# Data.Array


| | |
|---|---|
| **Copyright** | (c) The University of Glasgow 2001 |
| **License** | BSD-style (see the file libraries/base/LICENSE) |
| **Maintainer** | libraries@haskell.org |
| **Stability** | provisional |
| **Portability** | portable |
| **Safe Haskell** | Trustworthy |
| **Language** | Haskell2010 |


Basic non-strict arrays.

*Note:* The Data.Array.IArray module provides a more general interface to immutable arrays: it defines operations with the same names as those defined below, but with more general types, and also defines `Array` instances of the relevant classes. To use that more general interface, import Data.Array.IArray but not Data.Array.

**Contents**



## Immutable non-strict arrays

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 Data.Array so that modules need not import both Data.Array and Data.Ix.

```
module Data.Ix
```

```
data Array i e :: * -> * -> *
```

The type of immutable non-strict (boxed) arrays with indices in `i` and elements in `e`.

**Instances**

```
IArray Array e
Ix i => Functor (Array i)
(Ix i, Eq e) => Eq (Array i e)
(Ix i, Ord e) => Ord (Array i e)
(Ix a, Show a, Show b) => Show (Array a b)
Typeable (* -> * -> *) Array
```

## Array construction

**`array`**

```
:: Ix i
=> (i, i)
```
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))'.

```
-> [(i, e)]
```
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`.

```
-> Array i e
```

Construct an array with the specified bounds and containing values for given indices within these bounds.

The array is undefined (i.e. bottom) if any index in the list is out of bounds. The Haskell 2010 Report further specifies that if any two associations in the list have the same index, the value at that index is undefined (i.e.

bottom). However in GHC's implementation, the value at such an index is the value part of the last association with that index in the list.

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 non-strict 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. bottom).

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.

---

**`listArray`** `:: Ix i => (i, i) -> [e] -> Array i e`

Construct an array from a pair of bounds and a list of values in index order.

---

**`accumArray`**

| | |
|---|---|
| `:: Ix i` | |
| `=> (e -> a -> e)` | accumulating function |
| `-> e` | initial value |
| `-> (i, i)` | bounds of the array |
| `-> [(i, a)]` | association list |
| `-> Array i e` | |

The `accumArray` function deals with repeated indices in the association list using an *accumulating function* which combines the values of associations with the same index. 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 built with `array`, accumulated arrays should not in general be recursive.

## Accessing arrays

---

**`(!)`** `:: Ix i => Array i e -> i -> e` | `infixl 9`

The value at the given index in an array.

---

**`bounds`** `:: Ix i => Array i e -> (i, i)`

The bounds with which an array was constructed.

---

**`indices`** `:: Ix i => Array i e -> [i]`

The list of indices of an array in ascending order.

---

**`elems`** `:: Ix i => Array i e -> [e]`

The list of elements of an array in index order.

```
assocs :: Ix i => Array i e -> [(i, e)]
```

The list of associations of an array in index order.

## Incremental array updates

```
(//) :: Ix i => Array i e -> [(i, e)] -> Array i e   infixl 9
```

Constructs an array identical to the first argument except that it has been updated by the associations in the right argument. 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.

Repeated indices in the association list are handled as for `array`: Haskell 2010 specifies that the resulting array is undefined (i.e. bottom), but GHC's implementation uses the last association for each index.

```
accum :: Ix i => (e -> a -> e) -> Array i e -> [(i, a)] -> Array i e
```

`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])
```

## Derived arrays

```
ixmap :: (Ix i, Ix j) => (i, i) -> (i -> j) -> Array j e -> Array i e
```

`ixmap` allows for transformations on array indices. It may be thought of as providing function composition on the right with the mapping that the original array embodies.

A similar transformation of array values may be achieved using `fmap` from the `Array` instance of the `Functor` class.

---

Produced by Haddock version 2.14.2