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 **(*case* e *of* C)**
where **C** is

```
C = P -> e   | (P -> e) , C

P = x | CN_0 | CN_k(P_1, …,P_k)
```

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

(*case* e *of* P -> e1, C)
  ⇒ **e1** if match**(P,e)**
  ⇒ if ~match**(P,e)**
(*case* e *of* P -> e1)
  ⇒ **e1** if match**(P,e)**
  ⇒ if ~match**(P,e)**

The match Function

P = x | CN_0 | $CN_k(P_1, …,P_k)$

```
match[[x, t]]     = True

match[[CN_0, t]]  = CN_0 == tag(t)

match[[CN_k(P_1, …,P_k), t]] =
                    if tag(t) == CN_k
                    then
                      (match[[P_1, proj_1(t)]] &&
                       .
                       .
                       .
                       match[[P_k, proj_k(t)]])
                    else
                       False
```

pH Pattern Matching

```
TE[[(case e of C)]]    =
     (let t = e in TC[[t, C]])

TC[[t, (P -> e)]]      =
     if match[[P, t]]
          then (let bind[[P, t]] in e)
          else error "match failure"

TC[[t, ((P -> e),C)]] =
     if match[[P, t]]
          then (let bind[[P, t]] in e)
          else TC[[t, C]]
```


Pattern Matching: bind Function

bind[[x, t]] = x = t

bind[[CN_0 , t]] = ε

bind[[$CN_k(P_1, \ldots, P_k)$, t]] =
bind[[P_1, $proj_1(t)$]] ;
.
.
.
bind[[P_k, $proj_k(t)$]]

Refutable vs Irrefutable Patterns

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

```
let (x1, x2)       = e1
     x : xs        = e2
     y1: y2 : ys   = e3
in
   e
```

what if **e2** *evaluates to* **[]** *?*
e3 *to a one-element list ?*

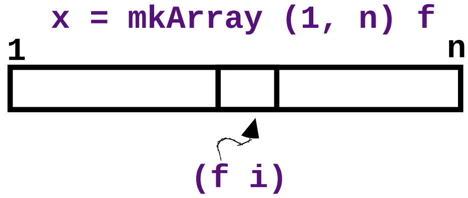
Should we disallow refutable patterns in bindings?
Too inconvenient!

Turn each binding into a case expression

Arrays

Cache for function values on a regular subdomain

x = mkArray (1, n) f

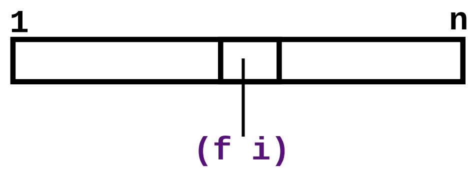


means **x!i = (f i)**
1 ≤ i ≤ n

Selection: **x!i** returns the value of the i^th slot

Bounds: **(bounds x)** returns the tuple containing the bounds

Efficiency is the Motivation for Arrays



(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 = mkArray  (1,10)  (plus 5)
```

Type

```
x  ::  (ArrayI t)
```

assuming

```
f ::  Int -> t
```

Array: An Abstract Data Type

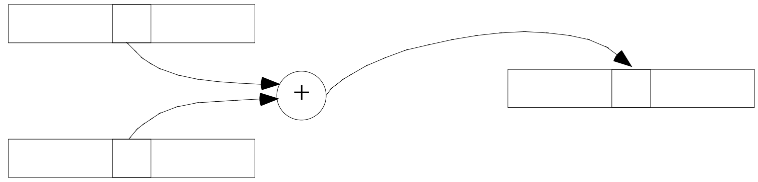
```
module ArrayI (ArrayI, mkArray, (!), bounds)
   where

   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

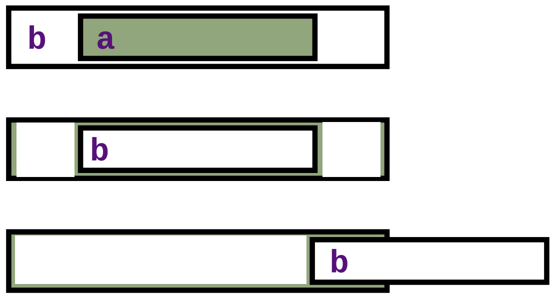
```
vs  a b  = let
                esum i = a!i + b!i
           in
                mkArray (bounds a) esum
```

Suppose

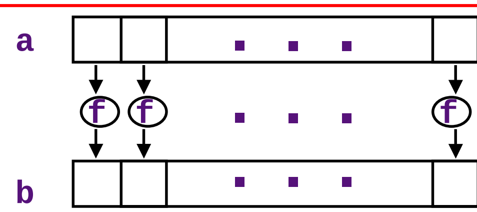
1.

2.

3.



Map Array

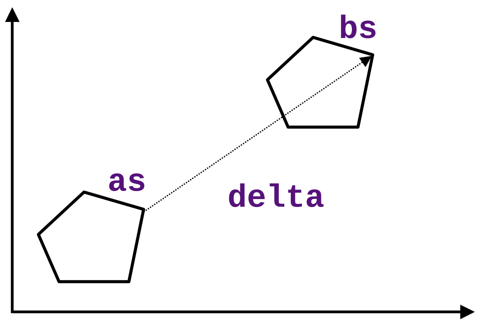



```
mapArray f a = let
                  g i = f (a!i)
               in
                  mkArray (bounds a) g
```

Example: scale a vector, that is, produce b such that **b_i = s * a_i**

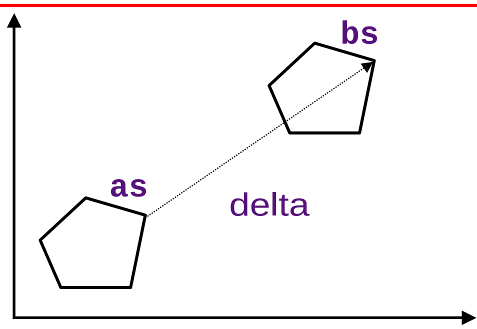
```
vscale a s = mapArray ((*) s)  a   ?
```

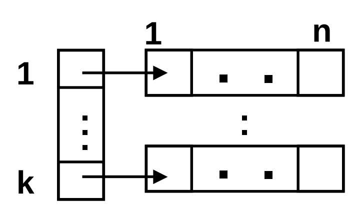
Dragging a Shape



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

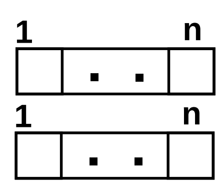
k-sided polygon: An array of points





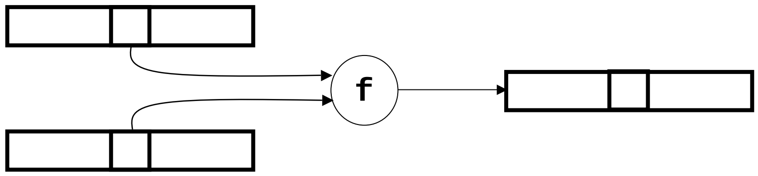
A point in n-dimensional space

distance delta in n-dimensional space




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

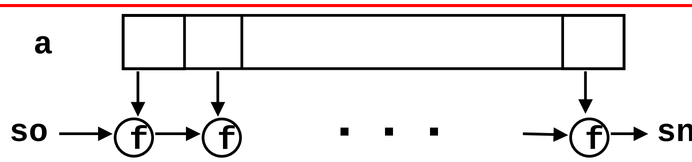
High-level Programming




```
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: $\Sigma\, a_i\, b_i$

```
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

```
class Ix a where
   range        :: (a,a) -> [a]
   index        :: (a,a) -> a -> Int
   inRange  :: (a,a) -> a -> 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 dimentional 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

```
x = mkArray ((l1,l2),(u1,u2)) f
```

means `x!(i,j) = f (i,j)` $l1 \leq i \leq u1$, $l2 \leq j \leq u2$

Type

```
x  :: (Array (Int,Int) t)
```

Assuming

```
f  :: (Int,Int) -> t
```

`mkArray` will work for higher dimensional matrices as well.

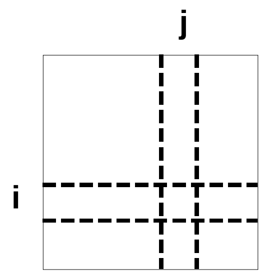
Array of Arrays

`(Array a (Array a t))` $\not\equiv$ `(Array (a,a) t)`

This allows flexibility in the implementation of higher dimensional arrays.

Matrices

```
add  (i,j) =  i + j

mkArray  ((1,1),(n,n))  add       ?
```



Transpose

```
transpose a =
     let
        ((l1,l2),(u1,u2)) = bounds a

        f (i,j) =  (j,i)             ?

     in
        mkArray  ((l2,l1),(u2,ui))  f
```

The *Wavefront* Example

$$x_{i,j} = x_{i-1,j} + x_{i,j-1}$$

1	1	1	1	1	1	1	1
1							
1							
1							
1							
1							
1							
1							

```
x = mkArray ((1,1),(n,n)) (f x)

f x (i, j) = if  i == 1 then 1
             else if  j == l  then  1
                  else x!(i-1,j) + x!(i,j-1)
```


Compute the least fix point.

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

1	1	1	1	1	1	1	1
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥

```
x = mkArray ((1,1),(n,n)) (f x)

f x (i, j) = if  i == 1 then 1
             else if  j == l  then  1
                  else x!(i-1,j) + x!(i,j-1)
```