



# CS 115

# Functional Programming

*Lecture 14:*  
Arrays



*Functional Programming*



# Today

- Immutable arrays in Haskell
  - The `Ix` type class
  - The `IArr` type class
  - Functional arrays (`Array`)
  - Useful functions on `Arrays`
- Mutable arrays in Haskell
  - The `MArr` type class
  - Imperative arrays (`IOArr` and `IOUArr`)
  - Useful functions on `IOArrays`





# Thought experiment

- Why do we want/need arrays?
  - We already have lists to store sequences of data values of a single type...
- Benefits of arrays?
  - $O(1)$  lookup
  - Changing values at specific locations in array (mutable arrays only); also  $O(1)$  if supported





# The **Ix** type class

- Most languages use only non-negative integers as array indices (starting from 0)
- This can be limiting because other notions of indexing exist, e.g.
  - 1-based indexing (starting from 1)
  - tuple-based indexing (to simulate multidimensional arrays)
- Haskell allows different kinds of array indices
  - defines the **Ix** (indexable) type class to specify operations that array indices must be able to do





# The **Ix** type class

```
class Ord a => Ix a where
    range      :: (a,a) -> [a]
    index      :: (a,a) -> a -> Int
    inRange    :: (a,a) -> a -> Bool
    rangeSize :: (a,a) -> Int
```

- **range** takes the bounds of an array and returns a list of all valid indices for that array
- **index** takes the array bounds and an index and converts it into an **Int** representing what the index would be assuming 0-based indexing with **Ints**





# The Ix type class

```
class Ord a => Ix a where
    range      :: (a,a) -> [a]
    index      :: (a,a) -> a -> Int
    inRange    :: (a,a) -> a -> Bool
    rangeSize :: (a,a) -> Int
```

- **inRange** takes the array bounds and a possible index and returns **True** if it's a valid index for that array
- **rangeSize** takes the array bounds and returns the number of valid indices for that array (rarely used)





# The **Ix** type class

```
class Ord a => Ix a where
    range      :: (a,a) -> [a]
    index      :: (a,a) -> a -> Int
    inRange    :: (a,a) -> a -> Bool
    rangeSize :: (a,a) -> Int
```

- Minimal definition: **range**, **index**, **inRange**





# Ix–amples

- **Integer** is an instance of **Ix**

```
Prelude> import Data.Ix
```

```
Prelude Data.Ix> range (1,10) :: [Integer]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
Prelude Data.Ix> index (3,10) 5
```

```
2
```

```
Prelude Data.Ix> inRange (3,10) 5
```

```
True
```

```
Prelude Data.Ix> rangeSize (3,10)
```

```
8
```





# Ix–amples

- (**Integer**, **Integer**) is an instance of **Ix**
  - Used to simulate 2-dimensional arrays

```
Prelude Data.Ix> range ((0,0),(2,3))  
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),  
(1,3),(2,0),(2,1),(2,2),(2,3)]
```

```
Prelude Data.Ix> index ((0,0),(2,3)) (1,2)
```

6

```
Prelude Data.Ix> inRange ((0,0),(2,3)) (3,3)
```

False

```
Prelude Data.Ix> rangeSize ((0,0),(2,3))
```

12





# Ix-trapolation

- We rarely have to write our own instances of **Ix**
  - Most useful instances have already been written
- Nevertheless, it's often a good idea to use **Ix** instances as the indexes for array functions you define instead of specifying a particular kind of index
- Most Haskell library functions on arrays work on **Ix** instances





# The IArray type class

- Haskell divides array types into "immutable" and "mutable" array types
- Immutable arrays (functional arrays) are instances of **IArray**
- Mutable arrays (imperative arrays) are instances of **MArray**





# The IArray type class

```
class IArray a e where
    bounds ::Ix i => a i e -> (i,i)
    numElements ::Ix i => a i e -> Int
    -- plus other "unsafe" methods
```

- **IArray** instances represent immutable arrays
- Array types have the form **a i e** where:
  - **a** is the array type constructor (kind **\* -> \* -> \***)
  - **i** is a type which is an instance of **Ix**
  - **e** is the element type





# The Array type

- **Array** is the basic array type for immutable arrays
- Implementation has this form:

```
data Array i e = ...
```

- where the `...` are implementation details (not important)
- Can work with any indexable type **i**, and any element type **e**
- **Array** is an instance of **IArray**, so **bounds** and **numElements** can always be found





# The Array type

- **Array** functions are found in both the **Data.Array** module and the **Data.Array.IArray** module
- **Data.Array.IArray** gives more general versions of array functions (work on any **IArray** instance)
- More specific versions found in **Data.Array** (only work on **Array** datatype — confusing!)
- Practically, it suffices to just import **Data.Array** if all you need are **Arrays**





# The Array type

- All the array functions to be described exist in **Array** and **IArray** versions
- Example: **listArray**
- **Array** version:

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

- **IArray** version:

```
listArray :: (IArray a e, Ix i) =>  
          (i, i) -> [e] -> a i e
```

- We'll use **Array** versions here for simplicity





# Creating Arrays

- Constructing an **Array** is done using the **array** function:

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

- Supply the lower, upper index bounds and a list of (index, element) pairs to generate the **Array**
- Example:

```
Prelude> array (1,5) [(i,i*i) | i <- [1..5]]  
array (1,5) [(1,1), (2,4), (3,9), (4,16),  
(5,25)]
```





# Creating Arrays

- Note that list of index/element pairs passed to `array` should contain all elements of the array:

```
Prelude> array (1,5) []
```

```
array (1,5) [(1,*** Exception: (Array.!) :  
undefined array element)
```

- Error doesn't occur at creation time, but will occur whenever you try to access an undefined value (like when you print `array` here)
- (We'll see what `!` means in a moment)





# Creating Arrays

- The `listArray` function creates an array from a list:

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

- Given a pair of indices, and a list of elements, create an array
- Example:

```
Prelude Data.Array> listArray (5, 10) [1..6]  
array (5,10)  
[(5,1), (6,2), (7,3), (8,4), (9,5), (10,6)]
```

- Note that this array's indices start at **5**, not **0** or **1**





# Creating Arrays

- The `accumArray` function is similar to `listArray`, but
  1. it takes a list of (index, element) pairs instead of just a list of elements
  2. (index, element) pairs with the same index are combined using a supplied combining function and an initial value
  3. Unsupplied indices get the initial value
- Type signature:

```
accumArray :: (Ix i) =>  
  (e -> e' -> e) -> e -> (i, i) -> [(i, e')] -> a i e
```





# Creating Arrays

- **accumArray** example: creating a histogram of the number of occurrences of an integer in a list of **Integers**

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





# Accessing Arrays

- The basic array accessor is the `(!)` operator:

```
(!) :: (Ix i) => Array i e -> i -> e
```

- For an array `arr`, `arr ! i` gives the `i`th element of `arr` (in constant ( $O(1)$ ) time)
- Other accessors include:

```
bounds :: (Ix i) => Array i e -> (i,i)  
-- a method of the IArray class
```

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

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

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





# "Changing" Arrays

- **Arrays** are functional, hence immutable
- Nevertheless, need to be able to generate new **Arrays** which are the same as old **Arrays** with some values changed
- Use the `(//)` operator for this:

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

- This operator takes an array and a list of (index, element) pairs to change, and returns the new array with the specified changes (takes O(N) time)





# "Changing" Arrays

- Example:

```
Prelude Data.Array> let arr = array (1, 10)  
[(i, i * i) | i <- [1..10]]
```

```
Prelude Data.Array> arr  
array (1,10)  
[(1,1), (2,4), (3,9), (4,16), (5,25), (6,36), (7,49),  
(8,64), (9,81), (10,100)]
```

```
Prelude Data.Array> arr // [(1,2), (2,8)]  
array (1,10)  
[(1,2), (2,8), (3,9), (4,16), (5,25), (6,36), (7,49),  
(8,64), (9,81), (10,100)]
```

- Note that original array **arr** is not changed by **//**





# "Changing" Arrays

- This is all you need to know in order to work with functional (immutable) arrays
- Let's move on to imperative (mutable) arrays





# The **MArray** type class

- Mutable arrays are instances of the **MArray** type class
- These arrays operate in some monad **m**, which is reflected in the type signatures of functions and methods





# The MArray type class

```
class (Monad m) => MArray a e m where
    getBounds ::Ix i => a i e -> m (i,i)
    getNumElements ::Ix i => a i e -> m Int
    newArray ::Ix i => (i,i) -> e -> m (a i e)
    newArray_ ::Ix i => (i,i) -> m (a i e)
    -- plus various "unsafe" methods
```

- **getBounds** and **getNumElements** are like **bounds** and **numElements** for **IArra**ry, except that they have a monadic return type





# The MArray type class

```
class (Monad m) => MArray a e m where
    getBounds ::Ix i => a i e -> m (i,i)
    getNumElements ::Ix i => a i e -> m Int
    newArray ::Ix i => (i,i) -> e -> m (a i e)
    newArray_ ::Ix i => (i,i) -> m (a i e)
    -- plus various "unsafe" methods
```

- **newArray** creates a new array with given bounds where all slots are filled with a particular element of type **e**
- **newArray\_** is like **newArray**, but element stored is undefined





# MArray functions

- Many functions, similar to **IArray** functions, to do common mutable array tasks:

-- Build an array from a list of elements:

```
newListArray :: (MArray a e m, Ix i) =>  
(i, i) -> [e] -> m (a i e)
```

-- Read an element from an array:

```
readArray :: (MArray a e m, Ix i) =>  
a i e -> i -> m e
```

-- Write an element to an array:

```
writeArray :: (MArray a e m, Ix i) =>  
a i e -> i -> e -> m ()
```





# MArray functions

- Note:

-- Write an element to an array:

```
writeArray :: (MArray a e m, Ix i) =>  
  a i e -> i -> e -> m ()
```

- Return type is different from the `//` operator used with **IArrays**
- Since the array is mutable, we're not returning a modified array, but changing the array in place
- An O(1) operation for most monads used with arrays





# MArray functions

- Other **MArray** functions:

```
getElems :: (MArray a e m, Ix i) => a i e -> m [e]
```

```
getAssocs :: (MArray a e m, Ix i) => a i e -> m [(i, e)]
```

- Analogous to **elems** and **assocs** for **IArrays**





# IOArray

- Two common datatypes exist which are instances of **MArray**
- Most often, we use the **IOArray** datatype for mutable arrays in the **IO** monad
- **IOArray** supports all the functions we've described, specialized to the **IO** monad
- **IOArray** is lazy in its elements
- Import module **Data.Array.IO** to use **IOArrays**





# IOUArray

- There is also an **IOUArray** datatype (where the **U** means "unboxed")
- **IOUArrays** are strict in their elements
- **IOUArrays** are more efficient than **IOArrays**; low-level representation is like a C language array
- **IOUArrays** only defined for some element types
  - e.g. **Bool**, **Char**, **Int**, **Float**, **Double** and a few others
  - use **IOArray** for arrays of more complex datatypes
- **IOUArrays** also live in **Data.Array.IO**





# IOArray example

- We'll write a function to sort an **IOArray** in place using an imperative selection sort
- Algorithm:
  - Start at the beginning of the array
  - Set aside the first element
  - Go through the rest of the elements
  - If any are smaller than the first element, swap them
  - Repeat, starting with the second element
  - Once get to end → done





# swap function

- We will need a function to swap two elements of an array:

```
-- Swap two elements in an array.  
-- Assume both indices are valid for this array.  
swap :: IOArray Integer a -> Integer -> Integer -> IO ()  
swap arr i j =  
  do vi <- readArray arr i  
      vj <- readArray arr j  
      writeArray arr i vj  
      writeArray arr j vi
```





# findIndex function

- We will need a function to find the index of the smallest element in the array, starting from a particular index
- Type signature:

```
-- Find index of smallest element.  
-- Assume that i is a valid index for this array.  
-- Assume that i <= imax.
```

```
.findIndex :: Ord a =>  
  IOArray Integer a -> Integer -> Integer -> IO Integer  
findIndex arr i imax = ...
```





# findIndex function

- Body of **.findIndex**:

```
.findIndex arr i imax =  
  if i == imax  
    then return i  
  else do i' <- findIndex arr (i + 1) imax  
         vi <- readArray arr i  
         vi' <- readArray arr i'  
         if vi < vi'  
           then return i  
           else return i'
```





# iter function

- Need a function to iterate through the array, doing the swaps, starting from a particular position in the array
- Let's call it **iter** and give it this type signature:

```
iter :: Ord a =>  
    IOArray Integer a -> Integer -> Integer -> IO ()  
iter arr i hi = ...
```





# iter function

- Body of **iter**:

```
iter arr i hi = ...  
  if i == hi  
    then return ()  
  else do i' <- findIndex arr i hi  
         swap arr i i'  
         iter arr (i + 1) hi
```





# iter function

- Note this code:

```
iter arr i hi = ...  
  if i == hi  
    then return ()  
  else do i' <- findIndex arr i hi  
          swap arr i i'  
          iter arr (i + 1) hi
```

- Conditionals inside monads where one clause is simply `return ()` are very common
- The `Control.Monad` module has functions called `when` and `unless` that can simplify this





# when and unless

```
when :: (Monad m) => Bool -> m () -> m ()
```

```
when p s = if p then s else return ()
```

```
unless :: (Monad m) => Bool -> m () -> m ()
```

```
unless p s = if p then return () else s
```





# iter function again

- We can use **unless** to simplify **iter** to:

```
iter arr i hi = ...  
  unless (i == hi)  -- or: when (i < hi)  
    (do i' <- findIndex arr i hi  
      swap arr i i'  
      iter arr (i + 1) hi)
```





# iter function again

- Let's add one minor optimization
- If `i == i'`, no need to `swap`
- Let's use `unless` again:

```
iter arr i hi =
  unless (i == hi)
    (do i' <- findIndex arr i hi
       unless (i == i') (swap arr i i')
       iter arr (i + 1) hi)
```





# selectionSort function

- Combine everything into the **selectionSort** function:

```
selectionSort :: Ord a => IOArray Integer a -> IO ()  
selectionSort arr =  
    do (lo, hi) <- getBounds arr  
        iter arr lo hi
```

- And that's it!





# Using selectionSort

- Write a simple `main` function to test `selectionSort`:

```
module Main where
import Control.Monad
import Data.Array.IO
-- [other function definitions go here]
main :: IO ()
main =
  do arr <- newListArray (1, 10)
     [4, 3, 1, 6, 5, 2, 9, 10, 8, 7]
     selectionSort arr
     elems <- getElems arr
     print elems
```





# Using selectionSort

- Compile and run:

```
$ ghc -W -o testss Sort.hs
```

```
[1 of 1] Compiling Main           ( Main.hs, Main.o )
```

```
Linking testss ...
```

```
$ testss
```

```
[1,2,3,4,5,6,7,8,9,10]
```

- Woo hoo!





# Conclusions

- You can write imperative code in Haskell which is essentially (aside from syntax) identical to the kind of code you would write in an imperative language like C
- Writing imperative code in Haskell often uses the **IO** monad, though there are alternatives (like the **ST** monad)
- Bad news: imperative code in Haskell is often somewhat clunky to write compared to e.g. C
  - **readArray arr i** instead of **arr[i]**
  - **writeArray arr i v** instead of **arr[i] = v**





# Conclusions

- Good news: you can write your own custom imperative control flow constructs directly in Haskell
  - like `when`, `unless`, `whileIO` (previous lecture)
- Good news: Haskell's type system keeps imperative and non-imperative code from interfering with each other
  - imperative code doesn't compromise purity of functional code





# Quote

- From Simon Peyton-Jones, lead developer of **ghc**:

"Haskell is the world's finest imperative programming language!"





# Coming up

- The list monad
- Error-handling monads

