JFP **13** (1): 235–240, January 2003.
DOI: 10.1017/S0956796803002910 Printed in the United Kingdom

Chapter 27

Random Numbers

```
module Random (
        RandomGen(next, split, genRange),
        StdGen, mkStdGen,
        Random( random,   randomR,
                randoms,  randomRs,
                randomIO, randomRIO ),
        getStdRandom, getStdGen, setStdGen, newStdGen
  ) where

---------------- The RandomGen class ------------------------
class RandomGen g where
  genRange :: g -> (Int, Int)
  next     :: g -> (Int, g)
  split    :: g -> (g, g)

---------------- A standard instance of RandomGen -----------
data StdGen = ...       -- Abstract

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen
```

```
---------------- The Random class ---------------------------
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)
   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]
   randomRIO :: (a,a) -> IO a
   randomIO  :: IO a

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...

---------------- The global random generator ----------------
newStdGen    :: IO StdGen
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

The `Random` library deals with the common task of pseudo-random number generation. The library makes it possible to generate repeatable results, by starting with a specified initial random number generator; or to get different results on each run by using the system-initialised generator, or by supplying a seed from some other source.

The library is split into two layers:

- A core *random number generator* provides a supply of bits. The class `RandomGen` provides a common interface to such generators.
- The class `Random` provides a way to extract particular values from a random number generator. For example, the `Float` instance of `Random` allows one to generate random values of type `Float`.

27.1 The `RandomGen` class, and the `StdGen` generator

The class `RandomGen` provides a common interface to random number generators.

```
class RandomGen g where
  genRange :: g -> (Int,Int)
  next     :: g  -> (Int, g)
  split    :: g -> (g, g)

  -- Default method
  genRange g = (minBound,maxBound)
```

https://doi.org/10.1017/S0956796803002910 Published online by Cambridge University Press

- The `genRange` operation yields the range of values returned by the generator.

  It is required that:

  - If $(a, b) =$ `genRange` g, then $a < b$.
  - `genRange` $\perp \neq \perp$.

  The second condition ensures that `genRange` cannot examine its argument, and hence the value it returns can be determined only by the instance of `RandomGen`. That in turn allows an implementation to make a single call to `genRange` to establish a generator's range, without being concerned that the generator returned by (say) `next` might have a different range to the generator passed to `next`.

- The `next` operation returns an `Int` that is uniformly distributed in the range returned by `genRange` (including both end points), and a new generator.

- The `split` operation allows one to obtain two independent random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls), but very little work has been done on statistically robust implementations of `split` (Burton and Page [2] and Hellekalek [7]] are the only examples we know of).

The `Random` library provides one instance of `RandomGen`, the abstract data type `StdGen`:

```
data StdGen = ...      -- Abstract

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen
```

The `StgGen` instance of `RandomGen` has a `genRange` of at least 30 bits.

The result of repeatedly using `next` should be at least as statistically robust as the "Minimal Standard Random Number Generator" described by Park and Miller [12] and Carta [3]. Until more is known about implementations of `split`, all we require is that `split` deliver generators that are (a) not identical and (b) independently robust in the sense just given.

The `Show`/`Read` instances of `StdGen` provide a primitive way to save the state of a random number generator. It is required that `read (show g) == g`.

In addition, `read` may be used to map an arbitrary string (not necessarily one produced by `show`) onto a value of type `StdGen`. In general, the `read` instance of `StdGen` has the following properties:

- It guarantees to succeed on any string.

https://doi.org/10.1017/S0956796803002910 Published online by Cambridge University Press

- It guarantees to consume only a finite portion of the string.
- Different argument strings are likely to result in different results.

The function `mkStdGen` provides an alternative way of producing an initial generator, by mapping an `Int` into a generator. Again, distinct arguments should be likely to produce distinct generators.

Programmers may, of course, supply their own instances of `RandomGen`.

Implementation warning. A superficially attractive implementation of `split` is

```
instance RandomGen MyGen where
  ...
  split g = (g, variantOf g)
```

Here, `split` returns `g` itself and a new generator derived from `g`. But now consider these two apparently-independent generators:

```
g1 = snd (split g)
g2 = snd (split (fst (split g)))
```

If `split` genuinely delivers independent generators (as specified), then `g1` and `g2` should be independent, but in fact they are both equal to `variantOf g`. Implementations of the above form do not meet the specification.

https://doi.org/10.1017/S0956796803002910 Published online by Cambridge University Press

27.2 The `Random` class

With a source of random number supply in hand, the `Random` class allows the programmer to extract random values of a variety of types:

```
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO :: IO a

     -- Default methods
   randoms g = x : randoms g'
                  where
                    (x,g') = random g
   randomRs = ...similar...

   randomIO        = getStdRandom random
   randomRIO range = getStdRandom (randomR range)

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...
```

- `randomR` takes a range (lo, hi) and a random number generator g, and returns a random value uniformly distributed in the closed interval $[lo, hi]$, together with a new generator. It is unspecified what happens if $lo > hi$. For continuous types there is no requirement that the values lo and hi are ever produced, but they may be, depending on the implementation and the interval.
- `random` does the same as `randomR`, but does not take a range.
  - For bounded types (instances of `Bounded`, such as `Char`), the range is normally the whole type.
  - For fractional types, the range is normally the semi-closed interval $[0, 1)$.
  - For `Integer`, the range is (arbitrarily) the range of `Int`.
- The plural versions, `randomRs` and `randoms`, produce an infinite list of random values, and do not return a new generator.

https://doi.org/10.1017/S0956796803002910 Published online by Cambridge University Press

- The `IO` versions, `randomRIO` and `randomIO`, use the global random number generator (see Section 27.3).

27.3 The global random number generator

There is a single, implicit, global random number generator of type `StdGen`, held in some global variable maintained by the `IO` monad. It is initialised automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernel random number generator. To get deterministic behaviour, use `setStdGen`.

```
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
newStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

- `getStdGen` and `setStdGen` get and set the global random number generator, respectively.
- `newStdGen` applies `split` to the current global random generator, updates it with one of the results, and returns the other.
- `getStdRandom` uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function. For example, `rollDice` gets a random integer between 1 and 6:

```
rollDice :: IO Int
rollDice = getStdRandom (randomR (1,6))
```

The Web site `http://random.mat.sbg.ac.at/` is a great source of information.

https://doi.org/10.1017/S0956796803002910 Published online by Cambridge University Press