JFP **13** (1): 173–178, January 2003.
DOI: 10.1017/S0956796803001813 Printed in the United Kingdom

Chapter 16

Arrays

```
module  Array (
        module Ix,  -- export all of Ix for convenience
        Array, array, listArray, (!), bounds, indices, elems, assocs,
        accumArray, (//), accum, ixmap ) where

import Ix

infixl 9  !, //

data  (Ix a)    => Array a b = ...      -- Abstract

array           :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray       :: (Ix a) => (a,a) -> [b] -> Array a b
(!)             :: (Ix a) => Array a b -> a -> b
bounds          :: (Ix a) => Array a b -> (a,a)
indices         :: (Ix a) => Array a b -> [a]
elems           :: (Ix a) => Array a b -> [b]
assocs          :: (Ix a) => Array a b -> [(a,b)]
accumArray      :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                             -> Array a b
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
accum           :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                             -> Array a b
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                             -> Array a c
```

```
instance                              Functor (Array a) where ...
instance  (Ix a, Eq b)           => Eq   (Array a b)  where ...
instance  (Ix a, Ord b)          => Ord  (Array a b)  where ...
instance  (Ix a, Show a, Show b) => Show (Array a b)  where ...
instance  (Ix a, Read a, Read b) => Read (Array a b)  where ...
```

Haskell provides indexable *arrays*, which may be thought of as functions whose domains are isomorphic to contiguous subsets of the integers. Functions restricted in this way can be implemented efficiently; in particular, a programmer may reasonably expect rapid access to the components. To ensure the possibility of such an implementation, arrays are treated as data, not as general functions.

Since most array functions involve the class `Ix`, this module is exported from `Array` so that modules need not import both `Array` and `Ix`.

16.1 Array Construction

If `a` is an index type and `b` is any type, the type of arrays with indices in `a` and elements in `b` is written `Array a b`. An array may be created by the function `array`. The first argument of `array` is a pair of *bounds*, each of the index type of the array. These bounds are the lowest and highest indices in the array, in that order. For example, a one-origin vector of length `10` has bounds `(1,10)`, and a one-origin `10` by `10` matrix has bounds `((1,1),(10,10))`.

The second argument of `array` is a list of *associations* of the form ($index$, $value$). Typically, this list will be expressed as a comprehension. An association `(i, x)` defines the value of the array at index `i` to be `x`. The array is undefined (i.e. $\perp$) if any index in the list is out of bounds. If any two associations in the list have the same index, the value at that index is undefined (i.e. $\perp$). Because the indices must be checked for these errors, `array` is strict in the bounds argument and in the indices of the association list, but nonstrict in the values. Thus, recurrences such as the following are possible:

```
a = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

Not every index within the bounds of the array need appear in the association list, but the values associated with indices that do not appear will be undefined (i.e. $\perp$). Figure 16.1 shows some examples that use the `array` constructor.

The `(!)` operator denotes array subscripting. The `bounds` function applied to an array returns its bounds. The functions `indices`, `elems`, and `assocs`, when applied to an array, return lists of the indices, elements, or associations, respectively, in index order. An array may be constructed from a pair of bounds and a list of values in index order using the function `listArray`.

If, in any dimension, the lower bound is greater than the upper bound, then the array is legal, but empty. Indexing an empty array always gives an array-bounds error, but `bounds` still yields the bounds with which the array was constructed.

https://doi.org/10.1017/S0956796803001813 Published online by Cambridge University Press

```
-- Scaling an array of numbers by a given number:
scale :: (Num a, Ix b) => a -> Array b a -> Array b a
scale x a = array b [(i, a!i * x) | i <- range b]
            where b = bounds a

-- Inverting an array that holds a permutation of its indices
invPerm :: (Ix a) => Array a a -> Array a a
invPerm a = array b [(a!i, i) | i <- range b]
            where b = bounds a

-- The inner product of two vectors
inner :: (Ix a, Num b) => Array a b -> Array a b -> b
inner v w = if b == bounds w
                then sum [v!i * w!i | i <- range b]
                else error "inconformable arrays for inner product"
            where b = bounds v
```

Figure 16.1: Array examples

16.1.1 Accumulated Arrays

Another array creation function, `accumArray`, relaxes the restriction that a given index may appear at most once in the association list, using an *accumulating function* which combines the values of associations with the same index. The first argument of `accumArray` is the accumulating function; the second is an initial value; the remaining two arguments are a bounds pair and an association list, as for the `array` function. For example, given a list of values of some index type, `hist` produces a histogram of the number of occurrences of each index within a specified range:

```
hist :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i<-is, inRange bnds i]
```

If the accumulating function is strict, then `accumArray` is strict in the values, as well as the indices, in the association list. Thus, unlike ordinary arrays, accumulated arrays should not in general be recursive.

16.2 Incremental Array Updates

The operator `(//)` takes an array and a list of pairs and returns an array identical to the left argument except that it has been updated by the associations in the right argument. (As with the `array` function, the indices in the association list must be unique for the updated elements to be defined.) For example, if `m` is a 1-origin, `n` by `n` matrix, then `m//[((i,i), 0) | i <- [1..n]]` is the same matrix, except with the diagonal zeroed.

`accum` f takes an array and an association list and accumulates pairs from the list into the array with the accumulating function f. Thus `accumArray` can be defined using `accum`:

```
accumArray f z b = accum f (array b [(i, z) | i <- range b])
```

https://doi.org/10.1017/S0956796803001813 Published online by Cambridge University Press

```
-- A rectangular subarray
subArray :: (Ix a) => (a,a) -> Array a b -> Array a b
subArray bnds = ixmap bnds (\i->i)

-- A row of a matrix
row :: (Ix a, Ix b) => a -> Array (a,b) c -> Array b c
row i x = ixmap (l',u') (\j->(i,j)) x where ((_,l'),(_,u')) = bounds x

-- Diagonal of a matrix (assumed to be square)
diag :: (Ix a) => Array (a,a) b -> Array a b
diag x = ixmap (l,u) (\i->(i,i)) x
       where
         ((l,_),(u,_)) = bounds x

-- Projection of first components of an array of pairs
firstArray :: (Ix a) => Array a (b,c) -> Array a b
firstArray = fmap (\(x,y)->x)
```

Figure 16.2: Derived array examples

16.3 Derived Arrays

The two functions `fmap` and `ixmap` derive new arrays from existing ones; they may be thought of as providing function composition on the left and right, respectively, with the mapping that the original array embodies. The `fmap` function transforms the array values while `ixmap` allows for transformations on array indices. Figure 16.2 shows some examples.

16.4 Library `Array`

```
module  Array (
    module Ix,  -- export all of Ix
    Array, array, listArray, (!), bounds, indices, elems, assocs,
    accumArray, (//), accum, ixmap ) where

import Ix
import List( (\\) )

infixl 9  !, //

data (Ix a) => Array a b = MkArray (a,a) (a -> b) deriving ()
```

https://doi.org/10.1017/S0956796803001813 Published online by Cambridge University Press

```
array       :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
array b ivs =
    if and [inRange b i | (i,_) <- ivs]
        then MkArray b
                     (\j -> case [v | (i,v) <- ivs, i == j] of
                            [v]   -> v
                            []    -> error "Array.!: \
                                           \undefined array element"
                            _     -> error "Array.!: \
                                           \multiply defined array element")
        else error "Array.array: out-of-range array association"

listArray             :: (Ix a) => (a,a) -> [b] -> Array a b
listArray b vs        =  array b (zipWith (\ a b -> (a,b)) (range b) vs)

(!)                   :: (Ix a) => Array a b -> a -> b
(!) (MkArray _ f)     =  f

bounds                :: (Ix a) => Array a b -> (a,a)
bounds (MkArray b _)  =  b

indices               :: (Ix a) => Array a b -> [a]
indices               =  range . bounds

elems                 :: (Ix a) => Array a b -> [b]
elems a               =  [a!i | i <- indices a]

assocs                :: (Ix a) => Array a b -> [(a,b)]
assocs a              =  [(i, a!i) | i <- indices a]

(//)                  :: (Ix a) => Array a b -> [(a,b)] -> Array a b
a // new_ivs          = array (bounds a) (old_ivs ++ new_ivs)
                      where
                        old_ivs = [(i,a!i) | i <- indices a,
                                             i `notElem` new_is]
                        new_is  = [i | (i,_) <- new_ivs]

accum                 :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                                   -> Array a b
accum f               =  foldl (\a (i,v) -> a // [(i,f (a!i) v)])

accumArray            :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                                   -> Array a b
accumArray f z b      =  accum f (array b [(i,z) | i <- range b])

ixmap                 :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                                         -> Array a c
ixmap b f a           = array b [(i, a ! f i) | i <- range b]

instance  (Ix a)          => Functor (Array a) where
    fmap fn (MkArray b f) =  MkArray b (fn . f)

instance  (Ix a, Eq b)  => Eq (Array a b)  where
    a == a' =  assocs a == assocs a'

instance  (Ix a, Ord b) => Ord (Array a b)  where
    a <= a' =  assocs a <= assocs a'
```

```
instance  (Ix a, Show a, Show b) => Show (Array a b)  where
    showsPrec p a = showParen (p > arrPrec) (
                    showString "array " .
                    showsPrec (arrPrec+1) (bounds a) . showChar ' ' .
                    showsPrec (arrPrec+1) (assocs a)                  )

instance  (Ix a, Read a, Read b) => Read (Array a b)  where
    readsPrec p = readParen (p > arrPrec)
           (\r -> [ (array b as, u)
                  | ("array",s) <- lex r,
                    (b,t)       <- readsPrec (arrPrec+1) s,
                    (as,u)      <- readsPrec (arrPrec+1) t ])

-- Precedence of the 'array' function is that of application itself
arrPrec = 10
```

https://doi.org/10.1017/S0956796803001813 Published online by Cambridge University Press