Title

Sources of Random Bits

Author

Sebastian Egner

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-27@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Draft: 2002-02-12--2002-04-12
- Revised: 2002-04-04
- Revised: 2002-04-10
- Revised: 2002-04-10
- Final: 2002-06-03

Abstract

This document specifies an interface to sources of random bits, or "random sources" for brevity. In particular, there are three different ways to use the interface, with varying demands on the quality of the source and the amount of control over the production process:

- The "no fuss" interface specifies that `(random-integer` *n*`)` produces the next random integer in {0, ..., *n*-1} and `(random-real)` produces the next random real number between zero and one. The details of how these random values are produced may not be very relevant, as long as they appear to be sufficiently random.
- For simulation purposes, on the contrary, it is usually necessary to know that the numbers are produced deterministically by a pseudo random number generator of high quality and to have explicit access to its state. In addition, one might want to use several independent sources of random numbers at the same time and it can be useful to have some simple form of randomization.
- For security applications a serious form of true randomization is essential, in the sense that it is difficult for an adversary to exploit or introduce imperfections into the distribution of random bits. Moreover, the linear complexity of the stream of random bits is more important than its statistical properties. In these applications, an entropy source (producing truely random bits at a low rate) is used to randomize a pseudo random number generator to increase the rate of available bits.

Once random sources provide the infrastructure to obtain random bits, these can be used to construct other random deviates. Most important are floating point numbers of various distributions and random discrete structures, such as permutations or graphs. As there is an essentially unlimited number of such objects (with limited use elsewhere), we do not include them in this SRFI. In other words, this SRFI is *not* about making all sorts of random objects---it is about obtaining random bits in a portable, flexible, reliable, and efficient way.

Rationale

This SRFI defines an interface for sources of random bits computed by a pseudo random number generator. The interface provides range-limited integer and real numbers. It allows accessing the state of the underlying generator. Moreover, it is possible to obtain a large number of independent generators and to invoke a mild form of true randomization.

The design aims at sufficient flexibility to cover the usage patterns of many applications as diverse as discrete structures, numerical simulations, and cryptographic protocols. At the same time, the interface aims at simplicity, which is important for occasional use. As there is no "one size fits all" random number generator, the design necessarily represents some form of compromise between the needs of the various applications.

Although strictly speaking not part of the specification, the emphasis of this proposal is on *high quality* random numbers and on *high performance*. As the state of the art in pseudo random number generators is still advancing considerably, the choice of method for the reference implementation should essentially be considered preliminary.

Specification

`(random-integer` *n*`) -> ` *x*

The next integer *x* in {0, ..., *n*-1} obtained from `default-random-source`. Subsequent results of this procedure appear to be independent uniformly distributed over the range {0, ..., *n*-1}. The argument *n* must be a positive integer, otherwise an error is signalled.

`(random-real) -> ` *x*

The next number 0 < *x* < 1 obtained from `default-random-source`. Subsequent results of this procedure appear to be independent uniformly distributed. The numerical type of the results and the quantization of the output range depend on the implementation; refer to `random-source-make-reals` for details.

`default-random-source`

A random source from which `random-integer` and `random-real` have been derived using `random-source-make-integers` and `random-source-make-reals`. Note that an assignment to `default-random-source` does not change

`random` or `random-real`; it is also strongly recommended not to assign a new value.

`(make-random-source) -> s`

Creates a new random source *s*. Implementations may accept additional, optional arguments in order to create different types of random sources. A random source created with `make-random-source` represents a deterministic stream of random bits generated by some form of pseudo random number generator. Each random source obtained as `(make-random-source)` generates the same stream of values, unless the state is modified with one of the procedures below.

`(random-source? obj) -> bool`

Tests if *obj* is a random source. Objects of type random source are distinct from all other types of objects.

`(random-source-state-ref s) -> state`
`(random-source-state-set! s state)`

Get and set the current state of a random source *s*. The structure of the object *state* depends on the implementation; the only portable use of it is as argument to `random-source-state-set!`. It is, however, required that a state possess an external representation.

`(random-source-randomize! s)`

Makes an effort to set the state of the random source *s* to a truly random state. The actual quality of this randomization depends on the implementation but it can at least be assumed that the procedure sets *s* to a different state for each subsequent run of the Scheme system.

`(random-source-pseudo-randomize! s i j)`

Changes the state of the random source *s* into the initial state of the (*i*, *j*)-th independent random source, where *i* and *j* are non-negative integers. This procedure provides a mechanism to obtain a large number of independent random sources (usually all derived from the same backbone generator), indexed by two integers. In contrast to `random-source-randomize!`, this procedure is entirely deterministic.

`(random-source-make-integers s) -> rand`

Obtains a procedure *rand* to generate random integers using the random source *s*. *Rand* takes a single argument *n*, which must be a positive integer, and returns the next uniformly distributed random integer from the interval {0, ..., *n*-1} by advancing the state of the source *s*.

If an application obtains and uses several generators for the same random source *s*, a call to any of these generators advances the state of *s*. Hence, the generators *do not* produce the same sequence of random integers each but rather share a state. This also holds for all other types of generators derived from a fixed random sources.

Implementations that support concurrency make sure that the state of a generator is properly advanced.

`(random-source-make-reals` *s*`) -> `*rand*
`(random-source-make-reals` *s unit*`) -> `*rand*

Obtains a procedure *rand* to generate random real numbers $0 < x < 1$ using the random source *s*. The procedure *rand* is called without arguments.

The optional parameter *unit* determines the type of numbers being produced by *rand* and the quantization of the output. *Unit* must be a number such that $0 < unit < 1$. The numbers created by *rand* are of the same numerical type as *unit* and the potential output values are spaced by at most *unit*. One can imagine *rand* to create numbers as *x*unit* where *x* is a random integer in {1, ..., floor(1/*unit*)-1}. Note, however, that this need not be the way the values are actually created and that the actual resolution of *rand* can be much higher than *unit*. In case *unit* is absent it defaults to a reasonably small value (related to the width of the mantissa of an efficient number format).

Design Rationale

Why not combine `random-integer` and `random-real`?

The two procedures are not combined into a single variable-arity procedures to save a little time and space during execution. Although some Scheme systems can deal with variable arity as efficiently as with fixed arity this is not always the case and time efficiency is very important here.

Why not some object-oriented interface?

There are many alternatives to the interface as specified in this SRFI. In particular, every framework for object-orientation can be used to define a class for random sources and specify the interface for the methods on random sources. However, as the object-oriented frameworks differ considerably in terms of syntax and functionality, this SRFI does not make use of any particular framework.

Why is there not just a generator with a fixed range?

A bare fixed-range generator is of very limited use. Nearly every application has to add some functionality to make use of the random numbers. The most fundamental task in manipulating random numbers is to change the range and quantization. This is exactly what is provided by `random-integer` and `random-real`. In addition, is saves the user from the pitfall of changing the range with a simple `modulo`-computation which may substantially reduce the quality of the numbers being produced.

The design of the interface is based on three prototype applications:

1. Repeatedly choose from relatively small sets: As the size of the set is likely to vary from call to call, `random-integer` accepts a range argument *n* in every call. The implementation should try to avoid boxing/unboxing of values if the ranges fit into immediate integers.
2. Generate a few large integers with a fixed number of bits: As generating the random number itself is expensive, passing the range argument in every call does not hurt performance. Hence, the same interface as in the first application can be used.
3. Generate real numbers: Unlike the choose-from-set case, the range and the quantization is constant over a potentially very large number of calls. In addition, there are usually just a few distinct instances of quantization and number type, most likely corresponding to underlying `float` and `double` representations. Therefore, `random-real` does not accept any parameters but the procedure `random-source-make-reals` creates a properly configured `random-real` procedure.

Why bother about floating point numbers at all?

A proper floating point implementation of a random number generator is potentially much more efficient that an integer implementation because it can use more powerful arithmetics hardware. If in addition the application needs floating point random numbers it would be an intolerable waste to run an integer generator to produce floating point random numbers. A secondary reason is to save the user from the 'not as easy as it seems' task of converting an integer generator into a real generator.

Why are zero and one excluded from `random-real`?

The procedure `random-real` does not return $x = 0$ or $x = 1$ in order to allow `(log` *x*`)` and `(log (- 1` *x*`))` without the danger of a numerical exception.

Implementation

Choice of generator

The most important decision about the implementation is the choice of the random number generator. The basic principle here is: *Let quality prevail!* In the end, a performance penalty of a better generator may be a cheap price to pay for some avoided catastrophes. It may be unexpected, but I have also seen many examples where the better generator was also the faster. Simple linear congruential generator cannot be recommended as they tend to be ill-behaved in several ways.

For this reason, my initial proposal was George Marsaglia's COMBO generator, which is the combination of a 32-bit multiplicative lagged Fibonacci-generator with a 16-bit multiply with carry generator. The COMBO generator passes all tests of Marsaglia's DIEHARD testsuite for random number generators and has a period of order 2^60.

As an improvement, Brad Lucier suggested suggested Pierre L'Ecuyer's MRG32k3a generator which is combination of two recursive generators of degree three, both of which fit into 54-bit arithmetics. The MRG32k3a generator also passes DIEHARD and in addition, has desireable spectral properties and a period in the order of 2^191. As a matter of fact, multiple recursive generators (MRGs) are theoretically much better understood than special constructions as the COMBO generator. This is the reason why the implementations provided here implements the MRG32k3a generator. When implemented in floating point arithmetics with sufficient mantissa-width, this generator is also very fast.

Choice of arithmetics

The next important decision about the implementation is the type of arithmetics to be used. The choice is difficult and depends heavily on the underlying Scheme system and even on the underlying hardware platform and architecture. For the MRG32k3a generator, use 64-bit arithmetics if you really have it. If not, use a floating point ALU if it gives you 54 or more bits of mantissa. And if you do not have floats either, then at least try to make sure you work with immediate integers (instead of allocated objects). Unfortunately, there is no portable way in Scheme to find out about native and emulated arithmetics.

As performance is critical to many applications, one might want to implement the actual generator itself in native code. For this reason, I provide three different implementations of the backbone generator as a source of inspiration. See the code below.

Data Type for Random Sources

An important aspect of the specification in this SRFI is that random sources are objects of a distinct type. Although this is straight-forward and available in nearly every Scheme implementation, there is no portable way to do this at present. One way to define the record type is to use SRFI-9.

The reference implementations below define a record type to contain the exported procedures. The actual state of the generator is stored in the binding time environment of `make-random-source`. This has the advantage that access to the state is fast even if the record type would be slow (which need not be the case).

Entropy Source for Randomization

Another problematic part of the specification with respect to portability is `random-source-randomize!` as it needs access to a real entropy source.

A reasonable choice for such as source is to use the system clock in order to obtain a value for randomization, for example in the way John David Stone recommends (see reference below). This is good enough for most applications with the notable exception of security related programs. One way to obtain the time in Scheme is to use SRFI-19.

Implementation of the specified interface

Once the portability issues are resolved, one can provide the remaining functionality as specified in this SRFI document.

For the reference implementation, a relatively large part of the code deals with the more advanced features of the MRG32k3a generator, in particular `random-source-pseudo-randomize!`. This code is inspired by Pierre L'Ecuyer's own implementation of the MRG32k3a generator.

Another part of this generic code deals with changing the range and quantization of the random numbers and with error checking to detect common mistakes and abuses.

Implementation Examples

Here are three alternative implementations of the SRFI. (Here are all files, tar-gzipped, 53K bytes.) Keep in mind that a SRFI is a "request for implementation", which means these implementations are merely examples to illustrate the specification and inspire people to implement it better and faster. The performance figures below are rough indications measured on a Pentium3, 800 Mhz, Linux; *x* int/s, *y* real/s means `(random-integer 2)` can be computed about *x* times a second and `(random-real)` about *y* times a second. The implementations are

a. for Scheme 48 0.57, using 54-bit `integer` only. This implementation aims at portability, not at performance (30000 ints/s, 3000/s reals/s).
b. for Scheme 48 0.57 with the core generator being implemented in C using `(double)`-arithmetics. The generator is made available in Scheme 48 via the C/Scheme interface. The performance of this generator is good (160000 ints/s, 180000 reals/s).
c. for Gambit 3.0, using `flonum` and 54-bit `integer`. This code is inspired by a program by Brad Lucier as posted to the discussion archive of this SRFI. The performance of this generator is good when compiled (5000 ints/s, 25000/s reals/s when interpreted, 200000 ints/s, 400000/s reals/s when compiled; see acknowledgements).

In addition to the implementations there is a small collection of confidence tests for the interface specified. The tests merely check a few assertions expressed by the specification. It is not the intention to provide a complete test of the interface here. It is even less the intention to provide statistical tests of the generator itself. However, there is a function to write random bits from the generators to a file in a way readable by the *DIEHARD* testsuite. This makes it easier for implementors to find out about their favorite generators and check their implementation.

Recommended Usage Patterns

Unless the functionality defined in this SRFI is sufficient, an application has to implement more procedures to construct other random deviates. This section contains some

recommendation on how to do this technically by presenting examples of increasing difficulty with respect to the interface. Note that the code below is not part of the specification, it is merely meant to illustrate the spirit

Generating Random Permutations

The following code defines procedures to generate random permutations of the set {0, ..., *n*-1}. Such a permutation is represented by a `vector` of length *n* for the images of the points.

Observe that the implementation first defines the procedure `random-source-make-permutations` to turn a random source *s* into a procedure to generate permutations of given degree *n*. In a second step, this is applied to the default source to define a ready-to-use procedure for permutations: `(random-permutation` *n*`)` constructs a random permutation of degree *n*.

```
(define (random-source-make-permutations s)
  (let ((rand (random-source-make-integers s)))
    (lambda (n)
      (let ((x (make-vector n 0)))
        (do ((i 0 (+ i 1)))
            ((= i n))
          (vector-set! x i i))
        (do ((k n (- k 1)))
            ((= k 1) x)
          (let* ((i (- k 1))
                 (j (rand k))
                 (xi (vector-ref x i))
                 (xj (vector-ref x j)))
            (vector-set! x i xj)
            (vector-set! x j xi)))))))

(define random-permutation
  (random-source-make-permutations default-random-source))
```

For the algorithm refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Algorithm P of Section 3.4.2.

Generating Exponentially-Distributed Random Numbers

The following code defines procedures to generate exponentially Exp(mu)-distributed random numbers. The technical difficulty of the interface addressed here is how to pass optional arguments to `random-source-make-reals`.

```
(define (random-source-make-exponentials s . unit)
  (let ((rand (apply random-source-make-reals s unit)))
    (lambda (mu)
```

```
      (- (* mu (log (rand)))))))

(define random-exponential
  (random-source-make-exponentials default-random-source))
```

The algorithm is folklore. Refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Section 3.4.1.D.

Generating Normally-Distributed Random Numbers

The following code defines procedures to generate normal N(mu, sigma)-distributed real numbers using the polar method.

The technical difficulty of the interface addressed here is that the polar method generates two results per computation. We return one of the result and store the second one to be returned by the next call to the procedure. Note that this implies that `random-source-state-set!` (and the other procedures modifying the state) does not necessarily affect the output of `random-normal` immediately!

```
(define (random-source-make-normals s . unit)
  (let ((rand (apply random-source-make-reals s unit))
        (next #f))
    (lambda (mu sigma)
      (if next
          (let ((result next))
            (set! next #f)
            (+ mu (* sigma result)))
          (let loop ()
            (let* ((v1 (- (* 2 (rand)) 1))
                   (v2 (- (* 2 (rand)) 1))
                   (s (+ (* v1 v1) (* v2 v2))))
              (if (>= s 1)
                  (loop)
                  (let ((scale (sqrt (/ (* -2 (log s)) s))))
                    (set! next (* scale v2))
                    (+ mu (* sigma scale v1)))))))))))

(define random-normal
  (random-source-make-normals default-random-source))
```

For the algorithm refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Algorithm P of Section 3.4.1.C.

Acknowledgements

I would like to thank all people who have participated in the discussion, in particular Brad Lucier and Pierre l'Ecuyer. Their contributions have greatly improved the design of this SRFI. Moreover, Brad has optimized the Gambit implementation quite substantially.

References

1. G. Marsaglia: Diehard -- Testsuite for Random Number Generators. stat.fsu.edu/~geo/diehard.html (Also contains some generators that do pass Diehard.)
2. D. E. Knuth: The Art of Computer Programming; Volume II Seminumerical Algorithms. 2nd ed. Addison-Wesley, 1981. (The famous chapter on random number generators.)
3. P. L'Ecuyer: "Software for Uniform Random Number Generation: Distinguishing the Good and the Bad", Proceedings of the 2001 Winter Simulation Conference, IEEE Press, Dec. 2001, 95--105. www.iro.umontreal.ca/~lecuyer/myftp/papers/wsc01rng.pdf (Profound discussion of random number generators.)
4. P. L'Ecuyer: "Good Parameter Sets for Combined Multiple Recursive Random Number Generators", Shorter version in Operations Research, 47, 1 (1999), 159--164. www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.ps (Actual numbers for good generators.)
5. P. L'Ecuyer: "Software for Uniform Random Number Generation: Distinguishing the Good and the Bad", Proceedings of the 2001 Winter Simulation Conference, IEEE Press, Dec. 2001, 95--105.
6. MIT Scheme v7.6: `random flo:random-unit *random-state* make-random-state random-state?` http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme_5.html#SEC53 (A mechanism to run a fixed unspecified generator.)
7. A. Jaffer: SLIB 2d2 with (require 'random): `random *random-state* copy-random-state seed->random-state make-random-state random:uniform random:exp random:normal-vector! random-hollow-sphere! random:solid-sphere!` http://swiss.csail.mit.edu/~jaffer/slib_5.html#SEC108 (Uses RC-4.)
8. R. Kelsey, J. Rees: Scheme 48 v0.57 'random.scm': `make-random make-random-vector` (Internal procedures of Scheme48; a fixed 28-bit generator.)
9. M. Flatt: PLT MzScheme Version 200alpha1: `random random-seed current-pseudo-random-generator make-pseudo-random-generator pseudo-random-generator?` http://download.plt-scheme.org/doc/200alpha1/html/mzscheme/mzscheme-Z-H-3.html#%_idx_144 (A mechanism to run a generator and to exchange the generator.)
10. H. Abelson, G. J. Sussmann, J. Sussman: Structure and Interpretation of Computer Programs. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-20.html#%_idx_2934 (The `rand`-example shows a textbook way to define a random number generator.)
11. John David Stone: A portable random-number generator. http://www.math.grin.edu/~stone/events/scheme-workshop/random.html (An implementation of a linear congruental generator in Scheme.)

12. Network Working Group: RFC1750: Randomness Recommendations for Security. http://www.cis.ohio-state.edu/htbin/rfc/rfc1750.html (A serious discussion of serious randomness for serious security.)
13. http://www.random.org/essay.html
http://www.taygeta.com/random/randrefs.html (Resources on random number generators and randomness.)

Copyright

Editor: *Mike Sperber*
Author: *Sebastian Egner*
Last modified: Wed Jul 21 08:44:49 MST 2010