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
```

- 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

```haskell
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

```haskell
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 \textit{scope} & \textit{don't cares}

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

\begin{verbatim}
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
  Line p1 _ -> p1
  Triangle p1 _ _ -> p1
  Quad p1 _ _ _ -> p1
\end{verbatim}

Pattern-matching \textit{more syntax}

- Functions can be defined directly using pattern-matching

\begin{verbatim}
anchorPnt :: Shape -> Pnt
anchorPnt (Line p1 _) = p1
anchorPnt (Triangle p1 _) = p1
anchorPnt (Quad p1 _) = p1
\end{verbatim}

- Pattern-matching can be used in list comprehensions (\textit{later})

\begin{verbatim}
(Line p1 p2) <- shapes
\end{verbatim}
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**
  
  \[ \text{Int} \to \text{Bool} \]

  Conceptually:
  
  \[ \text{Function} \; \text{Int} \; \text{Bool} \]

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

- **Tuple type constructor**
  
  \[ (\text{Int}, \text{Bool}) \]

  Conceptually:
  
  \[ \text{Tuple2} \; \text{Int} \; \text{Bool} \]

  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 Rational
  (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

\[\text{data} \ \text{Bool} = \text{False} \mid \text{True}\]
\[\text{data} \ \text{Day} = \text{Sun} \mid \text{Mon} \mid \text{Tue} \mid \text{Wed} \mid \text{Thu} \mid \text{Fri} \mid \text{Sat}\]
\[\text{data} \ \text{Maybe} \ a = \text{Nothing} \mid \text{Just} \ a\]
\[\text{data} \ \text{List} \ a = \text{Nil} \mid \text{Cons} \ a \ (\text{List} \ a)\]
\[\text{data} \ \text{Tree} \ a = \text{Leaf} \ a \mid \text{Node} \ (\text{Tree} \ a) \ (\text{Tree} \ a)\]
\[\text{data} \ \text{Tree'} \ a \ b = \text{Leaf'} \ a \mid \text{Nonleaf'} \ b \ (\text{Tree'} \ a \ b) \ (\text{Tree'} \ a \ b)\]
\[\text{data} \ \text{Course} = \text{Course} \ \text{String} \ \text{Int} \ \text{String} \ (\text{List} \ \text{Course})\]

Lists

\[\text{data} \ \text{List} \ t = \text{Nil} \mid \text{Cons} \ t \ (\text{List} \ t)\]

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Description</th>
<th>Pre-Reqs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Infix notation

\[ \text{Cons } x \, \text{xs} \equiv x:xs \]

\[ 2:3:6:\text{Nil} \equiv 2:(3:(6:\text{Nil})) \equiv [2,3,6] \]

This list may be visualized as follows:

```
2 3 6  
```

Simple List Programs

Sum of numbers in a list

\[
\begin{align*}
\text{sum } [] & = 0 \\
\text{sum } (x:xs) & = ?
\end{align*}
\]

Last element in a list

\[
\begin{align*}
\text{last } [] & = x \\
\text{last } (x:xs) & = ?
\end{align*}
\]

All but the last element in a list

\[
\begin{align*}
\text{init } [] & = [] \\
\text{init } (x:xs) & = ?
\end{align*}
\]

What do the following do?

\[
\begin{align*}
\text{init } (a:xs) \\
(a:(\text{init } xs))
\end{align*}
\]
Example: Split a list

```haskell
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

```haskell
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 \texttt{sum} in terms of \texttt{fold}

2. Write \texttt{split} using \texttt{foldr}
   \begin{equation*}
   \text{split :: (List Token) -> ((List String),(List Int))}
   \end{equation*}

3. What does function \texttt{fy} do?
   \begin{equation*}
   \text{fy xys = map second xys}
   \end{equation*}
   \begin{equation*}
   \text{second (x,y) = y}
   \end{equation*}
   \begin{equation*}
   \text{fy :: }
   \end{equation*}

Flattening a List of Lists

\begin{equation*}
\text{append :: (List t) -> (List t) -> (List t)}
\end{equation*}
\begin{equation*}
\text{append [] ys = ys}
\end{equation*}
\begin{equation*}
\text{append (x:xs) ys = (x:(append xs ys))}
\end{equation*}

\begin{equation*}
\text{flatten :: (List (List t)) -> (List t)}
\end{equation*}
\begin{equation*}
\text{flatten [] = []}
\end{equation*}
\begin{equation*}
\text{flatten (xs:xss) = append xs (concat xss)}
\end{equation*}
Zipping two lists

\[
\text{zipWith :: (tx -> ty -> tz) -> (List tx) -> (List ty) -> (List tz)}
\]

\[
\begin{align*}
\text{zipWith } f \; [] \; [] &= [] \\
\text{zipWith } f \; (x:xs) \; (y:ys) &= \\
\end{align*}
\]

What does \( f \) do?

\[
f \; xs = \text{zipWith append } \; xs \; (\text{init } ([]:xs))
\]

Suppose \( xs \) is:

\[
x_0, x_1, x_2, \ldots, x_n
\]

Arithmetic Sequences: Special Lists

\[
\begin{align*}
[1 \ldots 4] & \equiv [1,2,3,4] \\
[1,3 \ldots 10] & \equiv [1,3,5,7,9] \\
[5,4 \ldots 1] & \equiv [5,4,3,2,1] \\
[5,5 \ldots 10] & \equiv [5,5,5,\ldots] \\
[5 \ldots ] & \equiv [5,6,7,\ldots]
\end{align*}
\]
List Comprehensions

*a convenient syntax*

\[ \{ e \mid \text{gen, gen, ...} \} \]

Examples

\[ \{ f \, x \mid x \leftarrow xs \} \]
means \( \text{map} \, f \, \text{xs} \)

\[ \{ x \mid x \leftarrow xs, (p \, x) \} \]
means \( \text{filter} \, p \, \text{xs} \)

\[ \{ f \, x \, y \mid x \leftarrow xs, y \leftarrow ys \} \]
means the list
\[ \{(f \, x_1 \, y_1), \ldots, (f \, x_1 \, y_n), \]
\[ (f \, x_2 \, y_1), \ldots, (f \, x_m \, y_n)\] 
which is defined by
\( \text{flatten} \, (\text{map} \, (\lambda \, x \rightarrow (\text{map} \, (\lambda \, y \rightarrow e) \, \text{ys}) \, \text{xs})) \)

Three-Partitions

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

- \(nc1 < nc2 < nc3\)
- \(nc1 + nc2 + nc3 = m\)

\(\text{three_partitions} \, m = \)
\[ \{ (nc1, nc2, nc3) \mid nc1 \leftarrow [0..m], \]
\[ nc2 \leftarrow [0..m], \]
\[ \} \)
Efficient Three-Partitions

\[
\text{three_partitions } m = \\
[ (nc1, nc2, nc3) \mid nc1 \leftarrow [0..\text{floor}(m/3)], \,
nc2 \leftarrow 
\]

The Power of List Comprehensions

\[
[ (i, j) \mid i \leftarrow [1..n], j \leftarrow [1..m] ]
\]

using map

\[
\begin{align*}
\text{point } i \ j & \quad = (i, j) \\
\text{points } i & \quad = \text{map } \text{point } i \ [1..m] \\
\text{all_points} & \quad = \text{map } \text{points} \ [1..n]
\end{align*}
\]
Infinite Data Structures

1. \[ \text{ints_from } i = i: (\text{ints_from } (i+1)) \]
   \[
   \text{nth } n \ (x:xs) = \begin{cases} 
   x & \text{if } n == 1 \\
   \text{else } \text{nth} \ (n - 1) \ xs & \end{cases}
   \]
   \[
   \text{nth 50 \ (ints_from 1)} \rightarrow \quad ?
   \]

2. \[ \text{ones} = 1: \text{ones} \]
   \[
   \text{nth 50 \ ones} \rightarrow \quad ?
   \]

3. \[ \text{xs} = \{ \ f \ x \ | \ x \leftarrow \text{a:xs} \} \]
   \[
   \text{nth 10 \ xs} \rightarrow \quad ?
   \]

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

\[
\text{primes} = \text{sieve} \ [2..]
\]

\[
\text{sieve} \ (x:xs) = x:\ (\text{sieve} \ (\text{filter} \ (p \ x) \ xs))
\]

\[
p \ x \ y = (y \mod x) \neq 0
\]

\[
\text{nth 100 \ primes}
\]