Fair Enumeration Combinators

Anonymous

Abstract

Enumerations represented as bijections between the natural numbers and elements of some given type have recently garnered interest in property-based testing because of their efficiency and flexibility as counter-example generators. There are, however, many ways of defining these bijections, some of which are better than others. This paper offers a new property of enumeration combinators called *fairness* that identifies enumeration combinators that are more suited to property-based testing.

Intuitively, the result of a fair combinator indexes into its argument enumerations equally when constructing its result. For example, extracting the nth element from our enumeration of three-tuples indexes about $\sqrt[3]{n}$ elements into each of its components instead of, say, indexing $\sqrt[2]{n}$ into one and $\sqrt[4]{n}$ into the other two as you would if a three-tuple were built out of nested pairs. Similarly, extracting the nth element from our enumeration of a three-way union returns an element that is $\frac{n}{3}$ into one of the argument enumerators.

The paper presents a semantics of enumeration combinators, a theory of fairness, proofs establishing fairness of our new combinators and that certain combinations of fair combinators are not fair.

We also report on an evaluation of fairness for the purpose of finding bugs in operational semantics and type systems. We implemented a general-purpose enumeration library for Racket and used it to build generators for arbitrary Redex grammars. We used an existing benchmark suite of buggy Redex models to compare the bug finding capabilities of the original, ad hoc random generator to generators based on fair and unfair enumeration combinators. The enumeration using the fair combinators has complementary strengths to the ad hoc generator (better on short time scales and worse on long time scales) and using unfair combinators is worse across the board.

1. Introduction

In the past few years a number of different libraries have appeared that provide generic ways to build bijections between data structures and the natural numbers. First was Feat for Haskell in 2012[1], then SciFe for Scala in 2014[2], and `data/enumerate` for Racket this year.[3] (The last library is ours and was written as part of the work in this paper. Following the link reveals authorship.)

These libraries are efficient, providing the ability to extract the 2^{100}-th element of an enumeration of a data structure in milliseconds. What they lack, however, is a mathematically precise notion of the quality of their combinators. To be concrete, consider the pairing combinator, which all of the libraries provide. It accepts two enumerations and returns an enumeration of pairs of its inputs. There are many ways to build such an enumeration, based on the many ways to write a bijection between the natural numbers and pairs of natural numbers. One such function is given by $\lambda x.\lambda y.\ 2^y \cdot (2x+1) - 1$. This is a bijection (the inverse simply counts the number of times that 2 is a factor of its input to separate the "x" and "y" parts) that is easy to explain and efficient, taking logarithmic time in the result to compute in both directions. It is a poor choice for an enumeration library, however, because it explores "x" coordinate values much more quickly than the "y" coordinate. Indeed, in the first 10,000 pairs, the "x" coordinate has seen 4,999 but the biggest "y" coordinate seen is 13.

This paper offers a criterion called *fairness* that classifies enumeration combinators, rejecting the one in the previous paragraph as unfair and accepting ones based on the standard Cantor bijection and many others, including ones whose inverses are easier to compute in the n-tuple case (as explained later). Intuitively, a combinator is fair if indexing deeply into the result of the combinator goes equally deeply into all the arguments to the combinator.

The motivation for developing these enumeration libraries is bug-finding. Accordingly, we tested our concept of fairness via an empirical study of the capability of enumeration libraries to find bugs in formal models of type systems and operational semantics in Redex (Klein et al. 2012). We built a benchmark suite of 50 bugs (based on our experience writing Redex models and the experience of others mined from git repositories of Redex models) and compared the bug/second rate with three different generators. Two of the generators are based on a bijection between the expressions of the language and the natural numbers: one enumerates terms in order and the other selects a random (possibly large) natural number and uses that with the bijection. The third is an existing, ad hoc random generator that's been tuned for bug-finding in Redex models for more than a decade.

Our results show that fair in-order enumeration and ad hoc generation have complementary strengths, and that selecting a random natural number and using it with a fair enumeration is always slightly worse than one of the other two choices. We also replaced fair combinators with unfair ones and show that the bug-finding capabilities become significantly worse.

The next section introduces enumeration libraries, focusing on the Racket-based library to make the introduction concrete. Then, in section 3, we give an intuition-based definition of fairness and discuss our new n-ary combinators, whose designs are motivated by fairness. We follow up in section 4 with a formal definition of fair-

[1] `https://hackage.haskell.org/package/testing-feat`

[2] `http://kaptoxic.github.io/SciFe/`

[3] `http://docs.racket-lang.org/data/Enumerations.htmlREMOVEME`

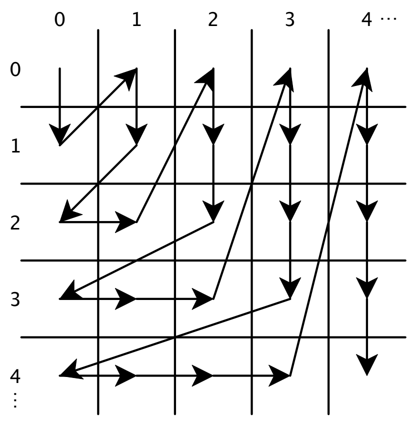


Figure 1: Pairing Order

ness and proofs that our combinators are fair and that a commonly-used combinator is unfair. Our evaluation of the different random generation strategies is discussed in section 5. The next two sections discuss related and future work, and the last section concludes.

2. Introduction to Enumeration

This section introduces the basics of enumeration via a tour of our enumeration library. Our library provides basic enumerations and combinators that build up more complex ones. Each enumeration consists of four pieces: a `to-nat` function that computes the index of any value in the enumeration, a `from-nat` function that computes a value from an index, the size of the enumeration, which can be either a natural number or positive infinity (written `+inf.0`), and a contract that captures exactly the values in the enumeration. Each enumeration has the invariant that the `to-nat` and `from-nat` functions form a bijection between the natural numbers (up to the size) and the values that satisfy the contract.[4]

The most basic enumeration is `below/e`. It accepts a natural number or `+inf.0` and returns an enumeration of that size. Its `to-nat` and `from-nat` functions are simply the identity functions. 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 disjoint union enumeration, `or/e`, takes two or more enumerations. 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 union of an enumeration of natural numbers and an enumeration of strings:

```
0    "a"    1    "b"    2    "c"    3    "d"
4    "e"    5    "f"    6    "g"    7    "h"
```

To compute the natural number given an element of the enumeration, each of the contracts on the input enumerations are tested. Accordingly, the `or/e` enumeration is a true union, not requiring explicit injection and projection into a new datatype.

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

[4] Our library also supports one-way enumerations as they can be useful in practice, but we do not talk about them here.

If one of the input enumerations is finite, the result enumeration loops through the finite enumeration, pairing each with an element from the infinite enumeration. If both are finite, it loops through the one with lesser cardinality. This corresponds to taking the quotient and 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 (using Szudzik (2006)'s bijection), as shown in figure 1. The first 12 elements of `(cons/e (below/e +inf.0) (below/e +inf.0))` enumerator 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)
```

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 combinator `delay/e` facilitates fixed points of enumerations, in order to build recursive enumerations. For example, we can construct an enumeration for lists of numbers:

```
(letrec ([lon/e
          (or/e (fin/e null)
                (cons/e (below/e +inf.0)
                        (delay/e lon/e)))])
  lon/e)
```

and here are its first 12 elements:

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

An expression like `(delay/e lon/e)` returns immediately, without evaluating the argument to `delay/e`. The first time a value is extracted from the enumeration, the expression is evaluated (and its result is cached). This means that a use of `delay/e` that is too eager, e.g.: `(define e (delay/e e))` will cause `from-nat` to fail to terminate. Switching the order of the arguments to `or/e` above also produces an enumeration that fails to terminate.

Our combinators rely on knowing the sizes of their arguments as they are constructed, 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 elements of the contract of a given enumeration and a new contract. Using this we can, for example, construct enumerations of natural numbers that start at some natural `i` beyond zero:

```
(define (naturals-above/e i)
  (map/e (λ (x) (+ x i))
         (λ (x) (- x i))
         (below/e +inf.0)
         #:contract (and/c exact-integer? (>=/c i))))
```

The first two arguments to `map/e` are functions that form a bijection between the values in the enumeration argument and the contract given as the final argument (`#:contract` is a keyword argument specifier). As it is easy to make simple mistakes when building the bijection, `map/e`'s contract randomly checks a few values of the enumeration to make sure they map back to themselves when passed through the two functions.

We exploit the bidirectionality of our enumerations to define the `except/e` enumeration. It accepts an element and an enumeration, and returns an enumeration without the given element. For example, the first 9 elements of (except/e (below/e +inf.0) 4) are

```
0    1    2    3    5    6    7    8    9
```

The from-nat function for except/e simply uses the original enumeration's to-nat on the given element and then either subtracts one (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 enumeration's to-nat function.

One important point about the combinators used so far: the conversion from a natural to a value takes time that is (a low-order) polynomial 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 numbers enumeration given above.

Our next combinator, cons/de, does not always have this property. It builds enumerations of pairs, but where the enumeration on one side of the pair depends on the element in the other side of the pair. For example, we can define an enumeration of ordered pairs (where the first position is smaller than the second) like this:

```
(cons/de [hd (below/e +inf.0)]
         [tl (hd) (naturals-above/e hd)])
```

A cons/de has two sub-expressions (in this example: (below/e +inf.0) and (naturals-above/e i)), each of which is named (hd and tl here). And one of the expressions may refer to the other's variable by putting it into parentheses (in this case, the tl expression can refer to hd). Here are the first 12 elements of the enumeration:

```
'(0 . 0)    '(0 . 1)    '(1 . 1)    '(1 . 2)
'(0 . 2)    '(1 . 3)    '(2 . 2)    '(2 . 3)
'(2 . 4)    '(0 . 3)    '(1 . 4)    '(2 . 5)
```

The implementation of cons/de has three different cases, depending on the cardinality of the enumerations it receives. If all of the enumerations are infinite, then it is just like cons/e, except using the dependent function to build the enumeration to select from for the second element of the pair. Similarly, if the independent enumeration is finite and the dependent ones are all infinite, then cons/de can use quotient and remainder to compute the indices to supply to the given enumerations when decoding. In both of these cases, cons/de preserves the good algorithmic properties of the previous combinators.

The troublesome case is when the dependent enumerations are all finite. In that case, we think of the dependent 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 cons/de enumeration must compute all of the enumerations for the second component as soon as a single (sufficiently large) number is passed to from-nat, which can, in the worst case, take time proportional to the magnitude of the number.

Our library has a number of other combinators not discussed here, but these are the most important ones and give a flavor of the capabilities of enumerations in the library. The documentation, http://docs.racket-lang.org/data/Enumerations.html, lists all of the combinators.

3. Fairness and Fair Combinators

This section introduces our definition of fairness in a precise but informal way and explains why we had to generalize pairing and alternation in a new way. The subsequent section makes the definitions formal and gives proofs of various related properties.

A fair enumeration combinator is one that indexes into its argument enumerations roughly equally, instead of indexing deeply into one and shallowly into another one. For example, imagine we wanted to build an enumeration for lists of length 4. This enumeration is one way to build it:

```
(cons/e (below/e +inf.0)
 (cons/e (below/e +inf.0)
  (cons/e (below/e +inf.0)
   (cons/e (below/e +inf.0)
    (fin/e null)))))
```

The 1,000,000,000th element is '(31622 70 11 0) and, as you can see, it has unfairly indexed far more deeply into the first (below/e +inf.0) than the others. In contrast, if we balance the cons/e expressions like this:

```
(cons/e
 (cons/e (below/e +inf.0) (below/e +inf.0))
 (cons/e (below/e +inf.0) (below/e +inf.0)))
```

(and then use map/e to adjust the elements of the enumeration to be lists), then the element at position 1,000,000,000 is '(177 116 70 132), which is much more balanced. This balance is not specific to just that index in the enumeration, either. Figure 2 shows histograms for each of the components when using the unfair and the fair four-tuple enumerations. The x-coordinates of the plots correspond to the different values that appear in the tuples and the height of each bar is the number of times that particular number appears when enumerating the first 1,500 tuples. As you can see, all four components have roughly the same set of values for the fair tupling operation, but the first tuple element is considerably different from the other three when using the unfair combination.

The definition of fairness requires some subtlety because we cannot just 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 (below/e +inf.0) with itself must eventually produce the pair '(1 . 1000), and that pair must come either before or after the pair '(1000 . 1). So if we insist that, at every point in the enumeration, the combinator's result enumeration has used all of its argument enumerations equally, then pairing would be impossible to do fairly.

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.

We also refine fair combinators, saying that a combinator is f-fair if the nth equilibrium point is at $f(n)$. Parameterizing fairness by this function gives us a way to quantify fair combinators, preferring those that reach equilibrium more often.

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 it has indexed into each of the four below/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 (below/e +inf.0) (below/e +inf.0)) reaches an equilibrium point at every perfect square.

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)
```

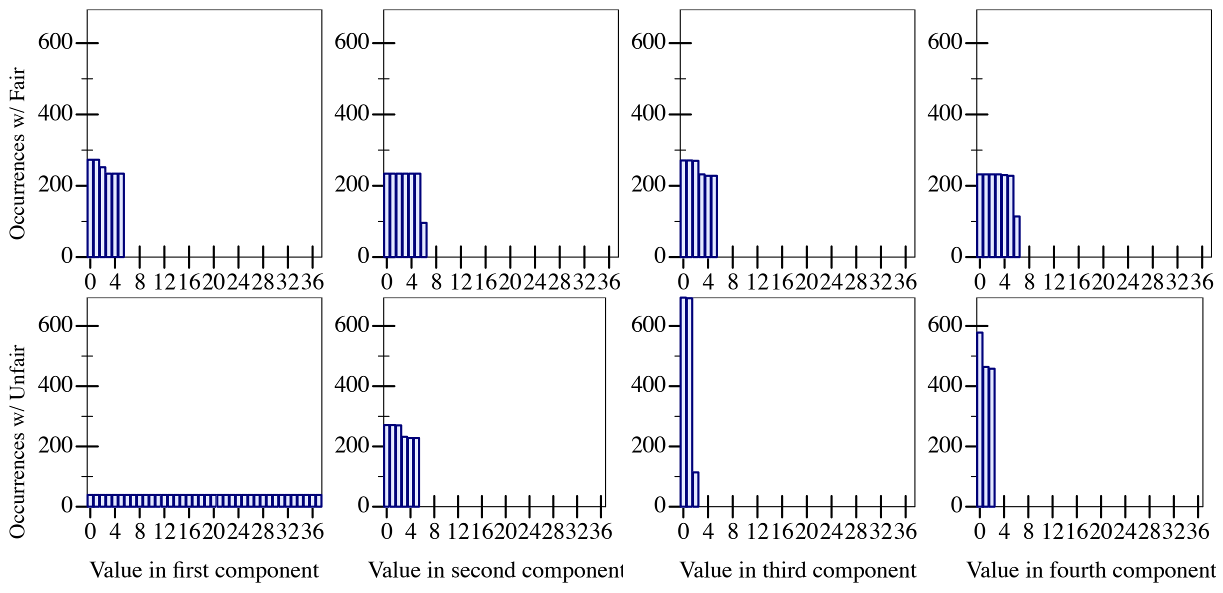


Figure 2: Histograms of the occurrences of each natural number in fair and unfair tuples

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`.

Once we know that nesting pairs is not going to be fair in general, how do we define a fair tupling operation? As we saw in section 2, the fair pairing operation traces out two of the edges of ever-increasing squares in the plane. These ever-increasing squares are at the heart of its fairness. In particular, the bottom-right-most point in each square is the equilibrium point, where it has used the two argument enumerations on exactly the same set of values.

We can generalize this idea to tracing out the faces of ever-increasing cubes for three-tuples, and ever-increasing hypercubes in general. And at each dimension, there is a "layering" property that is preserved. At the corners of the cubes in three dimensions, we will have traced out all three faces of each the current cube and all of the smaller cubes and thus have used all of the argument enumerations the same amount. And similarly for the corners of the hypercubes in n dimensions.

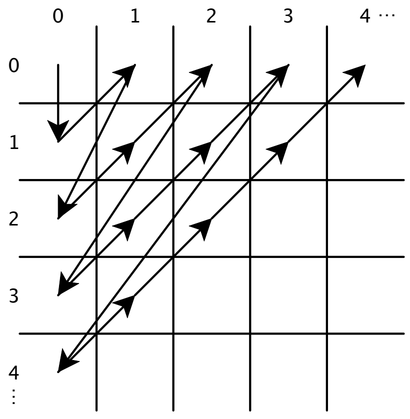
So, we need to generalize the pairing function to compute which face of which hypercube we are on at each point in the enumeration. Returning to two dimensions, say we want the 44th element of the enumeration. To do so, we first compute the integer square root, 6, and then compute the remainder (i.e. 44-6^2), 8. The 6 tells us that we are in the sixth layer, where all pairs have a 6 and there are no elements larger than 6. The remainder 8 tells us that we are at the 8th such pair, which is `'(6 . 2)` (since we (arbitrarily) decided to put pairs with a 6 in the second point first).

To perform the inverse, we take `'(6 . 2)` and note that its max is 6. Then we need to determine which position `'(6 . 2)` is in the enumeration of pairs with max of 6. It is 8, and so we then square the 6 and add 8 to get 44.

This process generalizes to n dimensions. We take the nth root to find which layer we are in. Then we take the remainder and use that to index into an enumeration of n-tuples that has at least one n and whose other values are all less than or equal to n. At a high-level, this has reduced the problem from a n-dimension problem to an $n-1$ dimensional problem, since we can think of the faces of the n dimensional hypercube as $n-1$ dimensional hypercubes. It is not just a recursive process at this point, however, since the $n-1$ dimensional problem has the additional constraint that the enumeration always produce $n-1$ tuples containing at least one n and no values larger than n.

We can, however, produce the enumerations inside the layers recursively. In the general case, we need to enumerate sequences of naturals whose elements have a fixed maximum (i.e. the elements of the sequence are all less than the maximum and yet the maximum definitely appears). This enumeration can be handled with the combinators discussed in section 2. Specifically, an n tuple that contains a maximum of m is either m consed onto the front of an $n-1$ tuple that has values between 0 and m or it is a number less than m combined with an $n-1$ tuple that has a maximum of m.

The combinatorially-inclined reader may wonder why we do not use the classic Cantor pairing function, which can be interpreted as a more triangular grid walk:



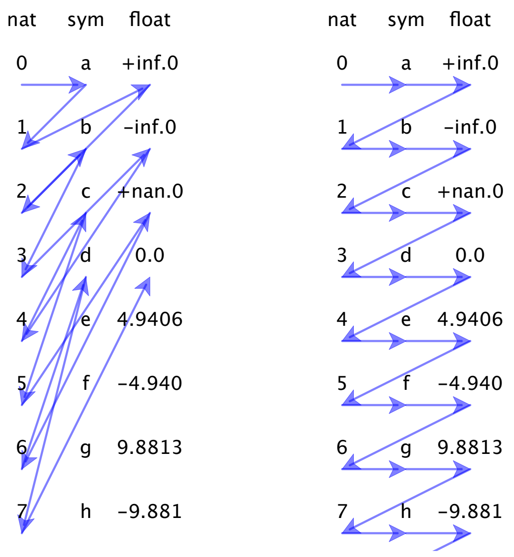


Figure 3: Unfair (left) and fair (right) disjoint union enumerations

The two bijections are quite similar; they are both quadratic functions with similar geometric interpretations. Szudzik (2006)'s traces out the edges of increasingly large squares and Cantor's traces out the bottoms of increasingly large triangles. Importantly, they are both fair (although with different equilibrium points).

For enumerations we are primarily concerned with the other direction of the bijection, since that is the one used to generate terms. In the pairing case, the Cantor function has a fairly straightforward inverse, but its generalization does not. This is the generalization of the cantor pairing function to length k tuples:

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

We can easily define an inefficient (but correct) way to compute the inverse by systematically trying every tuple by using a different untupling function, applying the original *cantor_tuple* function to see if it was the argument given. Tarau (2012) gives the best known algorithm that shrinks the search space considerably, but the algorithm there is still a search procedure, and we found it too slow to use in practice. That said, our library implements Tarau (2012)'s algorithm (via a keyword argument to `cons/e` and `list/e`), in case someone finds it useful.

The `or/e` enumeration's fairness follows a similar, but much simpler pattern. In particular, the binary `or/e` is fair because it alternates between its arguments. As with pairing, extending `or/e` to an n-ary combinator via nested calls of the binary combinator is unfair. Consider a trinary version implemented this way:

```
(define (or-three/e e_1 e_2 e_3)
  (or/e e_1 (or/e e_2 e_3)))
```

and consider passing in an enumeration of naturals, one of symbols, and one of floats. The left side of figure 3 shows the order used by the unfair nesting and the right side shows the fair ordering.

Fixing this enumeration is straightforward; divide the index by `k` and use the remainder to determine which argument enumeration to use and the quotient to determine what to index into the enumeration with.

4. Enumeration Semantics

Figure 4 shows a formal model of a subset of our enumerations. It defines the relation @, which relates an enumeration and an index to the value that the enumeration produces at the index. The T that follows the vertical bar is used in the definition of fairness; ignore it for now. The from-nat and to-nat functions are derived from @ by treating either the value or index argument as given and computing the other one. The contents of Figure 4 are automatically generated from a Redex model and we also built a Coq model of a subset of this semantics. All of the theorems stated in this section are proven with respect to the Coq model. The Redex model, Coq model, and our implementation are all tested against each other.

The upper right section of the figure contains four rules that govern the or/e combinator. The idea for how these work is that they alternate between the two enumerations until one runs out and then they just use the other. The two rules on top cover the case where neither has yet run out and the bottom two cover the situation where it has. The rules with "l" in the name end up producing a value from the left enumeration and the rules with an "r" produce a value from the right.

Just below the grammar is the simplest rule, the one for (below/e n+); it is just the identity. To its right is the map/e rule, showing how its bijection is used. To the right of map/e is the cons/e rule. It uses the unpair function, shown at the bottom of the figure. The unpair function accepts the sizes of the two enumerations, computed by the function near the bottom on the right of the figure (written using double vertical bars), and the index. The function maps indices as discussed in section 2.

The next two rules, reading down the figure, are the dep/e rules. The dep/e combinator is a simplified, functional interface to the `cons/de` combinator. It accepts an enumeration and a function from elements of the first enumeration to new enumerations. It produces pairs where the first position of the pair comes from the first enumeration and the second position's elements come from the enumeration returned by passing the first element of the pair to the given function. The dep/e rule exploits cons/e to get two indices when it deals with infinite enumerations and uses sum_up_to for finite enumerations (defined at the bottom of the figure).

Below dep/e are the rules for except/e, which behave as discussed in section 2, one rule for the situation where the value is below the excepted value and one for where it is above.

Beside the except/e rules is the fix/e rule. The fix/e combinator is like `delay/e`, except it provides an explicit name for the enumeration. The rule uses substitution (our implementation fails to terminate when an "infinite derivation" would be required). The last two rules are an unfair pairing operation using the bijection from the introduction and we return to the rule for trace/e shortly.

The Coq model is simpler than the model presented here and the model presented here is simpler than our implementation. The primary simplification is in the kinds of values that are enumerated. In our implementation, any value that can be captured with a contract in Racket's contract system can be enumerated. In the model presented here, we restrict those values to the ones captured by τ, and in the Coq model restrict that further by eliminating recursive types, subtraction types, and finite types. The typing rules for values are given in the box at the bottom right of figure 4, and the ty function maps enumerators to the type of values that it enumerates. All enumerators enumerate all of the values of their types.

The implementation also has many more combinators than the ones presented here, but they are either derivable from these or require only straightforward extensions. The Coq model has the combinators in figure 4, except for the fix/e combinator and the except/e combinator. In general, the Coq model is designed to be just enough for us to state and prove some results about fairness.

$$
\boxed{\begin{array}{rcl}
e & ::= & (\text{below/e } n{+}) \\
 & \mid & (\text{or/e } e\ e) \\
 & \mid & (\text{cons/e } e\ e) \\
 & \mid & (\text{unfair-cons/e } e\ e) \\
 & \mid & (\text{map/e } f\ f\ e) \\
 & \mid & (\text{dep/e } e\ f) \\
 & \mid & (\text{except/e } e\ v) \\
 & \mid & (\text{fix/e } x\ e) \\
 & \mid & (\text{trace/e } n\ e) \\
 & \mid & x \\
n{+} & ::= & n \mid \infty \\
\tau & ::= & n{+} \mid \tau \wedge \tau \mid \tau \vee \tau \\
 & \mid & \mu x.\,\tau \mid x \mid (\text{-}\ \tau\ v) \\
v & ::= & (\text{inl } v) \mid (\text{inr } v) \\
 & \mid & (\text{cons } v\ v) \mid n \\
n, i, j & ::= & \textit{natural}
\end{array}}
$$

$$
\frac{n \text{ is even} \qquad n < \min(\|e_1\|, \|e_2\|)\cdot 2 \qquad e_1 \mathbin{@} n/2 = v \mid T}{(\text{or/e } e_1\ e_2) \mathbin{@} n = (\text{inl } v) \mid T}\ [\text{or alt l}]
\qquad
\frac{n \text{ is odd} \qquad n < \min(\|e_1\|, \|e_2\|)\cdot 2 \qquad e_2 \mathbin{@} (n-1)/2 = v \mid T}{(\text{or/e } e_1\ e_2) \mathbin{@} n = (\text{inr } v) \mid T}\ [\text{or alt r}]
$$

$$
\frac{n \geq \min(\|e_1\|, \|e_2\|)\cdot 2 \qquad \|e_2\| < \|e_1\| \qquad e_1 \mathbin{@} n - \|e_2\| = v \mid T}{(\text{or/e } e_1\ e_2) \mathbin{@} n = (\text{inl } v) \mid T}\ [\text{or big l}]
\qquad
\frac{n \geq \min(\|e_1\|, \|e_2\|)\cdot 2 \qquad \|e_1\| < \|e_2\| \qquad e_2 \mathbin{@} n - \|e_1\| = v \mid T}{(\text{or/e } e_1\ e_2) \mathbin{@} n = (\text{inr } v) \mid T}\ [\text{or big r}]
$$

$$
\frac{n < n{+}}{(\text{below/e } n{+}) \mathbin{@} n = n \mid \varnothing}\ [\text{below/e}]
\qquad
\frac{e \mathbin{@} n = v_1 \mid T \qquad v_2 = (f_1\ v_1) \qquad v_1 = (f_2\ v_2)}{(\text{map/e } f_1\ f_2\ e) \mathbin{@} n = v_2 \mid T}\ [\text{map}]
$$

$$
\frac{\langle n_1, n_2\rangle = \text{unpair}(\|e_1\|, \|e_2\|, n) \qquad e_1 \mathbin{@} n_1 = v_1 \mid T_1 \qquad e_2 \mathbin{@} n_2 = v_2 \mid T_2}{(\text{cons/e } e_1\ e_2) \mathbin{@} n = (\text{cons } v_1\ v_2) \mid \lambda \text{x}.\ T_1(\text{x}) \cup T_2(\text{x})}\ [\text{cons}]
$$

$$
\frac{(\text{cons/e } e\ (\text{below/e } \infty)) \mathbin{@} n_1 = (\text{cons } v_1\ n_2) \mid T_1 \qquad (f\ v_1) \mathbin{@} n_2 = v_2 \mid T_2 \qquad \forall \text{ x} \in e,\ \|f(\text{x})\| = \infty}{(\text{dep/e } e\ f) \mathbin{@} n_1 = (\text{cons } v_1\ v_2) \mid \lambda \text{x}.\ T_1(\text{x}) \cup T_2(\text{x})}\ [\text{dep inf}]
$$

$$
\frac{\forall \text{ x} \in e,\ \|f(\text{x})\| < \infty \qquad \text{sum\_up\_to}(e, f, n_2) \leq n_1 < \text{sum\_up\_to}(e, f, n_2 + 1) \qquad e \mathbin{@} n_2 = v_1 \mid T_1 \qquad (f\ v_1) \mathbin{@} n_1 - \text{sum\_up\_to}(e, f, n_2) = v_2 \mid T_2}{(\text{dep/e } e\ f) \mathbin{@} n_1 = (\text{cons } v_1\ v_2) \mid \lambda \text{x}.\ T_1(\text{x}) \cup T_2(\text{x})}\ [\text{dep fin}]
$$

$$
\frac{e \mathbin{@} n_1 = v_1 \mid T_2 \qquad e \mathbin{@} n_2 = v_2 \mid T \qquad n_2 < n_1}{(\text{except/e } e\ v_1) \mathbin{@} n_2 = v_2 \mid T}\ [\text{ex<}]
\qquad
\frac{e \mathbin{@} n_1 = v_1 \mid T_2 \qquad e \mathbin{@} n_2 + 1 = v_2 \mid T \qquad n_2 \geq n_1}{(\text{except/e } e\ v_1) \mathbin{@} n_2 = v_2 \mid T}\ [\text{ex}\geq]
\qquad
\frac{e\{x := (\text{fix/e } x\ e)\} \mathbin{@} n = v \mid T}{(\text{fix/e } x\ e) \mathbin{@} n = v \mid T}\ [\text{fix}]
$$

$$
\frac{n = 2^i(2j+1) \qquad e_1 \mathbin{@} j = v_1 \mid T_1 \qquad e_2 \mathbin{@} i = v_2 \mid T_2}{(\text{unfair-cons/e } e_1\ e_2) \mathbin{@} n = (\text{cons } v_1\ v_2) \mid \lambda \text{x}.\ T_1(\text{x}) \cup T_2(\text{x})}\ [\text{unfair}]
\qquad
\frac{e \mathbin{@} n_2 = v \mid T}{(\text{trace/e } n_1\ e) \mathbin{@} n_2 = v \mid \lambda \text{x}.\ n_1{=}\text{x}\ ?\ \{n_2\} : \varnothing}\ [\text{trace}]
$$

$$
\begin{array}{lcl}
\text{ty}[\![e]\!] = \text{tye}[\![\varnothing, e]\!] & & \\
\text{tye}[\![\Gamma, (\text{below/e } n{+})]\!] & = & n{+} \\
\text{tye}[\![\Gamma, (\text{or/e } e_1\ e_2)]\!] & = & \text{tye}[\![\Gamma, e_1]\!] \vee \text{tye}[\![\Gamma, e_2]\!] \\
\text{tye}[\![\Gamma, (\text{cons/e } e_1\ e_2)]\!] & = & \text{tye}[\![\Gamma, e_1]\!] \wedge \text{tye}[\![\Gamma, e_2]\!] \\
\text{tye}[\![\Gamma, (\text{unfair-cons/e } e_1\ e_2)]\!] & = & \text{tye}[\![\Gamma, e_1]\!] \wedge \text{tye}[\![\Gamma, e_2]\!] \\
\text{tye}[\![\Gamma, (\text{map/e } f_1\ f_2\ e)]\!] & = & \text{rng}[\![f_1]\!] \quad \text{if tye}[\![\Gamma, e]\!] = \text{rng}[\![f_2]\!] \\
\text{tye}[\![\Gamma, (\text{dep/e } e\ f)]\!] & = & \text{tye}[\![\Gamma, e]\!] \wedge \text{rng}[\![f]\!] \\
\text{tye}[\![\Gamma, (\text{except/e } e\ v)]\!] & = & (\text{-}\ \text{tye}[\![\Gamma, e]\!]\ v) \\
\text{tye}[\![\Gamma, (\text{fix/e } x\ e)]\!] & = & \mu x.\,\text{tye}[\![\Gamma \cup \{x\}, e]\!] \\
\text{tye}[\![\Gamma, (\text{trace/e } n\ e)]\!] & = & \text{tye}[\![\Gamma, e]\!] \\
\text{tye}[\![\Gamma, x]\!] & = & x \quad \text{if } x \in \Gamma
\end{array}
$$

$$
\begin{array}{lcl}
\|(\text{below/e } n{+})\| & = & n{+} \\
\|(\text{or/e } e_1\ e_2)\| & = & \|e_1\| + \|e_2\| \\
\|(\text{cons/e } e_1\ e_2)\| & = & \|e_1\| \cdot \|e_2\| \\
\|(\text{unfair-cons/e } e_1\ e_2)\| & = & \|e_1\| \cdot \|e_2\| \\
\|(\text{map/e } f\ f\ e)\| & = & \|e\| \\
\|(\text{dep/e } e\ f)\| & = & \infty \quad \text{if } \forall \text{ x} \in e,\ \|f(\text{x})\| = \infty \\
\|(\text{dep/e } e\ f)\| & = & \|e\| \cdot (\Sigma \text{ x} \in e.\ \|f(\text{x})\|) \quad \text{if } \|e\| < \infty \\
\|(\text{except/e } e\ v)\| & = & \|e\| - 1 \\
\|(\text{fix/e } x\ e)\| & = & \|e\| \quad \text{if } e\{x := (\text{fix/e } x\ e)\} = e \\
\|(\text{fix/e } x\ e)\| & = & \infty \\
\|(\text{trace/e } n\ e)\| & = & \|e\|
\end{array}
$$

$$
\begin{array}{lcll}
\text{unpair}(\infty, \infty, n) & = & \langle n - \lfloor\sqrt{n}\rfloor^2, \lfloor\sqrt{n}\rfloor\rangle & \text{if } n - \lfloor\sqrt{n}\rfloor^2 < \lfloor\sqrt{n}\rfloor \\
\text{unpair}(\infty, \infty, n) & = & \langle \lfloor\sqrt{n}\rfloor, n - \lfloor\sqrt{n}\rfloor^2 - \lfloor\sqrt{n}\rfloor\rangle & \\
\text{unpair}(i, \infty, n) & = & \langle n\%i, \lfloor n/i\rfloor\rangle & \\
\text{unpair}(\infty, j, n) & = & \langle \lfloor n/j\rfloor, n\%j\rangle & \\
\text{unpair}(i, j, n) & = & \langle n\%i, \lfloor n/i\rfloor\rangle & \text{if } i < j \\
\text{unpair}(i, j, n) & = & \langle \lfloor n/j\rfloor, n\%j\rangle & \text{if } i \geq j
\end{array}
$$

$$
\boxed{
\frac{n < n{+}}{n : n{+}}
\qquad
\frac{v_1 : \tau_1 \qquad v_2 : \tau_2}{(\text{cons } v_1\ v_2) : \tau_1 \wedge \tau_2}
\qquad
\frac{v : \tau\{x := \mu x.\,\tau\}}{v : \mu x.\,\tau}
\qquad
\frac{v : \tau_2}{(\text{inr } v) : \tau_1 \vee \tau_2}
\qquad
\frac{v : \tau_1}{(\text{inl } v) : \tau_1 \vee \tau_2}
\qquad
\frac{v_1 : \tau_1 \qquad v_1 \neq v_2}{v_1 : (\text{-}\ \tau_1\ v_2)}
}
$$

$$
\text{sum\_up\_to}(e, f, n) = \Sigma\{\ \|f(v)\| \ \mid\ (e \mathbin{@} i = v \mid T) \text{ and } i < n\}
$$

Figure 4: Semantics of Enumeration Combinators

Before we define fairness, however, we first need to prove that the model actually defines two functions.

Theorem 1. *For all e (in the Coq model), n, there exists a unique v and T such that* $e \mathbin{@} n = v \mid T$ *and* $v : \mathsf{ty}[\![e]\!]$*, and we can compute v and T.*

Proof. The basic idea is that you can read the value off of the rules recursively, computing new values of n. In some cases there are multiple rules that apply for a given e, but the conditions on n in the premises ensure there is exactly one rule to use. Computing the T argument is straightforward. The full proof is given as `Enumerates_from_dec_uniq` in the supplementary material. □

Theorem 2. *For all e (in the Coq model), v, if* $v : \mathsf{ty}[\![e]\!]$*, then there exists a unique T and n such that* $e \mathbin{@} n = v \mid T$.

Proof. As with the previous theorem, we recursively process the rules to compute n. This is complicated by the fact that we need inverse functions for the formulas in the premises of the rules to go from the given n to the one to use in the recursive call, but these inverses exist. The full proof is given as `Enumerates_to_dec_uniq` in the supplementary material, and it includes proofs of the bijective nature of the formulas. □

Although we don't prove it formally, the situation where the $v : \mathsf{ty}[\![e]\!]$ condition does not hold in the second theorem corresponds to the situation where the value that we are attempting to convert to a number does not match the contract in the enumeration in our implementation.

We use these two results to connect the Coq code to our implementation. Specifically, we use Coq's `Eval compute` facility to print out values of the enumeration at specific points and then compare that to what our implementation produces. This is the same mechanism we use to test our Redex model against the Coq model. The testing code is in the supplementary material.

To define fairness, we need to be able to trace how an enumeration combinator uses its arguments, and this is the purpose of the trace/e combinator and the T component in the semantics. These two pieces work together to trace where a particular enumeration has been sampled. Specifically, wrapping an enumeration with trace/e means that it should be tracked and the n argument is a label used to identify a portion of the trace. The T component is the current trace; it is a function that maps the n arguments in the trace/e expressions to sets of natural numbers indicating which naturals the enumeration has been used with.

Furthermore, we also need to be able to collect all of the traces for all naturals up to some given n. We call this the "complete trace up to n". So, for some enumeration expression e, the complete trace up to n is the pointwise union of all of the T components for $e \mathbin{@} i = v \mid T$, for all values v and i strictly less than n.

For example, the complete trace of

```
(cons/e (trace/e 0 (below/e ∞))
        (trace/e 1 (below/e ∞)))
```

up to 256 maps both 0 and 1 to $\{x{:}\mathrm{nat} \mid 0 \leqslant x \leqslant 15\}$ whereas the complete trace of

```
(unfair-cons/e (trace/e 0 (below/e ∞))
               (trace/e 1 (below/e ∞)))
```

up to 256 maps 0 to $\{x{:}\mathrm{nat} \mid 0 \leqslant x \leqslant 127\}$ and 1 to $\{x{:}\mathrm{nat} \mid 0 \leqslant x \leqslant 8\}$. (See Theorem 7 for the definition of unfair-cons/e).

We say that an enumeration combinator $c^k : \mathit{enum} \ldots \to \mathit{enum}$ of arity k is fair if, for every natural number m, there exists a natural number $M > m$ such that in the complete trace up to M of c^k applied to (trace/e 1 enum_1) $\cdots$ (trace/e k enum_k), for any enumerations enum_1 to enum_k, is a function that maps each number between 1 and k to exactly the same set of numbers. Any other combinator is unfair. We say that a combinator is f-fair if the n-th equilibrium point is at $f(n)$. The Coq model contains this definition only for $k \in \{2, 3, 4\}$, called `Fair2`, `Fair3`, and `Fair4`.

Theorem 3. or/e *is* $\lambda n.\ 2n+2$*-fair.*

Proof. This can be proved by induction on n. The full proof is `SumFair` in the Coq model. □

Theorem 4. `or-three/e` *from section 3 is unfair.*

Proof. We show that after a certain point, there are no equilibria. For $n \geqslant 8$, there exist natural numbers m, p such that $2m \leqslant n < 4p$ while $p < m$. Then a complete trace from 0 to n maps 0 to a set that contains $\{0, \ldots, m\}$, but on the other hand maps 1 (and 2) to subset of $\{0, \ldots, p\}$. Since $p < m$, these sets are different. Thus `or-three/e` is unfair. The full proof is `NaiveSum3Unfair` in the Coq model. □

Theorem 5. cons/e *is* $\lambda n.\ (n+1)^2$*-fair.*

Proof. First, we show that tracing from n^2 to $(n+1)^2$ produces a trace that maps 0 and 1 to the set $\{0, \ldots, n\}$. Then we can prove that tracing from 0 to n^2 maps 0 and 1 to $\{0, \ldots, n-1\}$ and the result then holds by induction on n. The full proof is `PairFair` in the Coq model. □

Theorem 6. triple/e *from section 3 is unfair.*

Proof. For any natural $n \geqslant 16$, there exist natural numbers m, p such that $m^2 \leqslant n < p^4$ and $p < m$. Then a complete trace from 0 to n will map 0 to a set that includes everything in $\{0, \ldots, m\}$, but will map 1 (and 2) to sets that are subsets of $\{0, \ldots, p\}$. Since $p < m$, these sets are different, so triple/e is unfair. The full proof is `NaiveTripleUnfair` in the Coq model. □

Theorem 7. *The pairing operator* unfair-cons/e*, defined using the unfair bijection from the introduction, is unfair.*

Proof. A complete trace from 0 to n contains all of the values from 0 to $\lfloor n/2+1 \rfloor$ in the first component and all of the values from 0 to $\lfloor \log_2(n) \rfloor + 1$ in the second component. For any n greater than 8, the first component will always have more values than the second component and thus there will be no equilibrium points after 8. The full proof is `UnfairPairUnfair` in the Coq model. □

5. Empirical Evaluation

As the primary motivation for studying enumerations is test case generation, we performed an empirical evaluation of fair and unfair enumerations to try to understand the impact of using unfair combinators on test case generation. We also used a mature ad hoc random generator as a baseline for the comparison, to give our results some context. This section describes the setup of our evaluation and its results.

5.1 Setup

We conducted the evaluation in the context of Redex (Felleisen et al. 2009; Matthews et al. 2004), a domain-specific programming language for operational semantics, type systems, and their associated machinery. Redex gives semantics engineers the ability to formulate and check claims about their semantics and it includes a random test case generator that can be used to automatically falsify such claims.

Our evaluation used the Redex benchmark, which consists of a number of models, including the Racket virtual machine model (Klein et al. 2013), a polymorphic λ-calculus used for random testing (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

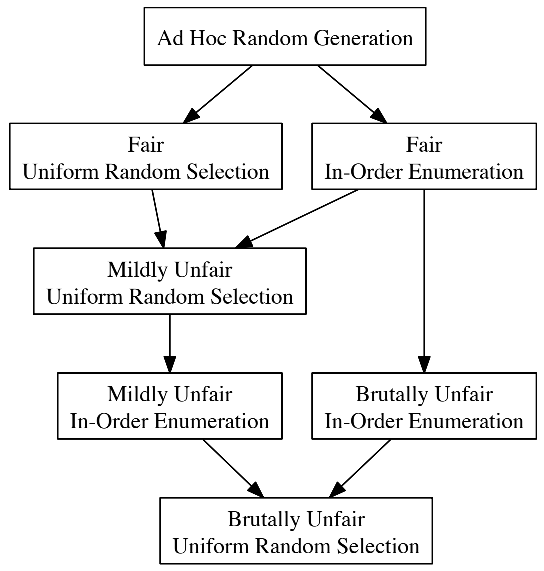

Ad Hoc Random Generation
Fair Uniform Random Selection
Fair In-Order Enumeration
Mildly Unfair Uniform Random Selection
Mildly Unfair In-Order Enumeration
Brutally Unfair In-Order Enumeration
Brutally Unfair Uniform Random Selection

Figure 5: Partial Order Between Generators Indicating Which Find More Bugs

ourselves based on our experience with random generation and to cover typical Redex models.[5] Each model comes with a number of buggy variations. Each model and bug pair is equipped with a property that should hold for every term, but does not, due to the bug. There are 8 models and 50 bugs in total.

The Redex benchmark comes equipped with a mechanism to add new generators to each model and bug pair, as well as a built-in ad hoc, random generator. We used the enumeration library described in section 2 to build two generators based on enumeration, one that just chooses terms in the order induced by the natural numbers, and one that selects a random natural and uses that to index into the enumeration.

The ad hoc random generation is Redex's existing random generator (Klein and Findler 2009). It has been tuned based on experience programming in Redex, but not recently. From the git logs, 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 studying enumeration.

The ad hoc random 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 random enumeration method, although the distribution is different.

To pick a random natural number to index into the enumeration, 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.

We chose 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 likely to be larger than from the mean of the old. We believe this is a good property to have when indexing into our enumerations so we avoid biasing our indices towards a small size.

[5] It is online: `http://docs.racket-lang.org/redex/benchmark.html`

The random-selection results are 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 enumeration method. Even with this artificial help, this method was still worse, overall, than the other two.

We used three variations on the enumeration combinators. The first is the fair combinators described in section 3. The second uses fair binary pairing and binary alternation combinators, but that are unfairly generalized via nesting (to create n-tuples or n-way alternations), which we call "mildly unfair". The third variation uses the unfair binary pairing combinator based on the bijection described in the introduction, also unfairly generalized to n-ary pairing. It uses an analogous unfair alternation combinator that goes exponentially deep into one argument as compared to the other, also unfairly generalized to n-ary alternation. The final one we call "brutally unfair".

For each of the 350 bug and generator combinations, 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. We ran the script in two rounds. The first round ran all 350 bug and generator combinations until either 24 hours elapsed or the standard error in the average became less than 10% of the average. Then we took all of the bugs where the 95% confidence interval was greater than 50% of the average and where at least one counterexample was found and ran each of those for an additional 8 days. All of the final averages have an 95% confidence interval that is less than 50% of the average.

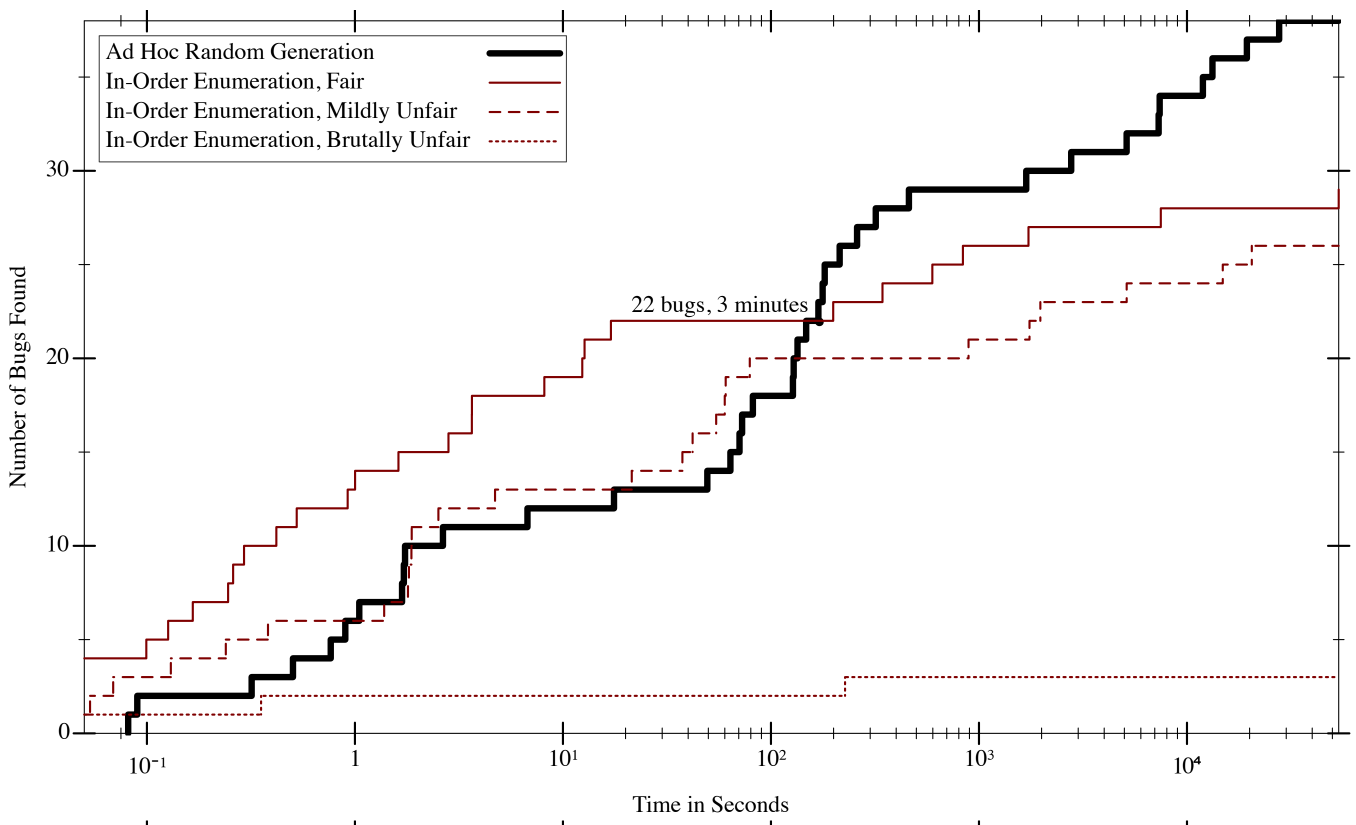
We used two identical 64 core AMD machines with Opteron 6274s running at 2,200 MHz with a 2 MB L2 cache to run the benchmarks. Each machine has 64 gigabytes of memory. Our script typically runs each model/bug combination sequentially, although we ran multiple different combinations in parallel and, for the bugs that ran for more than 24 hours, we ran tests in parallel. We used version 6.2.0.4 (from git on June 7, 2015) of Racket, of which Redex is a part.

5.2 Results

The graph in figure 5 gives a high-level view of which generators are more effective at finding bugs. There is an edge between two generators if the one above finds all the bugs that the one below finds and the one below was unable to find at least one bug that the one above found. By this metric, the ad hoc random generator is a clear winner, the fair enumerators are second and the unfair ones are third, with the mildly unfair enumerators usually doing better than the brutally unfair ones.

That overview lacks nuance, however, as it does not take into account how long it took for each generator to find the bugs that it found. The plots in figure 6 take time into account, showing how well each generator is doing as a function of time. Along x-axis is time in seconds in a log scale, varying from milliseconds to the left to a few hours on the right. Along the y-axis is the total number of counterexamples found for each point in time. The lines on each plot show how the number of counterexamples found changes as time passes.

The thicker, black line is the same on both plots. It shows the number of counterexamples found by the ad hoc random generator. The upper plot shows how drawing from the enumeration in order fares, compared to the random generator. The solid red (not thick) line is with fair combinators, the dashed line is with the mildly unfair combinators and the dotted line is with the brutally unfair combinators. Similarly, the bottom plot uses the same set of combinators, but randomly picks natural numbers (as described above)



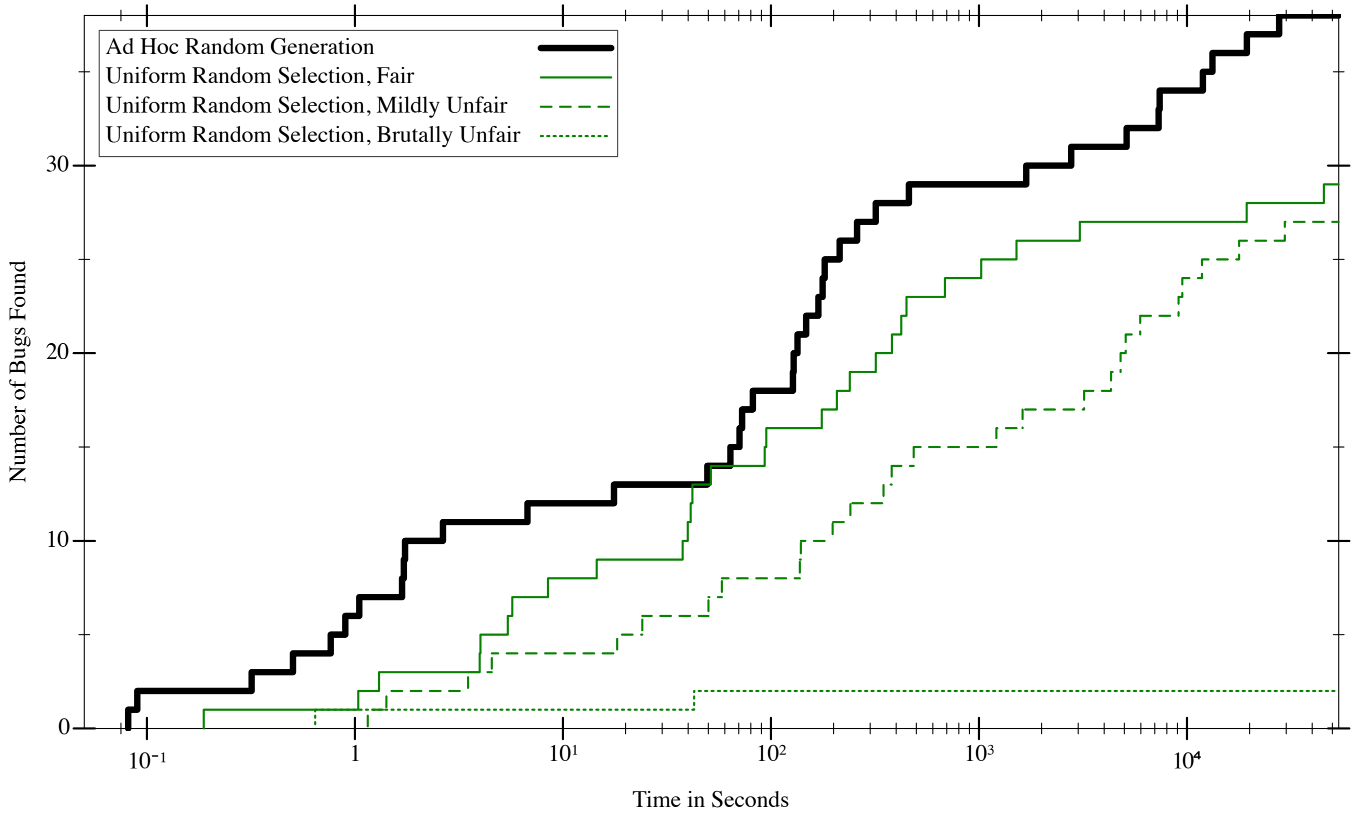


Figure 6: Overview of random testing performance of ad hoc generation, in-order enumeration, and random indexing into an enumeration, on a benchmark of Redex models.

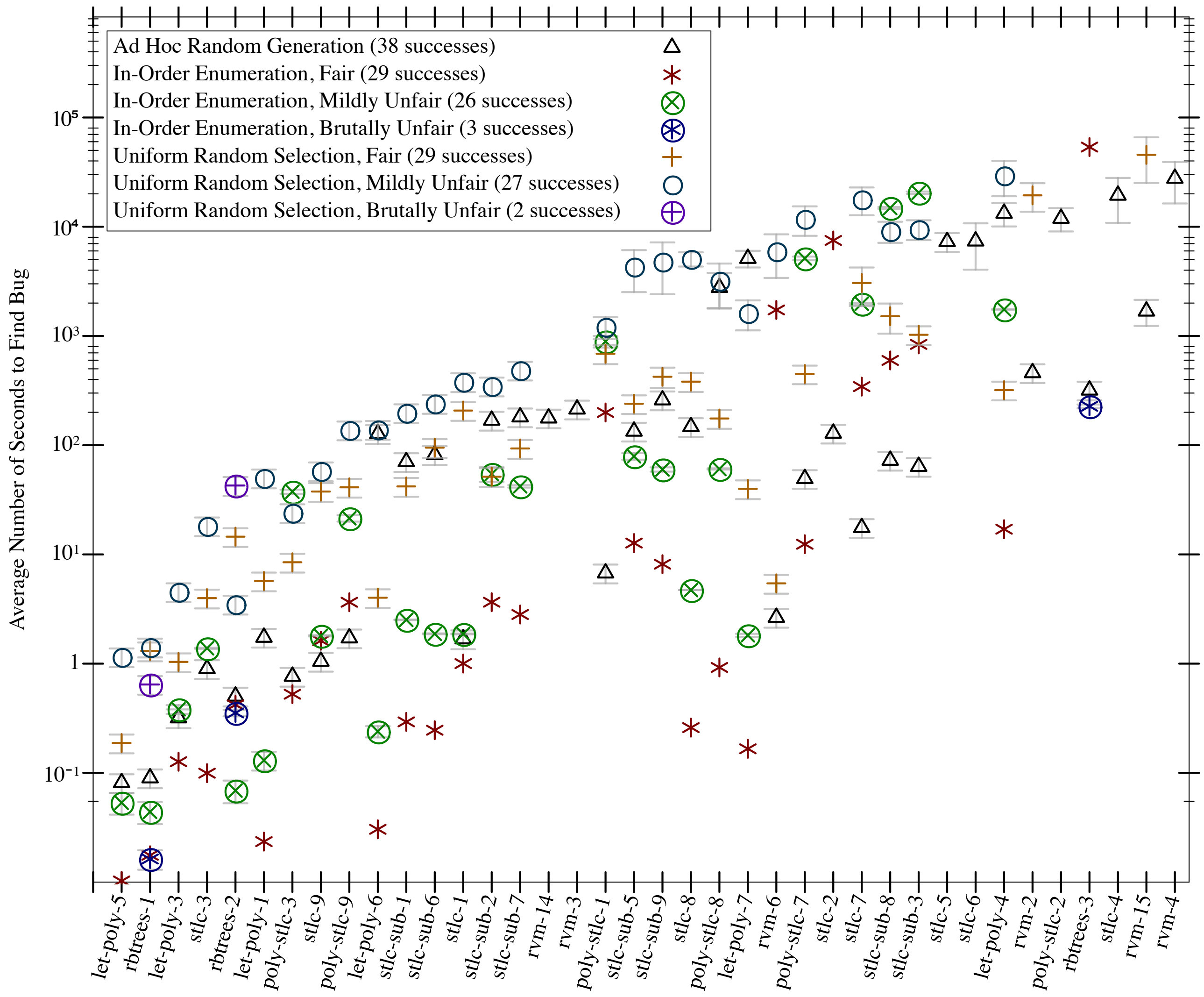


Figure 7: The mean time each generator takes to find the bugs, for each bug that some generator found; bars indicate 95% confidence intervals

and uses those to generate candidates. No strategy was able to find more than 38 of the 50 bugs in the benchmark.

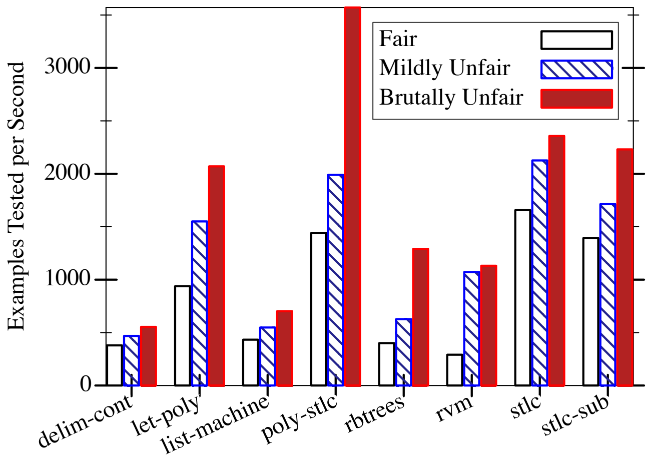
These plots also show that the mildly unfair combinators are worse than the fair ones and the brutally unfair combinators are much worse than either. But they also reveal that the ad hoc generator is only better than the best enumeration strategy after 22 minutes. Before that time, the fair in-order enumeration strategy is the best approach.

A more detailed version of our results is shown in figure 7. The x-axis has one entry for each different bug, for which a counterexample was found. Each point indicates the average number of seconds required to find that bug, with the bars indicating a 95% confidence interval. The different shapes and colors of the points indicate which method was used. The bugs are sorted along the x-axis by the average amount of time required to find the bug across all strategies.

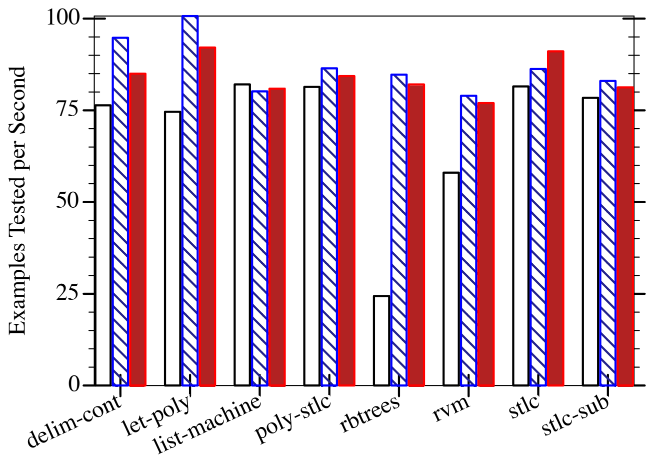
Figure 7's chart confirms the conclusion from figure 6's. Specifically, the circled points are the ones with unfair combinators and they are never significantly below their uncircled counterparts and often significantly above.

Figure 7 also shows that, for the most part, bugs that were easy (could be found in less than a few seconds) for the generator that selected at random from the enumerations were easy for all three generators. The ad hoc random generator and the fair in-order enumeration generator each had a number of bugs where they were at least one decimal order of magnitude faster than all of the other generators (and multiple generators found the bug). The ad hoc random generator was significantly better on: poly-stlc-1, rvm-15, rvm-2, stlc-2, stlc-7, and stlc-sub-3, and the fair in-order enumerator was significantly better on: let-poly-4, let-poly-7, poly-stlc-8, stlc-8, stlc-sub-2, and stlc-sub-7. The unfair enumerators were never significantly better on any bug.

We believe that the fair enumerators are better than the unfair ones because their more balanced exploration of the space leads to a wider variety of interesting examples being explored. Figure 8 provides some evidence for this theory. It shows the number of examples tested per second for each model (the redex bug benchmark



In-Order Enumeration



Uniform Random Selection

Figure 8: Examples tested per second for each benchmark model and enumeration-based generator

does not cause our generators or the ad hoc random generator to generate different per-bug examples, only different per-model examples) under the different generator strategies. The left-hand plot shows the in-order generators and the right-hand plot shows the generators that selected random natural numbers and used those to generate examples. In every case, the fair enumeration strategy generates fewer examples per second (except for the list-machine benchmark in the random generator, where it is only slightly faster). And yet the fair generators find more bugs. This suggests that the unfair generators are repeatedly generating simple examples that can be tested quickly, but that provide little new information about the model. We believe this is because the unfair generators spend a lot of time exploring programs that differ only in the names of the variables or other typically uninteresting variations.

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 (2013)'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 enumerations are fair (not just the binary ones). Second, our enumerations 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 enumeration 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 like we do. Unlike our approach, however, Feat's enumerations are not just bijective functions directly on naturals, but instead a sequence of finite bijections that, when strung together, combine into a bijection on the naturals. In other words, the Feat combinators get more information from their inputs than ours do, namely a partitioning of the naturals into consecutive finite subsets. This additional information means that our precise, technical definition of fairness does not apply directly to Feat's combinators. The intuition of fairness, however, does apply and Feat's pairing and alternation combinators are fair in the sense that they reach equilibrium infinitely often. Unfortunately, the distance between consecutive equilibrium points doubles at each step, meaning that equilibrium points are exponentially far apart, while ours are linearly far apart for alternation combinators or polynomially far apart for pairing combinators.

Kuraj and Kuncak (2014)'s SciFe library is closer to our library than Feat, but it has only one half of the bijection so it does not support `except/e`. It has fair binary pairing and alternation combinators, but no n-ary fair combinators. Its combinators use the same bijections as the mildly unfair combinators discussed in section 5. Its pairing operation is based on the Cantor pairing function, meaning that computing the n-ary fair version of it is expensive, as discussed in section 3.

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 analogous 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 bit-string 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.

6.2 Testing Studies

Our empirical evaluation is focused on the question of fairness, but it also sheds some light on the relative quality of enumeration and random-based generation strategies.

Even though enumeration-based testing methods have been explored in the literature, there are few studies that specifically contain empirical studies comparing 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, a 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. 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.

In surveying related work, we noticed that 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 seems to be a general impression that random testing is a poor choice for bug-finding. For example, Godefroid et al. (2005) and Heidegger and Thiemann (2010) both dismiss random testing using the relatively simple example: $\forall x, x * 2 \neq x + 10$ as support, suggesting that it is hard for random testing to find a counterexample to this property. 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 focus on a single example 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.

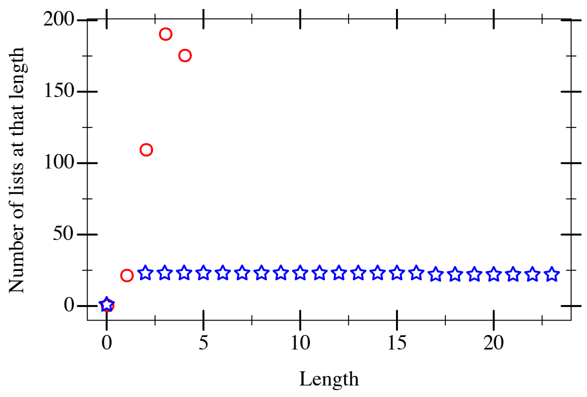
7. Future Work

We are unsatisfied with one aspect of our definition of fairness, namely that it does not capture what intuitively seems to be unfair behavior in enumerations of recursively specified data structures.

Our definition of fairness requires (at least) binary combinators. That is, our definition of enumerations of, for example, lists is not a candidate for our fairness definition because it accepts only one argument. There is, however, more than one way to define an enumeration of lists that seem to be either fair or not. For example, the `lon/e` enumeration from section 2 tends to bias towards shorter lists that have bigger numbers at the front of the list in an unfair way. Instead, consider an enumeration that first selects a length of the list and then uses a dependent enumeration to build a fair n-tuple of the corresponding length, like this code does:

```
(or/e (single/e '())
      (dep/e (below/e +inf.0)
             (λ (len)
               (define enums
                 (for/list ([i (in-range (+ len 1))])
                   (below/e +inf.0)))
               (apply list/e enums))))
```

This enumeration balances the length of the list with the elements of the list in a way that seems more fair. Concretely, here is a histogram of the lengths of the lists from the first 500 elements of the two enumerations. The red circles are the lengths of the `lon/e` enumeration and the blue stars are the lengths of the enumeration above that uses the dependent pair.



The idea of using dependent pairing to first select a "shape" for the data structure and then to stitch it together with enumerations for content of the data-structure is general and can be used to generate trees of arbitrary shapes. And this approach seems like it should be considered fair, but we do not yet have a formal characterization of fairness that captures this difference.

We hope that someday someone is able to capture this notion but there is one other wrinkle worth mentioning: the seemingly fair enumeration is much slower. Enough that the built-in list combinator in our enumeration library does not provide that enumeration strategy by default (although it is an option that is easy to use).

8. Conclusion

This paper presents a new concept for enumeration libraries that we call *fairness*, backing it up with a theoretical development of fair combinators, an implementation, and an empirical study showing that fair enumeration can support an effective testing tool and that unfair enumerations cannot.

Indeed, the results of our empirical study have convinced us to modify Redex's default 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 any configuration.

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.

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.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. MIT Press, 2009.

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.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proc. Symposium on Principles of Programming Languages*, 2012.

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.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *Proc. International Conference on Rewriting Techniques and Applications*, 2004.

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

Paul Tarau. Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings). *Theory and Practice of Logic Programming* 13(4–5), 2013.

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 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))))
```