JFP **13** (1): 169–172, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803001710 Printed in the United Kingdom

Chapter 15

Indexing Operations

```
module Ix ( Ix(range, index, inRange, rangeSize) ) where

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

instance                   Ix Char      where ...
instance                   Ix Int       where ...
instance                   Ix Integer   where ...
instance  (Ix a, Ix b)  => Ix (a,b)     where ...
-- et cetera
instance                   Ix Bool      where ...
instance                   Ix Ordering  where ...
```

The `Ix` class is used to map a contiguous subrange of values in a type onto integers. It is used primarily for array indexing (see Chapter 16). The `Ix` class contains the methods `range`, `index`, and `inRange`. The `index` operation maps a bounding pair, which defines the lower and upper bounds of the range, and a subscript, to an integer. The `range` operation enumerates all subscripts; the `inRange` operation tells whether a particular subscript lies in the range defined by a bounding pair.

An implementation is entitled to assume the following laws about these operations:

```
instance  (Ix a, Ix b)  => Ix (a,b) where
        range ((l,l’),(u,u’))
                = [(i,i’) | i <- range (l,u), i’ <- range (l’,u’)]
        index ((l,l’),(u,u’)) (i,i’)
                =  index (l,u) i * rangeSize (l’,u’) + index (l’,u’) i’
        inRange ((l,l’),(u,u’)) (i,i’)
                = inRange (l,u) i && inRange (l’,u’) i’
-- Instances for other tuples are obtained from this scheme:
--
--  instance  (Ix a1, Ix a2, ... , Ix ak) => Ix (a1,a2,...,ak)  where
--      range ((l1,l2,...,lk),(u1,u2,...,uk)) =
--          [(i1,i2,...,ik) | i1 <- range (l1,u1),
--                            i2 <- range (l2,u2),
--                            ...
--                            ik <- range (lk,uk)]
--
--      index ((l1,l2,...,lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--        index (lk,uk) ik + rangeSize (lk,uk) * (
--         index (lk-1,uk-1) ik-1 + rangeSize (lk-1,uk-1) * (
--          ...
--           index (l1,u1)))
--
--      inRange ((l1,l2,...lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--          inRange (l1,u1) i1 && inRange (l2,u2) i2 &&
--              ... && inRange (lk,uk) ik
```

Figure 15.1: Derivation of Ix instances

```
range (l,u) !! index (l,u) i == i   -- when i is in range
inRange (l,u) i              == i ‘elem‘ range (l,u)
map index (range (l,u))      == [0..rangeSize (l,u)]
```

15.1 Deriving Instances of Ix

It is possible to derive an instance of `Ix` automatically, using a `deriving` clause on a `data` declaration (Section 4.3.3 of the Language Report). Such derived instance declarations for the class `Ix` are only possible for enumerations (i.e. datatypes having only nullary constructors) and single-constructor datatypes, whose constituent types are instances of `Ix`. A Haskell implementation must provide `Ix` instances for tuples up to at least size 15.

- For an *enumeration*, the nullary constructors are assumed to be numbered left-to-right with the indices being 0 to $n - 1$ inclusive. This is the same numbering defined by the `Enum` class. For example, given the datatype:

  ```
  data Colour = Red | Orange | Yellow | Green
              | Blue | Indigo | Violet
  ```

https://doi.org/10.1017/S0956796803001710 Published online by Cambridge University Press

we would have:

```
range   (Yellow,Blue)        ==  [Yellow,Green,Blue]
index   (Yellow,Blue) Green  ==  1
inRange (Yellow,Blue) Red    ==  False
```

- For *single-constructor datatypes*, the derived instance declarations are as shown for tuples in Figure 15.1.

15.2 Library Ix

```
module Ix ( Ix(range, index, inRange, rangeSize) ) where

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

    rangeSize b@(l,h) | null (range b) = 0
                      | otherwise      = index b h + 1
        -- NB: replacing "null (range b)" by  "not (l <= h)"
        -- fails if the bounds are tuples.  For example,
        --      (1,2) <= (2,1)
        -- but the range is nevertheless empty
        --      range ((1,2),(2,1)) = []

instance  Ix Char  where
    range (m,n)         = [m..n]
    index b@(c,c') ci
        | inRange b ci  =  fromEnum ci - fromEnum c
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (c,c') i    =  c <= i && i <= c'

instance  Ix Int  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  i - m
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (m,n) i     =  m <= i && i <= n

instance  Ix Integer  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  fromInteger (i - m)
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (m,n) i     =  m <= i && i <= n

instance (Ix a,Ix b) => Ix (a, b) -- as derived, for all tuples
instance Ix Bool                  -- as derived
instance Ix Ordering              -- as derived
instance Ix ()                    -- as derived
```

https://doi.org/10.1017/S0956796803001710 Published online by Cambridge University Press

https://doi.org/10.1017/S0956796803001710 Published online by Cambridge University Press