CS 457/557: Functional Languages

Lists and Algebraic Datatypes

Mark P Jones
Portland State University

Why Lists?

- Lists are a heavily used data structure in many functional programs

- Special syntax is provided to make programming with lists more convenient

- Lists are a special case / an example of:
  - An algebraic datatype (coming soon)
  - A parameterized datatype (coming soon)
  - A monad (coming, but a little later)

Naming Convention:

- We often use a simple naming convention:
- If a typical value in a list is called x, then a typical list of such values might be called xs (i.e., the plural of x)
- … and a list of lists of values called x might be called xss
- A simple convention, minimal clutter, and a useful mnemonic too!

Prelude Functions:

```
(++)       :: [a] -> [a] -> [a]
reverse    :: [a] -> [a]
take       :: Int -> [a] -> [a]
drop       :: Int -> [a] -> [a]
takeWhile  :: (a -> Bool) -> [a] -> [a]
dropWhile  :: (a -> Bool) -> [a] -> [a]
replicate  :: Int -> a -> [a]
iterate    :: (a -> a) -> a -> [a]
repeat     :: a -> [a]
…
```

Constructor Functions:

- What if you can't find a function in the prelude that will do what you want to do?
- Every list takes the form:
  - [], an empty list
  - (x:xs), a non-empty list whose first element is x, and whose tail is xs
- Equivalently: the list type has two constructor functions:
  - The constant [] :: [a]
  - The operator (:) :: a -> [a] -> [a]
- Using "pattern matching", we can also take lists apart …

Functions on Lists:

```
null           :: [a] -> Bool
null []        = True
null (x:xs)    = False

head  :: [a] -> a
head (x:xs) = x

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

Recursive Functions:

```
last                :: [a] -> a
last (x:[])         = x
last (x:y:xs)       = last (y:xs)

init                :: [a] -> [a]
init (x:[])         = []
init (x:y:xs)       = x : init (y:xs)

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

… continued:

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

in List library

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

user defined

Why Does This Work?

- What does it mean to say that [] and (:) are the constructor functions for lists?
- No Junk: every list value is equal either to [], or else to a list of the form (x:xs) (ignoring non-termination, for now)
- No Confusion: if $x \neq y$, or $xs \neq ys$, then $x:xs \neq y:ys$
- A pair of equations
  ```
  f []      = …
  f (x:xs)  = …
  ```
  defines a unique function on list values

Algebraic Datatypes:

Algebraic Datatypes:

- Booleans and Lists are both examples of "algebraic datatypes":
- No Junk:
  - Every Boolean value can be constructed using either False or True
  - Every list can be described using (a combination of) [] and (:)
- No Confusion:
  - True $\neq$ False
  - [] $\neq$ (x:xs) and if (x:xs)=(y:ys), then x=y and xs=ys

In Haskell Notation:

data Bool = False | True
introduces:

- A type, Bool
- A constructor function, False :: Bool
- A constructor function, True :: Bool

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

- A type, List t, for each type t
- A constructor function, Nil :: List a
- A constructor function, Cons :: a -> List a -> List a

More Enumerations:

```
data Rainbow = Red | Orange | Yellow
             | Green | Blue  | Indigo | Violet
```

introduces:

- A type, Rainbow
- A constructor function, Red :: Rainbow
- …
- A constructor function, Violet :: Rainbow

No Junk: Every value of type Rainbow is one of the above seven colors

No Confusion: The seven colors above are distinct values of type Rainbow

More Recursive Types:

```
data Shape = Circle Radius
           | Polygon [Point]
           | Transform Transform Shape
data Transform
           = Translate Point
           | Rotate Angle
           | Compose Transform Transform
```

introduces:

- Two types, Shape and Transform
- Circle :: Radius -> Shape
- Polygon :: [Point] -> Shape
- Transform :: Transform -> Shape -> Shape
- ...

More Parameterized Types:

data Maybe a = Nothing | Just a

introduces:

- A type, Maybe t, for each type t
- A constructor function, Nothing :: Maybe a
- A constructor function, Just :: a -> Maybe a

data Pair a b = Pair a b

introduces

- A type, Pair t s, for any types t and s
- A constructor function Pair :: a -> b -> Pair a b

General Form:

Algebraic datatypes are introduced by top-level definitions of the form:

$$\textbf{data}\ T\ a_1\ \ldots\ a_n = c_1 \mid \ldots \mid c_m$$

where:

- T is the type name (must start with a capital letter)
- $a_1, \ldots, a_n$ are (distinct) (type) arguments/parameters/ variables (must start with lower case letter) ($n \geq 0$)
- Each of the c_i is an expression $F_i\ t_1\ \ldots\ t_k$ where:
  - $t_1, \ldots, t_k$ are type expressions that (optionally) mention the arguments $a_1, \ldots, a_n$
  - F_i is a new constructor function $F_i :: t_1 \rightarrow \ldots \rightarrow t_p \rightarrow T\ a_1\ \ldots\ a_n$
  - The <u>arity</u> of F_i, $k \geq 0$

Quite a lot for a single definition!

No Junk and Confusion:

- The key properties that are shared by all algebraic datatypes:
  - No Junk: Every value of type $T\ a_1 \ldots a_n$ can be written in the form $F_i\ e_1 \ldots e_k$ for some choice of constructor F_i and (appropriately typed) arguments $e_1, \ldots, e_k$
  - No Confusion: Distinct constructors or distinct arguments produce distinct results

- These are fundamental assumptions that we make when we write and/or reason about functional programs.

Pattern Matching:

- In addition to introducing a new type and a collection of constructor functions, each data definition also adds the ability to pattern match over values of the new type
- For example, given

```
data Maybe a = Nothing | Just a
```

then we can define functions like the following:

```
orElse                 :: Maybe a -> a -> a
Just x   `orElse` y  = x
Nothing `orElse` y  = y
```

Pattern Matching & Substitution:

- The result of a pattern match is either:
  - A failure
  - A success, accompanied by a substitution that provides a value for each of the values in the pattern

- Examples:
  - [] does not match the pattern (x:xs)
  - [1,2,3] matches the pattern (x:xs) with x=1 and xs=[2,3]

Patterns:

More formally, a pattern is either:

- An identifier
  - Matches any value, binds result to the identifier

- An underscore (a "wildcard")
  - Matches any value, discards the result

- A constructed pattern of the form $C\ p_1 \ldots p_n$, where C is a constructor of arity n and $p_1, \ldots, p_n$ are patterns of the appropriate type
  - Matches any value of the form $C\ e_1 \ldots e_n$, provided that each of the e_i values matches the corresponding p_i pattern.

Other Pattern Forms:

For completeness:

- "Sugared" constructor patterns:
  - Tuple patterns (p_1,p_2)
  - List patterns $[p_1, p_2, p_3]$
  - Strings, for example: "hi" = ('h' : 'i' : [])
- Labeled patterns
- Numeric Literals:
  - Can be considered as constructor patterns, but the implementation uses equality (==) to test for matches
- "as" patterns, id@pat
- Lazy patterns, ~pat
- (n+k) patterns

Function Definitions:

- In general, a function definition is written as a list of adjacent equations of the form:

  $$f\ p_1\ \ldots\ p_n = rhs$$

  where:
  - f is the name of the function that is being defined
  - $p_1, \ldots, p_n$ are patterns, and rhs is an expression

- All equations in the definition of f must have the same number of arguments (the "<u>arity</u>" of f)

… continued:

- Given a function definition with m equations:

  $f\ p_{1,1}\ \ldots\ p_{n,1} = rhs_1$
  $f\ p_{1,2}\ \ldots\ p_{n,2} = rhs_2$
  …
  $f\ p_{1,m}\ \ldots\ p_{n,m} = rhs_m$

- The value of $f\ e_1\ \ldots\ e_n$ is $S\ rhs_i$, where i is the smallest integer such that the expressions e_j match the patterns $p_{j,i}$ and S is the corresponding substitution.

Guards, Guards!

- A function definition may also include guards (Boolean expressions):

$$\begin{aligned} f\ p_1\ \ldots\ p_n\ & \mid g_1 = rhs_1 \\ & \mid g_2 = rhs_2 \\ & \mid g_3 = rhs_3 \end{aligned}$$

- An expression $f\ e_1\ \ldots\ e_n$ will only match an equation like this if all of the e_i match the corresponding p_i and, in addition, at least one of the guards g_j is True
- In that case, the value is $S\ rhs_j$, where j is the smallest index such that g_j is True
- (The prelude defines otherwise = True :: Bool for use in guards.)

Where Clauses:

- A function definition may also a where clause:

  $f\ p_1\ \ldots\ p_n = rhs$

  where decls

- Behaves like a let expression:

  $f\ p_1\ \ldots\ p_n =$ **let** decls **in** rhs

- Except that where clauses can scope across guards:

  $f\ p_1\ \ldots\ p_n \quad |\ g_1 = rhs_1$

  $|\ g_2 = rhs_2$

  $|\ g_3 = rhs_3$

  where decls

- Variables bound here in decls can be used in any of the g_i or rhs_i

Example: filter

```
filter              :: (a -> Bool) -> [a] -> [a]
filter p []         = []
filter p (x:xs)
    | p x           = x : rest
    | otherwise     = rest
      where rest = filter p xs
```

Example: Binary Search Trees

```
data Tree             = Leaf | Fork Tree Int Tree

insert                :: Int -> Tree -> Tree
insert n Leaf         = Fork Leaf n Leaf
insert n (Fork l m r)
    | n <= m          = Fork (insert n l) m r
    | otherwise  = Fork l m (insert n r)

lookup                :: Int -> Tree -> Bool
lookup n Leaf         = False
lookup n (Fork l m r)
        | n < m        = lookup n l
        | n > m        = lookup n r
        | otherwise   = True
```

Case Expressions:

- Case expressions can be used for pattern matching:

  case e **of**
  $p_1 \text{ -> } e_1$
  $p_2 \text{ -> } e_2$
  …
  $p_n \text{ -> } e_n$

- Equivalent to:

  let $f\ p_1 = e_1$
  $f\ p_2 = e_2$
  …
  $f\ p_n = e_n$
  in $f\ e$

... continued:

- Guards and where clauses can also be used in case expressions:

```
filter p xs = case xs of
                []                  -> []
                (x:xs) | p x        -> x:ys
                       | otherwise  -> ys
                  where ys = filter p xs
```

If Expressions:

- If expressions can be used to test Boolean values:

  if e **then** e_1 **else** e_2

- Equivalent to:

  case e **of**

  True -> e_1

  False -> e_2

Summary:

- Algebraic datatypes can support:
  - Enumeration types
  - Parameterized types
  - Recursive types
  - Products (composite/aggregate values); and
  - Sums (alternatives)
- Type constructors, Constructor functions, Pattern matching
- Unifying features: No junk, no confusion!

Example: transpose

```
transpose                   :: [[a]] -> [[a]]
transpose []                = []
transpose ([] : xss)        = transpose xss
transpose ((x:xs) : xss)
                    = (x : [h | (h:t) <- xss])
                        : transpose (xs : [ t | (h:t) <- xss])
```

Example:

```
transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]
```

Example: say

```
Say> putStr (say "hello")

H   H  EEEEE  L      L       OOO
H   H  E      L      L      O   O
HHHHH  EEEEE  L      L      O   O
H   H  E      L      L      O   O
H   H  EEEEE  LLLLL  LLLLL   OOO

Say>
```

… continued:

```
say = ('\n':)
      . unlines
      . map (foldr1 (\xs ys->xs++"  "++ys))
      . transpose
      . map picChar

picChar 'A' = ["  A  ",
               " A A ",
               "AAAAA",
               "A   A",
               "A   A"]
```

etc…

Composition and Reuse:

```
Say> (putStr . concat . map say . lines . say) "A"
                A
               A A
              AAAAA
              A   A
              A   A

         A             A
        A A           A A
       AAAAA         AAAAA
       A   A         A   A
       A   A         A   A

  A      A      A      A      A
 A A    A A    A A    A A    A A
AAAAA  AAAAA  AAAAA  AAAAA  AAAAA
A   A  A   A  A   A  A   A  A   A
A   A  A   A  A   A  A   A  A   A

  A                           A
 A A                         A A
AAAAA                       AAAAA
A   A                       A   A
A   A                       A   A

  A                           A
 A A                         A A
AAAAA                       AAAAA
A   A                       A   A
A   A                       A   A

Say>
```