Programming with Arrays

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 10

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

Pattern Matching
Pattern Matching: Syntax & Semantics

Let us represent a case as \( (\text{case } e \text{ of } C) \)
where \( C \) is
\[
C = P \rightarrow e \mid (P \rightarrow e) , C
\]
\[
P = x \mid \text{CN} \mid \text{CN}_k(P_1, \ldots, P_k)
\]

The rewriting rules for a case may be stated as follows:

\[
(\text{case } e \text{ of } P \rightarrow e_1, C)
\]
\[
\Rightarrow e_1 \quad \text{if } \text{match}(P, e)
\]
\[
\Rightarrow \quad \text{if } \lnot \text{match}(P, e)
\]

\[
(\text{case } e \text{ of } P \rightarrow e_1)
\]
\[
\Rightarrow e_1 \quad \text{if } \text{match}(P, e)
\]
\[
\Rightarrow \quad \text{if } \lnot \text{match}(P, e)
\]

The match Function

\[
P = x \mid \text{CN} \mid \text{CN}_k(P_1, \ldots, P_k)
\]

\[
\text{match}[[x, t]] = \text{True}
\]
\[
\text{match}[[\text{CN}_0, t]] = \text{CN}_0 == \text{tag}(t)
\]
\[
\text{match}[[\text{CN}_k(P_1, \ldots, P_k), t]] =
\]
\[
\quad \text{if } \text{tag}(t) == \text{CN}_k \quad \text{then}
\]
\[
\quad \quad \quad \text{match}[[P_1, \text{proj}_1(t)]] \&\&
\]
\[
\quad \quad \quad .
\]
\[
\quad \quad \quad .
\]
\[
\quad \quad \quad \text{match}[[P_k, \text{proj}_k(t)]]
\]
\[
\quad \text{else}
\]
\[
\text{False}
\]
### pH Pattern Matching

\[
\text{TE}[(\text{case } e \text{ of } C)] = \\
(\text{let } t = e \text{ in } \text{TC}[[t, C]])
\]

\[
\text{TC}[[t, (P \rightarrow e)]] = \\
\quad \text{if } \text{match}[[P, t]]
\quad \quad \text{then } (\text{let } \text{bind}[[P, t]] \text{ in } e)
\quad \quad \text{else error "match failure"}
\]

\[
\text{TC}[[t, ((P \rightarrow e), C)]] = \\
\quad \text{if } \text{match}[[P, t]]
\quad \quad \text{then } (\text{let } \text{bind}[[P, t]] \text{ in } e)
\quad \quad \text{else } \text{TC}[[t, C]]
\]

### Pattern Matching: bind Function

\[
\text{bind}[[x, t]] = x = t
\]

\[
\text{bind}[[\text{CN}_0, t]] = \varepsilon
\]

\[
\text{bind}[[\text{CN}_k(P_1, ..., P_k), t]] = \\
\quad \text{bind}[[P_1, \text{proj}_1(t)]]; \\
\quad \cdot \\
\quad \cdot \\
\quad \text{bind}[[P_k, \text{proj}_k(t)]]
\]
Refutable vs Irrefutable Patterns

Patterns are used in binding for destructuring an expression---but what if a pattern fails to match?

\[
\begin{align*}
\text{let } & (x_1, x_2) = e_1 \\
& x : xs = e_2 \\
& y_1 : y_2 : ys = e_3 \\
\text{in } & e
\end{align*}
\]

what if \( e_2 \) evaluates to []?  
\( e_3 \) to a one-element list?

Should we disallow refutable patterns in bindings?  
Too inconvenient!

Turn each binding into a case expression

Arrays

Cache for function values on a regular subdomain

\[
x = \text{mkArray } (1, n) \ f
\]

Selection: \( x!i \) returns the value of the \( i \)th slot

Bounds: \( \text{(bounds } x) \) returns the tuple containing the bounds
Efficiency is the Motivation for Arrays

\[ f(i) \]

\( f(i) \) is computed once and stored

\( x!i \) is simply a fetch of a precomputed value and should take constant time

A Simple Example

\[ x = \text{mkArray} \ (1,10) \ (\text{plus} \ 5) \]

Type

\[ x :: \ (\text{ArrayI} \ t) \]

assuming

\[ f :: \ \text{Int} \rightarrow \ t \]
Array: An Abstract Data Type

module ArrayI (ArrayI, mkArray, (!), bounds)
where

    infix 9 (!)

data ArrayI t
mkArray ::(Int,Int) -> (Int-> t) -> (ArrayI t)
(!) ::(ArrayI t) -> Int -> t
bounds ::(ArrayI t) -> (Int,Int)

Selection: x!i returns the value of the ith slot
Bounds: (bounds x) returns the tuple containing the bounds

Vector Sum

vs a b = let
    esum i = a!i + b!i
    in
    mkArray (bounds a) esum
Vector Sum - Error Behavior

vs a b = let
  esum i = a!i + b!i
  in
  mkArray (bounds a) esum

Suppose

1. b
2. b
3. b

Map Array

mapArray f a = let
  g i = f (a!i)
  in
  mkArray (bounds a) g

Example: scale a vector, that is, produce b such that \(b_i = s \cdot a_i\)

vscale a s = mapArray \((*) s\) a
Dragging a Shape

Move a k-sided polygon in an n-dimensional space by distance delta

k-sided polygon: An array of points

A point in n-dimensional space
distance delta in n-dimensional space

```
move_shape as delta =
mapArray (scale delta) as
```

http://www.csg.lcs.mit.edu/6.827
**High-level Programming**

```
mapArray2 f a b =
  let
    elem i = f (a!i) (b!i)
  in
  mkArray (bounds a) elem

vs = mapArray2 (+)
vvs = mapArray2 vs
vvvs = mapArray2 vvs
...
```

---

**Fold Array**

```
foldArray a f so =
  let (l, u) = bounds a
    one_fold s i =
      if i > u then s
      else one_fold (f s (a!i)) (i+1)
  in
    one_fold so l

foldArray a (+) 0
foldArray a min infinity
```
Inner Product: $\Sigma a_i b_i$

```haskell
vp a b = let
  elem i = a!i * b!i
  in
  mkArray (bounds a) elem

ip a b = foldArray (vp a b) (+) 0
```

Index Type Class

pH allows arrays to be indexed by any type that can be regarded as having a contiguous enumerable range

```haskell
class Ix a where
  range :: (a,a) \rightarrow [a]
  index :: (a,a) \rightarrow a \rightarrow Int
  inRange :: (a,a) \rightarrow a \rightarrow Bool
```

- `range`: Returns the list of index elements between a lower and an upper bound
- `index`: Given a `range` and an index, it returns an integer specifying the position of the index in the range based on 0
- `inRange`: Tests if an index is in the range
Examples of Index Type

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

An index function may be defined as follows:

index (Sun,Sat) Wed = 3
index (Sun,Sat) Sat = 6
...
```

A two dimensional space may be indexed as followed:

```
index ((li,lj), (ui,uj)) (i,j) =
   (i-li)*((uj-lj)+1) + j - lj
```

This indexing function enumerates the space in the row major order.

Arrays With Other Index Types

```
module Array (Array, mkArray, (!), bounds)
where

   infix 9 (!)

   data (Ix a) => Array a t
   mkArray :: (Ix a) => (a,a) -> (a->t) -> Array a t
   (!) :: (Ix a) => (Array a t) -> a -> t
   bounds :: (Ix a) => (Array a t) -> (a,a)

Thus,

   type ArrayI t = Array Int t
   type MatrixI t = Array (Int,Int) t
```
Higher Dimensional Arrays

\[ \mathbf{x} = \text{mkArray} ((l1,12),(u1,u2)) f \]

means \[ \mathbf{x}!(i,j) = f(i,j) \quad 11 \leq i \leq u1 \]
\[ 12 \leq j \leq u2 \]

Type
\[ \mathbf{x} :: (\text{Array (Int,Int) t}) \]

Assuming
\[ f :: (\text{Int,Int}) \rightarrow t \]

\text{mkArray} will work for higher dimensional matrices as well.

Array of Arrays

\((\text{Array a (Array a t)}) \neq (\text{Array (a,a) t})\)

This allows flexibility in the implementation of higher dimensional arrays.
Matrices

\[
\text{add } (i,j) = i + j
\]

\[
\text{mkArray } ((1,1),(n,n)) \text{ add } ?
\]

\[
\begin{array}{ccc}
& j & \\
\hline
i & & \\
\end{array}
\]

Transpose

\[
\text{transpose a } = \\
\begin{array}{l}
\text{let} \\
((l1,l2),(u1,u2)) = \text{bounds a} \\
\text{f } (i,j) = (j,i) \\
in \\
\text{mkArray } ((l2,l1),(u2,u1)) \text{ f}
\end{array}
\]
The *Wavefront Example*

\[ x_{i,j} = x_{i-1,j} + x_{i,j-1} \]

\[ x = \text{mkArray } ((1,1),(n,n)) (f \, x) \]

\[ f \, x \, (i, j) = \begin{cases} 1 & \text{if } i == 1 \text{ then } 1 \\ \text{else if } j == 1 \text{ then } 1 \\ \text{else } x!(i-1,j) + x!(i,j-1) \end{cases} \]

---

**Compute the least fix point.**

\[ x = \text{mkArray } ((1,1),(n,n)) (f \, x) \]

\[ f \, x \, (i, j) = \begin{cases} 1 & \text{if } i == 1 \text{ then } 1 \\ \text{else if } j == 1 \text{ then } 1 \\ \text{else } x!(i-1,j) + x!(i,j-1) \end{cases} \]