Clotho: A Racket Library for Parametric Randomness

Pierce Darragh
pdarragh@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

William Gallard Hatch
william@hatch.uno
University of Utah
Salt Lake City, UT, USA

Eric Eide
eeide@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Abstract

Programs such as simulators and fuzz testers often use randomness to walk through a large state space in search of interesting paths or outcomes. These explorations can be made more efficient by employing heuristics that "zero-in" on paths through the state space that are more likely to lead to interesting solutions. Given one path that exhibits a desired property, it may be beneficial to generate and explore similar paths to determine if they produce similarly interesting results. When the random decisions made during this path exploration can be manipulated in such a way that they correspond to discrete structural changes in the result, we call it *parametric randomness*.

Many programming languages, including Racket, provide only simple randomness primitives, making the implementation of parametric randomness somewhat difficult. To address this deficiency, we present Clotho: a Racket library for parametric randomness, designed to be both easy to use and flexible. Clotho supports multiple strategies for using parametric randomness in Racket applications without hassle.

1 Introduction

There are many applications in which a developer may want to use pseudo-random number generators (PRNGs) to explore a given search space while using the results of previous explorations to inform choices in subsequent navigation of the space. Examples include:

- Generating many random programs that share a common attribute.
- Producing sentences from a grammar with a common prefix.
- Walking a large graph, such as that of a social network, without changing an initial portion of the walk.
- Implementing a genetic algorithm.
- Modeling multiple, similar paths in a simulation.

Scheme '20, August 28, 2020, Jersey City, NJ, USA

Many mainstream programming languages only provide very simple randomness primitives and leave the more complex uses of these functions up to developers to implement on a per-case basis. This can be tedious and is prone to error.

While working on a random program generator (§2.1), we found ourselves in need of a system for manipulating the outcomes of randomness functions in a predictable manner. We wanted to "record" a sequence of randomly generated values, modify that sequence in some way, and then feed the modified recording back to our system to get a new—but similar—sequence of randomly generated values. Crucially, unchanged portions of the recording must produce the same results in subsequent executions as they did in the original generation. We call this process *parametric randomness*.

We define parametric randomness as a kind of random value generation that is amenable to predictable external manipulation. For example, consider a random value sequence generated without any manipulation: `[4, 8, 15, 16, 23, 42]`. After generating this initial sequence, one could employ parametric randomness to produce new sequences that are similar to the original:

- `[4, 8, 15, 16, 17]`
- `[4, 8, 15, 16, 23, 43]`
- `[4, 8, 15, 16, 23, 19, 68]`
- `[4, 8, 12, 16, 23, 42]`
- `[4, 3, 15, 16, 37, 42]`

Imagine that in each of these sequences, the values correspond to the choices made by a program that randomly explores a decision tree. This collection of sequences therefore represents multiple explorations of the decision tree. The paths exhibit some similar properties, but may lead to fundamentally different outcomes by the search program.

We have developed a Racket library, Clotho, that enables developers to easily engage in this style of search-space exploration with parametric randomness.[1] Our library implements the following functionality:

1. All the existing randomness functionality of `racket/base` and `racket/random`, which define the Racket standard library's randomness functions.
2. Convenience functions for generating common values using parametric random generation functions (e.g.,

[1] The library can be installed from the official Racket package catalog via `raco pkg install clotho`. Alternatively, the source code can be downloaded at https://gitlab.flux.utah.edu/xsmith/clotho.

Booleans, integers, Unicode characters, and Unicode strings).
3. Support for parametric randomness by specifying a seed sequence.
4. A macro for wrapping functions that use Racket's built-in randomness functions, to ensure the use of parametric randomness outside of the functions specified in our library.
5. A macro for supporting randomness abstraction (§5.2).

In this paper we explain the motivation, design, and implementation of Clotho with examples along the way. We present:

- Motivation for parametric randomness, and background on why Racket's existing randomness functionality is insufficient for our purposes. (§2)
- A high-level overview of the functionality provided by Clotho. (§3)
- Explanation of the under-the-hood implementation that powers Clotho's parametric generation functionality. (§4)
- Detailed examples that illustrate the capabilities of Clotho. (§5)
- Discussion on the limitations of, and potential future work for, Clotho. (§6)

2 Background

2.1 Motivation

We developed Clotho as part of an implementation of a random program generation tool [Hatch et al. 2020]. *Random program generation* is the process of automatically creating whole programs without human input. This technique has proven especially useful in the domain of testing programming language compilers and interpreters, where human-written tests can miss edge cases or are otherwise insufficient to trigger bugs in the implementation [Padhye et al. 2019; Yang et al. 2011].

When we use our tool, we sometimes want to generate programs that are similar to those that the tool has generated previously. One way to achieve this is with a *parametric generator* [Padhye et al. 2019]: a generator that inputs a sequence that encodes the "random" choices that the generator will make. The generator processes the input sequence and outputs a new test case. The crucial characteristic of a parametric generator is that simple (e.g., bit-level) modifications to the input sequence result in structural changes to the generated output in a relatively predictable manner. Essentially, each primitive element of the input sequence is a parameter that can be adjusted to modify the output.

If we consider that a random program generator is a function that randomly walks a path in the decision tree of possible output programs, then a parametric generator is one that exposes its decisions as parameters that can be tuned. Coupled with an external metric for recognizing "interesting" generator outputs—e.g., test cases that extend code coverage of the compiler or interpreter under test—a parametric random program generator can be driven to generate more interesting programs over time.

```
(require racket/random)

(random-seed 0)
(random)  ;; => 0.8571568490678037
(random)  ;; => 0.6594215608573717
(random)  ;; => 0.205654820840853
(random-seed 0)
(random)  ;; => 0.8571568490678037
(random)  ;; => 0.6594215608573717
(parameterize
    ([current-pseudo-random-generator
      (make-pseudo-random-generator)])
  (random-seed 0)
  (random)   ;; => 0.8571568490678037
  (random))  ;; => 0.6594215608573717
(random)  ;; => 0.205654820840853
```

Figure 1. Randomness in Racket using `random-seed`, where comments show the result of each `random` call. The PRNG created in the body of the `parameterize` expression has no impact on the PRNG that exists externally.

Clotho arose from our implementation of random program generation in Racket, but we have made Clotho a standalone package because it can be used in other domains as well.

2.2 What Racket Offers

Racket provides a number of functions for generating random values in its standard library, as well as some methods for directly manipulating the current source of randomness (a pseudo-random number generator, or PRNG) to make the outputs of randomness functions manipulable. However, we found Racket's built-in functionality to be lacking in expressive capability on its own.

The most fundamental of Racket's randomness functions is `random`. The `random` function, when called without any arguments, produces an inexact number in the interval $[0, 1)$ with uniform probability.

Behind the scenes, Racket uses a system-wide PRNG to generate random values on demand. This PRNG is a *parameter*[2] in Racket, which means that its value can be dynamically re-bound in a local context with a `parameterize` expression. The parameter's name is `current-pseudo-random-generator`, which we will call CPRG for short, and it conforms to the type predicate `pseudo-random-generator?`. The CPRG is instantiated automatically at run time without

[2]The term "parameter" is unfortunately overloaded in this paper by necessity. When referring to "Racket parameters" or the `parameterize` form, we mean the Racket-specific concept of a parameter as explained here: https://docs.racket-lang.org/reference/parameters.html.

any configuration, and it is used implicitly in all of the standard library randomness functions, so the average user never needs to interact with it directly.

For our purposes, though, direct interaction is necessary. Fortunately, Racket provides some mechanisms to manipulate the CPRG, which we could potentially use to induce parametric behavior.

The first mechanism lies in the `random-seed` function. This is a side-effecting function that takes as argument an integer in the interval $[0, 2^{31})$, and then uses that value to seed the CPRG. The values returned by subsequent calls to any of the randomness functions, such as `random`, are determined by whatever seed was passed to `random-seed`. This effectively means that the randomness can be manipulated indirectly by choosing seed values.

The second mechanism is to create an entirely new PRNG by calling either `make-pseudo-random-generator` (which takes no arguments and automatically creates a new PRNG seeded by the current system time) or `vector->pseudo-random-generator` (which takes a specially formatted vector as argument and produces a new PRNG object from an algorithm performed on those values). This new PRNG, which also conforms to `pseudo-random-generator?`, can then be used as the Racket-wide CPRG by using a `parameterize` expression. Within the body of this expression, all randomness is handled by the newly created PRNG.

One can see an example of some of these mechanisms in action in Figure 1. Unfortunately, there are some problems with these approaches:

1. They are unwieldy to use for manipulating randomness repeatedly.
2. Only the seeds are manipulable. It is quite difficult to implement a system using the provided mechanisms where we want to be able to manipulate *some* of the randomly generated values output by a PRNG.
3. Subsequently, there is no way to parameterize the random generation in the manner needed by a parametric generator.

Because we want to use parametric randomness to explore state spaces, it is points 2 and 3 that are the main concerns for us (though 1 is relevant in terms of library design). We want to be able to "replay" a sequence of random generations up to a point, and then deviate. Racket's provided functionality does not make this easy.

One might consider the possibility of simply replacing the CPRG with a custom value that conforms to the `pseudo-random-generator?` predicate. However, `pseudo-random-generator?` is not open to external implementation: the only values that conform are those produced by either `make-pseudo-random-generator` or `vector->pseudo-random-generator`. This severely limits the ability of developers to implement custom random-generation mechanisms.

At this point, we have a choice: do we develop an entirely new PRNG system separate from Racket's existing functionality, or do we attempt to wrap what Racket provides?

Although it may be tempting to implement everything fresh, we chose to wrap Racket's existing randomness functions. Many existing libraries depend on these functions (such as the `math/distributions` module in the standard library), and a completely custom PRNG solution would jeopardize support for these libraries within our random program-generation tool. Supporting these libraries is important to us (because we want to use them without implementing their functionality ourselves), so our decision is made for us: we wrap Racket's existing randomness functionality to support the parametricity we desire.

3 Design

Clotho's client-facing interface has three parts:

1. A `current-pseudo-random-generator`-like parameter that controls random generation and provides an interface for users to manipulate random generation as needed.
2. Two macros for enabling advanced functionality.
3. Various convenience functions to make random generation simpler, e.g., `random-int` and `random-bool`.

This section focuses on items 1 and 2, as the convenience functions of 3 are not interesting on their own. Where convenience functions are used in examples in this paper, they will be summarized appropriately on a case-by-base basis.

3.1 The Parameter and Its Maker

The underlying functionality of Clotho is managed by the `current-random-source` parameter, which we abbreviate CRS hereafter. All Clotho randomness functions must be called within a context in which the CRS has been parameterized:

```
(parameterize
    ([current-random-source ...])
  ...)
```

A new CRS is created by using the `make-random-source` function. This function can accept arguments in a few forms.

When called with *zero arguments*, `make-random-source` functions very similarly to Racket's built-in `make-pseudo-random-generator` function for creating PRNGs. Essentially, it will generate a new randomly seeded `random-source?` that is suitable for parameterization.

Alternatively, `make-random-source` can be called with *an integer argument*. In this case, that integer is treated as a random seed value. This initializes the generated `random-source?` deterministically (i.e., using the same seed value repeatedly will produce identical results each time).

Lastly, *a byte string argument* can be supplied. Clotho views this byte string as a sequence of four-byte integers. The first such integer is used internally and will be explained

Table 1. Clotho's external API, leaving out most convenience functions.

Function/Value Name	Brief Explanation
`current-random-source`	Parameter responsible for generating random values.
`random-source?`	Type predicate that tests whether the argument is a valid `current-random-source` parameter.
`make-random-source`	Generates a new `random-source?`.
`current-random-source-initialized?`	Tests if `current-random-source` is initialized.
`assert-current-random-source-initialized!`	Raise an error if `current-random-source` is not initialized.
`get-current-random-source-byte-string`	Returns the byte string from `current-random-source`.
`wrap-external-randomness`	Wraps randomness functions defined outside Clotho.
`with-new-random-source`	Wraps a region with a new `current-random-source`.
`random`	Like Racket's `random`, but using `current-random-source`.

later (§4). All subsequent integers are used to deterministically generate random values. (When a byte string is supplied that is not divisible into four-byte segments, it is padded with 0-value bytes at the end.)

3.2 The Macros

Clotho provides two macros to help with some of the more common advanced usage scenarios.

The `wrap-external-randomness` form enables the use of externally defined randomness functions within the parametric framework of Clotho. These "external" functions consult Racket's `cprg` directly (§3.5). This macro allows Clotho to be compatible with any such existing functions or libraries with very little effort on the part of the Clotho user.

The `with-new-random-source` form is a shorter way of parameterizing the `current-random-source`. Specifically, it uses the `current-random-source` to generate a new seed value, then creates a *new* `random-source?` and installs it as the `current-random-source` parameter. This enables *randomness abstraction*, which is elaborated upon in §5.

3.3 The #langs

In addition to the forms described above, Clotho provides a few `#lang`s. They are:

- `#lang clotho/racket/base`: Provides all of the bindings of `racket/base`, but without any randomness functionality.
- `#lang clotho`: Provides all of the bindings of Clotho, as well as all the bindings of `clotho/racket/base`. This is likely to be the most useful for most people.
- `#lang clotho/stateful`: In addition to providing the bindings of `#lang clotho`, this `#lang` also initializes a global `current-random-source` so that random generation can be performed imperatively, similar to how `#lang racket/base` works.

Of course, these can also be used as simple `require` forms. We find that interactions in the Racket REPL are greatly improved by starting the session with `(require clotho/stateful)`, because this enables easy execution of randomness functions without parameterizing the CRS each time. There are additional bindings provided in the module to interact with the random source, which are described in Clotho's documentation.

3.4 API Summary

Table 1 summarizes Clotho's exposed interface. The table omits Clotho's convenience functions for obtaining random data of various types, but they are straightforward: e.g., `random-bool` returns a random Boolean value, `random-int` produces a random signed integer value, and so on. Clotho also provides `random-ref`, which works identically to the Racket-provided function of the same name: i.e., when given a list, it returns an element of that list selected at random with a uniform distribution.

The `get-current-random-source-byte-string` function returns a byte string that encodes the history of values that have been returned by the CRS. This byte string can be used as-is to initialize a new `random-source?` that will replay the recorded values, assuming that the same sequence of calls is made to draw random values from the new source. Alternatively, one can use a *mutation* of the byte string to create a `random-source?` that will produce a *modified* sequence of values.

3.5 An Example

Figure 2 shows the core functionality in action. In the left-hand column, the example defines a `card` struct and some lists describing the suits and values that a card can have. The `random-card` function randomly selects a suit and a value by using the `random-ref` function. Note that the `current-random-source` parameter has not been seen yet; that will come later. The `make-deck` function builds an ordered deck of cards, and the `random-deck` function returns a randomly shuffled deck. We will discuss its use of the `wrap-external-randomness` macro at the end of this section.

Now look the right-hand column of Figure 2. The first segment of code shows the CRS being parameterized with a

```
#lang clotho
(require racket/list)

(struct card (suit value) #:transparent)
(define card-suits
  '(♠ ♡ ♣ ♢))
(define card-values
  '(A 2 3 4 5 6 7 8 9 10 J Q K))

(define (random-card)
  (card (random-ref card-suits)
        (random-ref card-values)))

(define (make-deck)
  (for*/list ([suit card-suits]
              [value card-values])
    (card suit value)))

(define (random-deck)
  (wrap-external-randomness
   (shuffle (make-deck))))

(define gcrsbs ;; alias to simplify later code
  get-current-random-source-byte-string)

;; code continues in the next column ->
```

```
(define-values
  (cv bl)     ;; "Card Value" and "Byte List"
  (parameterize ([current-random-source
                  (make-random-source)])
    (values
     (random-card)
     (bytes->list (gcrsbs)))))
(print cv)    ;; -> (card '♠ 10)
(print bl)    ;; -> '(196 156 203  55
              ;;      232 115   4 248
              ;;       19 113  78 202)

(define inbs  ;; "INput Byte String"
  (list->bytes
   (list-update bl 7 add1)))

(define-values
  (ncv nbl)   ;; "New CV" and "New BL"
  (parameterize ([current-random-source
                  (make-random-source inbs)])
    (values
     (random-card)
     (bytes->list (gcrsbs)))))
(print ncv)   ;; -> (card '♣ 10)
(print nbl)   ;; -> '(196 156 203  55
              ;;      232 115   4 249
              ;;       19 113  78 202)
```

Figure 2. A Clotho example. Modifying the byte string obtained from a random source leads to a different `random-card` result.

fresh random source. From within this parameterized region, a new random card and the CRS's byte string are returned.

Let us imagine that we want to repeat our invocation of `random-card`, but we want to modify the suit that comes out. We can do this by modifying the byte string that we obtained after our initial invocation. As shown, we can use the modified byte string to create a new random source in the second `parameterize` call. The result of our second call is as we had hoped: a new random card is returned, and its suit is changed while its value is unchanged.

Selecting the byte to modify to effect this change is (relatively) straightforward in this example. Clotho uses the first four bytes of the byte string for internal initialization (§4). After that, each call to `random-ref` is associated with four bytes of the byte string. The call to `random-ref` that determines the suit is the first invocation of a randomness function within the parameterization of the CRS, so the bytes that affect its outcome are at indices 4–7 of the byte string. The code in Figure 2 creates a new input byte string by incrementing the seventh byte of the original byte string by 1.

Finally, we return to `random-deck`, which uses the `wrap-external-randomness` macro (§3.2). When this macro is invoked, it consumes four bytes from the CRS's byte string to seed and parameterize a Racket-wide `cprg`. This is necessary for interfacing with Racket functions, e.g., `shuffle`, that do not use Clotho but instead use Racket's standard randomness functions. By having Clotho instantiate a new `cprg` before calling those "external" randomness functions, Clotho ensures that the values returned by those functions are determined by the seed that Clotho supplies—and thus, the outcomes of those functions can be reproduced. In summary, within the body of `wrap-external-randomness`, the outcomes of all external randomness functions are determined by a single seed supplied by Clotho. When it is possible to do so, we recommend wrapping calls to external randomness functions individually, so that each call will correspond to a unique portion of Clotho's byte string.

4 Implementation

At its core, Clotho's functionality is provided through a struct type that is never directly exposed to the user. This struct is the type of the value managed by the `current-random-source` parameter mentioned in previous sections. The definition of the struct is quite simple:

```
(struct random-source-struct
  ([bts #:mutable]
   [idx #:mutable]
   [prg]
   [add #:mutable]))
```

```
#lang clotho
;; minesweeper.rkt

(require racket/list)
(provide (all-defined-out))

(define (play-game n mines)
  (define (play-moves covered exposed moves)
    (if (empty? covered)
        `(win ,(reverse moves))
        (let* ([move (random n)]
               [moves (cons move moves)])
          (cond
            [(memq move mines)
             `(lose ,(reverse moves))]
            [(memq move exposed)
             `(illegal ,(reverse moves))]
            [else
             (play-moves (remq move covered)
                         (cons move exposed)
                         moves)]))))
  (play-moves
   (remq* mines (range n)) empty empty))

(define gcrsbs
  get-current-random-source-byte-string)

;; code continues in the next column ->
```

```
(define (mutate-bytes input-bytes move-num)
  (define target-index (* 4 move-num))
  (define original-int
    (integer-bytes->integer
     input-bytes #f (system-big-endian?)
     target-index (+ 4 target-index)))
  (define mutated-int (+ 1 original-int))
  (integer->integer-bytes
   mutated-int 4 #f (system-big-endian?)
   input-bytes target-index))

(define (solve-game n mines [src-bts (bytes)]
                    [outcomes (list)])
  (define-values (outcome outcome-bytes)
    (parameterize
        ([current-random-source
          (make-random-source src-bts)])
      (values (play-game n mines)
              (gcrsbs))))
  (let ([outcomes (cons outcome outcomes)])
    (if (eq? (first outcome) 'win)
        (values (reverse outcomes)
                outcome-bytes)
        (solve-game n mines
                    (mutate-bytes
                     outcome-bytes
                     (length (second outcome)))
                    outcomes))))
```

Figure 3. A simple, one-dimensional Minesweeper game and a naive mutational solver.

The types of these fields are as follows:

- `bts`: `bytes?`
- `idx`: `integer?`
- `prg`: `pseudo-random-generator?`
- `add`: list of `integer?`

A new `random-source-struct` is created using a byte string. If no byte string is supplied, a new byte string consisting of four `0`-value bytes is created. This byte string is stored in `bts`. An index value, `idx`, is kept to point to the next four-byte segment of the byte string to use for random generation. The first four bytes of the byte string are used to seed a new, Racket-standard PRNG that is stored in the `prg` field—which is why the byte string must always have at least four bytes. A list of temporarily stored integers is kept in `add`, explained in more detail below.

The struct's design enables two forms of value generation: one using the byte string to "replay" values, and the other producing new values using the PRNG stored in `prg`. They work together seamlessly. When a value needs to be generated, the next four bytes from the `bts` byte string are taken and used; if there are no bytes remaining to be used, a new value is generated from the PRNG.

When a segment of bytes is used for generating a value, they are not returned directly as the result of a randomness function. Consider the following code segment:

```
(if (random-bool)
    (random-int)
    (random-bool))
```

Assume that we run this code and observe the values `#t` and `42` being produced. We obtain the byte string that caused these results, manipulate it, create a new CRS and rerun the code. If the result of the first call to `random-bool` in the new run is `#f`, Clotho must *reinterpret* the bytes that previously determined the result from `random-int`: now those bytes must determine the result of the second invocation of `random-bool`. To avoid implementing its own conversions from bytes to return values, Clotho interprets four-byte segments of the `bts` byte string as (integer) seed values for PRNGs, which it creates on demand. When a random value is needed, it consumes the next four bytes from the `bts` byte string to produce an integer, seeds a new CPRG with that integer, and invokes the appropriate randomness function, which calls `random` from the Racket standard library. The CPRG determines the value returned by `random`.

If the `bts` byte string is exhausted when a value is requested, the PRNG stored in `prg` is used to generate a new

value. The value generated by the PRNG is then used to seed a CPRG and call the appropriate randomness function, as just described. However, the seed for the CPRG is stored in the `add` list until a client calls `get-current-random-source-byte-string` to obtain the byte string. When this happens, the `add`'s values are reversed and appended to the `bts` byte string, and then that (now extended) byte string is returned. The add-list reduces the time complexity of intermediate random generations, because the `bytes-append` function can be very costly. By storing unserialized generated values in a list, we avoid incurring significant overhead.

The index, `idx`, is used to keep place while drawing values from the `bts` byte string. When the byte string is exhausted, the `idx` is set to `#f`.

The `random-source-struct`, along with its automatically provided functions, is kept private from even the rest of Clotho. An API is provided to the rest of the library that ensures certain conditions are maintained:

1. The `bts` byte string must contain at least four bytes, to be used for seeding the source's `prg` PRNG.
2. The `idx` index value corresponds to the head of the next four-byte sequence to be read from the `bts` byte string.
3. If all of the bytes have been read from the `bts` byte string, the `idx` index value is set to `#f`.
4. Whenever the call is made to extract the `bts` byte string from the struct, the `add` add-list must be reversed, converted to bytes, and appended to the byte string. (The `add` list must then be cleared.)
5. When a new value is requested and the `idx` index is `#f`, the `prg` PRNG is used to generate a new value.

These conditions ensure that random generation works as explained in §3.

5 Using Clotho

We have explained the design and implementation of Clotho and provided some small examples to show its use, but how can it be used to accomplish parametric generation? In this section, we give a brief example client in the form of a simplified Minesweeper game and show how to write a naive mutational fuzzer to force a win.

(Note: The non-figure code in this section is meant to be read additively. The output of `print` is shown in a comment to the right of the call. We convert byte strings to lists of byte-as-integers using `bytes->list` so the values are easier to read.)

5.1 Playing a Game of Minesweeper

Figure 3 contains the implementation of the game and its fuzzer. The `play-game` function contains the core game logic, which we will mostly gloss over here except to point out the use of the `random` function used to make a guess on the board.

An example call to `play-game` might look like this:

```
(require "minesweeper.rkt")
(define-values
  (r bs)  ;; "Result" and "Byte String"
  (parameterize
      ([current-random-source
        (make-random-source)])
    (values
     (play-game 5 '(2 3))
     (bytes->list (gcrsbs)))))

(print r)   ;; -> '(illegal (4 0 4))
(print bs)  ;; -> '(125  35 151  62
            ;;         0   0   0   0
            ;;         1 236 216 117
            ;;        33  15  40  66)
```

This plays a game of Minesweeper consisting of 5 cells, with mines hidden in cells 2 and 3. The result of the game and the resulting byte string from the generated `current-random-source` are returned. In this game, the player made an illegal move by attempting to expose the 4 cell twice. However, the first two moves (guessing 4 and then 0) were legal, so our player was on the right track! Let's use Clotho to modify this game so the player wins.

There are 16 bytes in the returned byte string `bs`. The first four bytes of `bs` are devoted to creating a PRNG (§4), and each call to `random` after that adds an additional four bytes to the byte string. This means that $(16 - 4) \div 4 = 3$ moves were made—which lines up correctly with the output list of moves we saw: `'(4 0 4)`. Since the last move in the sequence is the one that caused a failure, we want to try mutating bytes 12–15 in the byte string (remembering that the string is zero-indexed).

A simple mutation to try (which is used by the `mutate-bytes` function in Figure 3) is to increment those four bytes by 1. We can use the mutated byte string to build a new `current-random-source` and see what the outcome is:

```
(define inbs  ;; "INput Byte String"
  (mutate-bytes (list->bytes bs) 3))
(define-values
  (nr nbs)
  (parameterize
      ([current-random-source
        (make-random-source inbs)])
    (values
     (play-game 5 '(2 3))
     (bytes->list (gcrsbs)))))

(print nr)   ;; -> '(win (4 0 1))
(print nbs)  ;; -> '(125  35 151  62
             ;;         0   0   0   0
             ;;         1 236 216 117
             ;;        33  15  40  67)
```

Hooray! With that change, the player made the correct final guess and won the game.

```
(define (play-games game-count n mines)
  (for/list ([_ (range game-count)])
    (with-new-random-source
      (play-game n mines))))

(define (solve-games game-count n mines
                     [src-bts (bytes)]
                     [results (list)])
  (define-values (outcomes outcome-bytes)
    (parameterize
        ([current-random-source
          (make-random-source src-bts)])
      (values (play-games game-count n mines)
              (gcrsbs))))

  (define lost-game-index
    (index-where
     outcomes
     (lambda (outcome)
       (not (eq? 'win (first outcome))))))

  (let ([results (cons outcomes results)])
    (if lost-game-index
        (solve-games
         game-count n mines
         (mutate-bytes outcome-bytes
                       (add1 lost-game-index))
         results)
        (values (reverse results)
                outcome-bytes))))
```

Figure 4. An abstracted solver for multiple Minesweeper games played in a series.

The `solve-game` function (Figure 3) packages up this iterative refinement process. Here, the `current-random-source` is parameterized (using an empty byte string by default) and a game is played. If the game is won, the outcome is returned along with the byte string. Otherwise, the byte string is mutated and `solve-game` calls itself recursively with the new byte string and the list of outcomes accumulated so far.

5.2 Playing Multiple Minesweeper Games

Nobody wants to play just one game of Minesweeper. Let's expand our example to play multiple consecutive games and solve them all!

A simple approach is to write a `play-games` function that uses `for/list` to call `play-game` multiple times and accumulate the results. However, this raises a question: at what level do we parameterize the `current-random-source`? We could parameterize it outside of the `for/list`, using a single random source for all the games. Alternatively, we could do the parameterization *inside* the `for/list`, using a new random source for each game.

Let's try them both out and see what works best!

We begin by implementing the first method where all of the games are parameterized together:

```
(define (play-games game-count n mines)
  (parameterize
      ([current-random-source
        (make-random-source)])
    (values
      (for/list ([_ (in-range game-count)])
        (play-game n mines))
      (gcrsbs))))
```

The return values are a list of the game results and the byte string obtained after the last game is played.

While straightforward to implement, this can cause unpredictable effects during later mutation (such as that implemented by our `solve-game` function). Each game in this example can consume 1–3 random values, depending on how many moves the player makes. If mutating a move in one of the earlier games of the sequence causes that game to complete in a different number of moves than it did previously, subsequent games will play differently than they did before. Often, this is undesirable.

Let us instead try moving the parameterization of the `current-random-source` to a per-game level:

```
(define (play-games game-count n mines)
  (for/list ([_ (in-range game-count)])
    (parameterize
        ([current-random-source
          (make-random-source)])
      (cons
        (play-game n mines)
        (gcrsbs)))))
```

In this version of `play-games`, the return value is a list of pairs of game results with each game's corresponding byte string. This means each game's randomness is independent from the rest, but the cost is a more complicated output—one that does not naturally play well with tools that operate on a single byte string!

There is actually a third option: use *both*, but make the inner parameterization depend on a value in the parent parameterization. Or, to put it in code:

```
(define (play-games game-count n mines)
  (parameterize
      ([current-random-source
        (make-random-source)])
    (values
     (for/list ([_ (in-range game-count)])
       (parameterize
           ([current-random-source
             (make-random-source
              (random-seed-value))])
         (play-game n mines)))
     (gcrsbs))))
```

(Note that this code uses the `random-seed-value` function, which is a convenience function that generates a value

suitable for use as a random seed in either the CRS or the Racket CPRG.)

This `play-games` returns two values: a list of the results of the games, and a single top-level byte string. However, this byte string is different from the ones we've seen so far. Previously, the 4-byte segments of the byte strings (after the initial 4-byte segment reserved for internal use) corresponded to individual moves made during the game. But in this latest byte string, each 4-byte segment corresponds to an entire game. Essentially, we have abstracted the randomness: instead of fine-grained control over each randomness function, we now have coarse-grained control over the meta-randomness function `play-game`.

To make it easier to use *randomness abstraction*, Clotho provides the `with-new-random-seed` macro (§3.2). Called without any arguments, it functions identically to the inner `parameterize` function in the previous code segment, creating a new `current-random-source` using a seed value generated from the parent random source.

Figure 4 shows an example of implementing the code in this way. The `play-games` function plays multiple games of Minesweeper with identical inputs, each executed within its own `with-new-random-source` body. This function is called from within `solve-games`, which mutates bytes according to which games have resulted in a loss. The `solve-games` function has its own parameterization of the CRS (which functions as the top-level parameterization in this code), and `play-games` uses the macro to abstract the randomness within each call to `play-game`.

Below, we show the output of one execution of this function as seen in the Racket REPL, with the `illegal` symbol abbreviated to `ill` and columns vertically aligned for readability. The output can be rather long because multiple games are played many times, so we show only an excerpt of the output that illustrates the point.

```
> (define-values
    (results outcome-bytes)
    (solve-games 3 5 '(2 3) #f))

...  ;; <output lines removed>
((lose (4 1 3)) (ill  (0 1 0)) (lose (0 3)))
((lose (2))     (ill  (0 1 0)) (lose (0 3)))
((win  (1 0 4)) (ill  (0 1 0)) (lose (0 3)))
((win  (1 0 4)) (ill  (4 1 4)) (lose (0 3)))
((win  (1 0 4)) (ill  (4 4))   (lose (0 3)))
((win  (1 0 4)) (lose (2))     (lose (0 3)))
((win  (1 0 4)) (ill  (4 0 0)) (lose (0 3)))
((win  (1 0 4)) (lose (3))     (lose (0 3)))
((win  (1 0 4)) (ill  (4 4))   (lose (0 3)))
((win  (1 0 4)) (lose (4 2))   (lose (0 3)))
((win  (1 0 4)) (win  (4 0 1)) (lose (0 3)))
((win  (1 0 4)) (win  (4 0 1)) (lose (2)))
((win  (1 0 4)) (win  (4 0 1)) (lose (3)))
((win  (1 0 4)) (win  (4 0 1)) (ill  (4 4)))
...  ;; <output lines removed>
```

From this excerpt, one can observe how modifying a game earlier in the series does not affect the outcomes of subsequent games. If either of the second or third games had initially resulted in a `win`, they would not be affected by the modifications to the earlier games.

5.3 Xsmith

A major motivation in the creation of Clotho has been to aid in feedback-directed fuzzing. We have performed some preliminary work using Clotho for fuzzing by using it in conjunction with Xsmith [Hatch et al. 2020] and AFL [Zalewski 2020] to fuzz an implementation of the Lua programming language [Ierusalimschy et al. 1996]. In the future we intend to do further work on feedback-directed fuzzing using Clotho and Xsmith.

While Clotho was designed with guiding Xsmith program generators in mind, it could be used for other structured data generators, such as QuickCheck [Claessen and Hughes 2000] or Racket's `data/enumerate` package [New 2020].

6 Discussion

In building Clotho, we found some limitations and uncovered potential future directions of investigation.

6.1 Limitations

Because Racket's `pseudo-random-generator?` type is closed (i.e., cannot be implemented by an external source), interfacing with libraries that use Racket's built-in randomness functions can be awkward. We introduced the `wrap-external-randomness` macro to address this, but it is not an ideal solution: it requires the user to wrap every call to any function defined an external library that uses randomness. This can be somewhat tedious. Clotho provides its own `clotho/math/distributions` library that automatically finds all top-level bindings in Racket's `math/distributions` library and wraps them in our macro, allowing them to be used in code that uses Clotho. A significant (though perhaps not wholly detrimental) caveat of this is that a Clotho client is unable to exercise as fine-grained control over random generation as it would be able to if it implemented the functions for itself. When external functions are wrapped, the granularity of Clotho's control is at the level of calls to the external library, rather than at the level of calls to individual randomness functions (§3.5).

Another issue lies with the `with-new-random-source` function, and can be seen in §5.2. This macro provides an abstraction layer for randomness, but abstraction comes with a cost: one can no longer make fine-grained adjustments! In the example shown, the use of `with-new-random-source` precludes the solver from manipulating specific *moves*, instead requiring it to iterate on the seed for the PRNG that is used for entire *games*. This loss of granularity can cause a significant decrease in solver efficiency, because the mutations

may take the solver further away from the solution before they bring it closer. However, despite this limitation, we feel that there are situations that warrant the use of randomness abstraction, and so we leave `with-new-random-source` in the Clotho library.

6.2 Future Work

In working around the limitations of `with-new-random-source`, we have started to speculate about the use of a different datatype for capturing random generation. Currently, randomness is encoded into a single byte string. Instead, one can imagine perhaps using a list of bytes, which itself may contain sub-lists of bytes. Using this data structure, `with-new-random-source` could represent the abstracted `current-random-source`'s encoding as a sub-list in the parent context's `current-random-source`. This would allow for parameterizing abstracted regions with fine granularity. Clotho could also include a number of mutation functions specifically intended to aid in the modification of these sub-lists, which would further improve a developer's ability to manage randomness in an application.

Another improvement that could be made to Clotho's data representation would be to use data from the input byte string directly as return values from Clotho's randomness functions, only turning to a PRNG when really necessary. This would allow solvers to more directly manipulate the byte string, which may improve efficiency in finding interesting solutions. However, it is not straightforward, due to the need for input bytes to potentially be consumed by *any* randomness function (§4).

7 Related Work

We created Clotho because we want to use it in Xsmith [Hatch et al. 2020], our tool for creating random program generators. We want the ability to control all of the choices that Xsmith makes while generating a program—an idea that we borrowed from the Zest fuzzing system [Padhye et al. 2019]. Zest explores the state space of a system under test (SUT) by invoking a random test-case generator and using code-coverage feedback from the SUT to tune subsequent generation. Zest requires the test-case generator to be built such that its "random" choices are determined by a bit sequence that is input to the generator. Zest provides bit sequences to the generator in order to create test cases; when a test case triggers new code coverage, Zest mutates the corresponding bit sequence to create new inputs to the test-case generator. The key insight of Zest is that bit-level manipulations to the generator's input produce structural changes in the resulting test cases: "Zest converts a random-input generator into an equivalent deterministic *parametric generator*" [Padhye et al. 2019, p. 332]. Clotho enables similar functionality in Xsmith by providing an easy-to-use library for making random generation parametric in the same way as Zest, while also providing additional benefits.

Other systems have also manipulated the input of a test-case generator toward increasing the code coverage of a SUT. Crowbar [Dolan and Preston 2017], for example, is a testing library for OCaml that leverages AFL [Zalewski 2020] to (1) generate bit-level inputs that Crowbar turns into structured test cases and (2) measure coverage within the SUT. The DeepState unit-testing library for C and C++ supports coverage-directed fuzzing and can initialize the state of the SUT using bits from a provided input sequence [Goodman and Groce 2018].

Some prior work has manipulated the input to a test-case generator not toward increasing the code coverage of a SUT, but instead toward finding small test cases that trigger a behavior of interest in the SUT, e.g., a program crash or other bug. For example, Seq-Reduce [Regehr et al. 2012] relied on the Csmith program generator and aimed to "automate most or all of the work required to reduce bug-triggering test cases for C compilers." Seq-Reduce would first run Csmith to generate a program in Csmith's the normal way (using a PRNG), but recording the random decisions that were made during generation. Seq-Reduce would then repeatedly run Csmith, each time trying to discover a new decision sequence that would yield a new, smaller program that preserves the "interesting behavior" of the original. Regehr et al. [2012] concluded that Seq-Reduce was not very effective in general: changes early in the decision sequence would greatly impact the program generator, making it unlikely that the newly generated test case would preserve the behavior of interest. In Clotho, one can mitigate this issue by using the `with-new-random-source` macro to organize test-case generation into subparts, each of which draws from an independently seeded source of values.

Hypothesis [MacIver and Donaldson 2020] is a more recent test-case generator that implements "internal test-case reduction," i.e., the idea of manipulating the random decisions made during test-case generation, toward coaxing the generator into producing small test cases that preserve a property of interest. Starting from an input "choice sequence" that yields an interesting but large test case, Hypothesis applies heuristics that simplify the choice sequence and yield smaller test cases. In contrast to the conclusion reached by Regehr et al. for Seq-Reduce, MacIver and Donaldson found that internal reduction with Hypothesis often produced good results. For this reason, we speculate that Clotho may be useful for adding internal test-case reduction to Hypothesis-like tools written in Racket.

Wingate et al. [2011] present a method for providing parametric randomness in the implementations of probabilistic programming languages. Their method creates a naming scheme for each program trace that accesses a source of random data, i.e., a random function or variable. While running,

each access to random data triggers a database lookup, keyed by the name of the current program trace. Values from the database are returned as the results of random-data accesses, and new random values are added to the database as necessary for new trace names. The database may be preserved and altered for future executions to provide parametric deterministic randomness. Unlike Clotho, the technique described by Wingate et al. requires a whole-program transformation to track and name each program trace.

8 Conclusion

We have introduced Clotho, a Racket library that provides parametric randomness where you need it. In addition to its own functions, Clotho wraps and re-provides all of the randomness functionality of the `racket/base`, `racket/random`, and `math/distributions` modules in the Racket standard library—making parametric randomness accessible without conflict. Clotho provides mechanisms for supporting other randomness functions on a case-by-case basis. Because Clotho provides support for mixing parametric randomness with traditional PRNG-based randomness, it is flexible for a wide range of use cases.

We built Clotho as part of a random program generator in Racket, but its potential use is far more general than that. We hope that Clotho will prove useful to other developers seeking to manipulate randomly generated values within their programs.

Acknowledgments

We thank the anonymous Scheme '20 reviewers for their valuable comments on this work. Their feedback helped us to greatly improve this paper. This material is based upon work supported by the National Science Foundation under Grant Number 1527638.

References

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. 268–279. https://doi.org/10.1145/351240.351266

Stephen Dolan and Mindy Preston. 2017. Testing with Crowbar. In *Proceedings of the OCaml Users and Developers Workshop*. https://ocaml.org/meetings/ocaml/2017/extended-abstract__2017__stephen-dolan_mindy-preston__testing-with-crowbar.pdf

Peter Goodman and Alex Groce. 2018. DeepState: Symbolic Unit Testing for C and C++. In *Proceedings of BAR 2018: Workshop on Binary Analysis Research*. https://www.ndss-symposium.org/wp-content/uploads/2018/07/bar2018_9_Goodman_paper.pdf

William Gallard Hatch, Pierce Darragh, and Eric Eide. 2020. Xsmith software repository. https://gitlab.flux.utah.edu/xsmith/xsmith

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua—An Extensible Extension Language. *Software: Practice and Experience* 26, 6 (June 1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights From the Hypothesis Reducer. In *34th European Conference on Object-Oriented Programming (ECOOP '20)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

Max S. New. 2020. data/enumerate library documentation. https://docs.racket-lang.org/data/Enumerations.html

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. 329–340. https://doi.org/10.1145/3293882.3330576

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 335–346. https://doi.org/10.1145/2254064.2254104

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.), Vol. 15. 770–778. http://proceedings.mlr.press/v15/wingate11a.html

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 283–294. https://doi.org/10.1145/1993498.1993532

Michal Zalewski. 2020. AFL software repository. https://github.com/google/AFL