Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

2019-12-02

6. Compositionality

This lecture

Compositionality

- Compiler overview
- Folding Back
- Matched parentheses
- Simple expressions
- A fold for all datatypes

6.1 Compiler overview

Phases of a compiler

Roughly:

- Lexing and parsing
- Analysis and type checking
- Desugaring
- Optimization
- Code generation

Phases of a compiler

Roughly:

- Lexing and parsing
- Analysis and type checking
- Desugaring
- Optimization
- Code generation

Note that not all compilers have all phases, and others may have more phases (typically multiple desugaring and optimization phases).

Abstract syntax trees

Abstract syntax trees (AST) play a central role:

- Some phases build ASTs (such as parsing).
- Most phases traverse ASTs (such as analysis, type checking, code generation).
- Some phases traverse one AST and build another (such as desugaring).

Status

So far

How to build ASTs using a combinator parser.

Status

So far

How to build ASTs using a combinator parser.

Now

How to traverse ASTs systematically in order to compute all sorts of information.

6.2 Folding Back

Functions over lists

$$\begin{aligned} \text{sum } [] &= 0 \\ \text{sum } (x : xs) &= x + \text{sum } xs \end{aligned}$$

$$\begin{aligned} \text{length } [] &= 0 \\ \text{length } (x : xs) &= 1 + \text{length } xs \end{aligned}$$

Functions over lists

$$\begin{aligned} \mathsf{sum}\ [] &= 0 \\ \mathsf{sum}\ (\mathsf{x} : \mathsf{xs}) &= \mathsf{x} + \mathsf{sum}\ \mathsf{xs} \end{aligned}$$

$$\begin{aligned} \mathsf{length}\ [] &= 0 \\ \mathsf{length}\ (\mathsf{x} : \mathsf{xs}) &= 1 + \mathsf{length}\ \mathsf{xs} \end{aligned}$$

We abstract the commonalities using a **fold**:

$$\begin{aligned} &\mathsf{foldr} :: (\mathsf{a} \to \mathsf{r} \to \mathsf{r}) \to \mathsf{r} \to [\mathsf{a}] \to \mathsf{r} \\ &\mathsf{foldr}\ \_\ \mathsf{v}\ [] = \mathsf{v} \\ &\mathsf{foldr}\ \mathsf{f}\ \ \mathsf{v}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{f}\ \mathsf{x}\ (\mathsf{foldr}\ \mathsf{f}\ \mathsf{v}\ \mathsf{xs}) \end{aligned}$$

$$\begin{aligned} \mathsf{sum} &= \mathsf{foldr}\ (+)\ 0 \\ \mathsf{length} &= \mathsf{foldr}\ (\lambda\_\ \mathsf{r} \to 1 + \mathsf{r})\ 0 \end{aligned}$$

List algebra

We can pack the arguments to foldr into a single one:

$$
\begin{aligned}
&\text{foldr} :: (r, a \to r \to r) \to [a] \to r \\
&\text{foldr}\ (v, \_)\ []\ \ \ \ \ \ \ \ = v \\
&\text{foldr}\ (v, f)\ \ (x : xs) = f\ x\ (\text{foldr}\ (v, f)\ xs)
\end{aligned}
$$

List algebra

We can pack the arguments to foldr into a single one:

```
foldr :: (r, a → r → r) → [a] → r
foldr (v, _) []       = v
foldr (v, f) (x : xs) = f x (foldr (v, f) xs)
```

The pair (v, f) is called a **list algebra**:

```
type ListAlgebra a r = (r, a → r → r)
foldr :: ListAlgebra a r → [a] → r
```

foldr receives a **list algebra** and a **list**, and returns a **result** from (the carrier of) the algebra.

map is a fold

Question

Write a list algebra mapAlg such that foldr (mapAlg f) = map f.

```
mapAlg :: (a → b) → ListAlgebra a [b]
```

map is a fold

Question

Write a list algebra mapAlg such that foldr (mapAlg f) = map f.

```
mapAlg :: (a → b) → ListAlgebra a [b]
```

Solution

```
mapAlg f = ([], λx ys → f x : ys)
```

6.3 Matched parentheses

Matched parentheses revisited

Grammar:

$$S \to (\ S\ )\ S \mid \varepsilon$$

Abstract syntax:

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

Matched parentheses revisited

Grammar:

$$S \to (\ S\ )\ S \mid \varepsilon$$

Abstract syntax:

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

Count the number of pairs:

```
count :: Parens → Int
count (Match p1 p2) = (count p1 + 1) + count p2
count Empty         = 0
```

Matched parentheses – contd.

Maximal nesting depth:

$$
\begin{aligned}
&\mathsf{depth} :: \mathsf{Parens} \to \mathsf{Int} \\
&\mathsf{depth}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = (\mathsf{depth}\ \mathsf{p}_1 + 1)\ \text{`}\mathsf{max}\text{`}\ \mathsf{depth}\ \mathsf{p}_2 \\
&\mathsf{depth}\ \mathsf{Empty} = 0
\end{aligned}
$$

Matched parentheses – contd.

Maximal nesting depth:

$$
\begin{aligned}
&\text{depth} :: \text{Parens} \to \text{Int} \\
&\text{depth}\ (\text{Match}\ p_1\ p_2) = (\text{depth}\ p_1 + 1)\ \text{`max`}\ \text{depth}\ p_2 \\
&\text{depth}\ \text{Empty} = 0
\end{aligned}
$$

String representation:

$$
\begin{aligned}
&\text{print} :: \text{Parens} \to \text{String} \\
&\text{print}\ (\text{Match}\ p_1\ p_2) = \text{"("} +\!\!+ \text{print}\ p_1 +\!\!+ \text{")"} +\!\!+ \text{print}\ p_2 \\
&\text{print}\ \text{Empty} = \text{""}
\end{aligned}
$$

Capturing the recursive structure

All the functions we have seen have the following structure:

$$
\begin{aligned}
&\mathsf{f} :: \mathsf{Parens} \to \ldots \\
&\mathsf{f}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \ldots \mathsf{f}\ \mathsf{p}_1 \ldots \mathsf{f}\ \mathsf{p}_2 \ldots \\
&\mathsf{f}\ \mathsf{Empty} \qquad\quad = \ldots
\end{aligned}
$$

Capturing the recursive structure

All the functions we have seen have the following structure:

$$
\begin{aligned}
&\mathsf{f} :: \mathsf{Parens} \to \ldots \\
&\mathsf{f}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \ldots \mathsf{f}\ \mathsf{p}_1 \ldots \mathsf{f}\ \mathsf{p}_2 \ldots \\
&\mathsf{f}\ \mathsf{Empty} \qquad\qquad = \ldots
\end{aligned}
$$

Idea

Let us abstract from this recursive structure.

Capturing the recursive structure – contd.

$$
\begin{aligned}
&\mathsf{f} :: \mathsf{Parens} \rightarrow \ \ldots \\
&\mathsf{f}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \ldots\ \mathsf{f}\ \mathsf{p}_1\ \ldots\ \mathsf{f}\ \mathsf{p}_2 \ldots \\
&\mathsf{f}\ \mathsf{Empty} \qquad\quad = \ldots
\end{aligned}
$$

Capturing the recursive structure – contd.

$$
\begin{aligned}
&\mathsf{f} :: \mathsf{Parens} \to \mathsf{r} \\
&\mathsf{f}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \ldots\ \mathsf{f}\ \mathsf{p}_1\ \ldots\ \mathsf{f}\ \mathsf{p}_2 \ldots \\
&\mathsf{f}\ \mathsf{Empty} \qquad\quad = \ldots
\end{aligned}
$$

Capturing the recursive structure – contd.

```
f :: Parens → r
f (Match p1 p2) = match (f p1) (f p2)
f Empty         = ...
```

Capturing the recursive structure – contd.

```
f :: Parens → r
f (Match p1 p2) = match (f p1) (f p2)
f Empty         = empty
```

Capturing the recursive structure – contd.

$$
\begin{aligned}
&\mathsf{f} :: \mathsf{Parens} \to \mathsf{r} \\
&\mathsf{f}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \mathsf{match}\ (\mathsf{f}\ \mathsf{p}_1)\ (\mathsf{f}\ \mathsf{p}_2) \\
&\mathsf{f}\ \mathsf{Empty} \qquad\quad = \mathsf{empty}
\end{aligned}
$$

Question

Given that the result type is r, what are the types of match and empty? And how do they compare to the types of Match and Empty?

Capturing the recursive structure – contd.

```
f :: Parens → r
f (Match p1 p2) = match (f p1) (f p2)
f Empty         = empty
```

Question

Given that the result type is r, what are the types of match and empty? And how do they compare to the types of Match and Empty?

```
match :: r → r → r
empty :: r
```

Capturing the recursive structure – contd.

```
f :: Parens → r
f (Match p1 p2) = match (f p1) (f p2)
f Empty         = empty
```

Question

Given that the result type is r, what are the types of match and empty? And how do they compare to the types of Match and Empty?

```
match :: r → r → r
empty :: r
```

```
Match :: Parens → Parens → Parens
Empty :: Parens
```

Capturing the recursive structure – contd.

For each of the functions count, depth and print we have to give different definitions for match and empty.

Capturing the recursive structure – contd.

For each of the functions count, depth and print we have to give different definitions for match and empty.

We have to abstract over these two functions:

```
type ParensAlgebra r = (r → r → r,    -- match
                        r)            -- empty
```

Capturing the recursive structure – contd.

For each of the functions count, depth and print we have to give different definitions for match and empty.

We have to abstract over these two functions:

```
type ParensAlgebra r = (r → r → r,    -- match
                        r)            -- empty
```

```
foldParens :: ParensAlgebra r → Parens → r
foldParens (match, empty) = f
  where f (Match p1 p2) = match (f p1) (f p2)
        f Empty         = empty
```

Using foldParens

```
countAlgebra :: ParensAlgebra Int
countAlgebra = (λc1 c2 → (c1 + 1) + c2, 0)
count = foldParens countAlgebra
```

```
depthAlgebra :: ParensAlgebra Int
depthAlgebra = (λd1 d2 → (d1 + 1) `max` d2, 0)
depth = foldParens depthAlgebra
```

```
printAlgebra :: ParensAlgebra String
printAlgebra = (λp1 p2 → "(" ++ p1 ++ ")" ++ p2, "")
print = foldParens printAlgebra
```

6.4 Simple expressions

Arithmetic expressions

Grammar:

$$
\begin{aligned}
E &\to E + E \\
E &\to - E \\
E &\to \mathrm{Nat} \\
E &\to (E)
\end{aligned}
$$

Transformed grammar:

$$
\begin{aligned}
E &\to E' + E \mid E' \\
E' &\to - E' \\
E' &\to \mathrm{Nat} \\
E' &\to (E)
\end{aligned}
$$

Arithmetic expressions

Grammar:

$$
\begin{aligned}
E &\to E + E \\
E &\to - E \\
E &\to \mathrm{Nat} \\
E &\to (E)
\end{aligned}
$$

Transformed grammar:

$$
\begin{aligned}
E &\to E' + E \mid E' \\
E' &\to - E' \\
E' &\to \mathrm{Nat} \\
E' &\to (E)
\end{aligned}
$$

Abstract syntax, based on original grammar:

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

Functions on expressions

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

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

Functions on expressions

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

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

Once more, the structure of the function reflects the structure of the datatype.

Can you write EAlgebra, foldE, **and the algebra for** eval**?**

Functions on expressions – contd.

Datatype:

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

Functions on expressions – contd.

Datatype:

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

Types of the constructors:

```
Add :: E → E → E
Neg :: E → E
Num :: Int → E
```

Functions on expressions – contd.

Datatype:

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

Types of the constructors:

```
Add :: E → E → E
Neg :: E → E
Num :: Int → E
```

Algebra:

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

Functions on expressions – contd.

With the algebra, we can define a fold:

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

Functions on expressions – contd.

With the algebra, we can define a fold:

```
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
```

Functions on expressions – contd.

With the algebra, we can define a fold:

```
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
```

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

eval = foldE evalAlgebra
```

6.5 A fold for all datatypes

How to build a fold, in general

For a datatype T, we can define a fold function as follows:

- Define an algebra type TAlgebra that is parameterized over all of T's parameters, plus a result type r.
- The algebra is a tuple containing one component per constructor function.
- The types of the components are like the types of the constructor functions, but all (recursive) occurrences of T are replaced with r.
- The fold function is defined by traversing the data structure, replacing constructors with their corresponding algebra components, and recursing where required.

Trees

Almost like Parens:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
Leaf  :: a → Tree a
Node :: Tree a → Tree a → Tree a
```

Trees

Almost like Parens:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
Leaf  :: a → Tree a
Node :: Tree a → Tree a → Tree a
```

```
type TreeAlgebra a r = (a → r,          -- leaf
                        r → r → r)      -- node
foldTree :: TreeAlgebra a r → Tree a → r
foldTree (leaf, node) = f
  where f (Leaf x)   = leaf x
        f (Node l r) = node (f l) (f r)
```

Tree algebra examples

```
sizeAlgebra    :: TreeAlgebra a Int
sumAlgebra     :: TreeAlgebra Int Int
inorderAlgebra :: TreeAlgebra a [a]
reverseAlgebra :: TreeAlgebra a (Tree a)
```

Tree algebra examples

```
sizeAlgebra    :: TreeAlgebra a Int
sumAlgebra     :: TreeAlgebra Int Int
inorderAlgebra :: TreeAlgebra a [a]
reverseAlgebra :: TreeAlgebra a (Tree a)
```

```
sizeAlgebra    = (const 1, (+))
sumAlgebra     = (id, (+))
inorderAlgebra = ((:[]), (++))
reverseAlgebra = (Leaf, flip Node)
```

Identity algebra

Question

```
idAlgebra :: TreeAlgebra a (Tree a)
```

Solution

```
idAlgebra = (Leaf, Node)
```

Every datatype has an **identity** algebra, which arises by using the **constructors** as components of the algebra.

User-defined lists

```
data List a = Nil
            | Cons a (List a)
Nil  :: List a
Cons :: a → List a → List a
```

Type parameters also appear as type parameters of the algebra:

```
type ListAlgebra a r = (r,
                        a → r → r)
foldList :: ListAlgebra a r → List a → r
foldList (nil, cons) = f
  where f Nil         = nil
        f (Cons x xs) = cons x (f xs)
```

Maybe

Works on non-recursive datatypes, too:

```
data Maybe a = Nothing
             | Just a

Nothing   :: Maybe a
Just      :: a → Maybe a

type MaybeAlgebra a r = (r,
                         a → r)

foldMaybe :: MaybeAlgebra a r → Maybe a → r
foldMaybe (nothing, just) = f
  where f Nothing = nothing
        f (Just x) = just x
```

foldMaybe **vs.** maybe

```
type MaybeAlgebra a r = (r,
                         a → r)

foldMaybe :: MaybeAlgebra a r → Maybe a → r
foldMaybe (nothing, just) = f
  where f Nothing = nothing
        f (Just x) = just x
```

```
maybe :: r → (a → r) → Maybe a → r
maybe nothing just Nothing = nothing
maybe nothing just (Just x) = just x

maybe nothing just == foldMaybe (nothing, just)
```

Bool

```
data Bool = True
          | False

True  :: Bool
False :: Bool
```

What is the algebra and the fold of Bool?

Bool

```
data Bool = True
          | False

True  :: Bool
False :: Bool
```

What is the algebra and the fold of Bool?

```
type BoolAlgebra r = (r,
                      r)

foldBool :: BoolAlgebra r → Bool → r
foldBool (true, false) True  = true
foldBool (true, false) False = false
```

Bool

```
data Bool = True
          | False

True  :: Bool
False :: Bool
```

What is the algebra and the fold of Bool?

```
type BoolAlgebra r = (r,
                      r)

foldBool :: BoolAlgebra r → Bool → r
foldBool (true, false) True  = true
foldBool (true, false) False = false
```

```
foldBool (true, false) x == if x then true else false
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Exercise 1

Write the type of the algebra for the following datatype:

```
data Expr v = Var v
            | App (Expr v) (Expr v)
            | Lam (Expr v)
```

This represents λ-expressions in which variables are represented by values of type v (the **λ-calculus**).

Exercise 2a

Here is the datatype of **symmetric lists**:

```
data SymList a = Zero
               | One a
               | Add a (SymList a) a
```

Write the algebra and the fold.

Exercise 2a

Here is the datatype of **symmetric lists**:

```
data SymList a = Zero
               | One a
               | Add a (SymList a) a
```

Write the algebra and the fold.

```
type SymListAlgebra a r = (r, a → r, a → r → a → r)

foldSymList :: SymListAlgebra a r → SymList a → r
foldSymList (z, _, _) Zero          = z
foldSymList (_, o, _) (One x)       = o x
foldSymList (z, o, a) (Add l c r) = a l (foldSymList (z, o, a) c) r
```

Exercise 2b

```
data SymList a = Zero
               | One a
               | Add a (SymList a) a
type SymListAlgebra a r = (r, a → r, a → r → a → r)
foldSymList :: SymListAlgebra a r → SymList a → r
```

Write an algebra to check whether a given symmetric list is a **palindrome** (it reads the same in the reverse order):

```
palinAlg :: Eq a ⇒ SymAlgebra a Bool
```

Advantages of using folds

- We stick to a systematic recursion pattern that is well known and easy to understand.
- Using a fold forces us to define semantics in a compositional fashion – the semantics of a whole term is composed from the semantics of its subterms.
- The systematic nature of a fold makes it easy to combine several folds into one. This is essential for efficiency in a compiler.

More on algebras

These ideas have spawned a lot of research:

- For more complicated traversals over datatypes, **attribute grammars** provide a common solution.
- Some algebras (like length) can be defined **independently** of the datatype. This is called **generic programming**.

Next lecture

- Mutually recursive datatypes.
- Defining algebras for more advanced computations.