Universiteit Utrecht

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

Talen en Compilers

2009/2010, periode 2

Andres Löh

Department of Information and Computing Sciences
Utrecht University

November 25, 2009

6. Compositionality

This lecture

Compositionality

- Compiler overview
- Example: Matched parentheses
- Example: simple expressions
- A fold for all datatypes
- Summary

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 Example: Matched parentheses

Matched parentheses revisited

Grammar:

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

Abstract syntax:

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

Matched parentheses revisited

Grammar:

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

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 revisited

Grammar:

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

Abstract syntax:

$$\begin{aligned}\textbf{data}\ \mathsf{Parens} &= \mathsf{Match}\ \mathsf{Parens}\ \mathsf{Parens} \\ &\mid\ \mathsf{Empty}\end{aligned}$$

Count the number of pairs:

$$\begin{aligned}&\mathsf{count} :: \mathsf{Parens} \rightarrow \mathsf{Int} \\ &\mathsf{count}\ (\mathsf{Match}\ p_1\ p_2) = (\mathsf{count}\ p_1 + 1) + \mathsf{count}\ p_2 \\ &\mathsf{count}\ \mathsf{Empty} \qquad\qquad\ \ = 0\end{aligned}$$

The function mirrors the recursive structure of the datatype.

Matched parentheses – contd.

Maximal nesting depth:

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

Matched parentheses – contd.

Maximal nesting depth:

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

String representation:

$$
\begin{aligned}
&\mathsf{print} :: \mathsf{Parens} \to \mathsf{String} \\
&\mathsf{print}\ (\mathsf{Match}\ \mathsf{p}_1\ \mathsf{p}_2) = \texttt{"("} +\!\!+\ \mathsf{print}\ \mathsf{p}_1 +\!\!+ \texttt{")"} +\!\!+\ \mathsf{print}\ \mathsf{p}_2 \\
&\mathsf{print}\ \mathsf{Empty} \qquad\qquad\quad = \texttt{""}
\end{aligned}
$$

Capturing the recursive structure

All the functions we have seen have the following structure:

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

Capturing the recursive structure

All the functions we have seen have the following structure:

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

Idea

Let us abstract from this recursive structure.

Capturing the recursive structure – contd.

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

Capturing the recursive structure – contd.

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

Capturing the recursive structure – contd.

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

Capturing the recursive structure – contd.

$$
\begin{aligned}
&f :: \mathsf{Parens} \to r \\
&f\ (\mathsf{Match}\ p_1\ p_2) = \mathsf{match}\ (f\ p_1)\ (f\ p_2) \\
&f\ \mathsf{Empty} \qquad\qquad\quad\ = \mathsf{empty}
\end{aligned}
$$

Capturing the recursive structure – contd.

$$
\begin{aligned}
&f :: \mathsf{Parens} \rightarrow r \\
&f\ (\mathsf{Match}\ p_1\ p_2) = match\ (f\ p_1)\ (f\ p_2) \\
&f\ \mathsf{Empty} \qquad\qquad\quad = empty
\end{aligned}
$$

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.

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

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?

$$
\begin{aligned}
&\mathsf{match} :: \mathsf{r} \to \mathsf{r} \to \mathsf{r} \\
&\mathsf{empty} :: \mathsf{r}
\end{aligned}
$$

Capturing the recursive structure – contd.

$$
\begin{aligned}
&f :: \mathsf{Parens} \to r \\
&f\ (\mathsf{Match}\ p_1\ p_2) = \mathit{match}\ (f\ p_1)\ (f\ p_2) \\
&f\ \mathsf{Empty} \qquad\qquad\ \ = \mathit{empty}
\end{aligned}
$$

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?

$$
\begin{aligned}
&\mathit{match} :: r \to r \to r \\
&\mathit{empty} :: r
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Match} :: \mathsf{Parens} \to \mathsf{Parens} \to \mathsf{Parens} \\
&\mathsf{Empty} :: \mathsf{Parens}
\end{aligned}
$$

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

$$\text{countAlgebra} :: \text{ParensAlgebra Int}$$
$$\text{countAlgebra} = (\lambda c_1\ c_2 \rightarrow (c_1 + 1) + c_2, 0)$$
$$\text{depthAlgebra} :: \text{ParensAlgebra Int}$$
$$\text{depthAlgebra} = (\lambda d_1\ d_2 \rightarrow (d_1 + 1)\ \text{`max`}\ d_2, 0)$$
$$\text{printAlgebra} :: \text{ParensAlgebra String}$$
$$\text{printAlgebra} = (\lambda p_1\ p_2 \rightarrow \texttt{"("} + p_1 + \texttt{")"} + p_2, \texttt{""})$$
$$\text{count} = \text{foldParens countAlgebra}$$
$$\text{depth} = \text{foldParens depthAlgebra}$$
$$\text{print} = \text{foldParens printAlgebra}$$

6.3 Example: simple expressions

Another example: expressions

Grammar:

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

Transformed grammar:

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

Another example: expressions

Grammar:

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

Transformed grammar:

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

Abstract syntax, based on original grammar:

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

Functions on expressions

$$\begin{aligned}\textbf{data}\ E &= \mathsf{Add}\ E\ E\\ &\mid \mathsf{Neg}\ E\\ &\mid \mathsf{Num}\ \mathsf{Int}\end{aligned}$$

$$\begin{aligned}&\mathsf{eval} :: E \rightarrow \mathsf{Int}\\ &\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2\\ &\mathsf{eval}\ (\mathsf{Neg}\ e) \quad\ \ = -(\mathsf{eval}\ e)\\ &\mathsf{eval}\ (\mathsf{Num}\ n) \quad\ = n\end{aligned}$$

Functions on expressions

$$
\begin{aligned}
\textbf{data}\ E &= \mathsf{Add}\ E\ E \\
&\mid \mathsf{Neg}\ E \\
&\mid \mathsf{Num}\ \mathsf{Int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eval} :: E \rightarrow \mathsf{Int} \\
&\mathsf{eval}\ (\mathsf{Add}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2 \\
&\mathsf{eval}\ (\mathsf{Neg}\ e) \qquad\ \ = -\,(\mathsf{eval}\ e) \\
&\mathsf{eval}\ (\mathsf{Num}\ n) \qquad = n
\end{aligned}
$$

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

Functions on expressions – contd.

Datatype:

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

Functions on expressions – contd.

Datatype:

$$\textbf{data}\ E = \mathrm{Add}\ E\ E \mid \mathrm{Neg}\ E \mid \mathrm{Num}\ \mathrm{Int}$$

Types of the constructors:

$$\begin{aligned} \mathrm{Add} &:: E \to E \to E \\ \mathrm{Neg} &:: E \to E \\ \mathrm{Num} &:: \mathrm{Int} \to E \end{aligned}$$

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
```

6.4 A fold for all datatypes

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)
```

Tree algebra examples

$$
\begin{aligned}
&\mathsf{sizeAlgebra} &&:: \mathsf{TreeAlgebra\ a\ Int}\\
&\mathsf{sumAlgebra} &&:: \mathsf{TreeAlgebra\ Int\ Int}\\
&\mathsf{inorderAlgebra} &&:: \mathsf{TreeAlgebra\ a\ [a]}\\
&\mathsf{reverseAlgebra} &&:: \mathsf{TreeAlgebra\ a\ (Tree\ a)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{sizeAlgebra} &&= (\mathsf{const}\ 1, (+))\\
&\mathsf{sumAlgebra} &&= (\mathsf{id}, (+))\\
&\mathsf{inorderAlgebra} &&= ((:[]), +\!\!+)\\
&\mathsf{reverseAlgebra} &&= (\mathsf{Leaf}, \mathsf{flip\ Node})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{idAlgebra} :: \mathsf{TreeAlgebra\ a\ (Tree\ a)}\\
&\mathsf{idAlgebra} = (\mathsf{Leaf}, \mathsf{Node})
\end{aligned}
$$

User-defined lists

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

Built-in lists

$$
\begin{aligned}
&\textbf{data}\ [a] = [\,]\\
&\qquad\qquad \mid\ a : [a]\\
&[\,]\ \ ::\ \ [a]\\
&(:)\ ::\ \ a \rightarrow [a] \rightarrow [a]\\
&\textbf{type}\ \mathrm{LAlgebra}\ a\ r = (r,\\
&\qquad\qquad\qquad\qquad a \rightarrow r \rightarrow r)\\
&\mathrm{foldL} :: \mathrm{LAlgebra}\ a\ r \rightarrow [a] \rightarrow r\\
&\mathrm{foldL}\ (\mathrm{nil}, \mathrm{cons}) = f\\
&\quad \textbf{where}\ f\ [\,] \qquad = \mathrm{nil}\\
&\qquad\qquad f\ (x : xs) = \mathrm{cons}\ x\ (f\ xs)
\end{aligned}
$$

foldL **vs.** foldr

```
type LAlgebra a r = (r,
                     a → r → r)
foldL :: LAlgebra a r → [a] → r
foldL (nil, cons) = f
   where f []       = nil
         f (x : xs) = cons x (f xs)
foldr :: (a → r → r) → r → [a] → r
foldr cons nil []       = nil
foldr cons nil (x : xs) = cons x (foldr cons nil xs)
foldr cons nil == foldL (nil, cons)
```


Universiteit Utrecht

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

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)
```


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

Bool

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Bool} = \mathsf{True} \\
&\qquad\qquad\ \ \mid \mathsf{False} \\
&\mathsf{True} :: \mathsf{Bool} \\
&\mathsf{False} :: \mathsf{Bool} \\
&\textbf{type}\ \mathsf{BoolAlgebra}\ \mathsf{r} = (\mathsf{r}, \\
&\qquad\qquad\qquad\qquad\quad \mathsf{r}) \\
&\mathsf{foldBool} :: \mathsf{BoolAlgebra}\ \mathsf{r} \to \mathsf{Bool} \to \mathsf{r} \\
&\mathsf{foldBool}\ (\mathsf{true}, \mathsf{false})\ \mathsf{True} = \mathsf{true} \\
&\mathsf{foldBool}\ (\mathsf{true}, \mathsf{false})\ \mathsf{False} = \mathsf{false}
\end{aligned}
$$

foldBool vs. if-then-else

```
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
```

6.5 Summary

Summary

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 occurrences of T are replaced with r.
- The fold function is define by traversing the data structure, replacing constructors with their corresponding algebra components, and recursing where required.

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.

Next lecture

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