Lists and Algebraic Types

Arvind
Laboratory for Computer Science
M.I.T.

October 2, 2002

http://www.csg.lcs.mit.edu/6.827

Algebraic types

- Algebraic types are *tagged unions of products*
- Example

```
data Shape = Line        Pnt   Pnt
           | Triangle    Pnt   Pnt   Pnt
           | Quad        Pnt   Pnt   Pnt   Pnt
```

keyword

"union"

new type

"products" (fields)

- *new "constructors" (a.k.a. "tags", "disjuncts", "summands")*
- *a k-ary constructor is applied to k type expressions*

Constructors are functions

- Constructors can be used as functions to *create* values of the type

```
let
    l1 :: Shape
    l1 = Line  e1  e2

    t1 :: Shape = Triangle  e3  e4  e5
    q1 :: Shape = Quad  e6  e7  e8  e9
in
    ...
```

where each "eJ" is an expression of type "Pnt"

Pattern-matching on algebraic types

- *Pattern-matching* is used to examine values of an algebraic type

```
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
                Line     p1 p2       -> p1
                Triangle p3 p4 p5    -> p3
                Quad     p6 p7 p8 p9 -> p6
```

- A pattern-match has two roles:
  - A test: "does the given value match this pattern?"
  - Binding ("if the given value matches the pattern, *bind* the variables in the pattern to the corresponding parts of the value")

Pattern-matching *scope & don't cares*

- Each clause starts a new *scope*: can re-use bound variables
- Can use "don't cares" for bound variables

```
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
                Line     p1  _       -> p1
                Triangle p1  _  _    -> p1
                Quad     p1  _  _  _ -> p1
```


Pattern-matching *more syntax*

- Functions can be defined directly using pattern-matching

```
anchorPnt :: Shape -> Pnt
anchorPnt (Line     p1 _)     = p1
anchorPnt (Triangle p1 _ _)   = p1
anchorPnt (Quad     p1 _ _ _) = p1
```

- Pattern-matching can be used in list comprehensions (*later)*

```
(Line p1 p2) <- shapes
```

Pattern-matching *Type safety*

- Given a "Line" object, it is impossible to read "the field corresponding to the third point in a Triangle object“ because:
  - all unions are *tagged* unions
  - fields of an algebraic type can only be examined *via* pattern-matching

Special syntax

- Function type constructor

  `Int -> Bool`

  Conceptually:

  `Function  Int  Bool`

  i.e., the arrow is an "infix" type constructor

- Tuple type constructor

  `(Int, Bool)`

  Conceptually:

  `Tuple2  Int  Bool`

  Similarly for Tuple3, ...

Type Synonyms

```
data Point = Point Int Int
```

versus

```
type Point = (Int,Int)
```

Type Synonyms do not create new types. It is just a convenience to improve readability.

```
move :: Point -> (Int,Int) -> Point
move (Point x y) (sx,sy) =
            Point (x + sx) (y + sy)
```

versus

```
move (x,y) (sx,sy) =
                  (x + sx, y + sy)
```


Abstract Types

A rational number is a pair of integers but suppose we want to express it in the reduced form only. Such a restriction cannot be enforced using an algebraic type.

```
module Rationalpackage
      (Rational,rational,rationalParts) where
   data Rational = RatCons Int Int

   rational :: Int -> Int -> Rational
   rational x y = let
                     d = gcd x y
                  in RatCons (x/d) (y/d)

   rationalParts :: Rational -> (Int,Int)
   rationalParts (RatCons x y)= (x,y)
```

No pattern matching on abstract data types

Examples of Algebraic types

```
data Bool = False | True
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
data Maybe a  = Nothing | Just a
data List a = Nil | Cons a (List a)
data Tree a = Leaf a | Node (Tree a) (Tree a)
data Tree' a b = Leaf' a
               | Nonleaf' b (Tree' a b) (Tree' a b)

data Course  = Course String Int String (List Course)
```

name number description pre-reqs

Lists

```
data  List t = Nil | Cons t (List t)
```

A list data type can be constructed in two different ways:

an empty list **Nil**
or a non-empty list **Cons x xs**

the first element
the rest of the elements

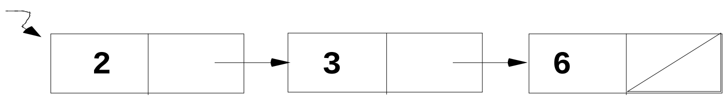
- All elements of a list have *the same type*
- The list type is *recursive* and *polymorphic*

Infix notation

$\boxed{\texttt{Cons x xs} \equiv \texttt{x:xs}}$

`2:3:6:Nil` ≡ `2:(3:(6:Nil))` ≡ `[2,3,6]`

This list may be visualized as follows:



Simple List Programs

Sum of numbers in a list

```
sum []        = 0
sum (x:xs)    =                        ?
```

Last element in a list

```
last []       = x
last (x:xs)   =                        ?
```

All but the last element in a list

```
init []       = []
init (x:xs)   =                        ?
```

What do the following do?

```
init (a:xs)
(a:(init xs))
```

Example: Split a list

```
data Token = Word String | Number Int
```

Split a list of tokens into two lists - a list words and a list of numbers.

```
split :: (List Token)->
               ((List String),(List Int))

split []     = ([],[])
split (t:ts) =                        ?
```


Higher-order List abstractions

```
map f []          = []
map f (x:xs)      =                         ?


foldl f z []      = z
foldl f z (x:xs)  =                         ?


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


filter p []       = []
filter p (x:xs)   =                         ?
```

Using maps and folds

1. Write **`sum`** in terms of **`fold`**

2. Write **`split`** using **`foldr`**

```
split :: (List Token) -> ((List String),(List Int))
```

3. What does function **`fy`** do?

```
fy xys = map second xys
second (x,y) = y
fy ::
```


Flattening a List of Lists

```
append :: (List t) -> (List t) -> (List t)
append [] ys      = ys
append (x:xs) ys  = (x:(append xs ys))
```

```
flatten :: (List (List t)) -> (List t)
flatten []        = []
flatten (xs:xss) = append xs (concat xss)
```

Zipping two lists

```
zipWith :: (tx -> ty -> tz) ->
                  (List tx) ->
                  (List ty) -> (List tz)

zipWith f [] []         = []
zipWith f (x:xs) (y:ys) =                    ?
```

What does `f` do?

```
f xs = zipWith append xs (init ([]:xs))
```

Suppose `xs` is:

x_0 , x_1 , x_2 , ... , x_n

Arithmetic Sequences: Special Lists

`[1 .. 4]`	≡	`[1,2,3,4]`
`[1,3 .. 10]`	≡	`[1,3,5,7,9]`
`[5,4 .. 1]`	≡	`[5,4,3,2,1]`
`[5,5 .. 10]`	≡	`[5,5,5,...]` ?
`[5 .. ]`	≡	`[5,6,7,...]` ?

List Comprehensions
a convenient syntax

```
[ e |  gen, gen, ...]
```

Examples

```
[ f x | x <- xs ]
```
means `map f xs`

```
[ x | x <- xs, (p x)]
```
means `filter p xs`

```
[ f x y | x <- xs, y <- ys ]
```
means the list
```
[(f x1 y1),...(f x1 yn),
 (f x2 y1),......(f xm yn)]
```
which is defined by
```
flatten (map (\ x -> (map (\ y -> e) ys) xs))
```


Three-Partitions

Generate a list containing all three-partitions (nc1, nc2, nc3) of a number m, such that

- nc1 $\leq$ nc2 $\leq$ nc3
- nc1 + nc2 + nc3 = m

```
three_partitions m =

  [ (nc1,nc2,nc3) | nc1 <- [0..m],
                    nc2 <- [0..m],
```

?

Efficient Three-Partitions

```
three_partitions m =

  [ (nc1,nc2,nc3) | nc1 <- [0..floor(m/3)],
                    nc2 <-
```

?

The Power of List Comprehensions

```
[ (i,j) | i <- [1..n], j <- [1..m] ]
```

using map

```
point i j     = (i,j)

points i      = map (point i) [1..m]

all_points    = map points [1..n]
```

?

Infinite Data Structures

1.
```
ints_from i  = i:(ints_from (i+1))

nth n (x:xs) = if n == 1 then x
               else nth (n - 1) xs
   nth 50 (ints_from 1) -->            ?
```

2.
```
ones = 1:ones
   nth 50 ones  -->                    ?
```

3.
```
xs = [ f x | x <- a:xs]
   nth 10 xs    -->                    ?
```

These are well defined but *deadly* programs in pH. You will get an answer but the program may *not* terminate.

Primes: *The Sieve of Eratosthenes*

```
primes = sieve [2..]

sieve (x:xs) = x:(sieve (filter (p x) xs))

p x y = (y mod x) ≠ 0

      nth 100 primes
```