



I - Structures and Open Lists

Arvind
Laboratory for Computer Science
M.I.T.

October 16, 2002

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

Array: An Abstract Datatype

```
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
```



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*

October 16, 2002

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

Higher Dimensional Arrays

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

means $x!(i,j) = f(i,j)$ $\begin{matrix} l1 \leq i \leq u1 \\ l2 \leq j \leq u2 \end{matrix}$

Type

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

Assuming

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

mkArray will work for higher dimensional matrices as well.

October 16, 2002

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

The Wavefront Example

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

1	1	1	1	1	1	1	1
1	2	3	4	5	⊥	⊥	⊥
1	3	6	10	⊥	⊥	⊥	⊥
1	4	10	⊥	⊥	⊥	⊥	⊥
1	5	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥

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

October 16, 2002

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

Array Comprehension

A special function to turn a list of (index,value) pairs into an array

```
array :: (Ix a) => (a,a) -> [(a,t)] -> (Array a t)
array ebound
      [(ie1,e1) | gen-pred, ..]
      ++ [(ie2,e2) | gen-pred, ..] ++ ...)
```

Thus,

```
mkArray (l,u) f =
  array (l,u) [(j,(f j)) | j <- range(l,u)]
```

List comprehensions and function array provide flexibility in constructing arrays, and the compiler can implement them efficiently

duplicates?

October 16, 2002

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

Array Comprehension: *Wavefront*

$$x[i,j] = x[i-1,j] + x[i,j-1]$$

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

```
x = array ((1,1),(n,n))
  ([(1,1), 1])
  ++ [(i,1), 1] |
  ++ [(1,j), 1] |
  ++ [(i,j), x!(i-1,j) + x!(i,j-1)]
  ])
```

October 16, 2002

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

Computed Indices

Inverse permutation
 $y!(x!i) = i$

2	5	6	1	3	4
↓					
4	1	5	6	2	3

x
y

```
find x i =
  let % find j such that x!j = i
      step j = if x!j == i then j
              else step j+1
  in
    step 1
  y = mkArray (1,n) (find x)
```

How many comparisons? Can we do better?

```
y = array (1,n) [( , ) | i <- [1..n]]
```

October 16, 2002

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

I-structures

In functional data structures, a *single construct* specifies:

- The *shape* of the data structure
- The value of its components

These two aspects are specified *separately* using I-structures

- efficiency
- parallelism

I-structures preserve *determinacy* but are *not* functional !

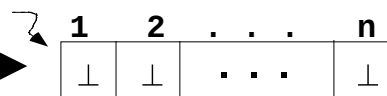
October 16, 2002

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

I-Arrays

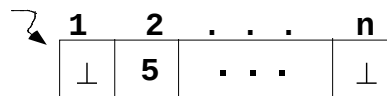
- Allocation expression

iArray (1,n) [] →



- Assignment

iAStore a 2 5
or **a!2 := 5**



provided the previous content was $\perp$
"The single assignment restriction."

- Selection expression

a!2 → 5

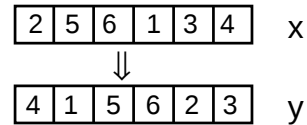
($\perp$ means empty)

October 16, 2002

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

Computed Indices Using I-structures

Inverse permutation
 $y ! (x ! i) = i$



```
let
  y = iArray (1,n) []
  _ = for i <- [1..n] do
    _ = iStore y (x!i) i
  finally ()    % unit data type
in
  y
```

What if x contains a duplicate ?

October 16, 2002

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


Multiple-Store Error

Multiple assignments to an iArray slot cause a multiple store error

A program with exposed store error is suppose to blow up!

Program --> T

The Top represents a contradiction

October 16, 2002

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


The Unit Type

```
data () = ()
```

means we cannot do much with an object of the unit type. However, it does allow us to drop ' = '

```
let
  y = iArray (1,n) []
  for i <- [1..n] do
    iASTore y (x!i) i
  finally ()           -- unit data type
in
  y
```

For better syntax replace

```
iASTore y (x!i) i by   y!(x!i) := i
```

October 16, 2002

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


I-Cell

```
data ICell a = ICell {contents :: . a}
```

Constructor

```
ICell :: a -> ICell a
```

I-Structure field

```
ICell e           or       ICell {contents = e}
```

or create an empty cell and fill it

```
ic = ICell {}
contents ic := e
```

Selector

```
contents ic       or
case ic of
  ICell x -> ... x ...
```

October 16, 2002

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


An Array of ICells

Example: Rearrange an array such that the negative numbers precede the positive numbers

2 8 -3 14 2 7 -5

-3 -5 2 8 14 2 7

Functional solutions are not efficient

```
let y = array (1,n) [(i,ICell {})| i<-[1..n]]
(l,r) = (0,n+1)
final_r = for j <- [1..n] do
  (l',r',k) =

  contents (y!k) := x!j
  next l = l'
  next r = r'
  finally r
in (y, final_r)
```

October 16, 2002

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

Type Issues

In the previous example

```
x :: Array Int
y :: Array (ICell Int)
```

1. We will introduce an I-Structure array to eliminate an extra level of indirection
2. The type of a functional array (Array) is different from the type of an IArray.

However, an IArray behaves like a functional Array after all its elements have been filled .

We provide a primitive function for this conversion

```
cvt_IArray_to_Array ia -> a
```

October 16, 2002

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

Types Issue *(cont.)*

Hindley-Milner type system has to be extended
to deal with I-structures

$\Rightarrow?$ *ref type* -- requires new rules
more on this later...



All functional data structures in pH
are implemented as I-structures.



Array Comprehensions: *a packaging of I-structures*

```
array dimension
  [(ie1,e1) | x <- xs, y <- ys]
++ [(ie2,e2) | z <- zs] )
```

translated into

```
let  a = iArray dimension []
    for x <- xs do
      for y <- ys do
        a!ie1 := e1
      finally ()
    finally ()
    for z <- zs do
      a!ie2 := e2
    finally ()
in  cvt_IArray_to_Array a
```

October 16, 2002

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


I-lists

```
data IList t = INil
              | ICons {hd :: t, tl :: (IList t)}
```

Allocation

```
x = ICons {hd = 5}
```

Assignment

```
tl x := e
```

The single assignment restriction.

If violated the program will blow up.

Selection

```
case xs of
  INil      -> ...
  ICons h t -> ...
```

we can also write `ICons {hd=h, tl=t} -> ...`

I-Structure field

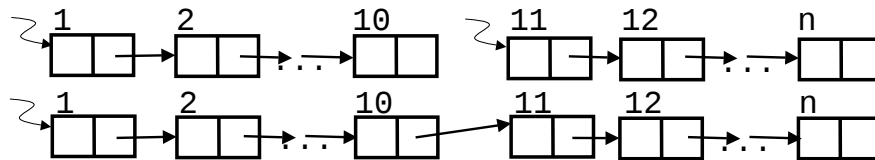
October 16, 2002

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


Open List Operations

A pair of I-list pointers for the *header* and the *trailer* cells.

joining two open lists



closing an open list



October 16, 2002

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


Open List Operation Definitions

```
type open_list t = ((IList t), (IList t))
```

```
nil_ol = (INil, INil)
```

```
close (hr, tr) =
  let
    case hr of
      INil -> ()
      ICons _ _ -> {t1 tr := INil}
  in cnv_Ilist_to_list hr
```

```
join (hr1, tr1) (hr2, tr2) =
  case hr1 of
    INil ->
      ICons _ _ ->
```

October 16, 2002

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


Map Using Open Lists

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

- *Inefficient because it is not tail recursive!*
- A tail recursive version can be written using open lists:
`map f xs = close (open_map f xs)`
 where

```
open_map f [] = (INil, INil)
open_map f (x:xs) =
  let tr = ICons {hd=(f x)}
      last = for x' <- xs do
```

```
      finally tr
  in (tr, last)
```

October 16, 2002

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

Implementing List Comprehensions

Functional
solution 1

```
[ e | x <- xs, y <- ys ] =>
  concatMap (\x->
    concatMap (\y-> [e]) ys) xs
```

Functional
solution 2

```
[ e | x <- xs, y <- ys ] =>
  let f [] = []
      f (x:xs') =
        let g [] = f xs'
            g (y:ys') = e:(g ys')
        in (g ys)
  in (f xs)
```

October 16, 2002

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

Implementing List Comprehensions Using Open Lists

`[ e | x <- xs, y <- ys ]`

1. Make *n open lists*, one for each *x* in *xs*
2. Join these lists together

```
let
  zs = nil_ol
in
  for x <- xs do
    z' = open_map (\y-> e) ys
  next zs = join zs z'
  finally zs
```

October 16, 2002

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

I-structures are *non functional*

```
f x y = let x!1 := 10
         y!1 := 20
         in ()
```

```
let x = iArray (1,2) []
in f x x
```

≡

```
f (iArray (1,2) []) (iArray (1,2) []) ?
```

October 16, 2002

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

The example

```
f x y = let x!1 := 10  
        y!1 := 20  
        in ()
```

```
let  
  x = iArray (1,2) []  
in  
  f x x  
  ↓↓
```

```
f (iArray (1,2) [])  
  (iArray (1,2) [])  
  ↓↓
```

