I - Structures and Open Lists

Arvind
Laboratory for Computer Science
M.I.T.

Lecture 11

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

Array: An Abstract Datatype

```haskell
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

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

---

Higher Dimensional Arrays

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

means

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

Type

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

Assuming

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

mkArray will work for higher dimensional matrices as well.
The Wavefront Example

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

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 & 5 & & & \\
1 & 3 & 6 & 10 & & & & \\
1 & 4 & 10 & & & & & \\
1 & 5 & & & & & & \\
1 & & & & & & & \\
1 & & & & & & & \\
1 & & & & & & & \\
\end{array}
\]

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

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

Array Comprehension

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

\[
\text{array} :: (\text{Ix } a) \Rightarrow (a,a) \Rightarrow [(a,t)] \Rightarrow (\text{Array } a \ t)
\]

\[
\text{array } \text{ebound} \\
\quad ([(i_{el},e1) | \text{gen-pred}, ..] \\
\quad + [(i_{el2},e2) | \text{gen-pred}, ..] + ...)
\]

Thus,

\[
\text{mkArray } (1,u) \ f = \\
\quad \text{array } (1,u) [(j, (f \ j)) | j <- \text{range}(1,u)]
\]

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

duplicates?
### Array Comprehension: Wavefront

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

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

\[
++ [((i,1), 1) | ]
++ [((1,j), 1) | ]
++ [((i,j), x!(i-1,j) + x!(i, j-1))]
\]

### Computed Indices

**Inverse permutation**

\[
y ! (x ! i) = i
\]

\[
\text{find } x i = \\
\quad \text{let } % \text{ find } j \text{ such that } x!j = i \\
\quad \quad \text{step } j = \text{if } x!j == i \text{ then } j \\
\quad \quad \text{else step } j+1 \\
\quad \text{in } \\
\quad \text{step 1} \\
\quad y = \text{mkArray } (1,n) \ (\text{find } x)
\]

*How many comparisons? Can we do better?*

\[
y = \text{array } (1,n) [\{ ( , ) | i <- [1..n] ]
\]
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!

I-Arrays

- Allocation expression
  
  ![iArray call](image)

- Assignment
  
  ![iAStore operation](image)

or `a!2 := 5`

provided the previous content was `⊥`

"The single assignment restriction."

- Selection expression

  ![Selection example](image)
Computed Indices Using I-structures

Inverse permutation
\[ y ! (x ! i) = i \]

\[
\begin{array}{cccccc}
2 & 5 & 6 & 1 & 3 & 4 \\
\text{\downarrow} & & & & & \\
4 & 1 & 5 & 6 & 2 & 3
\end{array}
\]

What if \( x \) contains a duplicate?

Multiple-Store Error

Multiple assignments to an iArray slot cause a multiple store error

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

Program --> T

The Top represents a contradiction
The Unit Type

\[ \text{data } () = () \]

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

\begin{verbatim}
let
  y = iArray (1,n) []
  for i <- [1..n] do
    iAStore y (x!i) i
  finally ()  -- unit data type
in
  y
\end{verbatim}

For better syntax replace

\begin{verbatim}
iAStore y (x!i) i by y!(x!i) := i
\end{verbatim}

I - Cell

\[ \text{data ICell a = ICell } \{ \text{contents :: . a} \} \]

\textbf{Constructor}

ICell :: a -> ICell a

ICell e          \textbf{or}        ICell \{ \text{contents = e} \}

or create an empty cell and fill it

\begin{verbatim}
ic = ICell {}
contents ic := e
\end{verbatim}

\textbf{Selector}

contents ic          \textbf{or}

\begin{verbatim}
\text{case ic of}
  ICell x -> ... x ...
\end{verbatim}
An Array of ICells

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

\[ 2 \quad 8 \quad -3 \quad 14 \quad 2 \quad 7 \quad -5 \]
\[ -3 \quad -5 \quad 2 \quad 8 \quad 14 \quad 2 \quad 7 \]

Functional solutions are not efficient

\[
\begin{align*}
\text{let } & y = \text{array } (1,n) \ [(i, \text{ICell } \{} \mid i<-\{1..n\}] \\
(1,r) & = (0,n+1) \\
\text{final}_r & = \text{for } j \leftarrow \{1..n\} \ \text{do} \\
& (l', r', k) = \\
& \quad \text{contents } (y!k) := x!j \\
& \quad \text{next } l = l' \\
& \quad \text{next } r = r' \\
& \quad \text{finally } r \\
\end{align*}
\]

in \ (y, \ final_r)

Type Issues

In the previous example

\[
\begin{align*}
x & :: \text{Array Int} \\
y & :: \text{Array (ICell Int)}
\end{align*}
\]

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

\[
cvt_{IArray\_to\_Array} \ ia \rightarrow a
\]
Types Issue (cont.)

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

⇒? 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!iel := e1
  finally ()
finally ()
for z <- zs do
  a!ie2 := e2
finally ()
in cvt_IArray_to_Array a
```

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} -> ...
```
Open List Operations

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

joining two open lists

1 2 10
\[ \rightarrow \]

11 12 n
\[ \rightarrow \]

1 2 10
\[ \rightarrow \]

11 12 n
\[ \rightarrow \]

closing an open list

1 10
\[ \rightarrow \]

11 n
\[ \rightarrow \]

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 _ _ -> {tl tr := INil}
    in cnv_Ilist_to_list hr

join (hr1,tr1) (hr2,tr2) =
    case hr1 of
        INil ->
        ICons _ _ ->
Map Using Open Lists

- Inefficient because it is not tail recursive!
- A tail recursive version can be written using open lists:

$$\text{map } f \text{ } xs = \text{close } (\text{open_map } f \text{ } xs)$$

where

$$\text{open_map } f \text{ } [] = (\text{INil}, \text{INil})$$

$$\text{open_map } f \text{ } (x:xs) =$$

let $$tr = \text{ICons } (\text{hd} = (f \text{ } x))$$

last = for $$x' \leftarrow xs$$ do

finally $$tr$$

in $$(tr, \text{last})$$

Implementing List Comprehensions

Functional solution 1

$$[ e \mid x \leftarrow xs, \ y \leftarrow ys ] \Rightarrow$$

$$\text{concatMap } (\lambda x \rightarrow \text{concatMap } (\lambda y \rightarrow [e]) \text{ } ys) \text{ } xs$$

Functional solution 2

$$[ e \mid x \leftarrow xs, \ y \leftarrow ys ] \Rightarrow$$

let $$f [] = []$$

$$f (x:xs') =$$

let $$g [] = f \text{ } xs'$$

$$g (y:ys') = e : (g \text{ } ys')$$

in $$(g \text{ } ys)$$

in $$(f \text{ } xs)$$
Implementing List Comprehensions Using Open Lists

\[ \{ e | x \leftarrow xs, y \leftarrow ys \} \]

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

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

I-structures are **non functional**

```haskell
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) [])
```

The example

\[
f \ x \ y = \text{let } x!1 := 10 \quad y!1 := 20 \
\quad \text{in } ()
\]

\[
\text{let } \\
\quad x = iArray (1,2) [] \\
\quad \text{in } \\
\quad f \ x \ x \\
\downarrow
\]

\[
f \ (iArray (1,2) []) \\
\quad (iArray (1,2) [] ) \\
\downarrow
\]