Practical, Fair, and Efficient Enumeration for Algebraic Data-Structures

Max New[1], Burke Fetscher[1], Jay McCarthy[2], and Robert Bruce Findler[1]

[1] Northwestern University
[2] Vassar College

Abstract. This paper reports on the design of a set of enumeration combinators that are efficient, fair, and practical. They are efficient because most of the combinators produce enumerations that support indexing in time proportional to the log of the given index. In practical terms, this means that we can typically compute the $2^{100,000}$th element of an enumeration in a few milliseconds.
Fairness means that when the combinators build a new result enumeration out of other ones, indexing into the result enumerator does not index disproportionally far into just a subset of the given enumerators. For example, this means that enumeration of the nth element of a product indexes about $\sqrt{n}$ elements into each of its components.
Our combinators are practical because they support the entire language of Redex models, providing a new generator for Redex's property-based testing. The paper reports on an empirical comparison between enumeration-based property generation and ad hoc random generation, showing that enumeration is more effective than ad hoc random generation in short time-frames.

1 Introduction

```
(define-language L
  (e ::=
     (e e)
     (λ (x : τ) e)
     x
     +
     integer)
  (τ ::= int (τ → τ))
  (x ::= variable))
```

This paper reports on a new enumeration library that provides functions that transform natural numbers into datastructures in a way that supports property-based testing. Our primary application of this library is in Redex, a domain-specific programming language for operational semantics. Redex programmers write down a high-level specification of a grammar, reduction rules, type systems, etc., and properties that should hold for all programs in these languages that relate, say, the reduction semantics to the type system. Redex can then generate example programs and test the property, looking for counterexamples. Before this work, we largely relied on an ad hoc random generator to find counterexamples but, inspired by the success of Lazy Small Check (Runciman et al. 2008) and FEAT (Duregård et al. 2012), we added enumeration-based generation.

To give a flavor for the new capability in Redex, consider the float above, which contains a Redex program that defines the grammar of a simply-typed calculus plus numeric constants. With only this much written down, a Redex programmer can ask for first nine terms:

```
'a                  '+                  0
'(a a)              '(λ (a : int) a)    'b
1                   '(a +)              '(λ (a : int) +)
```

or the 100,000,000th term:

```
(λ (r
    :
    (((int → int) → int)
     →
     (int → (int → (int → int)))))
  ((λ (a : (int → int)) 0) (0 0)))
```

which takes only 10 or 20 milliseconds to compute.

Thanks to our new library, we can randomly select large natural numbers and use them as a way to generate expressions for our property-based testing. We can also simply enumerate the first few thousand terms and use those as inputs.

The application of our combinators significantly influenced their design, leading us to put special emphasis on "fair" enumerators. At the application level, fairness ensures that simple modifications to the Redex grammar do not significantly change the quality of the terms that the enumerator generates. We give a fuller discussion of fairness in section 3, after introducing our library in section 2. Section 4 connects our combinators to Redex in more detail, explaining how we can support arbitrary Redex patterns (as they go significantly beyond simple tree structures).

To evaluate our combinator library, we conducted an empirical evaluation of its performance, as compared to the pre-existing ad hoc random generator. We compared them using a benchmark suite of Redex programs. We give a detailed report on the results in section 5, but the high-level takeaway is that our enumerators find more bugs per second in short time-frames, while the ad hoc random generator is more effective on long time-frames. Accordingly, the current implementation of Redex switches between generation modes based on the amount of time spent testing. Finally, section 6 discusses related work and section 7 concludes.

2 Enumeration Combinators

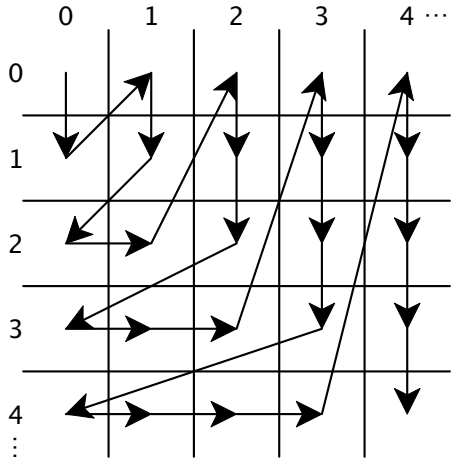
Our enumeration library provides some basic enumerations and combinators to build up more complex ones and this section gives an overview of the combinators. Each enumerator consist of three pieces: a `to-nat` function that computes the index of anything in the enumeration, a `from-nat` function that computes a value from an index, and a size of the enumeration, which can be either a natural number or `+inf.0`. In addition, the `to-nat` and `from-nat` functions form a bijection between the natural numbers (up to the size) and the values that are being enumerated.

The most basic enumerator is `nats/e`. Its `to-nat` and `from-nat` functions are simply the identity function and its size is `+inf.0`. The 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, where the elements are put in correspondence with the naturals in order they are given.

The next combinator is the pairing operator `cons/e`. It takes two enumerations and returns an enumeration of pairs of those values. If one of the enumerations is finite, the enumeration 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.

Pairing infinite enumerations require more care. If we imagine our sets as being laid out in an infinite two dimensional table, `cons/e` walks along the edge of ever-widening squares to enumerate all pairs:



which means that `(cons/e nat/e nat/e)`'s first `15` elements are

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

The n-ary `list/e` generalizes the binary `cons/e` that can be interpreted as a similar walk in an n-dimensional grid. We discuss this in detail in section 3.

The disjoint union enumerator, `disj-sum/e`, takes two or more pairs of enumerators and predicates. The predicates must distinguish the elements of the enumerations from each other. 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` positions. 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"    7      "g"    8      "h"
```

We generalize this combinator and describe it in detail in section 3 as well.

The combinator `fix/e : (enum → enum) → enum` computes takes fixed points of enumerator functionals in order to build recursive enumerations. 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 nat/e lon/e) cons?)))))
```

and here are its first 15 elements:

```
'()         '(0)        '(0 0)      '(1)        '(1 0)
'(0 0 0)    '(1 0 0)    '(2)        '(2 0)      '(2 0 0)
'(0 1)      '(1 1)      '(2 1)      '(3)        '(3 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 (λ (x) x))` will cause its `from-nat` function to 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))
         nat/e))
```

Also, we can exploit the bidirectionality of our enumerators 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 22 elements of `(except/e nat/e 13)` are

```
0    1    2    3    4    5    6    7    8    9    10
11   12   14   15   16   17   18   19   20   21   22
```

The `from-nat` function for `except/e` simply uses the original enumerator's `to-nat` on the given element and then either subtracts one before passing the natural number along (if it is above the given exception) or simply passes it along (if it is below). Similarly, the `except/e`'s `to-nat` function calls the input enumerator's `to-nat` function.

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` does not always have this property. It accepts an enumerator and a function from elements 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 nat/e (λ (i) (nats-above/e i)))
```

Here are the first 15 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)    '(2 . 4)    '(3 . 4)    '(4 . 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 Fairness

A fair enumeration combinator is one that indexes into its given enumerators roughly equally, instead of indexing deeply into one and shallowly into a different one. For example, imagine we waned to build an enumerator for lists of length 4. This enumerator is one way to build it:

```
(cons/e nat/e (cons/e nat/e
 (cons/e nat/e (cons/e nat/e
  (fin/e null)))))
```

Unfortunately, it is not fair. The 1,000,000,000th element is `'(31622 70 11 0)` and, as you can see, it has indexed far more deeply into the first `nat/e` than the others. In contrast, if we balance the `cons/e` expressions like this:

```
(cons/e
 (cons/e nat/e nat/e)
 (cons/e nat/e nat/e))
```

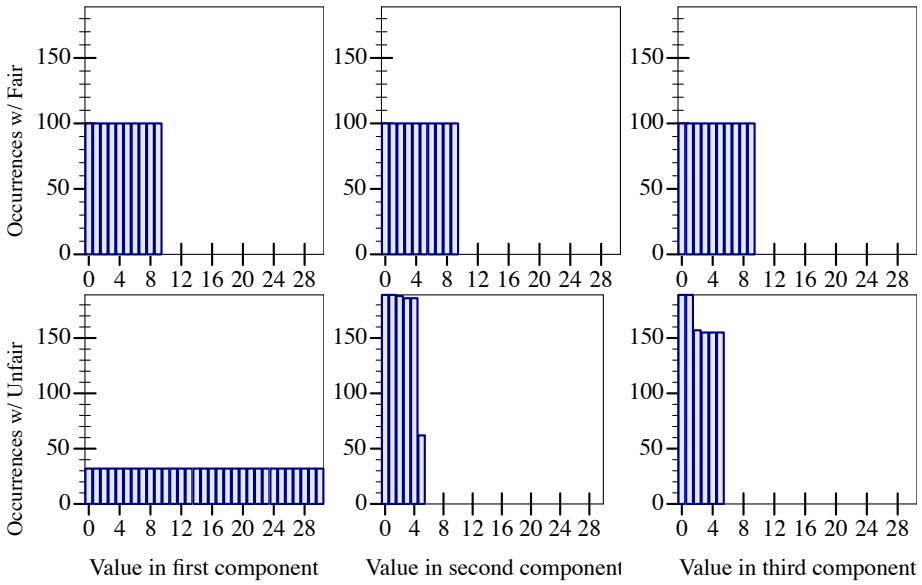


Figure 1: Histograms of the occurrences of each natural number in fair and unfair tuples

(and then were to use `map/e` to adjust the elements of the enumeration to actually be lists), then the 1,000,000,000 element is `'(177 70 116 132)`, which is much more balanced. This balance is not specific to just that index in the enumeration, either. Figure 1 shows histograms for each of the components when using the unfair `(cons/e nat/e (cons/e nat/e nat/e))` and when using a fair tupling that combines three `nat/e` enumerators. The x-coordinates of the plot correspond to the different values that appear in the tuples and the height of each bar is the number of times that particular number appeared when enumerating the first 1,000 tuples. As you can see, all three components have the same set of values for the fair tupling operation, but the first tuple element is considerably different from the other two when using the unfair combination.

The subtle point about fairness is that we cannot restrict the combinators to work completely in lock-step on their argument enumerations, or else we would not admit *any* pairing operation as fair. After all, a combinator that builds the pair of `nat/e` with itself we must eventually produce the pair `'(1 . 4)`, and that pair must come either before or after the pair `'(4 . 1)`. So if we insist that at every point in the enumeration that the combinator's result enumeration has used all of its argument enumerations equally, then pairing would be impossible, as would many other combinators.

Instead, we insist that there are infinitely many places in the enumeration where the combinators reach an equilibrium. That is, there are infinitely many points where the result of the combinator has used all of the argument enumerations equally.

As an example, consider the fair nested `cons/e` from the beginning of the section. As we saw, at the point 1,000,000,000, it was not at equilibrium. But at 999,999,999,999,

it produces `'(999 999 999 999)`, and indeed it has indexed into each of the four `nat/e` enumerations with each of the first 1,000 natural numbers.

In general, that fair four-tuple reaches an equilibrium point at every n^4 and `(cons/e nat/e nat/e)` reaches an equilibrium point at every perfect square. The diagonal in the square diagram from section 2 illustrates the first few equilibrium points for `(cons nat/e nat/e)`.

As an example of an unfair combinator consider `triple/e`:

```
(define (triple/e e_1 e_2 e_3)
  (cons/e e_1 (cons/e e_2 e_3)))
```

and the first 25 elements of its enumeration:

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

The first argument enumeration has been called with `3` before the other arguments have been called with `2` and the first argument is called with `4` before the others are called with `3` this behavior persists for all input indices, so that no matter how far we go into the enumeration, there will never be an equilibrium point after `'(0 0 . 0)`.

Fair combinators give us predictability for programs that use our enumerators. In Redex, our main application of enumeration combinators, fairness ensures that when a Redex programmer makes an innocuous change to the grammar of the language (e.g. changing the relative order of two subexpressions in an expression form) the enumeration quality is not significantly affected. For example, consider an application expression. From the perspective of the enumerator, an application expression looks just like a list of expressions. An unfair enumerator might cause our bug-finding search to spend a lot of time generating different argument expressions and always using similar (simple, boring) function expressions.

Of course, the flip-side of this coin is that using unfair combinators can improve the quality of the search in some cases, even over fair enumeration. For example, when we are enumerating expressions that consist of a choice between variables and other kinds of expressions, we do not want to spend lots of time trying out different variables because most models are sensitive only to when variables differ from each other, not their exact spelling. Accordingly unfairly biasing our enumerators away from different variables can be win for finding bugs. Overall, however, it is important that we do have a fair set of combinators that correspond to the way that Redex programs are constructed and then when Redex programs are compiled into the combinators, the compilation can use domain knowledge about Redex patterns to selectively choose targeted unfairness, but still use fair combinators when it has no special knowledge.

3.1 Formal Definition of Fairness

Our definition of fairness necessitates indexing enumerations with arbitrarily large natural numbers, so we restrict our attention to infinite enumerators.

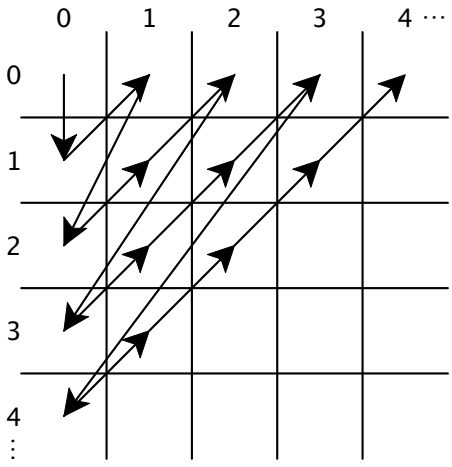
A function $c : Enum(a_1) \cdots Enum(a_k) \to Enum(T(a_1, \cdots, a_k))$, for some type-level function T, is an enumeration combinator if we can extract two functions that fully define its bijection. The first, $args_c : \mathbb{N} \to ([\mathbb{N}], \ldots, [\mathbb{N}])$ where the output tuple has length k, returns the k-tuple of lists of indices needed to index into the input enumerations when decoding from a given index. The second, $build_c : ([a_1], \ldots, [a_k]) \to T(a_1, \ldots, a_k)$ is a function that is linear in its input arguments, and thus using all of its inputs to construct its output. This function builds a value of the enumeration from components from the argument enumerations. These functions together fully specify the combinator; each of the elements of the lists of $args_c$'s result are supplied to the corresponding argument combinator and those results are then passed to $build_c$. If one of the lists has no elements, the corresponding argument combinator is not used and if one of the lists has multiple elements, the corresponding combinator is used multiple times.

For convenience, we say that two lists are equivalent if one is a permutation of the other.

We say that an enumeration combinator `c` is fair if, for every natural number m, there exists a natural number $M > m$ such that for every $h, j \in \{1, \ldots, k\}$, if you apply $args_c$ to every value greater than or equal to 0 and less than M, and concatenate all of the lists in the hth column into a list L_h and in the jth column into a list L_j then L_j and L_h are equivalent. In other words, M is an equilibrium point and thus when enumerating all values up to M in the result enumeration, the values supplied to argument enumerations will all be the same. Any other combinator is unfair.

3.2 Fair Tupling

The combinatorically-inclined reader may have noticed in our description of `cons/e` that we did not use the classic Cantor pairing function for our bijection, which can be interpreted as a more triangular grid walk:



Instead we use Szudzik (2006)'s bijection, that we refer to as "boxy" pairing. The two bijections are quite similar, they are both quadratic functions with similar geometric interpretations: boxy traces out the edges of increasingly large squares and Cantor traces out the bottoms of increasingly large triangles. This point of view leads to obvious generalizations to n-tuples. Generalized boxy should trace out the outer faces of an n cube and generalized Cantor should trace out the outer face of an n simplex.

Despite their conceptual similarity, we found the boxy enumeration lends itself to a more efficient implementation. To understand why, note that most combinatorics applications of pairing functions are chiefly concerned with one half of the bijection: the one from pairs of natural numbers to natural numbers.

$$cantor\_pair(m,n) = \frac{(n+m)(n+m+1)}{2} + m$$

$$box\_pair(m,n) = \begin{cases} x^2 + x + y & \text{if } x \geq y \\ x + y^2 & \text{if } x < y \end{cases}$$

For enumerations we are primarily concerned with the other half of the bijection, since that is the one used to generate terms. For the pairing case, these functions have fairly straightforward inverses, but their generalizations do not. This is the generalization of the cantor pairing function to length k tuples:

$$cantor\_tuple(n_1, n_2, \ldots, n_k) = \binom{k-1+n_1+\cdots+n_k}{n} + \cdots + \binom{1+n_1+n_2}{2} + \binom{n_1}{1}$$

This means to be able to invert such equations is to solve a certain class of arbitrary degree diophantine equations, which are not generally solvable. We can easily define a highly inefficient (but correct) way to compute the inverse by trying every natural number, in order, applying the original *cantor_tuple* function to see if it was the argument given. Tarau (2012) improves on this implementation, but the algorithm there is still a search procedure, and we found it too slow to use in practice.

So how do we generalize boxy pairing to boxy tupling and how do we compute an efficient inverse? A geometric interpretation gives the answer. If we look back at the grid-walk describing `cons/e` a pattern emerges, as the input indices grow, we trace out increasingly large squares. For example, when we encode `42` with `cons/e`, we first take the square root with remainder, giving us a root of `6` with a remainder of `8`. This tells us that the larger value in the pair is `6`, and it's the `8`th such pair. Then we construct an enumeration of pairs whose larger value is `6`, and index into that enumeration with `8`, giving us the pair `'(6 . 0)`. Then we can easily get the inverse transform by taking that pair, taking the maximum of `6` and `0` to get `6`, and then we use the other half of the enumeration of pairs above to find where in the enumeration of pairs with larger value `6` this `'(6 . 0)` is, and we find it is the `8`th such pair. Then we square `6` to get `36` and finally add `8` to get our original value of `42`.

The key idea is that we find what "layer" a value is on and we bootstrap the implementation with existing implementations of `cons/e` and `disj-sum/e` for finite enumerations, giving us both halves of the layer enumeration in one fell swoop. Fortunately, unlike the Cantor pairing function, this is naturally generalized to an n-tupling function, by using the nth root and nth power intead of the square root and squaring. Otherwise the description is the same.

We now prove that `list/e`, using the generalized boxy bijection, is fair. The following is a function that takes a positive number `k` and returns the decoding function the boxy bijection for `k`-tuples specialized to natural numbers:

```
(define (box-untuple k)
  (λ (n)
    (define layer (integer-root n k))
    (define smallest (expt layer k))
    (define layer/e (bounded-list/e k layer))
    (decode layer/e (- n smallest))))
```

Here `bounded-list/e` is a function that takes a positive integer list length `k` and a natural number bound `layer` and returns an enumeration of lists of length `k` that have a maximal value of `layer`. For example the values of `(bounded-list/e 3 2)` are

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

Since the elements of the enumerated lists are bounded by a specific number, `bounded-list/e` always returns a finite enumeration, which we denote `e`. Furthermore, enumerating every element of `e` will use all of its arguments in exactly the same way since for any tuple `(i_1 i_2 ... i_k)` in `e`, every permutation of that tuple is also in `e`, since it has the same maximum.

In order to show that `list/e` is fair as we've defined it, we must define its `args` and `build` functions. `args` is defined using the process above, which given an `i` produces a list `(i_1 ... i_k)` of indices with max $\lfloor i^{1/k} \rfloor$ but to satisfy the type of `args`, it wraps each of those indices in a list to return `((i_1)) ... ((i_k))`, meaning that the enumeration produced by `list/e` uses each of its argument enumerations once per decode. `build` is defined as `(define (build xs) (map first xs))`, and it is linear since each of its arguments will only have one element in them since they were produced by the `args` function.

Theorem 1. `list/e` *is a fair combinator*

Proof. With this lemma in hand, we prove that `list/e` is fair by showing that for any infinite argument enumerations `(e_1 e_2 ... e_k)` there is an infinite increasing sequence $(M_0, M_1, ...)$ of natural numbers such that for any M_i in the sequence, enumerating with all indices less than M_i in `(list/e e_1 e_2 ... e_k)` calls all arguments `e_j` with the same indices. This is sufficient to show that `list/e` is fair since for any natural number m there is some $M_i > m$ since $(M_0, M_1, ...)$ is infinite and increasing.

Specifically, our sequence is the sequence of kth powers, that is $M_i = (i+1)^k$. Let $h_1, h_2 \in \{1, \ldots, k\}$, representing two arbitrary argument enumerations. We proceed by induction on i. For $i = 0$, $M_0 = 1$, so we need only consider the call `(args 0)` which results in `((0) ... (0))` so $L_{h_1} = L_{h_2} = [0]$ i.e., both argument enumerations are used with the value `0` and only `0`. Next, assuming the theorem holds for all M_i with $i < l$ we seek to prove it holds for M_l.

If L'_{h_1}, L'_{h_2} are the lists of h_1, h_2 values resulting from calls to args for 0 up to $M_{l-1} - 1 = l^k - 1$, then by inductive hypothesis L'_{h_1}, L'_{h_2} are permutations of each other. Then if we can show that the lists of h_1, h_2 values from calls to args for $M_{l-1} = l^k$ to $M_l - 1 = (l+1)^k - 1$ are permutations of each other, then we know that the h_1, h_2 calls from 0 to $M_l - 1$ will be permutations of each other. Those indices j are precisely the natural numbers for which $\lfloor \sqrt[k]{j} \rfloor = l$ and thus together they fully enumerate the values of `(bounded-list/e k l)`, thus by our lemma, when called with those indices, the h_1, h_2 indices will be the same. Thus indexing from 0 to M_l uses all `e_i` equally, so by induction, `list/e` is fair. □

Now, let `cantor-list/e` be a version of `list/e` be a that uses the generalized Cantor n-tupling bijection described above. This is highly analogous to the boxy `list/e`. For example, their `args` functions both return lists of length 1 in every slot and their `build` functions are exactly the same. To be precise `(args i)` is equal to `((i_1) (i_2) ... (i_k))` where

$$i = \binom{i_1}{1} + \binom{1 + i_1 + i_2}{2} + \cdots + \binom{k - 1 + i_1 + \cdots + i_k}{k}$$

which the last equation is exactly generalized Cantor k-tupling.

Theorem 2. `cantor-list/e` *is fair*

Proof. We elide most details of the proof since it is almost exactly the same as the proof for boxy `list/e`. First, we note that as described in Cegielski and Richard (1999), the Cantor tupling bijection works in a similar way to the boxy bijection, that is, for k inputs it traces out the outer face of increasingly large k-simplices. This means it can be computed by taking a "root" of the input index and then using the remainder to index into a finite enumeration. In particular for k inputs, it takes the k-th simplicial root giving a root of l and remainder r then uses r to index into an enumeration of all lists of length k whose elements sum to l. And as with `bounded-list/e`, an enumeration of lists of length k that sum to the value l, when fully enumerated, calls the arguments `e_i` with the same values. Thus we can show that there is an infinite increasing sequence $(M_0, M_1, ...)$ where indexing 0 to M_i uses all `e_i` equally. For k arguments, $M_i = \binom{i+k-1}{k}$, the ith k-simplicial number. The proof is then precisely analagous to the proof for boxy `list/e`. □

Now recall `triple/e`, as defined at the beginning of this section. Let $args_{cons}$ be the args function for $cons/e$, which uses the boxy bijection on pairs. Then the *args* function for `triple/e` is $args(i) = ([i_1], [i_2], [i_3])$ where $([i_1], [j]) = args_{cons}(i)$ and $(i_2, i_3) = args_{cons}(j)$. The build function for `triple/e` is `(define (build is_1 is_2 is_3) (cons (first is_1) (cons (first is_2) (first is_3))))`.

Theorem 3. `triple/e` *is unfair*

Proof. To prove something is unfair we must show that there is a natural number M such that for every $m > M$, there are indices $h, j \in \{1, 2, 3\}$ such that L_h and L_j are not equivalent. To show that two lists are not equivalent it is sufficient to show that there

is a value in one that is not in the other. Specifically we will show that for all natural numbers greater than 4, L_1 contains an index greater than any found in L_2.

First we establish some elementary properties of $args_cons$, defined using the boxy bijection on 2 enumerations. First, for any natural number i, if $args(i) = ([i_1],[i_2])$ then $i_1, i_2 \leq \lfloor\sqrt{i}\rfloor$. This is a direct consequence of the definition of the boxy bijection, which is defined by taking the floor of the square root of i and then producing a pair whose max is $\lfloor\sqrt{i}\rfloor$. Next, for any positive natural number i, there is a natural number l such that $\lfloor\sqrt{l}\rfloor = \lfloor\sqrt{i}\rfloor - 1$ and $args(l) = ([\lfloor\sqrt{l}\rfloor],[0],[0])$.

First there is a natural number l with $\lfloor\sqrt{l}\rfloor = \lfloor\sqrt{i}\rfloor - 1$ for any $i > 0$. Then the rest of the statement is true by the definition of the boxy bijection since at least one of the lists in the triple $args(i)$ must be $\lfloor\sqrt{i}\rfloor$ since it is selected from `(bounded-list/e 3 (floor (sqrt i)))`, so the values in `(bounded-list/e 3 (floor (sqrt l)))` must have all been enumerated before `i` since $\lfloor\sqrt{l}\rfloor < \lfloor\sqrt{i}\rfloor$.

Thus for any natural number $i > 9$, if L_1, L_2 are the elements from the first and second column when applying $args$ to the values 0 to $i-1$, we get that L_1 contains some l such that $\lfloor\sqrt{l}\rfloor = \lfloor\sqrt{i}\rfloor - 1$ by our second lemma. On the other hand, since the values in L_2 go through two calls to $args_{cons}$, for any $x \in L_2$, $x \leq \lfloor\sqrt{\lfloor\sqrt{i}\rfloor}\rfloor$. So we need to prove that $\lfloor\sqrt{i}\rfloor - 1 > \lfloor\sqrt{\lfloor\sqrt{i}\rfloor}\rfloor$ which is true for all$i > 9$, so L_1 contains a value larger than any in L_2, so L_1 and L_2 are not equivalent. Thus `triple/e` is unfair. □

3.3 Fair Union

Next we turn to `disj-sum/e`, the operation corresponding to the union of several enumerators. Recall that the arguments for `disj-sum/e` are not just enumerators, but pairs consisting of an enumerator and a predicate to that returns true if a value is in that enumerator, in line with the Racket convention of using untagged union types. We will denote a call to `disj-sum/e` with `k` arguments by `(disj-sum/e (cons e_1 1?) (cons e_2 2?) ... (cons e_k k?))` where given that `x` is in one of the `e_i`, `(1? x)` is true if there is some index `i` such that `(decode e_1 i)` is `x`.

`disj-sum/e` is relatively easy to define fairly. Given two infinite argument enumerations, we can simply alternate between one and the other, so the first ten elements of `(disj-sum/e (nat/e nat?) (symbol/e sym?))` are simply the first five elements of `nat/e` and `string/e` interleaved, where `symbol/e` is some enumeration of all Racket symbols:

```
0    'a    1    'b    2    'c    3    'd    4    'e
```

Again, to achieve fairness we cannot simply use the binary version of `disj-sum/e` arbitrarily. For example, if we defined

```
(define (union-three/e ep_1 ep_2 ep_3)
  (define e_2 (car ep_2))
  (define 2?  (cdr ep_2))
  (define e_3 (car ep_3))
  (define 3?  (cdr ep_3))
  (define (2-or-3? x) (or (2? x) (3? x)))
```

```
(disj-sum/e ep_1 (cons (disj-sum/e ep_2 ep_3) 2-or-3?)))
```

then enumerating the first 10 elements of `(union-three/e (cons nat/e nat?) (cons symbol/e sym?) (cons float/e float?))` is unfairly weighted to the first argument, as shown on the left in figure 2.

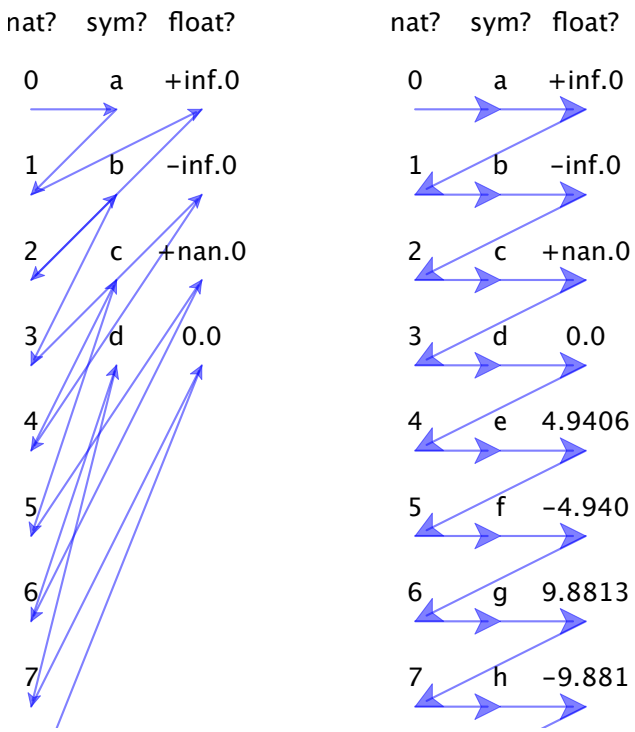


Figure 2: Unfair (left) and fair (right) disjoint sum enumerations

A fair generalization is fairly obvious. First we decode each argument with the value `0`, then each with `1`, and so on. So when given a call `(decode (disj-sum/e (cons e_1 1?) ... (cons e_k k?)) i)` we divide `i` by `k`, giving us a quotient of `q` and remainder of `r`. Then we call `(decode e_r q)`. We see that, like `list/e`, we have a notion of "layers", if we think of the input enumerations as infinitely tall columns side by side, each layer is a horizontal slice of the columns. So using the same enumerations as before, `(disj-sum/e (cons nat/e exact-integer?) (cons symbol/e sym?) (cons float/e float?))` looks like the right-hand side of figure 2.

Unlike `list/e`, `disj-sum/e` enumerator also has an intuitive notion of fairness for finite enumerations. For example this enumeration:

```
(disj-sum/e (cons (fin/e 'a 'b 'c 'd) sym?)
```

```
(cons nat/e number?)
(cons (fin/e "x" "y") string?))
```

cycles through its two finite enumeration arguments 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
```

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 some index.

Now we formalize `disj-sum/e` for our definition of a combinator. Unfortunately `disj-sum/e`'s arguments do not fit our definition of a combinator, because it takes predicates as well as enumerators. We could adapt the combinator definition to consider only the decoding part of an enumerator, but doing so would add complexity without increasing understanding, so we ignore the predicates for the purposes of the proof. The *args* function for `disj-sum/e` when called with `k` arguments is defined by $args(i) = ([],\ldots,[r],\ldots,[])$ where $i/k = q$ remainder r and the $[r]$ was in the qth element of the resulting tuple. The *build* function returns the element of the only non-empty list in its arguments, $build([],\ldots,[x],\ldots,[]) = x$.

Theorem 4. `disj-sum/e` *is fair.*

Proof. Let k be the number of input enumerations. The sequence $M_i = k(i+1)$ is an infinite increasing sequence for which when calling $args(j)$ with all $j = 0$ to $j = M_i - 1 = k(i+1) - 1$, and collecting results into lists $(L_1,\ldots,L_k)$ each list L_l is the same, specifically $[0,\cdots,i]$. We proceed by induction on i. For $i = 0$, $M_0 = k$ and for $j = 0\cdots(k-1)$, we have $args(j)$ is exactly $([],\ldots,[0],\ldots,[])$ where the non-empty list is the jth slot in the tuple soevery $L_l = [0]$.

Then assuming this holds for M_i, for $M_{i+1} = k(i+2)$ by inductive hypothesis, if we call $args(j)$ with j from 0 to $M_i - 1 = k(i+1) - 1$ and concatenate column-wise into lists $(A_1,\ldots,A_k)$, all of $A_1,\ldots,A_k$ are equivalent. Thus we need only show that calling $args(j)$ with j from $M_i = k(i+1)$ to $M_{i+1} - 1 = k(i+2) - 1$ and concatenating column-wise into lists $(B_1,\ldots,B_k)$ means that all of $B_1,\ldots,B_k$ are equivalent. Similarly to the base case, $args(k(i+2)+j)$ is equal to $([],\ldots,[k(i+1)],\ldots,[])$ where the $k(i+1)$ is in the jth slot, so $B_l = [k(i+1)]$ for each $l = 1,\ldots,k$. Thus `disj-sum/e` is fair. □

Next we turn to `union-three/e`. Its build function is the same as the build function for `disj-sum/e` specialized to three arguments. Its args function is defined by

$$args(i) = \begin{cases} ([\lfloor i/2 \rfloor],[],[]) & \text{if } i = 2 \bmod 2 \\ ([],[\lfloor i/4 \rfloor],[]) & \text{if } i = 1 \bmod 4 \\ ([],[],[\lfloor i/4 \rfloor]) & \text{if } i = 3 \bmod 4 \end{cases}$$

Theorem 5. `union-three/e` *is unfair.*

Proof. Now we will show that, similarly to the proof that `triple/e` is unfair, for any $n > 4$ there is an $i < n$ and h such that $args(i) = ([h], [], [])$ but there is no $j < n$ such that $args(i) = ([], [h], [])$. In that case, the lists L_1, L_2 produced by concatenating column-wise in calls of $args(i)$ for $i = 0, \ldots, n-1$ would not be equivalent.

This relies on two elementary properties of the *args* function. First, there is some $j \in 0, \ldots, n-1$ such that $args(j) = ([\lfloor (n-1)/2 \rfloor], [], [])$. This is true because either $n-1$ is even, so $args(n-1) = ([\lfloor (n-1)/2 \rfloor], [], [])$ or $n-1$ is odd, so $\lfloor (n-1)/2 \rfloor = \lfloor (n-2)/2 \rfloor$ and $args(n-2) = ([\lfloor (n-2)/2 \rfloor], [], [])$. Second, for every $j \in 0, \ldots, n-1$ such that $args(j) = ([], h, [])$ for some h, $h \leq \lfloor (n-1)/4 \rfloor$ which is a direct consequence of the definition of *args*. Finally, we rely on that fact that for $n > 9$, $\lfloor (n-2)/2 \rfloor > \lfloor (n-1)/4 \rfloor$, or equivalently that for $n > 8$, $\lfloor (n-1)/2 \rfloor > \lfloor n/4 \rfloor > 0$.

Thus for any n there is a value in L_1 that is greater than any in L_2, so L_1 and L_2 are not equivalent, so `decode-three` is unfair. □

4 Enumerating Redex Patterns

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 `x`s 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. For example, to generate lists of distinct naturals, we first define a helper function that takes as an argument a list of numbers to exclude

```
(define (no-dups-without eles)
  (fix/e (λ (lon/e)
           (disj-sum/e
            (cons (fin/e null) null?)
            (cons (dep/e
                   (except/e* nat/e eles)
                   (λ (new)
                     (no-dups-without
                      (cons new eles))))
                  cons?)))))
```

where `except/e*` simply calls `except/e` for each element of its input list. We can then define `(define no-dups/e (no-dups-without '()))` Here are the first `12` elements of the `no-dups/e` 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 subsequences 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 rule 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.

5 Empirical Evaluation

We compared 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 does not, due to the bug) and three functions that generate terms, each with a different strategy: in-order enumeration, random selection from a uniform distribution, and ad hoc random generation. The full benchmark consists of a number of models, including the Racket virtual machine model (Klein et al. 2013), a polymorphic λ-calculus used for random testing in other work (Pałka 2012; Pałka et al. 2011), the list machine benchmark (Appel et al. 2012), and a delimited continuation contract model (Takikawa et al. 2013), as well as a few models we built ourselves based on our experience with random generation and to cover typical Redex models.[3]

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 used two identical 64 core AMD machines with Opteron 6274s running at 2,200 MHz with a 2 MB L2 cache to run the benchmarks. Each machine has 64 gigabytes of memory. Our script runs each model/bug combination sequentially, although we ran multiple different combinations at once in parallel. We used the unreleased version 6.1.1.1 of Racket (of which Redex is a part); more precisely the version at git commit `878358ec9e`.[4]

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

[3] The full benchmark is online: `http://docs.racket-lang.org/redex/benchmark.html`

[4] `https://github.com/plt/racket/commit/878358ec9e2`

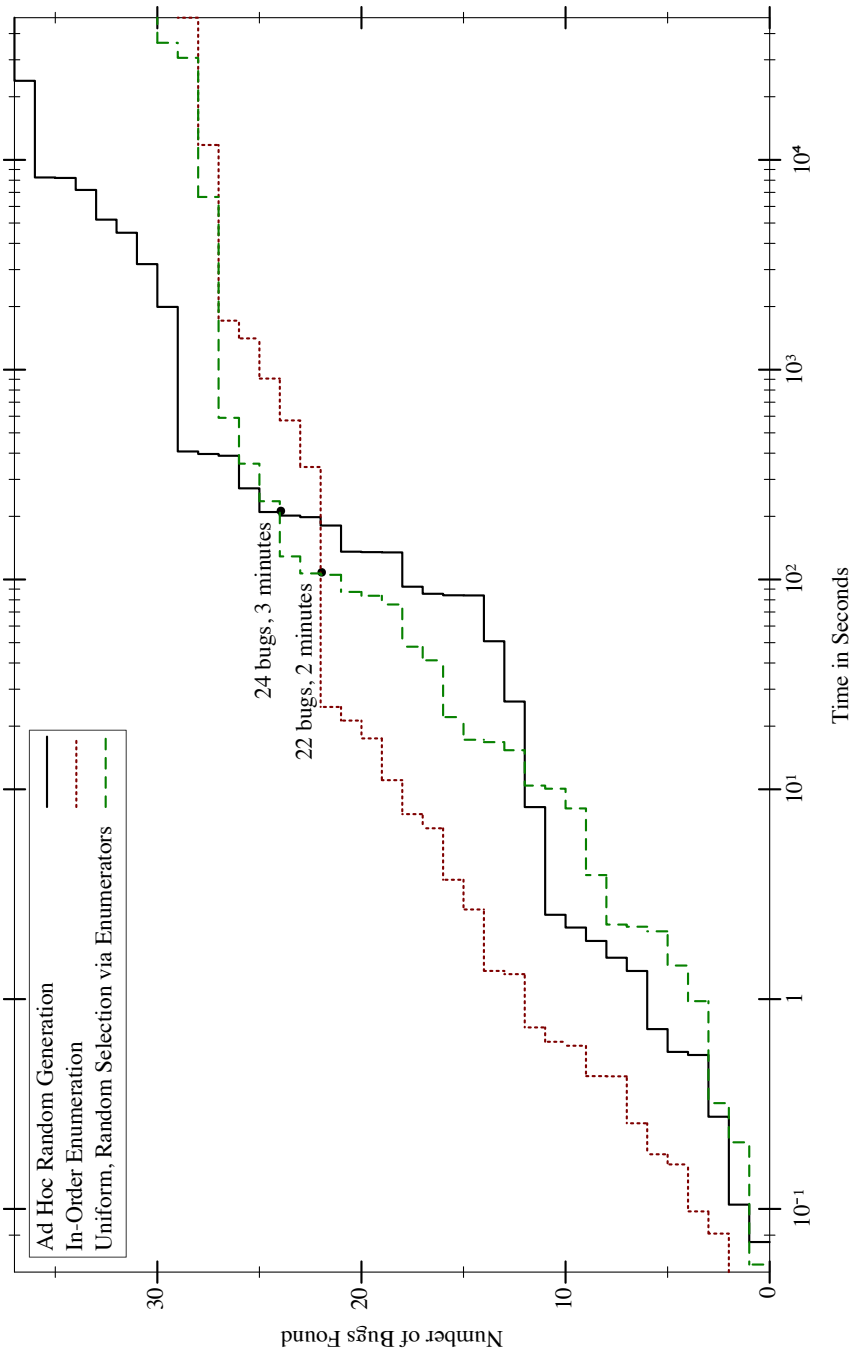


Figure 3: Overview of random testing performance of ad hoc generation, enumeration, and random indexing into an enumeration, on a benchmark of Redex models.

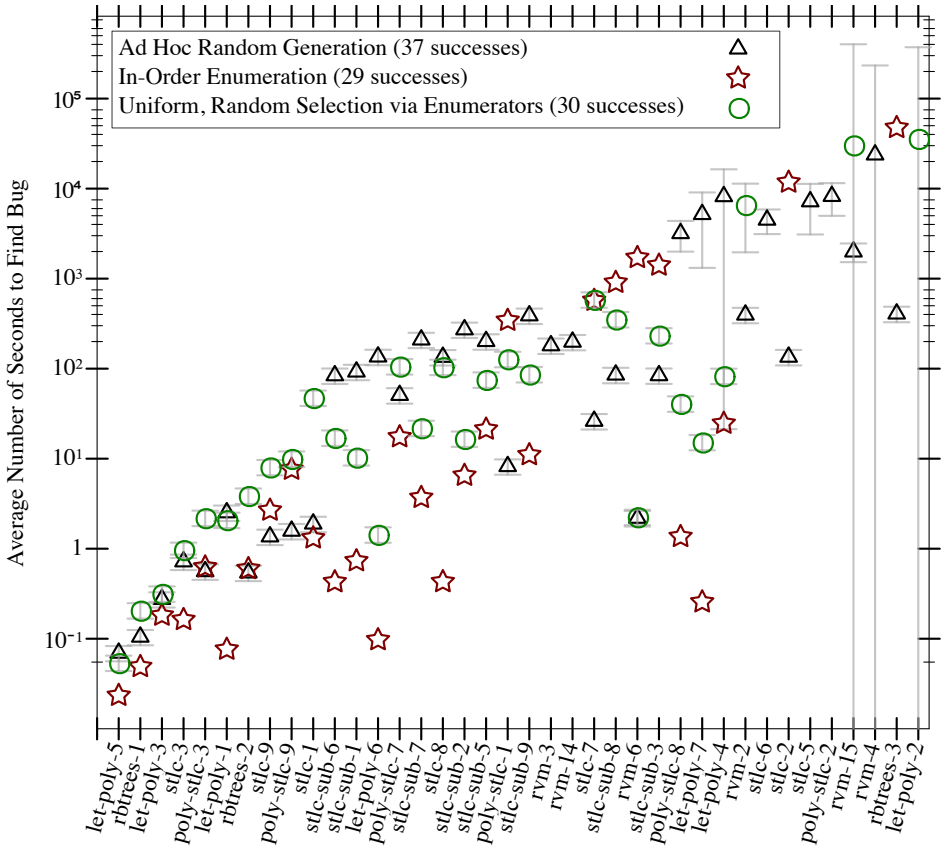


Figure 4: The mean time each generator takes to find the bugs, for each bug that some generator found; bars indicate 90% confidence intervals

We choose this distribution because it does not have a fixed mean. 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 larger than 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 our indices towards a small size.

The random-selection results are quite sensitive to the probability of picking the zero exponent from the geometric distribution. Because this method was our worst performing method, we empirically chose benchmark-specific numbers in an attempt to maximize the success of the random uniform distribution method. Even with this artificial help, this method was still worse, overall than the other two.

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 working on the enumerator.

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.

Figure 3 shows a high-level view of our results. Along its x-axis is time in seconds in 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 red 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 black 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 22 of the bugs found, the enumerator starts to lose and random selection from the uniform distribution starts to win until 3 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 50 bugs in the benchmark, no strategy was able to find more than 37 of them in a 24 hour period.

Figure 4 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 and #7 in let-poly) in much shorter times than the other approaches.

6 Related Work

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

6.1 Enumeration Methods

Tarau (2011)'s work on bijective encoding schemes for Prolog terms is most similar to ours. However, we differ in three main ways. First, our n-ary enumerators are fair (not just the binary ones). Second, 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. Third, 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 and fairness. Also, Feat has only one half of the bijection and thus cannot support `except/e` (and thus cannot easily support identifiers that contain all strings, except without a small set of keywords). Kuraj and Kuncak (2014) is similar to Feat, but with the addition of a dependent combinator. Like Feat, it does not support `except/e` or fairness.

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. This method also allows them to define an analagous dependent combinator. However, details of their system show that it is not well suited to using large indexes. In particular, 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 efficiently to enumerate all elements of the set being encoded.

Kiselyov et al. (2005) explicitly discuss fairness in the context of logic programming, but talk about it in the specific cases of fair disjunction and fair conjunction, but they do not have a unification of these two different types of fairness, nor do they have a concept of n-ary fair operators, only binary.

6.2 Testing Studies

Although the focus of our paper is on the combinators and fairness, the literature has few studies that specifically compare random testing and enumeration. We are aware of only one other, 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 experiment also 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 study compares selecting randomly from a uniform distribution like ours.

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.

Despite an early, convincing study on the value of random testing (Duran and Ntafos 1984) and an early influential paper (Miller et al. 1990), there are other papers that suggest that random testing is a poor choice for bug-finding. 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) 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.

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.

Of course, the reason Quickcheck and Redex find these bugs is because the distribution they use for integers is biased towards small integers, which is natural as those integers are usually more likely to be interesting during testing.

This paper is one of two papers submitted to ESOP that discusses random testing in the context of Redex. The other is entitled *Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System*. This paper has a less effective generator, but one that applies to all Redex models, unlike the other paper which applies only to models with type systems. The technical content is otherwise completely different, specifically the the enumeration in this paper and the goal-directed search in the other paper are independent and complementary.

7 Conclusion

Based on the results of our empirical study, 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 provides 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.

Patrick Cegielski and Denis Richard. On arithmetical first-order theories allowing encoding and decoding of lists. *Theoretical Computer Science* 222(1-2), 1999.

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.

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.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, Interleaving, and Terminating Monad Transformers. In *Proc. International Conference on Functional Programming*, 2005.

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.

Ivan Kuraj and Viktor Kuncak. SciFe: Scala Framework for Efficient Enumeration of Data Structures with Invariants. In *Proc. Scala Workshop*, 2014.

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.

Matthew Szudzik. An Elegant Pairing Function. 2006. `http://szudzik.com/ElegantPairing.pdf`

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.

Paul Tarau. Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection. In *Proc. International Conference on Logic Programming*, 2012.

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

8 Appendix

This section shows the precise code we used to get the numbers discussed in section 6. 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))))
```