An Empirical Comparison Between Random Generation and Enumeration for Testing Redex Models

Max New
Northwestern University
max.new@eecs.northwestern.edu

Burke Fetscher
Northwestern University
burke.fetscher@eecs.northwestern.edu

Jay McCarthy
Brigham Young University
jay@cs.byu.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Abstract

This paper presents a benchmark suite of buggy Redex models designed to test bug-finding techniques. Our benchmark contains large and small models, easy and hard to find bugs, bugs that we invented based on our experience programming in Redex and bugs in models written by others that happened during development.

We evaluate three testing techniques: a generic, ad hoc random generator tuned for Redex programs, random selection from a uniform distribution of Redex programs, and an in-order enumeration of Redex programs.

Our results contradict commonly-held, yet unsubstantiated wisdom regarding the relative value of these three approaches. Specifically, selecting uniformly at random is the worst-performing choice, and enumeration and random selection are incomparable, with random being better with more than 10 minutes but in-order enumeration being better in interactive time-frames.

1. Myths about Randomness and Enumeration

Despite an early, convincing study on the value of random testing (Duran and Ntafos 1984), random testing is often unfairly treated as a straw-man comparison in testing papers. For example, Heidegger and Thiemann (2010) write:

> Spotting this defect requires the test case generator to guess a value for x such that x * 2 == x + 10 holds, but a random integer solves this equation with probability 2^{-32}.

When we run this example in Quickcheck (Classen and Hughes 2000),[1] giving it 1000 attempts to find a counterexample, it finds it about half of the time, taking on average about 400 attempts when it succeeds. Redex's random generator does a little bit better, finding it nearly every time, typically in about 150 attempts.

Redex finds this bug quickly because it uses a geometric distribution for selecting integers, meaning that the probability it picks a given integer is related to how close that integer is to zero. Since `10` isn't too far away, Redex is likely to choose it fairly frequently. We aren't sure what distribution QuickCheck uses for integers, but a comment in the source code suggests that it too prefers numbers closer to zero than numbers further away.

Not to single out a single paper, Godefroid et al. (2005) use the same example and Blanchette et al. (2011) discuss this buggy property (the last `xs` should be `ys`):

```
nth (append xs ys) (length xs+n) = nth xs n
```

saying that

> [r]andom testing typically fails to find the counterexample, even with hundreds of iterations, because randomly chosen values for `n` are almost always out of bounds.

This property is easier for both Quickcheck and Redex, taking, on average, 4 attempts for Quickcheck and 5 for Redex to find a counterexample.

While we certainly agree that random testing cannot find every bug and more powerful methods exist, our results show

[1] For the precise code we used, see the appendix.

, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

that random testing can be an effective tool for bug-finding, even using a generic random test case generator, i.e, one that is not tailored to the property being tested. Random testing is especially attractive because it is especially easily and cheaply applied, even to complex systems.

Naturally, not all papers treat all random testing as hopelessly naive. There are a number of papers that suggest that ad hoc random generation is a poor choice, but hold up a variation on random generation that selects test inputs from a uniform distribution over a complex data-structure as an obviously-good choice.

For example Grygiel and Lescanne (2013) present a technique for selecting from a uniform distribution of simply typed terms and argue that their results are practical because they help "debug compilers or other programs manipulating terms, e.g., type checkers or pretty printers." Also, Canou and Darrasse (2009) write than when using a random generator like Quickcheck, "it is very hard not to bias the form of generated values, and thus to unknowingly concentrate the domain of tested values to an arbitrary subset of values." They go on to give a technique for building a random generation technique that "is uniform: the generator for a given type and size gives the same probability to be produced to each possible value. In a testing context this property ensures that no subclass will be missed because the generator is biased." Their paper even goes so far as to use the word "sound" for random generators that are uniform. The implication being that Quickcheck-style random generation (or, presumably even worse, a fixed random generator like the one in Redex where the distribution of random terms cannot be directly adjusted by the user of the tool) is a less-effective bug finding technique. These papers give no empirical evidence for why this approach to random generation results in a set of terms that is more likely to find bugs.

To try to put our understanding on a firmer footing, we have designed and built a new enumeration strategy for Redex. Our enumerator is based on Tarau (2011)'s, but Redex's pattern language requires significant extensions to the basic strategy (as discussed in section 7). We use the enumerator to provide a uniform distribution of terms that we select from at random as well as simply iterating through the enumeration searching for counterexamples.

We have also built a benchmark suite of buggy Redex programs together with falsifiable properties that witness the bugs.

We evaluate the different generation strategies against the benchmark, showing that random testing is the best overall strategy, but that in-order enumeration finds more bugs in short time-frames. Selecting from a uniform distribution is worse than the other two strategies on our benchmark suite.

The remainder of this paper explains our new enumeration strategy and its application to Redex in section 2 and section 3; explains the methodology we used in our experiments, our benchmark suite, and our results in section 4, section 5, and section 6; and concludes after a discussion of related work in section 7 and section 8.

2. Enumeration Combinators

We represent enumerations as bijections between the natural numbers (or a prefix of them) and a set of terms. Concretely, our enumerations are triples of a function that encodes a term as a natural number, a function that decodes a natural number into a term, and a size.

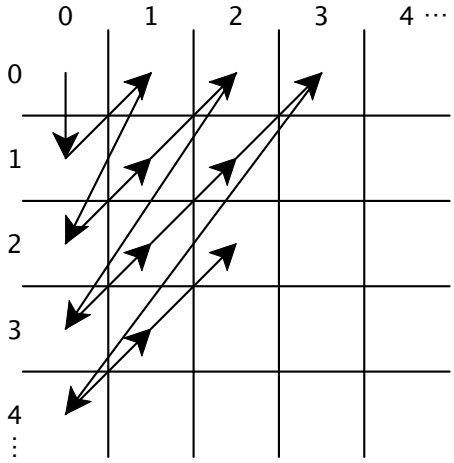
To get started, we build a few enumerators by directly supplying encode and decode functions, e.g., `nats/e` using the identity as both encoding and decoding functions.

In general, we want to avoid working directly with the functions because we must manually verify that the functions form a bijection. It is more convenient and less error-prone to instead use a combinator library to construct enumerations.

Our first combinator, `fin/e` builds a finite enumeration from its arguments, so `(fin/e 1 2 3 "a" "b" "c")` is an enumeration with the six given elements.

The next combinator is the pairing operator `cons/e`, that takes two enumerations and returns an enumeration of pairs of those values. If one of the enumerations is finite, the bijection loops through the finite enumeration, pairing each with an element from the other enumeration. If both are finite, we loop through the one with lesser cardinality. This corresponds to taking the quotient with remainder of the index with the lesser size.

Infinite enumerations require more care. If we imagine our sets as being laid out in an infinite table, this operator zig-zags through the table to enumerate all pairs:



which means that `(cons/e nats/e nats/e)`'s first `12` elements are

```
'(0 . 0)    '(0 . 1)    '(1 . 0)
'(0 . 2)    '(1 . 1)    '(2 . 0)
'(0 . 3)    '(1 . 2)    '(2 . 1)
'(3 . 0)    '(0 . 4)    '(1 . 3)
```

Generalizing pairs to n-ary tuples is not simply a matter of combining pairs together in an arbitrary way. In particular, here are two different ways to make 4-tuples of natural numbers:

```
(cons/e
 nats/e
 (cons/e        (cons/e
  nats/e         (cons/e nats/e nats/e)
  (cons/e        (cons/e nats/e nats/e))
   nats/e
   nats/e)))
```

After enumerating 4000 elements, the left-hand one has seen 87 in one component but only 3 in another, whereas the right-hand one has seen at most either 11 or 12 in all components. We refer to the right-hand version as being "fair" and always prefer fairness in our implementations, because it appears to correspond to the uniformity that is perceived as valuable with enumeration. In our experience, most of the time the obvious version of an enumerator is not fair and the details required to tweak it are non-intuitive. In this case, the key insight to achieve fairness is to map the leaves of the enumerated structure to the triangle numbers.

Another combinator is the disjoint union operator, `disj-sum/e`, that takes two or more enumerators and predicates to distinguish between their elements. It returns an enumeration of their union. The resulting enumeration alternates between the input enumerations, so that if given `n` infinite enumerations, the resulting enumeration will alternate through each of the enumerations every `n` numbers. For example, the following is the beginning of the disjoint sum of an enumeration of natural numbers and an enumeration of strings

```
0     ""    1     "a"   2     "b"   3
"c"   4     "d"   5     "e"   6     "f"
```

The `disj-sum/e` enumerator also has to be fair and to account for finite enumerations. So this enumeration:

```
(disj-sum/e (cons (fin/e 'a 'b 'c 'd) symbol?)
            (cons nats/e number?)
            (cons (fin/e "x" "y") string?))
```

has to cycle through the finite enumerations until they are exhausted before producing the rest of the natural numbers:

```
'a    0     "x"   'b    1     "y"   'c
2     'd    3     4     5     6     7
```

In general, this means that `disj-sum/e` must track the ranges of natural numbers when each finite enumeration is exhausted to compute which enumeration to use for a given index.

We provide a fixed-point combinator for recursive enumerations: `fix/e : (enum → enum) → enum`. For example, we can construct an enumerator for lists of numbers:

```
(fix/e (λ (lon/e)
         (disj-sum/e (cons (fin/e null)
                           null?)
                     (cons (cons/e nats/e lon/e)
                           cons?))))
```

and here are its first 12 elements:

```
'()          '(0)         '(0 0)
'(1)         '(0 0 0)     '(1 0)
'(2)         '(0 1)       '(1 0 0)
'(2 0)       '(3)         '(0 0 0 0)
```

A call like `(fix/e f)` enumerator calls `(f (fix/e f))` to build the enumerator, but it waits until the first time encoding or decoding happens before computing it. This means that a use of `fix/e` that is too eager, e.g.: `(fix/e (lambda (x) x))` will fail to terminate. Indeed, switching the order of the arguments to `disj-sum/e` in the above example also produces an enumeration that fails to terminate when decoding or encoding happens.

Our combinators rely on statically knowing the sizes of their arguments, but in a recursive enumeration this is begging the question. Since it is not possible to statically know whether a recursive enumeration uses its parameter, we leave it to the caller to determine the correct size, defaulting to infinite if not specified.

To build up more complex enumerations, it is useful to be able to adjust the elements of an existing enumeration. We use `map/e` which composes a bijection between any two sets with the bijection in an enumeration, so we can for example construct enumerations of natural numbers that start at some point beyond zero:

```
(define (nats-above/e i)
  (map/e (λ (x) (+ x i))
         (λ (x) (- x i))
         nats/e))
```

Also, we can exploit the bijection to define the `except/e` enumerator. It accepts an element and an enumeration, and returns one that doesn't have the given element. For example, the first 16 elements of `(except/e nats/e 13)` are

```
0     1     2     3     4     5     6     7
8     9     10    11    12    14    15    16
```

The decoder for `except/e` simply encodes the given element and then either subtracts one before passing the natural number along (if it is above the given exception) or doesn't (if it isn't). The decoder uses similar logic.

One important point about the combinators used so far: the decoding function is linear in the number of bits in the number it is given. This means, for example, that it takes only a few milliseconds to compute the $2^{100{,}000}$th element in the list of natural number enumeration given above.

Our next combinator `dep/e` doesn't always have this property. It accepts an enumerator and a function from ele-

ments to enumerators; it enumerates pairs of elements where the enumeration used for the second position in the pair depends on the actual values of the first position. For example, we can define an enumeration of ordered pairs (where the first position is smaller than the second) like this:

```
(dep/e nats/e (λ (i) (nats-above/e i)))
```

Here are the first 12 elements of the enumeration:

```
'(0 . 0)    '(0 . 1)    '(1 . 1)
'(0 . 2)    '(1 . 2)    '(2 . 2)
'(0 . 3)    '(1 . 3)    '(2 . 3)
'(3 . 3)    '(0 . 4)    '(1 . 4)
```

The dep/e combinator assumes that if any of the enumerations produced by the dependent function are infinite then all of them are. The implementation of dep/e has three different cases, depending on the cardinality of the enumerators it receives. If all of the enumerations are infinite, then it is just like cons/e, except using the dependent function to build the enumerator to select from for the second element of the pair. Similarly, if the second enumerations are all infinite but the first one is finite, then dep/e can use quotient and remainder to compute the indices to supply to the given enumerations when decoding. In both of these cases, dep/e preserves the good algorithmic properties of the previous combinators, requiring only linear time in the number of bits of the representation of the number for decoding.

The troublesome case is when the second enumerations are all finite. In that case, we think of the second component of the pair being drawn from a single enumeration that consists of all of the finite enumerations, one after the other. Unfortunately, in this case, the dep/e enumerator must compute all of the enumerators for the second component as soon as a single (sufficiently large) number is passed to decode, which can, in the worst case, take time proportional the magnitude of the number during decoding.

3. Enumerating Redex Patterns

Redex provides the construct define-language for specifying the grammar of a language. For example, this is the grammar for the simply-typed lambda calculus, augmented with a few numeric constants:

```
(define-language L
  (e ::=
     (e e)
     (λ (x : τ) e)
     x
     +
     integer)
  (τ ::= int (τ → τ))
  (x ::= variable-not-otherwise-mentioned))
```

Enumerating members of e can be done directly in terms of the combinators given in the previous section. Members of e are a disjoint sum of products of either literals (like λ and +) or recursive references. For example, this is the 100,000,000th term:

```
(λ (i
    :
    ((int → int)
     →
     (int
      →
      (int → (int → (int → int))))))
  (λ (a : ((int → int) → (int → int)))
    (λ (|| : int) 0)))
```

There are three patterns in Redex that require special care when enumerating. The first is repeated names. If the same meta-variable is used twice when defining a metafunction, reduction relation, or judgment form in Redex, then the same term must appear in both places. For example, a substitution function will have a case with a pattern like this:

```
(subst (λ (x : τ) e_1) x e_2)
```

to cover the situation where the substituted variable is the same as a parameter (in this case just returning the first argument, since there are no free occurrences of x). In contrast the two expressions e_1 and e_2 are independent since they have different subscripts. When enumerating patterns like this one, (subst (λ (a : int) a) a 1) is valid, but the term (subst (λ (a : int) a) b 1) is not.

To handle such patterns the enumerator makes a pass over the entire term and collects all of the variables. It then enumerates a pair where the first component is an environment mapping the found variables to terms and the second component is the rest of the term where the variables are replaced with constant enumerations that serve as placeholders. Once a term has been enumerated, Redex makes a pass over the term, replacing the placeholders with the appropriate entry in the environment.

In addition to a pattern that insists on duplicate terms, Redex also has a facility for insisting that two terms are different from each other. For example, if we write a subscript with _!_ in it, like this:

```
(subst (λ (x_!_1 : τ) e_1) x_!_1 e_2)
```

then the two xs must be different from each other.

Generating terms like these uses a very similar strategy to repeated names that must be the same, except that the environment maps x_!_1 to a sequence of expressions whose length matches the number of occurrences of x_!_1 and whose elements are all different. Then, during the final phase that replaces the placeholders with the actual terms, each placeholder gets a different element of the list.

Generating a list without duplicates requires the dep/e combinator and the except/e combinator. Here's how to generate lists of distinct naturals:

```
(define (lon-without eles)
  (fix/e (λ (lon/e)
           (disj-sum/e
            (cons (fin/e null) null?)
            (cons (dep/e
                   (except/e* nats/e eles)
                   (λ (new)
                     (lon-without
                      (cons new eles))))
                  cons?)))))
```

where `except/e*` simply calls `except/e` for each element of its input list. Here are the first 12 elements of the `(lon-without '())` enumeration:

```
'()          '(0)         '(0 1)
'(1)         '(0 1 2)     '(1 0)
'(2)         '(0 2)       '(1 0 2)
'(2 0)       '(3)         '(0 1 2 3)
```

This is the only place where dependent enumeration is used in the Redex enumeration library, and the patterns used are almost always infinite, so we have not encountered degenerate performance with dependent generation in practice.

The final pattern is a variation on Kleene star that requires that two distinct sub-sequences in a term have the same length.

To explain the need for this pattern, first consider the Redex pattern

```
((λ (x ...) e) v ...)
```

which matches application expressions where the function position has a lambda expression with some number of variables and the application itself has some number of arguments. That is, in Redex the appearance of `...` indicates that the term before may appear any number of times, possibly none. In this case, the term `((λ (x) x) 1)` would match, as would `((λ (x y) y) 1 2)` and so we might hope to use this as the pattern in a rewrite rule for function application. Unfortunately, the expression `((λ (x) x) 1 2 3 4)` also matches where the first ellipsis (the one referring to the `x`) has only a single element, but the second one (the one referring to `v`) has four elements.

In order to specify a rewrite rules that fires only when the arity of the procedure matches the number of actual arguments supplied, Redex allows the ellipsis itself to have a subscript. This means not that the entire sequences are the same, but merely that they have the same length. So, we would write:

```
((λ (x ..._1) e) v ..._1)
```

which allows the first two examples in the previous paragraph, but not the third.

To enumerate patterns like this, it is natural to think of using a dependent enumeration, where you first pick the length of the sequence and then separately enumerate sequences dependent on the list. Such a strategy is inefficient, however, because the dependent enumeration requires constructing enumerators during decoding.

Instead, if we separate the pattern into two parts, first one part that has the repeated elements, but now grouped together: `((x v) ...)` and then the remainder in a second part (just `(λ e)` in our example), then the enumerator can handle these two parts with the ordinary pairing operator and, once we have the term, we can rearrange it to match the original pattern.

This is the strategy that our enumerator implementation uses. Of course, ellipses can be nested, so the full implementation is more complex, but rearrangement is the key idea.

4. Methodology

Our case study compares three types of test-case generation using a set of buggy models. Each model and bug is equipped with a property that should hold for every term (but doesn't due to the bug) and three functions that generate terms, one for each of the different strategies. The three test-case generation strategies we evaluate (described below) are in-order enumeration, random selection from a uniform distribution, and ad hoc random generation.

For each bug and generator, we run a script that repeatedly asks for terms and checks to see if they falsify the property. As soon as it finds a counterexample to the property, it reports the amount of time it has been running. The script runs until the uncertainty in the average becomes acceptably small or until 24 hours elapses, whichever comes first.

We ran our script on one of two identical 64 core AMD machines with Opteron 6274s running at 2,200 MHz with a 2 MB L2 cache. Each machine has 64 gigabytes of memory. Our script runs each model/bug combination sequentially, although we ran multiple combinations at once in parallel.

We used the unreleased version 6.0.0.5 of Racket (of which Redex is a part); more precisely the version at git commit `a7d6809243`,[2] except for the in-order generation of the **rvm** model (discussed in section 5.7), because we recently discovered a bug in that model's in-order generator that could affect its running time. They were run from a slightly different version of Racket, namely commit `da158e6d95`. The only other difference between the two versions are some improvements to Typed Racket that are unlikely to affect our results.

For the in-order enumeration, we simply indexed into the decode functions (as described in section 2), starting at zero and incrementing by one each time.

For the random selection from the uniform distribution, we need a mechanism to pick a natural number. To do this, we first pick an exponent i in base 2 from the geometric distribution and then pick uniformly at random an integer that is between 2^{i-1} and 2^i. We repeat this process three

[2] `https://github.com/plt/racket/commit/a7d6809243`

times for and then take the largest – this helps make sure that the numbers are not always small.

We chose these numbers because there is not a fixed mean of the distribution of numbers. That is, if you take the mean of some number of samples and then add more samples and take the mean again, the mean of the new numbers is different from the mean of the old. We believe this is a good property to have when indexing into our uniform distribution so as to avoid biasing the choice of examples towards some small size.

The precise algorithm we used is implemented in these functions:

```
;; pick-an-index : ([0,1] -> Nat) ∩ (-> Nat)
(define (pick-an-index [prob-of-zero 0.01])
  (max (random-natural/no-mean prob-of-zero)
       (random-natural/no-mean prob-of-zero)
       (random-natural/no-mean prob-of-zero)))

;; random-natural/no-mean : [0,1] -> Nat
(define (random-natural/no-mean prob-zero)
  (define x (sample (geometric-dist prob-zero)))
  (define m1 (expt 2 (exact-floor x)))
  (define m0 (quotient m1 2))
  (random-integer m0 m1))
```

The random-selection results are quite sensitive to the probability of picking the zero exponent (the `prob-of-zero` argument). We empirically chose benchmark-specific numbers in an attempt to maximize the success of the random uniform distribution method.

For the ad hoc random generation, we use Redex's existing random generator (Klein and Findler 2009). It has been tuned based on our experience programming in Redex, but not recently. The most recent change to it was a bug fix in April of 2011 and the most recent change that affected the generation of random terms was in January of 2011, both well before we started the current study.

This generator, which is based on the method of recursively unfolding non-terminals, is parameterized over the depth at which it attempts to stop unfolding non-terminals. We chose a value of 5 for this depth since that seemed to be the most successful. This produces terms of a similar size to those of the uniform random generator, although the distribution is different.

5. Benchmark Overview and Bug-Specific Results

The programs in our benchmark come from two sources: synthetic examples based on our experience with Redex over the years and from models that we and others have developed and bugs that were encountered during the development process.

The benchmark has six different Redex models, each of which provides a grammar of terms for the model and a soundness property that is universally quantified over those terms. Most of the models are of programming languages and most of the soundness properties are type-soundness, but we also include red-black trees with the property that insertion preserves the red-black invariant, as well as one richer property for one of the programming language models (discussed in section 5.3).

For each model, we have manually introduced bugs into a number of copies of the model, such that each copy is identical to the correct one, except for a single bug. The bugs always manifest as a term that falsifies the soundness property.

The table in figure 1 gives an overview of the benchmark suite, showing some numbers for each model and bug. Each model has its name and the number of lines of code in the correct version (the buggy versions are always within a few lines of the originals). The line number counts include the model, the specification of the property, and a little bit of model-specific code to call into the different generators. The p-value column shows the argument to `pick-a-number` that we used to get numbers to index into the uniform distribution for the model. The mean and standard deviation are of the size of 10,000 random terms from the uniform distribution, picked with the given p value.

Each bug has a number and, with the exception of the rvm model, the numbers count from 1 up to the number of bugs. The rvm model bugs are all from Klein et al. (2013)'s work and we follow their numbering scheme (see section 5.7 for more information about how we chose the bugs from that paper).

The **S/M/D/U** column shows a classification of each bug as:

- **S** (Shallow) Errors in the encoding of the system into Redex, due to typos or a misunderstanding of subtleties of Redex.
- **M** (Medium) Errors in the algorithm behind the system, such as using too simple of a data-structure that doesn't allow some important distinction, or misunderstanding that some rule should have a side-condition that limits its applicability.
- **D** (Deep) Errors in the developer's understanding of the system, such as when a type system really isn't sound and the author doesn't realize it.
- **U** (Unnatural) Errors that are unlikely to have come up in real Redex programs but are included for our own curiosity. There are only two bugs in this category.

The size column shows the size of the term representing the smallest counterexample we know for each bug, where we measure size as the number of pairs of parentheses and atoms in the sexpression representation of the term.

Each subsection of this section introduces one of the models in the benchmark, along with the errors we introduced into each model. The subsections also discuss the spe-

cific bugs, pointing out some of the more interesting bug-specific results we found. Section 6 discusses the our results from a more global perspective.

The bug-specific results are summarized in figure 2. On the y-axis is time in seconds, using a log scale. The x-axis has all of the bugs, sorted by the average time required to find the bug for all three generators. The error bars represent 95% confidence intervals in the averages. (The in-order enumeration method is deterministic and thus has no uncertainty.) The blank columns on the right represent the bugs that no method was able to find in less than 24 hours, of which there are 13. The averages span nearly seven orders of magnitude from less than a tenth of a second to several hours and thus represent a wide range of bugs in terms of how difficult it is to generate counterexamples.

5.1 stlc

A simply-typed lambda calculus with base types of numbers and lists of numbers, including the constants `+`, which operates on numbers, and `cons`, `head`, `tail`, and `nil` (the empty list), all of which operate only on lists of numbers. The property checked is type soundness: the combination of preservation (if a term has a type and takes a step, then the resulting term has the same type) and progress (that well-typed non-values always take a reduction step).

We introduced nine different bugs into this system. The first confuses the range and domain types of the function in the application rule, and has the small counterexample: `(hd 0)`. We consider this to be a shallow bug, since it is essentially a typo and it is hard to imagine anyone with any knowledge of type systems making this conceptual mistake. Bug 2 neglects to specify that a fully applied `cons` is a value, thus the list `((cons 0) nil)` violates the progress property. We consider this be be a medium bug, as it is not a typo, but an oversight in the design of a system that is otherwise correct in its approach.

We consider the next three bugs to be shallow. Bug 3 reverses the range and the domain of function types in the type judgment for applications. This was one of the easiest bug for all of our approaches to find. Bug 4 assigns `cons` a result type of `int`. The fifth bug returns the head of a list when `tl` is applied. Bug 6 only applies the `hd` constant to a partially constructed list (i.e., the term `(cons 0)` instead of `((cons 0) nil)`). Only the grammar based random generation exposed bugs 5 and 6 and none of our approaches exposed bug 4.

The seventh bug, also classified as medium, omits a production from the definition of evaluation contexts and thus doesn't reduce the right-hand-side of function applications.

Bug 8 always returns the type `int` when looking up a variable's type in the context. This bug (and the identical one in the next system) are the only bugs we classify as unnatural. We included it because it requires a program to have a variable with a type that is more complex that just `int` and to actually use that variable somehow.

Bug 9 is simple; the variable lookup function has an error where it doesn't actually compare its input to variable in the environment, so it effectively means that each variable has the type of the nearest enclosing lambda expression.

5.2 poly-stlc

This is a polymorphic version of **stlc**, with a single numeric base type, polymorphic lists, and polymorphic versions of the list constants. No changes were made to the model except those necessary to make the list operations polymorphic. There is no type inference is the model, so all polymorphic terms are required to be instantiated with the correct types in order for the function to type check. Of course, this makes it much more difficult to automatically generate well-typed terms, and thus counterexamples. As with **stlc**, the property checked is type soundness.

All of the bugs in this system are identical to those in **stlc**, aside from any changes that had to be made to translate them to this model. Comparing the results for these two systems in figure 2, we can see indeed some bugs have become easier to find (such as bug 7) and some have become more difficult (such as bug 2).

This model is also a subset of the language specified in Pałka et al. (2011), who used a specialized and optimized QuickCheck generator for a similar type system to find bugs in GHC. We adapted this system (and its restriction in **stlc**) because it has already been used successfully with random testing, which makes it a reasonable target for an automated testing benchmark.

5.3 stlc-sub

The same language and type system as **stlc**, except that in this case all of the errors are in the substitution function.

Our own experience has been that it is easy to make subtle errors when writing substitution functions, so we added this set of tests specifically to target them with the benchmark. There are two soundness checks for this system. Bugs 1-5 are checked in the following way: given a candidate counterexample, if it type checks, then all βv-redexes in the term are reduced (but not any new ones that might appear) using the substitution function to get a second term. Then, these two terms are checked to see if they both still type check and have the same type and that the result of passing both to the evaluator is the same.

Bugs 4-9 are checked using type soundness for this system as specified in the discussion of the **stlc** model. We included two predicates for this system because we believe the first to be a good test for a substitution function but not something that a typical Redex user would write, while the second is something one would see in most Redex models but is less effective at catching bugs in the substitution function.

The first substitution bug we introduced simply omits the case that replaces the correct variable with the with the term to be substituted. We considered this to be a shallow error, and indeed all approaches were able to uncover it, although

Model	LoC	P-Value	Mean ± Stddev	Bug #	S/M/D/U	Size	Description of Bug
stlc	238	0.035	13.9 ± 20.6	1	S	3	app rule the range of the function is matched to the argument
				2	M	5	the ((cons v) v) value has been omitted
				3	S	8	the order of the types in the function position of application has been swapped
				4	S	9	the type of cons is incorrect
				5	S	7	the tail reduction returns the wrong value
				6	M	7	hd reduction acts on partially applied cons
				7	M	14	evaluation isn't allowed on the rhs of applications
				8	U	12	lookup always returns int
				9	S	15	variables aren't required to match in lookup
poly-stlc	254	0.065	17.0 ± 23.1	1	S	10	app rule the range of the function is matched to the argument
				2	M	11	the (([cons @ τ] v) v) value has been omitted
				3	S	14	the order of the types in the function position of application has been swapped
				4	S	15	the type of cons is incorrect
				5	S	16	the tail reduction returns the wrong value
				6	M	16	hd reduction acts on partially applied cons
				7	M	9	evaluation isn't allowed on the rhs of applications
				8	U	15	lookup always returns int
				9	S	18	variables aren't required to match in lookup
stlc-sub	273	0.035	14.0 ± 20.9	1	S	8	forgot the variable case
				2	S	13	wrong order of arguments to replace call
				3	S	10	swaps function and argument position in application
				4	D	22	variable not fresh enough
				5	SM	17	replace all variables
				6	S	8	forgot the variable case
				7	S	13	wrong order of arguments to replace call
				8	S	10	swaps function and argument position in application
				9	SM	15	replace all variables
list-machine	397	0.5	13.2 ± 4.5	1	S	12	confuses the lhs value for the rhs value in cons type rule
				2	M	12	var-set may skip a var with matching id (in reduction)
				3	S	19	cons doesn't actually update the store
rbtrees	263	0.25	15.5 ± 13.9	1	M	13	ins does no rebalancing
				2	M	15	the first case is removed from balance
				3	S	51	doesn't increment black depth in non-empty case
delim-cont	735	0.125	11.6 ± 9.7	1	M	32	guarded mark reduction doesn't wrap results with a list/c
				2	M	22	list/c contracts aren't applied properly in the cons case
				3	S	38	the function argument to call/comp has the wrong type
rvm	816	0.03	15.7 ± 20.8	2	M	24	stack offset / pointer confusion
				3	D	33	application slots not initialized properly
				4	M	17	mishandling branches when then branch needs more stack than else branch; bug in the boxenv case not checking a stack bound
				5	M	23	mishandling branches when then branch needs more stack than else branch; bug in the let-rec case not checking a stack bound
				6	M	15	forgot to implement the case-lam branch in verifier
				14	M	27	certain updates to initialized slots could break optimizer assumptions
				15	S	21	neglected to restict case-lam to accept only 'val' arguments

Figure 1: Benchmark Overview

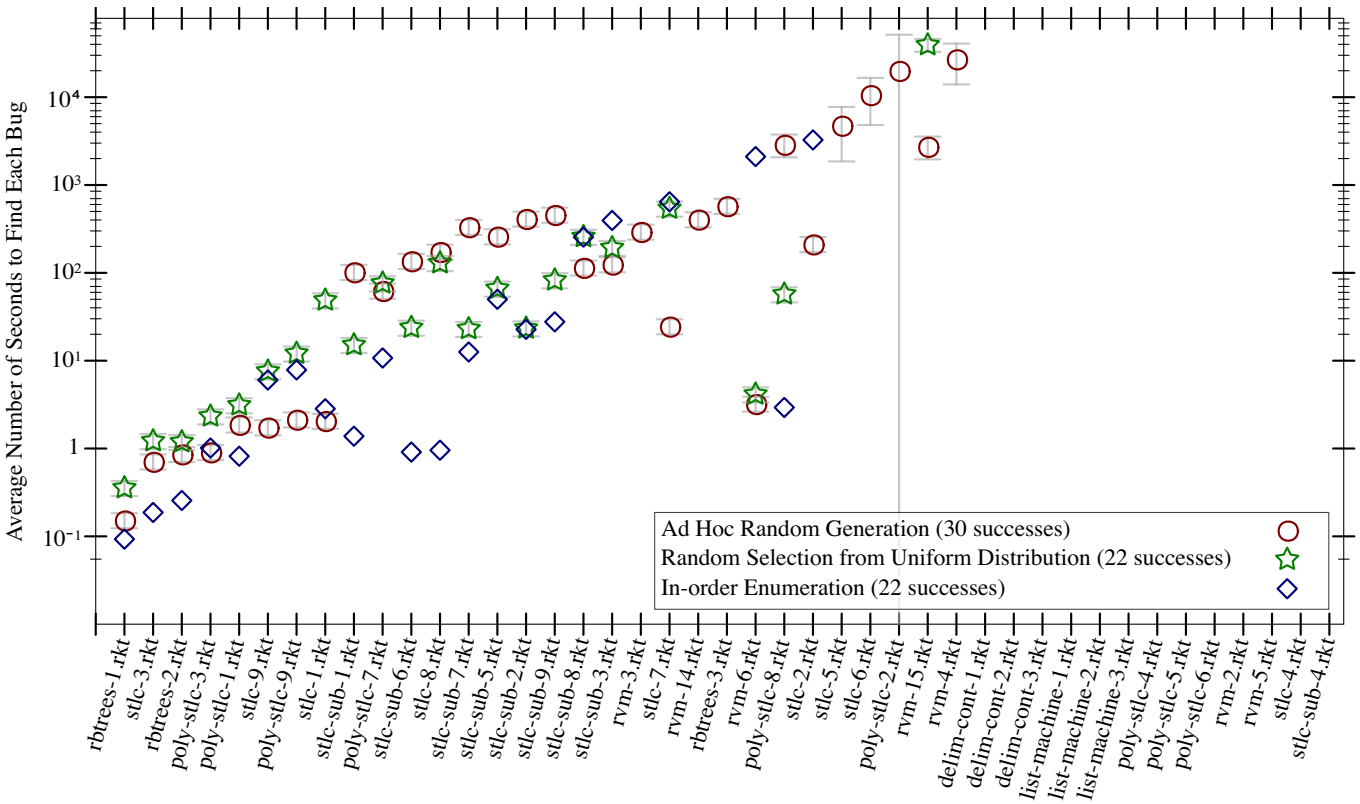


Figure 2: Performance Results by Individual Bug, following the naming scheme: «model name»-«bug number».rkt

the time it took to do so ranged from 1 second to around 2 minutes.

Bug 2 permutes the order of arguments when making a recursive call. This is also categorized as a shallow bug, although it is a common one, at least based on our experience writing substitutions in Redex.

Bug 3 swaps the function and argument positions of an application recurring, again essentially a typo and a shallow error, although one of the more difficult to find bugs in this model.

The fourth substitution bug neglects to make the renamed bound variable fresh enough when when recurring past a lambda. Specifically, it ensures that the new variable is not one that appears in the body of the function, but it fails to make sure that the variable is different from the bound variable or the substituted variable. We categorized this error as deep because it corresponds to a misunderstanding of how to generate fresh variables, a central concern of the substitution function.

Bug 5 carries out the substitution for all variables in the term, not just the given variable. We categorized it as SM, since it is essentially a missing side condition, although a fairly egregious one.

Bugs 6-9 are duplicates of bugs 1-3 and bug 5, except that they are tested with type soundness instead. (It is impossible to detect bug 4 with this property.) Surprisingly, there is not as a clear a difference as one might expect in the effectiveness of the two properties in our results, although type soundness is slightly less effective overall. (See the "sltc-sub" models in figure 2.)

5.4 list-machine

An implementation of Appel et al. (2012)'s list-machine benchmark. This is a reduction semantics (as a pointer machine operating over an instruction pointer and a store) and a type system for a seven-instruction first-order assembly language that manipulates `cons` and `nil` values. The property checked is type soundness as specified in Appel et al. (2012), namely that well-typed programs always step or halt. Three mutations are included. This model was one of the standard examples distributed with Redex that we adapted for the benchmark.

The first list-machine bug incorrectly uses the head position of a cons pair where it should use the tail position in the

cons typing rule. This bug amounts to a typo and is classified as simple.

The second bug is a missing side-condition in the rule that updates the store that has the effect of updating the first position in the store instead of the proper position in the store for all of the store update operations. We classify this as a medium bug.

The final list-machine bug is a missing subscript in one rule that has the effect that the list cons operator does not store its result. We classify this as a simple bug.

None of the list-machine bugs were found by any of our generation strategies, simply because the structure of the list-machine's typing rules definition. These rules require the generators to generate both a term and a type that match (which is difficult for the random generators) and the smallest expressions that type check, while not huge, are still larger than in the other models (which is difficult for the enumerator).

5.5 rbtrees

A model that implements the red-black tree insertion function and checks that insertion preserves the red-black tree invariant (and that the red-black tree is a binary search tree).

The first bug simply removes the re-balancing operation from insert. We classified this bug as medium since it seems like the kind of mistake that a developer might make in staging the implementation. That is, the re-balancing operation is separate and so might be put off initially, but then forgotten.

The second bug misses one situation in the re-balancing operation, namely when a black node has two red nodes under it, with the second red node to the right of the first. This is a medium bug.

The third bug is in the function that counts the black depth in the red-black tree predicate. It forgets to increment the count in one situation. This is a simple bug.

The first two bugs are among the easiest to find with all three generators finding the bug in at most a second or so. The third bug is one of the ones where the ad hoc random generator finds it in good time (about 10 minutes) but the other two generators cannot find it even with 24 hours of time.

5.6 delim-cont

Takikawa et al. (2013)'s model of a contract and type system for delimited control. The language is Plotkin's PCF extended with operators for delimited continuations, continuation marks, and contracts for those operations. The property checked is type soundness. We added three bugs to this model.

The first was a bug we found by mining the model's git repository's history. This bug fails to put a list contract around the result of extracting the marks from a continuation, which has the effect of checking the contract that is supposed to be on the elements of a list against the list itself instead. We classify this as a medium bug.

The second bug was in the rule for handling list contracts. When checking a contract against a cons pair, the rule didn't specify that it should apply only when the contract is actually a list contract, meaning that the cons rule would be used even on non-list contacts, leading to strange contract checking. We consider this a medium bug because the bug manifests itself as a missing `list/c` in the rule.

The last bug in this model makes a mistake in the typing rule for the continuation operator. The mistake is to leave off one-level of arrows, something that is easy to do with so many nested arrow types, as continuations tend to have. We classify this as a simple error.

None of our generators can find these three errors in 24 hours.

5.7 rvm

A existing model and test framework for the Racket virtual machine and bytecode verifier (Klein et al. 2013). The bugs were discovered during the development of the model and reported in section 7 of that paper. Unlike the rest of the models, we do not number the bugs for this model sequentially but instead use the numbers from Klein et al's work.

We included only some of the bugs, excluding bugs for two reasons:

- The paper tests two properties: an internal soundness property that relates the verifier to the virtual machine model, and an external property that relates the verifier model to the verifier implementation. We did no include any that require the latter properties because it requires building a complete, buggy version of the Racket runtime system to include in the benchmark.
- We included all of the internal properties except those numbered 1 and 7 for practical reasons. The first is the only bug in the machine model, as opposed to the just the verifier, which would have required us to include the entire VM model in the benchmark. The second would have required modifying the abstract representation of the stack in the verifier model in contorted way to mimic a more C-like implementation of a global, imperative stack. This bug was originally in the C implementation of the verifier (not the Redex model) and to replicate it in the Redex-based verifier model would require us to program in a low-level imperative way in the Redex model, something not easily done.

These bugs are described in detail in Klein et al's paper. These bugs are generally difficult for our generation strategies to uncover. Three of the bugs were found only by the ad hoc random generator, two were also found by the random selection from the uniform distribution and the remaining two were not found by any of our generators in 24 hours. We suspect that it is the relatively large grammar of the language that makes it difficult for the in order enumerator to find these bugs. There are 24 productions for expressions

which means that there are many terms even with relatively small sizes.

This model is also unique in our benchmark suite because it includes a function that makes terms more likely to be useful test cases. In more detail, the machine model does not have variables, but instead is stack-based; bytecode expressions also contain internal pointers that must be valid. Generating a random (or in-order) term is relatively unlikely to produce one that satisfies these constraints. For example, of the of the first 10,000 terms produced by the in-order enumeration only 1625 satisfy the constraints. The ad hoc random generator generators produces about 900 good terms in 10,000 attempts and the uniform random generator produces about 600 in 10,000 attempts.

To make terms more likely to be good test cases, this model includes a function that looks for out-of-bounds stack offsets and bogus internal pointers and replaces them with random good values. This function is applied to each of the generated terms before using them to test the model.

6. Global Trends in Our Results

Our primary concern with this study was to determine the relative merits of the three generation strategies. Figure 3 shows our data with this aim in mind. Along its x-axis is time in seconds, again with a log scale, and along the y-axis is the total number of bugs found for each point in time. There are three lines on the plot showing how the total number of bugs found changes as time passes.

The blue dashed line shows the performance of in-order enumeration and it is clearly the winner in the left-hand side of the graph. The solid red line shows the performance of the ad hoc random generator and it is clearly the winner on the right-hand side of the graph, i.e. the longer time-frames.

There are two crossover points marked on the graph with black dots. After 2 minutes, with 17 of the bugs found, the enumerator starts to lose and random selection from the uniform distribution starts to win until 7 minutes pass, at which time the ad hoc generator starts to win and it never gives up the lead.

Overall, we take this to mean that on interactive time-frames, the in-order enumeration is the best method and on longer time-frames ad hoc generation is the best. While selection from the uniform distribution does win briefly, it does not hold its lead for long and there are no bugs that it finds that ad hoc generation does not also find.

Although there are 43 bugs in the benchmark, no strategy was able to find more than 30 of them in a 24 hour period.

Figure 1 also shows that, for the most part, bugs that were easy (could be found in less than a few seconds) for either the ad hoc generator or the generator that selected at random from the uniform distribution were easy for all three generators. The in-order enumeration, however, was able to find several bugs (such as bug #8 in poly-stlc) in much shorter times than the other approaches.

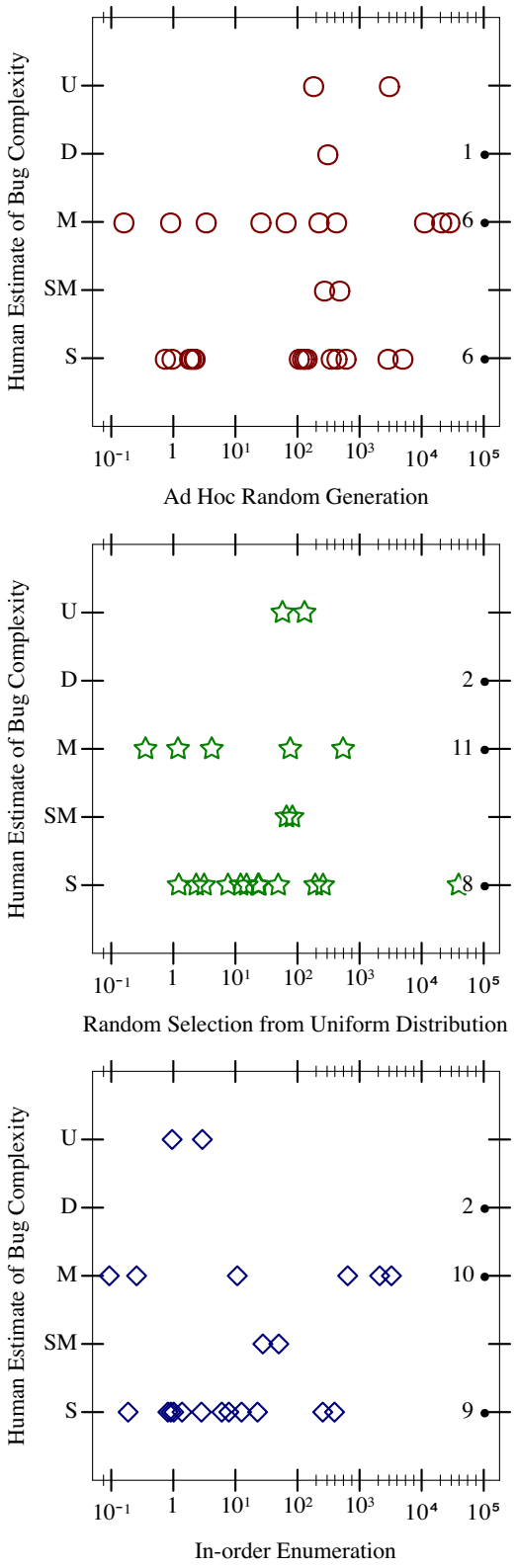


Figure 4: Scatter Plot Between Human Estimate of Complexity and Random Generation Success Time in Seconds (Right-most column shows bugs whose counterexamples were never found)

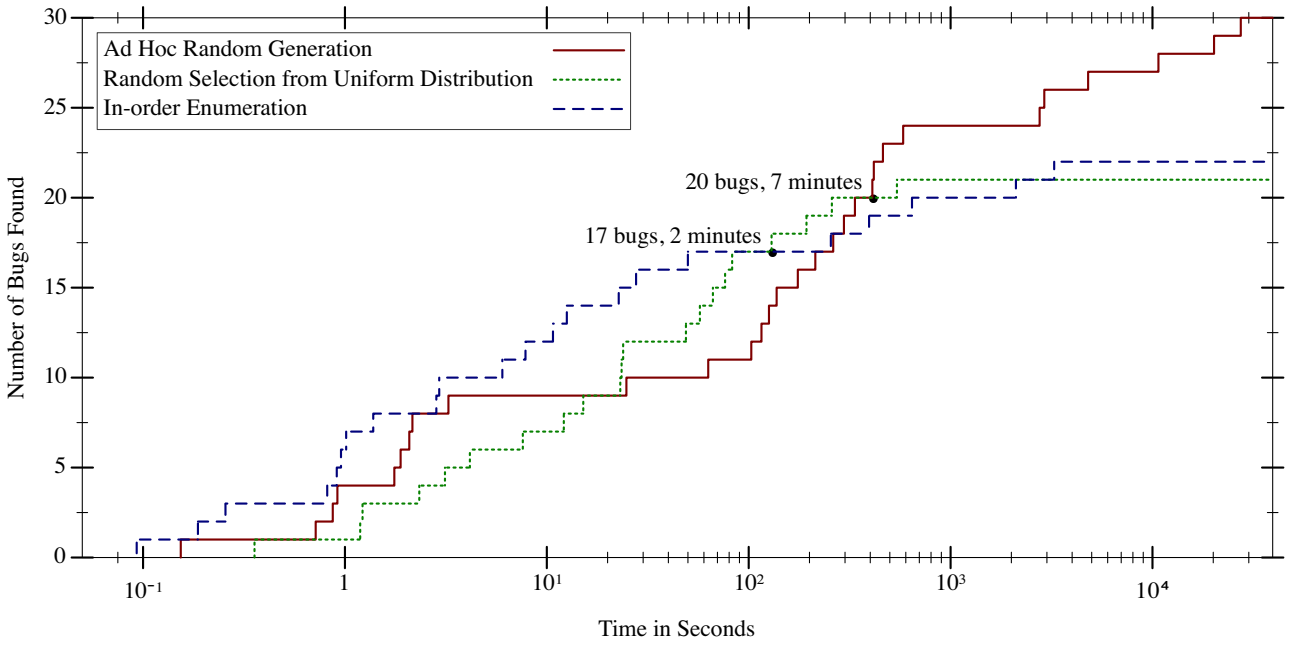


Figure 3: Random testing performance of in-order enumeration, random indexing into an enumeration, and recursive generation from a grammar on a benchmark of Redex models.

We also compared the human notion of complexity of the bugs to how well the three random generators do, using the scatter plots in figure 4. The x-axis shows the amount of time that a given generator took to find the bug and y-axis has the human-ranked complexity of the bug. For bugs that were never found, a single black dot (along with the count of bugs) is placed in a column on the right-hand side of the graph. These plots show that there is no correlation between how humans view the importance of the bugs and how effective our generators are at finding it.

7. Related Work

The related work divides into two categories: papers about enumeration and papers with studies about random testing.

7.1 Enumeration Methods

Tarau (2011)'s work on bijective encoding schemes for Prolog terms is most similar to ours. However, we differ in two main ways. First, our enumerators deal with enumeration of finite sets wherever they appear in the larger structure. This is complicated because it forces our system to deal with mismatches between the cardinalities of two sides of a pair: for instance, the naive way to implement pairing is to give odd bits to the left element and even bits to the right element, but this cannot work if one side of the pair, say the left, can be exhausted as there will be arbitrarily numbers of bits that do not enumerate more elements on the left. Second, we have a dependent pairing enumerator that allows the right element of a pair to depend on the actual value produced on the left. Like finite sets, this is challenging because of the way each pairing of an element on the left with a set on the right consumes an unpredictable number of positions in the enumeration.

Duregård et al. (2012)'s Feat is a system for enumeration that distinguishes itself from "list" perspectives on enumeration by focusing on the "function" perspective. We use the "function" perspective as well. While our approach is closer to Tarau's, we share support for finite sets with Feat, but are distinct from Feat in our support for dependent pairing, which can be used to implement what Feat refers to as "invariants", although not efficiently.

Kennedy and Vytiniotis (2010) take a different approach to something like enumeration, viewing the bits of an encoding as a sequence of messages responding to an interactive question-and-answer game. The crucial insight in this perspective is that the questioner may have its own memory to help it avoid asking "silly" questions, such as whether a term is a number when the type system rules out a number at that position in the larger term structure. This aspect of their system, which is crucial in what they call dependent composition, is the same as our dependent pairing.

However, details of their system show that it is not an enumeration system. In particularly, the strongest proof they have is that if a game is total and proper, then "every bitstring encodes some value or is the prefix of such a bitstring". This means, that even for total, proper games there are some

bitstrings that do not encode a value. As such, it cannot be used to enumerate all elements of the set being encoded. Furthermore, they show that many useful games are non-proper and must be converted into proper games by filtering a non-proper game: enumerating all possible elements and removing those that do not match a predicate. This rejection-based approach to generating well-typed terms, for example, is costly; as discussed in detail, for instance, in the paper on Feat. Our system has a similar problem with dependent pairs where to decode an element from the n+1-st set, you must have a count for each of the prior n sets. However, when these counts are predictable, they need not be constructed; and when they have been previously computed, they can be reused (and our implementation caches them).

7.2 Testing Studies

We are aware of only one other study that specifically compares random testing and enumeration, namely in Runciman et al. (2008)'s original paper on SmallCheck. SmallCheck is an enumeration-based testing library for Haskell and the paper contains a comparison with QuickCheck, Haskell random testing library.

Their study is not as in-depth as ours; the paper does not say, for example, how many errors were found by each of the techniques or in how much time, only that there were two errors that were found by enumeration that were not found by random testing. The paper, however, does conclude that "SmallCheck, Lazy SmallCheck and QuickCheck are complementary approaches to property-based testing in Haskell," a stance that our study supports (but for Redex).

Bulwahn (2012) compares a single tool that supports both random testing and enumeration against a tool that reduces conjectures to boolean satisfiability and then uses a solver. The study concludes that the two techniques complement each other.

Neither of the studies compare selecting randomly from a uniform distribution like ours does.

Pałka (2012)'s work is similar in spirit to Redex, as it focuses on testing programming languages. Pałka builds a specialized random generator for well-typed terms that found several bugs in GHC, the premier Haskell compiler. Similarly, Yang et al. (2011)'s work also presents a test-case generator tailored to testing programming languages with complex well-formedness constraints, but this time C. Miller et al. (1990) designed a random generator for streams of characters with various properties (e.g. including nulls or not, to include newline characters at specific points) and used it to find bugs in unix utilities.

All three of these papers provide empirical evidence that random generation techniques that do not sample from a uniform distribution can be highly successful at finding bugs.

8. Conclusion

Our study shows that the relationship between ad hoc random generation and in-order enumeration is subtle, and that selecting randomly from a uniform distribution is not as effective for testing as the literature claims.

Based on these findings we have modified Redex's random testing functionality. The new default strategy for random testing first tests a property using the in-order enumeration for 10 seconds, then alternates between enumeration and the ad hoc random generator for 10 minutes, then finally switches over to just random generation. This will provide users with the complementary benefits of in-order and random enumeration as shown in our results without the need for manual configuration.

Acknowledgements. Thanks to Neil Toronto for helping us find a way to select from the natural numbers at random. Thanks to Hai Zhou, Li Li, Yuankai Chen, and Peng Kang for graciously sharing their compute servers with us. Thanks to Matthias Felleisen for helpful comments on the writing.

Bibliography

Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. *Journal of Automated Reasoning* 49(3), pp. 453–491, 2012.

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In *Proc. Frontiers of Combining Systems*, 2011.

Lukas Bulwahn. The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing Under One Roof. In *Proc. International Conference on Certified Programs and Proofs*, 2012.

Benjamin Canou and Alexis Darrasse. Fast and Sound Random Generation for Automated Testing and Benchmarking in Objective Caml. In *Proc. Workshop on ML*, 2009.

Koen Classen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. International Conference on Functional Programming*, 2000.

Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions of Software Engineering* 10(4), 1984.

Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional Enumeration of Algebraic Types. In *Proc. Haskell Symposium*, 2012.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proc. Programming Language Design and Implementation*, 2005.

Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming* 23(5), 2013.

Phillip Heidegger and Peter Thiemann. Contract-Driven Testing of JavaScript Code. In *Proc. International Conference on Objects, Models, Components, Patterns*, 2010.

Andrew J. Kennedy and Dimitrios Vytiniotis. Every Bit Counts: The binary representation of typed data and programs. *Journal of Functional Programming* 22(4-5), 2010.

Casey Klein and Robert Bruce Findler. Randomized Testing in PLT Redex. In *Proc. Scheme and Functional Programming*, pp. 26–36, 2009.

Casey Klein, Robert Bruce Findler, and Matthew Flatt. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, 2013.

Barton P. Miller, Lars Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* 33(12), 1990.

Michał H. Pałka. Testing an Optimising Compiler by Generating Random Lambda Terms. Licentiate of Philosophy dissertation, Chalmers University of Technology and Göteborg University, 2012.

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proc. International Workshop on Automation of Software Test*, 2011.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Proc. Haskell Symposium*, 2008.

Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. In *Proc. European Symposium on Programming*, pp. 229–248, 2013.

Paul Tarau. Bijective Term Encodings. In *Proc. International Colloquium on Implementation of Constraint LOgic Programming Systems*, 2011.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. Programming Language Design and Implementation*, 2011.

9. Appendix

This section shows the precise code we used to get the numbers discussed in section 1. This is the Quickcheck code for the first conjecture:

```
main :: IO ()
main = quickCheckWith
       stdArgs {maxSuccess=1000}
       prop_arith

prop_arith :: Int -> Int -> Bool
prop_arith x y =
  not ((x /= y) &&
       (x*2 == x+10))
```

The corresponding Redex code:

```
(define-language empty-language)
(redex-check
 empty-language
 (integer_x integer_y)
 (let ([x (term integer_x)]
       [y (term integer_y)])
   (not (and (not (= x y))
             (= (* x 2) (+ x 10))))))
```

The Quickcheck code for the second conjecture:

```
main :: IO ()
main = quickCheckWith
       stdArgs {maxSuccess=1000}
       prop

nth :: [a] -> Int -> Maybe a
nth xs n = if (n >= 0) && (n < length xs)
           then Just (xs !! n)
           else Nothing

prop :: [Int] -> [Int] -> Int -> Bool
prop xs ys n =
  (nth (xs ++ ys) (length xs + n)) ==
  (nth xs n)
```

The corresponding Redex code:

```
(define (nth l n)
  (with-handlers ([exn:fail? void])
    (list-ref l n)))
(redex-check
 empty-language
 ((any_1 ...) (any_2 ...) natural)
 (let ([xs (term (any_1 ...))]
       [ys (term (any_2 ...))]
       [n (term natural)])
   (equal? (nth (append xs ys) (+ (length xs)
n))
           (nth xs n))))
```