e)   = dcl x (fe e)
```

Utrecht University

# Adapting evaluation (direct recursion)

```
evalE :: E →          Env → Int
evalE    (Add e1 e2) env = evalE e1 env + evalE e2 env
evalE    (Num e)     env = negate (evalE e env)
evalE    (Num n)     env = n
evalE    (Var x)     env = env!x
evalE    (Def d e)   env = evalE e (evalD d env)

evalD :: D →      Env → Env
evalD   (Dcl x e) env = insert x (evalE e env) env
```

Utrecht University

# Adapting evaluation (as a fold)

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env)
evalAlgebra=
  (\e1 e2 → \env → e1 env + e2 env
  ,\e →     \env → negate (e env)
  ,\n →     \env → n
  ,\x →     \env → env!x
  ,\d e →   \env → e (d env)
  ,\x e →   \env → insert x (e env) env)
```


# Lists of declarations

Utrecht University

# Multiple declarations

We often want to introduce multiple definitions:

```
data E = Add E E
       | Neg E
       | Num Int
       | Var Id
       | Def [D]E

data D = Dcl Id E
```

(Or create a separate datatype Ds for lists of declarations)

Utrecht University

# Adapting the algebra and fold

```
type EDAlgebra e d =
  (…
  ,[d] → e → e
  ,…)

foldE :: EDAlgebra e d → Expr → e
foldE (add,neg,num,var,def,dcl) = fe
where …
      fe (Def ds e) = def (map fd d) (fe e)
      fd (Dcl x e) = dcl x (fe e)
```

Utrecht University

# Adapting evaluation

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env)
evalAlgebra=
  (\e1 e2 → \env → e1 env + e2 env
  ,\e →     \env → negate (e env)
  ,\n →     \env → n
  ,\x →     \env → env!x
  ,\ds e →  \env → e (process ds env)
  ,\x e →   \env → insert x (e env) env)

process :: [Env → Env] → Env → Env
process ds env = foldl (flip ($)) env ds
```


# Use before def

Utrecht University

# Def before use, use before def

Earlier we mentioned for

```
let x=1 in let x=x+1 in x
```

No recursive definitions: variable not available in rhs of definition

Suppose we want to allow

```
let { x = y + 1
    ; y = 2
    ; z = x + y + 3 }
in z
```

Utrecht University

# Current and final environment

The result type for declarations now becomes

`Env → Env → Env`

The first environment is the current environment, which we will extend

The second environment is the final environment, which we use to evaluate the expression in

Utrecht University

# Adapting evaluation

```
evalAlgebra :: EDAlgebra (Env → Int) (Env → Env)
evalAlgebra=
  (…
  ,\ds e →  \env → let fenv = process ds env fenv
                   in e fenv
  ,\x e →   \env → \fenv → insert x (e fenv) env)


process :: [Env → Env → Env] → Env → Env → Env
process ds env fenv =
  foldl (\cenv d → d cenv fenv) env ds
```


# Summary

- We can define algebras and folds also for families of mutually recursive types, also if lists (or other types) surround the recursive positions
- Often, the result types of algebras are themselves functions
- Function arguments represent information that is distributed over the abstract syntax tree
- Function results represent information that is computed from the abstract syntax tree (and the distributed values)

Utrecht University

# Computing with parsers

Utrecht University

# Example: Matched parentheses

S → (S)S | ε

Examples: (())(()), (), (()()), …

Abstract syntax:

```
data Parens = Match Parens Parens
            | Empty
```

Utrecht University

# Nesting parentheses

```
parens  :: Parser Char Parentheses
parens  =  Match <$ open <*> parens <* close <*> parens
       <|> succeed Empty

nesting :: Parser Char Int
nesting =  (\b d → max (1+b) d) <$
             open <*> nesting <* close <*> nesting
       <|> succeed 0

nesting’  :: Parser Char Int
nesting’  =  depthParentheses <$> parens
```

Utrecht University

# (De)forestation

Transform `nesting'` into `nesting`

Utrecht University

# Class instead of abstract syntax

```
class Parens a where
  match :: a → a → a
  empty :: a

parens  :: Parens a => Parser Char a
parens  =  match <$ open <*> parens <* close <*> parens
       <|> succeed empty
```


# Class instances

```
instance Parens Parentheses where
  match = Match
  empty = Empty

instance Parens Int where
  match b d = max (1+b) d
  empty = 0
```

Just one instance per type

Utrecht University

# Passing an algebra to the parser

```
parens :: ParenthesesAlgebra a -> Parser Char a
parens (match,empty) = par where
  par  =  match <$ open <*> par <* close <*> par
      <|> succeed empty

nesting, breadth :: Parser Char Int
nesting  =  parens (\b d → max (1+b) d,0)
breadth  =  parens (\b d → d+1,0)
```

Utrecht University

# Summary

The compositional approach to developing compilers can be used in a deforested computation