Intro to FP
Part 2: Lists and other Algebraic Datatypes

Jeremy Gibbons
University of Oxford

John Hughes 2023

2.1   List notation

- lists are central to functional programming (cf LISP!)
- sequences of elements of the same type
- enclosed in square brackets, comma-separated: $[1, 2, 3]$, $[\,]$
- the type of lists with elements of type A is $[A]$
- strings are just lists of characters: $[\,$'H','e','l','l','o'$\,]$

    type *String* = $[Char]$

  but with special syntax "Hello"

- list elements can be any type:

    $[1, 2, 3]$ :: $[Integer]$
    $[[1, 2], [\,], [3]]$ :: $[[Integer]]$
    $[(+), (\times)]$ :: $[Integer \rightarrow Integer \rightarrow Integer]$

2.2   List constructors

- a list is either
    - ▶ empty, written [ ]
    - ▶ or consists of an element x followed by a list xs, written $x : xs$
- every finite list can be built up from [ ] using :
- eg $[1, 2, 3] = 1 : (2 : (3 : [\,])) = 1 : 2 : 3 : [\,]$

2.2 List constructors

- a list is either
  - ▶ empty, written [ ]
  - ▶ or consists of an element *x* followed by a list *xs*, written *x* : *xs*
- every finite list can be built up from [ ] using : *in a unique way*
- eg [1, 2, 3] = 1 : (2 : (3 : [ ])) = 1 : 2 : 3 : [ ]

- [ ] and : are called *constructors*

2.3 Pattern matching

- constructors are exhaustive
- to define function over lists, it suffices to consider the two cases [ ] and :
- eg to test if list is empty

    $null :: [Integer] \rightarrow Bool$
    $null\ [\ ] = True$
    $null\ (x : xs) = False$

- eg to return first element of non-empty list

    $head :: [Integer] \rightarrow a$
    $head\ (x : xs) = x$

2.3 Case analysis

- cases can also be analysed using a **case**-expression

    null :: [*Integer*] → *Bool*
    null xs = **case** *xs* **of**
                  [ ]   → *True*
                  (*x* : *xs'*) → *False*

- *declaration style*: equation using patterns;
  expression style: **case**-expression using patterns

2.3 Recursive definitions

- definitions by pattern-matching can be recursive too
- natural as the type is also recursively defined
- eg sum of a list

    $sum :: [Integer] \rightarrow Integer$
    $sum\ [\ ] \qquad = 0$
    $sum\ (x : xs) = x + sum\ xs$

- eg length of a list

    $length :: [Integer] \rightarrow Int$
    $length\ [\ ] \qquad = 0$
    $length\ (x : xs) = 1 + length\ xs$

2.3   List definition pattern

- remember: every type comes with a definition pattern
- *task:* define a function $f :: [P] \to S$
- *step 1:* solve the problem for the empty list

    $f\,[\,] = \ldots$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for *xs* at hand;
  extend the intermediate solution to a solution for $x : xs$

    $f\,[\,] \qquad = \ldots$
    $f\,(x : xs) = \ldots x \ldots xs \ldots f\,xs \ldots$

  you have to program only a *step*
- put on your problem-solving glasses

2.4   Some list operations

- append: $[1, 2, 3] + [4, 5] = [1, 2, 3, 4, 5]$

    $(+) :: [Integer] \rightarrow [Integer] \rightarrow [Integer]$
    $[\ ] \qquad + ys = ys$
    $(x : xs) + ys = x : (xs + ys)$

- concatenation: $concat [[1, 2], [\ ], [3]] = [1, 2, 3]$

    $concat :: [[Integer]] \rightarrow [Integer]$
    $concat [\ ] \qquad = [\ ]$
    $concat (xs : xss) = xs + concat\ xss$

- reverse: $reverse [1, 2, 3] = [3, 2, 1]$

    $reverse :: [Integer] \rightarrow [Integer]$
    $reverse [\ ] \qquad = [\ ]$
    $reverse (x : xs) = reverse\ xs + [x]$

    (exercise: complexity? improve!)

2.5 List comprehensions

- two useful operators on lists: *map* and *filter*
- list comprehensions provide a convenient syntax for expressions involving *map*, *filter*, *concat*
- analogous to a database query language
- useful for constructing new lists from old lists

2.5 Map

- applies given function to every element of given list
- eg *map square* $[1, 2, 3] = [1, 4, 9]$
- a higher-order function
- definition

  > *map* :: (*Integer* → *Integer*) → [*Integer*] → [*Integer*]
  > *map f* [ ]   = [ ]
  > *map f* (*x* : *xs*) = *f x* : *map f xs*

- another eg: *sum* (*map square* [1 . . 10])
- (special syntax [*m* . . *n*] for enumerations)

2.5 Filter

- returns sublist of the argument whose elements satisfy given predicate
- eg (*sum* ∘ *map square* ∘ *filter odd*) [1..5] = 35
- definition

    filter :: (*Integer* → *Bool*) → [*Integer*] → [*Integer*]
    filter p [ ] = [ ]
    filter p (*x* : *xs*)
        | *p x*   = *x* : *filter p xs*
        | *otherwise* =   *filter p xs*

- another higher-order function

2.5   Comprehensions

- special convenient syntax for list-generating expressions
- eg $sum \, [\, square \; x \mid x \leftarrow [\,1 \mathinner{\ldotp\ldotp} 5\,], odd \; x\,]$
- formally, a comprehension $[\, e \mid Qs\,]$ for expression e and non-empty comma-separated sequence of qualifiers Qs
- qualifier may be *generator* (of the form $x \leftarrow xs$) or *guard* (a boolean expression)

2.5   Examples of comprehensions

- eg primes up to a given bound

    primes, divisors :: *Integer* → [*Integer*]
    primes m = [*n* | *n* ← [1 .. *m*], *divisors n* == [1, *n*]]
    divisors n = [*d* | *d* ← [1 .. *n*], *n* '*mod*' *d* == 0]

- eg database query

    overdue =
      [(*name, addr*) | (*key, name, addr*) ← *customers*,
                          (*key', date, amount*) ← *invoices, key* == *key',*
                          date < *today*]

- eg Quicksort

    quicksort :: [*Integer*] → [*Integer*]
    quicksort [ ]   = [ ]
    quicksort (*x* : *xs*) = *quicksort* [*y* | *y* ← *xs, y* < *x*]
                        ++ [*x*] ++
                        quicksort [*y* | *y* ← *xs, y* ⩾ *x*]

2.5 Another point of view

- list comprehension is 'really' a form of nested loop
- eg $[f\,a\,b \mid a \leftarrow x, b \leftarrow g\,a, p\,b]$ is related to

```
foreach (int a in x)
  foreach (int b in g a)
    if (p b)
      yield (f a b)
```

2.6 Polymorphism (a sneak preview)

- most of those list functions not specific to [*Integer*]
- how to capture generality, maintaining safety? polymorphism!

  ```
  [ ]       :: [a]                           -- these are all parametric polymorphism
  (:)       :: a → [a] → [a]
  null      :: [a] → Bool
  length    :: [a] → Int
  (+)       :: [a] → [a] → [a]
  concat    :: [[a]] → [a]
  reverse   :: [a] → [a]
  map       :: (a → b) → [a] → [b]
  filter    :: (a → Bool) → [a] → [a]
  quicksort :: (Ord a) ⇒ [a] → [a]   -- this is constrained polymorphism
  ```

- covered properly tomorrow. . .

2.7   Compositional programming

- exploring the library *Data.List*

    import *Data.List*

- *concat* :: [[*a*]] → [*a*] eg *concat* [[1, 2], [ ], [3]] = [1, 2, 3]
- *length* :: [*a*] → *Int* eg *length* [1, 2, 3] = 3
- *reverse* :: [*a*] → [*a*] eg *reverse* "oxford" = "drofxo"
- *map* :: (*a* → *b*) → [*a*] → [*b*] eg *map* (+1) [1, 2, 3] = [2, 3, 4]
- *lines* :: *String* → [*String*] eg *lines* "a\nbc\nd" = ["a", "bc", "d"]
- *unlines* :: [*String*] → *String* eg *unlines* ["a", "bc", "d"] = "a\nbc\nd\n"
- *tails* :: [*a*] → [[*a*]] eg
  tails "oxford" = ["oxford", "xford", "ford", "ord", "rd", "d", ""]

2.7   How to solve it?

- write down the type (what's the input?, what's the output?)
- can you solve it using existing vocabulary?
- use function application and function composition

- some exercises: given a string (a list of characters)
  - ▶ remove newlines
  - ▶ count the number of lines
  - ▶ flip text upside down
  - ▶ flip text from left to right
  - ▶ determine the list of all substrings

2.7 Solutions

- remove newlines

    unwrap :: *String* → *String*
    unwrap = *concat* ∘ *lines*

- count the number of lines

    countLines :: *String* → *Int*
    countLines = *length* ∘ *lines*

- flip text upside down

    upsideDown :: *String* → *String*
    upsideDown = *unlines* ∘ *reverse* ∘ *lines*

- flip text from left to right

    leftRight :: *String* → *String*
    leftRight = *unlines* ∘ *map reverse* ∘ *lines*

2.7 Solutions continued

- determine the list of all prefixes
  (actually, also defined in the library: *inits*)

  > *suffixes, prefixes* :: *String* → [*String*]
  > *suffixes* = *tails*
  > *prefixes* = *map reverse* ∘ *tails* ∘ *reverse*

- determine the list of all substrings

  > *substrings* :: *String* → [*String*]
  > *substrings* = *concat* ∘ *map prefixes* ∘ *suffixes*

2.8 Fold, scan, and unfold

- many recursive definitions on lists share a *pattern* of computation
- capture that pattern as a function
  (abstraction, conciseness, general properties, familiarity, . . .)
- *map* and *filter* are two common patterns
- folds and unfolds capture many more

2.8 Fold right

- consider following pattern of definition

    $h \, [\,] \quad\quad = e$
    $h \, (x : xs) = x \, \text{'op'} \, h \, xs$

    (simple variant of list definition pattern: xs is only used in the recursive call)

- then

    $h \, (x : (y : (z : [\,]))) = x \, \text{'op'} \, (y \, \text{'op'} \, (z \, \text{'op'} \, e))$

- h replaces constructors by functions

- capture pattern as *foldr*

    $foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [\,a\,] \rightarrow b$
    $foldr \, op \, e \, [\,] \quad\quad = e$
    $foldr \, op \, e \, (x : xs) = x \, \text{'op'} \, foldr \, op \, e \, xs$

2.8   Examples of fold right

- many examples:

    sum $= foldr\ (+)\ 0$
    id $= foldr\ (:)\ [\ ]$
    $length$ $= foldr\ (\lambda x\ n \rightarrow 1 + n)\ 0$
    $map\ f$ $= foldr\ ((:) \circ f)\ [\ ]$
    $concat$ $= foldr\ (\!+\!\!+)\ [\ ]$
    $reverse = foldr\ snoc\ [\ ]$ **where** $snoc\ x\ ys = ys \!+\!\!+ [x]$
    $xs \!+\!\!+ ys = foldr\ (:)\ ys\ xs$

- right-to-left computation
- operator may be associative $(+, \!+\!\!+)$ but need not $(:, snoc)$

2.8   Sorting

- given

    $insert :: (Ord\ a) \Rightarrow a \rightarrow [a] \rightarrow [a]$
    $insert\ x\ [\ ] \quad = [x]$
    $insert\ x\ (y : ys)$
    $\quad | \ x \leqslant y \quad = x : y : ys$
    $\quad | \ otherwise = y : insert\ x\ ys$

- we have

    $insertSort :: (Ord\ a) \Rightarrow [a] \rightarrow [a]$
    $insertSort = foldr\ insert\ [\ ]$

2.8   Fold left

- not every list function is a *foldr*
- eg *decimal* $[1, 2, 3] = 123$
- efficient algorithm using *Horner's rule*:

  $$decimal\,[x, y, z] = 10 \times (10 \times (10 \times 0 + x) + y) + z$$

- left-to-right computation — hence *foldl*

  $$foldl\;op\;e\,[x, y, z] = ((e\;{`op`}\;x)\;{`op`}\;y)\;{`op`}\;z$$

- definition

  $foldl :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
  $foldl\;op\;e\;[\,] \qquad\;\; = e$
  $foldl\;op\;e\;(x : xs) = foldl\;op\;(e\;{`op`}\;x)\;xs$

  then $decimal = foldl\;(\lambda n\;d \rightarrow 10 \times n + d)\;0$

2.8   Accumulating parameter

- recall *reverse* program

    reverse :: [a] → [a]
    reverse = *foldr* (λx ys → ys ++ [x]) [ ]

- another definition

    reverse' :: [a] → [a]
    reverse' = *foldl* (*flip* (:)) [ ]

- (now what is complexity?)
- second argument of *foldl* is an *accumulating parameter*

2.8   Fold over non-empty lists

- consider computing maximum element in a list of numbers
- *maximum* $(x : xs) = x$ '*max*' *maximum xs* suggests *foldr*
- but what is the starting value *e*?
- *e* should be *maximum* [ ]; also require x '*max*' $e = x$
- want to choose *e* to be smallest possible value
- could restrict to context *Bounded*, with *e* = *minBound*
- alternatively, don't use on empty list!

- capture pattern via two folds on non-empty lists

    $foldr1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
    $foldr1\ op\ (x : xs)$
       $|\ null\ xs\quad = x$
       $|\ otherwise = x\ `op`\ foldr1\ op\ xs$

    $foldl1 :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
    $foldl1\ op\ (x : xs) = foldl\ op\ x\ xs$

- eg

    $foldr1\ op\ [x, y, z] = x\ `op`\ (y\ `op`\ z)$
    $foldl1\ op\ [x, y, z] = (x\ `op`\ y)\ `op`\ z$

- now $maximum = foldr1\ max = foldl1\ max$
- we have defined our own customized loop scheme!

2.8 Scan

- sometimes convenient to apply *foldl* to every initial segment of a list

    scanl op e [x, y, z] = [$e, e\ 'op'\ x, (e\ 'op'\ x)\ 'op'\ y, ((e\ 'op'\ x)\ 'op'\ y)\ 'op'\ z$]

- eg *scanl* (+) 0 computes *prefix sums*
- eg *scanl* (×) 1 [1 . . *n*] computes first *n* + 1 factorials
- start with specification

    scanl :: ($b \to a \to b$) $\to b \to$ [a] $\to$ [b]
    scanl op e = *map* (*foldl op e*) ∘ *inits*

    inits :: [a] $\to$ [[a]]
    inits [ ]   = [[ ]]
    inits ($x : xs$) = [ ] : *map* (x:) (*inits xs*)

(exercise: *inits* matches pattern for *foldr*)

2.8 Efficient scan

- inefficient, as quadratically many applications of *op*

- straightforward to synthesize more efficient implementation

  scanl op e [ ]   = [*e*]
  scanl op e (*x* : *xs*) = *e* : *scanl op* (*op e x*) *xs*

- dually,

  scanr :: (*a* → *b* → *b*) → *b* → [*a*] → [*b*]
  scanr op e = *map* (*foldr op e*) ∘ *tails*

  tails :: [*a*] → [[*a*]]
  tails [ ]   = [[ ]]
  tails (*x* : *xs*) = (*x* : *xs*) : *tails xs*

  has more efficient implementation

  scanr op e = *foldr* (λ*x ys* → *op x* (*head ys*) : *ys*) [*e*]

2.8   Duality: unfold

- so far we have focused on *consumers*
- *producers* are important too
- producers (unfolds) are *dual* to consumers (folds)
- common pattern

    $unfoldr :: (b \to Bool) \to (b \to a) \to (b \to b) \to b \to [a]$
    $unfoldr\ p\ f\ g\ z = \textbf{if}\ p\ z\ \textbf{then}\ [\,]\ \textbf{else}\ f\ z : unfoldr\ p\ f\ g\ (g\ z)$

- *unfoldr* is *dual* to *foldr* (in what way. . . ?)
- relation to OO iterators?

2.8 Examples of unfold

- $[m..n]$ aka *enumFromTo m n*

    enumFromTo :: $(Num\ a, Ord\ a) \Rightarrow a \to a \to [a]$
    enumFromTo m n = *unfoldr* $(>n)$ *id* $(+1)$ *m*

- *map* can also be expressed as an unfold

    map :: $(a \to b) \to ([a] \to [b])$
    map f = *unfoldr null* $(f \circ head)$ *tail*

- another sorting algorithm, selection sort

    selectsort = *unfoldr null minimum deleteMin*
      where *deleteMin* = ...   -- exercise!

2.9 Algebraic datatypes

- eg people with names and ages

     type *Name* = *String*
     type *Age* = *Integer*
     data *Person* = *P Name Age*

- then *P* :: *Name* → *Age* → *Person*
- such *constructor functions* do not simplify, they are in normal form
- so they can be used in pattern-matching:

     showPerson :: *Person* → *String*
     showPerson (*P n a*) = "Name: " + *n* + ", Age: " + *show a*

2.9   Sum types

- datatypes can have multiple *variants*

     data *Maybe a = Just a | Nothing*

- eg *Just* 13 :: *Maybe Int*; so *Just* :: $a \rightarrow$ *Maybe a* and *Nothing* :: *Maybe a*

- useful for modelling exceptions

     safeHead :: [*a*] $\rightarrow$ *Maybe a*
     safeHead [ ]   = *Nothing*
     safeHead (*x* : *xs*) = *Just x*

- the library definition of unfold:

     unfoldr :: ($b \rightarrow$ *Maybe* (*a*, *b*)) $\rightarrow$ ($b \rightarrow$ [*a*])
     unfoldr f z = **case** *f z* **of**
                    Nothing $\rightarrow$ [ ]
                    Just (*x*, *z'*) $\rightarrow$ *x* : *unfoldr f z'*

2.10   Arithmetic expressions

- algebraic datatypes may be recursive too
- eg datatype of arithmetic expressions

    data *Expr = Lit Integer | Add Expr Expr | Mul Expr Expr*

- an arithmetic expressions is either a literal,
  or two expressions added together, or two multiplied

2.10 Constructing expressions

- constructing expressions

    expr1, *expr2* :: *Expr*
    expr1 = *Add* (*Mul* (*Lit* 4) (*Lit* 7)) (*Lit* 11)
    expr2 = *Mul* (*Add* (*Lit* 4) (*Lit* 7)) (*Lit* 11)

- note the difference between *syntax*

    Main⟩ *Add* (*Lit* 4) (*Mul* (*Lit* 7) (*Lit* 11))
    Add (*Lit* 4) (*Mul* (*Lit* 7) (*Lit* 11))

- and *semantics*

    Main⟩ $4 + 7 \times 11$
    81

2.10 *Expr* **definition pattern**

- recursive definitions by pattern-matching

  evaluate :: *Expr* → *Integer*
  evaluate (*Lit i*)   = *i*
  evaluate (*Add e1 e2*) = *evaluate e1* + *evaluate e2*
  evaluate (*Mul e1 e2*) = *evaluate e1* × *evaluate e2*

- the evaluator essentially replaces syntax (*Add* and *Mul*) by semantics (+ and ×)

2.10 *Expr* **definition pattern**

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Expr \rightarrow S$
- *step 1:* solve the problem for literals

    $f\,(Lit\,n)\,=\,...\,n\,...$

- *step 2:* solve the problem for addition;
  assume that you already have the solution for x and y at hand;
  extend the intermediate solution to a solution for $Add\,x\,y$

    $f\,(Lit\,n)\quad\ =\ ...\,n\,...$
    $f\,(Add\,x\,y)\ =\ ...\,x\,...\,y\,...\,f\,x\,...\,f\,y\,...$

  you have to program only a *step*

- *step 3:* do the same for $Mul\,x\,y$

    $f\,(Lit\,n)\quad\ =\ ...\,n\,...$
    $f\,(Add\,x\,y)\ =\ ...\,x\,...\,y\,...\,f\,x\,...\,f\,y\,...$
    $f\,(Mul\,x\,y)\ =\ ...\,x\,...\,y\,...\,f\,x\,...\,f\,y\,...$

2.10 Naturals

- *Peano* definition of natural numbers (non-negative integers)

    data *Nat* = *Zero* | *Succ Nat*

- every natural is either *Zero* or the *Succ*essor of a natural
- eg *Succ* (*Succ* (*Succ Zero*)) corresponds to 3
- extraction

    nat2int :: *Nat* → *Integer*
    nat2int Zero   = 0
    nat2int (*Succ n*) = 1 + *nat2int n*

- addition

    plus :: *Nat* → *Nat* → *Nat*
    plus Zero   *n* = *n*
    plus (*Succ m*) *n* = *Succ* (*plus m n*)

    (does this look familiar?)

2.10   Peano definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Nat \rightarrow S$
- *step 1:* solve the problem for *Zero*

    $f\ Zero = \ ...$

- *step 2:* solve the problem for *Succ n*;
  assume that you already have the solution for *n* at hand;
  extend the intermediate solution to a solution for *Succ n*

    $f\ Zero \qquad = \ ...$
    $f\ (Succ\ n) = \ ...\ n\ ...\ f\ n\ ...$

  you have to program only a *step*

- put on your problem-solving glasses
- (exercise: multiplication, exponentiation)

2.10   Lists

- built-in type of lists is not special (has only special syntax)

    data *List a = Nil | Cons a* (*List a*)

- eg [1, 2, 3] or 1 : 2 : 3 : [ ] corresponds to *Cons* 1 (*Cons* 2 (*Cons* 3 *Nil*))
- recursive definitions by pattern-matching

    mapList :: (*a → b*) → (*List a → List b*)
    mapList f Nil   = *Nil*
    mapList f (*Cons x xs*) = *Cons* (*f x*) (*mapList f xs*)

2.10   List definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: List\ P \to S$
- *step 1:* solve the problem for the empty list

    $f\ Nil = \ ...$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for *xs* at hand;
  extend the intermediate solution to a solution for *Cons x xs*

    $f\ Nil \qquad\quad = \ ...$
    $f\ (Cons\ x\ xs) = \ ...\ x\ ...\ xs\ ...\ f\ xs\ ...$

    you have to program only a *step*

- put on your problem-solving glasses

2.10   Binary search trees

- definition

    data *BST k v = Leaf | Branch (BST k v) k v (BST k v)*

- eg insertion

    insert :: Ord k ⇒ k → v → BST k v → BST k v
    insert k v Leaf = Branch Leaf k v Leaf   -- insert at leaf
    insert k v (Branch l k' v' r)
       | k < k'   = Branch (insert k v l) k' v' r   -- insert to the left
       | k > k'   = Branch l k' v' (insert k v r)   -- insert to the right
       | otherwise = Branch l k v r   -- overwrite the root

2.10 Binary search tree definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: BST\ K\ V \to S$
- *step 1:* solve the problem for the empty tree

    $f\ Leaf = \ldots$

- *step 2:* solve the problem for non-empty trees;
  assume that you already have solutions for l and r at hand;
  extend the intermediate solution to a solution for $Branch\ l\ k\ v\ r$

    $f\ Leaf \qquad\qquad = \ldots$
    $f\ (Branch\ l\ k\ v\ r) = \ldots\ k\ \ldots\ v\ \ldots\ l\ \ldots\ f\ l\ \ldots\ r\ \ldots\ f\ r\ \ldots$

    you have to program only a *step*

- put on your problem-solving glasses