Universiteit Utrecht

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

Talen en Compilers

2023 - 2024

David van Balen

Department of Information and Computing Sciences
Utrecht University

2023-12-07

6. Compositionality

This lecture

Compositionality

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

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

Functions over lists

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

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

Functions over lists

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

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

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

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

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

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:

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

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

$$\begin{aligned}&\textbf{type}\ \mathsf{ListAlgebra}\ \mathsf{a}\ \mathsf{r} = (\mathsf{r}, \mathsf{a} \to \mathsf{r} \to \mathsf{r})\\ &\mathsf{foldr} :: \mathsf{ListAlgebra}\ \mathsf{a}\ \mathsf{r} \to [\mathsf{a}] \to \mathsf{r}\end{aligned}$$

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.

$$\text{mapAlg} :: (a \to b) \to \text{ListAlgebra a } [b]$$
$$\text{mapAlg f} = (\_, \_)$$

map is a fold

Question

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

```
mapAlg :: (a → b) → ListAlgebra a [b]
mapAlg f = (_, _)
```

```
mapAlg f = ([], λa bs → _)
```

map is a fold

Question

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

```
mapAlg :: (a → b) → ListAlgebra a [b]
mapAlg f = (_, _)
```

```
mapAlg f = ([], λa bs → _)
```

```
mapAlg f = ([], λa bs → f a : bs)
```

filter is a fold

Write a list algebra filterAlg: foldr (filterAlg f) = filter f.

```
filterAlg :: (a → Bool) → ListAlgebra a [a]
filterAlg f = (_, _)
```

filter is a fold

Write a list algebra filterAlg: foldr (filterAlg f) = filter f.

$$\text{filterAlg} :: (a \to \text{Bool}) \to \text{ListAlgebra a [a]}$$
$$\text{filterAlg f} = (\_, \_)$$

$$\text{filterAlg f} = ([], \lambda x\ xs \to \_)$$

filter **is a fold**

Write a list algebra filterAlg: foldr (filterAlg f) = filter f.

$$\text{filterAlg} :: (a \to \text{Bool}) \to \text{ListAlgebra}\ a\ [a]$$
$$\text{filterAlg}\ f = (\_, \_)$$

$$\text{filterAlg}\ f = ([\,], \lambda x\ xs \to \_)$$

$$\text{filterAlg}\ f = ([\,], \lambda x\ xs \to \textbf{if}\ f\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)$$

6.3 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:

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

Matched parentheses – contd.

Maximal nesting depth:

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

Matched parentheses – contd.

Maximal nesting depth:

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

String representation:

```
print :: Parens → String
print (Match p1 p2) = "(" ++ print p1 ++ ")" ++ print p2
print Empty         = ""
```

Capturing the recursive structure

All the functions we have seen have the following structure:

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

Capturing the recursive structure

All the functions we have seen have the following structure:

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

Idea

Let us abstract from this recursive structure.

Capturing the recursive structure – contd.

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

Capturing the recursive structure – contd.

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

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.

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

Capturing the recursive structure – contd.

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

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.

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

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?

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

Capturing the recursive structure – contd.

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

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?

$$\begin{aligned}
&\text{match} :: r \rightarrow r \rightarrow r\\
&\text{empty} :: r
\end{aligned}$$

$$\begin{aligned}
&\text{Match} :: \text{Parens} \rightarrow \text{Parens} \rightarrow \text{Parens}\\
&\text{Empty} :: \text{Parens}
\end{aligned}$$

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.

$$\textbf{type}\ \mathsf{ParensAlgebra}\ r = (r \to r \to r, \quad \text{-- match}$$
$$r) \quad \text{-- empty}$$

Capturing the recursive structure – contd.

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

```
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

$$\mathsf{countAlgebra} :: \mathsf{ParensAlgebra\ Int}$$
$$\mathsf{countAlgebra} = (\lambda c_1\ c_2 \to c_1 + c_2 + 1, 0)$$
$$\mathsf{count} = \mathsf{foldParens\ countAlgebra}$$

$$\mathsf{depthAlgebra} :: \mathsf{ParensAlgebra\ Int}$$
$$\mathsf{depthAlgebra} = (\lambda d_1\ d_2 \to (d_1 + 1)\ \text{`max`}\ d_2, 0)$$
$$\mathsf{depth} = \mathsf{foldParens\ depthAlgebra}$$

$$\mathsf{printAlgebra} :: \mathsf{ParensAlgebra\ String}$$
$$\mathsf{printAlgebra} = (\lambda p_1\ p_2 \to \texttt{"("} +\!\!+\ p_1 +\!\!+ \texttt{")"} +\!\!+\ p_2, \texttt{""})$$
$$\mathsf{print} = \mathsf{foldParens\ printAlgebra}$$

6.4 Simple expressions

Arithmetic expressions

Grammar:

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

Transformed grammar:

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

Arithmetic expressions

Grammar:

$$E \rightarrow E + E$$
$$E \rightarrow - E$$
$$E \rightarrow Nat$$
$$E \rightarrow (E)$$

Transformed grammar:

$$E \rightarrow E' + E \mid E'$$
$$E' \rightarrow - E'$$
$$E' \rightarrow Nat$$
$$E' \rightarrow (E)$$

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

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

Leaf :: a → Tree a
Node :: Tree a → Tree a → Tree a
```

Trees

```
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

```
idAlgebra :: TreeAlgebra a (Tree a)
idAlgebra = (Leaf, Node)
```

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

Maybe

```
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

```
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
```

Exercise 1

Write the type of the algebra for the following datatype:

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

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

Exercise 1

Write the type of the algebra for the following datatype:

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

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

```
type ExprAlgebra v r = (v → r, r → r → r, v → r → r)
foldExpr :: ExprAlgebra v r → Expr v → r
foldExpr (var, app, lam) = f
   where f (Var v)     = var v
         f (App x y) = app (f x) (f y)
         f (Lam v e) = lam v (f e)
```

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
```


Universiteit Utrecht

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

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.

6.6 Advanced folds

Combining algebras: Fusion

You can combine two algebras to produce pairs of results:

$$\text{combine} :: \text{LAlgebra a x} \rightarrow \text{LAlgebra a y} \rightarrow \text{LAlgebra a } (x, y)$$
$$\text{combine } (v_1, f_1)\ (v_2, f_2)$$
$$= ((v_1, v_2), \lambda x\ (r_1, r_2) \rightarrow (f_1\ x\ r_1, f_2\ x\ r_2))$$

Now you only need to traverse the data structure **once**!

Universiteit Utrecht

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

Combining algebras: Fusion

You can combine two algebras to produce pairs of results:

$$\text{combine} :: \text{LAlgebra a x} \to \text{LAlgebra a y} \to \text{LAlgebra a } (x, y)$$
$$\text{combine } (v_1, f_1)\ (v_2, f_2)$$
$$= ((v_1, v_2), \lambda x\ (r_1, r_2) \to (f_1\ x\ r_1, f_2\ x\ r_2))$$

Now you only need to traverse the data structure **once**!

You can fuse a fold with a map:

$$\text{foldr f v . map g} == \text{foldr } (\lambda x\ xs \to f\ (g\ x)\ xs)\ v$$

$$\text{mapAlg} :: \text{LAlgebra b x} \to (a \to b) \to \text{LAlgebra a x}$$
$$\text{mapAlg } (v, f)\ g = (v, \lambda x\ xs \to f\ (g\ x)\ xs)$$

Fusing more algebras

$$
\begin{aligned}
\text{combine1} &:: \text{LAlgebra a x} \\
&\rightarrow \text{LAlgebra a y} \\
&\rightarrow \text{LAlgebra a } (x, y) \\
\text{combine2} &:: \text{LAlgebra a x} \\
&\rightarrow \text{LAlgebra a } (x \rightarrow y) \\
&\rightarrow \text{LAlgebra a } (x, x \rightarrow y, y) \\
\text{combine3} &:: \text{LAlgebra a x} \\
&\rightarrow \text{LAlgebra x y} \\
&\rightarrow \text{LAlgebra a } (x, y)
\end{aligned}
$$

Record Syntax

$$\textbf{type}\ \mathsf{ListAlgebra\ a\ r} = (\mathsf{r}, \mathsf{a} \rightarrow \mathsf{r} \rightarrow \mathsf{r})$$

$$\mathsf{foldList}\ (\mathsf{nil}, \mathsf{cons})\ [\,] = \mathsf{nil}$$
$$\mathsf{foldList}\ (\mathsf{nil}, \mathsf{cons})\ (\mathsf{x} : \mathsf{xs}) = \mathsf{cons}\ \mathsf{x}\ (\mathsf{foldList}\ (\mathsf{nil}, \mathsf{cons})\ \mathsf{xs})$$

$$\mathsf{lengthAlg} = (0, \lambda\_\ \mathsf{l} \rightarrow \mathsf{l} + 1)$$

Record Syntax

$$\textbf{type}\ \mathsf{ListAlgebra\ a\ r} = (\mathsf{r}, \mathsf{a} \to \mathsf{r} \to \mathsf{r})$$

$$\mathsf{foldList\ (nil, cons)\ []} = \mathsf{nil}$$
$$\mathsf{foldList\ (nil, cons)\ (x : xs)} = \mathsf{cons\ x\ (foldList\ (nil, cons)\ xs)}$$

$$\mathsf{lengthAlg} = (0, \lambda\_\ \mathsf{l} \to \mathsf{l} + 1)$$

$$\textbf{data}\ \mathsf{ListAlgebra\ a\ r} = \mathsf{ListAlg}$$
$$\{\mathsf{nil} \quad :: \mathsf{r}$$
$$, \mathsf{cons} :: \mathsf{a} \to \mathsf{r} \to \mathsf{r}\}$$

$$\mathsf{foldList\ alg\ []} = \mathsf{nil\ alg}$$
$$\mathsf{foldList\ alg\ (x : xs)} = \mathsf{cons\ alg\ x\ (foldList\ alg\ xs)}$$

$$\mathsf{lengthAlg} = \mathsf{ListAlg}\ \{\mathsf{nil} = 0, \mathsf{cons} = \lambda\_\ \mathsf{l} \to \mathsf{l} + 1\}$$

One Fix to rule them all...

We can separate the recursive structure of a datatype:

```
data Parens' r = Match' r r | Empty'
data List' a  r = Nil' | Cons' a r
```

and tie it back together using a **fixpoint**:

```
newtype Fix f = Fix {unFix :: f (Fix f)}
type Parens = Fix Parens'
type List a  = Fix (List' a)
```

```
[1, 2] == Fix (Cons' 1 (Fix (Cons' 2 (Fix Nil'))))
```

...and in the darkness fold them

```
type Algebra f a = f a → a

foldAlg :: Functor f ⇒ Algebra f a → Fix f → a
foldAlg alg (Fix x) = alg (fmap (foldAlg alg) x)
```

The length of a list can be defined that way:

```
lengthAlg :: Algebra (List′ Char) Int -- that is
          :: List′ Char Int → Int
lengthAlg Nil′        = 0
lengthAlg (Cons′ _ l) = 1 + l
```

Next lecture

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