Programming Languages

Recursion schemes
Inductive data types

Brief review

Recursion schemes on lists

Algebraic data types

Recursion schemes on other structures

The functions `map` and `filter`

Last class we saw the following functions:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x
                     then x : filter p xs
                     else filter p xs
```

What type does the expression `map filter` have?
How could we use it?

Anonymous functions

“Lambda” notation

An expression of the form:

```
\ x -> e
```

represents a function that takes a parameter `x` and returns `e`.

`(\ x1 x2 ... xn -> e)` $\equiv$ `(\ x1 -> (\ x2 -> ... (\ xn -> e)))`

Example

```
>> map (\ x -> (x, x)) [1, 2, 3]
⇝ [(1, 1), (2, 2), (3, 3)]
```

Structural recursion

Let `g :: [a] -> b` be defined by two equations:

$$\begin{array}{lcl}\texttt{g []} & = & \langle \textit{base case} \rangle \\ \texttt{g (x : xs)} & = & \langle \textit{recursive case} \rangle\end{array}$$

We say that the definition of `g` is given by **structural recursion** if:

1. The base case returns a "fixed" value `z` (it does not depend on `g`).
2. The recursive case **cannot** use the variables `g` or `xs`, except in the expression `(g xs)`:

$$\begin{array}{lcl}\texttt{g []} & = & \texttt{z} \\ \texttt{g (x : xs)} & = & \ldots \texttt{x} \ldots \texttt{(g xs)} \ldots\end{array}$$

Structural recursion

Examples of structural recursion

```
sum :: [Int] -> Int
sum []       = 0
sum (x : xs) = x + sum xs

(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

-- Insertion sort
isort :: Ord a => [a] -> [a]
isort []       = []
isort (x : xs) = insert x (isort xs)
```

Structural recursion

Example: recursion that is **not** structural

```
-- Selection sort
ssort :: Ord a => [a] -> [a]
ssort []       = []
ssort (x : xs) = minimum (x : xs)
               : ssort (removeMin (x : xs))
```

Folding lists to the right

The function `foldr` abstracts the structural recursion scheme:

```
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

What is its type?

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Every structural recursion is an instance of `foldr`.

Folding lists to the right

Example — sum with foldr

```
sum :: [Int] -> Int
sum = foldr (+) 0
```

```
  sum [1, 2]  ⇝   foldr (+) 0 [1, 2]
              ⇝   (+) 1 (foldr (+) 0 [2])
              ⇝   (+) 1 ((+) 2 (foldr (+) 0 []))
              ⇝   (+) 1 ((+) 2 0)
              ⇝*  3
```

Analogously:

```
product :: [Int] -> Int
product = foldr (*) 1

and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False
```

Folding lists to the right

Example — reverse with foldr

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

It is structurally recursive. How would we write it using `foldr`?

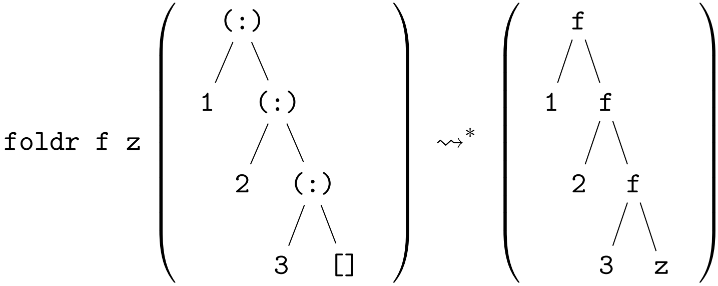
```
reverse = foldr (\ x rec -> rec ++ [x]) []
```

Other equivalent forms:

```
reverse = foldr (\ x rec -> flip (++) [x] rec) []
reverse = foldr (\ x -> flip (++) [x]) []
reverse = foldr (\ x -> flip (++) ((: []) x)) []
reverse = foldr (\ x -> (flip (++) . (: [])) x) []
reverse = foldr (flip (++) . (: [])) []
```

Folding lists to the right

Graphical illustration of right fold



In particular, it can be shown that:

```
        foldr (:) []  =  id
   foldr ((:) . f) []  =  map f
foldr (const (+ 1)) 0  =  length
```

Primitive recursion

Let `g :: [a] -> b` be defined by two equations:

```
g []        =  ⟨base case⟩
g (x : xs)  =  ⟨recursive case⟩
```

We say that the definition of g is given by **primitive recursion** if:

1. The base case returns a "fixed" value `z` (it does not depend on g).
2. The recursive case **cannot** use the variable g, except in the expression `(g xs)`.
   (It can use the variable `xs`).

```
g []        =  z
g (x : xs)  =  ... x ... xs ... (g xs) ...
```

Similar to structural recursion, but allows referring to `xs`.

Primitive recursion

Observation

- All definitions given by structural recursion are also given by primitive recursion.
- There are definitions given by primitive recursion that are not given by structural recursion.

Example

Given a text, remove all leading spaces.

```
trim :: String -> String
>> trim "   Hello PLP" ⇝ "Hello PLP"

trim []       = []
trim (x : xs) = if x == ' ' then trim xs else x : xs
```

Let's try to write it with `foldr`.

Primitive recursion

Let's write a function `recr` to abstract the primitive recursion scheme:

```
recr f z []       = z
recr f z (x : xs) = f x xs (recr f z xs)
```

What is its type?

```
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
```

Every primitive recursion is an instance of `recr`.

Now let's write `trim` using `recr`:

```
trim = recr (\ x xs rec -> if x == ' '
                            then rec
                            else x : xs)
            []
```

Iterative recursion

Let `g :: b -> [a] -> b` be defined by two equations:

```
g acc []         = ⟨base case⟩
g acc (x : xs)   = ⟨recursive case⟩
```

Iterative recursion

We say that the definition of `g` is given by *iterative recursion* if:

1. The base case returns the accumulator `acc`.
2. The recursive case immediately invokes `(g acc' xs)`, where `acc'` is the updated accumulator based on its previous value and the value of `x`.

Iterative recursion

Examples of iterative recursion

```
-- Reverse with accumulator.
reverse' :: [a] -> [a] -> [a]
reverse' acc []       = acc
reverse' acc (x : xs) = reverse' (x : acc) xs

-- Binary to decimal conversion with accumulator.
-- Precondition: receives a list of 0s and 1s.
bin2dec' :: Int -> [Int] -> Int
bin2dec' acc []       = acc
bin2dec' acc (b : bs) = bin2dec' (b + 2 * acc) bs

-- Insertion sort with accumulator.
isort' :: Ord a => [a] -> [a] -> [a]
isort' acc []       = acc
isort' acc (x : xs) = isort' (insert x acc) xs
```

Folding lists to the left

Let's write a function `foldl` to abstract the iterative recursion scheme:

```
foldl f acc []       = acc
foldl f acc (x : xs) = foldl f (f acc x) xs
```

What is its type?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Every iterative recursion is an instance of `foldl`.

Folding lists to the left

In general `foldr` and `foldl` have different behaviors:

```
foldr (★) z [a, b, c]  =  a ★ (b ★ (c ★ z))
foldl (★) z [a, b, c]  =  ((z ★ a) ★ b) ★ c
```

If (★) is an associative and commutative operator, `foldr` and `foldl` define the same function. For example:

```
sum     = foldr (+) 0      = foldl (+) 0
product = foldr (*) 1      = foldl (*) 1
and     = foldr (&&) True  = foldl (&&) True
or      = foldr (||) False = foldl (||) False
```

Folding lists to the left

Example — binary to decimal conversion

```haskell
bin2dec :: [Int] -> Int
bin2dec = foldl (\ acc b -> b + 2 * acc) 0
```

```
    bin2dec [1, 0, 0]
⇝   foldl (\ acc b -> b + 2 * acc)  0                           [1, 0, 0]
⇝   foldl (\ acc b -> b + 2 * acc)  (1 + 0)                     [0, 0]
⇝   foldl (\ acc b -> b + 2 * acc)  (0 + 2 * (1 + 0))           [0]
⇝   foldl (\ acc b -> b + 2 * acc)  (0 + 2 * (0 + 2 * (1 + 0))) []
⇝   0 + 2 * (0 + 2 * (1 + 0))
⇝*  4
```

Folding lists to the left

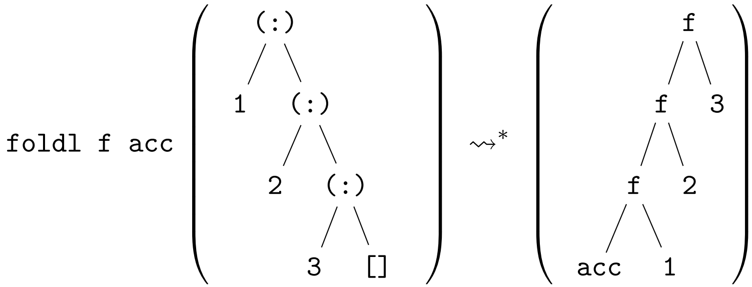
The function `foldl` is an iteration operator.

Imperative pseudocode:

```
function foldl f acc xs {
    while xs is not empty {
        acc := f acc (head xs)
        xs := tail xs
    }
    return acc
}
```

Folding lists to the left

Graphical illustration of left fold



In particular, it can be shown that:

```
foldl (flip (:)) []  =  reverse
```

Summary: recursion schemes on lists

We saw the following recursion schemes on lists:

1. Structural recursion. .... `foldr`
2. Primitive recursion. .... `recr`
3. Iterative recursion. .... `foldl`

Exercises to think about

Simultaneous recursion over more than one structure

Define the following function using `foldr`. (Not so easy).

```
zip :: [a] -> [b] -> [(a, b)]
zip []       []       = []
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

Relationship between structural and primitive recursion

1. Define `foldr` in terms of `recr`. (Easy).
2. Define `recr` in terms of `foldr`. (Not so easy).
   Idea: return a tuple with a copy of the original list.

Relationship between structural and iterative recursion

1. Define `foldl` in terms of `foldr`.
2. Define `foldr` in terms of `foldl`.

Algebraic data types

We know some "primitive" data types:

```
Char   Int   Float   (a -> b)   (a, b)   [a]
```

`String` (synonym for `[Char]`)

New data types can be defined with the `data` clause:

`data Type =` ⟨*declaration of constructors*⟩

Algebraic data types

Example — enumerated types

Many constructors without parameters:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Declares that there exist constructors:

```
Sun :: Day     Mon :: Day   ...   Sat :: Day
```

It also declares that these are the **only** constructors of the type `Day`.

```
isWeekend :: Day -> Bool
isWeekend Sat = True
isWeekend Sun = True
isWeekend _   = False
```

```
>> isWeekend Fri
⇝ False
```

Algebraic data types

Example — product types (tuples/structures/records/...)

A single constructor with many parameters:

```
data Person = Person String String Int
```

Declares that the type `Person` has one constructor (and **only that one**):

```
    Person :: String -> String -> Int -> Person
```

```
firstName, lastName :: Person -> String
birthYear            :: Person -> Int
firstName           (Person n _ _) = n
lastName            (Person _ a _) = a
birthYear           (Person _ _ y) = y
```

```
rebeccaGuber = Person "Rebecca" "Guber" 1926
>> lastName rebeccaGuber
⇝  "Guber"
```

Algebraic data types

Example

A type can have many constructors with many parameters:

```
data Shape = Rectangle Float Float
           | Circle Float
```

Declares that the type Shape has two constructors (and **only those**):

```
      Rectangle  :: Float -> Float -> Shape
      Circle     :: Float -> Shape
```

```
area :: Shape -> Float
area (Rectangle width height) = width * height
area (Circle radius)          = radius * radius *
  pi
```

Algebraic data types

Example

Some constructors can be **recursive**:

```
data Nat = Zero
         | Succ Nat
```

Declares that the type `Nat` has two constructors (and **only those**):

```
Zero  :: Nat
Succ  :: Nat -> Nat
```

What form do values of type `Nat` take?

```
Zero

Succ Zero

Succ (Succ Zero)

Succ (Succ (Succ Zero))

...
```

Algebraic data types

Functions on data types with recursive constructors are normally defined by recursion:

```
double :: Nat -> Nat
double Zero     = Zero
double (Succ n) = Succ (Succ (double n))
```

The following equation, does it define a value of type `Nat` or is it an error?

```
infinity :: Nat
infinity = Succ infinity
```

Answer:

- It depends on how recursive definitions are interpreted.
- Generally we are interested in finite structures.
- In Haskell, it is allowed to work with infinite structures.
  Technically speaking: in Haskell, recursive definitions are interpreted *coinductively* rather than *inductively*.
- Occasionally we will talk about infinite structures.

Algebraic data types

Algebraic data type — general case

In general, an algebraic data type has the following form:

$$\begin{array}{lll} \texttt{data T} & = & \texttt{CBase}_1 \ \langle \textit{parameters} \rangle \\ & & \ldots \\ & | & \texttt{CBase}_n \ \langle \textit{parameters} \rangle \\ & | & \texttt{CRecursive}_1 \ \langle \textit{parameters} \rangle \\ & & \ldots \\ & | & \texttt{CRecursive}_m \ \langle \textit{parameters} \rangle \end{array}$$

- Base constructors do not receive parameters of type `T`.
- Recursive constructors receive at least one parameter of type `T`.
- Values of type `T` are those that can be built by applying base and recursive constructors a **finite** number of times, and **only** those.

(We understand the definition of `T` **inductively**).

Example: lists

Lists are a particular case of an algebraic type:

```
data List a = Empty | Cons a (List a)
```

Or, with the already known notation:

```
data [a] = [] | a : [a]
```

```
cartesianProduct :: [a] -> [b] -> [(a, b)]

cartesianProduct xs ys =
  concat (map (\ x -> map (\ y -> (x, y)) ys) xs)
```

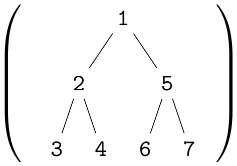
Example: binary trees

```
data BT a = Nil | Bin (BT a) a (BT a)
```

Let's define the following functions:

```
preorder  :: BT a -> [a]
postorder :: BT a -> [a]
inorder   :: BT a -> [a]
```

t =



```
preorder  t  ⇝*  [1, 2, 3, 4, 5, 6, 7]
postorder t  ⇝*  [3, 4, 2, 6, 7, 5, 1]
inorder   t  ⇝*  [3, 2, 4, 1, 6, 5, 7]
```

Example: binary trees

```
insert :: Ord a => a -> BT a -> BT a
```

Pre: the input tree is a BST (without duplicates).
Post: the resulting tree is a BST (without duplicates) that contains the elements of the input BST and the given element.

```
insert x Nil = Bin Nil x Nil
insert x (Bin left y right)
  | x < y     = Bin (insert x left) y right
  | x > y     = Bin left y (insert x right)
  | otherwise = Bin left y right
```

Brief review

Recursion schemes on lists

Algebraic data types

Recursion schemes on other structures

Structural recursion

In the case of lists, given a function `g :: [a] -> b`:

```
g []        = ⟨base case⟩
g (x : xs)  = ⟨recursive case⟩
```

we said that `g` was given by structural recursion if:

- The base case returns a fixed value `z`.
- The recursive case **cannot** use the parameters `g` or `xs`, except in the expression `(g xs)`:

Structural recursion

Structural recursion generalizes to algebraic types in general.

Suppose `T` is an algebraic type.

Given a function `g :: T -> Y` defined by equations:

$$\begin{array}{lcl}
\texttt{g (CBase}_1\ \langle \textit{parameters} \rangle\texttt{)} & = & \langle \textit{base case}_1 \rangle \\
\ldots & & \\
\texttt{g (CBase}_n\ \langle \textit{parameters} \rangle\texttt{)} & = & \langle \textit{base case}_n \rangle \\
\texttt{g (CRecursive}_1\ \langle \textit{parameters} \rangle\texttt{)} & = & \langle \textit{recursive case}_1 \rangle \\
\ldots & & \\
\texttt{g (CRecursive}_m\ \langle \textit{parameters} \rangle\texttt{)} & = & \langle \textit{recursive case}_m \rangle
\end{array}$$

We say that `g` is given by **structural recursion** if:

1. Each base case is written by combining the parameters.
2. Each recursive case:
   - Does not use the function `g`.
   - Does not use the parameters of the constructor that are of type `T`.

   But it can:
   - Make recursive calls on parameters of type `T`.
   - Use the parameters of the constructor that are *not* of type `T`.

Structural recursion

```
data BT a = Nil
          | Bin (BT a) a (BT a)
```

Example

Let's define a function `foldBT` that abstracts the structural recursion scheme on binary trees.

```
foldBT :: b -> (b -> a -> b -> b) -> BT a -> b
```

```
foldBT cNil cBin Nil          = cNil
foldBT cNil cBin (Bin l r d) =
  cBin (foldBT cNil cBin l) r (foldBT cNil cBin d)
```

Structural recursion

To think about

1. What function is `(foldBT Nil Bin)`?
2. Define `mapBT :: (a -> b) -> BT a -> BT b` using `foldBT`.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Recommended reading

Hutton's article.

Graham Hutton. *A tutorial on the universality and expressiveness of fold.*
J. Functional Programming 9 (4): 355–372, July 1999.

Comments: structural recursion

Degenerate cases of structural recursion

It is structural recursion (does not use the head):

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

It is structural recursion (does not use the recursive call on the tail):

```
head :: [a] -> a
head []      = error "No head."
head (x : _) = x
```